Book609.pdf

Reinforcement Learning: An Introduction

Second edition, in progress

****Complete Draft****

November 5, 2017

Richard S. Sutton and Andrew G. Barto

The text is now complete, except possibly for one more case study to be added to Chapter 16. The references still need to be thoroughly checked, and an index still needs to be added. Please send any errors to rich@richsutton.com and barto@cs.umass.edu. We are also very interested in correcting any important omissions in the "Bibliographical and Historical Remarks" at the end of each chapter. If you think of something that really should have been cited, please let us know and we can try to get it corrected before the final version is printed.

A Bradford Book

The MIT Press
Cambridge, Massachusetts
London, England

Chapter 11

*Off-policy Methods with Approximation

This book has treated on-policy and off-policy learning methods since Chapter 5 primarily as two alternative ways of handling the conflict between exploitation and exploration inherent in learning forms of generalized policy iteration. The two chapters preceding this have treated the *on*-policy case with function approximation, and in this chapter we treat the *off*-policy case with function approximation. The extension to function approximation turns out to be significantly different and harder for off-policy learning than it is for on-policy learning. The tabular off-policy methods developed in Chapters 6 and 7 readily extend to semi-gradient algorithms, but these algorithms do not converge as robustly as they do under on-policy training. In this chapter we explore the convergence problems, take a closer look at the theory of linear function approximation, introduce a notion of learnability, and then discuss new algorithms with stronger convergence guarantees for the off-policy case. In the end we will have improved methods, but the theoretical results will not be as strong, nor the empirical results as satisfying, as they are for on-policy learning. Along the way, we will gain a deeper understanding of approximation in reinforcement learning for on-policy learning as well as off-policy learning.

Recall that in off-policy learning we seek to learn a value function for a *target policy* π, given data due to a different *behavior policy* b. In the prediction case, both policies are static and given, and we seek to learn either state values $\hat{v} \approx v_\pi$ or action values $\hat{q} \approx q_\pi$. In the control case, action values are learned, and both policies typically change during learning—π being the greedy policy with respect to $\hat{q}$, and b being something more exploratory such as the ε-greedy policy with respect to $\hat{q}$.

The challenge of off-policy learning can be divided into two parts, one that arises in the tabular case and one that arises only with function approximation. The first part of the challenge has to do with the target of the update (not to be confused with the target policy), and the second part has to do with the distribution of the updates. The techniques related to importance sampling developed in Chapters 5 and 7 deal with the first part; these may increase variance but are needed in all successful algorithms, tabular and approximate. The extension of these techniques to function approximation are quickly dealt with in the first section of this chapter.

Something more is needed for the second part of the challenge of off-policy learning with function approximation because the distribution of updates in the off-policy case is not according to the on-policy distribution. The on-policy distribution is important to the stability of semi-gradient methods. Two general approaches have been explored to deal with this. One is to use importance sampling methods again, this time to warp the update distribution back to the on-policy distribution, so that semi-gradient methods are guaranteed to converge (in the linear case). The other is to develop true gradient methods that do not rely on any special distribution for stability. We present methods based

on both approaches. This is a cutting-edge research area, and it is not clear which of these approaches is most effective in practice.

11.1 Semi-gradient Methods

We begin by describing how the methods developed in earlier chapters for the off-policy case extend readily to function approximation as semi-gradient methods. These methods address the first part of the challenge of off-policy learning (changing the update targets) but not the second part (changing the update distribution). Accordingly, these methods may diverge in some cases, and in that sense are not sound, but still they are often successfully used. Remember that these methods *are* guaranteed stable and asymptotically unbiased for the tabular case, which corresponds to a special case of function approximation. So it may still be possible to combine them with feature selection methods in such a way that the combined system could be assured stable. In any event, these methods are simple and thus a good place to start.

In Chapter 7 we described a variety of tabular off-policy algorithms. To convert them to semi-gradient form, we simply replace the update to an array (V or Q) to an update to a weight vector ($\mathbf{w}$), using the approximate value function ($\hat{v}$ or $\hat{q}$) and its gradient. Many of these algorithms use the per-step importance sampling ratio:

$$\rho_t \doteq \rho_{t:t} = \frac{\pi(A_t|S_t)}{b(A_t|S_t)}. \tag{11.1}$$

For example, the one-step, state-value algorithm is semi-gradient off-policy TD(0), which is just like the corresponding on-policy algorithm (page 166) except for the addition of ρ_t:

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha\rho_t\delta_t\nabla\hat{v}(S_t,\mathbf{w}_t), \tag{11.2}$$

where δ_t is defined appropriately depending on whether the problem is episodic and discounted, or continuing and undiscounted using average reward:

$$\delta_t \doteq R_{t+1} + \gamma\hat{v}(S_{t+1},\mathbf{w}_t) - \hat{v}(S_t,\mathbf{w}_t), \text{ or} \tag{11.3}$$

$$\delta_t \doteq R_{t+1} - \bar{R}_t + \hat{v}(S_{t+1},\mathbf{w}_t) - \hat{v}(S_t,\mathbf{w}_t). \tag{11.4}$$

For action values, the one-step algorithm is semi-gradient Expected Sarsa:

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha\delta_t\nabla\hat{q}(S_t, A_t, \mathbf{w}_t), \text{ with} \tag{11.5}$$

$$\delta_t \doteq R_{t+1} + \gamma\sum_a \pi(a|S_{t+1})\hat{q}(S_{t+1}, a, \mathbf{w}_t) - \hat{q}(S_t, A_t, \mathbf{w}_t), \text{ or} \qquad \text{(episodic)}$$

$$\delta_t \doteq R_{t+1} - \bar{R}_t + \sum_a \pi(a|S_{t+1})\hat{q}(S_{t+1}, a, \mathbf{w}_t) - \hat{q}(S_t, A_t, \mathbf{w}_t). \qquad \text{(continuing)}$$

Note that this algorithm does not use importance sampling. In the tabular case it is clear that this is appropriate because the only sample action is A_t, and in learning its value we do not have to consider any other actions. With function approximation it is less clear because we might want to weight different state–action pairs differently once they all contribute to the same overall approximation. Proper resolution of this issue awaits a more thorough understanding of the theory of function approximation in reinforcement learning.

In the multi-step generalizations of these algorithms, both the state-value and action-value algorithms involve importance sampling. For example, the n-step version of semi-gradient Expected Sarsa is

$$\mathbf{w}_{t+n} \doteq \mathbf{w}_{t+n-1} + \alpha\rho_{t+1}\cdots\rho_{t+n-1}\left[G_{t:t+n} - \hat{q}(S_t, A_t, \mathbf{w}_{t+n-1})\right]\nabla\hat{q}(S_t, A_t, \mathbf{w}_{t+n-1}) \tag{11.6}$$

with

$$G_{t:t+n} \doteq R_{t+1} + \cdots + \gamma^{n-1}R_{t+n} + \gamma^n\hat{q}(S_{t+n}, A_{t+n}, \mathbf{w}_{t+n-1}), \text{ or} \qquad \text{(episodic)}$$

$$G_{t:t+n} \doteq R_{t+1} - \bar{R}_t + \cdots + R_{t+n} - \bar{R}_{t+n-1} + \hat{q}(S_{t+n}, A_{t+n}, \mathbf{w}_{t+n-1}), \qquad \text{(continuing)}$$

where here we are being slightly informal in our treatment of the ends of episodes. In the first equation, the ρ_ks for $k \geq T$ (where T is the last time step of the episode) should be taken to be 1, and $G_{t:n}$ should be taken to be G_t if $t + n \geq T$.

Recall that we also presented in Chapter 7 an off-policy algorithm that does not involve importance sampling at all: the n-step tree-backup algorithm. Here is its semi-gradient version:

$$\mathbf{w}_{t+n} \doteq \mathbf{w}_{t+n-1} + \alpha\left[G_{t:t+n} - \hat{q}(S_t, A_t, \mathbf{w}_{t+n-1})\right]\nabla\hat{q}(S_t, A_t, \mathbf{w}_{t+n-1}), \text{ with} \tag{11.7}$$

$$G_{t:t+n} \doteq \hat{q}(S_t, A_t, \mathbf{w}_{t-1}) + \sum_{k=t}^{t+n-1}\delta_k\prod_{i=t+1}^{k}\gamma\pi(A_i|S_i), \tag{11.8}$$

with δ_t as defined at the top of this page for Expected Sarsa. We also defined in Chapter 7 an algorithm that unifies all action-value algorithms: n-step $Q(\sigma)$. We leave the semi-gradient form of that algorithm, and also of the n-step state-value algorithm, as exercises for the reader.

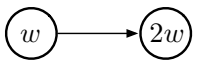
Exercise 11.1 Convert the equation of n-step off-policy TD (7.7) to semi-gradient form. Give accompanying definitions of the return for both the episodic and continuing cases. □

*__Exercise 11.2__ Convert the equations of n-step $Q(\sigma)$ (7.9, 7.14, 7.16, and 7.17) to semi-gradient form. Give definitions that cover both the episodic and continuing cases. □

11.2 Examples of Off-policy Divergence

In this section we begin to discuss the second part of the challenge of off-policy learning with function approximation—that the distribution of updates does not match the on-policy distribution. We describe some instructive counterexamples to off-policy learning—cases where semi-gradient and other simple algorithms are unstable and diverge.

To establish intuitions, it is best to consider first a very simple example. Suppose, perhaps as part of a larger MDP, there are two states whose estimated values are of the functional form w and $2w$, where the parameter vector $\mathbf{w}$ consists of only a single component w. This occurs under linear function approximation if the feature vectors for the two states are each simple numbers (single-component vectors), in this case 1 and 2. In the first state, only one action is available, and it results deterministically in a transition to the second state with a reward of 0:



where the expressions inside the two circles indicate the two state's values.

Suppose initially $w = 10$. The transition will then be from a state of estimated value 10 to a state of estimated value 20. It will look like a good transition, and w will be increased to raise the first

state's estimated value. If γ is nearly 1, then the TD error will be nearly 10, and, if $\alpha = 0.1$, then w will be increased to nearly 11 in trying to reduce the TD error. However, the second state's estimated value will also be increased, to nearly 22. If the transition occurs again, then it will be from a state of estimated value $\approx$11 to a state of estimated value $\approx$22, for a TD error of $\approx$11—larger, not smaller, than before. It will look even more like the first state is undervalued, and its value will be increased again, this time to $\approx$12.1. This looks bad, and in fact with further updates w will diverge to infinity.

To see this definitively we have to look more carefully at the sequence of updates. The TD error on a transition between the two states is

$$\delta_t = R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w}_t) - \hat{v}(S_t, \mathbf{w}_t) = 0 + \gamma 2 w_t - w_t = (2\gamma - 1) w_t,$$

and the off-policy semi-gradient TD(0) update (from (11.2)) is

$$w_{t+1} = w_t + \alpha \rho_t \delta_t \nabla \hat{v}(S_t, w_t) = w_t + \alpha \cdot 1 \cdot (2\gamma - 1) w_t \cdot 1 = \big(1 + \alpha(2\gamma - 1)\big) w_t.$$

Note that the importance sampling ratio, ρ_t, is 1 on this transition because there is only one action available from the first state, so its probabilities of being taken under the target and behavior policies must both be 1. In the final update above, the new parameter is the old parameter times a scalar constant, $1 + \alpha(2\gamma - 1)$. If this constant is greater than 1, then the system is unstable and w will go to positive or negative infinity depending on its initial value. Here this constant is greater than 1 whenever $\gamma > 0.5$. Note that stability does not depend on the specific step size, as long as $\alpha > 0$. Smaller or larger step sizes would affect the rate at which w goes to infinity, but not whether it goes there or not.

Key to this example is that the one transition occurs repeatedly without w being updated on other transitions. This is possible under off-policy training because the behavior policy might select actions on those other transitions which the target policy never would. For these transitions, ρ_t would be zero and no update would be made. Under on-policy training, however, ρ_t is always one. Each time there is a transition from the w state to the $2w$ state, increasing w, there would also have to be a transition out of the $2w$ state. That transition would reduce w, unless it were to a state whose value was higher (because $\gamma < 1$) than $2w$, and then that state would have to be followed by a state of even higher value, or else again w would be reduced. Each state can support the one before only by creating a higher expectation. Eventually the piper must be paid. In the on-policy case the promise of future reward must be kept and the system is kept in check. But in the off-policy case, a promise can be made and then, after taking an action that the target policy never would, forgotten and forgiven.

This simple example communicates much of the reason why off-policy training can lead to divergence, but it is not completely convincing because it is not complete—it is just a fragment of a complete MDP. Can there really be a complete system with instability? A simple complete example of divergence is *Baird's counterexample*. Consider the episodic seven-state, two-action MDP shown in Figure 11.1. The dashed action takes the system to one of the six upper states with equal probability, whereas the solid action takes the system to the seventh state. The behavior policy b selects the dashed and solid actions with probabilities $\frac{6}{7}$ and $\frac{1}{7}$, so that the next-state distribution under it is uniform (the same for all nonterminal states), which is also the starting distribution for each episode. The target policy π always takes the solid action, and so the on-policy distribution (for π) is concentrated in the seventh state. The reward is zero on all transitions. The discount rate is $\gamma = 0.99$.

Consider estimating the state-value under the linear parameterization indicated by the expression shown in each state circle. For example, the estimated value of the leftmost state is $2w_1 + w_8$, where the subscript corresponds to the component of the overall weight vector $\mathbf{w} \in \mathbb{R}^8$; this corresponds to a feature vector for the first state being $\mathbf{x}(1) = (2, 0, 0, 0, 0, 0, 0, 1)^\top$. The reward is zero on all transitions, so the true value function is $v_\pi(s) = 0$, for all s, which can be exactly approximated if $\mathbf{w} = \mathbf{0}$. In fact, there are many solutions, as there are more components to the weight vector (8) than there are nonterminal states (7). Moreover, the set of feature vectors, $\{\mathbf{x}(s) : s \in \mathcal{S}\}$, is a linearly independent set. In all these ways this task seems a favorable case for linear function approximation.

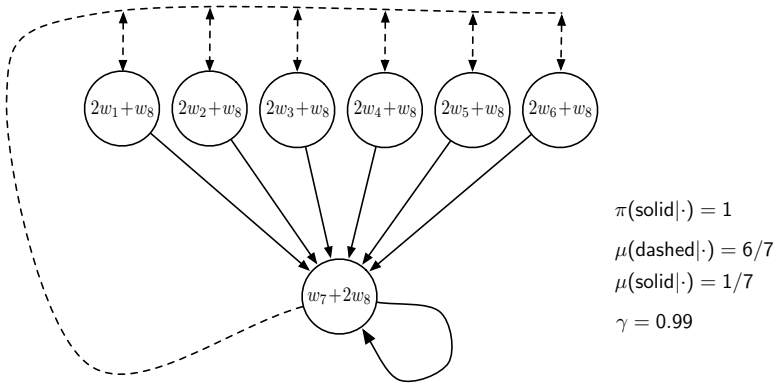


Figure 11.1: Baird's counterexample. The approximate state-value function for this Markov process is of the form shown by the linear expressions inside each state. The solid action usually results in the seventh state, and the dashed action usually results in one of the other six states, each with equal probability. The reward is always zero.

If we apply semi-gradient TD(0) to this problem (11.2), then the weights diverge to infinity, as shown in Figure 11.2 (left). The instability occurs for any positive step size, no matter how small. In fact, it even occurs if a expected update is done as in dynamic programming (DP). If we do a DP-style expected update instead of a sample (learning) update, as shown in Figure 11.2 (right). That is, if the weight vector, $\mathbf{w}_k$, is updated for all states all at the same time in a semi-gradient way, using the DP (expectation-based) target:

$$\mathbf{w}_{k+1} \doteq \mathbf{w}_k + \frac{\alpha}{|\mathcal{S}|} \sum_s \Big(\mathbb{E}[R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w}_k) \mid S_t = s] - \hat{v}(s, \mathbf{w}_k) \Big) \nabla \hat{v}(s, \mathbf{w}_k). \tag{11.9}$$

In this case, there is no randomness and no asynchrony, just as in a classical DP update. The method is conventional except in its use of semi-gradient function approximation. Yet still the system is unstable.

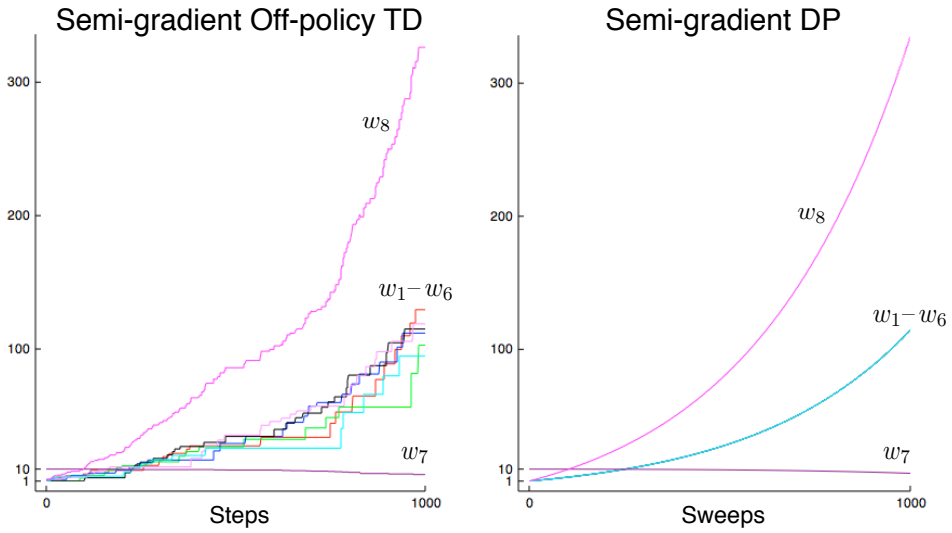


Figure 11.2: Demonstration of instability on Baird's counterexample. Shown are the evolution of the components of the parameter vector $\mathbf{w}$ of the two semi-gradient algorithms. The step size was $\alpha = 0.01$, and the initial weights were $\mathbf{w} = (1, 1, 1, 1, 1, 1, 10, 1)^\top$.

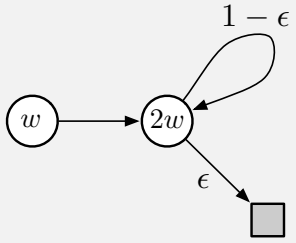
If we alter just the distribution of DP updates in Baird's counterexample, from the uniform distribution to the on-policy distribution (which generally requires asynchronous updating), then convergence is guaranteed to a solution with error bounded by (9.14). This example is striking because the TD and DP methods used are arguably the simplest and best-understood bootstrapping methods, and the linear, semi-descent method used is arguably the simplest and best-understood kind of function approximation. The example shows that even the simplest combination of bootstrapping and function approximation can be unstable if the updates are not done according to the on-policy distribution.

There are also counterexamples similar to Baird's showing divergence for Q-learning. This is cause for concern because otherwise Q-learning has the best convergence guarantees of all control methods. Considerable effort has gone into trying to find a remedy to this problem or to obtain some weaker, but still workable, guarantee. For example, it may be possible to guarantee convergence of Q-learning as long as the behavior policy is sufficiently close to the target policy, for example, when it is the ε-greedy policy. To the best of our knowledge, Q-learning has never been found to diverge in this case, but there has been no theoretical analysis. In the rest of this section we present several other ideas that have been explored.

Suppose that instead of taking just a step toward the expected one-step return on each iteration, as in Baird's counterexample, we actually change the value function all the way to the best, least-squares approximation. Would this solve the instability problem? Of course it would if the feature vectors, $\{\mathbf{x}(s) : s \in \mathcal{S}\}$, formed a linearly independent set, as they do in Baird's counterexample, because then exact approximation is possible on each iteration and the method reduces to standard tabular DP. But of course the point here is to consider the case when an exact solution is *not* possible. In this case stability is not guaranteed even when forming the best approximation at each iteration, as shown by the counterexample in the box.

Tsitsiklis and Van Roy's Counterexample to DP policy evaluation with least-squares linear function approximation

The simplest full counterexample to the least-squares idea is the w-to-$2w$ example (from earlier in this section) extended with a terminal state, as shown to the right. As before, the estimated value of the first state is w, and the estimated value of the second state is $2w$. The reward is zero on all transitions, so the true values are zero at both states, which is exactly representable with $w = 0$. If we set w_{k+1} at each step so as to minimize the $\overline{\text{VE}}$ between the estimated value and the expected one-step return, then we have



$$\begin{aligned} w_{k+1} &= \arg\min_{w\in\mathbb{R}} \sum_{s\in\mathcal{S}} \Big(\hat{v}(s,w) - \mathbb{E}_\pi\big[R_{t+1} + \gamma\hat{v}(S_{t+1},w_k) \;\big|\; S_t = s\big]\Big)^2 \\ &= \arg\min_{w\in\mathbb{R}} \big(w - \gamma 2w_k\big)^2 + \big(2w - (1-\varepsilon)\gamma 2w_k\big)^2 \\ &= \frac{6-4\epsilon}{5}\gamma w_k. \end{aligned} \tag{11.10}$$

The sequence $\{w_k\}$ diverges when $\gamma > \frac{5}{6-4\varepsilon}$ and $w_0 \neq 0$.

Another way to try to prevent instability is to use special methods for function approximation. In particular, stability is guaranteed for function approximation methods that do not extrapolate from the observed targets. These methods, called *averagers*, include nearest neighbor methods and locally weighted regression, but not popular methods such as tile coding and artificial neural networks.

11.3 The Deadly Triad

Our discussion so far can be summarized by saying that the danger of instability and divergence arises whenever we combine all of the following three elements, making up what we call *the deadly triad*:

Function approximation A powerful, scalable way of generalizing from a state space much larger than the memory and computational resources (e.g., linear function approximation or artificial neural networks).

Bootstrapping Update targets that include existing estimates (as in dynamic programming or TD methods) rather than relying exclusively on actual rewards and complete returns (as in MC methods).

Off-policy training Training on a distribution of transitions other than that produced by the target policy. Sweeping through the state space and updating all states uniformly, as in dynamic programming, does not respect the target policy and is an example of off-policy training.

In particular, note that the danger is *not* due to control, or to generalized policy iteration. Those cases are more complex to analyze, but the instability arises in the simpler prediction case whenever it includes all three elements of the deadly triad. The danger is also *not* due to learning or to uncertainties about the environment, because it occurs just as strongly in planning methods, such as dynamic programming, in which the environment is completely known.

If any two elements of the deadly triad are present, but not all three, then instability can be avoided. It is natural, then, to go through the three and see if there is any one that can be given up.

Of the three, *function approximation* most clearly cannot be given up. We need methods that scale to large problems and to great expressive power. We need at least linear function approximation with many features and parameters. State aggregation or nonparametric methods whose complexity grows with data are too weak or too expensive. Least-squares methods such as LSTD are of quadratic complexity and are therefore too expensive for large problems.

Doing without *bootstrapping* is possible, at the cost of computational and data efficiency. Perhaps most important are the losses in computational efficiency. Monte Carlo (non-bootstrapping) methods require memory to save everything that happens between making each prediction and obtaining the final return, and all their computation is done once the final return is obtained. The cost of these computational issues is not apparent on serial von Neumann computers, but would be on specialized hardware. With bootstrapping and eligibility traces (Chapter 12), data can be dealt with when and where it is generated, then need never be used again. The savings in communication and memory made possible by bootstrapping are great.

The losses in data efficiency by giving up *bootstrapping* are also significant. We have seen this repeatedly, such as in Chapters 7 (Figure 7.2) and 9 (Figure 9.2), where some degree of bootstrapping performed much better than Monte Carlo methods on the random-walk prediction task, and in Chapter 10 where the same was seen on the Mountain-Car control task (Figure 10.4). Many other problems show much faster learning with bootstrapping (e.g., see Figure 12.14). Bootstrapping often results in faster learning because it allows learning to take advantage of the state property, the ability to recognize a state upon returning to it. On the other hand, bootstrapping can impair learning on problems where the state representation is poor and causes poor generalization (e.g., this seems to be the case on Tetris, see Şimşek, Algórta, and Kothiyal, 2016). A poor state representation can also result in bias; this is the reason for the poorer bound on the asymptotic approximation quality of bootstrapping methods (Equation 9.14). On balance, the ability to bootstrap has to be considered extremely valuable. One may sometimes choose not to use it by selecting long multistep updates (or a large bootstrapping parameter, $\lambda \approx 1$; see Chapter 12) but often bootstrapping greatly increases efficiency. It is an ability that we would very much like to keep in our toolkit.

Finally, there is *off-policy learning*; can we give that up? On-policy methods are often adequate. For model-free reinforcement learning, one can simply use Sarsa rather than Q-learning. Off-policy methods free behavior from the target policy. This could be considered an appealing convenience but not a necessity. However, off-policy learning *is essential* to other anticipated use cases, cases that we have not yet mentioned in this book but may be important to the larger goal of creating a powerful intelligent agent.

In these use cases, the agent learns not just a single value function and single policy, but large numbers of them in parallel. There is extensive psychological evidence that people and animals learn to predict many different sensory events, not just rewards. We can be surprised by unusual events, and correct our predictions about them, even if they are of neutral valence (neither good nor bad). This kind of prediction presumably underlies predictive models of the world such as are used in planning. We predict what we will see after eye movements, how long it will take to walk home, the probability of making a jump shot in basketball, and the satisfaction we will get from taking on a new project. In all these cases, the events we would like to predict depend on our acting in a certain way. To learn them all, in parallel, requires learning from the one stream of experience. There are many target policies, and thus the one behavior policy cannot equal all of them. Yet parallel learning is conceptually possible because the behavior policy may overlap in part with many of the target policies. To take full advantage of this requires off-policy learning.

11.4 Linear Value-function Geometry

To better understand the stability challenge of off-policy learning, it is helpful to think about value function approximation more abstractly and independently of how learning is done. We can imagine the space of all possible state-value functions—all functions from states to real numbers $v : \mathcal{S} \to \mathbb{R}$. Most of these value functions do not correspond to any policy. More important for our purposes is that most are not representable by the function approximator, which by design has far fewer parameters than there are states.

Given an enumeration of the state space $\mathcal{S} = \{s_1, s_2, \ldots, s_{|\mathcal{S}|}\}$, any value function v corresponds to a vector listing the value of each state in order $[v(s_1), v(s_2), \ldots, v(s_{|\mathcal{S}|})]^\top$. This vector representation of a value function has as many components as there are states. In most cases where we want to use function approximation, this would be far too many components to represent the vector explicitly. Nevertheless, the idea of this vector is conceptually useful. In the following, we treat a value function and its vector representation interchangably.

To develop intuitions, consider the case with three states $\mathcal{S} = \{s_1, s_2, s_3\}$ and two parameters $\mathbf{w} = (w_1, w_2)^\top$. We can then view all value functions/vectors as points in a three-dimensional space. The parameters provide an alternative coordinate system over a two-dimensional subspace. Any weight vector $\mathbf{w} = (w_1, w_2)^\top$ is a point in the two-dimensional subspace and thus also a complete value function $v_\mathbf{w}$ that assigns values to all three states. With general function approximation the relationship between the full space and the subspace of representable functions could be complex, but in the case of *linear* value-function approximation the subspace is a simple plane, as suggested by Figure 11.3.

Now consider a single fixed policy π. We assume that its true value function, v_π, is too complex to be represented exactly as an approximation. Thus v_π is not in the subspace; in the figure it is depicted as being above the planar subspace of representable functions.

If v_π cannot be represented exactly, what representable value function is closest to it? This turns out to be a subtle question with multiple answers. To begin, we need a measure of the distance between two value functions. Given two value functions v_1 and v_2, we can talk about the vector difference between them, $v = v_1 - v_2$. If v is small, then the two value functions are close to each other. But how are we to measure the size of this difference vector? The conventional Euclidean norm is not appropriate

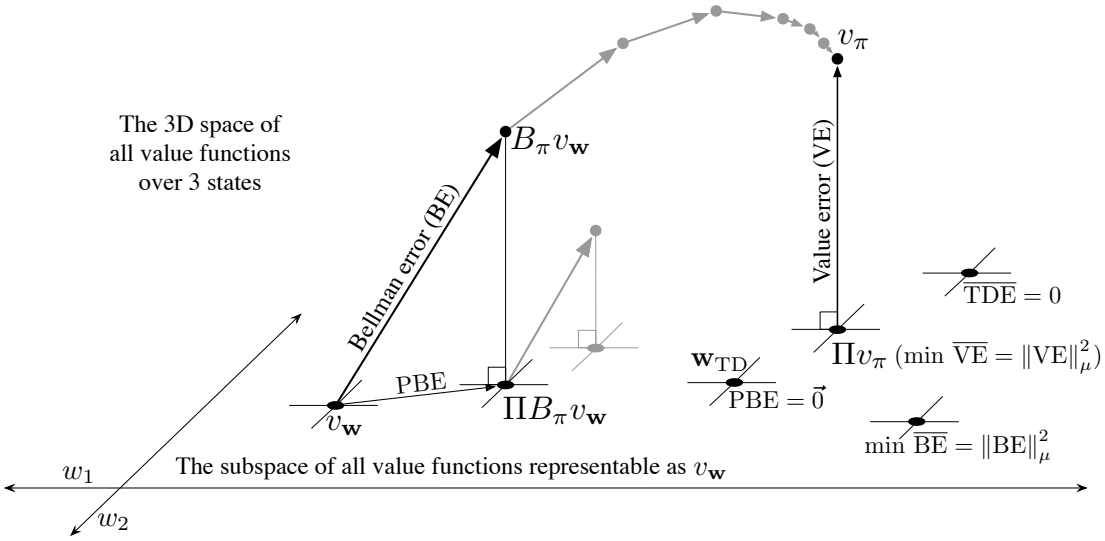


Figure 11.3: The geometry of linear value-function approximation. Shown is the three-dimensional space of all value functions over three states, while shown as a plane is the subspace of all value functions representable by a linear function approximator with parameter $\mathbf{w} = (w_1, w_2)^\top$. The true value function v_π is in the larger space and can be projected down (into the subspace, using a projection operator Π) to its best approximation in the value error (VE) sense. The best approximators in the Bellman error (BE), projected Bellman error (PBE), and temporal difference error (TDE) senses are all potentially different and are shown in the lower right. (VE, BE, and PBE are all treated as the corresponding vectors in this figure.) The Bellman operator takes a value function in the plane to one outside, which can then be projected back. If you iteratively applied the Bellman operator outside the space (shown in gray above) you would reach the true value function, as in conventional dynamic programming. If instead you kept projecting back into the subspace at each step, as in the lower step shown in gray, then the fixed point would be the point of vector-zero PBE.

because, as discussed in Section 9.2, some states are more important than others because they occur more frequently or because we are more interested in them (Section 9.10). As in Section 9.2, let us use the weighting $\mu : \mathcal{S} \to \mathbb{R}$ to specify the degree to which we care about different states being accurately valued (often taken to be the on-policy distribution). We can then define the distance between value functions using the norm

$$\|v\|^2_\mu \doteq \sum_{s \in \mathcal{S}} \mu(s) v(s)^2. \tag{11.11}$$

Note that the $\overline{\text{VE}}$ from Section 9.2 can be written simply using this norm as $\overline{\text{VE}}(\mathbf{w}) = \|v_{\mathbf{w}} - v_\pi\|^2_\mu$. For any value function v, the operation of finding its closest value function in the subspace of representable value functions is a projection operation. We define a projection operator Π that takes an arbitrary value function to the representable function that is closest in our norm:

$$\Pi v \doteq v_{\mathbf{w}} \quad \text{where} \quad \mathbf{w} = \arg\min_{\mathbf{w}} \|v - v_{\mathbf{w}}\|^2_\mu \,. \tag{11.12}$$

The representable value function that is closest to the true value function v_π is thus its projection, Πv_π, as suggested in Figure 11.3. This is the solution asymptotically found by Monte Carlo methods, albeit often very slowly. The projection operation is discussed more fully in the box on the next page.

The projection matrix

For a linear function approximator, the projection operation is linear, which implies that it can be represented as an $|\mathcal{S}| \times |\mathcal{S}|$ matrix:

$$\Pi \doteq \mathbf{X}\left(\mathbf{X}^\top \mathbf{D}\mathbf{X}\right)^{-1}\mathbf{X}^\top \mathbf{D}, \tag{11.13}$$

where, as in Section 9.4, $\mathbf{D}$ denotes the $|\mathcal{S}| \times |\mathcal{S}|$ diagonal matrix with the $\mu(s)$ on the diagonal, and $\mathbf{X}$ denotes the $|\mathcal{S}| \times d$ matrix whose rows are the feature vectors $\mathbf{x}(s)^\top$, one for each state s. If the inverse in does not exist, then the pseudoinverse is substituted. Using these matrices, the norm of a vector can be written

$$\|v\|_\mu^2 = v^\top \mathbf{D} v, \tag{11.14}$$

and the approximate linear value function can be written

$$v_{\mathbf{w}} = \mathbf{X}\mathbf{w}. \tag{11.15}$$

TD methods find different solutions. To understand their rationale, recall that the Bellman equation for value function v_π is

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r \,|\, s, a)\left[r + \gamma v_\pi(s')\right], \qquad \text{for all } s \in \mathcal{S}. \tag{11.16}$$

v_π is the only value function that solves this equation exactly. If an approximate value function $v_{\mathbf{w}}$ were substituted for v_π, the difference between the right and left sides of the modified equation could be used as a measure of how far off $v_{\mathbf{w}}$ is from v_π. We call this the *Bellman error* at state s:

$$\bar{\delta}_{\mathbf{w}}(s) \doteq \left(\sum_a \pi(a|s) \sum_{s',r} p(s', r \,|\, s, a)\left[r + \gamma v_{\mathbf{w}}(s')\right]\right) \; - \; v_{\mathbf{w}}(s) \tag{11.17}$$

$$= \mathbb{E}\big[R_{t+1} + \gamma v_{\mathbf{w}}(S_{t+1}) - v_{\mathbf{w}}(S_t) \;\big|\; S_t = s, A_t \sim \pi\big], \tag{11.18}$$

which shows clearly the relationship of the Bellman error to the TD error (11.3). The Bellman error is the expectation of the TD error.

The vector of all the Bellman errors, at all states, $\bar{\delta}_{\mathbf{w}} \in \mathbb{R}^{|\mathcal{S}|}$, is called the *Bellman error vector* (shown as BE in Figure 11.3). The overall size of this vector, in the norm, is an overall measure of the error in the value function, called the *Mean Squared Bellman Error*:

$$\overline{\text{BE}}(\mathbf{w}) = \left\|\bar{\delta}_{\mathbf{w}}\right\|_\mu^2. \tag{11.19}$$

It is not possible in general to reduce the $\overline{\text{BE}}$ to zero (at which point $v_{\mathbf{w}} = v_\pi$), but for linear function approximation there is a unique value of $\mathbf{w}$ for which the $\overline{\text{BE}}$ is minimized. This point in the representable-function subspace (labeled $\min \overline{\text{BE}}$ in Figure 11.3) is different in general from that which minimizes the $\overline{\text{VE}}$ (shown as Πv_π). Methods that seek to minimize the $\overline{\text{BE}}$ are discussed in the next two sections.

The Bellman error vector is shown in Figure 11.3 as the result of applying the *Bellman operator* $B_\pi : \mathbb{R}^{|\mathcal{S}|} \to \mathbb{R}^{|\mathcal{S}|}$ to the approximate value function. The Bellman operator is defined by

$$(B_\pi v)(s) \doteq \sum_a \pi(a|s) \sum_{s',r} p(s', r \,|\, s, a)\left[r + \gamma v(s')\right], \tag{11.20}$$

for all $s \in \mathcal{S}$ and $v : \mathcal{S} \to \mathbb{R}$. The Bellman error vector for v can be written $\bar{\delta}_{\mathbf{w}} = B_\pi v_{\mathbf{w}} - v_{\mathbf{w}}$.

If the Bellman operator is applied to a value function in the representable subspace, then, in general, it will produce a new value function that is outside the subspace, as suggested in the figure. In dynamic programming (without function approximation), this operator is applied repeatedly to the points outside the representable space, as suggested by the gray arrows in the top of Figure 11.3. Eventually that process converges to the true value function v_π, the only fixed point for the Bellman operator, the only value function for which

$$v_\pi = B_\pi v_\pi, \tag{11.21}$$

which is just another way of writing the Bellman equation for π (11.16).

With function approximation, however, the intermediate value functions lying outside the subspace cannot be represented. The gray arrows in the upper part of Figure 11.3 cannot be followed because after the first update (dark line) the value function must be projected back into something representable. The next iteration then begins within the subspace; the value function is again taken outside of the subspace by the Bellman operator and then mapped back by the projection operator, as suggested by the lower gray arrow and line. Following these arrows is a DP-like process with approximation.

In this case we are interested in the projection of the Bellman error vector back into the representable space. This is the projected Bellman error vector $\Pi\bar{\delta}_{v_\mathbf{w}}$, shown in Figure 11.3 as PBE. The size of this vector, in the norm, is another measure of error in the approximate value function. For any approximate value function v, we define the *Mean Square Projected Bellman Error*, denoted $\overline{\text{PBE}}$, as

$$\overline{\text{PBE}}(\mathbf{w}) = \left\|\Pi\bar{\delta}_\mathbf{w}\right\|_\mu^2. \tag{11.22}$$

With linear function approximation there always exists an approximate value function (within the subspace) with zero $\overline{\text{PBE}}$; this is the TD fixed point, $\mathbf{w}_{\text{TD}}$, introduced in Section 9.4. As we have seen, this point is not always stable under semi-gradient TD methods and off-policy training. As shown in the figure, this value function is generally different from those minimizing $\overline{\text{VE}}$ or $\overline{\text{BE}}$. Methods that are guaranteed to converge to it are discussed in Sections 11.7 and 11.8.

11.5 Stochastic Gradient Descent in the Bellman Error

Armed with a better understanding of value function approximation and its various objectives, we return now to the challenge of stability in off-policy learning. We would like to apply the approach of stochastic gradient descent (SGD, Section 9.3), in which updates are made that in expectation are equal to the negative gradient of an objective function. These methods always go downhill (in expectation) in the objective and because of this are typically stable with excellent convergence properties. Among the algorithms investigated so far in this book, only the Monte Carlo methods are true SGD methods. These methods converge robustly under both on-policy and off-policy training as well as for general non-linear (differentiable) function approximators, though they are often slower than semi-gradient methods with bootstrapping, which are not SGD methods. Semi-gradient methods may diverge under off-policy training, as we have seen earlier in this chapter, and under contrived cases of non-linear function approximation (Tsitsiklis and Van Roy, 1997). With a true SGD method such divergence would not be possible.

The appeal of SGD is so strong that great effort has gone into finding a practical way of harnessing it for reinforcement learning. The starting place of all such efforts is the choice of an error or objective function to optimize. In this and the next section we explore the origins and limits of the most popular proposed objective function, that based on the *Bellman error* introduced in the previous section. Although this has been a popular and influential approach, the conclusion that we reach here is that it is a misstep and yields no good learning algorithms. On the other hand, this approach fails in an interesting way that offers insight into what might constitute a good approach.

To begin, let us consider not the Bellman error, but something more immediate and naive. Temporal difference learning is driven by the TD error. Why not take the minimization of the expected square of the TD error as the objective? In the general function-approximation case, the one-step TD error with discounting is

$$\delta_t = R_{t+1} + \gamma \hat{v}(S_{t+1},\mathbf{w}_t) - \hat{v}(S_t,\mathbf{w}_t).$$

A possible objective function then is what one might call the *Mean Squared TD Error*:

$$\begin{aligned}
\overline{\text{TDE}}(\mathbf{w}) &= \sum_{s\in\mathcal{S}} \mu(s)\mathbb{E}\big[\delta_t^2 \;\big|\; S_t{=}s, A_t{\sim}\pi\big] \\
&= \sum_{s\in\mathcal{S}} \mu(s)\mathbb{E}\big[\rho_t\delta_t^2 \;\big|\; S_t{=}s, A_t{\sim}b\big] \\
&= \mathbb{E}_b\big[\rho_t\delta_t^2\big]\,. \qquad \text{(if } \mu \text{ is the distribution encountered under } b\text{)}
\end{aligned}$$

The last equation is of the form needed for SGD; it gives the objective as an expectation that can be sampled from experience (remember the experience is due to the behavior policy b. Thus, following the standard SGD approach, one can derive the per-step update based on a sample of this expected value:

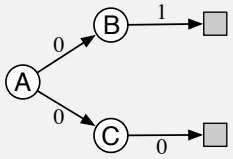
$$\begin{aligned}
\mathbf{w}_{t+1} &= \mathbf{w}_t - \frac{1}{2}\alpha\nabla(\rho_t\delta_t^2) \\
&= \mathbf{w}_t - \alpha\rho_t\delta_t\nabla\delta_t \\
&= \mathbf{w}_t + \alpha\rho_t\delta_t\big(\nabla\hat{v}(S_t,\mathbf{w}_t) - \gamma\nabla\hat{v}(S_{t+1},\mathbf{w}_t)\big),
\end{aligned} \tag{11.23}$$

which you will recognize as the same as the semi-gradient TD algorithm (11.2) except for the additional final term. This term completes the gradient and makes this a true SGD algorithm with excellent convergence guarantees. Let us call this algorithm the *naive residual-gradient* algorithm (after Baird, 1993).

Although the naive residual-gradient algorithm converges robustly, it does not always converge to a desirable place, as the *A-split example* in the box shows. In this example a tabular representation is used, so the true state values can be exactly represented, yet the naive residual-gradient algorithm finds different values, and these values have lower $\overline{\text{TDE}}$ than do the true values. Minimizing the $\overline{\text{TDE}}$ is naive; by penalizing all TD errors it achieves something more like temporal smoothing than accurate prediction.

A-split example, showing the naiveté of the naive residual gradient algorithm

Consider the following three-state episodic MRP:



Episodes begin in state A and then 'split' stochastically, half the time going to B and then invariably going on to terminate with a reward of 1, and half the time going to state C and then invariably terminating with a reward of zero. Reward for the first transition, out of A, is always zero whichever way the episode goes. As this is an episodic problem, we can take γ to be 1. We also assume on-policy training, so that ρ_t is always 1, and tabular function approximation, so that the learning algorithms are free to give arbitrary, independent values to all three states. So it should be an easy problem.

What should the values be? From A, half the time the return is 1, and half the time the return is 0; A should have value $\frac{1}{2}$. From B the return is always 1, so its value should be 1, and similarly from C the return is always 0, so its value should be 0. These are the true values and, as this is a tabular problem, all the methods presented previously converge to them exactly.

However, the naive residual-gradient algorithm finds different values for B and C. It converges with B having a value of $\frac{3}{4}$ and C having a value of $\frac{1}{4}$ (A converges correctly to $\frac{1}{2}$). These are in fact the values that minimize the $\overline{\text{TDE}}$.

Let us compute the $\overline{\text{TDE}}$ for these values. The first transition of each episode is either up from A's $\frac{1}{2}$ to B's $\frac{3}{4}$, a change of $\frac{1}{4}$, or down from A's $\frac{1}{2}$ to C's $\frac{1}{4}$, a change of $-\frac{1}{4}$. Because the reward is zero on these transitions, and $\gamma = 1$, these changes *are* the TD errors, and thus the squared TD error is always $\frac{1}{16}$ on the first transition. The second transition is similar; it is either up from B's $\frac{3}{4}$ to a reward of 1 (and a terminal state of value 0), or down from C's $\frac{1}{4}$ to a reward of 0 (again with a terminal state of value 0). Thus, the TD error is always $\pm\frac{1}{4}$, for a squared error of $\frac{1}{16}$ on the second step. Thus, for this set of values, the $\overline{\text{TDE}}$ on both steps is $\frac{1}{16}$.

Now let's compute the $\overline{\text{TDE}}$ for the true values (B at 1, C at 0, and A at $\frac{1}{2}$). In this case the first transition is either from $\frac{1}{2}$ up to 1, at B, or from $\frac{1}{2}$ down to 0, at C; in either case the absolute error is $\frac{1}{2}$ and the squared error is $\frac{1}{4}$. The second transition has zero error because the starting value, either 1 or 0 depending on whether the transition is from B or C, always exactly matches the immediate reward and return. Thus the squared TD error is $\frac{1}{4}$ on the first transition and 0 on the second, for a mean reward over the two transitions of $\frac{1}{8}$. As $\frac{1}{8}$ is bigger that $\frac{1}{16}$, this solution is worse according to the $\overline{\text{TDE}}$. On this simple problem, the true values do not have the smallest $\overline{\text{TDE}}$.

A better idea would seem to be minimizing the Bellman error. If the exact values are learned, the Bellman error is zero everywhere. Thus, a Bellman-error-minimizing algorithm should have no trouble with the A-split example. We cannot expect to achieve zero Bellman error in general, as it would involve finding the true value function, which we presume is outside the space of representable value functions. But getting close to this ideal is a natural-seeming goal. As we have seen, the Bellman error is also closely related to the TD error. The Bellman error for a state is the expected TD error in that state. So let's repeat the derivation above with the expected TD error (all expectations here are implicitly conditional on S_t):

$$\begin{aligned}
\mathbf{w}_{t+1} &= \mathbf{w}_t - \frac{1}{2}\alpha\nabla\big(\mathbb{E}_\pi[\delta_t]^2\big) \\
&= \mathbf{w}_t - \frac{1}{2}\alpha\nabla\big(\mathbb{E}_b[\rho_t\delta_t]^2\big) \\
&= \mathbf{w}_t - \alpha\mathbb{E}_b[\rho_t\delta_t]\,\nabla\mathbb{E}_b[\rho_t\delta_t] \\
&= \mathbf{w}_t - \alpha\mathbb{E}_b\big[\rho_t(R_{t+1} + \gamma\hat{v}(S_{t+1},\mathbf{w}) - \hat{v}(S_t,\mathbf{w}))\big]\,\mathbb{E}_b[\rho_t\nabla\delta_t] \\
&= \mathbf{w}_t + \alpha\Big[\mathbb{E}_b\big[\rho_t(R_{t+1} + \gamma\hat{v}(S_{t+1},\mathbf{w}))\big] - \hat{v}(S_t,\mathbf{w})\Big]\Big[\nabla\hat{v}(S_t,\mathbf{w}) - \gamma\mathbb{E}_b\big[\rho_t\nabla\hat{v}(S_{t+1},\mathbf{w})\big]\Big].
\end{aligned}$$

This update and various ways of sampling it are referred to as the *residual gradient algorithm*. If you simply used the sample values in all the expectations, then the equation above reduces almost exactly to (11.23), the naive residual-gradient algorithm.[1] But this is naive, because the equation above involves the next state, S_{t+1}, appearing in two expectations that are multiplied together. To get an unbiased

[1] For state values there remains a small difference in the treatment of the importance sampling ratio ρ_t. In the analagous action-value case (which is the most important case for control algorithms), the residual gradient algorithm would reduce exactly to the naive version.

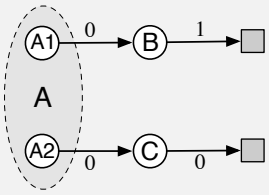
sample of the product, two independent samples of the next state are required, but during normal interaction with an external environment only one is obtained. One expectation or the other can be sampled, but not both.

There are two ways to make the residual gradient algorithm work. One is in the case of deterministic environments. If the transition to the next state is deterministic, then the two samples will necessarily be the same, and the naive algorithm is valid. The other way is to obtain *two* independent samples of the next state, S_{t+1}, from S_t, one for the first expectation and another for the second expectation. In real interaction with an environment, this would not seem possible, but when interacting with a simulated environment, it is. One simply rolls back to the previous state and obtains an alternate next state before proceeding forward from the first next state. In either of these cases the residual gradient algorithm is guaranteed to converge to a minimum of the $\overline{\text{BE}}$ under the usual conditions on the step-size parameter. As a true SGD method, this convergence is robust, applying to both linear and non-linear function approximators. In the linear case, convergence is always to the *unique* $\mathbf{w}$ that minimizes the $\overline{\text{BE}}$.

However, there remain at least three ways in which the convergence of the residual gradient method is unsatisfactory. The first of these is that empirically it is slow, much slower that semi-gradient methods. Indeed, proponents of this method have proposed increasing its speed by combining it with faster semi-gradient methods initially, then gradually switching over to residual gradient for the convergence guarantee (Baird and Moore, 1999). The second way in which the residual-gradient algorithm is unsatisfactory is that it still seems to converge to the wrong values. It does get the right values in all tabular cases, such as the A-split example, as for those an exact solution to the Bellman equation is possible. But if we examine examples with genuine function approximation, then the residual-gradient algorithm, and indeed the $\overline{\text{BE}}$ objective, seem to find the wrong value functions. One of the most telling such examples is the variation on the A-split example shown in the box. On the A-*pre*split example the residual-gradient algorithm finds the same poor solution as its naive version. This example shows intuitively that minimizing the $\overline{\text{BE}}$ (which the residual-gradient algorithm surely does) may not be a desirable goal.

A-presplit example, a counterexample for the $\overline{\text{BE}}$

Consider the following three-state episodic MRP:



Episodes start in either A1 or A2, with equal probability. These two states look exactly the same to the function approximator, like a single state A whose feature representation is distinct from and unrelated to the feature representation of the other two states, B and C, which are also distinct from each other. Specifically, the parameter of the function approximator has three components, one giving the value of state B, one giving the value of state C, and one giving the value of both states A1 and A2. Other than the selection of the initial state, the system is deterministic. If it starts in A1, then it transitions to B with a reward of 0 and then on to termination with a reward of 1. If it starts in A2, then it transitions to C, and then to termination, with both rewards zero.

To a learning algorithm, seeing only the features, the system looks identical to the A-split example. The system seems to always start in A, followed by either B or C with equal probability, and then terminating with a 1 or a 0 depending deterministically on the previous state. As in

the A-split example, the true values of B and C are 1 and 0, and the best shared value of A1 and A2 is $\frac{1}{2}$, by symmetry.

Because this problem appears externally identical to the A-split example, we already know what values will be found by the algorithms. Semi-gradient TD converges to the ideal values just mentioned, while the naive residual-gradient algorithm converges to values of $\frac{3}{4}$ and $\frac{1}{4}$ for B and C respectively. All state transitions are deterministic, so the non-naive residual-gradient algorithm will also converge to these values (it is the same algorithm in this case). It follows then that this 'naive' solution must also be the one that minimizes the $\overline{\text{BE}}$, and so it is. On a deterministic problem, the Bellman errors and TD errors are all the same, so the $\overline{\text{BE}}$ is always the same as the $\overline{\text{TDE}}$. Optimizing the $\overline{\text{BE}}$ on this example gives rise to the same failure mode as with the naive residual-gradient algorithm on the A-split example.

The third way in which the convergence of the residual-gradient algorithm is not satisfactory is explained in the next section. Like the second way, the third way is also a problem with the $\overline{\text{BE}}$ objective itself rather than with any particular algorithm for achieving it.

11.6 The Bellman Error is Not Learnable

The concept of learnability that we introduce in this section is different from that commonly used in machine learning. There, a hypothesis is said to be "learnable" if it is *efficiently* learnable, meaning that it can be learned within a polynomial rather than an exponential number of examples. Here we use the term in a more basic way, to mean learnable at all, with any amount of experience. It turns out many quantities of apparent interest in reinforcement learning cannot be learned even from an infinite amount of experiential data. These quantities are well defined and can be computed given knowledge of the internal structure of the environment, but cannot be computed or estimated from the observed sequence of feature vectors, actions, and rewards.[2] We say that they are not *learnable*. It will turn out that the Bellman error objective ($\overline{\text{BE}}$) introduced in the last two sections is not learnable in this sense. That the Bellman error objective cannot be learned from the observable data is probably the strongest reason not to seek it.

To make the concept of learnability clear, let's start with some simple examples. Consider the two Markov reward processes[3] (MRPs) diagrammed below:

Where two edges leave a state, both transitions are assumed to occur with equal probability, and the numbers indicate the reward received. All the states appear the same; they all produce the same single-component feature vector $x = 1$ and have approximated value w. Thus, the only varying part of the data trajectory is the reward sequence. The left MRP stays in the same state and emits an endless stream of 0s and 2s at random, each with 0.5 probability. The right MRP, on every step, either stays in its current state or switches to the other, with equal probability. The reward is deterministic in this MRP, always a 0 from one state and always a 2 from the other, but because the each state is equally likely on each step, the observable data is again an endless stream of 0s and 2s at random, identical to that produced by the left MRP. (We can assume the right MRP starts in one of two states at random

[2] They would of course be estimated if the *state* sequence were observed rather than only the corresponding feature vectors.

[3] All MRPs can be considered MDPs with a single action in all states; what we conclude about MRPs here applies as well to MDPs.

with equal probability.) Thus, even given even an infinite amount of data, it would not be possible to tell which of these two MRPs was generating it. In particular, we could not tell if the MRP has one state or two, is stochastic or deterministic. These things are not learnable.

This pair of MRPs also illustrates that the $\overline{\text{VE}}$ objective (9.1) is not learnable. If $\gamma = 0$, then the true values of the three states (in both MRPs), left to right, are 1, 0, and 2. Suppose $w = 1$. Then the $\overline{\text{VE}}$ is 0 for the left MRP and 1 for the right MRP. Because the $\overline{\text{VE}}$ is different in the two problems, yet the data generated has the same distribution, the $\overline{\text{VE}}$ cannot be learned. The $\overline{\text{VE}}$ is not a unique function of the data distribution. And if it cannot be learned, then how could the $\overline{\text{VE}}$ possibly be useful as an objective for learning?

If an objective cannot be learned, it does indeed draw its utility into question. In the case of the $\overline{\text{VE}}$, however, there is a way out. Note that the same solution, $w = 1$, is optimal for both MRPs above (assuming μ is the same for the two indistinguishable states in the right MRP). Is this a coincidence, or could it be generally true that all MDPs with the same data distribution also have the same optimal parameter vector? If this is true—and we will show next that it is—then the $\overline{\text{VE}}$ remains a usable objective. The $\overline{\text{VE}}$ is not learnable, but the parameter that optimizes it is!

To understand this, it is useful to bring in another natural objective function, this time one that is clearly learnable. One error that is always observable is that between the value estimate at each time and the return from that time. The Mean Square Return Error, denoted $\overline{\text{RE}}$, is the expectation, under μ, of the square of this error. In the on-policy case the $\overline{\text{RE}}$ can be written

$$\begin{aligned}\overline{\text{RE}}(\mathbf{w}) &= \mathbb{E}\Big[\big(G_t - \hat{v}(S_t,\mathbf{w})\big)^2\Big] \\ &= \overline{\text{VE}}(\mathbf{w}) + \mathbb{E}\Big[\big(G_t - v_\pi(S_t)\big)^2\Big]. \end{aligned} \tag{11.24}$$

Thus, the two objectives are the same except for a variance term that does not depend on the parameter vector. The two objectives must therefore have the same optimal parameter value $\mathbf{w}^*$. The overall relationships are summarized in Figure 11.4.

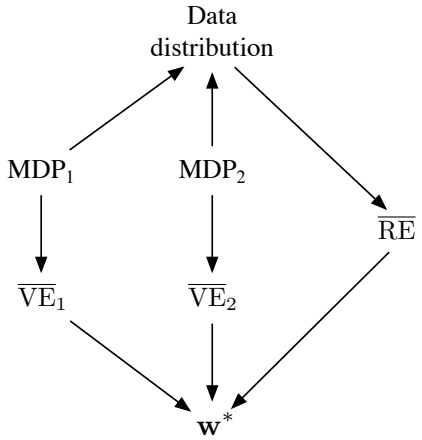


Figure 11.4: Causal relationships among the data distribution, MDPs, and errors for Monte-Carlo objectives. Two different MDPs can produce the same data distribution yet also produce different $\overline{\text{VE}}$s, proving that the $\overline{\text{VE}}$ objective cannot be determined from data and is not learnable. However, all such $\overline{\text{VE}}$s must have the same optimal parameter vector, $\mathbf{w}^*$! Moreover, this same $\mathbf{w}^*$ can be determined from another objective, the $\overline{\text{RE}}$, which *is* uniquely determined from the data distribution. Thus $\mathbf{w}^*$ and the $\overline{\text{RE}}$ are learnable even though the $\overline{\text{VE}}$s are not.

***Exercise 11.3** Prove (11.24). Hint: Write the $\overline{\text{RE}}$ as an expectation over possible states s of the expectation of the squared error given that $S_t = s$. Then add and subtract the true value of state s

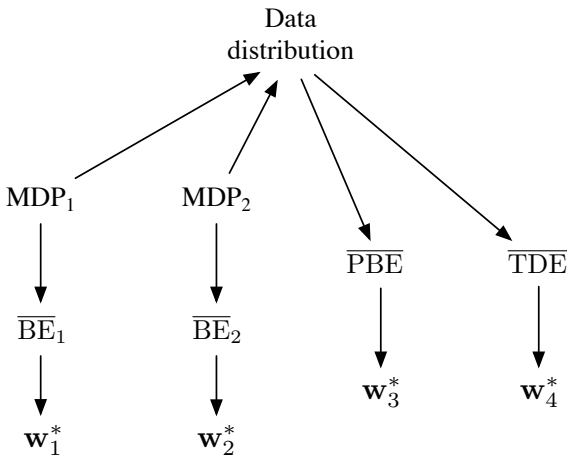


Figure 11.5: Causal relationships among the data distribution, MDPs, and errors for bootstrapping objectives. Two different MDPs can produce the same data distribution yet also produce different $\overline{\text{BE}}$s *and* have different minimizing parameter vectors; these are not learnable from the data distribution. The $\overline{\text{PBE}}$ and $\overline{\text{TDE}}$ objectives and their (different) minima can be directly determined from data and thus are learnable.

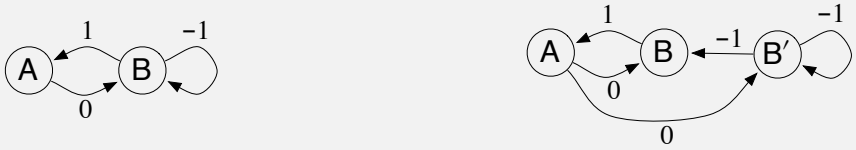
from the error (before squaring), grouping the subtracted true value with the return and the added true value with the estimated value. Then, if you expand the square, the most complex term will end up being zero, leaving you with (11.24).

Now let us return to the $\overline{\text{BE}}$. The $\overline{\text{BE}}$ is like the $\overline{\text{VE}}$ in that it can be computed from knowledge of the MDP but is not learnable from data. But it is not like the $\overline{\text{VE}}$ in that its minimum solution is not learnable. The box on the next page presents a counterexample—two MRPs that generate the same data distribution but whose minimizing parameter vector is different, proving that the optimal parameter vector is not a function of the data and thus cannot be learned from it. The other bootstrapping objectives that we have considered, the $\overline{\text{PBE}}$ and $\overline{\text{TDE}}$, can be determined from data (are learnable) and determine optimal solutions that are in general different from each other and the $\overline{\text{BE}}$ minimums. The general case is summarized in Figure 11.5.

Thus, the $\overline{\text{BE}}$ is not learnable; it cannot be estimated from feature vectors and other observable data. This limits the $\overline{\text{BE}}$ to model-based settings. There can be no algorithm that minimizes the $\overline{\text{BE}}$ without access to the underlying MDP states beyond the feature vectors. The residual-gradient algorithm is only able to minimize $\overline{\text{BE}}$ because it is allowed to double sample from the same state—not a state that has the same feature vector, but one that is guaranteed to be the same underlying state. We can see now that there is no way around this. Minimizing the $\overline{\text{BE}}$ requires some such access to the nominal, underlying MDP. This is an important limitation of the $\overline{\text{BE}}$ beyond that identified in the A-presplit example on page 222. All this directs more attention toward the $\overline{\text{PBE}}$.

Counterexample to the learnability of the $\overline{\text{BE}}$ and its minima

To show the full range of possibilities we need a slightly more complex pair of Markov reward processes (MRPs) than those considered earlier. Consider the following two MRPs:



Where two edges leave a state, both transitions are assumed to occur with equal probability, and the numbers indicate the reward received. The MRP on the left has two states that are represented distinctly. The MRP on the right has three states, two of which, B and B′, appear the same and must be given the same approximate value. Specifically, $\mathbf{w}$ has two components and the value of state A is given by the first component and the value of B and B′ is given by the second. The second MRP has been designed so that equal time is spent in all three states, so we can take $\mu(s) = \frac{1}{3}$, for all s.

Note that the observable data distribution is identical for the two MRPs. In both cases the agent will see single occurrences of A followed by a 0, then some number of apparent Bs, each followed by a -1 except the last, which is followed by a 1, then we start all over again with a single A and a 0, etc. All the statistical details are the same as well; in both MRPs, the probability of a string of k Bs is 2^{-k}.

Now suppose $\mathbf{w} = \mathbf{0}$. In the first MRP, this is an exact solution, and the $\overline{\text{BE}}$ is zero. In the second MRP, this solution produces a squared error in both B and B′ of 1, such that $\overline{\text{BE}} = \mu(\text{B})1 + \mu(\text{B}')1 = \frac{2}{3}$. These two MRPs, which generate the same data distribution, have different $\overline{\text{BE}}$s; the $\overline{\text{BE}}$ is not learnable.

Moreover (and unlike the earlier example for the $\overline{\text{VE}}$) the minimizing value of $\mathbf{w}$ is different for the two MRPs. For the first MRP, $\mathbf{w} = \mathbf{0}$ minimizes the $\overline{\text{BE}}$ for any γ. For the second MRP, the minimizing $\mathbf{w}$ is a complicated function of γ, but in the limit, as $\gamma \to 1$, it is $(-\frac{1}{2}, 0)^\top$. Thus the solution that minimizes $\overline{\text{BE}}$ cannot be estimated from data alone; knowledge of the MRP beyond what is revealed in the data is required. In this sense, it is impossible in principle to pursue the $\overline{\text{BE}}$ as an objective for learning.

It may be surprising that in the second MRP the $\overline{\text{BE}}$-minimizing value of A is so far from zero. Recall that A has a dedicated weight and thus its value is unconstrained by function approximation. A is followed by a reward of 0 and transition to a state with a value of nearly 0, which suggests $v_{\mathbf{w}}(\text{A})$ should be 0; why is its optimal value substantially negative rather than 0? The answer is that making the value of A negative reduces the error upon arriving in A from B. The reward on this deterministic transition is 1, which implies that B should have a value 1 more than A. Because B's value is approximately zero, A's value is driven toward -1. The $\overline{\text{BE}}$-minimizing value of $\approx -\frac{1}{2}$ for A is a compromise between reducing the errors on leaving and on entering A.

11.7 Gradient-TD Methods

We now consider SGD methods for minimizing the $\overline{\text{PBE}}$. As true SGD methods, these *gradient-TD methods* have robust convergence properties even under off-policy training and non-linear function approximation. Remember that in the linear case there is always an exact solution, the TD fixed point $\mathbf{w}_{\text{TD}}$, at which the $\overline{\text{PBE}}$ is zero. This solution could be found by least-squares methods (Section 9.7), but only by methods of quadratic $O(d^2)$ complexity in the number of parameters. We seek instead an SGD method, which should be $O(d)$ and have robust convergence properties. Gradient-TD methods come close to achieving these goals, at the cost of a rough doubling of computational complexity.

To derive an SGD method for the $\overline{\text{PBE}}$ (assuming linear function approximation) we begin by expanding and rewriting the objective (11.22) in matrix terms:

$$\begin{aligned}
\overline{\text{PBE}}(\mathbf{w}) &= \left\|\Pi\bar{\delta}_{\mathbf{w}}\right\|_{\mu}^2 \\
&= (\Pi\bar{\delta}_{\mathbf{w}})^\top \mathbf{D}\Pi\bar{\delta}_{\mathbf{w}} && \text{(from (11.14))} \\
&= \bar{\delta}_{\mathbf{w}}^\top \Pi^\top \mathbf{D}\Pi\bar{\delta}_{\mathbf{w}} \\
&= \bar{\delta}_{\mathbf{w}}^\top \mathbf{D}\mathbf{X}\big(\mathbf{X}^\top\mathbf{D}\mathbf{X}\big)^{-1}\mathbf{X}^\top\mathbf{D}\bar{\delta}_{\mathbf{w}} && (11.25)
\end{aligned}$$

(using (11.13) and the identity $\Pi^\top\mathbf{D}\Pi = \mathbf{D}\mathbf{X}\left(\mathbf{X}^\top\mathbf{D}\mathbf{X}\right)^{-1}\mathbf{X}^\top\mathbf{D}$)

$$= \big(\mathbf{X}^\top\mathbf{D}\bar{\delta}_{\mathbf{w}}\big)^\top\big(\mathbf{X}^\top\mathbf{D}\mathbf{X}\big)^{-1}\big(\mathbf{X}^\top\mathbf{D}\bar{\delta}_{\mathbf{w}}\big). \tag{11.26}$$

The gradient with respect to $\mathbf{w}$ is

$$\nabla\overline{\text{PBE}}(\mathbf{w}) = 2\nabla\big[\mathbf{X}^\top\mathbf{D}\bar{\delta}_{\mathbf{w}}\big]^\top\big(\mathbf{X}^\top\mathbf{D}\mathbf{X}\big)^{-1}\big(\mathbf{X}^\top\mathbf{D}\bar{\delta}_{\mathbf{w}}\big).$$

To turn this into an SGD method, we have to sample something on every time step that has this quantity as its expected value. Let us take μ to be the distribution of states visited under the behavior policy. All three of the factors above can then be written in terms of expectations under this distribution. For example, the last factor can be written

$$\mathbf{X}^\top\mathbf{D}\bar{\delta}_{\mathbf{w}} = \sum_s \mu(s)\mathbf{x}(s)\bar{\delta}_{\mathbf{w}}(s) = \mathbb{E}[\rho_t\delta_t\mathbf{x}_t]\,,$$

which is just the expectation of the semi-gradient TD(0) update (11.2). The first factor is the transpose of the gradient of this update:

$$\begin{aligned}
\nabla\mathbb{E}[\rho_t\delta_t\mathbf{x}_t]^\top &= \mathbb{E}\big[\rho_t\nabla\delta_t^\top\mathbf{x}_t^\top\big] \\
&= \mathbb{E}\big[\rho_t\nabla(R_{t+1} + \gamma\mathbf{w}^\top\mathbf{x}_{t+1} - \mathbf{w}^\top\mathbf{x}_t)^\top\mathbf{x}_t^\top\big] && \text{(using episodic } \delta_t\text{)} \\
&= \mathbb{E}\big[\rho_t(\gamma\mathbf{x}_{t+1} - \mathbf{x}_t)\mathbf{x}_t^\top\big]\,.
\end{aligned}$$

Finally, the middle factor is the inverse of the expected outer-product matrix of the feature vectors:

$$\mathbf{X}^\top\mathbf{D}\mathbf{X} = \sum_s \mu(s)\mathbf{x}_s\mathbf{x}_s^\top = \mathbb{E}\big[\mathbf{x}_t\mathbf{x}_t^\top\big]\,.$$

Substituting these expectations for the three factors in our expression for the gradient of the $\overline{\text{PBE}}$, we get

$$\nabla\overline{\text{PBE}}(\mathbf{w}) = 2\mathbb{E}\big[\rho_t(\gamma\mathbf{x}_{t+1} - \mathbf{x}_t)\mathbf{x}_t^\top\big]\,\mathbb{E}\big[\mathbf{x}_t\mathbf{x}_t^\top\big]^{-1}\,\mathbb{E}[\rho_t\delta_t\mathbf{x}_t]\,. \tag{11.27}$$

It might not be obvious that we have made any progress by writing the gradient in this form. It is a product of three expressions and the first and last are not independent. They both depend on the next

feature vector $\mathbf{x}_{t+1}$; we cannot simply sample both of these expectations and then multiply the samples. This would give us a biased estmate of the gradient just as in the naive residual-gradient algorithm.

Another idea would be to estimate the three expectations separately and then combine them to produce an unbiased estimate of the gradient. This would work, but would require a lot of computational resources, particularly to store the first two expectations, which are $d \times d$ matrices, and to compute the inverse of the second. This idea can be improved. If two of the three expectations are estimated and stored, then the third could be sampled and used in conjunction with the two stored quantities. For example, you could store estimates of the second two quantities (using the increment inverse-updating techniques in Section 9.7) and then sample the first expression. Unfortunately, the overall algorithm would still be of quadratic complexity (of order $O(d^2)$).

The idea of storing some estimates separately and then combining them with samples is a good one and is also used in gradient-TD methods. Gradient-TD methods estimate and store *the product* of the second two factors in (11.27). These factors are a $d \times d$ matrix and a d-vector, so their product is just a d-vector, like $\mathbf{w}$ itself. We denote this second learned vector as $\mathbf{v}$:

$$\mathbf{v} \approx \mathbb{E}\big[\mathbf{x}_t \mathbf{x}_t^\top\big]^{-1} \mathbb{E}[\rho_t \delta_t \mathbf{x}_t]\,. \tag{11.28}$$

This form is familiar to students of linear supervised learning. It is the solution to a linear least-squares problem that tries to approximate $\rho_t \delta_t$ from the features. The standard SGD method for incrementally finding the vector $\mathbf{v}$ that minimizes the expected squared error $\big(\mathbf{v}^\top \mathbf{x}_t - \rho_t \delta_t\big)^2$ is known as the Least Mean Square (LMS) rule:

$$\mathbf{v}_{t+1} = \mathbf{v}_t + \beta \rho_t \left(\delta_t - \mathbf{v}_t^\top \mathbf{x}_t\right) \mathbf{x}_t,$$

where $\beta > 0$ is another step-size parameter. We can use this method to effectively achieve (11.28) with $O(d)$ storage and per-step computation.

Given a stored estimate $\mathbf{v}_t$ approximating (11.28), we can update our main parameter vector $\mathbf{w}_t$ using SGD methods based on (11.27). The simplest such rule is

$$\begin{aligned}
\mathbf{w}_{t+1} &= \mathbf{w}_t - \frac{1}{2}\alpha \nabla \overline{\text{PBE}}(\mathbf{w}_t) && \text{(the general SGD rule)} \\
&= \mathbf{w}_t - \frac{1}{2}\alpha 2 \mathbb{E}\big[\rho_t(\gamma \mathbf{x}_{t+1} - \mathbf{x}_t)\mathbf{x}_t^\top\big]\, \mathbb{E}\big[\mathbf{x}_t \mathbf{x}_t^\top\big]^{-1} \mathbb{E}[\rho_t \delta_t \mathbf{x}_t] && \text{(from (11.27))} \\
&= \mathbf{w}_t + \alpha \mathbb{E}\big[\rho_t(\mathbf{x}_t - \gamma \mathbf{x}_{t+1})\mathbf{x}_t^\top\big]\, \mathbb{E}\big[\mathbf{x}_t \mathbf{x}_t^\top\big]^{-1} \mathbb{E}[\rho_t \delta_t \mathbf{x}_t] && (11.29) \\
&= \mathbf{w}_t + \alpha \mathbb{E}\big[\rho_t(\mathbf{x}_t - \gamma \mathbf{x}_{t+1})\mathbf{x}_t^\top\big]\, \mathbf{v}_t && \text{(based on (11.28))} \\
&= \mathbf{w}_t + \alpha \rho_t \left(\mathbf{x}_t - \gamma \mathbf{x}_{t+1}\right) \mathbf{x}_t^\top \mathbf{v}_t. && \text{(sampling)}
\end{aligned}$$

This algorithm is called *GTD2*. Note that if the final inner product $(\mathbf{x}_t^\top \mathbf{v}_t)$ is done first, then the entire algorithm is of $O(d)$ complexity.

A slightly better algorithm can be derived by doing a few more analytic steps before substituting in $\mathbf{v}_t$. Continuing from (11.29):

$$\begin{aligned}
\mathbf{w}_{t+1} &= \mathbf{w}_t + \alpha \mathbb{E}\big[\rho_t(\mathbf{x}_t - \gamma \mathbf{x}_{t+1})\mathbf{x}_t^\top\big]\, \mathbb{E}\big[\mathbf{x}_t \mathbf{x}_t^\top\big]^{-1} \mathbb{E}[\rho_t \delta_t \mathbf{x}_t] \\
&= \mathbf{w}_t + \alpha \left(\mathbb{E}\big[\rho_t \mathbf{x}_t \mathbf{x}_t^\top\big] - \gamma \mathbb{E}\big[\rho_t \mathbf{x}_{t+1} \mathbf{x}_t^\top\big]\right) \mathbb{E}\big[\mathbf{x}_t \mathbf{x}_t^\top\big]^{-1} \mathbb{E}[\rho_t \delta_t \mathbf{x}_t] \\
&= \mathbf{w}_t + \alpha \left(\mathbb{E}[\mathbf{x}_t \rho_t \delta_t] - \gamma \mathbb{E}\big[\rho_t \mathbf{x}_{t+1} \mathbf{x}_t^\top\big]\, \mathbb{E}\big[\mathbf{x}_t \mathbf{x}_t^\top\big]^{-1} \mathbb{E}[\rho_t \delta_t \mathbf{x}_t]\right) \\
&= \mathbf{w}_t + \alpha \left(\mathbb{E}[\mathbf{x}_t \rho_t \delta_t] - \gamma \mathbb{E}\big[\rho_t \mathbf{x}_{t+1} \mathbf{x}_t^\top\big]\, \mathbf{v}_t\right) \\
&= \mathbf{w}_t + \alpha \rho_t \left(\delta_t \mathbf{x}_t - \gamma \mathbf{x}_{t+1} \mathbf{x}_t^\top \mathbf{v}_t\right), && \text{(sampling)}
\end{aligned}$$

which again is $O(d)$ if the final product $(\mathbf{x}_t^\top \mathbf{v}_t)$ is done first. This algorithm is known as either *TD(0) with gradient correction (TDC)* or, alternatively, as *GTD(0)*.

Figure 11.6 shows a sample and the expected behavior of TDC on Baird's counterexample. As intended, the $\overline{\text{PBE}}$ falls to zero, but note that the individual components of the parameter vector do not approach zero. In fact, these values are still far from an optimal solution, $\hat{v}(s) = 0$, for all s, for which $\mathbf{w}$ would have to be proportional to $(1, 1, 1, 1, 1, 1, 4, -2)^\top$. After 1000 iterations we are still far from an optimal solution, as we can see from the $\overline{\text{VE}}$, which remains almost 2. The system is actually converging to an optimal solution, but progress is extremely slow because the $\overline{\text{PBE}}$ is already so close to zero.

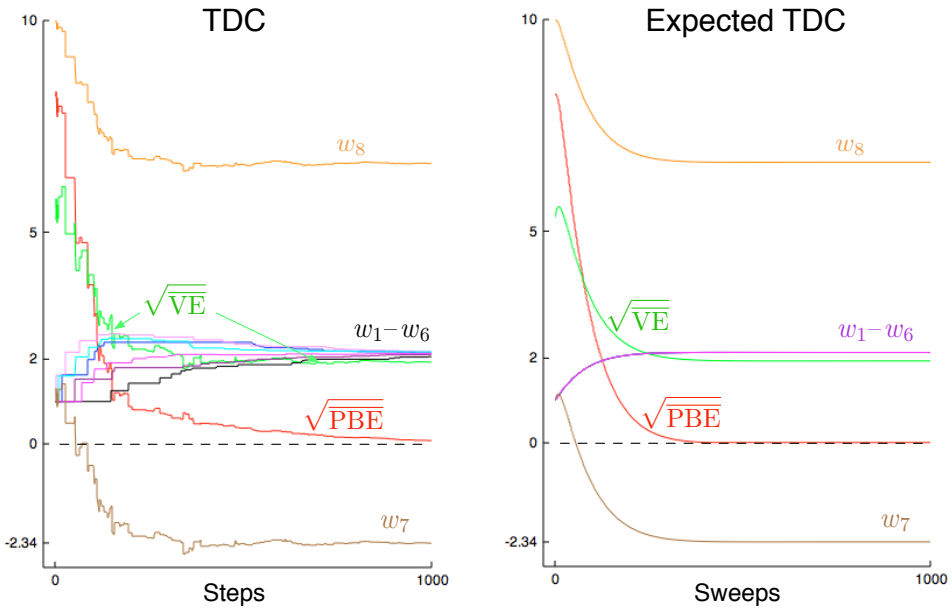


Figure 11.6: The behavior of the TDC algorithm on Baird's counterexample. On the left is shown a typical single run, and on the right is shown the expected behavior of this algorithm if the updates are done synchronously (analogous to (11.9), except for the two TDC parameter vectors). The step sizes were $\alpha = 0.005$ and $\beta = 0.05$.

GTD2 and TDC both involve two learning processes, a primary one for $\mathbf{w}$ and a secondary one for $\mathbf{v}$. The logic of the primary learning process relies on the secondary learning process having finished, at least approximately, whereas the secondary learning process proceeds without being influenced by the first. We call this sort of asymmetrical dependence a *cascade*. In cascades we often assume that the secondary learning process is proceeding faster and thus is always at its asymptotic value, ready and accurate to assist the primary learning process. The convergence proofs for these methods often make this assumption explicitly. These are called *two-time-scale* proofs. The fast time scale is that of the secondary learning process, and the slower time scale is that of the primary learning process. If α is the step size of the primary learning process, and β is the step size of the secondary learning process, then these convergence proofs will typically require that in the limit $\beta \to 0$ and $\frac{\alpha}{\beta} \to 0$.

Gradient-TD methods are currently the most well understood and widely used stable off-policy methods. There are extensions to action values and control (GQ, Maei et al., 2010), to eligibility traces (GTD(λ) and GQ(λ), Maei, 2011; Maei and Sutton, 2010), and to nonlinear function approximation (Maei et al., 2009). There have also been proposed hybrid algorithms midway between semi-gradient TD and gradient TD. The Hybrid TD (HTD, Hackman, 2012; White and White, 2016) algorithm behaves like GTD in states where the target and behavior policies are very different, and behaves like semi-gradient TD in states where the target and behavior policies are the same. Finally, the gradient-

TD idea has been combined with the ideas of proximal methods and control variates to produce more efficient methods (Mahadevan et al., 2014).

11.8 Emphatic-TD Methods

We turn now to the second major strategy that has been extensively explored for obtaining a cheap and efficient off-policy learning method with function approximation. Recall that linear semi-gradient TD methods are efficient and stable when trained under the on-policy distribution, and that we showed in Section 9.4 that this has to do with the positive definiteness of the matrix $\mathbf{A}$ (9.11) and the match between the on-policy state distribution μ_π and the state-transition probabilities $p(s|s,a)$ under the target policy. In off-policy learning, we reweight the state transitions using importance weighting so that they become appropriate for learning about the target policy, but the state distribution is still that of the behavior policy. There is a mismatch. A natural idea is to somehow reweight the states, emphasizing some and de-emphasizing others, so as to return the distribution of updates to the on-policy distribution. There would then be a match, and stability and convergence would follow from existing results. This is the idea of Emphatic-TD methods, first introduced for on-policy training in Section 9.10.

Actually, the notion of "the on-policy distribution" is not quite right, as there are many on-policy distributions, and any one of these is sufficient to guarantee stability. Consider an undiscounted episodic problem. The way episodes terminate is fully determined by the transition probabilities, but there may be several different ways the episodes might begin. However the episodes start, if all state transitions are due to the target policy, then the state distribution that results is an on-policy distribution. You might start close to the terminal state and visit only a few states with high probability before ending the episode. Or you might start far away and pass through many states before terminating. Both are on-policy distributions, and training on both with a linear semi-gradient method would be guaranteed to be stable. However the process starts, an on-policy distribution results as long as all states encountered are updated up until termination.

If there is discounting, it can be treated as partial or probabilistic termination for these purposes. If $\gamma = 0.9$, then we can consider that with probability 0.1 the process terminates on every time step and then immediately restarts in the state that is transitioned to. A discounted problem is one that is continually terminating and restarting with probability $1-\gamma$ on every step. This way of thinking about discounting is an example of a more general notion of *pseudo termination*—termination that does not affect the sequence of state transitions, but does affect the learning process and the quantities being learned. This kind of pseudo termination is important to off-policy learning because the restarting is optional—remember we can start any way we want to—and the termination relieves the need to keep including encountered states within the on-policy distribution. That is, if we don't consider the new states as restarts, then discounting quickly give us a limited on-policy distribution.

The one-step emphatic-TD algorithm for learning episodic state values is defined by:

$$\delta_t = R_{t+1} + \gamma \hat{v}(S_{t+1},\mathbf{w}_t) - \hat{v}(S_t,\mathbf{w}_t),$$

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha M_t \rho_t \delta_t \nabla \hat{v}(S_t,\mathbf{w}_t),$$

$$M_t = \gamma \rho_{t-1} M_{t-1} + I_t,$$

with I_t, the *interest*, being arbitrary and M_t, the *emphasis*, being initialized to $M_{t-1} = 0$. How does this algorithm perform on Baird's counterexample? Figure 11.7 shows the trajectory in expectation of the components of the parameter vector (for the case in which $I_t = 1$, for all t). There are some oscillations but eventually everything converges and the $\overline{\text{VE}}$ goes to zero. These trajectories are obtained by iteratively computing the expectation of the parameter vector trajectory without any of the variance due to sampling of transitions and rewards. We do not show the results of applying the emphatic-TD

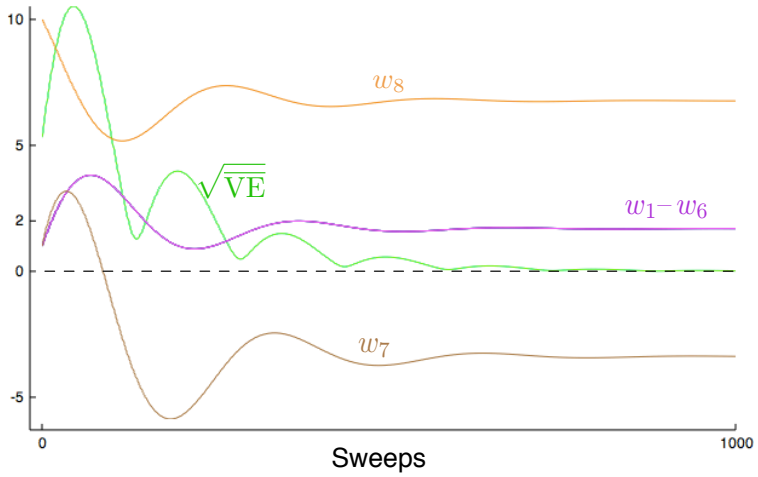


Figure 11.7: The behavior of the one-step emphatic-TD algorithm in expectation on Baird's counterexample. The step size was $\alpha = 0.03$.

algorithm directly because its variance on Baird's counterexample is so high that it is nigh impossible to get consistent results in computational experiments. The algorithm converges to the optimal solution in theory on this problem, but in practice it does not. We turn to the topic of reducing the variance of all these algorithms in the next section.

11.9 Reducing Variance

Off-policy learning is inherently of greater variance than on-policy learning. This is not surprising; if you receive data less closely related to a policy, you should expect to learn less about the policy's values. In the extreme, one may be able to learn nothing. You can't expect to learn how to drive by cooking dinner, for example. Only if the target and behavior policies are related, if they visit similar states and take similar actions, should one be able to make significant progress in off-policy training.

On the other hand, any policy has many neighbors, many similar policies with considerable overlap in states visited and actions chosen, and yet which are not identical. The raison d'être of off-policy learning is to enable generalization to this vast number of related-but-not-identical policies. The problem remains of how to make the best use of the experience. Now that we have some methods that are stable in expected value (if the step sizes are set right), attention naturally turns to reducing the variance of the estimates. There are many possible ideas, and we can just touch on of a few of them in this introductory text.

Why is controlling variance especially critical in off-policy methods based on importance sampling? As we have seen, importance sampling often involves products of policy ratios. The ratios are always one in expectation (5.11), but their actual values may be very high or as low as zero. Successive ratios are uncorrelated, so their products are also always one in expected value, but they can be of very high variance. Recall that these ratios multiply the step size in SGD methods, so high variance means taking steps that vary greatly in their sizes. This is problematic for SGD because of the occasional very large steps. They must not be so large as to take the parameter to a part of the space with a very different gradient. SGD methods rely on averaging over multiple steps to get a good sense of the gradient, and if they make large moves from single samples they become unreliable. If the step-size parameter is set small enough to prevent this, then the expected step can end up being very small, resulting in very slow learning. The notions of momentum (Derthick, 1984), of Polyak-Ruppert averaging (Polyak, 1991;

Ruppert, 1988; Polyak and Juditsky, 1992), or further extensions of these ideas may significantly help. Methods for adaptively setting separate step sizes for different components of the parameter vector are also pertinent (e.g., Jacobs, 1988; Sutton, 1992), as are the "importance weight aware" updates of Karampatziakis and Langford (2010).

In Chapter 5 we saw how weighted importance sampling is significantly better behaved, with lower variance updates, than ordinary importance sampling. However, adapting weighted importance sampling to function approximation is challenging and can probably only be done approximately with $O(d)$ complexity (Mahmood and Sutton, 2015).

The Tree Backup algorithm (Section 7.5) shows that it is possible to perform some off-policy learning without using importance sampling. This idea has been extended to the off-policy case to produce stable and more efficient methods by Munos, Stepleton, Harutyunyan, and Bellemare (2016) and by Mahmood, Yu and Sutton (2017).

Another, complementary strategy is to allow the target policy to be determined in part by the behavior policy, in such a way that it never can be so different from it to create large importance sampling ratios. For example, the target policy can be defined by reference to the behavior policy, as in the "recognizers" proposed by Precup et al. (2005).

11.10 Summary

Off-policy learning is a tempting challenge, testing our ingenuity in designing stable and efficient learning algorithms. Tabular Q-learning makes off-policy learning seem easy, and it has natural generalizations to Expected Sarsa and to the Tree Backup algorithm. But as we have seen in this chapter, the extension of these ideas to significant function approximation, even linear function approximation, involves new challenges and forces us to deepen our understanding of reinforcement learning algorithms.

Why go to such lengths? One reason to seek off-policy algorithms is to give flexibility in dealing with the tradeoff between exploration and exploitation. Another is to free behavior from learning, and avoid the tyranny of the target policy. TD learning appears to hold out the possibility of learning about multiple things in parallel, of using one stream of experience to solve many tasks simultaneously. We can certainly do this in special cases, just not in every case that we would like to or as efficiently as we would like to.

In this chapter we divided the challenge of off-policy learning into two parts. The first part, correcting the targets of learning for the behavior policy, is straightforwardly dealt with using the techniques devised earlier for the tabular case, albeit at the cost of increasing the variance of the updates and thereby slowing learning. High variance will probably always remains a challenge for off-policy learning.

The second part of the challenge of off-policy learning emerges as the instability of semi-gradient TD methods that involve bootstrapping. We seek powerful function approximation, off-policy learning, and the efficiency and flexibility of bootstrapping TD methods, but it is challenging to combine all three aspects of this *deadly triad* in one algorithm without introducing the potential for instability. There have been several attempts. The most popular has been to seek to perform true stochastic gradient descent (SGD) in the Bellman error (a.k.a. the Bellman residual). However, our analysis concludes that this is not an appealing goal in many cases, and that anyway it is impossible to achieve with a learning algorithm—the gradient of the $\overline{\text{BE}}$ is not learnable from experience that reveals only feature vectors and not underlying states. Another approach, Gradient-TD methods, performs SGD in the *projected* Bellman error. The gradient of the $\overline{\text{PBE}}$ is learnable with $O(d)$ complexity, but at the cost of a second parameter vector with a second step size. The newest family of methods, Emphatic-TD methods, refine an old idea for reweighting updates, emphasizing some and de-emphasizing others. In this way they restore the special properties that make on-policy learning stable with computationally simple semi-gradient methods.

The whole area of off-policy learning is relatively new and unsettled. Which methods are best or even adequate is not yet clear. Are the complexities of the new methods introduced at the end of this chapter really necessary? Which of them can be combined effectively with variance reductions methods? The potential for off-policy learning remains tantalizing, the best way to achieve it still a mystery.

Bibliographical and Historical Remarks

11.1 The first semi-gradient method was linear TD(λ) (Sutton, 1988). The name "semi-gradient" is more recent (Sutton, 2015a). Semi-gradient off-policy TD(0) with general importance-sampling ratio may not have been explicitly stated until Sutton, Mahmood, and White (2016), but the action-value forms were introduced by Precup, Sutton, and Singh (2000), who also did eligibility trace forms of these algorithms (see Chapter 12). Their continuing, undiscounted forms have not been significantly explored. The atomic multi-step forms given here are new.

11.2 The earliest w-to-$2w$ example was given by Tsitsiklis and Van Roy (1996), who also introduced the specific counterexample in the box on page 214. Baird's counterexample is due to Baird (1995), though the version we present here is slightly modified. Averaging methods for function approximation were developed by Gordon (1995, 1996). Other examples of instability with off-policy DP methods and more complex methods of function approximation are given by Boyan and Moore (1995). Bradtke (1993) gives an example in which Q-learning using linear function approximation in a linear quadratic regulation problem converges to a destabilizing policy.

11.3 The deadly triad was first identified by Sutton (1995b) and thoroughly analyzed by Tsitsiklis and Van Roy (1997). The name "deadly triad" is due to Sutton (2015a).

11.4 This kind of linear analysis was pioneered by Tsitsiklis and Van Roy (1996; 1997), including the dynamic programming operator. Diagrams like Figure 11.3 were introduced by Lagoudakis and Parr (2003).

11.5 The $\overline{\text{BE}}$ was first proposed as an objective function for dynamic programming by Schweitzer and Seidmann (1985). Baird (1995, 1999) extended it to TD learning based on stochastic gradient descent, and Engel, Mannor, and Meir (2003) extended it to least squares ($O(d^2)$) methods known as Gaussian Process TD learning. In the literature, $\overline{\text{BE}}$ minimization is often referred to as Bellman residual minimization.

The earliest A-split example is due to Dayan (1992). The two forms given here were introduced by Sutton et al. (2009).

11.6 The contents of this section are new to this text.

11.7 Gradient-TD methods were introduced by Sutton, Szepesvári, and Maei (2009). The methods highlighted in this section were introduced by Sutton et al. (2009) and Mahmood et al. (2014). The most sensitive empirical investigations to date of gradient-TD and related methods are given by Geist and Scherrer (2014), Dann, Neumann, and Peters (2014), and White (2015).

11.8 Emphatic-TD methods were introduced by Sutton, Mahmood, and White (2016). Full convergence proofs and other theory were later established by Yu (2015a; 2015b; Yu, Mahmood, and Sutton, 2017) and Hallak, Tamar, Munos, and Mannor (2015).

Chapter 12

Eligibility Traces

Eligibility traces are one of the basic mechanisms of reinforcement learning. For example, in the popular TD(λ) algorithm, the λ refers to the use of an eligibility trace. Almost any temporal-difference (TD) method, such as Q-learning or Sarsa, can be combined with eligibility traces to obtain a more general method that may learn more efficiently.

Eligibility traces unify and generalize TD and Monte Carlo methods. When TD methods are augmented with eligibility traces, they produce a family of methods spanning a spectrum that has Monte Carlo methods at one end ($\lambda = 1$) and one-step TD methods at the other ($\lambda = 0$). In between are intermediate methods that are often better than either extreme method. Eligibility traces also provide a way of implementing Monte Carlo methods online and on continuing problems without episodes.

Of course, we have already seen one way of unifying TD and Monte Carlo methods: the n-step TD methods of Chapter 7. What eligibility traces offer beyond these is an elegant algorithmic mechanism with significant computational advantages. The mechanism is a short-term memory vector, the *eligibility trace* $\mathbf{z}_t \in \mathbb{R}^d$, that parallels the long-term weight vector $\mathbf{w}_t \in \mathbb{R}^d$. The rough idea is that when a component of $\mathbf{w}_t$ participates in producing an estimated value, then the corresponding component of $\mathbf{z}_t$ is bumped up and then begins to fade away. Learning will then occur in that component of $\mathbf{w}_t$ if a nonzero TD error occurs before the trace falls back to zero. The trace-decay parameter $\lambda \in [0, 1]$ determines the rate at which the trace falls.

The primary computational advantage of eligibility traces over n-step methods is that only a single trace vector is required rather than a store of the last n feature vectors. Learning also occurs continually and uniformly in time rather than being delayed and then catching up at the end of the episode. In addition learning can occur and affect behavior immediately after a state is encountered rather than being delayed n steps.

Eligibility traces illustrate that a learning algorithm can sometimes be implemented in a different way to obtain computational advantages. Many algorithms are most naturally formulated and understood as an update of a state's value based on events that follow that state over multiple future time steps. For example, Monte Carlo methods (Chapter 5) update a state based on all the future rewards, and n-step TD methods (Chapter 7) update based on the next n rewards and state n steps in the future. Such formulations, based on looking forward from the updated state, are called *forward views*. Forward views are always somewhat complex to implement because the update depends on later things that are not available at the time. However, as we show in this chapter it is often possible to achieve nearly the same updates—and sometimes *exactly* the same updates—with an algorithm that uses the current TD error, looking backward to recently visited states using an eligibility trace. These alternate ways of looking at and implementing learning algorithms are called *backward views*. Backward views, transformations between forward-views and backward-views, and equivalences between them date back to the

introduction of temporal difference learning, but have become much more powerful and sophisticated since 2014. Here we present the basics of the modern view.

As usual, first we fully develop the ideas for state values and prediction, then extend them to action values and control. We develop them first for the on-policy case then extend them to off-policy learning. Our treatment pays special attention to the case of linear function approximation, for which the results with eligibility traces are stronger. All these results apply also to the tabular and state aggregation cases because these are special cases of linear function approximation.

12.1 The λ-return

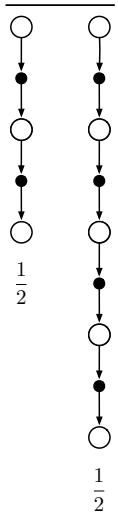
In Chapter 7 we defined an n-step return as the sum of the first n rewards plus the estimated value of the state reached in n steps, each appropriately discounted (7.1). The general form of that equation, for any parameterized function approximator, is

$$G_{t:t+n} \doteq R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n \hat{v}(S_{t+n}, \mathbf{w}_{t+n-1}), \quad 0 \leq t \leq T - n. \tag{12.1}$$

We noted in Chapter 7 that each n-step return, for $n \geq 1$, is a valid update target for a tabular learning update, just as it is for an approximate SGD learning update such as (9.7).

Now we note that a valid update can be done not just toward any n-step return, but toward any *average* of n-step returns. For example, an update can be done toward a target that is half of a two-step return and half of a four-step return: $\frac{1}{2}G_{t:t+2} + \frac{1}{2}G_{t:t+4}$. Any set of n-step returns can be averaged in this way, even an infinite set, as long as the weights on the component returns are positive and sum to 1. The composite return possesses an error reduction property similar to that of individual n-step returns (7.3) and thus can be used to construct updates with guaranteed convergence properties. Averaging produces a substantial new range of algorithms. For example, one could average one-step and infinite-step returns to obtain another way of interrelating TD and Monte Carlo methods. In principle, one could even average experience-based updates with DP updates to get a simple combination of experience-based and model-based methods (cf. Chapter 8).

An update that averages simpler component updates is called a *compound update*. The backup diagram for a compound update consists of the backup diagrams for each of the component updates with a horizontal line above them and the weighting fractions below. For example, the compound update for the case mentioned at the start of this section, mixing half of a two-step return and half of a four-step return, has the diagram shown to the right. A compound update can only be done when the longest of its component updates is complete. The update at the right, for example, could only be done at time $t + 4$ for the estimate formed at time t. In general one would like to limit the length of the longest component update because of the corresponding delay in the updates.



The TD(λ) algorithm can be understood as one particular way of averaging n-step updates. This average contains all the n-step updates, each weighted proportional to λ^{n-1}, where $\lambda \in [0, 1]$, and is normalized by a factor of $1 - \lambda$ to ensure that the weights sum to 1 (see Figure 12.1). The resulting update is toward a return, called the *λ-return*, defined in its state-based form by

$$G_t^\lambda \doteq (1 - \lambda) \sum_{n=1}^{\infty} \lambda^{n-1} G_{t:t+n}. \tag{12.2}$$

Figure 12.2 further illustrates the weighting on the sequence of n-step returns in the λ-return. The one-step return is given the largest weight, $1 - \lambda$; the two-step return is given the next largest weight, $(1 - \lambda)\lambda$; the three-step return is given the weight $(1 - \lambda)\lambda^2$; and so on. The

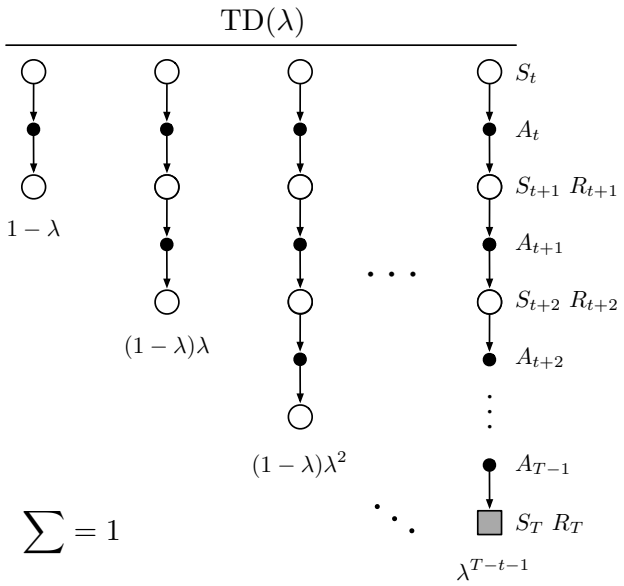


Figure 12.1: The update digram for TD(λ). If $\lambda = 0$, then the overall update reduces to its first component, the one-step TD update, whereas if $\lambda = 1$, then the overall update reduces to its last component, the Monte Carlo update.

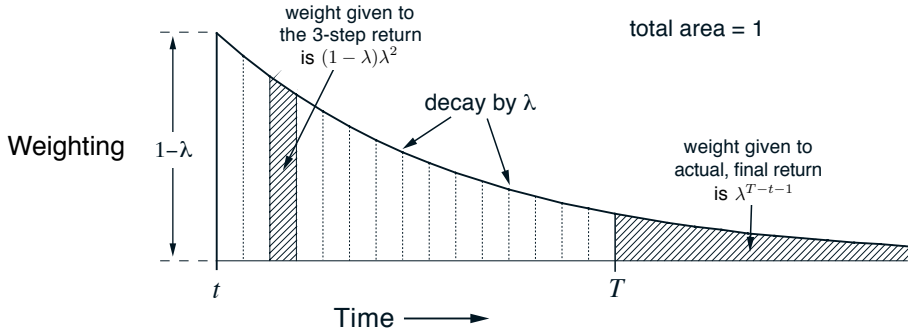


Figure 12.2: Weighting given in the λ-return to each of the n-step returns.

weight fades by λ with each additional step. After a terminal state has been reached, all subsequent n-step returns are equal to G_t. If we want, we can separate these post-termination terms from the main sum, yielding

$$G_t^\lambda \quad = \quad (1-\lambda) \sum_{n=1}^{T-t-1} \lambda^{n-1} G_{t:t+n} \quad + \quad \lambda^{T-t-1} G_t, \tag{12.3}$$

as indicated in the figures. This equation makes it clearer what happens when $\lambda = 1$. In this case the main sum goes to zero, and the remaining term reduces to the conventional return, G_t. Thus, for $\lambda = 1$, updating according to the λ-return is a Monte Carlo algorithm. On the other hand, if $\lambda = 0$, then the λ-return reduces to $G_{t:t+1}$, the one-step return. Thus, for $\lambda = 0$, updating according to the λ-return is a one-step TD method.

Exercise 12.1 Just as the return can be written recursively in terms of the first reward and itself one-step later (3.9), so can the λ-return. Derive the analogous recursive relationship from (12.2) and

(12.1). □

Exercise 12.2 The parameter λ characterizes how fast the exponential weighting in Figure 12.2 falls off, and thus how far into the future the λ-return algorithm looks in determining its update. But a rate factor such as λ is sometimes an awkward way of characterizing the speed of the decay. For some purposes it is better to specify a time constant, or half-life. What is the equation relating λ and the half-life, τ_λ, the time by which the weighting sequence will have fallen to half of its initial value? □

We are now ready to define our first learning algorithm based on the λ-return: the *off-line λ-return algorithm*. As an off-line algorithm, it makes no changes to the weight vector during the episode. Then, at the end of the episode, a whole sequence of off-line updates are made according to our usual semi-gradient rule, using the λ-return as the target:

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha\Big[G_t^\lambda - \hat{v}(S_t,\mathbf{w}_t)\Big]\nabla\hat{v}(S_t,\mathbf{w}_t), \quad t = 0,\ldots,T-1. \tag{12.4}$$

The λ-return gives us an alternative way of moving smoothly between Monte Carlo and one-step TD methods that can be compared with the n-step TD way of Chapter 7. There we assessed effectiveness on a 19-state random walk task (Example 7.1). Figure 12.3 shows the performance of the off-line λ-return algorithm on this task alongside that of the n-step methods (repeated from Figure 7.2). The experiment was just as described earlier except that for the λ-return algorithm we varied λ instead of n. The performance measure used is the estimated root-mean-squared error between the correct and estimated values of each state measured at the end of the episode, averaged over the first 10 episodes and the 19 states. Note that overall performance of the off-line λ-return algorithms is comparable to that of the n-step algorithms. In both cases we get best performance with an intermediate value of the bootstrapping parameter, n for n-step methods and λ for the offline λ-return algorithm.

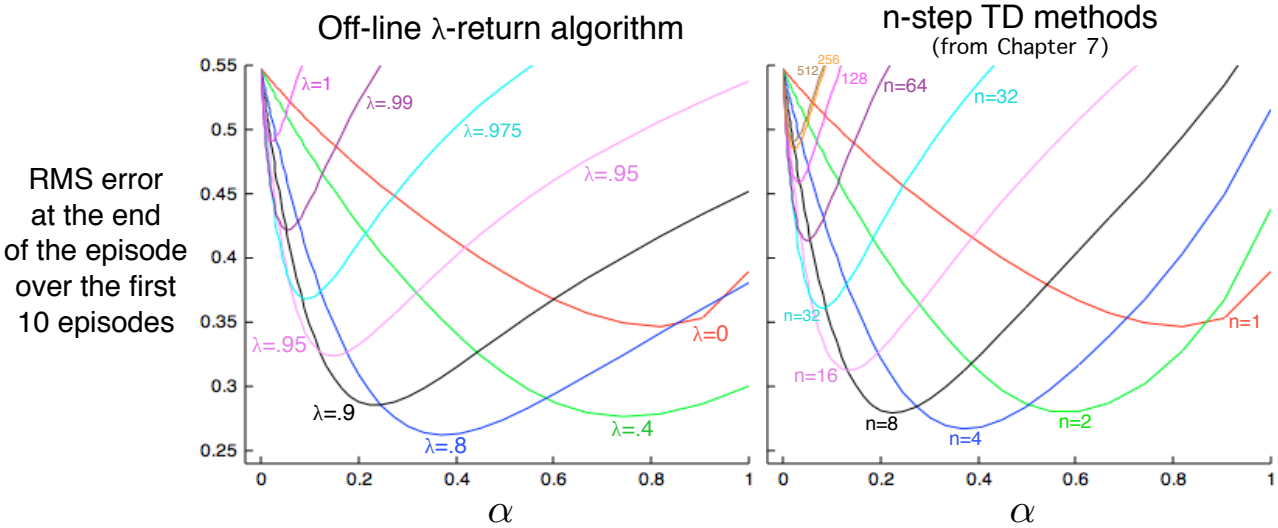


Figure 12.3: 19-state Random walk results (Example 7.1): Performance of the offline λ-return algorithm alongside that of the n-step TD methods. In both case, intermediate values of the bootstrapping parameter (λ or n) performed best. The results with the off-line λ-return algorithm are slightly better at the best values of α and λ, and at high α.

The approach that we have been taking so far is what we call the theoretical, or *forward*, view of a learning algorithm. For each state visited, we look forward in time to all the future rewards and decide how best to combine them. We might imagine ourselves riding the stream of states, looking forward from each state to determine its update, as suggested by Figure 12.4. After looking forward from and updating one state, we move on to the next and never have to work with the preceding state again.

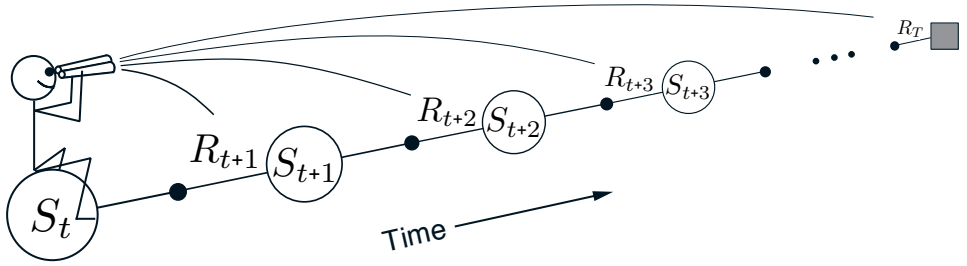


Figure 12.4: The forward view. We decide how to update each state by looking forward to future rewards and states.

Future states, on the other hand, are viewed and processed repeatedly, once from each vantage point preceding them.

12.2 TD(λ)

TD(λ) is one of the oldest and most widely used algorithms in reinforcement learning. It was the first algorithm for which a formal relationship was shown between a more theoretical forward view and a more computationally congenial backward view using eligibility traces. Here we will show empirically that it approximates the off-line λ-return algorithm presented in the previous section.

TD(λ) improves over the off-line λ-return algorithm in three ways. First it updates the weight vector on every step of an episode rather than only at the end, and thus its estimates may be better sooner. Second, its computations are equally distributed in time rather that all at the end of the episode. And third, it can be applied to continuing problems rather than just episodic problems. In this section we present the semi-gradient version of TD(λ) with function approximation.

With function approximation, the eligibility trace is a vector $\mathbf{z}_t \in \mathbb{R}^d$ with the same number of components as the weight vector $\mathbf{w}_t$. Whereas the weight vector is a long-term memory, accumulating over the lifetime of the system, the eligibility trace is a short-term memory, typically lasting less time than the length of an episode. Eligibility traces assist in the learning process; their only consequence is that they affect the weight vector, and then the weight vector determines the estimated value.

In TD(λ), the eligibility trace vector is initialized to zero at the beginning of the episode, is incremented on each time step by the value gradient, and then fades away by $\gamma\lambda$:

$$\begin{aligned} &\mathbf{z}_{-1} \doteq \mathbf{0}, \\ &\mathbf{z}_t \doteq \gamma\lambda\mathbf{z}_{t-1} + \nabla\hat{v}(S_t,\mathbf{w}_t), \quad 0 \leq t \leq T, \end{aligned} \tag{12.5}$$

where γ is the discount rate and λ is the parameter introduced in the previous section. The eligibility trace keeps track of which components of the weight vector have contributed, positively or negatively, to recent state valuations, where "recent" is defined in terms $\gamma\lambda$. The trace is said to indicate the eligibility of each component of the weight vector for undergoing learning changes should a reinforcing event occur. The reinforcing events we are concerned with are the moment-by-moment one-step TD errors. The TD error for state-value prediction is

$$\delta_t \doteq R_{t+1} + \gamma\hat{v}(S_{t+1},\mathbf{w}_t) - \hat{v}(S_t,\mathbf{w}_t). \tag{12.6}$$

In TD(λ), the weight vector is updated on each step proportional to the scalar TD error and the vector eligibility trace:

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha\delta_t\mathbf{z}_t, \tag{12.7}$$

Complete pseudocode for TD(λ) is given in the box, and a picture of its operation is suggested by Figure 12.5.

Semi-gradient TD(λ) for estimating $\hat{v} \approx v_\pi$

Input: the policy π to be evaluated
Input: a differentiable function $\hat{v} : \mathcal{S}^+ \times \mathbb{R}^d \to \mathbb{R}$ such that $\hat{v}(\text{terminal},\cdot) = 0$

Initialize value-function weights $\mathbf{w}$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)
Repeat (for each episode):
  Initialize S
  $\mathbf{z} \leftarrow \mathbf{0}$ (a d-dimensional vector)
  Repeat (for each step of episode):
  . Choose $A \sim \pi(\cdot|S)$
  . Take action A, observe R, S'
  . $\mathbf{z} \leftarrow \gamma\lambda\mathbf{z} + \nabla\hat{v}(S,\mathbf{w})$
  . $\delta \leftarrow R + \gamma\hat{v}(S',\mathbf{w}) - \hat{v}(S,\mathbf{w})$
  . $\mathbf{w} \leftarrow \mathbf{w} + \alpha\delta\mathbf{z}$
  . $S \leftarrow S'$
  until S' is terminal

TD(λ) is oriented backward in time. At each moment we look at the current TD error and assign it backward to each prior state according to how much that state contributed to the current eligibility trace at that time. We might imagine ourselves riding along the stream of states, computing TD errors, and shouting them back to the previously visited states, as suggested by Figure 12.5. Where the TD error and traces come together, we get the update given by (12.7).

To better understand the backward view, consider what happens at various values of λ. If $\lambda = 0$, then by (12.5) the trace at t is exactly the value gradient corresponding to S_t. Thus the TD(λ) update (12.7) reduces to the one-step semi-gradient TD update treated in Chapter 9 (and, in the tabular case, to the simple TD rule (6.2)). This is why that algorithm was called TD(0). In terms of Figure 12.5, TD(0) is the case in which only the one state preceding the current one is changed by the TD error. For larger values of λ, but still $\lambda < 1$, more of the preceding states are changed, but each more temporally distant state is changed less because the corresponding eligibility trace is smaller, as suggested by the figure. We say that the earlier states are given less *credit* for the TD error.

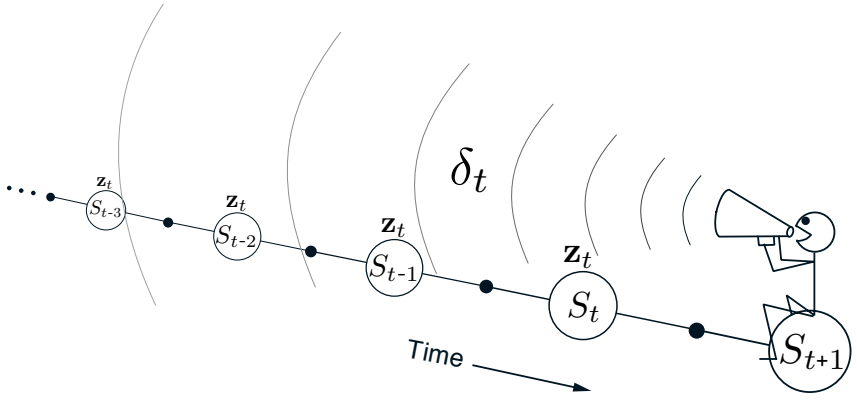


Figure 12.5: The backward or mechanistic view. Each update depends on the current TD error combined with the current eligibility traces of past events.

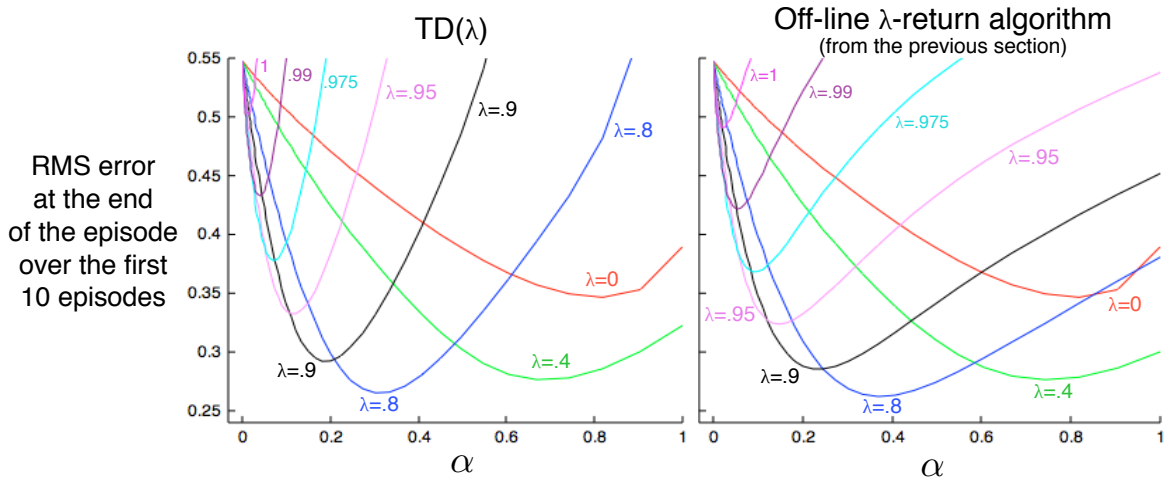


Figure 12.6: 19-state Random walk results (Example 7.1): Performance of TD(λ) alongside that of the off-line λ-return algorithm. The two algorithms performed virtually identically at low (less than optimal) α values, but TD(λ) was worse at high α values.

If $\lambda = 1$, then the credit given to earlier states falls only by γ per step. This turns out to be just the right thing to do to achieve Monte Carlo behavior. For example, remember that the TD error, δ_t, includes an undiscounted term of R_{t+1}. In passing this back k steps it needs to be discounted, like any reward in a return, by γ^k, which is just what the falling eligibility trace achieves. If $\lambda = 1$ and $\gamma = 1$, then the eligibility traces do not decay at all with time. In this case the method behaves like a Monte Carlo method for an undiscounted, episodic task. If $\lambda = 1$, the algorithm is also known as TD(1).

TD(1) is a way of implementing Monte Carlo algorithms that is more general than those presented earlier and that significantly increases their range of applicability. Whereas the earlier Monte Carlo methods were limited to episodic tasks, TD(1) can be applied to discounted continuing tasks as well. Moreover, TD(1) can be performed incrementally and on-line. One disadvantage of Monte Carlo methods is that they learn nothing from an episode until it is over. For example, if a Monte Carlo control method takes an action that produces a very poor reward but does not end the episode, then the agent's tendency to repeat the action will be undiminished during the episode. On-line TD(1), on the other hand, learns in an n-step TD way from the incomplete ongoing episode, where the n steps are all the way up to the current step. If something unusually good or bad happens during an episode, control methods based on TD(1) can learn immediately and alter their behavior on that same episode.

It is revealing to revisit the 19-state random walk example (Example 7.1) to see how well TD(λ) does in approximating the off-line λ-return algorithm. The results for both algorithms are shown in Figure 12.6. For each λ value, if α is selected optimally for it (or smaller), then the two algorithms perform virtually identically. If α is chosen larger than is optimal, however, then the λ-return algorithm is only a little worse whereas TD(λ) is much worse and may even be unstable. This is not catastrophic for TD(λ) on this problem, as these higher parameter values are not what one would want to use anyway, but for other problems it can be a significant weakness.

Linear TD(λ) has been proved to converge in the on-policy case if the step-size parameter is reduced over time according to the usual conditions (2.7). Just as discussed in Section 9.4, convergence is not to the minimum-error weight vector, but to a nearby weight vector that depends on λ. The bound on solution quality presented in that section (9.14) can now be generalized to apply to any λ. For the continuing discounted case,

$$\overline{\text{VE}}(\mathbf{w}_\infty) \quad \leq \quad \frac{1-\gamma\lambda}{1-\gamma} \min_{\mathbf{w}} \overline{\text{VE}}(\mathbf{w}). \tag{12.8}$$

That is, the asymptotic error is no more than $\frac{1-\gamma\lambda}{1-\gamma}$ times the smallest possible error. As λ approaches 1, the bound approaches the minimum error (and it is loosest at $\lambda=0$). In practice, however, $\lambda=1$ is often the poorest choice, as will be illustrated later in Figure 12.14.

Exercise 12.3 Some insight into how TD(λ) can closely approximate the off-line λ-return algorithm can be gained by seeing that the latter's error term (from (12.4)) can be written as the sum of TD errors (12.6) for a single fixed $\mathbf{w}$. Show this, following the pattern of (6.6), and using the recursive relationship for the λ-return you obtained in Exercise 12.1. □

*__Exercise 12.4__ Although online TD(λ) is not equivalent to the λ-return algorithm, perhaps there's a slightly different online TD method that would maintain equivalence. One idea is to define the TD error instead as $\mu'_t \doteq R_{t+1}+\gamma V_t(S_{t+1})-V_{t-1}(S_t)$. Show that in this case the modified TD(λ) algorithm would then achieve exactly

$$\Delta V_t(S_t) = \alpha\Big[G^\lambda_t - V_{t-1}(S_t)\Big],$$

even in the case of on-line updating with large α. In what ways might this modified TD(λ) be better or worse than the conventional one described in the text? Describe an experiment to assess the relative merits of the two algorithms. □

12.3 n-step Truncated λ-return Methods

The off-line λ-return algorithm is an important ideal, but it's of limited utility because it uses the λ-return (12.2), which is not known until the end of the episode. In the continuing case, the λ-return is technically never known, as it depends on n-step returns for arbitrarily large n, and thus on rewards arbitrarily far in the future. However, the dependence gets weaker for long-delayed rewards, falling by $\gamma\lambda$ for each step of delay. A natural approximation then would be to truncate the sequence after some number of steps. Our existing notion of n-step returns provides a natural way to do this in which the missing rewards are replaced with estimated values.

In general, we define the *truncated λ-return* for time t, given data only up to some later horizon, h, as

$$G^\lambda_{t:h} \;\doteq\; (1-\lambda)\sum_{n=1}^{h-t-1}\lambda^{n-1}G_{t:t+n} \;+\; \lambda^{h-t-1}G_{t:h}, \qquad 0\le t<h\le T. \tag{12.9}$$

If you compare this equation with the λ-return (12.3), it is clear that the horizon h is playing the same role as was previously played by T, the time of termination. Whereas in the λ-return there is a residual weighting given to the true return, here it is given to the longest available n-step return, the $(h-t)$-step return (Figure 12.2).

The truncated λ-return immediately gives rise to a family of n-step λ-return algorithms similar to the n-step methods of Chapter 7. In all these algorithms, updates are delayed by n steps and only take into account the first n rewards, but now all the k-step returns are included for $1\le k\le n$ (whereas the earlier n-step algorithms used only the n-step return), weighted geometrically as in Figure 12.2. In the state-value case, this family of algorithms is known as truncated TD(λ), or TTD(λ). The compound backup diagram, shown in Figure 12.7, is similar to that for TD(λ) (Figure 12.1) except that the longest component update is at most n steps rather than always going all the way to the end of the episode. TTD(λ) is defined by (cf. (9.15)):

$$\mathbf{w}_{t+n} \doteq \mathbf{w}_{t+n-1} + \alpha\left[G^\lambda_{t:t+n} - \hat{v}(S_t,\mathbf{w}_{t+n-1})\right]\nabla\hat{v}(S_t,\mathbf{w}_{t+n-1}), \qquad 0\le t<T. \tag{12.10}$$

This algorithm can be implemented efficiently so that per-step computation does not scale with n (though of course memory must). Much as in n-step TD methods, no updates are made on the first

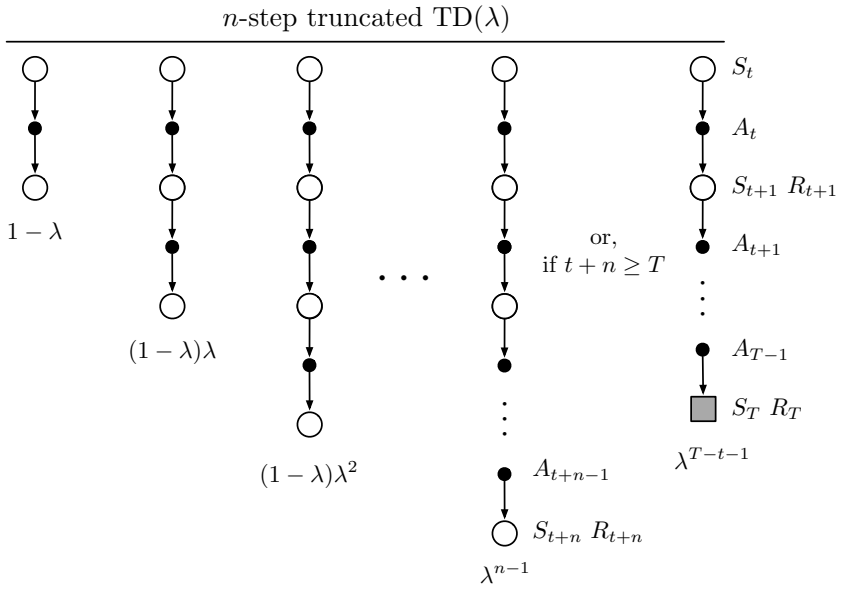


Figure 12.7: The backup diagram for truncated TD(λ).

$n-1$ time steps, and $n-1$ additional updates are made upon termination. Efficient implementation relies on the fact that the k-step λ-return can be written exactly as

$$G^{\lambda}_{t:t+k} = \hat{v}(S_t,\mathbf{w}_{t-1}) + \sum_{i=t}^{t+k-1} (\gamma\lambda)^{i-t}\delta'_i, \tag{12.11}$$

where

$$\delta'_t \doteq R_{t+1} + \gamma\hat{v}(S_{t+1},\mathbf{w}_t) - \hat{v}(S_t,\mathbf{w}_{t-1}). \tag{12.12}$$

Exercise 12.5 Several times in this book (often in exercises) we have established that returns can be written as sums of TD errors if the value function is held constant. Why is (12.11) another instance of this? Prove (12.11). □

12.4 Redoing Updates: The Online λ-return Algorithm

Choosing the truncation parameter n in Truncated TD(λ) involves a tradeoff. n should be large so that the method closely approximates the off-line λ-return algorithm, but it should also be small so that the updates can be made sooner and can influence behavior sooner. Can we get the best of both? Well, yes, in principle we can, albeit at the cost of computational complexity.

The idea is that, on each time step as you gather a new increment of data, you go back and redo all the updates since the beginning of the current episode. The new updates will be better than the ones you previously made because now they can take into account the time step's new data. That is, the updates are always towards an n-step truncated λ-return target, but they always use the latest horizon. In each pass over that episode you can use a slightly longer horizon and obtain slightly better results. Recall that the n-step truncated λ-return is defined by

$$G^{\lambda}_{t:h} \doteq (1-\lambda)\sum_{n=1}^{h-t-1} \lambda^{n-1}G_{t:t+n} + \lambda^{h-t-1}G_{t:h}. \tag{12.9}$$

Let us step through how this target could ideally be used if computational complexity was not an issue. The episode begins with an estimate at time 0 using the weights $\mathbf{w}_0$ from the end of the previous episode. Learning begins when the data horizon is extended to time step 1. The target for the estimate at step 0, given the data up to horizon 1, could only be the one-step return $G_{0:1}$, which includes R_1 and bootstraps from the estimate $\hat{v}(S_1,\mathbf{w}_0)$. Note that this is exactly what $G^{\lambda}_{0:1}$ is, with the sum in the first term of (12.9) degenerating to zero. Using this update target, we construct $\mathbf{w}_1$. Then, after advancing the data horizon to step 2, what do we do? We have new data in the form of R_2 and S_2, as well as the new $\mathbf{w}_1$, so now we can construct a better update target $G^{\lambda}_{0:2}$ for the first update from S_0 as well as a better update target $G^{\lambda}_{0:2}$ for the second update from S_1. We perform both of these updates in sequence to produce $\mathbf{w}_2$. Now we advance the horizon to step 3 and repeat, going all the way back to produce three new updates and finally $\mathbf{w}_3$, and so on.

This conceptual algorithm involves multiple passes over the episode, one at each horizon, each generating a different sequence of weight vectors. To describe it clearly we have to distinguish between the weight vectors computed at the different horizons. Let us use $\mathbf{w}^h_t$ to denote the weights used to generate the value at time t in the sequence at horizon h. The first weight vector $\mathbf{w}^h_0$ in each sequence is that inherited from the previous episode, and the last weight vector $\mathbf{w}^h_h$ in each sequence defines the ultimate weight-vector sequence of the algorithm. At the final horizon $h = T$ we obtain the final weights $\mathbf{w}^T_T$ which will be passed on to form the initial weights of the next episode. With these conventions, the three first sequences described in the previous paragraph can be given explicitly:

$$
\begin{aligned}
h = 1: \quad & \mathbf{w}^1_1 \doteq \mathbf{w}^1_0 + \alpha \left[G^{\lambda}_{0:1} - \hat{v}(S_0,\mathbf{w}^1_0)\right] \nabla \hat{v}(S_0,\mathbf{w}^1_0), \\
\\
h = 2: \quad & \mathbf{w}^2_1 \doteq \mathbf{w}^2_0 + \alpha \left[G^{\lambda}_{0:2} - \hat{v}(S_0,\mathbf{w}^2_0)\right] \nabla \hat{v}(S_0,\mathbf{w}^2_0), \\
& \mathbf{w}^2_2 \doteq \mathbf{w}^2_1 + \alpha \left[G^{\lambda}_{1:2} - \hat{v}(S_1,\mathbf{w}^2_1)\right] \nabla \hat{v}(S_1,\mathbf{w}^2_1), \\
\\
h = 3: \quad & \mathbf{w}^3_1 \doteq \mathbf{w}^3_0 + \alpha \left[G^{\lambda}_{0:3} - \hat{v}(S_0,\mathbf{w}^3_0)\right] \nabla \hat{v}(S_0,\mathbf{w}^3_0), \\
& \mathbf{w}^3_2 \doteq \mathbf{w}^3_1 + \alpha \left[G^{\lambda}_{1:3} - \hat{v}(S_1,\mathbf{w}^3_1)\right] \nabla \hat{v}(S_1,\mathbf{w}^3_1), \\
& \mathbf{w}^3_3 \doteq \mathbf{w}^3_2 + \alpha \left[G^{\lambda}_{2:3} - \hat{v}(S_2,\mathbf{w}^3_2)\right] \nabla \hat{v}(S_2,\mathbf{w}^3_2).
\end{aligned}
$$

The general form for the update is

$$
\mathbf{w}^h_{t+1} \doteq \mathbf{w}^h_t + \alpha \left[G^{\lambda}_{t:h} - \hat{v}(S_t,\mathbf{w}^h_t)\right] \nabla \hat{v}(S_t,\mathbf{w}^h_t), \quad 0 \le t < h \le T. \tag{12.13}
$$

This update, together with $\mathbf{w}_t \doteq \mathbf{w}^t_t$ defines the *online λ-return algorithm*.

The online λ-return algorithm is fully online, determining a new weight vector $\mathbf{w}_t$ at each step t during an episode, using only information available at time t. It's main drawback is that it is computationally complex, passing over the entire episode so far on every step. Note that it is strictly more complex than the off-line λ-return algorithm, which passes through all the steps at the time of termination but does not make any updates during the episode. In return, the online algorithm can be expected to perform better than the off-line one, not only during the episode when it makes an update while the off-line algorithm makes none, but also at the end of the episode because the weight vector used in bootstrapping (in $G^{\lambda}_{t:h}$) has had a greater number of informative updates. This effect can be seen if one looks carefully at Figure 12.8, which compares the two algorithms on the 19-state random walk task.

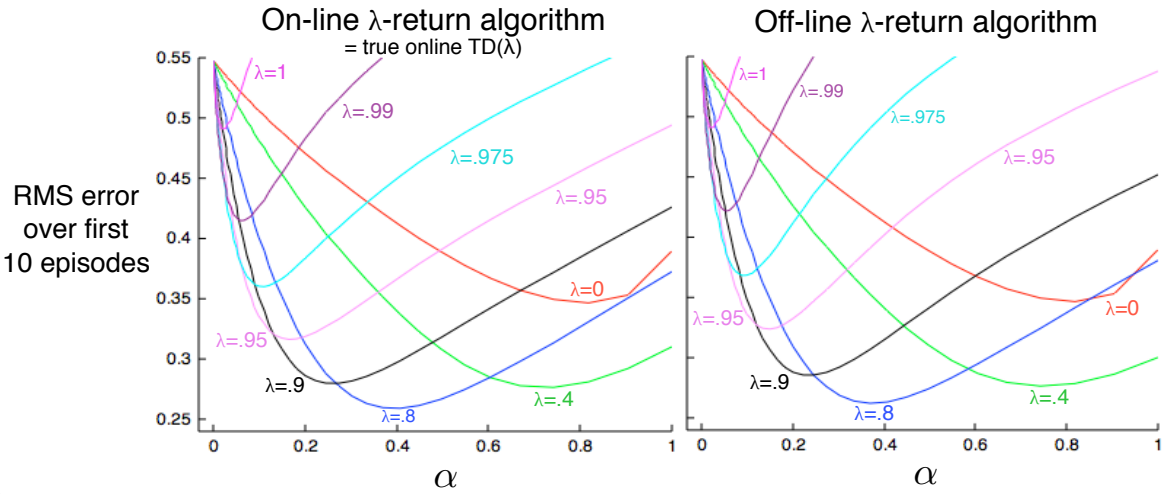


Figure 12.8: 19-state Random walk results (Example 7.1): Performance of online and off-line λ-return algorithms. The performance measure here is the $\overline{\text{VE}}$ at the end of the episode, which should be the best case for the off-line algorithm. Nevertheless, the on-line algorithm performs subtlely better. For comparison, the $\lambda = 0$ line is the same for both methods.

12.5 True Online TD(λ)

The on-line λ-return algorithm just presented is currently the best performing temporal-difference algorithm. It is an ideal which online TD(λ) only approximates. As presented, however, the on-line λ-return algorithm is very complex. Is there a way to invert this forward-view algorithm to produce an efficient backward-view algorithm using eligibility traces? It turns out that there is indeed an exact computationally congenial implementation of the on-line λ-return algorithm for the case of linear function approximation. This implementation is known as the true online TD(λ) algorithm because it is "truer" to the ideal of the online λ-return algorithm than the TD(λ) algorithm is.

The derivation of true on-line TD(λ) is a little too complex to present here (see the next section and the appendix to the paper by van Seijen et al., 2016) but its strategy is simple. The sequence of weight vectors produced by the on-line λ-return algorithm can be arranged in a triangle:

$$\begin{array}{ccccccc}
\mathbf{w}_0^0 & & & & & & \\
\mathbf{w}_0^1 & \mathbf{w}_1^1 & & & & & \\
\mathbf{w}_0^2 & \mathbf{w}_1^2 & \mathbf{w}_2^2 & & & & \\
\mathbf{w}_0^3 & \mathbf{w}_1^3 & \mathbf{w}_2^3 & \mathbf{w}_3^3 & & & \\
\vdots & \vdots & \vdots & \vdots & \ddots & & \\
\mathbf{w}_0^T & \mathbf{w}_1^T & \mathbf{w}_2^T & \mathbf{w}_3^T & \cdots & \mathbf{w}_T^T
\end{array} \tag{12.14}$$

One row of this triangle is produced on each time step. It turns out that only the weight vectors on the diagonal, the $\mathbf{w}_t^t$, are really needed. The first, $\mathbf{w}_0^0$, is the input, the last, $\mathbf{w}_T^T$, is the output, and each weight vector along the way, $\mathbf{w}_t^t$, plays a role in bootstrapping in the n-step returns of the updates. In the final algorithm the diagonal weight vectors are renamed without a superscript, $\mathbf{w}_t \doteq \mathbf{w}_t^t$. The strategy then is to find a compact, efficient way of computing each $\mathbf{w}_t^t$ from the one before. If this is done, for the linear case in which $\hat{v}(s,\mathbf{w}) = \mathbf{w}^\top \mathbf{x}(s)$, then we arrive at the true online TD(λ) algorithm:

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha\delta_t \mathbf{z}_t + \alpha\left(\mathbf{w}_t^\top \mathbf{x}_t - \mathbf{w}_{t-1}^\top \mathbf{x}_t\right)(\mathbf{z}_t - \mathbf{x}_t), \tag{12.15}$$

where we have used the shorthand $\mathbf{x}_t \doteq \mathbf{x}(S_t)$, δ_t is defined as in TD(λ) (12.6), and $\mathbf{z}_t$ is defined by

$$\mathbf{z}_t \doteq \gamma\lambda\mathbf{z}_{t-1} + \big(1 - \alpha\gamma\lambda\mathbf{z}_{t-1}^\top\mathbf{x}_t\big)\,\mathbf{x}_t. \tag{12.16}$$

This algorithm has been proven to produce exactly the same sequence of weight vectors, $\mathbf{w}_t, 0 \le t \le T$, as the on-line λ-return algorithm (van Siejen et al. 2016). Thus the results on the random walk task on the left of Figure 12.8 are also its results on that task. Now, however, the algorithm is much less expensive. The memory requirements of true online TD(λ) are identical to those of conventional TD(λ), while the per-step computation is increased by about 50% (there is one more inner product in the eligibility-trace update). Overall, the per-step computational complexity remains of $O(d)$, the same as TD(λ). Pseudocode for the complete algorithm is given in the box.

True Online TD(λ) for estimating $\mathbf{w}^\top\mathbf{x} \approx v_\pi$

Input: the policy π to be evaluated

Initialize value-function weights $\mathbf{w}$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)
Repeat (for each episode):
  Initialize state and obtain initial feature vector $\mathbf{x}$
  $\mathbf{z} \leftarrow \mathbf{0}$ (an d-dimensional vector)
  $V_{old} \leftarrow 0$ (a scalar temporary variable)
  Repeat (for each step of episode):
    Choose $A \sim \pi$
    Take action A, observe R, $\mathbf{x}'$ (feature vector of the next state)
    $V \leftarrow \mathbf{w}^\top\mathbf{x}$
    $V' \leftarrow \mathbf{w}^\top\mathbf{x}'$
    $\delta \leftarrow R + \gamma V' - V$
    $\mathbf{z} \leftarrow \gamma\lambda\mathbf{z} + \big(1 - \alpha\gamma\lambda\mathbf{z}^\top\mathbf{x}\big)\,\mathbf{x}$
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha(\delta + V - V_{old})\mathbf{z} - \alpha(V - V_{old})\mathbf{x}$
    $V_{old} \leftarrow V'$
    $\mathbf{x} \leftarrow \mathbf{x}'$
  until $\mathbf{x}' = \mathbf{0}$ (signaling arrival at a terminal state)

The eligibility trace (12.16) used in true online TD(λ) is called a *dutch trace* to distinguish it from the trace (12.5) used in TD(λ), which is called an *accumulating trace*. Earlier work often used a third kind of trace called the *replacing trace*, defined only for the tabular case or for binary feature vectors such as those produced by tile coding. The replacing trace is defined on a component-by-component basis depending on whether the component of the feature vector was 1 or 0:

$$z_{i,t} \doteq \begin{cases} 1 & \text{if } x_{i,t} = 1 \\ \gamma\lambda z_{i,t-1} & \text{otherwise.} \end{cases} \tag{12.17}$$

Nowadays, use of the replacing trace is deprecated; a dutch trace should almost always be used instead.

12.6 Dutch Traces in Monte Carlo Learning

Although eligibility traces are closely associated historically with TD learning, in fact they have nothing to do with it. In fact, eligibility traces arise even in Monte Carlo learning, as we show in this section. We show that the linear MC algorithm (Chapter 9), taken as a forward view, can be used to derive an equivalent yet computationally cheaper backward-view algorithm using dutch traces. This is the only equivalence of forward- and backward-views that we explicitly demonstrate in this book. It gives some of the flavor of the proof of equivalence of true online TD(λ) and the on-line λ-return algorithm, but is much simpler.

The linear version of the gradient Monte Carlo prediction algorithm (page 165) makes the following sequence of updates, one for each time step of the episode:

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha \left[G - \mathbf{w}_t^\top \mathbf{x}_t\right] \mathbf{x}_t, \quad 0 \leq t < T. \tag{12.18}$$

To make the example simpler, we assume here that the return G is a single reward received at the end of the episode (this is why G is not subscripted by time) and that there is no discounting. In this case the update is also known as the least mean square (LMS) rule. As a Monte Carlo algorithm, all the updates depend on the final reward/return, so none can be made until the end of the episode. The MC algorithm is an offline algorithm and we do not seek to improve this aspect of it. Rather we seek merely an implementation of this algorithm with computational advantages. We will still update the weight vector only at the end of the episode, but we will do some computation during each step of the episode and less at its end. This will give a more equal distribution of computation—$O(d)$ per step—and also remove the need to store the feature vectors at each step for use later at the end of each episode. Instead, we will introduce an additional vector memory, the eligibility trace, keeping in it a summary of all the feature vectors seen so far. This will be sufficient to efficiently recreate exactly the same overall update as the sequence of MC updates (12.18), by the end of the episode:

$$\begin{aligned}
\mathbf{w}_T &= \mathbf{w}_{T-1} + \alpha \left(G - \mathbf{w}_{T-1}^\top \mathbf{x}_{T-1}\right) \mathbf{x}_{T-1} \\
&= \mathbf{w}_{T-1} + \alpha \mathbf{x}_{T-1} \left(-\mathbf{x}_{T-1}^\top \mathbf{w}_{T-1}\right) + \alpha G \mathbf{x}_{T-1} \\
&= \left(\mathbf{I} - \alpha \mathbf{x}_{T-1} \mathbf{x}_{T-1}^\top\right) \mathbf{w}_{T-1} + \alpha G \mathbf{x}_{T-1} \\
&= \mathbf{F}_{T-1} \mathbf{w}_{T-1} + \alpha G \mathbf{x}_{T-1}
\end{aligned}$$

where $\mathbf{F}_t \doteq \mathbf{I} - \alpha \mathbf{x}_t \mathbf{x}_t^\top$ is a *forgetting*, or *fading*, matrix. Now, recursing,

$$\begin{aligned}
&= \mathbf{F}_{T-1} \left(\mathbf{F}_{T-2} \mathbf{w}_{T-2} + \alpha G \mathbf{x}_{T-2}\right) + \alpha G \mathbf{x}_{T-1} \\
&= \mathbf{F}_{T-1} \mathbf{F}_{T-2} \mathbf{w}_{T-2} + \alpha G \left(\mathbf{F}_{T-1} \mathbf{x}_{T-2} + \mathbf{x}_{T-1}\right) \\
&= \mathbf{F}_{T-1} \mathbf{F}_{T-2} \left(\mathbf{F}_{T-3} \mathbf{w}_{T-3} + \alpha G \mathbf{x}_{T-3}\right) + \alpha G \left(\mathbf{F}_{T-1} \mathbf{x}_{T-2} + \mathbf{x}_{T-1}\right) \\
&= \mathbf{F}_{T-1} \mathbf{F}_{T-2} \mathbf{F}_{T-3} \mathbf{w}_{T-3} + \alpha G \left(\mathbf{F}_{T-1} \mathbf{F}_{T-2} \mathbf{x}_{T-3} + \mathbf{F}_{T-1} \mathbf{x}_{T-2} + \mathbf{x}_{T-1}\right) \\
&\;\;\vdots \\
&= \underbrace{\mathbf{F}_{T-1} \mathbf{F}_{T-2} \cdots \mathbf{F}_0 \mathbf{w}_0}_{\mathbf{a}_{T-1}} + \underbrace{\alpha G \sum_{k=0}^{T-1} \mathbf{F}_{T-1} \mathbf{F}_{T-2} \cdots \mathbf{F}_{k+1} \mathbf{x}_k}_{\mathbf{z}_{T-1}} \\
&= \mathbf{a}_{T-1} + \alpha G \mathbf{z}_{T-1},
\end{aligned} \tag{12.19}$$

where $\mathbf{a}_{T-1}$ and $\mathbf{z}_{T-1}$ are the values at time $T-1$ of two auxilary memory vectors that can be updated incrementally without knowledge of G and with $O(d)$ complexity per time step. The $\mathbf{z}_t$ vector is in fact

a dutch-style eligibility trace. It is initialized to $\mathbf{z}_0 = \mathbf{x}_0$ and then updated according to

$$\begin{aligned}
\mathbf{z}_t &\doteq \sum_{k=0}^{t} \mathbf{F}_t \mathbf{F}_{t-1} \cdots \mathbf{F}_{k+1} \mathbf{x}_k, \qquad 1 \leq t < T \\
&= \sum_{k=0}^{t-1} \mathbf{F}_t \mathbf{F}_{t-1} \cdots \mathbf{F}_{k+1} \mathbf{x}_k + \mathbf{x}_t \\
&= \mathbf{F}_t \sum_{k=0}^{t-1} \mathbf{F}_{t-1} \mathbf{F}_{t-2} \cdots \mathbf{F}_{k+1} \mathbf{x}_k + \mathbf{x}_t \\
&= \mathbf{F}_t \mathbf{z}_{t-1} + \mathbf{x}_t \\
&= \left(\mathbf{I} - \alpha \mathbf{x}_t \mathbf{x}_t^\top\right) \mathbf{z}_{t-1} + \mathbf{x}_t \\
&= \mathbf{z}_{t-1} - \alpha \mathbf{x}_t \mathbf{x}_t^\top \mathbf{z}_{t-1} + \mathbf{x}_t \\
&= \mathbf{z}_{t-1} - \alpha \left(\mathbf{z}_{t-1}^\top \mathbf{x}_t\right) \mathbf{x}_t + \mathbf{x}_t \\
&= \mathbf{z}_{t-1} + \left(1 - \alpha \mathbf{z}_{t-1}^\top \mathbf{x}_t\right) \mathbf{x}_t,
\end{aligned}$$

which is the dutch trace for the case of $\gamma\lambda = 1$ (cf. Eq. 12.16). The $\mathbf{a}_t$ auxilary vector is initialized to $\mathbf{a}_0 = \mathbf{w}_0$ and then updated according to

$$\mathbf{a}_t \doteq \mathbf{F}_t \mathbf{F}_{t-1} \cdots \mathbf{F}_0 \mathbf{w}_0 = \mathbf{F}_t \mathbf{a}_{t-1} = \mathbf{a}_{t-1} - \alpha \mathbf{x}_t \mathbf{x}_t^\top \mathbf{a}_{t-1}, \quad 1 \leq t < T. \tag{12.20}$$

The auxiliary vectors, $\mathbf{a}_t$ and $\mathbf{z}_t$, are updated on each time step $t < T$ and then, at time T when G is observed, they are used in (12.19) to compute $\mathbf{w}_T$. In this way we achieve exactly the same final result as the MC/LMS algorithm with poor computational properties (12.18), but with an incremental algorithm whose time and memory complexity per step is $O(d)$. This is surprising and intriguing because the notion of an eligibility trace (and the dutch trace in particular) has arisen in a setting without temporal-difference (TD) learning (in contrast to Van Seijen and Sutton, 2014). It seems eligibility traces are not specific to TD learning at all; they are more fundamental than that. The need for eligibility traces seems to arise whenever one tries to learn long-term predictions in an efficient manner.

12.7 Sarsa(λ)

Very few changes in the ideas already presented in this chapter are required in order to extend eligibility-traces to action-value methods. To learn approximate action values, $\hat{q}(s, a, \mathbf{w})$, rather than approximate state values, $\hat{v}(s,\mathbf{w})$, we need to use the action-value form of the n-step return, from Chapter 10:

$$G_{t:t+n} \doteq R_{t+1} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n \hat{q}(S_{t+n}, A_{t+n}, \mathbf{w}_{t+n-1}), \tag{10.4}$$

for all n and t such that $n \geq 1$ and $0 \leq t < T - n$. Using this, we can form the action-value form of the truncated λ-return, which is otherwise identical to the state-value form (12.9). The action-value form of the off-line λ-return algorithm (12.4) simply uses $\hat{q}$ rather than $\hat{v}$:

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha \Big[G_t^\lambda - \hat{q}(S_t, A_t \mathbf{w}_t)\Big] \nabla \hat{q}(S_t, A_t, \mathbf{w}_t), \quad t = 0, \ldots, T-1, \tag{12.21}$$

where $G_t^\lambda \doteq G_{t:\infty}^\lambda$. The compound backup diagram for this forward view is shown in Figure 12.9. Notice the similarity to the diagram of the TD(λ) algorithm (Figure 12.1). The first update looks ahead one full step, to the next state–action pair, the second looks ahead two steps, to the second state–action

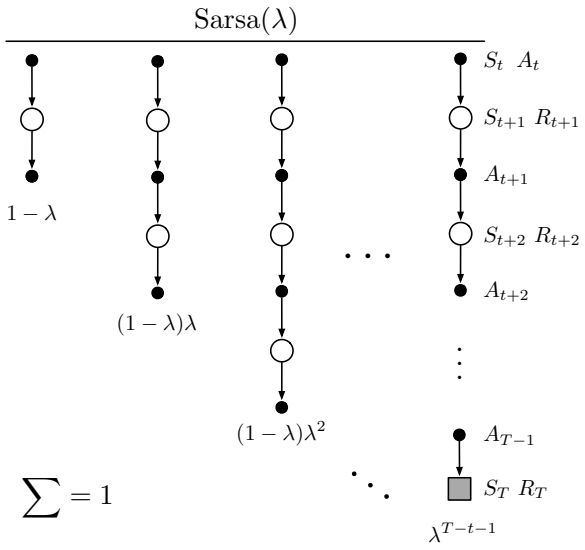


Figure 12.9: Sarsa(λ)'s backup diagram. Compare with Figure 12.1.

pair, and so on. A final update is based on the complete return. The weighting of each n-step update in the λ-return is just as in TD(λ) and the λ-return algorithm (12.3).

The temporal-difference method for action values, known as *Sarsa(λ)*, approximates this forward view. It has the same update rule as given earlier for TD(λ):

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha \delta_t \mathbf{z}_t, \tag{12.7}$$

except, naturally, using the action-value form of the TD error:

$$\delta_t \doteq R_{t+1} + \gamma \hat{q}(S_{t+1}, A_{t+1}, \mathbf{w}_t) - \hat{q}(S_t, A_t, \mathbf{w}_t), \tag{12.22}$$

and the action-value form of the eligibility trace:

$$\begin{aligned} &\mathbf{z}_{-1} \doteq \mathbf{0}, \\ &\mathbf{z}_t \doteq \gamma\lambda \mathbf{z}_{t-1} + \nabla \hat{q}(S_t, A_t, \mathbf{w}_t), \quad 0 \leq t \leq T \end{aligned} \tag{12.23}$$

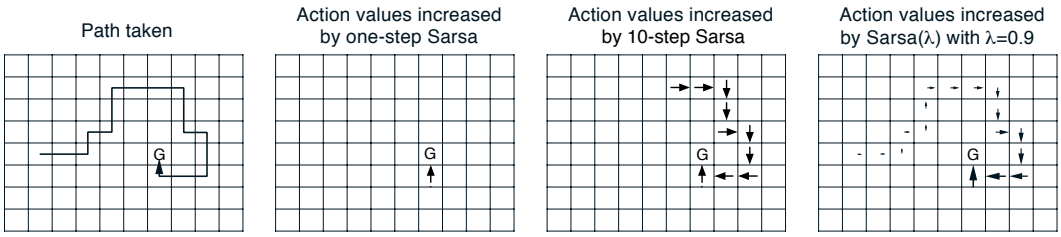
(or, alternatively, the replacing trace given by (12.17)). Complete pseudocode for Sarsa(λ) with linear function approximation, binary features, and either accumulating or replacing traces is given in the box on the next page. This pseudocode highlights a few optimizations possible in the special case of binary features (features are either active (=1) or inactive (=0).

Sarsa(λ) with binary features and linear function approximation for estimating $\hat{q} \approx q_\pi$ or $\hat{q} \approx q_*$

Input: a function $\mathcal{F}(s,a)$ returning the set of (indices of) active features for s, a
Input: a policy π to be evaluated, if any

Initialize parameter vector $\mathbf{w} = (w_1, \ldots, w_n)$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)
Loop for each episode:
  Initialize S
  Choose $A \sim \pi(\cdot|S)$ or ε-greedy according to $\hat{q}(S, \cdot, \mathbf{w})$
  $\mathbf{z} \leftarrow \mathbf{0}$
  Loop for each step of episode:
    Take action A, observe R, S'
    $\delta \leftarrow R$
    Loop for i in $\mathcal{F}(S, A)$:
      $\delta \leftarrow \delta - w_i$
      $z_i \leftarrow z_i + 1$ (accumulating traces)
      or $z_i \leftarrow 1$ (replacing traces)
    If S' is terminal then:
      $\mathbf{w} \leftarrow \mathbf{w} + \alpha\delta\mathbf{z}$
      Go to next episode
    Choose $A' \sim \pi(\cdot|S')$ or near greedily $\sim \hat{q}(S', \cdot, \mathbf{w})$
    Loop for i in $\mathcal{F}(S', A')$: $\delta \leftarrow \delta + \gamma w_i$
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha\delta\mathbf{z}$
    $\mathbf{z} \leftarrow \gamma\lambda\mathbf{z}$
    $S \leftarrow S'; A \leftarrow A'$

Example 12.1: Traces in Gridworld The use of eligibility traces can substantially increase the efficiency of control algorithms over one-step methods and even over n-step methods. The reason for this is illustrated by the gridworld example below.



The first panel shows the path taken by an agent in a single episode. The initial estimated values were zero, and all rewards were zero except for a positive reward at the goal location marked by G. The arrows in the other panels show, for various algorithms, which action-values would be increased, and by how much, upon reaching the goal. A one-step method would increment only the last action value, whereas an n-step method would equally increment the last n action's values, and an eligibility trace method would update all the action values up to the beginning of the episode to different degrees, fading with recency. The fading strategy is often the best tradeoff, strongly learning how to reach the goal from the right, yet not as strongly learning the roundabout path to the goal from the left that was taken in this episode. ■

Exercise 12.6 Modifiy the pseudocode for Sarsa(λ) to use dutch traces (12.16) without the other features of a true online algorithm. Assume linear function approximation and binary features. □

Example 12.2: Sarsa(λ) on Mountain Car Figure 12.10 (left) shows results with Sarsa(λ) on the Mountain Car task introduced in Example 10.1. The function approximation, action selection, and environmental details were exactly as in Chapter 10, and thus it is appropriate to numerically compare these results with the Chapter 10 results for n-step Sarsa (right side of the figure). The earlier results varied the update length n whereas here for Sarsa(λ) we vary the trace parameter λ, which plays a similar role. The fading-trace bootstrapping strategy of Sarsa(λ) appears to result in more efficient learning on this problem.

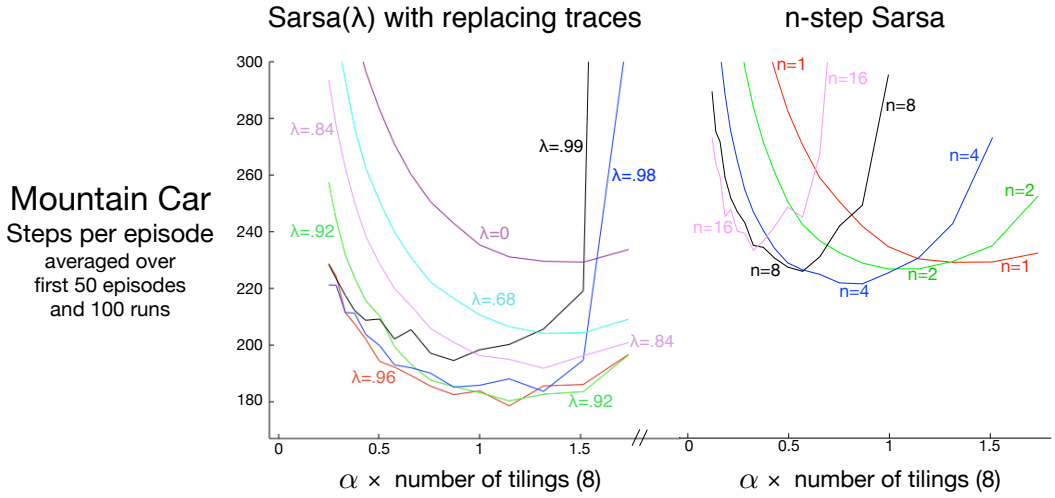


Figure 12.10: Early performance on the Mountain Car task of Sarsa(λ) with replacing traces and n-step Sarsa (copied from Figure 10.4) as a function of the step size, α. ■

There is also an action-value version of our ideal TD method, the *online* λ-return algorithm presented in Section 12.4. Everything in that section goes through without change other than to use the action-value form of the n-step return given at the beginning of this section. In the case of linear function approximation, the ideal algorithm again has an exact, efficient $O(d)$ implementation, called *True Online Sarsa(λ)*. The analyses in Sections 12.5 and 12.6 carry through without change other than to use state–action feature vectors $\mathbf{x}_t = \mathbf{x}(S_t, A_t)$ instead of state feature vectors $\mathbf{x}_t = \mathbf{x}(S_t)$. The pseudocode for this algorithm is given in the box on the next page. Figure 12.11 compares the performance of various versions of Sarsa(λ) on the Mountain Car example.

True Online Sarsa(λ) for estimating $\mathbf{w}^\top\mathbf{x} \approx q_\pi$ or q_*

Input: a feature function $\mathbf{x} : \mathcal{S}^+ \times \mathcal{A} \to \mathbb{R}^d$ s.t. $\mathbf{x}(terminal, \cdot) = \mathbf{0}$
Input: the policy π to be evaluated, if any

Initialize parameter $\mathbf{w}$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)
Loop for each episode:
  Initialize S
  Choose $A \sim \pi(\cdot|S)$ or near greedily from S using $\mathbf{w}$
  $\mathbf{x} \leftarrow \mathbf{x}(S, A)$
  $\mathbf{z} \leftarrow \mathbf{0}$
  $Q_{old} \leftarrow 0$ (a scalar temporary variable)
  Loop for each step of episode:
    Take action A, observe R, S'
    Choose $A' \sim \pi(\cdot|S')$ or near greedily from S' using $\mathbf{w}$
    $\mathbf{x}' \leftarrow \mathbf{x}(S', A')$
    $Q \leftarrow \mathbf{w}^\top\mathbf{x}$
    $Q' \leftarrow \mathbf{w}^\top\mathbf{x}'$
    $\delta \leftarrow R + \gamma Q' - Q$
    $\mathbf{z} \leftarrow \gamma\lambda\mathbf{z} + \big(1 - \alpha\gamma\lambda\mathbf{z}^\top\mathbf{x}\big)\,\mathbf{x}$
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha(\delta + Q - Q_{old})\mathbf{z} - \alpha(Q - Q_{old})\mathbf{x}$
    $Q_{old} \leftarrow Q'$
    $\mathbf{x} \leftarrow \mathbf{x}'$
    $A \leftarrow A'$
  until S' is terminal

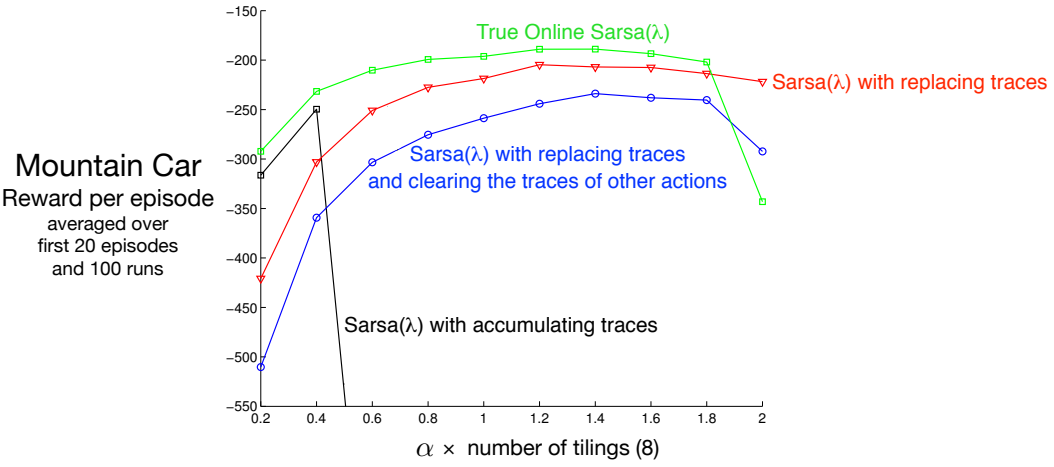


Figure 12.11: Summary comparison of Sarsa(λ) algorithms on the Mountain Car task. True Online Sarsa(λ) performed better than regular Sarsa(λ) with both accumulating and replacing traces. Also included is a version of Sarsa(λ) with replacing traces in which, on each time step, the traces for the state and the actions not selected were set to zero.

12.8 Variable λ and γ

We are starting now to reach the end of our development of fundamental TD learning algorithms. To present the final algorithms in their most general forms, it is useful to generalize the degree of bootstrapping and discounting beyond constant parameters to functions potentially dependent on the state and action. That is, each time step will have a different λ and γ, denoted λ_t and γ_t. We change notation now so that $\lambda : \mathcal{S} \times \mathcal{A} \to [0,1]$ is now a whole function from states and actions to the unit interval such that $\lambda_t \doteq \lambda(S_t, A_t)$, and similarly, $\gamma : \mathcal{S} \to [0,1]$ is a function from states to the unit interval such that $\gamma_t \doteq \gamma(S_t)$.

The latter generalization, to *state-dependent discounting*, is particularly significant because it changes the return, the fundamental random variable whose expectation we seek to estimate. Now the return is defined more generally as

$$\begin{aligned} G_t &\doteq R_{t+1} + \gamma_{t+1} G_{t+1} \\ &= R_{t+1} + \gamma_{t+1} R_{t+2} + \gamma_{t+1}\gamma_{t+2} R_{t+3} + \gamma_{t+1}\gamma_{t+2}\gamma_{t+3} R_{t+4} + \cdots \\ &= \sum_{k=t}^{\infty} R_{k+1} \prod_{i=t+1}^{k} \gamma_i, \end{aligned} \tag{12.24}$$

where, to assure the sums are finite, we require that $\prod_{k=t}^{\infty} \gamma_k = 0$ with probability one for all t. One convenient aspect of this definition is that it allows us to dispense with episodes, start and terminal states, and T as special cases and quantities. A terminal state just becomes a state at which $\gamma(s) = 0$ and which transitions to the start state. In that way (and by choosing $\gamma(\cdot)$ as a constant function) we can recover the classical episodic setting as a special case. State dependent discounting includes other prediction cases such as *soft termination*, when we seek to predict a quantity that becomes complete but does not alter the flow of the Markov process. Discounted returns themselves can be thought of as such a quantity, and state dependent discounting is a deep unification of the episodic and discounted-continuing cases. (The undiscounted-continuing case still needs some special treatment.)

The generalization to variable bootstrapping is not a change in the problem, like discounting, but a change in the solution strategy. The generalization affects the λ-returns for states and actions. The new state-based λ-return can be written recursively as

$$G_t^{\lambda s} \doteq R_{t+1} + \gamma_{t+1}\left((1-\lambda_{t+1})\hat{v}(S_{t+1},\mathbf{w}_t) + \lambda_{t+1} G_{t+1}^{\lambda s}\right), \tag{12.25}$$

where now we have added the "s" to the superscript λ to remind us that this is a return that bootstraps from state values, distinguishing it from returns that bootstrap from action values, which we present below with "a" in the superscript. This equation says that the λ-return is the first reward, undiscounted and unaffected by bootstrapping, plus possibly a second term to the extent that we are not discounting at the next state (that is, according to γ_{t+1}; recall that this is zero if the next state is terminal). To the extent that we aren't terminating at the next state, we have a second term which is itself divided into two cases depending on the degree of bootstrapping in the state. To the extent we are bootstrapping, this term is the estimated value at the state, whereas, to the extent that we not bootstrapping, the term is the λ-return for the next time step. The action-based λ-return is either the Sarsa form

$$G_t^{\lambda a} \doteq R_{t+1} + \gamma_{t+1}\Big((1-\lambda_{t+1})\hat{q}(S_{t+1}, A_{t+1}, \mathbf{w}_t) + \lambda_{t+1} G_{t+1}^{\lambda a}\Big), \tag{12.26}$$

or the Expected Sarsa form,

$$G_t^{\lambda a} \doteq R_{t+1} + \gamma_{t+1}\Big((1-\lambda_{t+1})\bar{Q}_{t+1} + \lambda_{t+1} G_{t+1}^{\lambda a}\Big), \tag{12.27}$$

where

$$\bar{Q}_t \doteq \sum_a \pi(a|S_t)\hat{q}(S_t, a, \mathbf{w}_{t-1}). \tag{12.28}$$

Exercise 12.7 Generalize the three recursive equations above to their truncated versions, defining $G^{\lambda s}_{t:h}$ and $G^{\lambda a}_{t:h}$. □

12.9 Off-policy Eligibility Traces

The final step is to incorporate importance sampling. Unlike in the case of n-step methods, for full non-truncated λ-returns one does not have a practical option in which the importance sampling is done outside the target return. Instead, we move directly to the bootstrapping generalization of per-reward importance sampling (Section 7.4). In the state case, our final definition of the λ-return generalizes (12.25), after the model of (7.10), to

$$G^{\lambda s}_t \doteq \rho_t\Big(R_{t+1} + \gamma_{t+1}\big((1-\lambda_{t+1})\hat{v}(S_{t+1},\mathbf{w}_t) + \lambda_{t+1}G^{\lambda s}_{t+1}\big)\Big) + (1-\rho_t)\hat{v}(S_t,\mathbf{w}_t) \tag{12.29}$$

where $\rho_t = \frac{\pi(A_t|S_t)}{b(A_t|S_t)}$ is the usual single-step importance sampling ratio. Much like the other returns we have seen in this book, the truncated version of this return can be approximated simply in terms of sums of the state-based TD error,

$$\delta^s_t = R_{t+1} + \gamma_{t+1}\hat{v}(S_{t+1},\mathbf{w}_t) - \hat{v}(S_t,\mathbf{w}_t), \tag{12.30}$$

as

$$G^{\lambda s}_t \approx \hat{v}(S_t,\mathbf{w}_t) + \rho_t \sum_{k=t}^{\infty} \delta^s_k \prod_{i=t+1}^{k} \gamma_i\lambda_i\rho_i \tag{12.31}$$

with the approximation becoming exact if the approximate value function does not change.

Exercise 12.8 Prove that (12.31) becomes exact if the value function does not change. To save writing, consider the case of $t = 0$, and use the notation $V_k \doteq \hat{v}(S_k,\mathbf{w})$. □

Exercise 12.9 The truncated version of the general off-policy return is denoted $G^{\lambda s}_{t:h}$. Guess the correct equation, based on (12.31). □

The above form of the λ-return (12.31) is convenient to use in a forward-view update,

$$\begin{aligned}\mathbf{w}_{t+1} &= \mathbf{w}_t + \alpha\left(G^{\lambda s}_t - \hat{v}(S_t,\mathbf{w}_t)\right)\nabla\hat{v}(S_t,\mathbf{w}_t)\\ &\approx \mathbf{w}_t + \alpha\rho_t\left(\sum_{k=t}^{\infty}\delta^s_k\prod_{i=t+1}^{k}\gamma_i\lambda_i\rho_i\right)\nabla\hat{v}(S_t,\mathbf{w}_t),\end{aligned}$$

which to the experienced eye looks like an eligibility-based TD update—the product is like an eligibility trace and it is multiplied by TD errors. But this is just one time step of a forward view. The relationship that we are looking for is that the forward-view update, summed over time, is approximately equal to a backward-view update, summed over time (this relationship is only approximate because again we ignore changes in the value function). The sum of the forward-view update over time is

$$\begin{aligned}\sum_{t=1}^{\infty}(\mathbf{w}_{t+1}-\mathbf{w}_t) &\approx \sum_{t=1}^{\infty}\sum_{k=t}^{\infty}\alpha\rho_t\delta^s_k\nabla\hat{v}(S_t,\mathbf{w}_t)\prod_{i=t+1}^{k}\gamma_i\lambda_i\rho_i\\ &= \sum_{k=1}^{\infty}\sum_{t=1}^{k}\alpha\rho_t\nabla\hat{v}(S_t,\mathbf{w}_t)\delta^s_k\prod_{i=t+1}^{k}\gamma_i\lambda_i\rho_i\\ &\qquad\qquad\text{(using the summation rule: } \textstyle\sum_{t=x}^{y}\sum_{k=t}^{y} = \sum_{k=x}^{y}\sum_{t=x}^{k}\text{)}\\ &= \sum_{k=1}^{\infty}\alpha\delta^s_k\sum_{t=1}^{k}\rho_t\nabla\hat{v}(S_t,\mathbf{w}_t)\prod_{i=t+1}^{k}\gamma_i\lambda_i\rho_i,\end{aligned}$$

which would be in the form of the sum of a backward-view TD update if the entire expression from the second sum left could be written and updated incrementally as an eligibility trace, which we now show can be done. That is, we show that if this expression was the trace at time k, then we could update it from its value at time $k-1$ by:

$$\begin{aligned}
\mathbf{z}_k &= \sum_{t=1}^{k} \rho_t \nabla \hat{v}(S_t,\mathbf{w}_t) \prod_{i=t+1}^{k} \gamma_i \lambda_i \rho_i \\
&= \sum_{t=1}^{k-1} \rho_t \nabla \hat{v}(S_t,\mathbf{w}_t) \prod_{i=t+1}^{k} \gamma_i \lambda_i \rho_i \quad + \quad \rho_k \nabla \hat{v}(S_k,\mathbf{w}_k) \\
&= \gamma_k \lambda_k \rho_k \underbrace{\sum_{t=1}^{k-1} \rho_t \nabla \hat{v}(S_t,\mathbf{w}_t) \prod_{i=t+1}^{k-1} \gamma_i \lambda_i \rho_i}_{\mathbf{z}_{k-1}} \quad + \quad \rho_k \nabla \hat{v}(S_k,\mathbf{w}_k) \\
&= \rho_k \big(\gamma_k \lambda_k \mathbf{z}_{k-1} + \nabla \hat{v}(S_k,\mathbf{w}_k)\big),
\end{aligned}$$

which, changing the index from k to t, is the general accumulating trace update for state values:

$$\mathbf{z}_t \doteq \rho_t \big(\gamma_t \lambda_t \mathbf{z}_{t-1} + \nabla \hat{v}(S_t,\mathbf{w}_t)\big), \tag{12.32}$$

This eligibility trace, together with the usual semi-gradient parameter-update rule for TD(λ) (12.7), forms a general TD(λ) algorithm that can be applied to either on-policy or off-policy data. In the on-policy case, the algorithm is exactly TD(λ) because ρ_t is alway 1 and (12.32) becomes the usual accumulating trace (12.5) (extended to variable λ and γ). In the off-policy case, the algorithm often works well but, as an semi-gradient method, is not guaranteed to be stable. In the next few sections we will consider extensions of it that do guarantee stability.

A very similar series of steps can be followed to derive the off-policy eligibility traces for *action*-value methods and corresponding general Sarsa(λ) algorithms. One could start with either recursive form for the general action-based λ-return, (12.26) or (12.27), but the former works out to be simpler. We extend (12.26) to the off-policy case after the model of (7.11) to produce

$$\begin{aligned}
G_t^{\lambda a} &\doteq R_{t+1} + \gamma_{t+1}\Big((1-\lambda_{t+1})\big(\rho_{t+1}\hat{q}(S_{t+1},A_{t+1},\mathbf{w}_t) + (1-\rho_{t+1})\bar{Q}_{t+1}\big) \\
&\qquad\qquad + \lambda_{t+1}\big(\rho_{t+1}G_{t+1}^{\lambda a} + (1-\rho_{t+1})\bar{Q}_{t+1}\big)\Big) \\
&= R_{t+1} + \gamma_{t+1}\Big((1-\lambda_{t+1})\rho_{t+1}\hat{q}(S_{t+1},A_{t+1},\mathbf{w}_t) + \lambda_{t+1}\rho_{t+1}G_{t+1}^{\lambda a} + (1-\rho_{t+1})\bar{Q}_{t+1}\big)\Big)
\end{aligned} \tag{12.33}$$

where $\bar{Q}_{t+1}$ is as given by (12.28). Again the λ-return can be written approximately as the sum of TD errors,

$$G_t^{\lambda a} \approx \hat{q}(S_t,A_t,\mathbf{w}_t) + \sum_{k=t}^{\infty} \delta_k^a \prod_{i=t+1}^{k} \gamma_i \lambda_i \rho_i, \tag{12.34}$$

using a novel form of the TD error:

$$\delta_t^a = R_{t+1} + \gamma_{t+1}\Big(\rho_{t+1}\hat{q}(S_{t+1},A_{t+1},\mathbf{w}_t) + (1-\rho_{t+1})\bar{Q}_{t+1}\Big) - \hat{q}(S_t,A_t,\mathbf{w}_t). \tag{12.35}$$

As before, the approximation becomes exact if the approximate value function does not change.

Exercise 12.10 Prove that (12.34) becomes exact if the value function does not change. To save writing, consider the case of $t=0$, and use the notation $Q_k = \hat{q}(S_k,A_k,\mathbf{w})$. Hint: Start by writing out δ_0^a and $G_0^{\lambda a}$, then $G_0^{\lambda a} - Q_0$. □

Exercise 12.11 The truncated version of the general off-policy return is denoted $G^{\lambda a}_{t:h}$. Guess the correct equation, based on (12.34). □

Using steps entirely analogous to those for the state case, one can write a forward-view update based on (12.34), transform the sum of the updates using the summation rule, and finally derive the following form for the eligibility trace for action values:

$$\mathbf{z}_t \doteq \gamma_t \lambda_t \rho_t \mathbf{z}_{t-1} + \nabla \hat{q}(S_t, A_t, \mathbf{w}_t). \tag{12.36}$$

This eligibility trace, together with the usual semi-gradient parameter-update rule (12.7), forms a general Sarsa(λ) algorithm that can be applied to either on-policy or off-policy data. In the on-policy case with constant λ and γ this algorithm is identical to the Sarsa(λ) algorithm presented in Section 12.7. In the off-policy case this algorithm is not stable unless combined with one of the methods presented in the following sections.

Exercise 12.12 Show in detail the steps outlined above for deriving (12.36) from (12.34). Start with the update (12.21), substitute $G^{\lambda a}_t$ from (12.33) for G^{λ}_t, then follow similar steps as led to (12.32). □

*__Exercise 12.13__ Show how similar steps can be followed starting from the Expected Sarsa form of the action-based λ-return (12.27) and (12.28) to derive the same eligibility trace algorithm as (12.36), but with a different TD error:

$$\delta^a_t = R_{t+1} + \gamma_{t+1}\bar{Q}_{t+1} - \hat{q}(S_t, A_t, \mathbf{w}_t) + \gamma_{t+1}\lambda_{t+1}\rho_{t+1}\big(\hat{q}(S_{t+1}, A_{t+1}, \mathbf{w}_t) - \bar{Q}_{t+1}\big). \square$$

At $\lambda = 1$, these algorithms become closely related to corresponding Monte Carlo algorithms. One might expect that an exact equivalence would hold for episodic problems and off-line updating, but in fact the relationship is subtler and slightly weaker than that. Under these most favorable conditions still there is not an episode by episode equivalence of updates, only of their expectations. This should not be surprising as these method make irrevocable updates as a trajectory unfolds, whereas true Monte Carlo methods would make no update for a trajectory if any action within it has zero probability under the target policy. In particular, all of these methods, even at $\lambda = 1$, still bootstrap in the sense that their targets depend on the current value estimates—its just that the dependence cancels out in expected value. Whether this is a good or bad property in practice is another question. Recently methods have been proposed that do achieve an exact equivalence (Sutton, Mahmood, Precup and van Hasselt, 2014). These methods require an additional vector of "provisional weights" that keep track of updates which have been made but may need to be retracted (or emphasized) depending on the actions taken later. The state and state–action versions of these methods are called PTD(λ) and PQ(λ) respectively, where the 'P' stands for Provisional.

The practical consequences of all these new off-policy methods have not yet been established. Undoubtedly, issues of high variance will arise as they do in all off-policy methods using importance sampling (Section 11.9).

If $\lambda < 1$, then all these off-policy algorithms involve bootstrapping and the deadly triad applies (Section 11.3), meaning that they can be guaranteed stable only for the tabular case, for state aggregation, and for other limited forms of function approximation. For linear and more-general forms of function approximation the parameter vector may diverge to infinity as in the examples in Chapter 11. As we discussed there, the challenge of off-policy learning has two parts. Off-policy eligibility traces deal effectively with the first part of the challenge, correcting for the expected value of the targets, but not at all with the second part of the challenge, having to do with the distribution of updates. Algorithmic strategies for meeting the second part of the challenge of off-policy learning with eligibility traces are summarized in Section 12.11.

Exercise 12.14 What are the dutch-trace and replacing-trace versions of off-policy eligibility traces for state-value and action-value methods? □

12.10 Watkins's Q(λ) to Tree-Backup(λ)

Several methods have been proposed over the years to extend Q-learning to eligibility traces. The original is *Watkins's Q(λ)*, which decays its eligibility traces in the usual way as long as a greedy action was taken, then cuts the traces to zero after the first non-greedy action. The backup diagram for Watkins's Q(λ) is shown in Figure 12.12. In Chapter 6, we unified Q-learning and Expected Sarsa in the off-policy version of the latter, which includes Q-learning as a special case, and generalizes it to arbitrary target policies, and in the previous section of this chapter we completed our treatment of Expected Sarsa by generalizing it to off-policy eligibility traces. In Chapter 7, however, we distinguished multi-step Expected Sarsa from multi-step Tree Backup, where the latter retained the property of not using importance sampling. It remains then to present the eligibility trace version of Tree Backup, which we well call *Tree-Backup(λ)*, or *TB(λ)* for short. This is arguably the true successor to Q-learning because it retains its appealing absence of importance sampling even though it can be applied to off-policy data.

The concept of TB(λ) is straightforward. As shown in its backup diagram in Figure 12.13, the tree-backup updates of each length (from Section 7.5) are weighted in the usual way dependent on the bootstrapping parameter λ. To get the detailed equations, with the right indexes on the general bootstrapping and discounting parameters, it is best to start with a recursive form (12.27) for the λ-return using action values, and then expand the bootstrapping case of the target after the model of

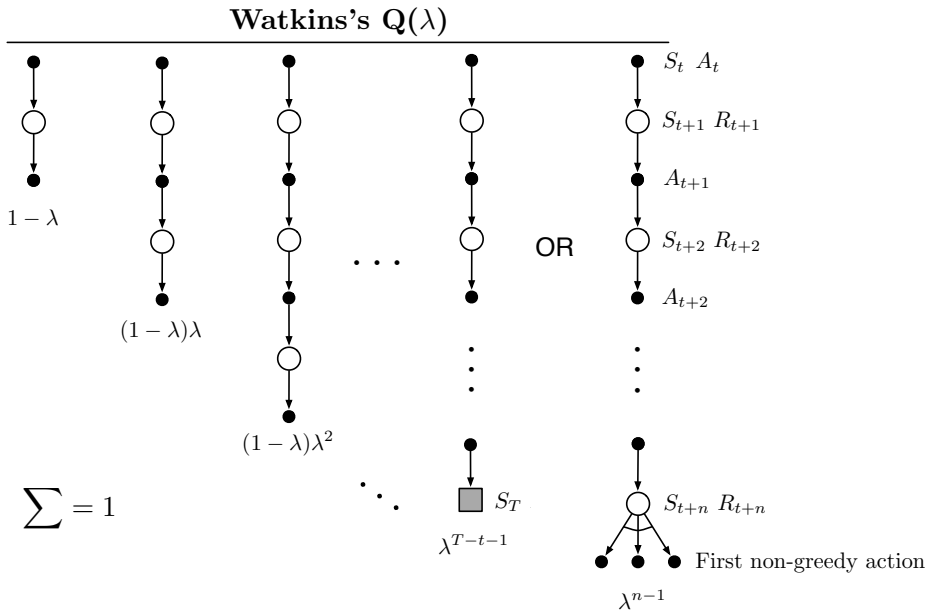


Figure 12.12: The backup diagram for Watkins's Q(λ). The series of component updates ends either with the end of the episode or with the first nongreedy action, whichever comes first.

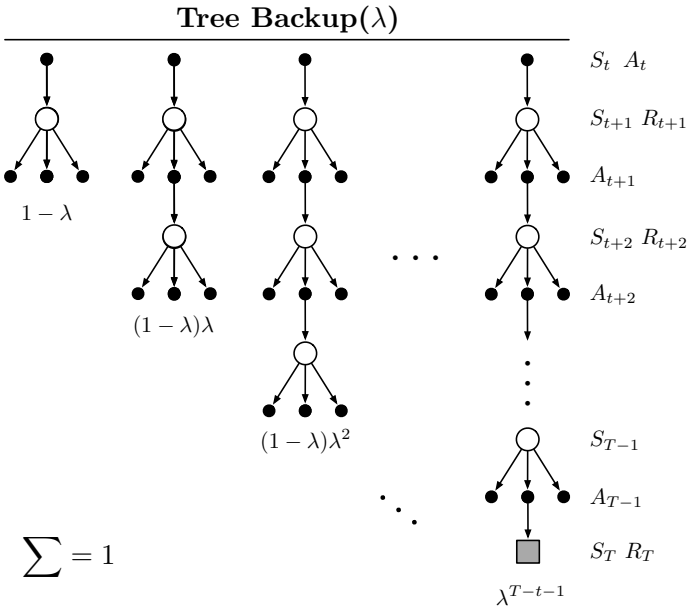


Figure 12.13: The backup diagram for the λ version of the Tree Backup algorithm.

(7.13):

$$\begin{aligned} G_t^{\lambda a} &\doteq R_{t+1} + \gamma_{t+1}\Big((1-\lambda_{t+1})\bar{Q}_{t+1} + \lambda_{t+1}\Big(\\ &\qquad \sum_{a \neq A_{t+1}} \pi(a|S_{t+1})\hat{q}(S_{t+1}, a, \mathbf{w}_t) \quad + \quad \pi(A_{t+1}|S_{t+1})G_{t+1}^{\lambda a}\Big)\Big) \\ &= R_{t+1} + \gamma_{t+1}\Big(\bar{Q}_{t+1} + \lambda_{t+1}\pi(A_{t+1}|S_{t+1})\Big(G_{t+1}^{\lambda a} - \hat{q}(S_{t+1}, A_{t+1}, \mathbf{w}_t)\Big)\Big) \end{aligned}$$

As per the usual pattern, it can also be written approximately (ignoring changes in the approximate value function) as a sum of TD errors,

$$G_t^{\lambda a} \approx \hat{q}(S_t, A_t, \mathbf{w}_t) + \sum_{k=t}^{\infty} \delta_k^a \prod_{i=t+1}^{k} \gamma_i \lambda_i \pi(A_i|S_i), \tag{12.37}$$

using the expectation form of the action-based TD error:

$$\delta_t^a = R_{t+1} + \gamma_{t+1}\bar{Q}_{t+1} - \hat{q}(S_t, A_t, \mathbf{w}_t). \tag{12.38}$$

Following the same steps as in the previous section, we arrive at a special eligibility trace update involving the target-policy probabilities of the selected actions,

$$\mathbf{z}_t \doteq \gamma_t \lambda_t \pi(A_t|S_t)\mathbf{z}_{t-1} + \nabla \hat{q}(S_t, A_t, \mathbf{w}_t). \tag{12.39}$$

This, together with the usual parameter-update rule (12.7), defines the TB(λ) algorithm. Like all semi-gradient algorithms, TB(λ) is not guaranteed to be stable when used with off-policy data and with a powerful function approximator. For that it would have to be combined with one of the methods presented in the next section.

Exercise 12.15 (programming) De Asis (personal communication) has proposed a new algorithm he calls "Rigid Tree Backup" that combines (12.38) with (12.36) in the usual way (12.7). Implement this algorithm and compare it empirically with TB(λ) and general Sarsa(λ) on the 19-state random walk. □

12.11 Stable Off-policy Methods with Traces

Several methods using eligibility traces have been proposed that achieve guarantees of stability under off-policy training, and here we present four of the most important using this book's standard notation, including general bootstrapping and discounting functions. All are based on either the Gradient-TD or the Emphatic-TD ideas presented in Sections 11.7 and 11.8. All the algorithms assume linear function approximation, though extensions to nonlinear function approximation can also be found in the literature.

GTD(λ) is the eligibility-trace algorithm analogous to TDC, the better of the two state-value Gradient-TD prediction algorithms discussed in Section 11.7. Its goal is to learn a parameter $\mathbf{w}_t$ such that $\hat{v}(s,\mathbf{w}) \doteq \mathbf{w}_t^\top \mathbf{x}(s) \approx v_\pi(s)$ even from data that is due to following another policy b. Its update is

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha\delta_t^s \mathbf{z}_t - \alpha\gamma_{t+1}(1-\lambda_{t+1})\left(\mathbf{z}_t^\top \mathbf{v}_t\right)\mathbf{x}_{t+1}, \tag{12.40}$$

with δ_t^s, $\mathbf{z}_t$, and ρ_t defined in the usual ways for state values (12.30) (12.32) (11.1), and

$$\mathbf{v}_{t+1} \doteq \beta\delta_t^s \mathbf{z}_t - \beta\left(\mathbf{v}_t^\top \mathbf{x}_t\right)\mathbf{x}_t, \tag{12.41}$$

where, as in Section 11.7, $\mathbf{v} \in \mathbb{R}^d$ is a vector of the same dimension as $\mathbf{w}$, initialized to $\mathbf{v}_0 = \mathbf{0}$, and $\beta > 0$ is a second step-size parameter.

GQ(λ) is the Gradient-TD algorithm for action values with eligibility traces. Its goal is to learn a parameter $\mathbf{w}_t$ such that $\hat{q}(s,a,\mathbf{w}_t) \doteq \mathbf{w}_t^\top \mathbf{x}(s,a) \approx q_\pi(s,a)$ from off-policy data. If the target policy is ε-greedy, or otherwise biased toward the greedy policy for $\hat{q}$, then GQ(λ) can be used as a control algorithm. Its update is

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha\delta_t^a \mathbf{z}_t - \alpha\gamma_{t+1}(1-\lambda_{t+1})\left(\mathbf{z}_t^\top \mathbf{v}_t\right)\bar{\mathbf{x}}_{t+1}, \tag{12.42}$$

where $\bar{\mathbf{x}}_t$ is the average feature vector for S_t under the target policy,

$$\bar{\mathbf{x}}_t \doteq \sum_a \pi(a|S_t)\mathbf{x}(S_t,a), \tag{12.43}$$

δ_t^a is the expectation form of the TD error, which can be written,

$$\delta_t^a \doteq R_{t+1} + \gamma_{t+1}\mathbf{w}_t^\top \bar{\mathbf{x}}_{t+1} - \mathbf{w}_t^\top \mathbf{x}_t, \tag{12.44}$$

$\mathbf{z}_t$ is defined in the usual ways for action values (12.36), and the rest is as in GTD(λ), including the update for $\mathbf{v}_t$ (12.41).

HTD(λ) is a hybrid state-value algorithm combining aspects of GTD(λ) and TD(λ). Its most appealing feature is that it is a strict generalization of TD(λ) to off-policy learning, meaning that if the behavior policy happens to be the same as the target policy, then HTD(λ) becomes the same as TD(λ), which is not true for GTD(λ). This is appealing because TD(λ) is often faster than GTD(λ) when both algorithms converge, and TD(λ) requires setting only a single step size. HTD(λ) is defined by

$$\begin{aligned}
\mathbf{w}_{t+1} &\doteq \mathbf{w}_t + \alpha\delta_t^s \mathbf{z}_t + \alpha\left((\mathbf{z}_t - \mathbf{z}_t^b)^\top \mathbf{v}_t\right)(\mathbf{x}_t - \gamma_{t+1}\mathbf{x}_{t+1}), \\
\mathbf{v}_{t+1} &\doteq \mathbf{v}_t + \beta\delta_t^s \mathbf{z}_t - \beta\left(\mathbf{z}_t^{b\top} \mathbf{v}_t\right)(\mathbf{x}_t - \gamma_{t+1}\mathbf{x}_{t+1}), && \mathbf{v}_0 \doteq \mathbf{0}, \\
\mathbf{z}_t &\doteq \rho_t\big(\gamma_t\lambda_t \mathbf{z}_{t-1} + \mathbf{x}_t\big), && \mathbf{z}_{-1} \doteq \mathbf{0}, \\
\mathbf{z}_t^b &\doteq \gamma_t\lambda_t \mathbf{z}_{t-1}^b + \mathbf{x}_t, && \mathbf{z}_{-1}^b \doteq \mathbf{0},
\end{aligned} \tag{12.45}$$

where $\beta > 0$ again is a second step-size parameter that becomes irrelevant in the on-policy case in which $b = \pi$. In addition to the second set of weights, $\mathbf{v}_t$, HTD(λ) also has a second set of eligibility traces, $\mathbf{z}_t^b$. These are a conventional accumulating eligibility trace for the behavior policy and become equal to $\mathbf{z}_t$ if all the ρ_t are 1, which causes the second term in the $\mathbf{w}_t$ update to be zero and the overall update to reduce to TD(λ).

Emphatic TD(λ) is the extension of the one-step Emphatic-TD algorithm from Section 11.8 to eligibility traces. The resultant algorithm retains strong off-policy convergence guarantees while enabling any degree of bootstrapping, albeit at the cost of high variance and potentially slow convergence. Emphatic TD(λ) is defined by

$$\begin{aligned}
\boldsymbol{\theta}_{t+1} &\doteq \boldsymbol{\theta}_t + \alpha\delta_t\mathbf{z}_t \\
\delta_t &\doteq R_{t+1} + \gamma_{t+1}\boldsymbol{\theta}_t^\top\mathbf{x}_{t+1} - \boldsymbol{\theta}_t^\top\mathbf{x}_t \\
\mathbf{z}_t &\doteq \rho_t\big(\gamma_t\lambda_t\mathbf{z}_{t-1} + M_t\mathbf{x}_t\big), && \text{with } \mathbf{z}_{-1} \doteq \mathbf{0} \\
M_t &\doteq \lambda_t I_t + (1-\lambda_t)F_t \\
F_t &\doteq \rho_{t-1}\gamma_t F_{t-1} + I_t, && \text{with } F_0 \doteq i(S_0),
\end{aligned} \tag{12.46}$$

where $M_t \geq 0$ is the general form of *emphasis*, $F_t \geq 0$ is termed the *followon trace*, and $I_t \geq 0$ is the *interest*, as described in Section 11.8. Note that M_t, like δ_t, is not really an additional memory variable. It can be removed from the algorithm by substituting its definition into the eligibility-trace equation. Pseudocode and software for the true online version of Emphatic-TD(λ) are available on the web (Sutton, 2015b).

In the on-policy case ($\rho_t = 0$, for all t), Emphatic-TD(λ) is similar to conventional TD(λ), but still significantly different. In fact, whereas Emphatic-TD(λ) is guaranteed to converge for all state-dependent λ functions, TD(λ) is not. TD(λ) is guaranteed convergent only for all constant λ. See Yu's counterexample (Ghiassian, Rafiee, and Sutton, 2016).

12.12 Implementation Issues

It might at first appear that methods using eligibility traces are much more complex than one-step methods. A naive implementation would require every state (or state–action pair) to update both its value estimate and its eligibility trace on every time step. This would not be a problem for implementations on single-instruction, multiple-data, parallel computers or in plausible neural implementations, but it is a problem for implementations on conventional serial computers. Fortunately, for typical values of λ and γ the eligibility traces of almost all states are almost always nearly zero; only those that have recently been visited will have traces significantly greater than zero. In practice, only these few states need to be updated to closely approximate these algorithms.

In practice, then, implementations on conventional computers may keep track of and update only the few states with nonzero traces. Using this trick, the computational expense of using traces is typically just a few times that of a one-step method. The exact multiple of course depends on λ and γ and on the expense of the other computations. Note that the tabular case is in some sense the worst case for the computational complexity of eligibility traces. When function approximation is used, the computational advantages of not using traces generally decrease. For example, if artificial neural networks and backpropagation are used, then eligibility traces generally cause only a doubling of the required memory and computation per step. Truncated λ-return methods (Section 12.3) can be computationally efficient on conventional computers though they always require some additional memory.

12.13 Conclusions

Eligibility traces in conjunction with TD errors provide an efficient, incremental way of shifting and choosing between Monte Carlo and TD methods. The atomic multi-step methods of Chapter 7 also enabled this, but eligibility trace methods are more general, often faster to learn, and offer different computational complexity tradeoffs. This chapter has offered an introduction to the elegant, emerging theoretical understanding of eligibility traces for on- and off-policy learning and for variable bootstrapping and discounting. One aspect of this elegant theory is true online methods, which exactly reproduce the behavior of expensive ideal methods while retaining the computational congeniality of conventional TD methods. Another aspect is the possibility of derivations that automatically convert from intuitive *forward-view* methods to more efficient incremental backward-view algorithms. We illustrated this general idea in a derivation that started with a classical, expensive Monte Carlo algorithm and ended with a cheap incremental non-TD implementation using the same novel eligibility trace used in true online TD methods.

As we mentioned in Chapter 5, Monte Carlo methods may have advantages in non-Markov tasks because they do not bootstrap. Because eligibility traces make TD methods more like Monte Carlo methods, they also can have advantages in these cases. If one wants to use TD methods because of their other advantages, but the task is at least partially non-Markov, then the use of an eligibility trace method is indicated. Eligibility traces are the first line of defense against both long-delayed rewards and non-Markov tasks.

By adjusting λ, we can place eligibility trace methods anywhere along a continuum from Monte Carlo to one-step TD methods. Where shall we place them? We do not yet have a good theoretical answer to this question, but a clear empirical answer appears to be emerging. On tasks with many steps per episode, or many steps within the half-life of discounting, it appears significantly better to use eligibility traces than not to (e.g., see Figure 12.14). On the other hand, if the traces are so long as to produce a pure Monte Carlo method, or nearly so, then performance degrades sharply. An intermediate mixture appears to be the best choice. Eligibility traces should be used to bring us toward Monte Carlo methods, but not all the way there. In the future it may be possible to vary the trade-off between TD and Monte Carlo methods more finely by using variable λ, but at present it is not clear how this can be done reliably and usefully.

Methods using eligibility traces require more computation than one-step methods, but in return they offer significantly faster learning, particularly when rewards are delayed by many steps. Thus it often makes sense to use eligibility traces when data are scarce and cannot be repeatedly processed, as is often the case in on-line applications. On the other hand, in off-line applications in which data can be generated cheaply, perhaps from an inexpensive simulation, then it often does not pay to use eligibility traces. In these cases the objective is not to get more out of a limited amount of data, but simply to process as much data as possible as quickly as possible. In these cases the speedup per datum due to traces is typically not worth their computational cost, and one-step methods are favored.

*__Exercise 12.16__ How might Double Expected Sarsa be extended to eligibility traces? □

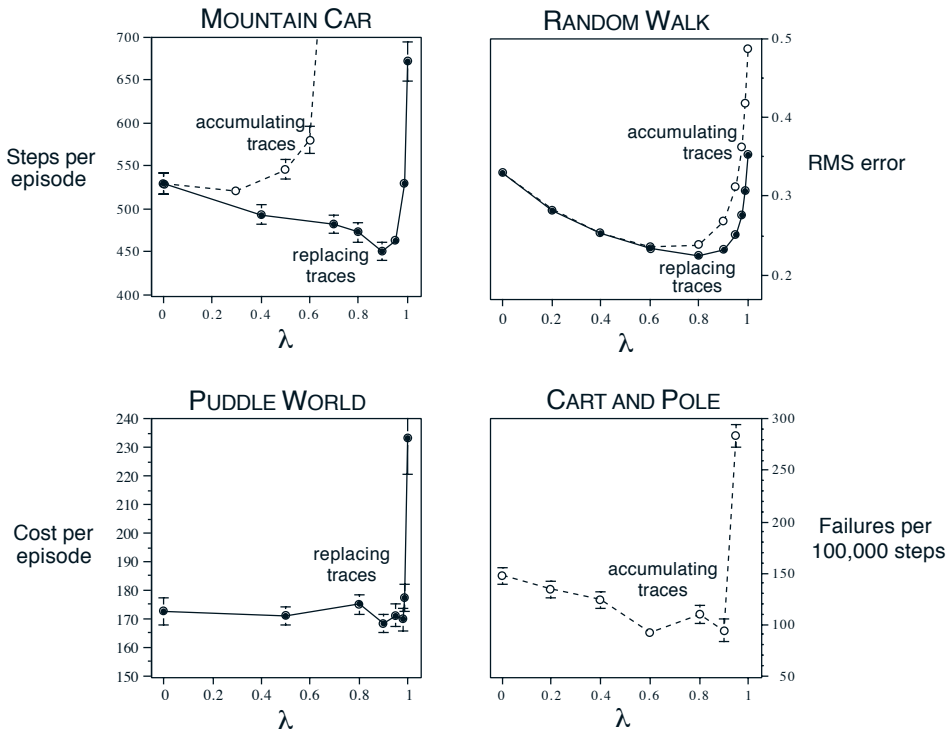


Figure 12.14: The effect of λ on reinforcement learning performance in four different test problems. In all cases, *lower* numbers represent better performance. The two left panels are applications to simple continuous-state control tasks using the Sarsa(λ) algorithm and tile coding, with either replacing or accumulating traces (Sutton, 1996). The upper-right panel is for policy evaluation on a random walk task using TD(λ) (Singh and Sutton, 1996). The lower right panel is unpublished data for the pole-balancing task (Example 3.4) from an earlier study (Sutton, 1984).

Bibliographical and Historical Remarks

Eligibility traces came into reinforcement learning via the fecund ideas of Klopf (1972). Our use of eligibility traces is based on Klopf's work (Sutton, 1978a, 1978b, 1978c; Barto and Sutton, 1981a, 1981b; Sutton and Barto, 1981a; Barto, Sutton, and Anderson, 1983; Sutton, 1984). We may have been the first to use the term "eligibility trace" (Sutton and Barto, 1981). The idea that stimuli produce aftereffects in the nervous system that are important for learning is very old. See Chapter 14. Some of the earliest uses of eligibility traces were in the actor–critic methods discussed in Chapter 13 (Barto, Sutton, and Anderson, 1983; Sutton, 1984).

12.1 The λ-return and its error-reduction properties were introduced by Watkins (1989) and further developed by Jaakkola, Jordan and Singh (1994). The random walk results in this and subsequent sections are new to this text, as are the terms "forward view" and "backward view." The notion of a λ-return algorithm was introduced in the first edition of this text. The more refined treatment presented here was developed in conjunction with Harm van Seijen (e.g., van Seijen and Sutton, 2014).

12.2 TD(λ) with accumulating traces was introduced by Sutton (1988, 1984). Convergence in the mean was proved by Dayan (1992), and with probability 1 by many researchers, including Peng (1993), Dayan and Sejnowski (1994), and Tsitsiklis (1994) and Gurvits, Lin, and Hanson

(1994). The bound on the error of the asymptotic λ-dependent solution of linear TD(λ) is due to Tsitsiklis and Van Roy (1997).

12.3-5 Truncated TD methods were developed by Cichosz (1995) and van Seijen (2016). True online TD(λ) and the other ideas presented in these sections are primarily due to work of van Seijen (van Seijen and Sutton, 2014; van Seijen et al., 2016) Replacing traces are due to Singh and Sutton (1996).

12.6 The material in this section is from van Hasselt and Sutton (2015).

12.7 Sarsa(λ) with accumulating traces was first explored as a control method by Rummery and Niranjan (1994; Rummery, 1995). True Online Sarsa(λ) was introduced by van Seijen and Sutton (2014). The algorithm on page 252 was adapted from van Seijen et al. (2016). The Mountain Car results were made new for this text, except for Figure 12.11 which is adapted from van Seijen and Sutton (2014).

12.8 Perhaps the first published discussion of variable λ was by Watkins (1989), who pointed out that the cutting off of the update sequence (Figure 12.12) in his Q(λ) when a nongreedy action was selected could be implemented by temporarily setting λ to 0.

Variable λ was introduced in the first edition of this text. The roots of variable γ are in the work on options (Sutton, Precup, and Singh, 1999) and its precursors (Sutton, 1995a), becoming explicit in the GQ(λ) paper (Maei and Sutton, 2010), which also introduced some of these recursive forms for the λ-returns.

A different notion of variable λ has been developed by Yu (2012).

12.9 Off-policy eligibility traces were introduced by Precup et al. (2000, 2001), then further developed by Bertsekas and Yu (2009), Maei (2011; Maei and Sutton, 2010), Yu (2012), and by Sutton, Mahmood, Precup, and van Hasselt (2014). The latter reference in particular gives a powerful forward view for off-policy TD methods with general state-dependent λ and γ. The presentation here seems to be new.

12.10 Watkins's Q(λ) is due to Watkins (1989). Convergence has still not been proved for any control method for $0 < \lambda < 1$. Tree Backup(λ) is due to Precup, Sutton, and Singh (2000).

12.11 GTD(λ) is due to Maei (2011). GQ(λ) is due to Maei and Sutton (2010). HTD(λ) is due to White and White (2016) based on the one-step HTD algorithm introduced by Hackman (2012). Emphatic TD(λ) was introduced by Sutton, Mahmood, and White (2016), who proved its stability, then was proved to be convergent by Yu (2015a,b), and developed further by Hallak, Tamar, Munos, and Mannor (2016).

Chapter 13

Policy Gradient Methods

In this chapter we consider something new. So far in this book almost all the methods have learned the values of actions and then selected actions based on their estimated action values[1]; their policies would not even exist without the action-value estimates. In this chapter we consider methods that instead learn a *parameterized policy* that can select actions without consulting a value function. A value function may still be used to *learn* the policy parameter, but is not required for action selection. We use the notation $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ for the policy's parameter vector. Thus we write $\pi(a|s,\boldsymbol{\theta}) = \Pr\{A_t{=}a \mid S_t{=}s, \boldsymbol{\theta}_t{=}\boldsymbol{\theta}\}$ for the probability that action a is taken at time t given that the environment is in state s at time t with parameter $\boldsymbol{\theta}$. If a method uses a learned value function as well, then the value function's weight vector is denoted $\mathbf{w} \in \mathbb{R}^d$ as usual, as in $\hat{v}(s,\mathbf{w})$.

In this chapter we consider methods for learning the policy parameter based on the gradient of some performance measure $J(\boldsymbol{\theta})$ with respect to the policy parameter. These methods seek to *maximize* performance, so their updates approximate gradient *ascent* in J:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha \widehat{\nabla J(\boldsymbol{\theta}_t)}, \tag{13.1}$$

where $\widehat{\nabla J(\boldsymbol{\theta}_t)}$ is a stochastic estimate whose expectation approximates the gradient of the performance measure with respect to its argument $\boldsymbol{\theta}_t$. All methods that follow this general schema we call *policy gradient methods*, whether or not they also learn an approximate value function. Methods that learn approximations to both policy and value functions are often called *actor–critic methods*, where 'actor' is a reference to the learned policy, and 'critic' refers to the learned value function, usually a state-value function. First we treat the episodic case, in which performance is defined as the value of the start state under the parameterized policy, before going on to consider the continuing case, in which performance is defined as the average reward rate, as in Section 10.3. In the end we are able to express the algorithms for both cases in very similar terms.

13.1 Policy Approximation and its Advantages

In policy gradient methods, the policy can be parameterized in any way, as long as $\pi(a|s,\boldsymbol{\theta})$ is differentiable with respect to its parameters, that is, as long as $\nabla_{\boldsymbol{\theta}}\pi(a|s,\boldsymbol{\theta})$ exists and is always finite. In practice, to ensure exploration we generally require that the policy never becomes deterministic (i.e., that $\pi(a|s,\boldsymbol{\theta}) \in (0,1)$, for all $s,a,\boldsymbol{\theta}$). In this section we introduce the most common parameterization

[1]The lone exception is the gradient bandit algorithms of Section 2.8. In fact, that section goes through many of the same steps, in the single-state bandit case, as we go through here for full MDPs. Reviewing that section would be good preparation for fully understanding this chapter.

for discrete action spaces and point out the advantages it offers over action-value methods. Policy-based methods also offer useful ways of dealing with continuous action spaces, as we describe later in Section 13.7.

If the action space is discrete and not too large, then a natural kind of parameterization is to form parameterized numerical preferences $h(s, a, \boldsymbol{\theta}) \in \mathbb{R}$ for each state–action pair. The actions with the highest preferences in each state are given the highest probabilities of being selected, for example, according to an exponential softmax distribution:

$$\pi(a|s, \boldsymbol{\theta}) = \frac{\exp(h(s, a, \boldsymbol{\theta}))}{\sum_b \exp(h(s, b, \boldsymbol{\theta}))}, \tag{13.2}$$

where $\exp(x) = e^x$, where $e \approx 2.71828$ is the base of the natural logarithm. Note that the denominator here is just what is required so that the action probabilities in each state sum to one. The preferences themselves can be parameterized arbitrarily. For example, they might be computed by a deep neural network, where $\boldsymbol{\theta}$ is the vector of all the connection weights of the network (as in the AlphaGo system described in Section 16.6). Or the preferences could simply be linear in features,

$$h(s, a, \boldsymbol{\theta}) = \boldsymbol{\theta}^\top \mathbf{x}(s, a), \tag{13.3}$$

using feature vectors $\mathbf{x}(s, a) \in \mathbb{R}^{d'}$ constructed by any of the methods described in Chapter 9.

An immediate advantage of selecting actions according to the softmax in action preferences (13.2) is that the approximate policy can approach a deterministic policy, whereas with ε-greedy action selection over action values there is always an ε probability of selecting a random action. Of course, one could select according to a softmax over action values, but this alone would not allow the policy to approach a deterministic policy. Instead, the action-value estimates would converge to their corresponding true values, which would differ by a finite amount, translating to specific probabilities other than 0 and 1. If the softmax included a temperature parameter, then the temperature could be reduced over time to approach determinism, but in practice it would be difficult to choose the reduction schedule, or even the initial temperature, without more prior knowledge of the true action values than we would like to assume. Action preferences are different because they do not approach specific values; instead they are driven to produce the optimal stochastic policy. If the optimal policy is deterministic, then the preferences of the optimal actions will be driven infinitely higher than all suboptimal actions (if permited by the parameterization).

In problems with significant function approximation, the best approximate policy may be stochastic. For example, in card games with imperfect information the optimal play is often to do two different things with specific probabilities, such as when bluffing in Poker. Action-value methods have no natural way of finding stochastic optimal policies, whereas policy approximating methods can, as shown in Example 13.1. This is a second significant advantage of policy-based methods.

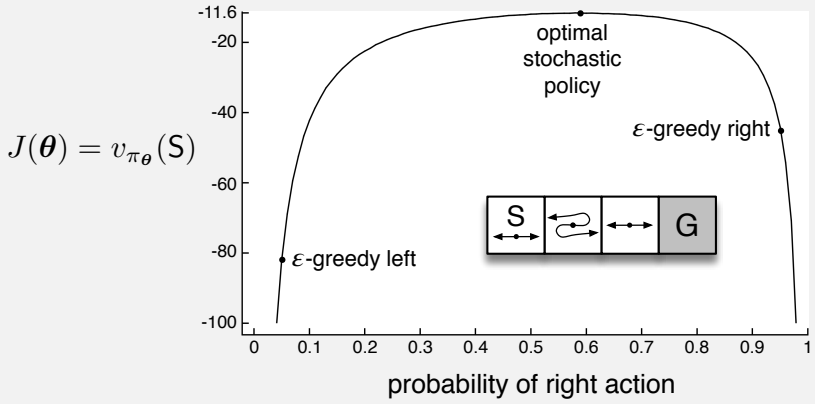
Exercise 13.1 Use your knowledge of the gridworld and its dynamics to determine an *exact* symbolic expression for the optimal probability of selecting the right action in Example 13.1.

Perhaps the simplest advantage that policy parameterization may have over action-value parameterization is that the policy may be a simpler function to approximate. Problems vary in the complexity of their policies and action-value functions. For some, the action-value function is simpler and thus easier to approximate. For others, the policy is simpler. In the latter case a policy-based method will typically learn faster and yield a superior asymptotic policy (as seems to be the case with Tetris; see Şimşek, Algórta, and Kothiyal, 2016).

Finally, we note that the choice of policy parameterization is sometimes a good way of injecting prior knowledge about the desired form of the policy into the reinforcement learning system. This is often the most important reason for using a policy-based learning method.

Example 13.1 Short corridor with switched actions

Consider the small corridor gridworld shown inset in the graph below. The reward is -1 per step, as usual. In each of the three nonterminal states there are only two actions, right and left. These actions have their usual consequences in the first and third states (left causes no movement in the first state), but in the second state they are reversed, so that right moves to the left and left moves to the right. The problem is difficult because all the states appear identical under the function approximation. In particular, we define $\mathbf{x}(s, \text{right}) = [1, 0]^\top$ and $\mathbf{x}(s, \text{left}) = [0, 1]^\top$, for all s. An action-value method with ε-greedy action selection is forced to choose between just two policies: choosing right with high probability $1 - \varepsilon/2$ on all steps or choosing left with the same high probability on all time steps. If $\varepsilon = 0.1$, then these two policies achieve a value (at the start state) of less than -44 and -82, respectively, as shown in the graph. A method can do significantly better if it can learn a specific probability with which to select right. The best probability is about 0.59, which achieves a value of about -11.6.



13.2 The Policy Gradient Theorem

In addition to the practical advantages of policy parameterization over ε-greedy action selection, there is also an important theoretical advantage. With continuous policy parameterization the action probabilities change smoothly as a function of the learned parameter, whereas in ε-greedy selection the action probabilities may change dramatically for an arbitrarily small change in the estimated action values, if that change results in a different action having the maximal value. Largely because of this stronger convergence guarantees are available for policy-gradient methods than for action-value methods. In particular, it is the continuity of the policy dependence on the parameters that enables policy-gradient methods to approximate gradient ascent (13.1).

The episodic and continuing cases define the performance measure, $J(\boldsymbol{\theta})$, differently and thus have to be treated separately to some extent. Nevertheless, we will try to present both cases uniformly, and we develop a notation so that the major theoretical results can be decribed with a single set of equations.

In this section we treat the episodic case, for which we define the performance measure as the value of the start state of the episode. We can simplify the notation without losing any meaningful generality by assuming that every episode starts in some particular (non-random) state s_0. Then, in the episodic case we define performance as

$$J(\boldsymbol{\theta}) \doteq v_{\pi_{\boldsymbol{\theta}}}(s_0), \tag{13.4}$$

where $v_{\pi_{\boldsymbol{\theta}}}$ is the true value function for $\pi_{\boldsymbol{\theta}}$, the policy determined by $\boldsymbol{\theta}$. From here on in our discussion we will assume no discounting ($\gamma = 1$) for the episodic case, although for completeness we do include the possibility of discounting in the boxed algorithms.

With function approximation, it may seem challenging to change the policy parameter in a way that ensures improvement. The problem is that performance depends on both the action selections and the distribution of states in which those selections are made, and that both of these are affected by the policy parameter. Given a state, the effect of the policy parameter on the actions, and thus on reward, can be computed in a relatively straightforward way from knowledge of the parameterization. But the effect of the policy on the state distribution is a function of the environment and is typically unknown. How can we estimate the performance gradient with respect to the policy parameter when the gradient depends on the unknown effect of policy changes on the state distribution?

Fortunately, there is an excellent theoretical answer to this challenge in the form of the *policy gradient theorem*, which provides us an analytic expression for the gradient of performance with respect to the policy parameter (which is what we need to approximate for gradient ascent (13.1)) that does *not* involve the derivative of the state distribution. The policy gradient theorem establishes that

$$\nabla J(\boldsymbol{\theta}) \propto \sum_s \mu(s) \sum_a q_\pi(s,a) \nabla_{\boldsymbol{\theta}} \pi(a|s,\boldsymbol{\theta}), \tag{13.5}$$

where the gradients are column vectors of partial derivatives with respect to the components of $\boldsymbol{\theta}$, and π denotes the policy corresponding to parameter vector $\boldsymbol{\theta}$. The symbol $\propto$ here means "proportional to". In the episodic case, the constant of proportionality is the average length of an episode, and in the continuing case it is 1, so that the relationship is actually an equality. The distribution μ here (as in Chapters 9 and 10) is the on-policy distribution under π (see page 163). The policy gradient theorem is proved for the episodic case in the box on the next page.

Proof of the Policy Gradient Theorem (episodic case)

With just elementary calculus and re-arranging terms we can prove the policy gradient theorem from first principles. To keep the notation simple, we leave it implicit in all cases that π is a function of $\boldsymbol{\theta}$, and all gradients are also implicitly with respect to $\boldsymbol{\theta}$. First note that the gradient of the state-value function can be written in terms of the action-value function as

$$
\begin{aligned}
\nabla v_\pi(s) &= \nabla \left[\sum_a \pi(a|s) q_\pi(s,a) \right], \quad \text{for all } s \in \mathcal{S} && \text{(Exercise 3.15)} \\
&= \sum_a \Big[\nabla \pi(a|s) q_\pi(s,a) + \pi(a|s) \nabla q_\pi(s,a) \Big] && \text{(product rule)} \\
&= \sum_a \Big[\nabla \pi(a|s) q_\pi(s,a) + \pi(a|s) \nabla \sum_{s',r} p(s',r\,|\,s,a) \big(r + v_\pi(s') \big) \Big] \\
& && \text{(Exercise 3.16 and Equation 3.2)} \\
&= \sum_a \Big[\nabla \pi(a|s) q_\pi(s,a) + \pi(a|s) \sum_{s'} p(s'\,|\,s,a) \nabla v_\pi(s') \Big] && \text{(Eq. 3.4)} \\
&= \sum_a \Big[\nabla \pi(a|s) q_\pi(s,a) + \pi(a|s) \sum_{s'} p(s'\,|\,s,a) && \text{(unrolling)} \\
&\qquad \sum_{a'} \big[\nabla \pi(a'|s') q_\pi(s',a') + \pi(a'|s') \sum_{s''} p(s''\,|\,s',a') \nabla v_\pi(s'') \big] \Big] \\
&= \sum_{x \in \mathcal{S}} \sum_{k=0}^{\infty} \Pr(s \to x, k, \pi) \sum_a \nabla \pi(a|x) q_\pi(x,a),
\end{aligned}
$$

after repeated unrolling, where $\Pr(s \to x, k, \pi)$ is the probability of transitioning from state s to state x in k steps under policy π. It is then immediate that

$$
\begin{aligned}
\nabla J(\boldsymbol{\theta}) &= \nabla v_\pi(s_0) \\
&= \sum_s \left(\sum_{k=0}^{\infty} \Pr(s_0 \to s, k, \pi) \right) \sum_a \nabla \pi(a|s) q_\pi(s,a) \\
&= \sum_s \eta(s) \sum_a \nabla \pi(a|s) q_\pi(s,a) && \text{(box page 163)} \\
&= \left(\sum_s \eta(s) \right) \sum_s \frac{\eta(s)}{\sum_s \eta(s)} \sum_a \nabla \pi(a|s) q_\pi(s,a) \\
&\propto \sum_s \mu(s) \sum_a \nabla \pi(a|s) q_\pi(s,a). \qquad \text{Q.E.D.} && \text{(Eq. 9.3)}
\end{aligned}
$$

13.3 REINFORCE: Monte Carlo Policy Gradient

We are now ready for our first policy-gradient learning algorithm. Recall our overall strategy of stochastic gradient ascent (13.1), which requires a way to obtain samples such that the expectation of the sample gradient is proportional to the actual gradient of the performance measure as a function of the parameter. The sample gradients need only be proportional to the gradient because any constant of proportionality can be absorbed into the step size α, which is otherwise arbitrary. The policy gradient theorem gives an exact expression proportional to the gradient; all that is needed is some way of sampling whose expectation equals or approximates this expression. Notice that the right-hand side of the policy gradient theorem is a sum over states weighted by how often the states occur under the target policy π; if π is followed, then states will be encountered in these proportions. Thus

$$\begin{aligned}\nabla J(\boldsymbol{\theta}) &\propto \sum_s \mu(s) \sum_a q_\pi(s,a) \nabla_{\boldsymbol{\theta}} \pi(a|s,\boldsymbol{\theta}), \\ &= \mathbb{E}_\pi\left[\sum_a q_\pi(S_t,a) \nabla_{\boldsymbol{\theta}} \pi(a|S_t,\boldsymbol{\theta})\right].\end{aligned} \tag{13.5}$$

This is good progress, and we would like to carry it further and handle the action in the same way (replacing a with the sample action A_t). The remaining part of the expectation above is a sum over actions; if only each term were weighted by the probability of selecting the actions, that is, according to $\pi(a|S_t,\boldsymbol{\theta})$, then the replacement could be done. We can arrange for this by multiplying and dividing by this probability. Continuing from the previous equation, this gives

$$\begin{aligned}\nabla J(\boldsymbol{\theta}) &= \mathbb{E}_\pi\left[\sum_a \pi(a|S_t,\boldsymbol{\theta}) q_\pi(S_t,a) \frac{\nabla_{\boldsymbol{\theta}} \pi(a|S_t,\boldsymbol{\theta})}{\pi(a|S_t,\boldsymbol{\theta})}\right] \\ &= \mathbb{E}_\pi\left[q_\pi(S_t,A_t) \frac{\nabla_{\boldsymbol{\theta}} \pi(A_t|S_t,\boldsymbol{\theta})}{\pi(A_t|S_t,\boldsymbol{\theta})}\right] && \text{(replacing } a \text{ by the sample } A_t \sim \pi) \\ &= \mathbb{E}_\pi\left[G_t \frac{\nabla_{\boldsymbol{\theta}} \pi(A_t|S_t,\boldsymbol{\theta})}{\pi(A_t|S_t,\boldsymbol{\theta})}\right], && \text{(because } \mathbb{E}_\pi[G_t|S_t,A_t] = q_\pi(S_t,A_t))\end{aligned}$$

where G_t is the return as usual. The final expression in the brackets is exactly what is needed, a quantity that can be sampled on each time step whose expectation is equal to the gradient. Using this sample to instantiate our generic stochastic gradient ascent algorithm (13.1), yields the update

$$\boldsymbol{\theta}_{t+1} \doteq \boldsymbol{\theta}_t + \alpha G_t \frac{\nabla_{\boldsymbol{\theta}} \pi(A_t|S_t,\boldsymbol{\theta}_t)}{\pi(A_t|S_t,\boldsymbol{\theta}_t)}. \tag{13.6}$$

We call this algorithm REINFORCE (after Williams, 1992). Its update has an intuitive appeal. Each increment is proportional to the product of a return G_t and a vector, the gradient of the probability of taking the action actually taken divided by the probability of taking that action. The vector is the direction in parameter space that most increases the probability of repeating the action A_t on future visits to state S_t. The update increases the parameter vector in this direction proportional to the return, and inversely proportional to the action probability. The former makes sense because it causes the parameter to move most in the directions that favor actions that yield the highest return. The latter makes sense because otherwise actions that are selected frequently are at an advantage (the updates will be more often in their direction) and might win out even if they do not yield the highest return.

Note that REINFORCE uses the complete return from time t, which includes all future rewards up until the end of the episode. In this sense REINFORCE is a Monte Carlo algorithm and is well defined only for the episodic case with all updates made in retrospect after the episode is completed (like the Monte Carlo algorithms in Chapter 5). This is shown explicitly in the boxed pseudocode below.

REINFORCE, A Monte-Carlo Policy-Gradient Method (episodic)

Input: a differentiable policy parameterization $\pi(a|s,\boldsymbol{\theta})$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$
Repeat forever:
  Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot,\boldsymbol{\theta})$
  For each step of the episode $t = 0, \ldots, T-1$:
    $G \leftarrow$ return from step t
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha\gamma^t G \nabla_{\boldsymbol{\theta}} \ln \pi(A_t|S_t,\boldsymbol{\theta})$

Figure 13.1 shows the performance of REINFORCE, averaged over 100 runs, on the gridworld from Example 13.1.

The vector $\frac{\nabla_{\boldsymbol{\theta}}\pi(A_t|S_t,\boldsymbol{\theta}_t)}{\pi(A_t|S_t,\boldsymbol{\theta}_t)}$ in the REINFORCE update is the only place the policy parameterization appears in the algorithm. This vector has been given several names and notations in the literature; we will refer to it simply as the *eligibility vector*. The eligibility vector is often written in the compact form $\nabla_{\boldsymbol{\theta}} \ln \pi(A_t|S_t,\boldsymbol{\theta}_t)$, using the identity $\nabla \ln x = \frac{\nabla x}{x}$. This form is used in all the boxed pseudocode in this chapter. In earlier examples in this chapter we considered exponential softmax policies (13.2) with linear action preferences (13.3). For this parameterization, the eligibility vector is

$$\nabla_{\boldsymbol{\theta}} \ln \pi(a|s,\boldsymbol{\theta}) = \mathbf{x}(s,a) - \sum_b \pi(b|s,\boldsymbol{\theta})\mathbf{x}(s,b). \tag{13.7}$$

As a stochastic gradient method, REINFORCE has good theoretical convergence properties. By construction, the expected update over an episode is in the same direction as the performance gradient.[2] This assures an improvement in expected performance for sufficiently small α, and convergence to a

[2]Technically, this is only true if each episode's updates are done *off-line*, meaning they are accumulated on the side during the episode and only used to change $\boldsymbol{\theta}$ by their sum at the episode's end. However, this would probably be a worse algorithm in practice, and its desirable theoretical properties would probably be shared by the algorithm as given (although this has not been proved).

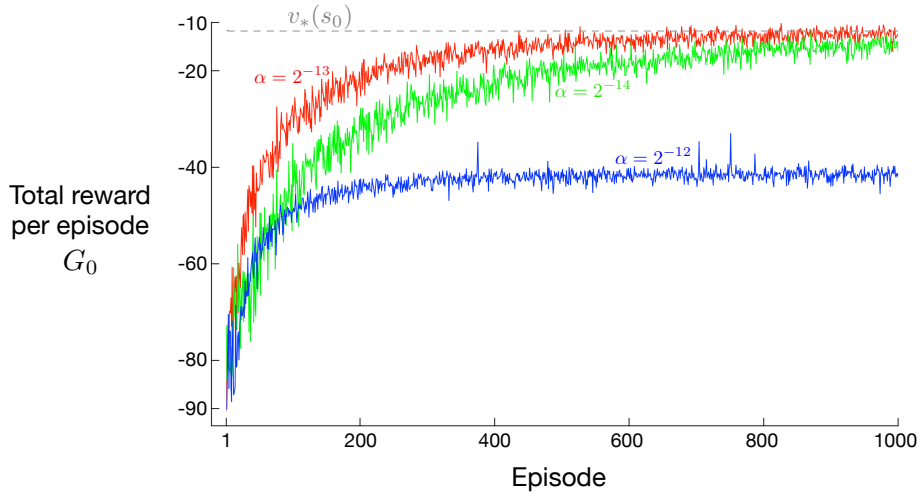


Figure 13.1: REINFORCE on the short-corridor gridworld (Example 13.1). With a good step size, the total reward per episode approaches the optimal value of the start state.

local optimum under standard stochastic approximation conditions for decreasing α. However, as a Monte Carlo method REINFORCE may be of high variance and thus produce slow learning.

Exercise 13.2 Prove (13.7) using the definitions and elementary calculus. □

13.4 REINFORCE with Baseline

The policy gradient theorem (13.5) can be generalized to include a comparison of the action value to an arbitrary *baseline* $b(s)$:

$$\nabla J(\boldsymbol{\theta}) \propto \sum_s \mu(s) \sum_a \Big(q_\pi(s,a) - b(s)\Big) \nabla_{\boldsymbol{\theta}} \pi(a|s,\boldsymbol{\theta}). \tag{13.8}$$

The baseline can be any function, even a random variable, as long as it does not vary with a; the equation remains valid because the subtracted quantity is zero:

$$\sum_a b(s) \nabla_{\boldsymbol{\theta}} \pi(a|s,\boldsymbol{\theta}) \;=\; b(s) \nabla_{\boldsymbol{\theta}} \sum_a \pi(a|s,\boldsymbol{\theta}) \;=\; b(s) \nabla_{\boldsymbol{\theta}} 1 \;=\; 0.$$

The policy gradient theorem with baseline (13.8) can be used to derive an update rule using similar steps as in the previous section. The update rule that we end up with is a new version of REINFORCE that includes a general baseline:

$$\boldsymbol{\theta}_{t+1} \doteq \boldsymbol{\theta}_t + \alpha \Big(G_t - b(S_t)\Big) \frac{\nabla_{\boldsymbol{\theta}} \pi(A_t|S_t,\boldsymbol{\theta}_t)}{\pi(A_t|S_t,\boldsymbol{\theta}_t)}. \tag{13.9}$$

Because the baseline could be uniformly zero, this update is a strict generalization of REINFORCE. In general, the baseline leaves the expected value of the update unchanged, but it can have a large effect on its variance. For example, we saw in Section 2.8 that an analogous baseline can significantly reduce the variance (and thus speed the learning) of gradient bandit algorithms. In the bandit algorithms the baseline was just a number (the average of the rewards seen so far), but for MDPs the baseline should vary with state. In some states all actions have high values and we need a high baseline to differentiate the higher valued actions from the less highly valued ones; in other states all actions will have low values and a low baseline is appropriate.

One natural choice for the baseline is an estimate of the state value, $\hat{v}(S_t,\mathbf{w})$, where $\mathbf{w} \in \mathbb{R}^m$ is a weight vector learned by one of the methods presented in previous chapters. Because REINFORCE is a Monte Carlo method for learning the policy parameter, $\boldsymbol{\theta}$, it seems natural to also use a Monte Carlo method to learn the state-value weights, $\mathbf{w}$. A complete pseudocode algorithm for REINFORCE with baseline is given in the box on the next page using such a learned state-value function as the baseline.

This algorithm has two step sizes, denoted $\alpha^{\boldsymbol{\theta}}$ and $\alpha^{\mathbf{w}}$ (where $\alpha^{\boldsymbol{\theta}}$ is the α in (13.9)). The step size for values (here $\alpha^{\mathbf{w}}$) is relatively easy; in the linear case we have rules of thumb for setting it, such as $\alpha^{\mathbf{w}} = 0.1/\mathbb{E}\big[\|\nabla_{\mathbf{w}} \hat{v}(S_t,\mathbf{w})\|_\mu^2\big]$. It is much less clear how to set the step size $\alpha^{\boldsymbol{\theta}}$ for the policy parameters. It depends on the range of variation of the rewards and on the policy parameterization.

REINFORCE with Baseline (episodic)

Input: a differentiable policy parameterization $\pi(a|s,\boldsymbol{\theta})$
Input: a differentiable state-value parameterization $\hat{v}(s,\mathbf{w})$
Parameters: step sizes $\alpha^{\boldsymbol{\theta}} > 0$, $\alpha^{\mathbf{w}} > 0$

Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ and state-value weights $\mathbf{w} \in \mathbb{R}^{d}$
Repeat forever:
  Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot,\boldsymbol{\theta})$
  For each step of the episode $t = 0, \ldots, T-1$:
    $G_t \leftarrow$ return from step t
    $\delta \leftarrow G_t - \hat{v}(S_t,\mathbf{w})$
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}} \gamma^t \delta \nabla_{\mathbf{w}} \hat{v}(S_t,\mathbf{w})$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} \gamma^t \delta \nabla_{\boldsymbol{\theta}} \ln \pi(A_t|S_t,\boldsymbol{\theta})$

Figure 13.2 compares the behavior of REINFORCE with and without a baseline on the short-corridor gridword (Example 13.1). Here the approximate state-value function used in the baseline is $\hat{v}(s,\mathbf{w}) = w$. That is, $\mathbf{w}$ is a single component, w. The step size used for the baseline was $\beta = 0.1$.

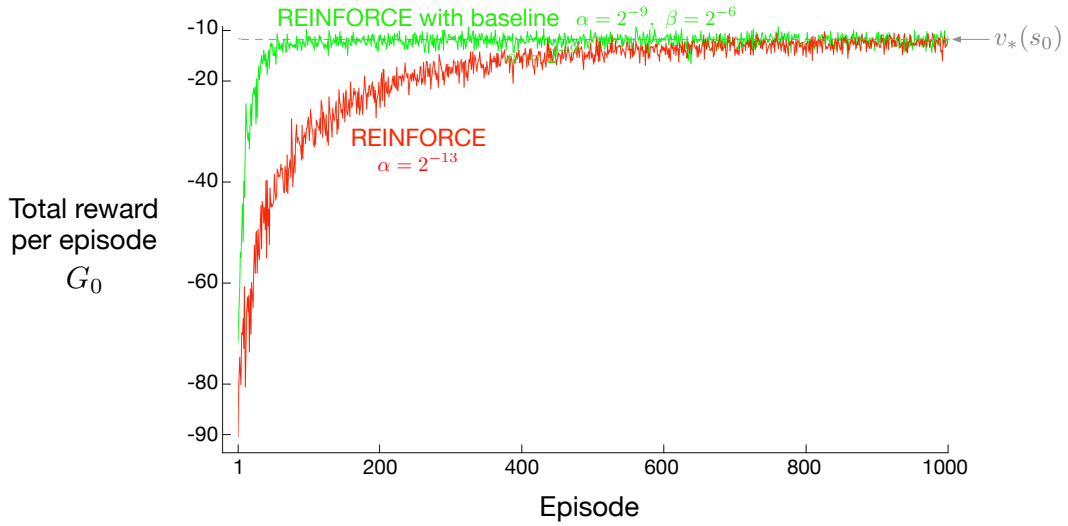


Figure 13.2: Adding a baseline to REINFORCE can make it learn much faster, as illustrated here on the short-corridor gridworld (Example 13.1). The step size used here for plain REINFORCE is that at which it performs best (to the nearest power of two; see Figure 13.1). Each line is an average over 100 independent runs.

13.5 Actor–Critic Methods

Although the REINFORCE-with-baseline method learns both a policy and a state-value function, we do not consider it to be an actor–critic method because its state-value function is used only as a baseline, not as a critic. That is, it is not used for bootstrapping (updating the value estimate for a state from the estimated values of subsequent states), but only as a baseline for the state whose estimate is being updated. This is a useful distinction, for only through bootstrapping do we introduce

bias and an asymptotic dependence on the quality of the function approximation. As we have seen, the bias introduced through bootstrapping and reliance on the state representation is often beneficial because it reduces variance and accelerates learning. REINFORCE with baseline is unbiased and will converge asymptotically to a local minimum, but like all Monte Carlo methods it tends to learn slowly (produce estimates of high variance) and to be inconvenient to implement online or for continuing problems. As we have seen earlier in this book, with temporal-difference methods we can eliminate these inconveniences, and through multi-step methods we can flexibly choose the degree of bootstrapping. In order to gain these advantages in the case of policy gradient methods we use actor–critic methods with a bootstrapping critic.

First consider one-step actor–critic methods, the analog of the TD methods introduced in Chapter 6 such as TD(0), Sarsa(0), and Q-learning. The main appeal of one-step methods is that they are fully online and incremental, yet avoid the complexities of eligibility traces. They are a special case of the eligibility trace methods, and not as general, but easier to understand. One-step actor–critic methods replace the full return of REINFORCE (13.9) with the one-step return (and use a learned state-value function as the baseline) as follows:

$$\boldsymbol{\theta}_{t+1} \doteq \boldsymbol{\theta}_t + \alpha\Big(G_{t:t+1} - \hat{v}(S_t,\mathbf{w})\Big)\frac{\nabla_{\boldsymbol{\theta}}\pi(A_t|S_t,\boldsymbol{\theta}_t)}{\pi(A_t|S_t,\boldsymbol{\theta}_t)} \tag{13.10}$$

$$= \boldsymbol{\theta}_t + \alpha\Big(R_{t+1} + \gamma\hat{v}(S_{t+1},\mathbf{w}) - \hat{v}(S_t,\mathbf{w})\Big)\frac{\nabla_{\boldsymbol{\theta}}\pi(A_t|S_t,\boldsymbol{\theta}_t)}{\pi(A_t|S_t,\boldsymbol{\theta}_t)} \tag{13.11}$$

$$= \boldsymbol{\theta}_t + \alpha\delta_t\frac{\nabla_{\boldsymbol{\theta}}\pi(A_t|S_t,\boldsymbol{\theta}_t)}{\pi(A_t|S_t,\boldsymbol{\theta}_t)}. \tag{13.12}$$

The natural state-value-function learning method to pair with this is semi-gradient TD(0). Pseudocode for the complete algorithm is given in the box below. Note that it is now a fully online, incremental algorithm, with states, actions, and rewards processed as they occur and then never revisited.

One-step Actor–Critic (episodic)

Input: a differentiable policy parameterization $\pi(a|s,\boldsymbol{\theta})$
Input: a differentiable state-value parameterization $\hat{v}(s,\mathbf{w})$
Parameters: step sizes $\alpha^{\boldsymbol{\theta}} > 0$, $\alpha^{\mathbf{w}} > 0$

Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ and state-value weights $\mathbf{w} \in \mathbb{R}^d$
Repeat forever:
  Initialize S (first state of episode)
  $I \leftarrow 1$
  While S is not terminal:
    $A \sim \pi(\cdot|S,\boldsymbol{\theta})$
    Take action A, observe S', R
    $\delta \leftarrow R + \gamma\hat{v}(S',\mathbf{w}) - \hat{v}(S,\mathbf{w})$ (if S' is terminal, then $\hat{v}(S',\mathbf{w}) \doteq 0$)
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}} I\delta\nabla_{\mathbf{w}}\hat{v}(S,\mathbf{w})$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} I\delta\nabla_{\boldsymbol{\theta}}\ln\pi(A|S,\boldsymbol{\theta})$
    $I \leftarrow \gamma I$
    $S \leftarrow S'$

The generalizations to the forward view of multi-step methods and then to a λ-return algorithm are straightforward. The one-step return in (13.10) is merely replaced by $G^{\lambda}_{t:t+k}$ and G^{λ}_t respectively. The backward views are also straightforward, using separate eligibility traces for the actor and critic, each after the patterns in Chapter 12. Pseudocode for the complete algorithm is given in the box below.

Actor–Critic with Eligibility Traces (episodic)

Input: a differentiable policy parameterization $\pi(a|s,\boldsymbol{\theta})$
Input: a differentiable state-value parameterization $\hat{v}(s,\mathbf{w})$
Parameters: trace-decay rates $\lambda^{\boldsymbol{\theta}} \in [0,1]$, $\lambda^{\mathbf{w}} \in [0,1]$; step sizes $\alpha^{\boldsymbol{\theta}} > 0$, $\alpha^{\mathbf{w}} > 0$

Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ and state-value weights $\mathbf{w} \in \mathbb{R}^{d}$
Repeat forever (for each episode):
  Initialize S (first state of episode)
  $\mathbf{z}^{\boldsymbol{\theta}} \leftarrow \mathbf{0}$ (d'-component eligibility trace vector)
  $\mathbf{z}^{\mathbf{w}} \leftarrow \mathbf{0}$ (d-component eligibility trace vector)
  $I \leftarrow 1$
  While S is not terminal (for each time step):
    $A \sim \pi(\cdot|S,\boldsymbol{\theta})$
    Take action A, observe S', R
    $\delta \leftarrow R + \gamma \hat{v}(S',\mathbf{w}) - \hat{v}(S,\mathbf{w})$ (if S' is terminal, then $\hat{v}(S',\mathbf{w}) \doteq 0$)
    $\mathbf{z}^{\mathbf{w}} \leftarrow \gamma \lambda^{\mathbf{w}} \mathbf{z}^{\mathbf{w}} + I \nabla_{\mathbf{w}} \hat{v}(S,\mathbf{w})$
    $\mathbf{z}^{\boldsymbol{\theta}} \leftarrow \gamma \lambda^{\boldsymbol{\theta}} \mathbf{z}^{\boldsymbol{\theta}} + I \nabla_{\boldsymbol{\theta}} \ln \pi(A|S,\boldsymbol{\theta})$
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}} \delta \mathbf{z}^{\mathbf{w}}$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} \delta \mathbf{z}^{\boldsymbol{\theta}}$
    $I \leftarrow \gamma I$
    $S \leftarrow S'$

13.6 Policy Gradient for Continuing Problems

As discussed in Section 10.3, for continuing problems without episode boundaries we need to define performance in terms of the average rate of reward per time step:

$$\begin{aligned} J(\boldsymbol{\theta}) \doteq r(\pi) &\doteq \lim_{h\to\infty} \frac{1}{h} \sum_{t=1}^{h} \mathbb{E}[R_t \mid A_{0:t-1} \sim \pi] \\ &= \lim_{t\to\infty} \mathbb{E}[R_t \mid A_{0:t-1} \sim \pi] \\ &= \sum_s \mu(s) \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a) r, \end{aligned} \tag{13.13}$$

where μ is the steady-state distribution under π, $\mu(s) \doteq \lim_{t\to\infty} \Pr\{S_t = s | A_{0:t} \sim \pi\}$, which is assumed to exist and to be independent of S_0 (an ergodicity assumption). Remember that this is the special distribution under which, if you select actions according to π, you remain in the same distribution:

$$\sum_s \mu(s) \sum_a \pi(a|s,\boldsymbol{\theta}) p(s'|s,a) = \mu(s'). \tag{13.14}$$

We also define values, $v_\pi(s) \doteq \mathbb{E}_\pi[G_t|S_t = s]$ and $q_\pi(s,a) \doteq \mathbb{E}_\pi[G_t|S_t = s, A_t = a]$, with respect to the differential return:

$$G_t \doteq R_{t+1} - \eta(\pi) + R_{t+2} - \eta(\pi) + R_{t+3} - \eta(\pi) + \cdots. \tag{13.15}$$

With these alternate definitions, the policy gradient theorem as given for the episodic case (13.5) remains true for the continuing case. A proof is given in the box on the next page. The forward and backward view equations also remain the same. Complete pseudocode for the actor–critic algorithm in the continuing case (backward view) is given in the box on page 277.

Proof of the Policy Gradient Theorem (continuing case)

The proof of the policy gradient theorem for the continuing case begins similarly to the episodic case. Again we leave it implicit in all cases that π is a function of $\boldsymbol{\theta}$ and that the gradients are with respect to $\boldsymbol{\theta}$. Recall that in the continuing case $J(\boldsymbol{\theta}) = r(\pi)$ (13.13) and that v_π and q_π denote values with respect to the differential return (13.15). The gradient of the state-value function can be written, for any $s \in \mathcal{S}$, as

$$\begin{aligned}
\nabla v_\pi(s) &= \nabla \left[\sum_a \pi(a|s) q_\pi(s,a) \right], \quad \text{for all } s \in \mathcal{S} && \text{(Exercise 3.15)} \\
&= \sum_a \Big[\nabla \pi(a|s) q_\pi(s,a) + \pi(a|s) \nabla q_\pi(s,a) \Big] && \text{(product rule of calculus)} \\
&= \sum_a \Big[\nabla \pi(a|s) q_\pi(s,a) + \pi(a|s) \nabla \sum_{s',r} p(s', r \,|\, s, a) \big(r - r(\boldsymbol{\theta}) + v_\pi(s') \big) \Big] \\
&= \sum_a \Big[\nabla \pi(a|s) q_\pi(s,a) + \pi(a|s) \big[-\nabla r(\boldsymbol{\theta}) + \sum_{s'} p(s' \,|\, s, a) \nabla v_\pi(s') \big] \Big].
\end{aligned}$$

After re-arranging terms, we obtain

$$\nabla r(\boldsymbol{\theta}) = \sum_a \Big[\nabla \pi(a|s) q_\pi(s,a) + \pi(a|s) \sum_{s'} p(s' \,|\, s, a) \nabla v_\pi(s') \Big] - \nabla v_\pi(s).$$

Notice that the left-hand side can be written $\nabla J(\boldsymbol{\theta})$ and that it does not depend on s. Thus the right-hand side does not depend on s either, and we can safely sum it over all $s \in \mathcal{S}$, weighted by $\mu(s)$, without changing it (because $\sum_s \mu(s) = 1$). Thus

$$\begin{aligned}
\nabla J(\boldsymbol{\theta}) &= \sum_s \mu(s) \sum_a \Big[\nabla \pi(a|s) q_\pi(s,a) + \pi(a|s) \sum_{s'} p(s' \,|\, s, a) \nabla v_\pi(s') \Big] - \nabla v_\pi(s) \\
&= \sum_s \mu(s) \sum_a \nabla \pi(a|s) q_\pi(s,a) \\
&\quad + \mu(s) \sum_a \pi(a|s) \sum_{s'} p(s' \,|\, s, a) \nabla v_\pi(s') - \mu(s) \sum_a \nabla v_\pi(s) \\
&= \sum_s \mu(s) \sum_a \nabla \pi(a|s) q_\pi(s,a) \\
&\quad + \sum_{s'} \underbrace{\sum_s \mu(s) \sum_a \pi(a|s) p(s' \,|\, s, a)}_{\mu(s') \text{ (13.14)}} \nabla v_\pi(s') - \sum_s \mu(s) \nabla v_\pi(s) \\
&= \sum_s \mu(s) \sum_a \nabla \pi(a|s) q_\pi(s,a) + \sum_{s'} \mu(s') \nabla v_\pi(s') - \sum_s \mu(s) \nabla v_\pi(s) \\
&= \sum_s \mu(s) \sum_a \nabla \pi(a|s) q_\pi(s,a). \qquad \text{Q.E.D.}
\end{aligned}$$

Actor–Critic with Eligibility Traces (continuing)

Input: a differentiable policy parameterization $\pi(a|s,\boldsymbol{\theta})$
Input: a differentiable state-value parameterization $\hat{v}(s,\mathbf{w})$
Parameters: trace-decay rates $\lambda^{\boldsymbol{\theta}} \in [0,1]$, $\lambda^{\mathbf{w}} \in [0,1]$; step sizes $\alpha^{\boldsymbol{\theta}} > 0$, $\alpha^{\mathbf{w}} > 0$, $\eta > 0$

$\mathbf{z}^{\boldsymbol{\theta}} \leftarrow \mathbf{0}$ (d'-component eligibility trace vector)
$\mathbf{z}^{\mathbf{w}} \leftarrow \mathbf{0}$ (d-component eligibility trace vector)
Initialize $\bar{R} \in \mathbb{R}$ (e.g., to 0)
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ and state-value weights $\mathbf{w} \in \mathbb{R}^{d}$ (e.g., to $\mathbf{0}$)
Initialize $S \in \mathcal{S}$ (e.g., to s_0)
Repeat forever:
  $A \sim \pi(\cdot|S,\boldsymbol{\theta})$
  Take action A, observe S', R
  $\delta \leftarrow R - \bar{R} + \hat{v}(S',\mathbf{w}) - \hat{v}(S,\mathbf{w})$ (if S' is terminal, then $\hat{v}(S',\mathbf{w}) \doteq 0$)
  $\bar{R} \leftarrow \bar{R} + \eta\delta$
  $\mathbf{z}^{\mathbf{w}} \leftarrow \lambda^{\mathbf{w}}\mathbf{z}^{\mathbf{w}} + \nabla_{\mathbf{w}}\hat{v}(S,\mathbf{w})$
  $\mathbf{z}^{\boldsymbol{\theta}} \leftarrow \lambda^{\boldsymbol{\theta}}\mathbf{z}^{\boldsymbol{\theta}} + \nabla_{\boldsymbol{\theta}} \ln \pi(A|S,\boldsymbol{\theta})$
  $\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}}\delta\mathbf{z}^{\mathbf{w}}$
  $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}}\delta\mathbf{z}^{\boldsymbol{\theta}}$
  $S \leftarrow S'$

13.7 Policy Parameterization for Continuous Actions

Policy-based methods offer practical ways of dealing with large actions spaces, even continuous spaces with an infinite number of actions. Instead of computing learned probabilities for each of the many actions, we instead learn statistics of the probability distribution. For example, the action set might be the real numbers, with actions chosen from a normal (Gaussian) distribution.

The probability density function for the normal distribution is conventionally written

$$p(x) \doteq \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right), \tag{13.16}$$

where μ and σ here are the mean and standard deviation of the normal distribution, and of course π here is just the number $\pi \approx 3.14159$. The probability density functions for several different means and standard deviations are shown in Figure 13.3. The value $p(x)$ is the *density* of the probability at x, not the probability. It can be greater than 1; it is the total area under $p(x)$ that must sum to 1. In general, one can take the integral under $p(x)$ for any range of x values to get the probability of x falling within that range.

To produce a policy parameterization, the policy can be defined as the normal probability density over a real-valued scalar action, with mean and standard deviation given by parametric function approximators that depend on the state. That is,

$$\pi(a|s,\boldsymbol{\theta}) \doteq \frac{1}{\sigma(s,\boldsymbol{\theta})\sqrt{2\pi}} \exp\left(-\frac{(a-\mu(s,\boldsymbol{\theta}))^2}{2\sigma(s,\boldsymbol{\theta})^2}\right), \tag{13.17}$$

where $\mu : \mathcal{S} \times \mathbb{R}^{d'} \to \mathbb{R}$ and $\sigma : \mathcal{S} \times \mathbb{R}^{d'} \to \mathbb{R}^+$ are two parameterized function approximators. To complete the example we need only give a form for these approximators. For this we divide the policy's parameter vector into two parts, $\boldsymbol{\theta} = [\boldsymbol{\theta}_\mu, \boldsymbol{\theta}_\sigma]^\top$, one part to be used for the approximation of the mean and one part for the approximation of the standard deviation. The mean can be approximated as a

$$p(x) \doteq \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$$

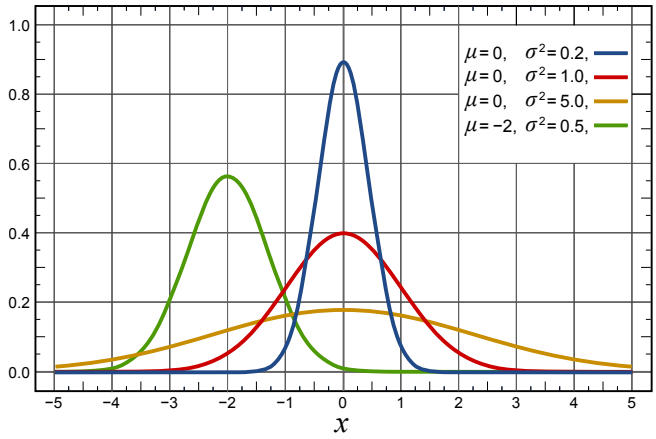


Figure 13.3: The probability density function of the normal distribution for different means and variances.

linear function. The standard deviation must always be positive and is better approximated as the exponential of a linear function. Thus

$$\mu(s, \boldsymbol{\theta}) \doteq \boldsymbol{\theta}_\mu{}^\top \mathbf{x}(s) \quad \text{and} \quad \sigma(s, \boldsymbol{\theta}) \doteq \exp\Big(\boldsymbol{\theta}_\sigma{}^\top \mathbf{x}(s)\Big), \tag{13.18}$$

where $\mathbf{x}(s)$ is a state feature vector perhaps constructed by one of the methods described in Chapter 9. With these definitions, all the algorithms described in the rest of this chapter can be applied to learn to select real-valued actions.

Exercise 13.3 A *Bernoulli-logistic unit* is a stochastic neuron-like unit used in some artificial neural networks (Section 9.6). Its input at time t is a feature vector $\mathbf{x}(S_t)$; its output, A_t, is a random variable having two values, 0 and 1, with $\Pr\{A_t = 1\} = P_t$ and $\Pr\{A_t = 0\} = 1 - P_t$ (the Bernoulli distribution). Let $h(s, 0, \boldsymbol{\theta})$ and $h(s, 1, \boldsymbol{\theta})$ be the preferences in state s for the unit's two actions given policy parameter $\boldsymbol{\theta}$. Assume that the difference between the preferences is given by a weighted sum of the unit's input vector, that is, assume that $h(s, 1, \boldsymbol{\theta}) - h(s, 0, \boldsymbol{\theta}) = \boldsymbol{\theta}^\top \mathbf{x}(s)$, where $\boldsymbol{\theta}$ is the unit's weight vector.

(a) Show that if the exponential softmax distribution (13.2) is used to convert preferences to policies, then $P_t = \pi(1|S_t, \boldsymbol{\theta}_t) = 1/(1 + \exp(-\boldsymbol{\theta}_t^\top \mathbf{x}(S_t)))$ (the logistic function).

(b) What is the Monte-Carlo REINFORCE update of $\boldsymbol{\theta}_t$ to $\boldsymbol{\theta}_{t+1}$ upon receipt of return G_t?

(c) Express the eligibility $\nabla_{\boldsymbol{\theta}} \ln \pi(a|s, \boldsymbol{\theta})$ for a Bernoulli-logistic unit, in terms of a, $\mathbf{x}(s)$, and $\pi(a|s, \boldsymbol{\theta})$ by calculating the gradient.

Hint: separately for each action compute the derivative of the logorithm first with respect to $P_t = \pi(a|s, \boldsymbol{\theta}_t)$, combine the two results into one expression that depends on a and P_t, and then use the chain rule, noting that the derivative of the logistic function $f(x)$ is $f(x)(1 - f(x))$. □

13.8 Summary

Prior to this chapter, this book focused on *action-value methods*—meaning methods that learn action values and then use them to determine action selections. In this chapter, on the other hand, we considered methods that learn a parameterized policy that enables actions to be taken without consulting action-value estimates—though action-value estimates may still be learned and used to update the policy parameter. In particular, we have considered *policy-gradient methods*—meaning methods that update the policy parameter on each step in the direction of an estimate of the gradient of performance with respect to the policy parameter.

Methods that learn and store a policy parameter have many advantages. They can learn specific probabilities for taking the actions. They can learn appropriate levels of exploration and approach deterministic policies asymptotically. They can naturally handle continuous state spaces. All these things are easy for policy-based methods but awkward or impossible for ε-greedy methods and for action-value methods in general. In addition, on some problems the policy is just simpler to represent parametrically than the value function; these problems are more suited to parameterized policy methods.

Parameterized policy methods also have an important theoretical advantage over action-value methods in the form of the *policy gradient theorem*, which gives an exact formula for how performance is affected by the policy parameter that does not involve derivatives of the state distribution. This theorem provides a theoretical foundation for all policy gradient methods.

The REINFORCE method follows directly from the policy gradient theorem. Adding a state-value function as a *baseline* reduces REINFORCE's variance without introducing bias. Using the state-value function for bootstrapping introduces bias but is often desirable for the same reason that bootstrapping TD methods are often superior to Monte Carlo methods (substantially reduced variance). The state-value function assigns credit to—critizes—the policy's action selections, and accordingly the former is termed the *critic* and the latter the *actor*, and these overall methods are sometimes termed *actor–critic* methods.

Overall, policy-gradient methods provide a significantly different set of strengths, and weaknesses than action-value methods. Today they are less well understood in some respects, but a subject of excitement and ongoing research.

Bibliographical and Historical Remarks

Methods that we now see as related to policy gradients were actually some of the earliest to be studied in reinforcement learning (Witten, 1977; Barto, Sutton, and Anderson, 1983; Sutton, 1984; Williams, 1987, 1992) and in predecessor fields (Phansalkar and Thathachar, 1995). They were largely supplanted in the 1990s by the action-value methods that are the focus of the other chapters of this book. In recent years, however, attention has returned to actor–critic methods and to policy-gradient methods in general. Among the further developments beyond what we cover here are natural-gradient methods (Amari, 1998; Kakade, 2002, Peters, Vijayakumar and Schaal, 2005; Peters and Schall, 2008; Park, Kim and Kang, 2005; Bhatnagar, Sutton, Ghavamzadeh and Lee, 2009; see Grondman, Busoniu, Lopes and Babuska, 2012), and deterministic policy gradient (Silver et al., 2014). Major applications include acrobatic helicopter autopilots and AlphaGo (see Section 16.6).

Our presentation in this chapter is based primarily on that by Sutton, McAllester, Singh, and Mansour (2000, see also Sutton, Singh, and McAllester, 2000), who introduced the term "policy gradient methods." A useful overview is provided by Bhatnagar et al. (2003). One of the earliest related works is by Aleksandrov, Sysoyev, and Shemeneva (1968).

13.1 Example 13.1 and the results with it in this chapter were developed with Eric Graves.

13.2 The policy gradient theorem here and on page 276 was first obtained by Marbach and Tsitsiklis (1998, 2001) and then independently by Sutton et al. (2000). A similar expression was obtained by Cao and Chen (1997). Other early results are due to Konda and Tsitsiklis (2000, 2003) and Baxter and Bartlett (2000). Some additional results are developed by Sutton

13.3 REINFORCE is due to Williams (1987, 1992).

Phansalkar and Thathachar (1995) proved both local and global convergence theorems for modified versions of REINFORCE algorithms.

13.4 The baseline was introduced in Williams's (1987, 1992) original work. Greensmith, Bartlett, and Baxter (2004) analyzed an arguably better baseline (see Dick, 2015).

13.5–6 Actor–critic methods were among the earliest to be investigated in reinforcement learning (Witten, 1977; Barto, Sutton, and Anderson, 1983; Sutton, 1984). The algorithms presented here and in Section 13.6 are based on the work of Degris, White, and Sutton (2012), who also introduced the study of off-policy policy-gradient methods.

13.7 The first to show how continuous actions could be handled this way appears to have been Williams (1987, 1992).

Part III: Looking Deeper

In this last part of the book we look beyond the standard reinforcement learning ideas presented in the first two parts of the book to briefly survey their relationships with psychology and neuroscience, a sampling of reinforcement learning applications, and some of the active frontiers for future reinforcement learning research.

Chapter 14

Psychology

In previous chapters we developed ideas for algorithms based on computational considerations alone. In this chapter we look at some of these algorithms from another perspective: the perspective of psychology and its study of how animals learn. The goals of this chapter are, first, to discuss ways that reinforcement learning ideas and algorithms correspond to what psychologists have discovered about animal learning, and second, to explain the influence reinforcement learning is having on the study of animal learning. The clear formalism provided by reinforcement learning that systemizes tasks, returns, and algorithms is proving to be enormously useful in making sense of experimental data, in suggesting new kinds of experiments, and in pointing to factors that may be critical to manipulate and to measure. The idea of optimizing return over the long term that is at the core of reinforcement learning is contributing to our understanding of otherwise puzzling features of animal learning and behavior.

Some of the correspondences between reinforcement learning and psychological theories are not surprising because the development of reinforcement learning drew inspiration from psychological learning theories. However, as developed in this book, reinforcement learning explores idealized situations from the perspective of an artificial intelligence researcher or engineer, with the goal of solving computational problems with efficient algorithms, rather than to to replicate or explain in detail how animals learn. As a result, some of the correspondences we describe connect ideas that arose independently in their respective fields. We believe these points of contact are specially meaningful because they expose computational principles important to learning, whether it is learning by artificial or by natural systems.

For the most part, we describe correspondences between reinforcement learning and learning theories developed to explain how animals like rats, pigeons, and rabbits learn in controlled laboratory experiments. Thousands of these experiments were conducted throughout the 20th century, and many are still being conducted today. Although sometimes dismissed as irrelevant to wider issues in psychology, these experiments probe subtle properties of animal learning, often motivated by precise theoretical questions. As psychology shifted its focus to more cognitive aspects of behavior, that is, to mental processes such as thought and reasoning, animal learning experiments came to play less of a role in psychology than they once did. But this experimentation led to the discovery of learning principles that are elemental and widespread throughout the animal kingdom, principles that should not be neglected in designing artificial learning systems. In addition, as we shall see, some aspects of cognitive processing connect naturally to the computational perspective provided by reinforcement learning.

This chapter's final section includes references relevant to the connections we discuss as well as to connections we neglect. We hope this chapter encourages readers to probe all of these connections more deeply. Also included in this final section is a discussion of how the terminology used in reinforcement learning relates to that of psychology. Many of the terms and phrases used in reinforcement learning are

borrowed from animal learning theories, but the computational/engineering meanings of these terms and phrases do not always coincide with their meanings in psychology.

14.1 Prediction and Control

The algorithms we describe in this book fall into two broad categories: algorithms for *prediction* and algorithms for *control*. These categories arise naturally in solution methods for the reinforcement learning problem presented in Chapter 3. In many ways these categories respectively correspond to categories of learning extensively studied by psychologists: *classical*, or *Pavlovian*, *conditioning* and *instrumental*, or *operant*, *conditioning*. These correspondences are not completely accidental because of psychology's influence on reinforcement learning, but they are nevertheless striking because they connect ideas arising from different objectives.

The prediction algorithms presented in this book estimate quantities that depend on how features of an agent's environment are expected to unfold over the future. We specifically focus on estimating the amount of reward an agent can expect to receive over the future while it interacts with its environment. In this role, prediction algorithms are *policy evaluation algorithms*, which are integral components of algorithms for improving policies. But prediction algorithms are not limited to predicting future reward; they can predict any feature of the environment (see, for example, Modayil, White, and Sutton, 2014). The correspondence between prediction algorithms and classical conditioning rests on their common property of predicting upcoming stimuli, whether or not those stimuli are rewarding (or punishing).

The situation in an instrumental, or operant, conditioning experiment is different. Here, the experimental apparatus is set up so that an animal is given something it likes (a reward) or something it dislikes (a penalty) depending on what the animal did. The animal learns to increase its tendency to produce rewarded behavior and to decrease its tendency to produce penalized behavior. The reinforcing stimulus is said to be *contingent* on the animal's behavior, whereas in classical conditioning it is not (although it is difficult to remove all behavior contingencies in a classical conditioning experiment). Instrumental conditioning experiments are like those that inspired Thorndike's Law of Effect that we briefly discuss in Chapter 1. *Control* is at the core of this form of learning, which corresponds to the operation of reinforcement learning's policy-improvement algorithms.[1]

Thinking of classical conditioning in terms of prediction, and instrumental conditioning in terms of control, is a starting point for connecting our computational view of reinforcement learning to animal learning, but in reality, the situation is more complicated than this. There is more to classical conditioning than prediction; it also involves action, and so is a mode of control, sometimes called *Pavlovian control*. Further, classical and instrumental conditioning interact in interesting ways, with both sorts of learning likely being engaged in most experimental situations. Despite these complications, aligning the classical/instrumental distinction with the prediction/control distinction is a convenient first approximation in connecting reinforcement learning to animal learning.

In psychology, the term reinforcement is used to describe learning in both classical and instrumental conditioning. Originally referring only to the strengthening of a pattern of behavior, it is frequently also used for the weakening of a pattern of behavior. A stimulus considered to be the cause of the change in behavior is called a reinforcer, wether or not it is contingent on the animal's previous behavior. At the end of this chapter we discuss this terminology in more detail and how it relates to terminology used in machine learning.

[1] What control means for us is different from what it typically means in animal learning theories; there the environment controls the agent instead of the other way around. See our comments on terminology at the end of this chapter.

14.2 Classical Conditioning

While studying the activity of the digestive system, the celebrated Russian physiologist Ivan Pavlov found that an animal's innate responses to certain triggering stimuli can come to be triggered by other stimuli that are quite unrelated to the inborn triggers. His experimental subjects were dogs that had undergone minor surgery to allow the intensity of their salivary reflex to be accurately measured. In one case he describes, the dog did not salivate under most circumstances, but about 5 seconds after being presented with food it produced about six drops of saliva over the next several seconds. After several repetitions of presenting another stimulus, one not related to food, in this case the sound of a metronome, shortly before the introduction of food, the dog salivated in response to the sound of the metronome in the same way it did to the food. "The activity of the salivary gland has thus been called into play by impulses of sound—a stimulus quite alien to food" (Pavlov, 1927, p. 22). Summarizing the significance of this finding, Pavlov wrote:

> It is pretty evident that under natural conditions the normal animal must respond not only to stimuli which themselves bring immediate benefit or harm, but also to other physical or chemical agencies—waves of sound, light, and the like—which in themselves only *signal* the approach of these stimuli; though it is not the sight and sound of the beast of prey which is in itself harmful to the smaller animal, but its teeth and claws. (Pavlov, 1927, p. 14)

Connecting new stimuli to innate reflexes in this way is now called classical, or Pavlovian, conditioning. Pavlov (or more exactly, his translators) called inborn responses (e.g., salivation in his demonstration described above) "unconditioned responses" (URs), their natural triggering stimuli (e.g., food) "unconditioned stimuli" (USs), and new responses triggered by predictive stimuli (e.g., here also salivation) "conditioned responses" (CRs). A stimulus that is initially neutral, meaning that it does not normally elicit strong responses (e.g., the metronome sound), becomes a "conditioned stimulus" (CS) as the animal learns that it predicts the US and so comes to produce a CR in response to the CS. These terms are still used in describing classical conditioning experiments (though better translations would have been "conditional" and "unconditional" instead of conditioned and unconditioned). The US is called a reinforcer because it reinforces producing a CR in response to the CS.

Figure 14.1 shows the arrangement of stimuli in two types of classical conditioning experiments: in delay conditioning, the CS extends throughout the interstimulus interval, or ISI, which is the time interval between the CS onset and the US onset (with the CS ending when the US ends in a common version shown here). In trace conditioning, the US begins after the CS ends, and the time interval between CS offset and US onset is called the trace interval.

The salivation of Pavlov's dogs to the sound of a metronome is just one example of classical conditioning, which has been intensively studied across many response systems of many species of animals. URs are often preparatory in some way, like the salivation of Pavlov's dog, or protective in some way, like an eye blink in response to something irritating to the eye, or freezing in response to seeing a predator. Experiencing the CS-US predictive relationship over a series of trials causes the animal to learn that the CS predicts the US so that the animal can respond to the CS with a CR that prepares the animal for, or protects it from, the predicted US. Some CRs are similar to the UR but begin earlier and differ in ways that increase their effectiveness. In one intensively studied type of experiment, for example, a tone CS reliably predicts a puff of air (the US) to a rabbit's eye, triggering a UR consisting of the closure of a protective inner eyelid called the nictitating membrane. After one or more trials, the tone comes to trigger a CR consisting of membrane closure that begins before the air puff and eventually becomes timed so that peak closure occurs just when the air puff is likely to occur. This CR, being initiated in anticipation of the air puff and appropriately timed, offers better protection than simply initiating closure as a reaction to the irritating US. The ability to act in anticipation of important events by learning about predictive relationships among stimuli is so beneficial that it is widely present across the animal kingdom.

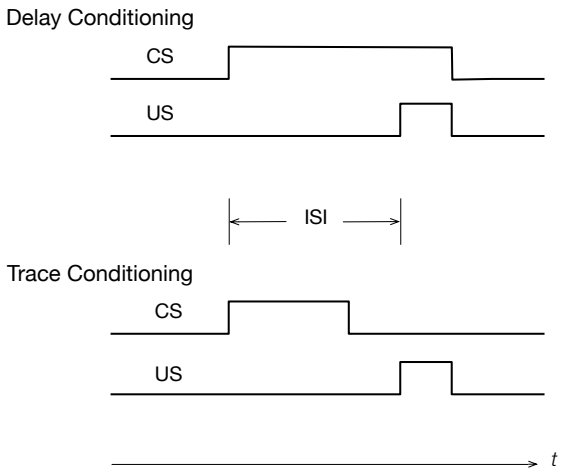


Figure 14.1: Arrangement of stimuli in two types of classical conditioning experiments. In delay conditioning, the CS extends throughout the interstimulus interval, or ISI, which is the time interval between the CS onset and the US onset (often with the CS and US ending at the same time as shown here). In trace conditioning, there is a time interval, called the trace interval, between CS offset and US onset.

14.2.1 Blocking and Higher-order Conditioning

Many interesting properties of classical conditioning have been observed in experiments. Beyond the anticipatory nature of CRs, two widely observed properties figured prominently in the development of classical conditioning models: *blocking* and *higher-order conditioning*. Blocking occurs when an animal fails to learn a CR when a potential CS in presented along with another CS that had been used previously to condition the animal to produce that CR. For example, in the first stage of a blocking experiment involving rabbit nictitating membrane conditioning, a rabbit is first conditioned with a tone CS and an air puff US to produce the CR of closing its nictitating membrane in anticipation of the air puff. The experiment's second stage consists of additional trials in which a second stimulus, say a light, is added to the tone to form a compound tone/light CS followed by the same air puff US. In the experiment's third phase, the second stimulus alone—the light—is presented to the rabbit to see if the rabbit has learned to respond to it with a CR. It turns out that the rabbit produces very few, or no, CRs in response to the light: learning to the light had been *blocked* by the previous learning to the tone.[2] Blocking results like this challenged the idea that conditioning depends only on simple temporal contiguity, that is, that a necessary and sufficient condition for conditioning is that a US frequently follows a CS closely in time. In the next section we describe the *Rescorla–Wagner model* (Rescorla and Wagner, 1972) that offered an influential explanation for blocking.

Higher-order conditioning occurs when a previously-conditioned CS acts as a US in conditioning another initially neutral stimulus. Pavlov described an experiment in which his assistant first conditioned a dog to salivate to the sound of a metronome that predicted a food US, as described above. After this stage of conditioning, a number of trials were conducted in which a black square, to which the dog was initially indifferent, was placed in the dog's line of vision followed by the sound of the metronome—and this was *not* followed by food. In just ten trials, the dog began to salivate merely upon seeing the black square, despite the fact that the sight of it had never been followed by food. The sound of

[2]Comparison with a control group is necessary to show that the previous conditioning to the tone is responsible for blocking learning to the light. This is done by trials with the tone/light CS but with no prior conditioning to the tone. Learning to the light in this case is unimpaired. Moore and Schmajuk (2008) give a full account of this procedure.

the metronome itself acted as a US in conditioning a salivation CR to the black square CS. This was second-order conditioning. If the black square had been used as a US to establish salivation CRs to another otherwise neutral CS, it would have been third-order conditioning, and so on. Higher-order conditioning is difficult to demonstrate, especially above the second order, in part because a higher-order reinforcer loses its reinforcing value due to not being repeatedly followed by the original US during higher-order conditioning trials. But under the right conditions, such as intermixing first-order trials with higher-order trials or by providing a general energizing stimulus, higher-order conditioning beyond the second order can be demonstrated. As we describe below, the *TD model of classical conditioning* uses the bootstrapping idea that is central to our approach to extend the Rescorla–Wagner model's account of blocking to include both the anticipatory nature of CRs and higher-order conditioning.

Higher-order instrumental conditioning occurs as well. In this case, a stimulus that consistently predicts primary reinforcement becomes a reinforcer itself, where reinforcement is primary if its rewarding or penalizing quality has been built into the animal by evolution. The predicting stimulus becomes a *secondary reinforcer*, or more generally, a *higher-order* or *conditioned reinforcer*—the latter being a better term when the predicted reinforcing stimulus is itself a secondary, or an even higher-order, reinforcer. A conditioned reinforcer delivers *conditioned reinforcement*: conditioned reward or conditioned penalty. Conditioned reinforcement acts like primary reinforcement in increasing an animal's tendency to produce behavior that leads to conditioned reward, and to decrease an animal's tendency to produce behavior that leads to conditioned penalty. (See our comments at the end of this chapter that explain how our terminology sometimes differs, as it does here, from terminology used in psychology.)

Conditioned reinforcement is a key phenomenon that explains, for instance, why we work for the conditioned reinforcer money, whose worth derives solely from what is predicted by having it. In actor–critic methods described in Section 13.5 (and discussed in the context of neuroscience in Sections 15.7 and 15.8), the critic uses a TD method to evaluate the actor's policy, and its value estimates provide conditioned reinforcement to the actor, allowing the actor to improve its policy. This analog of higher-order instrumental conditioning helps address the credit-assignment problem mentioned in Section 1.7 because the critic gives moment-by-moment reinforcement to the actor when the primary reward signal is delayed. We discuss this more below in Section 14.4.

14.2.2 The Rescorla–Wagner Model

Rescorla and Wagner created their model mainly to account for blocking. The core idea of the Rescorla–Wagner model is that an animal only learns when events violate its expectations, in other words, only when the animal is surprised (although without necessarily implying any *conscious* expectation or emotion). We first present Rescorla and Wagner's model using their terminology and notation before shifting to the terminology and notation we use to describe the TD model.

Here is how Rescorla and Wagner described their model. The model adjusts the "associative strength" of each component stimulus of a compound CS, which is a number representing how strongly or reliably that component is predictive of a US. When a compound CS consisting of several component stimuli is presented in a classical conditioning trial, the associative strength of each component stimulus changes in a way that depends on an associative strength associated with the entire stimulus compound, called the "aggregate associative strength," and not just on the associative strength of each component itself.

Rescorla and Wagner considered a compound CS AX, consisting of component stimuli A and X, where the animal may have already experienced stimulus A, and stimulus X might be new to the animal. Let V_{A}, V_{X}, and V_{AX} respectively denote the associative strengths of stimuli A, X, and the compound AX. Suppose that on a trial the compound CS AX is followed by a US, which we label stimulus Y. Then the associative strengths of the stimulus components change according to these expressions:

$$\begin{aligned}\Delta V_{\mathrm{A}} &= \alpha_{\mathrm{A}}\beta_{\mathrm{Y}}(R_{\mathrm{Y}} - V_{\mathrm{AX}}) \\ \Delta V_{\mathrm{X}} &= \alpha_{\mathrm{X}}\beta_{\mathrm{Y}}(R_{\mathrm{Y}} - V_{\mathrm{AX}}),\end{aligned}$$

where $\alpha_{\text{A}}\beta_{\text{Y}}$ and $\alpha_{\text{X}}\beta_{\text{Y}}$ are the step-size parameters, which depend on the identities of the CS components and the US, and R_{Y} is the asymptotic level of associative strength that the US Y can support. (Rescorla and Wagner used λ here instead of R, but we use R to avoid confusion with our use of λ and because we usually think of this as the magnitude of a reward signal, with the caveat that the US in classical conditioning is not necessarily rewarding or penalizing.) A key assumption of the model is that the aggregate associative strength V_{AX} is equal to $V_{\text{A}} + V_{\text{X}}$. The associative strengths as changed by these Δs become the associative strengths at the beginning of the next trial.

To be complete, the model needs a response-generation mechanism, which is a way of mapping values of Vs to CRs. Since this mapping would depend on details of the experimental situation, Rescorla and Wagner did not specify a mapping but simply assumed that larger Vs would produce stronger or more likely CRs, and that negative Vs would mean that there would be no CRs.

The Rescorla–Wagner model accounts for the acquisition of CRs in a way that explains blocking. As long as the aggregate associative strength, V_{AX}, of the stimulus compound is below the asymptotic level of associative strength, R_{Y}, that the US Y can support, the prediction error $R_{\text{Y}} - V_{\text{AX}}$ is positive. This means that over successive trials the associative strengths V_{A} and V_{X} of the component stimuli increase until the aggregate associative strength V_{AX} equals R_{Y}, at which point the associative strengths stop changing (unless the US changes). When a new component is added to a compound CS to which the animal has already been conditioned, further conditioning with the augmented compound produces little or no increase in the associative strength of the added CS component because the error has already been reduced to zero, or to a low value. The occurrence of the US is already predicted nearly perfectly, so little or no error—or surprise—is introduced by the new CS component. Prior learning blocks learning to the new component.

To transition from Rescorla and Wagner's model to the TD model of classical conditioning (which we just call the TD model), we first recast their model in terms of the concepts that we are using throughout this book. Specifically, we match the notation we use for learning with linear function approximation (Section 9.4), and we think of the conditioning process as one of learning to predict the "magnitude of the US" on a trial on the basis of the compound CS presented on that trial, where the magnitude of a US Y is the R_{Y} of the Rescorla–Wagner model as given above. We also introduce states. Because the Rescorla–Wagner model is a *trial-level* model, meaning that it deals with how associative strengths change from trial to trial without considering any details about what happens within and between trials, we do not have to consider how states change during a trial until we present the full TD model in the following section. Instead, here we simply think of a state as a way of labeling a trial in terms of the collection of component CSs that are present on the trial.

Therefore, assume that trial-type, or state, s is described by a real-valued vector of features $\mathbf{x}(s) = (x_1(s), x_2(s), \ldots, x_d(s))^\top$ where $x_i(s) = 1$ if CS_i, the i^{th} component of a compound CS, is present on the trial and 0 otherwise. Then if the d-dimensional vector of associative strengths is $\mathbf{w}$, the aggregate associative strength for trial-type s is

$$\hat{v}(s,\mathbf{w}) = \mathbf{w}^\top \mathbf{x}(s). \tag{14.1}$$

This corresponds to a *value estimate* in reinforcement learning, and we think of it as the *US prediction*.

Now temporally let t denote the number of a complete trial and not its usual meaning as a time step (we revert to t's usual meaning when we extend this to the TD model below), and assume that S_t is the state corresponding to trial t. Conditioning trial t updates the associative strength vector $\mathbf{w}_t$ to $\mathbf{w}_{t+1}$ as follows:

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha\,\delta_t\,\mathbf{x}(S_t), \tag{14.2}$$

where α is the step-size parameter, and—because here we are describing the Rescorla–Wagner model—δ_t is the *prediction error*

$$\delta_t = R_t - \hat{v}(S_t,\mathbf{w}_t). \tag{14.3}$$

R_t is the target of the prediction on trial t, that is, the magnitude of the US, or in Rescorla and Wagner's terms, the associative strength that the US on the trial can support. Note that because of the factor $\mathbf{x}(S_t)$ in (14.2), only the associative strengths of CS components present on a trial are adjusted as a result of that trial. You can think of the prediction error as a measure of surprise, and the aggregate associative strength as the animal's expectation that is violated when it does not match the target US magnitude.

From the perspective of machine learning, the Rescorla–Wagner model is an error-correction supervised learning rule. It is essentially the same as the Least Mean Square (LMS), or Widrow-Hoff, learning rule (Widrow and Hoff, 1960) that finds the weights—here the associative strengths—that make the average of the squares of all the errors as close to zero as possible. It is a "curve-fitting," or regression, algorithm that is widely used in engineering and scientific applications (see Section 9.4).[3]

The Rescorla–Wagner model was very influential in the history of animal learning theory because it showed that a "mechanistic" theory could account for the main facts about blocking without resorting to more complex cognitive theories involving, for example, an animal's explicit recognition that another stimulus component had been added and then scanning its short-term memory backward to reassess the predictive relationships involving the US. The Rescorla–Wagner model showed how traditional contiguity theories of conditioning—that temporal contiguity of stimuli was a necessary and sufficient condition for learning—could be adjusted in a simple way to account for blocking (Moore and Schmajuk, 2008).

The Rescorla–Wagner model provides a simple account of blocking and some other features of classical conditioning, but it is not a complete or perfect model of classical conditioning. Different ideas account for a variety of other observed effects, and progress is still being made toward understanding the many subtleties of classical conditioning. The TD model, which we describe next, though also not a complete or perfect model model of classical conditioning, extends the Rescorla–Wagner model to address how within-trial and between-trial timing relationships among stimuli can influence learning and how higher-order conditioning might arise.

14.2.3 The TD Model

The TD model is a *real-time* model, as opposed to a trial-level model like the Rescorla–Wagner model. A single step t in the our formulation of Rescorla and Wagner's model above represents an entire conditioning trial. The model does not apply to details about what happens during the time a trial is taking place, or what might happen between trials. Within each trial an animal might experience various stimuli whose onsets occur at particular times and that have particular durations. These timing relationships strongly influence learning. The Rescorla–Wagner model also does not include a mechanism for higher-order conditioning, whereas for the TD model, higher-order conditioning is a natural consequence of the bootstrapping idea that is at the base of TD algorithms.

To describe the TD model we begin with the formulation of the Rescorla–Wagner model above, but t now labels time steps within or between trials instead of complete trials. Think of the time between t and $t+1$ as a small time interval, say .01 second, and think of a trial as a sequences of states, one associated with each time step, where the state at step t now represents details of how stimuli are represented at t instead of just a label for the CS components present on a trial. In fact, we can completely abandon the idea of trials. From the point of view of the animal, a trial is just a fragment of its continuing experience interacting with its world. Following our usual view of an agent interacting with its environment, imagine that the animal is experiencing an endless sequence of states s, each represented by a feature vector $\mathbf{x}(s)$. That said, it is still often convenient to refer to trials as fragments

[3]The only differences between the LMS rule and the Rescorla–Wagner model are that for LMS the input vectors $\mathbf{x}_t$ can have any real numbers as components, and—at least in the simplest version of the LMS rule—the step-size parameter α does not depend on the input vector or the identity of the stimulus setting the prediction target.

of time during which patterns of stimuli repeat in an experiment.

State features are not restricted to describing the external stimuli that an animal experiences; they can describe neural activity patterns that external stimuli produce in an animal's brain, and these patterns can be history-dependent, meaning that they can be persistent patterns produced by sequences of external stimuli. Of course, we do not know exactly what these neural activity patterns are, but a real-time model like the TD model allows one to explore the consequences on learning of different hypotheses about the internal representations of external stimuli. For these reasons, the TD model does not commit to any particular state representation. In addition, because the TD model includes discounting and eligibility traces that span time intervals between stimuli, the model also makes it possible to explore how discounting and eligibility traces interact with stimulus representations in making predictions about the results of classical conditioning experiments.

Below we describe some of the state representations that have been used with the TD model and some of their implications, but for the moment we stay agnostic about the representation and just assume that each state s is represented by a feature vector $\mathbf{x}(s) = (x_1(s), x_2(s), \ldots, x_n(s))^\top$. Then the aggregate associative strength corresponding to a state s is given by (14.1), the same as for the Rescorla-Wgner model, but the TD model updates the associative strength vector, $\mathbf{w}$, differently. With t now labeling a time step instead of a complete trial, the TD model governs learning according to this update:

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha \delta_t \mathbf{z}_t, \tag{14.4}$$

which replaces $\mathbf{x}_t(S_t)$ in the Rescorla–Wagner update (14.2) with $\mathbf{z}_t$, a vector of eligibility traces, and instead of the δ_t of (14.3), here δ_t is a TD error:

$$\delta_t = R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w}_t) - \hat{v}(S_t, \mathbf{w}_t), \tag{14.5}$$

where γ is a discount factor (between 0 and 1), R_t is the prediction target at time t, and $\hat{v}(S_{t+1}, \mathbf{w}_t)$ and $\hat{v}(S_t, \mathbf{w}_t)$ are aggregate associative strengths at $t+1$ and t as defined by (14.1).

Each component i of the eligibility-trace vector $\mathbf{z}_t$ increments or decrements according to the component $x_i(S_t)$ of the feature vector $\mathbf{x}(S_t)$, and otherwise decays with a rate determined by $\gamma\lambda$:

$$\mathbf{z}_{t+1} = \gamma \lambda \mathbf{z}_t + \mathbf{x}(S_t). \tag{14.6}$$

Here λ is the usual eligibility trace decay parameter.

Note that if $\gamma = 0$, the TD model reduces to the Rescorla–Wagner model with the exceptions that: the meaning of t is different in each case (a trial number for the Rescorla–Wagner model and a time step for the TD model), and in the TD model there is a one-time-step lead in the prediction target R. The TD model is equivalent to the backward view of the semi-gradient TD(λ) algorithm with linear function approximation (Chapter 12), except that R_t in the model does not have to be a reward signal as it does when the TD algorithm is used to learn a value function for policy-improvement.

14.2.4 TD Model Simulations

Real-time conditioning models like the TD model are interesting primarily because they make predictions for a wide range of situations that cannot be represented by trial-level models. These situations involve the timing and durations of conditionable stimuli, the timing of these stimuli in relation to the timing of the US, and the timing and shapes of CRs. For example, the US generally must begin after the onset of a neutral stimulus for conditioning to occur, with the rate and effectiveness of learning depending on the inter-stimulus interval, or ISI, the interval between the onsets of the CS and the US. When CRs appear, they generally begin before the appearance of the US and their temporal profiles change during learning. In conditioning with compound CSs, the component stimuli of the compound

CSs may not all begin and end at the same time, sometimes forming what is called a *serial compound* in which the component stimuli occur in a sequence over time. Timing considerations like these make it important to consider how stimuli are represented, how these representations unfold over time during and between trials, and how they interact with discounting and eligibility traces.

Figure 14.2 shows three of the stimulus representations that have been used in exploring the behavior of the TD model: the *complete serial compound* (CSC), the *microstimulus* (MS), and the *presence* representations (Ludvig, Sutton, and Kehoe, 2012). These representations differ in the degree to which they force generalization among nearby time points during which a stimulus is present.

The simplest of the representations shown in Figure 14.2 is the presence representation in the figure's right column. This representation has a single feature for each component CS present on a trial, where the feature has value 1 whenever that component is present, and 0 otherwise.[4] The presence representation is not a realistic hypothesis about how stimuli are represented in an animal's brain, but as we describe below, the TD model with this representation can produce many of the timing phenomena seen in classical conditioning.

For the CSC representation (left column of Figure 14.2), the onset of each external stimulus initiates a sequence of precisely-timed short-duration internal signals that continues until the external stimulus

[4]In our formalism, there is a different state, S_t, for each time step t during a trial, and for a trial in which a compound CS consists of n component CSs of various durations occurring at various times throughout the trial, there is a feature, x_i, for each component CS_i, $i = 1, \ldots, n$, where $x_i(S_t) = 1$ for all times t when the CS_i is present, and equals zero otherwise.

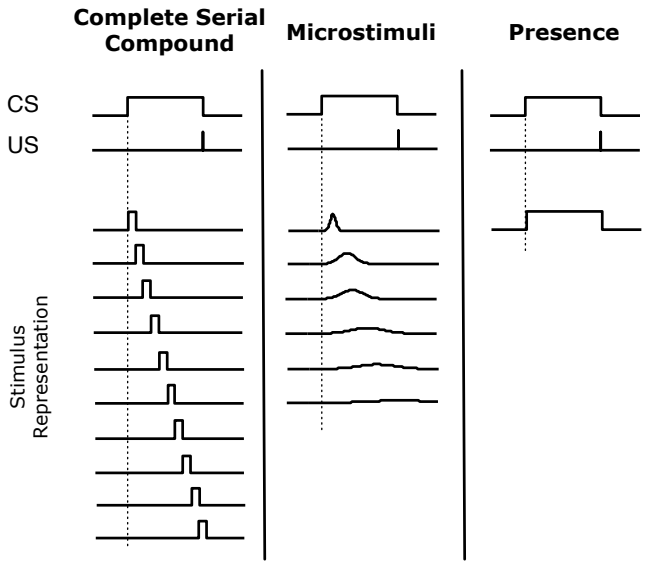


Figure 14.2: Three stimulus representations (in columns) sometimes used with the TD model. Each row represents one element of the stimulus representation. The three representations vary along a temporal generalization gradient, with no generalization between nearby time points in the complete serial compound (left column) and complete generalization between nearby time points in the presence representation (right column). The microstimulus representation occupies a middle ground. The degree of temporal generalization determines the temporal granularity with which US predictions are learned. Adapted with minor changes from *Learning & Behavior*, Evaluating the TD Model of Classical Conditioning, volume 40, 2012, p. 311, E. A. Ludvig, R. S. Sutton, E. J. Kehoe. With permission of Springer.

ends.[5] This is like assuming the animal's nervous system has a clock that keeps precise track of time during stimulus presentations; it is what engineers call a "tapped delay line." Like the presence representation, the CSC representation is unrealistic as a hypothesis about how the brain internally represents stimuli, but Ludvig et al. (2012) call it a "useful fiction" because it can reveal details of how the TD model works when relatively unconstrained by the stimulus representation. The CSC representation is also used in most TD models of dopamine-producing neurons in the brain, a topic we take up in Chapter 15. The CSC representation is often viewed as an essential part of the TD model, although this view is mistaken.

The MS representation (center column of Figure 14.2) is like the CSC representation in that each external stimulus initiates a cascade of internal stimuli, but in this case the internal stimuli—the microstimuli—are not of such limited and non-overlapping form; they are extended over time and overlap. As time elapses from stimulus onset, different sets of microstimuli become more or less active, and each subsequent microstimulus becomes progressively wider in time and reaches a lower maximal level. Of course, there are many MS representations depending on the nature of the microstimuli, and a number of examples of MS representations have been studied in the literature, in some cases along with proposals for how an animal's brain might generate them (see the Bibliographic and Historical Comments at the end of this chapter). MS representations are more realistic than the presence or CSC representations as hypotheses about neural representations of stimuli, and they allow the behavior of the TD model to be related to a broader collection of phenomena observed in animal experiments. In particular, by assuming that cascades of microstimuli are initiated by USs as well as by CSs, and by studying the significant effects on learning of interactions between microstimuli, eligibility traces, and discounting, the TD model is helping to frame hypotheses to account for many of the subtle phenomena of classical conditioning and how an animal's brain might produce them. We say more about this below, particularly in Chapter 15 where we discuss reinforcement learning and neuroscience.

Even with the simple presence representation, however, the TD model produces all the basic properties of classical conditioning that are accounted for by the Rescorla–Wagner model, plus features of conditioning that are beyond the scope of trial-level models. For example, as we have already mentioned, a conspicuous feature of classical conditioning is that the US generally must begin *after* the onset of a neutral stimulus for conditioning to occur, and that after conditioning, the CR begins *before* the appearance of the US. In other words, conditioning generally requires a positive ISI, and the CR generally anticipates the US. How the strength of conditioning (e.g., the percentage of CRs elicited by a CS) depends on the ISI varies substantially across species and response systems, but it typically has the following properties: it is negligible for a zero or negative ISI, i.e., when the US onset occurs simultaneously with, or earlier than, the CS onset (although research has found that associative strengths sometimes increase slightly or become negative with negative ISIs); it increases to a maximum at a positive ISI where conditioning is most effective; and it then decreases to zero after an interval that varies widely with response systems. The precise shape of this dependency for the TD model depends on the values of its parameters and details of the stimulus representation, but these basic features of ISI-dependency are core properties of the TD model.

One of the theoretical issues arising with serial-compound conditioning, that is, conditioning with a compound CS whose components occur in a sequence, concerns the facilitation of remote associations. It has been found that if the empty trace interval between the CS and the US is filled with a second CS to form a serial-compound stimulus, then conditioning to the first CS is facilitated. Figure 14.3 shows the behavior of the TD model with the presence representation in a simulation of such an experiment whose timing details are shown at the top of the figure. Consistent with the experimental results (Kehoe, 1982), the model shows facilitation of both the rate of conditioning and the asymptotic level

[5] In our formalism, for each CS component $\mathrm{CS_i}$ present on a trial, and for each time step t during a trial, there is a separate feature x_i^t, where $x_i^t(S_{t'}) = 1$ if $t = t'$ for any t' at which $\mathrm{CS_i}$ is present, and equals 0 otherwise. This is different from the CSC representation in Sutton and Barto (1990) in which there are the same distinct features for each time step but no reference to external stimuli; hence the name complete serial compound.

of conditioning of the first CS due to the presence of the second CS.

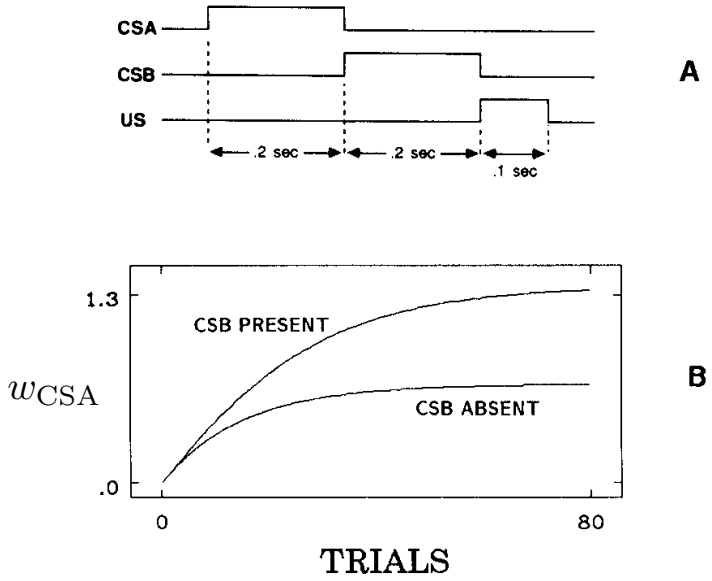


Figure 14.3: Facilitation of a remote association by an intervening stimulus in the TD model. Top: temporal relationships among stimuli within a trial. Bottom: behavior over trials of CSA's associative strength when CSA is presented in a serial compound as shown in the top panel, and when presented in an identical temporal relationship to the US, only without CSB. Adapted from Sutton and Barto (1990).

A well-known demonstration of the effects on conditioning of temporal relationships among stimuli within a trial is an experiment by Egger and Miller (1962) that involved two overlapping CSs in a delay configuration as shown in the top panel of Figure 14.4. Although CSB was in a better temporal relationship with the US, the presence of CSA substantially reduced conditioning to CSB as compared to controls in which CSA was absent. The bottom panel of Figure 14.4 shows the same result being generated by the TD model in a simulation of this experiment with the presence representation.

The TD model accounts for blocking because it is an error-correcting learning rule like the Rescorla–Wagner model. Beyond accounting for basic blocking results, however, the TD model predicts (with the presence representation and more complex representations a well) that blocking is reversed if the blocked stimulus is moved earlier in time so that its onset occurs before the onset of the blocking stimulus. This feature of the TD model's behavior deserves attention because it had not been observed at the time of the model's introduction. Recall that in blocking, if an animal has already learned that one CS predicts a US, then learning that a newly-added second CS also predicts the US is much reduced, i.e., is blocked. But if the newly-added second CS begins earlier than the pretrained CS, then—according to the TD model— learning to the newly-added CS is not blocked. In fact, as training continues and the newly-added CS gains associative strength, and the pretrained CS loses associative strength. The behavior of the TD model under these conditions is shown in Figure 14.5. This simulation experiment differed from the Egger-Miller experiment of Figure 14.4 in that the shorter CS with the later onset was given prior training until it was fully associated with the US. This surprising prediction led Kehoe, Scheurs, and Graham (1987) to conduct the experiment using the well-studied rabbit nictitating membrane preparation. Their results confirmed the model's prediction, and they noted that non-TD models have considerable difficulty explaining their data.

With the TD model, an earlier predictive stimulus takes precedence over a later predictive stimulus

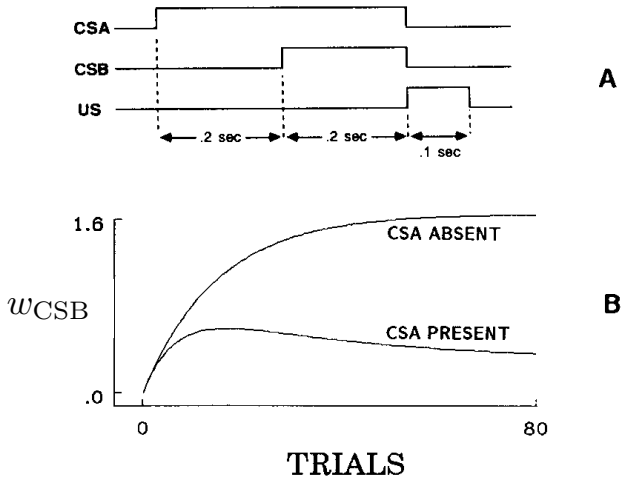


Figure 14.4: The Egger-Miller, or primacy, effect in the TD model. Top: temporal relationships among stimuli within a trial. Bottom: behavior over trials of CSB's associative strength when CSB is presented with and without CSA. Adapted from Sutton and Barto (1990).

because, like all the prediction methods described in this book, the TD model is based on the backing-up or bootstrapping idea: updates to associative strengths shift the strengths at a particular state toward the strength at later states. Another consequence of bootstrapping is that the TD model provides an account of higher-order conditioning, a feature of classical conditioning that is beyond the scope of the Rescoral-Wagner and similar models. As we described above, higher-order conditioning is the phenomenon in which a previously-conditioned CS can act as a US in conditioning another initially neutral stimulus. Figure 14.6 shows the behavior of the TD model (again with the presence representation) in a higher-order conditioning experiment—in this case it is second-order conditioning. In the first phase (not shown in the figure), CSB is trained to predict a US so that its associative strength increases, here to 1.6. In the second phase, CSA is paired with CSB in the absence of the US, in the sequential arrangement shown at the top of the figure. CSA acquires associative strength even though it is never paired with the US. With continued training, CSA's associative strength reaches a peak and then decreases because the associative strength of CSB, the secondary reinforcer, decreases so that it loses its ability to provide secondary reinforcement. CSB's associative strength decreases because the US does not occur in these higher-order conditioning trials. These are *extinction trials* for CSB because its predictive relationship to the US is disrupted so that its ability to act as a reinforcer decreases. This same pattern is seen in animal experiments. This extinction of conditioned reinforcement in higher-order conditioning trials makes it difficult to demonstrate higher-order conditioning unless the original predictive relationships are periodically refreshed by occasionally inserting first-order trials.

The TD model produces an analog of second- and higher-order conditioning because $\gamma\hat{v}(S_{t+1},\mathbf{w}_t) - \hat{v}(S_t,\mathbf{w}_t)$ appears in the TD error δ_t (14.5). This means that as a result of previous learning, $\gamma\hat{v}(S_{t+1},\mathbf{w}_t)$ can differ from $\hat{v}(S_t,\mathbf{w}_t)$, making δ_t non-zero (a temporal difference). This difference has the same status as R_{t+1} in (14.5), implying that as far as learning is concerned there is no difference between a temporal difference and the occurrence of a US. In fact, this feature of the TD algorithm is one of the major reasons for its development, which we now understand through its connection to dynamic programming as described in Chapter 6. Bootstrapping values is intimately related to second-order, and higher-order, conditioning.

In the examples of the TD model's behavior described above, we examined only the changes in

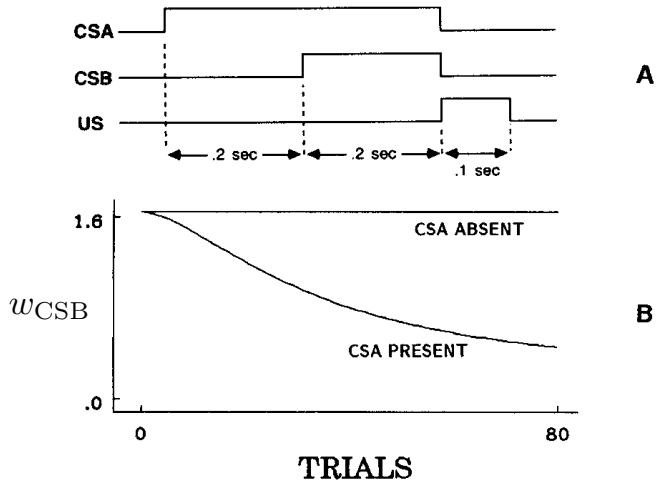


Figure 14.5: Temporal primacy overriding blocking in the TD model. Top: temporal relationships between stimuli. Bottom: behavior over trials of CSB's associative strength when CSB is presented with and without CSA. The only difference between this simulation and that shown in Figure 14.4 was that here CSB started out fully conditioned—CSB's associative strength was initially set to 1.653, the final level reached when CSB was presented alone for 80 trials, as in the "CSA-absent" case in Figure 14.4. Adapted from Sutton and Barto (1990).

the associative strengths of the CS components; we did not look at what the model predicts about properties of an animal's conditioned responses (CRs): their timing, shape, and how they develop over conditioning trials. These properties depend on the species, the response system being observed, and parameters of the conditioning trials, but in many experiments with different animals and different response systems, the magnitude of the CR, or the probability of a CR, increases as the expected time of the US approaches. For example, in classical conditioning of a rabbit's nictitating membrane response that we mentioned above, over conditioning trials the delay from CS onset to when the nictitating membrane begins to move across the eye decreases over trials, and the amplitude of this anticipatory closure gradually increases over the interval between the CS and the US until the membrane reaches maximal closure at the expected time of the US. The timing and shape of this CR is critical to its adaptive significance—covering the eye too early reduces vision (even though the nictitating membrane is translucent), while covering it too late is of little protective value. Capturing CR features like these is challenging for models of classical conditioning.

The TD model does not include as part of its definition any mechanism for translating the time course of the US prediction, $\hat{v}(S_t,\mathbf{w}_t)$, into a profile that can be compared with the properties of an animal's CR. The simplest choice is to let the time course of a simulated CR equal the time course of the US prediction. In this case, features of simulated CRs and how they change over trials depend only on the stimulus representation chosen and the values of the model's parameters α, γ, and λ.

Figure 14.7 shows the time courses of US predictions at different points during learning with the three representations shown in Figure 14.2. For these simulations the US occurred 25 time steps after the onset of the CS, and $\alpha = .05$, $\lambda = .95$ and $\gamma = .97$. With the CSC representation (Figure 14.7 left), the curve of the US prediction formed by the TD model increases exponentially throughout the interval between the CS and the US until it reaches a maximum exactly when the US occurs (at time step 25). This exponential increase is the result of discounting in the TD model learning rule. With the presence representation (Figure 14.7 middle), the US prediction is nearly constant while the stimulus is present

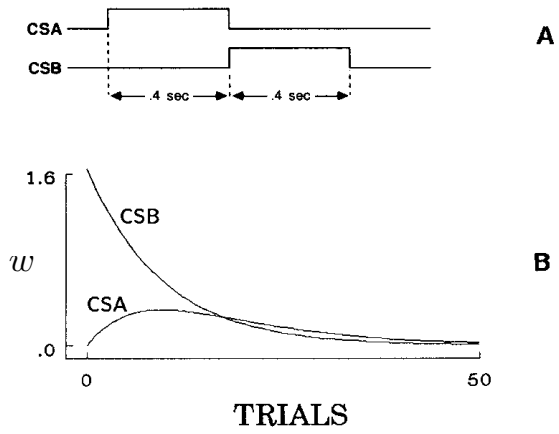


Figure 14.6: Second-order conditioning with the TD model. Top: temporal relationships between stimuli. Bottom: behavior of the associative strengths associated with CSA and CSB over trials. The second stimulus, CSB, has an initial associative strength of 1.653 at the beginning of the simulation. Adapted from Sutton and Barto (1990).

because there is only one weight, or associative strength, to be learned for each stimulus. Consequently, the TD model with the presence representation cannot recreate many features of CR timing. With an MS representation (Figure 14.7 right), the development of the TD model's US prediction is more complicated. After 200 trials the prediction's profile is a reasonable approximation of the US prediction curve produced with the CSC representation.

The US prediction curves shown in Figure 14.7 were not intended to precisely match profiles of CRs as they develop during conditioning in any particular animal experiment, but they illustrate the strong influence that the stimulus representation has on predictions derived from the TD model. Further, although we can only mention it here, how the stimulus representation interacts with discounting and eligibility traces is important in determining properties of the US prediction profiles produced by the TD model. Another dimension beyond what we can discuss here is the influence of different response-generation mechanisms that translate US predictions into CR profiles; the profiles shown in Figure 14.7 are "raw" US prediction profiles. Even without any special assumption about how an animal's brain might produce overt responses from US predictions, however, the profiles in Figure 14.7 for the CSC and MS representations increase as the time of the US approaches and reach a maximum at the time of the US, as is seen in many animal conditioning experiments.

The TD model, when combined with particular stimulus representations and response-generation mechanisms, is able to account for a surprisingly-wide range of phenomena observed in animal classical conditioning experiments, but it is far from being a perfect model. To generate other details of classical conditioning the model needs to be extended, perhaps by adding model-based elements and mechanisms for adaptively altering some of its parameters. Other approaches to modeling classical conditioning depart significantly from the Rescorla–Wagner-style error-correction process. Bayesian models, for example, work within a probabilistic framework in which experience revises probability estimates. All of these models usefully contribute to our understanding of classical conditioning.

Perhaps the most notable feature of the TD model is that it is based on a theory—the theory we have described in this book—that suggests an account of what an animal's nervous system is *trying to do* while undergoing conditioning: it is trying to form accurate *long-term predictions*, consistent with the limitations imposed by the way stimuli are represented and how the nervous system works. In other words, it suggests a *normative account* of classical conditioning in which long-term, instead of

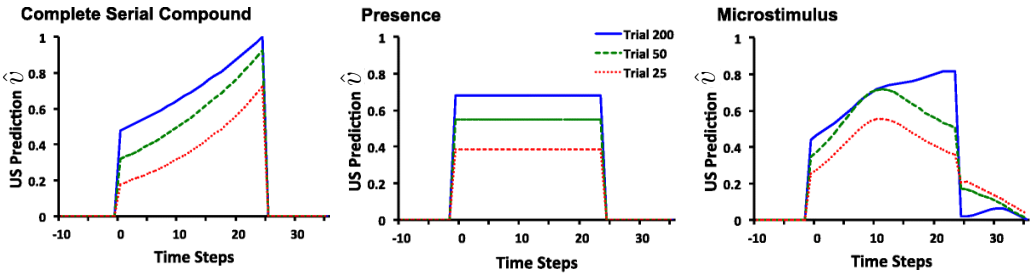


Figure 14.7: Time course of US prediction over the course of acquisition for the TD model with three different stimulus representations. Left: With the complete serial compound (CSC), the US prediction increases exponentially through the interval, peaking at the time of the US. At asymptote (trial 200), the US prediction peaks at the US intensity (1 in these simulations). Middle: With the presence representation, the US prediction converges to an almost constant level. This constant level is determined by the US intensity and the length of the CS–US interval. Right: With the microstimulus representation, at asymptote, the TD model approximates the exponentially increasing time course depicted with the CSC through a linear combination of the different microstimuli. Adapted with minor changes from *Learning & Behavior*, Evaluating the TD Model of Classical Conditioning, volume 40, 2012, E. A. Ludvig, R. S. Sutton, E. J. Kehoe. With permission of Springer.

immediate, prediction is a key feature.

The development of the TD model of classical conditioning is one instance in which the explicit goal was to model some of the details of animal learning behavior. In addition to its standing as an *algorithm*, then, TD learning is also the basis of this *model* of aspects of biological learning. As we discuss in Chapter 15, TD learning has also turned out to underlie an influential model of the activity of neurons that produce dopamine, a chemical in the brain of mammals that is deeply involved in reward processing. These are instances in which reinforcement learning theory makes detailed contact with animal behavioral and neural data.

We now turn to considering correspondences between reinforcement learning and animal behavior in instrumental conditioning experiments, the other major type of laboratory experiment studied by animal learning psychologists.

14.3 Instrumental Conditioning

In *instrumental conditioning* experiments learning depends on the consequences of behavior: the delivery of a reinforcing stimulus is contingent on what the animal does. In classical conditioning experiments, in contrast, the reinforcing stimulus—the US—is delivered independently of the animal's behavior. Instrumental conditioning is usually considered to be the same as *operant conditioning*, the term B. F. Skinner (1938, 1961) introduced for experiments with behavior-contingent reinforcement, though the experiments and theories of those who use these two terms differ in a number of ways, some of which we touch on below. We will exclusively use the term instrumental conditioning for experiments in which reinforcement is contingent upon behavior. The roots of instrumental conditioning go back to experiments performed by the American psychologist Edward Thorndike one hundred years before publication of the first edition of this book.

Thorndike observed the behavior of cats when they were placed in "puzzle boxes" from which they could escape by appropriate actions (Figure 14.8). For example, a cat could open the door of one box by performing a sequence of three separate actions: depressing a platform at the back of the box, pulling a string by clawing at it, and pushing a bar up or down. When first placed in a puzzle box,

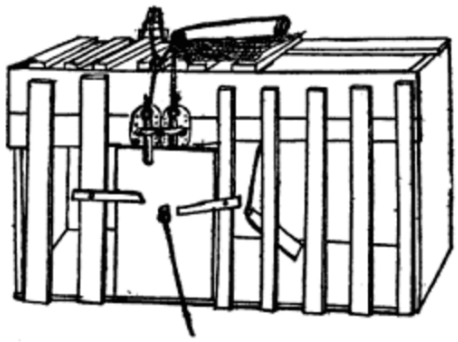


Figure 14.8: One of Thorndike's puzzle boxes. Reprinted from Thorndike, Animal Intelligence: An Experimental Study of the Associative Processes in Animals, *The Psychological Review, Series of Monograph Supplements*, II(4), Macmillan, New York, 1898.

with food visible outside, all but a few of Thorndike's cats displayed "evident signs of discomfort" and extraordinarily vigorous activity "to strive instinctively to escape from confinement" (Thorndike, 1898).

In experiments with different cats and boxes with different escape mechanisms, Thorndike recorded the amounts of time each cat took to escape over multiple experiences in each box. He observed that the time almost invariably decreased with successive experiences, for example, from 300 seconds to 6 or 7 seconds. He described cats' behavior in a puzzle box like this:

> The cat that is clawing all over the box in her impulsive struggle will probably claw the string or loop or button so as to open the door. And gradually all the other non-successful impulses will be stamped out and the particular impulse leading to the successful act will be stamped in by the resulting pleasure, until, after many trials, the cat will, when put in the box, immediately claw the button or loop in a definite way. (Thorndike 1898, p. 13)

These and other experiments (some with dogs, chicks, monkeys, and even fish) led Thorndike to formulate a number of "laws" of learning, the most influential being the *Law of Effect*, a version of which we quoted in Chapter 1. This law describes what is generally known as learning by trial and error. As mentioned in Chapter 1, many aspects of the Law of Effect have generated controversy, and its details have been modified over the years. Still the law—in one form or another—expresses an enduring principle of learning.

Essential features of reinforcement learning algorithms correspond to features of animal learning described by the Law of Effect. First, reinforcement learning algorithms are *selectional*, meaning that they try alternatives and select among them by comparing their consequences. Second, reinforcement learning algorithms are *associative*, meaning that the alternatives found by selection are associated with particular situations, or states, to form the agent's policy. Like learning described by the Law of Effect, reinforcement learning is not just the process of *finding* actions that produce a lot of reward, but also of *connecting* these actions to situations or states. Thorndike used the phrase learning by "selecting and connecting" (Hilgard, 1956). Natural selection in evolution is a prime example of a selectional process, but it is not associative (at least as it is commonly understood); supervised learning is associative, but it is not selectional because it relies on instructions that directly tell the agent how to change its behavior.

In computational terms, the Law of Effect describes an elementary way of combining *search* and *memory*: search in the form of trying and selecting among many actions in each situation, and memory in the form of associations linking situations with the actions found—so far—to work best in those

situations. Search and memory are essential components of all reinforcement learning algorithms, whether memory takes the form of an agent's policy, value function, or environment model.

A reinforcement learning algorithm's need to search means that it has to explore in some way. Animals clearly explore as well, and early animal learning researchers disagreed about the degree of guidance an animal uses in selecting its actions in situations like Thorndike's puzzle boxes. Are actions the result of "absolutely random, blind groping" (Woodworth, 1938, p. 777), or is there some degree of guidance, either from prior learning, reasoning, or other means? Although some thinkers, including Thorndike, seem to have taken the former position, others favored more deliberate exploration. Reinforcement learning algorithms allow wide latitude for how much guidance an agent can employ in selecting actions. The forms of exploration we have used in the algorithms presented in this book, such as ϵ-greedy and upper-confidence-bound action selection, are merely among the simplest. More sophisticated methods are possible, with the only stipulation being that there has to be *some* form of exploration for the algorithms to work effectively.

The feature of our treatment of reinforcement learning allowing the set of actions available at any time to depend on the environment's current state echoes something Thorndike observed in his cats' puzzle-box behaviors. The cats selected actions from those that they instinctively perform in their current situation, which Thorndike called their "instinctual impulses." First placed in a puzzle box, a cat instinctively scratches, claws, and bites with great energy: a cat's instinctual responses to finding itself in a confined space. Successful actions are selected from these and not from every possible action or activity. This is like the feature of our formalism where the action selected from a state s belongs to a set of admissible actions, $\mathcal{A}(s)$. Specifying these sets is an important aspect of reinforcement learning because it can radically simplify learning. They are like an animal's instinctual impulses. On the other hand, Thorndike's cats might have been exploring according to an instinctual context-specific *ordering* over actions rather than by just selecting from a set of instinctual impulses. This is another way to make reinforcement learning easier.

Among the most prominent animal learning researchers influenced by the Law of Effect were Clark Hull (e.g., Hull, 1943) and B. F. Skinner (e.g., Skinner, 1938). At the center of their research was the idea of selecting behavior on the basis of its consequences. Reinforcement learning has features in common with Hull's theory, which included eligibility-like mechanisms and secondary reinforcement to account for the ability to learn when there is a significant time interval between an action and the consequent reinforcing stimulus (see Section 14.4). Randomness also played a role in Hull's theory through what he called "behavioral oscillation" to introduce exploratory behavior.

Skinner did not fully subscribe to the memory aspect of the Law of Effect. Being averse to the idea of associative linkages, he instead emphasized selection from spontaneously-emitted behavior. He introduced the term "operant" to emphasize the key role of an action's effects on an animal's environment. Unlike the experiments of Thorndike and others, which consisted of sequences of separate trials, Skinner's operant conditioning experiments allowed animal subjects to behave for extended periods of time without interruption. He invented the operant conditioning chamber, now called a "Skinner box," the most basic version of which contains a lever or key that an animal can press to obtain a reward, such as food or water, which would be delivered according to a well-defined rule, called a reinforcement schedule. By recording the cumulative number of lever presses as a function of time, Skinner and his followers could investigate the effect of different reinforcement schedules on the animal's rate of lever-pressing. Modeling results from experiments likes these using the reinforcement learning principles we present in this book is not well developed, but we mention some exceptions in the Bibliographic and Historical Remarks section at the end of this chapter.

Another of Skinner's contributions resulted from his recognition of the effectiveness of training an animal by reinforcing successive approximations of the desired behavior, a process he called *shaping*. Although this technique had been used by others, including Skinner himself, its significance was impressed upon him when he and colleagues were attempting to train a pigeon to bowl by swiping a wooden ball

with its beak. After waiting for a long time without seeing any swipe that they could reinforce, they

> ... decided to reinforce any response that had the slightest resemblance to a swipe—perhaps, at first, merely the behavior of looking at the ball—and then to select responses which more closely approximated the final form. The result amazed us. In a few minutes, the ball was caroming off the walls of the box as if the pigeon had been a champion squash player. (Skinner, 1958, p. 94)

Not only did the pigeon learn a behavior that is unusual for pigeons, it learned quickly through an interactive process in which its behavior and the reinforcement contingencies changed in response to each other. Skinner compared the process of altering reinforcement contingencies to the work of a sculptor shaping clay into a desired form. Shaping is a powerful technique for computational reinforcement learning systems as well. When it is difficult for an agent to receive any non-zero reward signal at all, either due to sparseness of rewarding situations or their inaccessibility given initial behavior, starting with an easier problem and incrementally increasing its difficulty as the agent learns can be an effective, and sometimes indispensable, strategy.

A concept from psychology that is especially relevant in the context of instrumental conditioning is *motivation*, which refers to processes that influence the direction and strength, or vigor, of behavior. Thorndike's cats, for example, were motivated to escape from puzzle boxes because they wanted the food that was sitting just outside. Obtaining this goal was rewarding to them and reinforced the actions allowing them to escape. It is difficult to link the concept of motivation, which has many dimensions, in a precise way to reinforcement learning's computational perspective, but there are clear links with some of its dimensions.

In one sense, a reinforcement learning agent's reward signal is at the base of its motivation: the agent is motivated to maximize the total reward it receives over the long run. A key facet of motivation, then, is what makes an agent's experience rewarding. In reinforcement learning, reward signals depend on the state of the reinforcement learning agent's environment and the agent's actions. Further, as pointed out in Chapter 1, the state of the agent's environment not only includes information about what is external to the machine, like an organism or a robot, that houses the agent, but also what is internal to this machine. Some internal state components correspond to what psychologists call an animal's *motivational state*, which influences what is rewarding to the animal. For example, an animal will be more rewarded by eating when it is hungry than when it has just finished a satisfying meal. The concept of state dependence is broad enough to allow for many types of modulating influences on the generation of reward signals.

Value functions provide a further link to psychologists' concept of motivation. If the most basic motive for selecting an action is to obtain as much reward as possible, for a reinforcement learning agent that selects actions using a value function, a more proximal motive is to *ascend the gradient of its value function*, that is, to select actions expected to lead to the most highly-valued next states (or what is essentially the same thing, to select actions with the greatest action-values). For these agents, value functions are the main driving force determining the direction of their behavior.

Another dimension of motivation is that an animal's motivational state not only influences learning, but also influences the strength, or vigor, of the animal's behavior after learning. For example, after learning to find food in the goal box of a maze, a hungry rat will run faster to the goal box than one that is not hungry. This aspect of motivation does not link so cleanly to the reinforcement learning framework we present here, but in the Bibliographical and Historical Remarks section at the end of this chapter we cite several publications that propose theories of behavioral vigor based on reinforcement learning.

We turn now to the subject of learning when reinforcing stimuli occur well after the events they reinforce. The mechanisms used by reinforcement learning algorithms to enable learning with delayed reinforcement—eligibility traces and TD learning—closely correspond to psychologists' hypotheses

about how animals can learn under these conditions.

14.4 Delayed Reinforcement

The Law of Effect requires a backward effect on connections, and some early critics of the law could not conceive of how the present could affect something that was in the past. This concern was amplified by the fact that learning can even occur when there is a considerable delay between an action and the consequent reward or penalty. Similarly, in classical conditioning, learning can occur when US onset follows CS offset by a non-negligible time interval. We call this the problem of delayed reinforcement, which is related to what Minsky (1961) called the "credit-assignment problem for learning systems": how do you distribute credit for success among the many decisions that may have been involved in producing it? The reinforcement learning algorithms presented in this book include two basic mechanisms for addressing this problem. The first is the use of eligibility traces, and the second is the use of TD methods to learn value functions that provide nearly immediate evaluations of actions (in tasks like instrumental conditioning experiments) or that provide immediate prediction targets (in tasks like classical conditioning experiments). Both of these methods correspond to similar mechanisms proposed in theories of animal learning.

Pavlov (1927) pointed out that every stimulus must leave a trace in the nervous system that persists for some time after the stimulus ends, and he proposed that stimulus traces make learning possible when there is a temporal gap between the CS offset and the US onset. To this day, conditioning under these conditions is called *trace conditioning* (Figure 14.1). Assuming a trace of the CS remains when the US arrives, learning occurs through the simultaneous presence of the trace and the US. We discuss some proposals for trace mechanisms in the nervous system in Chapter 15.

Stimulus traces were also proposed as a means for bridging the time interval between actions and consequent rewards or penalties in instrumental conditioning. In Hull's influential learning theory, for example, "molar stimulus traces" accounted for what he called an animal's *goal gradient*, a description of how the maximum strength of an instrumentally-conditioned response decreases with increasing delay of reinforcement (Hull, 1932, 1943). Hull hypothesized that an animal's actions leave internal stimuli whose traces decay exponentially as functions of time since an action was taken. Looking at the animal learning data available at the time, he hypothesized that the traces effectively reach zero after 30 to 40 seconds.

The eligibility traces used in the algorithms described in this book are like Hull's traces: they are decaying traces of past state visitations, or of past state–action pairs. Eligibility traces were introduced by Klopf (1972) in his neuronal theory in which they are temporally-extended traces of past activity at synapses, the connections between neurons. Klopf's traces are more complex than the exponentially-decaying traces our algorithms use, and we discuss this more when we take up his theory in Section 15.9.

To account for goal gradients that extend over longer time periods than spanned by stimulus traces, Hull (1943) proposed that longer gradients result from conditioned reinforcement passing backwards from the goal, a process acting in conjunction with his molar stimulus traces. Animal experiments showed that if conditions favor the development of conditioned reinforcement during a delay period, learning does not decrease with increased delay as much as it does under conditions that obstruct secondary reinforcement. Conditioned reinforcement is favored if there are stimuli that regularly occur during the delay interval. Then it is as if reward is not actually delayed because there is more immediate conditioned reinforcement. Hull therefore envisioned that there is a primary gradient based on the delay of the primary reinforcement mediated by stimulus traces, and that this is progressively modified, and lengthened, by conditioned reinforcement.

Algorithms presented in this book that use both eligibility traces and value functions to enable learning with delayed reinforcement correspond to Hull's hypothesis about how animals are able to

learn under these conditions. The actor–critic architecture discussed in Sections 13.5, 15.7, and 15.8 illustrates this correspondence most clearly. The critic uses a TD algorithm to learn a value function associated with the system's current behavior, that is, to predict the current policy's return. The actor updates the current policy based on the critic's predictions, or more exactly, on changes in the critic's predictions. The TD error produced by the critic acts as a conditioned reinforcement signal for the actor, providing an immediate evaluation of performance even when the primary reward signal itself is considerably delayed. Algorithms that estimate action-value functions, such as Q-learning and Sarsa, similarly use TD learning principles to enable learning with delayed reinforcement by means of conditioned reinforcement. The close parallel between TD learning and the activity of dopamine producing neurons that we discuss in Chapter 15 lends additional support to links between reinforcement learning algorithms and this aspect of Hull's learning theory.

14.5 Cognitive Maps

Model-based reinforcement learning algorithms use environment models that have elements in common with what psychologists call *cognitive maps*. Recall from our discussion of planning and learning in Chapter 8 that by an environment model we mean anything an agent can use to predict how its environment will respond to its actions in terms of state transitions and rewards, and by planning we mean any process that computes a policy from such a model. Environment models consist of two parts: the state-transition part encodes knowledge about the effect of actions on state changes, and the reward-model part encodes knowledge about the reward signals expected for each state or each state–action pair. A model-based algorithm selects actions by using a model to predict the consequences of possible courses of action in terms of future states and the reward signals expected to arise from those states. The simplest kind of planning is to compare the predicted consequences of collections of "imagined" sequences of decisions.

Questions about whether or not animals use environment models, and if so, what are the models like and how are they learned, have played influential roles in the history of animal learning research. Some researchers challenged the then-prevailing stimulus-response (S–R) view of learning and behavior, which corresponds to the simplest model-free way of learning policies, by demonstrating *latent learning*. In the earliest latent learning experiment, two groups of rats were run in a maze. For the experimental group, there was no reward during the first stage of the experiment, but food was suddenly introduced into the goal box of the maze at the start of the second stage. For the control group, food was in the goal box throughout both stages. The question was whether or not rats in the experimental group would have learned anything during the first stage in the absence of food reward. Although the experimental rats did not *appear* to learn much during the first, unrewarded, stage, as soon as they discovered the food that was introduced in the second stage, they rapidly caught up with the rats in the control group. It was concluded that "during the non-reward period, the rats [in the experimental group] were developing a latent learning of the maze which they were able to utilize as soon as reward was introduced" (Blodgett, 1929).

Latent learning is most closely associated with the psychologist Edward Tolman, who interpreted this result, and others like it, as showing that animals could learn a "cognitive map of the environment" in the absence of rewards or penalties, and that they could use the map later when they were motivated to reach a goal (Tolman, 1948). A cognitive map could also allow a rat to plan a route to the goal that was different from the route the rat had used in its initial exploration. Explanations of results like these led to the enduring controversy lying at the heart of the behaviorist/cognitive dichotomy in psychology. In modern terms, cognitive maps are not restricted to models of spatial layouts but are more generally environment models, or models of an animal's "task space" (e.g., Wilson, Takahashi, Schoenbaum, and Niv, 2014). The cognitive map explanation of latent learning experiments is analogous to the claim that animals use model-based algorithms, and that environment models can be learned even without

explicit rewards or penalties. Models are then used for planning when the animal is motivated by the appearance of rewards or penalties.

Tolman's account of how animals learn cognitive maps was that they learn stimulus-stimulus, or S–S, associations by experiencing successions of stimuli as they explore an environment. In psychology this is called *expectancy theory*: given S–S associations, the occurrence of a stimulus generates an expectation about the stimulus to come next. This is much like what control engineers call *system identification*, in which a model of a system with unknown dynamics is learned from labeled training examples. In the simplest discrete-time versions, training examples are S–S′ pairs, where S is a state and S′, the subsequent state, is the label. When S is observed, the model creates the "expectation" that S′ will be observed next. Models more useful for planning involve actions as well, so that examples look like SA–S′, where S′ is expected when action A is executed in state S. It is also useful to learn how the environment generates rewards. In this case, examples are of the form S–R or SA–R, where R is a reward signal associated with S or the SA pair. These are all forms of supervised learning by which an agent can acquire cognitive-like maps whether or not it receives any non-zero reward signals while exploring its environment.

14.6 Habitual and Goal-directed Behavior

The distinction between model-free and model-based reinforcement learning algorithms corresponds to the distinction psychologists make between *habitual* and *goal-directed* control of learned behavioral patterns. Habits are behavior patterns triggered by appropriate stimuli and then performed more-or-less automatically. Goal-directed behavior, according to how psychologists use the phrase, is purposeful in the sense that it is controlled by knowledge of the value of goals and the relationship between actions and their consequences. Habits are sometimes said to be controlled by antecedent stimuli, whereas goal-directed behavior is said to be controlled by its consequences (Dickinson, 1980, 1985). Goal-directed control has the advantage that it can rapidly change an animal's behavior when the environment changes its way of reacting to the animal's actions. While habitual behavior responds quickly to input from an accustomed environment, it is unable to quickly adjust to changes in the environment. The development of goal-directed behavioral control was likely a major advance in the evolution of animal intelligence.

Figure 14.9 illustrates the difference between model-free and model-based decision strategies in a hypothetical task in which a rat has to navigate a maze that has distinctive goal boxes, each delivering an associated reward of the magnitude shown (Figure 14.9 top). Starting at S_1, the rat has to first select left (L) or right (R) and then has to select L or R again at S_2 or S_3 to reach one of the goal boxes. The goal boxes are the terminal states of each episode of the rat's episodic task. A model-free strategy (Figure 14.9 lower left) relies on stored values for state–action pairs. These action values (Q-values) are estimates of the highest return the rat can expect for each action taken from each (nonterminal) state. They are obtained over many trials of running the maze from start to finish. When the action values have become good enough estimates of the optimal returns, the rat just has to select at each state the action with the largest action value in order to make optimal decisions. In this case, when the action-value estimates become accurate enough, the rat selects L from S_1 and R from S_2 to obtain the maximum return of 4. A different model-free strategy might simply rely on a cached policy instead of action values, making direct links from S_1 to L and from S_2 to R. In neither of these strategies do decisions rely on an environment model. There is no need to consult a state-transition model, and no connection is required between the features of the goal boxes and the rewards they deliver.

Figure 14.9 (lower right) illustrates a model-based strategy. It uses an environment model consisting of a state-transition model and a reward model. The state-transition model is shown as a decision tree, and the reward model associates the distinctive features of the goal boxes with the rewards to be found in each. (The rewards associated with states S_1, S_2, and S_3 are also part of the reward model, but here they are zero and are not shown.) A model-based agent can decide which way to turn at each state

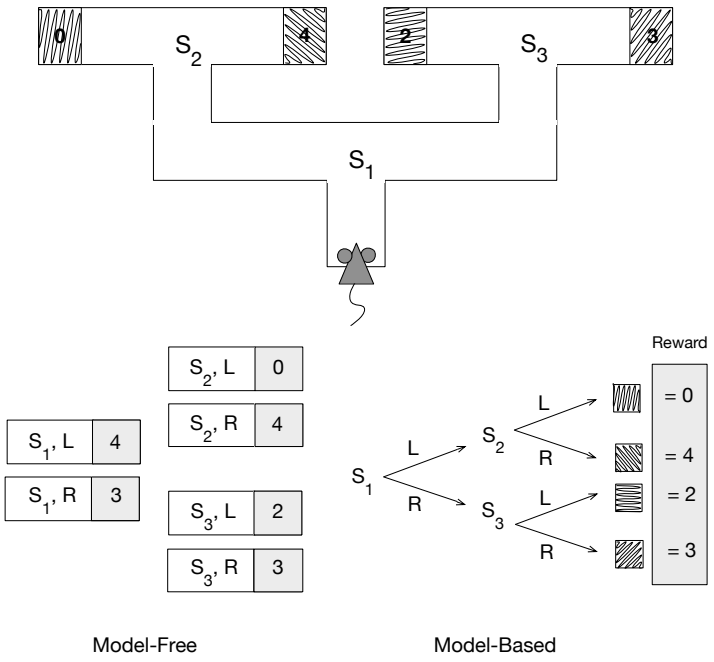


Figure 14.9: Model-based and model-free strategies to solve a hypothetical sequential action-selection problem. Top: a rat navigates a maze with distinctive goal boxes, each associated with a reward having the value shown. Lower left: a model-free strategy relies on stored action values for all the state–action pairs obtained over many learning trials. To make decisions the rat just has to select at each state the action with the largest action value for that state. Lower right: in a model-based strategy, the rat learns an environment model, consisting of knowledge of state–action-next-state transitions and a reward model consisting of knowledge of the reward associated with each distinctive goal box. The rat can decide which way to turn at each state by using the model to simulate sequences of action choices to find a path yielding the highest return. Adapted from *Trends in Cognitive Science*, volume 10, number 8, Y. Niv, D. Joel, and P. Dayan, A Normative Perspective on Motivation, p. 376, 2006, with permission from Elsevier.

by using the model to simulate sequences of action choices to find a path yielding the highest return. In this case the return is the reward obtained from the outcome at the end of the path. Here, with a sufficiently accurate model, the rat would select L and then R to obtain reward of 4. Comparing the predicted returns of simulated paths is a simple form of planning, which can be done in a variety of ways as discussed in Chapter 8.

When the environment of a model-free agent changes the way it reacts to the agent's actions, the agent has to acquire new experience in the changed environment during which it can update its policy and/or value function. In the model-free strategy shown in Figure 14.9 (lower left), for example, if one of the goal boxes were to somehow shift to delivering a different reward, the rat would have to traverse the maze, possibly many times, to experience the new reward upon reaching that goal box, all the while updating either its policy or its action-value function (or both) based on this experience. The key point is that for a model-free agent to change the action its policy specifies for a state, or to change an action value associated with a state, it has to move to that state, act from it, possibly many times, and experience the consequences of its actions.

A model-based agent can accommodate changes in its environment without this kind of 'personal experience' with the states and actions affected by the change. A change in its model automatically (through planning) changes its policy. Planning can determine the consequences of changes in the environment that have never been linked together in the agent's own experience. For example, again referring to the maze task of Figure 14.9, imagine that a rat with a previously learned transition and reward model is placed directly in the goal box to the right of S_2 to find that the reward available there now has value 1 instead of 4. The rat's reward model will change even though the action choices required to find that goal box in the maze were not involved. The planning process will bring knowledge of the new reward to bear on maze running without the need for additional experience in the maze; in this case changing the policy to right turns at both S_1 and S_3 to obtain a return of 3.

Exactly this logic is the basis of *outcome-devaluation experiments* with animals. Results from these experiments provide insight into whether an animal has learned a habit or if its behavior is under goal-directed control. Outcome-devaluation experiments are like latent-learning experiments in that the reward changes from one stage to the next. After an initial rewarded stage of learning, the reward value of an outcome is changed, including being shifted to zero or even to a negative value.

An early important experiment of this type was conducted by Adams and Dickinson (1981). They trained rats via instrumental conditioning until the rats energetically pressed a lever for sucrose pellets in a training chamber. The rats were then placed in the same chamber with the lever retracted and allowed non-contingent food, meaning that pellets were made available to them independently of their actions. After 15-minutes of this free-access to the pellets, rats in one group were injected with the nausea-inducing poison lithium chloride. This was repeated for three sessions, in the last of which none of the injected rats consumed any of the non-contingent pellets, indicating that the reward value of the pellets had been decreased—the pellets had been devalued. In the next stage taking place a day later, the rats were again placed in the chamber and given a session of extinction training, meaning that the response lever was back in place but disconnected from the pellet dispenser so that pressing it did not release pellets. The question was whether the rats that had the reward value of the pellets decreased would lever-press less than rats that did not have the reward value of the pellets decreased, even without experiencing the devalued reward as a result of lever-pressing. It turned out that the injected rats had significantly lower response rates than the non-injected rats *right from the start of the extinction trials*.

Adams and Dickinson concluded that the injected rats associated lever pressing with consequent nausea by means of a cognitive map linking lever pressing to pellets, and pellets to nausea. Hence, in the extinction trials, the rats "knew" that the consequences of pressing the lever would be something they did not want, and so they reduced their lever-pressing right from the start. The important point is that they reduced lever-pressing without ever having experienced lever-pressing directly followed by being sick: no lever was present when they were made sick. They seemed able to combine knowledge of the outcome of a behavioral choice (pressing the lever will be followed by getting a pellet) with the reward value of the outcome (pellets are to be avoided) and hence could alter their behavior accordingly. Not every psychologist agrees with this "cognitive" account of this kind of experiment, and it is not the only possible way to explain these results, but the model-based planning explanation is widely accepted.

Nothing prevents an agent from using both model-free and model-based algorithms, and there are good reasons for using both. We know from our own experience that with enough repetition, goal-directed behavior tends to turn into habitual behavior. Experiments show that this happens for rats too. Adams (1982) conducted an experiment to see if extended training would convert goal-directed behavior into habitual behavior. He did this by comparing the effect of outcome devaluation on rats that experienced different amounts of training. If extended training made the rats less sensitive to devaluation compared to rats that received less training, this would be evidence that extended training made the behavior more habitual. Adams' experiment closely followed the Adams and Dickinson (1981) experiment just described. Simplifying a bit, rats in one group were trained until they made 100 rewarded lever-presses, and rats in the other group—the overtrained group—were trained until they made 500 rewarded lever-presses. After this training, the reward value of the pellets was decreased

(using lithium chloride injections) for rats in both groups. Then both groups of rats were given a session of extinction training. Adams' question was whether devaluation would effect the rate of lever-pressing for the overtrained rats less than it would for the non-overtrained rats, which would be evidence that extended training reduces sensitivity to outcome devaluation. It turned out that devaluation strongly decreased the lever-pressing rate of the non-overtrained rats. For the overtrained rats, in contrast, devaluation had little effect on their lever-pressing; in fact, if anything, it made it more vigorous. (The full experiment included control groups showing that the different amounts of training did not by themselves significantly effect lever-pressing rates after learning.) This result suggested that while the non-overtrained rats were acting in a goal-directed manner sensitive to their knowledge of the outcome of their actions, the overtrained rats had developed a lever-pressing habit.

Viewing this and other results like it from a computational perspective provides insight as to why one might expect animals to behave habitually in some circumstances, in a goal-directed way in others, and why they shift from one mode of control to another as they continue to learn. While animals undoubtedly use algorithms that do not exactly match those we have presented in this book, one can gain insight into animal behavior by considering the tradeoffs that various reinforcement learning algorithms imply. An idea developed by computational neuroscientists Daw, Niv, and Dayan (2005) is that animals use both model-free and model-based processes. Each process proposes an action, and the action chosen for execution is the one proposed by the process judged to be the more trustworthy of the two as determined by measures of confidence that are maintained throughout learning. Early in learning the planning process of a model-based system is more trustworthy because it chains together short-term predictions which can become accurate with less experience than long-term predictions of the model-free process. But with continued experience, the model-free process becomes more trustworthy because planning is prone to making mistakes due to model inaccuracies and short-cuts necessary to make planning feasible, such as various forms of "tree-pruning": the removal of unpromising search tree branches. According to this idea one would expect a shift from goal-directed behavior to habitual behavior as more experience accumulates. Other ideas have been proposed for how animals arbitrate between goal-directed and habitual control, and both behavioral and neuroscience research continues to examine this and related questions.

The distinction between model-free and model-based algorithms is proving to be useful for this research. One can examine the computational implications of these types of algorithms in abstract settings that expose basic advantages and limitations of each type. This serves both to suggest and to sharpen questions that guide the design of experiments necessary for increasing psychologists' understanding of habitual and goal-directed behavioral control.

14.7 Summary

Our goal in this chapter has been to discuss correspondences between reinforcement learning and the experimental study of animal learning in psychology. We emphasized at the outset that reinforcement learning as described in this book is not intended to model details of animal behavior. It is an abstract computational framework that explores idealized situations from the perspective of artificial intelligence and engineering. But many of the basic reinforcement learning algorithms were inspired by psychological theories, and in some cases, these algorithms have contributed to the development of new animal learning models. This chapter described the most conspicuous of these correspondences.

The distinction in reinforcement learning between algorithms for prediction and algorithms for control parallels animal learning theory's distinction between classical, or Pavlovian, conditioning and instrumental conditioning. The key difference between instrumental and classical conditioning experiments is that in the former the reinforcing stimulus is contingent upon the animal's behavior, whereas in the latter it is not. Learning to predict via a TD algorithm corresponds to classical conditioning, and we described the *TD model of classical conditioning* as one instance in which reinforcement learning

principles account for some details of animal learning behavior. This model generalizes the influential Rescorla–Wagner model by including the temporal dimension where events within individual trials influence learning, and it provides an account of second-order conditioning, where predictors of reinforcing stimuli become reinforcing themselves. It also is the basis of an influential view of the activity of dopamine neurons in the brain, something we take up in Chapter 15.

Learning by trial and error is at the base of the control aspect of reinforcement learning. We presented some details about Thorndike's experiments with cats and other animals that led to his *Law of Effect*, which we discussed here and in Chapter 1. We pointed out that in reinforcement learning, exploration does not have to be limited to "blind groping"; trials can be generated by sophisticated methods using innate and previously learned knowledge as long as there is *some* exploration. We discussed the training method B. F. Skinner called *shaping* in which reward contingencies are progressively altered to train an animal to successively approximate a desired behavior. Shaping is not only indispensable for animal training, it is also an effective tool for training reinforcement learning agents. There is also a connection to the idea of an animal's motivational state, which influences what an animal will approach or avoid and what events are rewarding or punishing for the animal.

The reinforcement learning algorithms presented in this book include two basic mechanisms for addressing the problem of delayed reinforcement: eligibility traces and value functions learned via TD algorithms. Both mechanisms have antecedents in theories of animal learning. Eligibility traces are similar to stimulus traces of early theories, and value functions correspond to the role of secondary reinforcement in providing nearly immediate evaluative feedback.

The next correspondence the chapter addressed is that between reinforcement learning's environment models and what psychologists call *cognitive maps*. Experiments conducted in the mid 20th century purported to demonstrate the ability of animals to learn cognitive maps as alternatives to, or as additions to, state–action associations, and later use them to guide behavior, especially when the environment changes unexpectedly. Environment models in reinforcement learning are like cognitive maps in that they can be learned by supervised learning methods without relying on reward signals, and then they can be used later to plan behavior.

Reinforcement learning's distinction between *model-free* and *model-based* algorithms corresponds to the distinction in psychology between *habitual* and *goal-directed* behavior. Model-free algorithms make decisions by accessing information that has been strored in a policy or an action-value function, whereas model-based methods select actions as the result of planning ahead using a model of the agent's environment. Outcome-devaluation experiments provide information about whether an animal's behavior is habitual or under goal-directed control. Reinforcement learning theory has helped clarify thinking about these issues.

Animal learning clearly informs reinforcement learning, but as a type of machine learning, reinforcement learning is directed toward designing and understanding effective learning algorithms, not toward replicating or explaining details of animal behavior. We focused on aspects of animal learning that relate in clear ways to methods for solving prediction and control problems, highlighting the fruitful two-way flow of ideas between reinforcement learning and psychology without venturing deeply into many of the behavioral details and controversies that have occupied the attention of animal learning researchers. Future development of reinforcement learning theory and algorithms will likely exploit links to many other features of animal learning as the computational utility of these features becomes better appreciated. We expect that a flow of ideas between reinforcement learning and psychology will continue to bear fruit for both disciplines.

Many connections between reinforcement learning and areas of psychology and other behavioral sciences are beyond the scope of this chapter. We largely omit discussing links to the psychology of decision making, which focuses on how actions are selected, or how decisions are made, *after* learning has taken place. We also do not discuss links to ecological and evolutionary aspects of behavior studied by ethologists and behavioral ecologists: how animals relate to one another and to their physical

surroundings, and how their behavior contributes to evolutionary fitness. Optimization, MDPs, and dynamic programming figure prominently in these fields, and our emphasis on agent interaction with dynamic environments connects to the study of agent behavior in complex "ecologies." Multi-agent reinforcement learning, omitted in this book, has connections to social aspects of behavior. Despite the lack of treatment here, reinforcement learning should by no means be interpreted as dismissing evolutionary perspectives. Nothing about reinforcement learning implies a *tabula rasa* view of learning and behavior. Indeed, experience with engineering applications has highlighted the importance of building into reinforcement learning systems knowledge that is analogous to what evolution provides to animals.

Bibliographical and Historical Remarks

Ludvig, Bellemare, and Pearson (2011) and Shah (2012) review reinforcement learning in the contexts of psychology and neuroscience. These publications are useful companions to this chapter and the following chapter on reinforcement learning and neuroscience.

14.1 Dayan, Niv, Seymour, and Daw (2006) focused on interactions between classical and instrumental conditioning, particularly situations where classically-conditioned and instrumental responses are in conflict. They proposed a Q-learning framework for modeling aspects of this interaction. Modayil and Sutton (2014) used a mobile robot to demonstrate the effectiveness of a control method combining a fixed response with online prediction learning. Calling this *Pavlovian control*, they emphasized that it differs from the usual control methods of reinforcement learning, being based on predictively executing fixed responses and not on reward maximization. The electro-mechanical machine of Ross (1933) and especially the learning version of Walter's turtle (Walter, 1951) were very early illustrations of Pavlovian control. What is now called Pavlovian-instrumental transfer was first observed by Estes (1943, 1948).

14.2.1 Kamin (1968) first reported blocking, now commonly known as Kamin blocking, in classical conditioning. Moore and Schmajuk (2008) provide an excellent summary of the blocking phenomenon, the research it stimulated, and its lasting influence on animal learning theory. Gibbs, Cool, Land, Kehoe, and Gormezano (1991) describe second-order conditioning of the rabbit's nictitating membrane response and its relationship to conditioning with serial-compound stimuli. Finch and Culler (1934) reported obtaining fifth-order conditioning of a dog's foreleg withdrawal "when the *motivation* of the animal is maintained through the various orders."

14.2.2 The idea built into the Rescorla–Wagner model that learning occurs when animals are surprised is derived from Kamin (1969). Models of classical conditioning other than Rescorla and Wagner's include the models of Klopf (1988), Grossberg (1975), Mackintosh (1975), Moore and Stickney (1980), Pearce and Hall (1980), and Courville, Daw, and Touretzky (2006). Schmajuk (2008) review models of classical conditioning.

14.2.3 An early version of the TD model of classical conditioning appeared in Sutton and Barto (1981), which also included the early model's prediction that temporal primacy overrides blocking, later shown by Kehoe, Scheurs, and Graham (1987) to occur in the rabbit nictitating membrane preparation. Sutton and Barto (1981) contains the earliest recognition of the near identity between the Rescorla–Wagner model and the Least-Mean-Square (LMS), or Widrow-Hoff, learning rule (Widrow and Hoff, 1960). This early model was revised following Sutton's development of the TD algorithm (Sutton, 1984, 1988) and was first presented as the TD model in Sutton and Barto (1987) and more completely in Sutton and Barto (1990), upon which this section is largely based. Additional exploration of the TD model and its possible neural implementation

was conducted by Moore and colleagues (Moore, Desmond, Berthier, Blazis, Sutton, and Barto, 1986; Moore and Blazis, 1989; Moore, Choi, and Brunzell, 1998; Moore, Marks, Castagna, and Polewan, 2001). Klopf's (1988) drive-reinforcement theory of classical conditioning extends the TD model to address additional experimental details, such as the S-shape of acquisition curves. In some of these publications TD is taken to mean Time Derivative instead of Temporal Difference.

14.2.4 Ludvig, Sutton, and Kehoe (2012) evaluated the performance of the TD model in previously unexplored tasks involving classical conditioning and examined the influence of various stimulus representations, including the microstimulus representation that they introduced earlier (Ludvig, Sutton, and Kehoe, 2008). Earlier investigations of the influence of various stimulus representations and their possible neural implementations on response timing and topography in the context of the TD model are those of Moore and colleagues cited above. Although not in the context of the TD model, representations like the microstimulus representation of Ludvig et al. (2012) have been proposed and studied by Grossberg and Schmajuk (1989), Brown, Bullock, and Grossberg (1999), Buhusi and Schmajuk (1999), and Machado (1997).

14.4 Section 1.7 includes comments on the history of trial-and-error learning and the Law of Effect. The idea that Thorndikes cats might have been exploring according to an instinctual context-specific ordering over actions rather than by just selecting from a set of instinctual impulses was suggested by Peter Dayan (personal communication). Selfridge, Sutton, and Barto (1985) illustrated the effectiveness of shaping in a pole-balancing reinforcement learning task. Other examples of shaping in reinforcement learning are Gullapalli and Barto (1992), Mahadevan and Connell (1992), Mataric (1994), Dorigo and Colombette (1994), Saksida, Raymond, and Touretzky (1997), and Randløv and Alstrøm (1998). Ng (2003) and Ng, Harada, and Russell (1999) used the term shaping in a sense somewhat different from Skinner's, focussing on the problem of how to alter the reward signal without altering the set of optimal policies.

Dickinson and Balleine (2002) discuss the complexity of the interaction between learning and motivation. Wise (2004) provides an overview of reinforcement learning and its relation to motivation. Daw and Shohamy (2008) link motivation and learning to aspects of reinforcement learning theory. See also McClure, Daw, and Montague (2003), Niv, Joel, and Dayan (2006), Rangel et al. (2008), and Dayan and Berridge (2014). McClure et al. (2003), Niv, Daw, and Dayan (2005), and Niv, Daw, Joel, and Dayan (2007) present theories of behavioral vigor related to the reinforcement learning framework.

14.4 Spence, Hull's student and collaborator at Yale, elaborated the role of higher-order reinforcement in addressing the problem of delayed reinforcement (Spence, 1947). Learning over very long delays, as in taste-aversion conditioning with delays up to several hours, led to interference theories as alternatives to decaying-trace theories (e.g., Revusky and Garcia, 1970; Boakes and Costa, 2014). Other views of learning under delayed reinforcement invoke roles for awareness and working memory (e.g., Clark and Squire, 1998; Seo, Barraclough, and Lee, 2007).

14.5 Thistlethwaite (1951) is an extensive review of latent learning experiments up to the time of its publication. Ljung (1998) is an overview of model learning, or system identification, techniques in engineering. Gopnik, Glymour, Sobel, Schulz, Kushnir, and Danks (2004) present a Bayesian theory about how children learn models.

14.6 Connections between habitual and goal-directed behavior and model-free and model-based reinforcement learning were first proposed by Daw, Niv, and Dayan (2005). The hypothetical maze task used to explain habitual and goal-directed behavioral control is based on the explanation of Niv, Joel, and Dayan (2006). Dolan and Dayan (2013) review four generations of

experimental research related to this issue and discuss how it can move forward on the basis of reinforcement learning's model-free/model-based distinction. Dickinson (1980, 1985) and Dickinson and Balleine (2002) discuss experimental evidence related to this distinction. Donahoe and Burgos (2000) alternatively argue that model-free processes can account for the results of outcome-devaluation experiments. Dayan and Berridge (2014) argue that classical conditioning involves model-based processes. Rangel, Camerer, and Montague (2008) review many of the outstanding issues involving habitual, goal-directed, and Pavlovian modes of control.

Comments on Terminology— The traditional meaning of *reinforcement* in psychology is the strengthening of a pattern of behavior (by increasing either its intensity or frequency) as a result of an animal receiving a stimulus (or experiencing the omission of a stimulus) in an appropriate temporal relationship with another stimulus or with a response. Reinforcement produces changes that remain in future behavior. Sometimes in psychology reinforcement refers to the process of producing lasting changes in behavior, whether the changes strengthen or weaken a behavior pattern (Mackintosh, 1983). Letting reinforcement refer to weakening in addition to strengthening is at odds with the everyday meaning of reinforce, and its traditional use in psychology, but it is a useful extension that we have adopted here. In either case, a stimulus considered to be the cause of the behavioral change is called a *reinforcer*.

Psychologists do not generally use the specific phrase *reinforcement learning* as we do. Animal learning pioneers probably regarded reinforcement and learning as being synonymous, so it would be redundant to use both words. Our use of the phrase follows its use in computational and engineering research, influenced mostly by Minsky (1961). But the phrase is lately gaining currency in psychology and neuroscience, likely because strong parallels have surfaced between reinforcement learning algorithms and animal learning—parallels described in this chapter and the next.

According to common usage, a *reward* is an object or event that an animal will approach and work for. A reward may be given to an animal in recognition of its 'good' behavior, or given in order to make the animal's behavior 'better.' Similarly, a *penalty* is an object or event that the animal usually avoids and that is given as a consequence of 'bad' behavior, usually in order to change that behavior. *Primary reward* is reward due to machinery built into an animal's nervous system by evolution to improve its chances of survival and reproduction, e.g., reward produced by the taste of nourishing food, sexual contact, successful escape, and many other stimuli and events that predicted reproductive success over the animal's ancestral history. As explained in Section 14.2.1, *higher-order reward* is reward delivered by stimuli that predict primary reward, either directly or indirectly by predicting other stimuli that predict primary reward. Reward is *secondary* if its rewarding quality is the result of directly predicting primary reward.

I this book we call R_t the 'reward signal at time t' or sometimes just the 'reward at time t,' but we do not think of it as an object or event in the agent's environment. Because R_t is a number—not an object or an event—it is more like a reward signal in neuroscience, which is a signal internal to the brain, like the activity of neurons, that influences decision making and learning. This signal might be triggered when the animal perceives an attractive (or an aversive) object, but it can also be triggered by things that do not physically exist in the animal's external environment, such as memories, ideas, or hallucinations. Because our R_t can be positive, negative, or zero, it might be better to call a negative R_t a penalty, and an R_t equal to zero a neutral signal, but for simplicity we generally avoid these terms.

In reinforcement learning, the process that generates all the R_ts defines the problem the agent is trying to solve. The agent's objective is to keep the magnitude of R_t as large as possible over time. In this respect, R_t is like primary reward for an animal if we think of the problem the animal faces as the problem of obtaining as much primary reward as possible over its lifetime (and thereby, through the prospective "wisdom" of evolution, improve its chances of solving its real problem, which is to pass its genes on to future generations). However, as we suggest in Chapter 15, it is unlikely that there is a single "master" reward signal like R_t in an animal's brain.

Not all reinforcers are rewards or penalties. Sometimes reinforcement is not the result of an animal

receiving a stimulus that evaluates its behavior by labeling the behavior good or bad. A behavior pattern can be reinforced by a stimulus that arrives to an animal no matter how the animal behaved. As described in Section 14.1, whether the delivery of reinforcer depends, or does not depend, on preceding behavior is the defining difference between instrumental, or operant, conditioning experiments and classical, or Pavlovian, conditioning experiments. Reinforcement is at work in both types of experiments, but only in the former is it feedback that evaluates past behavior. (Though it has often been pointed out that even when the reinforcing US in a classical conditioning experiment is not contingent on the subject's preceding behavior, its reinforcing value can be influenced by this behavior, an example being that a closed eye makes an air puff to the eye less aversive.)

The distinction between reward signals and reinforcement signals is a crucial point when we discuss neural correlates of these signals in the next chapter. Like a reward signal, for us, the reinforcement signal at any specific time is a positive or negative number, or zero. A reinforcement signal is the major factor directing changes a learning algorithm makes in an agent's policy, value estimates, or environment models. The definition that makes the most sense to us is that a reinforcement signal at any time is a number that multiplies (possibly along with some constants) a vector to determine parameter updates in some learning algorithm.

For some algorithms, the reward signal alone is the critical multiplier in the parameter-update equation. For these algorithms the reinforcement signal is the same as the reward signal. But for most of the algorithms we discuss in this book, reinforcement signals include terms in addition to the reward signal, an example being a TD error $\delta_t = R_{t+1} + \gamma V(S_{t+1}) - V(S_t)$, which is the reinforcement signal for TD state-value learning (and analogous TD errors for action-value learning). In this reinforcement signal, R_{t+1} is the *primary reinforcement* contribution, and the temporal difference in predicted values, $\gamma V(S_{t+1}) - V(S_t)$ (or an analogous temporal difference for action values), is the *conditioned reinforcement* contribution. Thus, whenever $\gamma V(S_{t+1}) - V(S_t) = 0$, δ_t signals 'pure' primary reinforcement; and whenever $R_{t+1} = 0$, it signals 'pure' conditioned reinforcement, but it often signals a mixture of these. Note as we mentioned in Section 6.1, this δ_t is not available until time $t+1$. We therefore think of δ_t as the reinforcement signal at time $t+1$, which is fitting because it reinforces predictions and/or actions made earlier at step t.

A possible source of confusion is the terminology used by the famous psychologist B.F. Skinner and his followers. For Skinner, positive reinforcement occurs when the consequences of an animal's behavior increase the frequency of that behavior; punishment occurs when the behavior's consequences decrease that behavior's frequency. Negative reinforcement occurs when behavior leads to the removal of an aversive stimulus (that is, a stimulus the animal does not like), thereby increasing the frequency of that behavior. Negative punishment, on the other hand, occurs when behavior leads to the removal of an appetitive stimulus (that is, a stimulus the animal likes), thereby decreasing the frequency of that behavior. We find no critical need for these distinctions because our approach is more abstract than this, with both reward and reinforcement signals allowed to take on both positive and negative values. (But note especially that when our reinforcement signal is negative, it is not the same as Skinner's negative reinforcement.)

On the other hand, it has often been pointed out that using a single number as a reward or a penalty signal, depending only on its sign, is at odds with the fact that animals' appetitive and aversive systems have qualitatively different properties and involve different brain mechanisms. This points to a direction in which the reinforcement learning framework might be developed in the future to exploit computational advantages of separate appetitive and aversive systems, but for now we are passing over these possibilities.

Another discrepancy in terminology is how we use the word *action*. To many cognitive scientists, an action is purposeful in the sense of being the result of an animal's knowledge about the relationship between the behavior in question and the consequences of that behavior. An action is goal-directed and the result of a decision, in contrast to a response, which is triggered by a stimulus; the result of a reflex or

a habit. We use the word action without differentiating among what others call actions, decisions, and responses. These are important distinctions, but for us they are encompassed by differences between model-free and model-based reinforcement learning algorithms, which we discussed above in relation to habitual and goal-directed behavior in Section 14.6. Dickinson (1985) discusses the distinction between responses and actions.

A term used a lot in this book is *control*. What we mean by control is entirely different from what it means to animal learning psychologists. By control we mean that an agent influences its environment to bring about states or events that the agent prefers: the agent exerts control over its environment. This is the sense of control used by control engineers. In psychology, on the other hand, control typically means that an animal's behavior is influenced by—is controlled by—the stimuli the animal receives (stimulus control) or the reinforcement schedule it experiences. Here the environment is controlling the agent. Control in this sense is the basis of behavior modification therapy. Of course, both of these directions of control are at play when an agent interacts with its environment, but our focus is on the agent as controller; not the environment as controller. A view equivalent to ours, and perhaps more illuminating, is that the agent is actually controlling the input it receives from its environment (Powers, 1973). This is *not* what psychologists mean by stimulus control.

Sometimes reinforcement learning is understood to refer solely to learning policies directly from rewards (and penalties) without the involvement of value functions or environment models. This is what psychologists call stimulus-response, or S-R, learning. But for us, along with most of today's psychologists, reinforcement learning is much broader than this, including in addition to S-R learning, methods involving value functions, environment models, planning, and other processes that are commonly thought to belong to the more cognitive side of mental functioning.

Chapter 15

Neuroscience

Neuroscience is the multidisciplinary study of nervous systems: how they regulate bodily functions; control behavior; change over time as a result of development, learning, and aging; and how cellular and molecular mechanisms make these functions possible. One of the most exciting aspects of reinforcement learning is the mounting evidence from neuroscience that the nervous systems of humans and many other animals implement algorithms that correspond in striking ways to reinforcement learning algorithms. The main objective of this chapter is to explain these parallels and what they suggest about the neural basis of reward-related learning in animals.

The most remarkable point of contact between reinforcement learning and neuroscience involves dopamine, a chemical deeply involved in reward processing in the brains of mammals. Dopamine appears to convey temporal-difference (TD) errors to brain structures where learning and decision making take place. This parallel is expressed by the *reward prediction error hypothesis of dopamine neuron activity*, a hypothesis that resulted from the convergence of computational reinforcement learning and results of neuroscience experiments. In this chapter we discuss this hypothesis, the neuroscience findings that led to it, and why it is a significant contribution to understanding brain reward systems. We also discuss parallels between reinforcement learning and neuroscience that are less striking than this dopamine/TD-error parallel but that provide useful conceptual tools for thinking about reward-based learning in animals. Other elements of reinforcement learning have the potential to impact the study of nervous systems, but their connections to neuroscience are still relatively undeveloped. We discuss several of these evolving connections that we think will grow in importance over time.

As we outlined in the history section of this book's introductory chapter (Section 1.7), many aspects of reinforcement learning were influenced by neuroscience. A second objective of this chapter is to acquaint readers with ideas about brain function that have contributed to our approach to reinforcement learning. Some elements of reinforcement learning are easier to understand when seen in light of theories of brain function. This is particularly true for the idea of the eligibility trace, one of the basic mechanisms of reinforcement learning, that originated as a conjectured property of synapses, the structures by which nerve cells—neurons—communicate with one another.

In this chapter we do not delve very deeply into the enormous complexity of the neural systems underlying reward-based learning in animals: this chapter is too short, and we are not neuroscientists. We do not try to describe—or even to name—the very many brain structures and pathways, or any of the molecular mechanisms, believed to be involved in these processes. We also do not do justice to hypotheses and models that are alternatives to those that align so well with reinforcement learning. It should not be surprising that there are differing views among experts in the field. We can only provide a glimpse into this fascinating and developing story. We hope, though, that this chapter convinces you that a very fruitful channel has emerged connecting reinforcement learning and its theoretical

underpinnings to the neuroscience of reward-based learning in animals.

Many excellent publications cover links between reinforcement learning and neuroscience, some of which we cite in this chapter's final section. Our treatment differs from most of these because we assume familiarity with reinforcement learning as presented in the earlier chapters of this book, but we do not assume knowledge of neuroscience. We begin with a brief introduction to the neuroscience concepts needed for a basic understanding of what is to follow.

15.1 Neuroscience Basics

Some basic information about nervous systems is helpful for following what we cover in this chapter. Terms that we refer to later are italicized. Skipping this section will not be a problem if you already have an elementary knowledge of neuroscience.

Neurons, the main components of nervous systems, are cells specialized for processing and transmitting information using electrical and chemical signals. They come in many forms, but a neuron typically has a cell body, *dendrites*, and a single *axon*. Dendrites are structures that branch from the cell body to receive input from other neurons (or to also receive external signals in the case of sensory neurons). A neuron's axon is a fiber that carries the neuron's output to other neurons (or to muscles or glands). A neuron's output consists of sequences of electrical pulses called *action potentials* that travel along the axon. Action potentials are also called *spikes*, and a neuron is said to *fire* when it generates a spike. In models of neural networks it is common to use real numbers to represent a neuron's *firing rate*, the average number of spikes per some unit of time.

A neuron's axon can branch widely so that the neuron's action potentials reach many targets. The branching structure of a neuron's axon is called the neuron's *axonal arbor*. Because the conduction of an action potential is an active process, not unlike the burning of a fuse, when an action potential reaches an axonal branch point it "lights up" action potentials on all of the outgoing branches (although propagation to a branch can sometimes fail). As a result, the activity of a neuron with a large axonal arbor can influence many target sites.

A *synapse* is a structure generally at the termination of an axon branch that mediates the communication of one neuron to another. A synapse transmits information from the *presynaptic* neuron's axon to a dendrite or cell body of the *postsynaptic* neuron. With a few exceptions, synapses release a chemical *neurotransmitter* upon the arrival of an action potential from the presynaptic neuron. (The exceptions are cases of direct electric coupling between neurons, but these will not concern us here.) Neurotransmitter molecules released from the presynaptic side of the synapse diffuse across the *synaptic cleft*, the very small space between the presynaptic ending and the postsynaptic neuron, and then bind to receptors on the surface of the postsynaptic neuron to excite or inhibit its spike-generating activity, or to modulate its behavior in other ways. A particular neurotransmitter may bind to several different types of receptors, with each producing a different effect on the postsynaptic neuron. For example, there are at least five different receptor types by which the neurotransmitter dopamine can affect a postsynaptic neuron. Many different chemicals have been identified as neurotransmitters in animal nervous systems.

A neuron's *background* activity is its level of activity, usually its firing rate, when the neuron does not appear to be driven by synaptic input related to the task of interest to the experimenter, for example, when the neuron's activity is not correlated with a stimulus delivered to a subject as part of an experiment. Background activity can be irregular due to input from the wider network, or due to noise within the neuron or its synapses. Sometimes background activity is the result of dynamic processes intrinsic to the neuron. A neuron's *phasic* activity, in contrast to its background activity, consists of bursts of spiking activity usually caused by synaptic input. Activity that varies slowly and often in a graded manner, whether as background activity or not, is called a neuron's *tonic* activity.

The strength or effectiveness by which the neurotransmitter released at a synapse influences the postsynaptic neuron is the synapse's *efficacy*. One way a nervous system can change through experience is through changes in synaptic efficacies as a result of combinations of the activities of the presynaptic and postsynaptic neurons, and sometimes by the presence of a *neuromodulator*, which is a neurotransmitter having effects other than, or in addition to, direct fast excitation or inhibition.

Brains contain several different neuromodulation systems consisting of clusters of neurons with widely branching axonal arbors, with each system using a different neurotransmitter. Neuromodulation can alter the function of neural circuits, mediate motivation, arousal, attention, memory, mood, emotion, sleep, and body temperature. Important here is that a neuromodulatory system can distribute something like a scalar signal, such as a reinforcement signal, to alter the operation of synapses in widely distributed sites critical for learning.

The ability of synaptic efficacies to change is called *synaptic plasticity*. It is one of the primary mechanisms responsible for learning. The parameters, or weights, adjusted by learning algorithms correspond to synaptic efficacies. As we detail below, modulation of synaptic plasticity via the neuromodulator dopamine is a plausible mechanism for how the brain might implement learning algorithms like many of those described in this book.

15.2 Reward Signals, Reinforcement Signals, Values, and Prediction Errors

Links between neuroscience and computational reinforcement learning begin as parallels between signals in the brain and signals playing prominent roles in reinforcement learning theory and algorithms. In Chapter 3 we said that any problem of learning goal-directed behavior can be reduced to the three signals representing actions, states, and rewards. However, to explain links that have been made between neuroscience and reinforcement learning, we have to be less abstract than this and consider other reinforcement learning signals that correspond, in certain ways, to signals in the brain. In addition to reward signals, these include reinforcement signals (which we argue are different from reward signals), value signals, and signals conveying prediction errors. When we label a signal by its function in this way, we are doing it in the context of reinforcement learning theory in which the signal corresponds to a term in an equation or an algorithm. On the other hand, when we refer to a signal in the brain, we mean a physiological event such as a burst of action potentials or the secretion of a neurotransmitter. Labeling a neural signal by its function, for example calling the phasic activity of a dopamine neuron a reinforcement signal, means that the neural signal behaves like, and is conjectured to function like, the corresponding theoretical signal.

Uncovering evidence for these correspondences involves many challenges. Neural activity related to reward processing can be found in nearly every part of the brain, and it is difficult to interpret results unambiguously because representations of different reward-related signals tend to be highly correlated with one another. Experiments need to be carefully designed to allow one type of reward-related signal to be distinguished with any degree of certainty from others—or from an abundance of other signals not related to reward processing. Despite these difficulties, many experiments have been conducted with the aim of reconciling aspects of reinforcement learning theory and algorithms with neural signals, and some compelling links have been established. To prepare for examining these links, in the rest of this section we remind the reader of what various reward-related signals mean according to reinforcement learning theory.

In our Comments on Terminology at the end of the previous chapter, we said that R_t is like a reward signal in an animal's brain and not as an object or event in the animal's environment. In reinforcement learning, the reward signal (along with an agent's environment) defines the problem a reinforcement learning agent is trying to solve. It this respect, R_t is like a signal in an animal's brain that distributes

primary reward to sites throughout the brain. But it is unlikely that a unitary master reward signal like R_t exists in an animal's brain. It is best to think of R_t as an abstraction summarizing the overall effect of a multitude of neural signals generated by many systems in the brain that assess the rewarding or punishing qualities of sensations and states.

Reinforcement signals in reinforcement learning are different from reward signals. The function of a reinforcement signal is to direct the changes a learning algorithm makes in an agent's policy, value estimates, or environment models. For a TD method, for instance, the reinforcement signal at time t is the TD error $\delta_{t-1} = R_t + \gamma V(S_t) - V(S_{t-1})$.[1] The reinforcement signal for some algorithms could be just the reward signal, but for most of the algorithms we consider the reinforcement signal is the reward signal adjusted by other information, such as the value estimates in TD errors.

Estimates of state values or of action values, that is, V or Q, specify what is good or bad for the agent over the long run. They are predictions of the total reward an agent can expect to accumulate over the future. Agents make good decisions by selecting actions leading to states with the largest estimated state values, or by selecting actions with the largest estimated action values.

Prediction errors measure discrepancies between expected and actual signals or sensations. Reward prediction errors (RPEs) specifically measure discrepancies between the expected and the received reward signal, being positive when the reward signal is greater than expected, and negative otherwise. TD errors like (6.5) are special kinds RPEs that signal discrepancies between current and earlier expectations of reward over the long-term. When neuroscientists refer to RPEs they generally (though not always) mean TD RPEs, which we simply call TD errors throughout this chapter. Also in this chapter, a TD error is generally one that does not depend on actions, as opposed to TD errors used in learning action-values by algorithms like Sarsa and Q-learning. This is because the most well-known links to neuroscience are stated in terms of action-free TD errors, but we do not mean to rule out possible similar links involving action-dependent TD errors. (TD errors for predicting signals other than rewards are useful too, but that case will not concern us here. See, for example, Modayil, White, and Sutton, 2014.)

One can ask many questions about links between neuroscience data and these theoretically-defined signals. Is an observed signal more like a reward signal, a value signal, a prediction error, a reinforcement signal, or something altogether different? And if it is an error signal, is it an RPE, a TD error, or a simpler error like the Rescorla–Wagner error (14.3)? And if it is a TD error, does it depend on actions like the TD error of Q-learning or Sarsa? As indicated above, probing the brain to answer questions like these is extremely difficult. But experimental evidence suggests that one neurotransmitter, specifically the neurotransmitter dopamine, signals RPEs, and further, that the phasic activity of dopamine-producing neurons in fact conveys TD errors (see Section 15.1 for a definition of phasic activity). This evidence led to the *reward prediction error hypothesis of dopamine neuron activity*, which we describe next.

15.3 The Reward Prediction Error Hypothesis

The *reward prediction error hypothesis of dopamine neuron activity* proposes that one of the functions of the phasic activity of dopamine-producing neurons in mammals is to deliver an error between an old and a new estimate of expected future reward to target areas throughout the brain. This hypothesis (though not in these exact words) was first explicitly stated by Montague, Dayan, and Sejnowski (1996), who showed how the TD error concept from reinforcement learning accounts for many features of the phasic activity of dopamine neurons in mammals. The experiments that led to this hypothesis were performed in the 1980s and early 1990s in the laboratory of neuroscientist Wolfram Schultz. Section 15.5 describes

[1] As we mentioned in Section 6.1, δ_t in our notation is defined to be $R_{t+1} + \gamma V(S_{t+1}) - V(S_t)$, so δ_t is not available until time $t+1$. The TD error *available* at t is actually $\delta_{t-1} = R_t + \gamma V(S_t) - V(S_{t-1})$. Since we are thinking of time steps as very small, or even infinitesimal, time intervals, one should not attribute undue importance to this one-step time shift.

these influential experiments, Section 15.6 explains how the results of these experiments align with TD errors, and the Bibliographical and Historical Remarks section at the end of this chapter includes a guide to the literature surrounding the development of this influential hypothesis.

Montague et al. (1996) compared the TD errors of the TD model of classical conditioning with the phasic activity of dopamine-producing neurons during classical conditioning experiments. Recall from Section 14.2 that the TD model of classical conditioning is basically the semi-gradient-descent TD(λ) algorithm with linear function approximation. Montague et al. made several assumptions to set up this comparison. First, since a TD error can be negative but neurons cannot have a negative firing rate, they assumed that the quantity corresponding to dopamine neuron activity is $\delta_{t-1} + b_t$, where b_t is the background firing rate of the neuron. A negative TD error corresponds to a drop in a dopamine neuron's firing rate below its background rate.[2]

A second assumption was needed about the states visited in each classical conditioning trial and how they are represented as inputs to the learning algorithm. This is the same issue we discussed in Section 14.2.4 for the TD model. Montague et al. chose a complete serial compound (CSC) representation as shown in the left column of Figure 14.2, but where the sequence of short-duration internal signals continues until the onset of the US, which here is the arrival of a non-zero reward signal. This representation allows the TD error to mimic the fact that dopamine neuron activity not only predicts a future reward, but that it is also sensitive to *when* after a predictive cue that reward is expected to arrive. There has to be some way to keep track of the time between sensory cues and the arrival of reward. If a stimulus initiates a sequence of internal signals that continues after the stimulus ends, and if there is a different signal for each time step following the stimulus, then each time step after the stimulus is represented by a distinct state. Thus, the TD error, being state-dependent, can be sensitive to the timing of events within a trial.

In simulated trials with these assumptions about background firing rate and input representation, TD errors of the TD model are remarkably similar to dopamine neuron phasic activity. Previewing our description of details about these similarities in Section 15.5 below, the TD errors parallel the following features of dopamine neuron activity: 1) the phasic response of a dopamine neuron only occurs when a rewarding event is unpredicted; 2) early in learning, neutral cues that precede a reward do not cause substantial phasic dopamine responses, but with continued learning these cues gain predictive value and come to elicit phasic dopamine responses; 3) if an even earlier cue reliably precedes a cue that has already acquired predictive value, the phasic dopamine response shifts to the earlier cue, ceasing for the later cue; and 3) if after learning, the predicted rewarding event is omitted, a dopamine neuron's response decreases below its baseline level shortly after the expected time of the rewarding event.

Although not every dopamine neuron monitored in the experiments of Schultz and colleagues behaved in all of these ways, the striking correspondence between the activities of most of the monitored neurons and TD errors lends strong support to the reward prediction error hypothesis. There are situations, however, in which predictions based on the hypothesis do not match what is observed in experiments. The choice of input representation is critical to how closely TD errors match some of the details of dopamine neuron activity, particularly details about the timing of dopamine neuron responses. Different ideas, some of which we discuss below, have been proposed about input representations and other features of TD learning to make the TD errors fit the data better, though the main parallels appear with the CSC representation that Montague et al. used. Overall, the reward prediction error hypothesis has received wide acceptance among neuroscientists studying reward-based learning, and it has proven to be remarkably resilient in the face of accumulating results from neuroscience experiments.

To prepare for our description of the neuroscience experiments supporting the reward prediction error hypothesis, and to provide some context so that the significance of the hypothesis can be appreciated, we next present some of what is known about dopamine, the brain structures it influences, and how it

[2]In the literature relating TD errors to the activity of dopamine neurons, their δ_t is the same as our $\delta_{t-1} = R_t + \gamma V(S_t) - V(S_{t-1})$.

is involved in reward-based learning.

15.4 Dopamine

Dopamine is produced as a neurotransmitter by neurons whose cell bodies lie mainly in two clusters of neurons in the midbrain of mammals: the substantia nigra pars compacta (SNpc) and the ventral tegmental area (VTA). Dopamine plays essential roles in many processes in the mammalian brain. Prominent among these are motivation, learning, action-selection, most forms of addiction, and the disorders schizophrenia and Parkinson's disease. Dopamine is called a neuromodulator because it performs many functions other than direct fast excitation or inhibition of targeted neurons. Although much remains unknown about dopamine's functions and details of its cellular effects, it is clear that it is fundamental to reward processing in the mammalian brain. Dopamine is not the only neuromodulator involved in reward processing, and its role in aversive situations—punishment—remains controversial. Dopamine also can function differently in non-mammals. But no one doubts that dopamine is essential for reward-related processes in mammals, including humans.

An early, traditional view is that dopamine neurons broadcast a reward signal to multiple brain regions implicated in learning and motivation. This view followed from a famous 1954 paper by James Olds and Peter Milner that described the effects of electrical stimulation on certain areas of a rat's brain. They found that electrical stimulation to particular regions acted as a very powerful reward in controlling the rat's behavior: "... the control exercised over the animal's behavior by means of this reward is extreme, possibly exceeding that exercised by any other reward previously used in animal experimentation" (Olds and Milner, 1954). Later research revealed that the sites at which stimulation was most effective in producing this rewarding effect excited dopamine pathways, either directly or indirectly, that ordinarily are excited by natural rewarding stimuli. Effects similar to these with rats were also observed with human subjects. These observations strongly suggested that dopamine neuron activity signals reward.

But if the reward prediction error hypothesis is correct—even if it accounts for only some features of a dopamine neuron's activity—this traditional view of dopamine neuron activity is not entirely correct: phasic responses of dopamine neurons signal reward prediction errors, not reward itself. In reinforcement learning's terms, a dopamine neuron's phasic response at a time t corresponds to $\delta_{t-1} = R_t + \gamma V(S_t) - V(S_{t-1})$, not to R_t.

Reinforcement learning theory and algorithms help reconcile the reward-prediction-error view with the conventional notion that dopamine signals reward. In many of the algorithms we discuss in this book, δ functions as a reinforcement signal, meaning that it is the main driver of learning. For example, δ is the critical factor in the TD model of classical conditioning, and δ is the reinforcement signal for learning both a value function and a policy in an actor–critic architecture (Sections 13.5 and 15.7). Action-dependent forms of δ are reinforcement signals for Q-learning and Sarsa. The reward signal R_t is a crucial component of δ_{t-1}, but it is not the complete determinant of its reinforcing effect in these algorithms. The additional term $\gamma V(S_t) - V(S_{t-1})$ is the higher-order reinforcement part of δ_{t-1}, and even if reward occurs ($R_t \neq 0$), the TD error can be silent if the reward is fully predicted (which is fully explained in Section 15.6 below).

A closer look at Olds' and Milner's 1954 paper, in fact, reveals that it is mainly about the reinforcing effect of electrical stimulation in an instrumental conditioning task. Electrical stimulation not only energized the rats' behavior—through dopamine's effect on motivation—it also led to the rats quickly learning to stimulate themselves by pressing a lever, which they would do frequently for long periods of time. The activity of dopamine neurons triggered by electrical stimulation reinforced the rats' lever pressing.

More recent experiments using optogenetic methods clinch the role of phasic responses of dopamine

neurons as reinforcement signals. These methods allow neuroscientists to precisely control the activity of selected neuron types at a millisecond timescale in awake behaving animals. Optogenetic methods introduce light-sensitive proteins into selected neuron types so that these neurons can be activated or silenced by means of flashes of laser light. The first experiment using optogenetic methods to study dopamine neurons showed that optogenetic stimulation producing phasic activation of dopamine neurons in mice was enough to condition the mice to prefer the side of a chamber where they received this stimulation as compared to the chamber's other side where they received no, or lower-frequency, stimulation (Tsai et al. 2009). In another example, Steinberg et al. (2013) used optogenetic activation of dopamine neurons to create artificial bursts of dopamine neuron activity in rats at the times when rewarding stimuli were expected but omitted—times when dopamine neuron activity normally pauses. With these pauses replaced by artificial bursts, responding was sustained when it would ordinarily decrease due to lack of reinforcement (in extinction trials), and learning was enabled when it would ordinarily be blocked due to the reward being already predicted (the blocking paradigm; Section 14.2.1).

Additional evidence for the reinforcing function of dopamine comes from optogenetic experiments with fruit flies, except in these animals dopamine's effect is the opposite of its effect in mammals: optically triggered bursts of dopamine neuron activity act just like electric foot shock in reinforcing avoidance behavior, at least for the population of dopamine neurons activated (Claridge-Chang et al. 2009). Although none of these optogenetic experiments showed that phasic dopamine neuron activity is specifically like a TD error, they convincingly demonstrated that phasic dopamine neuron activity acts just like δ acts (or perhaps like *minus* δ acts in fruit flies) as the reinforcement signal in algorithms for both prediction (classical conditioning) and control (instrumental conditioning).

Dopamine neurons are particularly well suited to broadcasting a reinforcement signal to many areas of the brain. These neurons have huge axonal arbors, each releasing dopamine at 100 to 1,000 times more synaptic sites than reached by the axons of typical neurons. Figure 15.1 shows the axonal arbor of a single dopamine neuron whose cell body is in the SNpc of a rat's brain. Each axon of a SNpc or VTA dopamine neuron makes roughly 500,000 synaptic contacts on the dendrites of neurons in targeted brain areas.

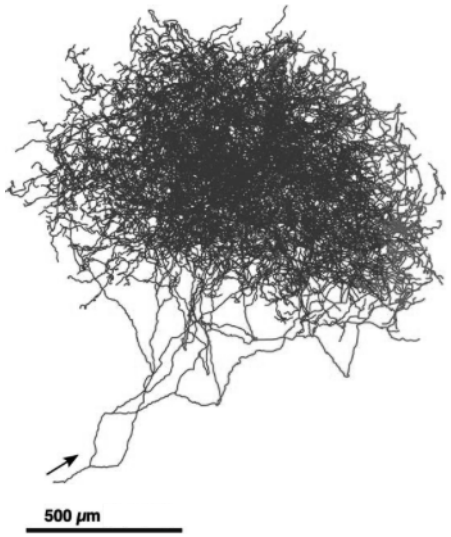


Figure 15.1: Axonal arbor of a single neuron producing dopamine as a neurotransmitter whose cell body is in the SNpc of a rat's brain. These axons make synaptic contacts with a huge number of dendrites of neurons in targeted brain areas. Adapted from *Journal of Neuroscience*, Matsuda, Furuta, Nakamura, Hioki, Fujiyama, Arai, and Kaneko, volume 29, 2009, page 451.

If dopamine neurons broadcast a reinforcement signal like reinforcement learning's δ, then since this is a scalar signal, i.e., a single number, all dopamine neurons in both the SNpc and VTA would be expected to activate more-or-less identically so that they would act in near synchrony to send the same signal to all of the sites their axons target. Although it has been a common belief that dopamine neurons do act together like this, modern evidence is pointing to the more complicated picture that different subpopulations of dopamine neurons respond to input differently depending on the structures to which they send their signals and the different ways these signals act on their target structures. Dopamine has functions other than signaling RPEs, and even for dopamine neurons that do signal RPEs, it can make sense to send different RPEs to different structures depending on the roles these structures play in producing reinforced behavior. This is beyond what we treat in any detail in this book, but vector-valued RPE signals make sense from the perspective of reinforcement learning when decisions can be decomposed into separate sub-decisions, or more generally, as a way to address the *structural* version of the credit assignment problem: How do you distribute credit for success (or blame for failure) of a decision among the many component structures that could have been involved in producing it? We say a bit more about this in Section 15.10 below.

The axons of most dopamine neurons make synaptic contact with neurons in the frontal cortex and the basal ganglia, areas of the brain involved in voluntary movement, decision making, learning, and cognitive functions such a planning. Since most ideas relating dopamine to reinforcement learning focus on the basal ganglia, and the connections from dopamine neurons are particularly dense there, we focus on the basal ganglia here. The basal ganglia are a collection of neuron groups, or nuclei, lying at the base of the forebrain. The main input structure of the basal ganglia is called the striatum. Essentially all of the cerebral cortex, among other structures, provides input to the striatum. The activity of cortical neurons conveys a wealth of information about sensory input, internal states, and motor activity. The axons of cortical neurons make synaptic contacts on the dendrites of the main input/output neurons of the striatum, called medium spiny neurons. Output from the striatum loops back via other basal ganglia nuclei and the thalamus to frontal areas of cortex, and to motor areas, making it possible for the striatum to influence movement, abstract decision processes, and reward processing. Two main subdivisions of the striatum are important for reinforcement learning: the dorsal striatum, primarily implicated in influencing action selection, and the ventral striatum, thought to be critical for different aspects of reward processing, including the assignment of affective value to sensations.

The dendrites of medium spiny neurons are covered with spines on whose tips the axons of neurons in the cortex make synaptic contact. Also making synaptic contact with these spines—in this case contacting the spine stems—are axons of dopamine neurons (Figure 15.2). This arrangement brings together presynaptic activity of cortical neurons, postsynaptic activity of medium spiny neurons, and input from dopamine neurons. What actually occurs at these spines is complex and not completely understood. Figure 15.2 hints at the complexity by showing two types of receptors for dopamine, receptors for glutamate—the neurotransmitter of the cortical inputs—and multiple ways that the various signals can interact. But evidence is mounting that changes in the efficacies of the synapses on the pathway from the cortex to the striatum, which neuroscientists call *corticostriatal synapses*, depend critically on appropriately-timed dopamine signals.

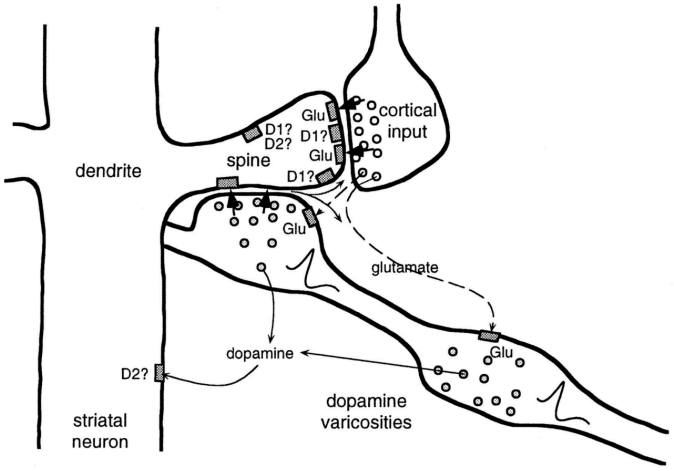


Figure 15.2: Spine of a striatal neuron showing input from both cortical and dopamine neurons. Axons of cortical neurons influence striatal neurons via corticostriatal synapses releasing the neurotransmitter glutamate at the tips of spines covering the dendrites of striatal neurons. An axon of a VTA or SNpc dopamine neuron is shown passing by the spine (from the lower right). "Dopamine varicosities" on this axon release dopamine at or near the spine stem, in an arrangement that brings together presynaptic input from cortex, postsynaptic activity of the striatal neuron, and dopamine, making it possible that several types of learning rules govern the plasticity of corticostriatal synapses. Each axon of a dopamine neuron makes synaptic contact with the stems of roughly 500,000 spines. Some of the complexity omitted from our discussion is shown here by other neurotransmitter pathways and multiple receptor types, such as D1 an D2 dopamine receptors by which dopamine can produce different effects at spines and other postsynaptic sites. From *Journal of Neurophysiology*, W. Schultz, vol. 80, 1998, page 10.

15.5 Experimental Support for the Reward Prediction Error Hypothesis

Dopamine neurons respond with bursts of activity to intense, novel, or unexpected visual and auditory stimuli that trigger eye and body movements, but very little of their activity is related to the movements themselves. This is surprising because degeneration of dopamine neurons is a cause of Parkinson's disease, whose symptoms include motor disorders, particularly deficits in self-initiated movement. Motivated by the weak relationship between dopamine neuron activity and stimulus-triggered eye and body movements, Romo and Schultz (1990) and Schultz and Romo (1990) took the first steps toward the reward prediction error hypothesis by recording the activity of dopamine neurons and muscle activity while monkeys moved their arms.

They trained two monkeys to reach from a resting hand position into a bin containing a bit of apple, a piece of cookie, or a raisin, when the monkey saw and heard the bin's door open. The monkey could then grab and bring the food to its mouth. After a monkey became good at this, it was trained on two additional tasks. The purpose of the first task was to see what dopamine neurons do when movements are self-initiated. The bin was left open but covered from above so that the monkey could not see inside but could reach in from below. No triggering stimuli were presented, and after the monkey reached for and ate the food morsel, the experimenter usually (though not always), silently and unseen by the monkey, replaced food in the bin by sticking it onto a rigid wire. Here too, the activity of the dopamine neurons Romo and Schultz monitored was not related to the monkey's movements, but a

large percentage of these neurons produced phasic responses whenever the monkey first touched a food morsel. These neurons did not respond when the monkey touched just the wire or explored the bin when no food was there. This was good evidence that the neurons were responding to the food and not to other aspects of the task.

The purpose of Romo and Schultz's second task was to see what happens when movements are triggered by stimuli. This task used a different bin with a movable cover. The sight and sound of the bin opening triggered reaching movements to the bin. In this case, Romo and Schultz found that after some period of training, the dopamine neurons no longer responded to the touch of the food but instead responded to the sight and sound of the opening cover of the food bin. The phasic responses of these neurons had shifted from the reward itself to stimuli predicting the availability of the reward. In a followup study, Romo and Schultz found that most of the dopamine neurons whose activity they monitored did not respond to the sight and sound of the bin opening outside the context of the behavioral task. These observations suggested that the dopamine neurons were responding neither to the initiation of a movement nor to the sensory properties of the stimuli, but were rather signaling an expectation of reward.

Schultz's group conducted many additional studies involving both SNpc and VTA dopamine neurons. A particular series of experiments was influential in suggesting that the phasic responses of dopamine neurons correspond to TD errors and not to simpler errors like those in the Rescorla–Wagner model (14.3). In the first of these experiments (Ljungberg, Apicella, and Schultz, 1992), monkeys were trained to depress a lever after a light was illuminated as a 'trigger cue' to obtain a drop of apple juice. As Romo and Schultz had observed earlier, many dopamine neurons initially responded to the reward—the drop of juice (Figure 15.3, top panel). But many of these neurons lost that reward response as training continued and developed responses instead to the illumination of the light that predicted the reward (Figure 15.3, middle panel). With continued training, lever pressing became faster while the number of dopamine neurons responding to the trigger cue decreased.

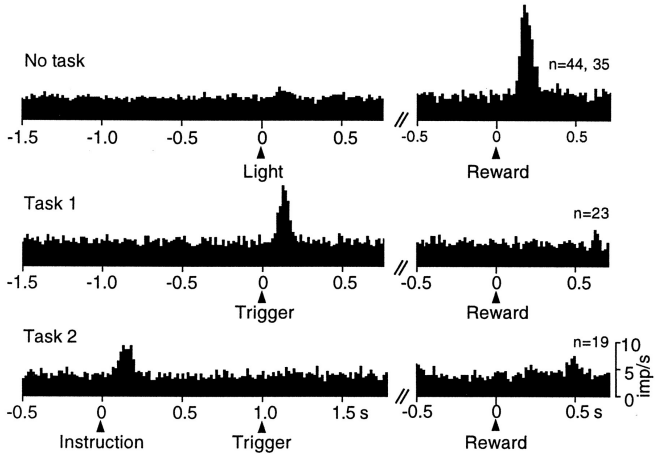


Figure 15.3: The response of dopamine neurons shifts from initial responses to primary reward to earlier predictive stimuli. These are plots of the number of action potentials produced by monitored dopamine neurons within small time intervals, averaged over all the monitored dopamine neurons (ranging from 23 to 44 neurons for these data). Top: dopamine neurons are activated by the unpredicted delivery of drop of apple juice. Middle: with learning, dopamine neurons developed responses to the reward-predicting trigger cue and lost responsiveness to the delivery of reward. Bottom: with the addition of an instruction cue preceding the trigger cue by 1 second, dopamine neurons shifted their responses from the trigger cue to the earlier instruction cue. From Schultz et al. (1995), MIT Press.

Following this study, the same monkeys were trained on a new task (Schultz, Apicella, and Ljungberg, 1993). Here the monkeys faced two levers, each with a light above it. Illuminating one of these lights was an 'instruction cue' indicating which of the two levers would produce a drop of apple juice. In this task, the instruction cue preceded the trigger cue of the previous task by a fixed interval of 1 second. The monkeys learned to withhold reaching until seeing the trigger cue, and dopamine neuron activity increased, but now the responses of the monitored dopamine neurons occurred almost exclusively to the earlier instruction cue and not to the trigger cue (Figure 15.3, bottom panel). Here again the number of dopamine neurons responding to the instruction cue was much reduced when the task was well learned. During learning across these tasks, dopamine neuron activity shifted from initially responding to the reward to responding to the earlier predictive stimuli, first progressing to the trigger stimulus then to the still earlier instruction cue. As responding moved earlier in time it disappeared from the later stimuli. This shifting of responses to earlier reward predictors, while losing responses to later predictors is a hallmark of TD learning (see, for example, Figure 14.5).

The task just described revealed another property of dopamine neuron activity shared with TD learning. The monkeys sometimes pressed the wrong key, that is, the key other than the instructed one, and consequently received no reward. In these trials, many of the dopamine neurons showed a sharp decrease in their firing rates below baseline shortly after the reward's usual time of delivery, and this happened without the availability of any external cue to mark the usual time of reward delivery (Figure 15.4). Somehow the monkeys were internally keeping track of the timing of the reward. (Response timing is one area where the simplest version of TD learning needs to be modified to account for some of the details of the timing of dopamine neuron responses. We consider this issue in the following section.)

The observations from the studies described above led Schultz and his group to conclude that dopamine neurons respond to unpredicted rewards, to the earliest predictors of reward, and that dopamine neuron activity decreases below baseline if a reward, or a predictor of reward, does not occur at its expected time. Researchers familiar with reinforcement learning were quick to recognize that these results are strikingly similar to how the TD error behaves as the reinforcement signal in a TD algorithm. The next section explores this similarity by working through a specific example in detail.

15.6 TD Error/Dopamine Correspondence

This section explains the correspondence between the TD error δ and the phasic responses of dopamine neurons observed in the experiments just described. We examine how δ changes over the course of learning in a task something like the one described above where a monkey first sees an instruction cue and then a fixed time later has to respond correctly to a trigger cue in order to obtain reward. We use a simple idealized version of this task, but we go into a lot more detail than is usual because we want to emphasize the theoretical basis of the parallel between TD errors and dopamine neuron activity.

The first simplifying assumption is that the agent has already learned the actions required to obtain reward. Then its task is just to learn accurate predictions of future reward for the sequence of states it experiences. This is then a prediction task, or more technically, a policy-evaluation task: learning the value function for a fixed policy (Sections 4.1 and 6.1). The value function to be learned assigns to each state a value that predicts the return that will follow that state if the agent selects actions according to the given policy, where the return is the (possibly discounted) sum of all the future rewards. This is unrealistic as a model of the monkey's situation because the monkey would likely learn these predictions at the same time that it is learning to act correctly (as would a reinforcement learning algorithm that learns policies as well as value functions, such as an actor–critic algorithm), but this scenario is simpler to describe than one in which a policy and a value function are learned simultaneously.

Now imagine that the agent's experience divides into multiple trials, in each of which the same sequence of states repeats, with a distinct state occurring on each time step during the trial. Further

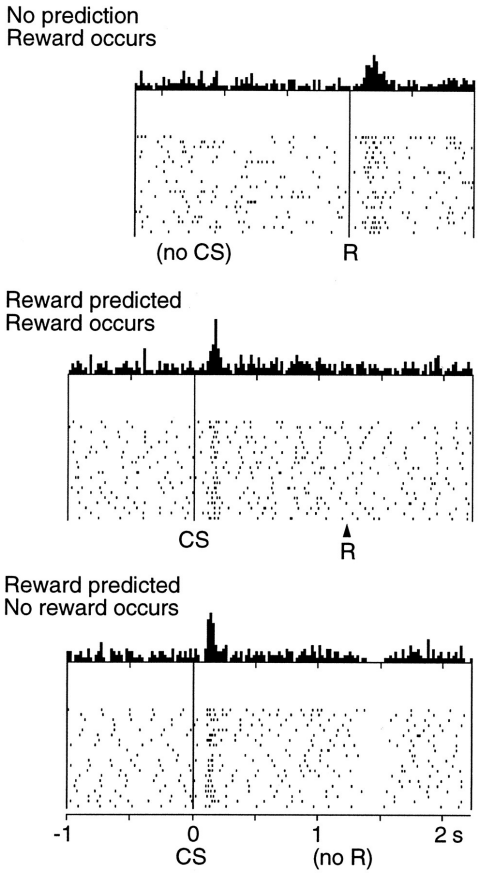


Figure 15.4: The response of dopamine neurons drops below baseline shortly after the time when an expected reward fails to occur. Top: dopamine neurons are activated by the unpredicted delivery of a drop of apple juice. Middle: dopamine neurons respond to a conditioned stimulus (CS) that predicts reward and do not respond to the reward itself. Bottom: when the reward predicted by the CS fails to occur, the activity of dopamine neurons drops below baseline shortly after the time the reward is expected to occur. At the top of each of these panels is shown the average number of action potentials produced by monitored dopamine neurons within small time intervals around the indicated times. The raster plots below show the activity patterns of the individual dopamine neurons that were monitored; each dot represents an action potential. From Schultz, Dayan, and Montague, A Neural Substrate of Prediction and Reward, *Science*, vol. 275, issue 5306, pages 1593-1598, March 14, 1997. Reprinted with permission from AAAS.

imagine that the return being predicted is limited to the return over a trial, which makes a trial analogous to a reinforcement learning episode as we have defined it. In reality, of course, the returns being predicted are not confined to single trials, and the time interval between trials is an important factor in determining what an animal learns. This is true for TD learning as well, but here we assume that returns do not accumulate over multiple trials. Given this, then, a trial in experiments like those conducted by Schultz and colleagues is equivalent to an episode of reinforcement learning. (Though in this discussion, we will use the term trial instead of episode to relate better to the experiments.)

As usual, we also need to make an assumption about how states are represented as inputs to the learning algorithm, an assumption that influences how closely the TD error corresponds to dopamine neuron activity. We discuss this issue later, but for now we assume the same CSC representation used by Montague et al. (1996) in which there is a separate internal stimulus for each state visited at each

time step in a trial. This reduces the process to the tabular case covered in the first part of this book. Finally, we assume that the agent uses TD(0) to learn a value function, V, stored in a lookup table initialized to be zero for all the states. We also assume that this is a deterministic task and that the discount factor, γ, is very nearly one so that we can ignore it.

Figure 15.5 shows the time courses of R, V, and δ at several stages of learning in this policy-evaluation task. The time axes represent the time interval over which a sequence of states is visited in a trial (where for clarity we omit showing individual states). The reward signal is zero throughout each trial except when the agent reaches the rewarding state, shown near the right end of the time line, when the reward signal becomes some positive number, say $R^\star$. The goal of TD learning is to predict the return for each state visited in a trial, which in this undiscounted case and given our assumption that predictions are confined to individual trials, is simply $R^\star$ for each state.

Preceding the rewarding state is a sequence of reward-predicting states, with the *earliest reward-predicting state* shown near the left end of the time line. This is like the state near the start of a trial, for example like the state marked by the instruction cue in a trial of the monkey experiment of Schultz et al. (1993) described above. It is the first state in a trial that reliably predicts that trial's reward. (Of course, in reality states visited on preceding trials are even earlier reward-predicting states, but since we are confining predictions to individual trials, these do not qualify as predictors of *this* trial's reward. Below we give a more satisfactory, though more abstract, description of an earliest reward-predicting state.) The *latest reward-predicting state* in a trial is the state immediately preceding the trial's rewarding state. This is the state near the far right end of the time line in Figure 15.5. Note that the rewarding state of a trial does not predict the return for that trial: the value of this state would come to predict the return over all the *following* trials, which here we are assuming to be zero in this episodic formulation.

Figure 15.5 shows the first-trial time courses of V and δ as the graphs labeled 'early in learning.'

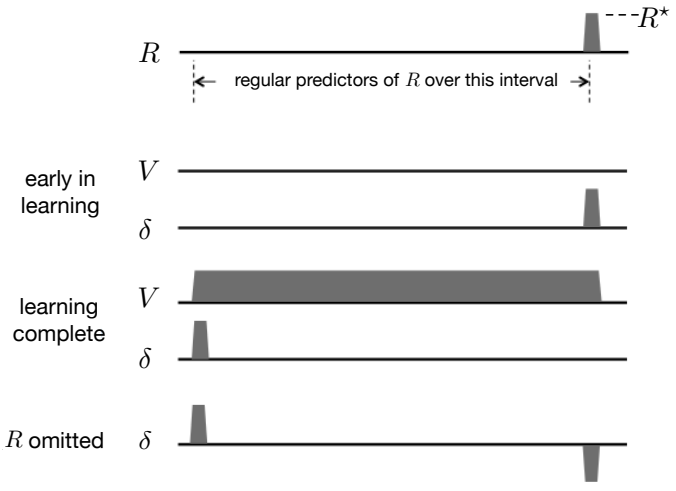


Figure 15.5: The behavior of the TD error δ during TD learning is consistent with features of the phasic activation of dopamine neurons. (Here δ is the TD error *available* at time t, i.e., δ_{t-1}). *Top*: a sequence of states, shown as an interval of regular predictors, is followed by a non-zero reward $R^\star$. *Early in learning*: the initial value function, V, and initial δ, which at first is equal to $R^\star$. *Learning complete*: the value function accurately predicts future reward, δ is positive at the earliest predictive state, and $\delta = 0$ at the time of the non-zero reward. $R^\star$ *omitted*: at the time the predicted reward is omitted, δ becomes negative. See text for a complete explanation of why this happens.

Because the reward signal is zero throughout the trial except when the rewarding state is reached, and all the V-values are zero, the TD error is also zero until it becomes $R^\star$ at the rewarding state. This follows because $\delta_{t-1} = R_t + V_t - V_{t-1} = R_t + 0 - 0 = R_t$, which is zero until it equals $R^\star$ when the reward occurs. Here V_t and V_{t-1} are respectively the estimated values of the states visited at times t and $t-1$ in a trial. The TD error at this stage of learning is analogous to a dopamine neuron responding to an unpredicted reward, e.g., a drop apple juice, at the start of training.

Throughout this first trial and all successive trials, TD(0) updates occur at each state transition as described in Chapter 6. This successively increases the values of the reward-predicting states, with the increases spreading backwards from the rewarding state, until the values converge to the correct return predictions. In this case (since we are assuming no discounting) the correct predictions are equal to $R^\star$ for all the reward-predicting states. This can be seen in Figure 15.5 as the graph of V labeled 'learning complete' where the values of all the states from the earliest to the latest reward-predicting states all equal $R^\star$. The values of the states preceding the earliest reward-predicting state remain low (which Figure 15.5 shows as zero) because they are not reliable predictors of reward.

When learning is complete, that is, when V attains its correct values, the TD errors associated with transitions *from* any reward-predicting state are zero because the predictions are now accurate. This is because for a transition from a reward-predicting state to another reward-predicting state, we have $\delta_{t-1} = R_t + V_t - V_{t-1} = 0 + R^\star - R^\star = 0$, and for the transition from the latest reward-predicting state to the rewarding state, we have $\delta_{t-1} = R_t + V_t - V_{t-1} = R^\star + 0 - R^\star = 0$. On the other hand, the TD error on a transition from any state *to* the earliest reward-predicting state is positive because of the mismatch between this state's low value and the larger value of the following reward-predicting state. Indeed, if the value of a state preceding the earliest reward-predicting state were zero, then after the transition to the earliest reward-predicting state, we would have that $\delta_{t-1} = R_t + V_t - V_{t-1} = 0 + R^\star - 0 = R^\star$. The 'learning complete' graph of δ in Figure 15.5 shows this positive value at the earliest reward-predicting state, and zeros everywhere else.

The positive TD error upon transitioning to the earliest reward-predicting state is analogous to the persistence of dopamine responses to the earliest stimuli predicting reward. By the same token, when learning is complete, a transition from the latest reward-predicting state to the rewarding state produces a zero TD error because the latest reward-predicting state's value, being correct, cancels the reward. This parallels the observation that fewer dopamine neurons generate a phasic response to a fully predicted reward than to an unpredicted reward.

After learning, if the reward is suddenly omitted, the TD error goes negative at the usual time of reward because the value of the latest reward-predicting state is then too high: $\delta_{t-1} = R_t + V_t - V_{t-1} = 0 + 0 - R^\star = -R^\star$, as shown at the right end of the 'R omitted' graph of δ in Figure 15.5. This is like dopamine neuron activity decreasing below baseline at the time an expected reward is omitted as seen in the experiment of Schultz et al. (1993) described above and shown in Figure 15.4.

The idea of an *earliest reward-predicting state* deserves more attention. In the scenario described above, since experience is divided into trials, and we assumed that predictions are confined to individual trials, the earliest reward-predicting state is always the first state of a trial. Clearly this is artificial. A more general way to think of an earliest reward-predicting state is that it is an *unpredicted predictor* of reward, and there can be many such states. In an animal's life, many different states may precede an earliest reward-predicting state. However, because these states are more often followed by *other* states that do not predict reward, their reward-predicting powers, that is, their values, remain low. A TD algorithm, if operating throughout the animal's life, would update the values of these states too, but the updates would not consistently accumulate because, by assumption, none of these states reliably precedes an earliest reward-predicting state. If any of them did, they would be reward-predicting states as well. This might explain why with overtraining, dopamine responses decrease to even the earliest reward-predicting stimulus in a trial. With overtraining one would expect that even a formerly-unpredicted predictor state would become predicted by stimuli associated with earlier states:

the animal's interaction with its environment both inside and outside of an experimental task would become commonplace. Upon breaking this routine with the introduction of a new task, however, one would see TD errors reappear, as indeed is observed in dopamine neuron activity.

The example described above explains why the TD error shares key features with the phasic activity of dopamine neurons when the animal is learning in a task similar to the idealized task of our example. But not every property of the phasic activity of dopamine neurons coincides so neatly with properties of δ. One of the most troubling discrepancies involves what happens when a reward occurs *earlier* than expected. We have seen that the omission of an expected reward produces a negative prediction error at the reward's expected time, which corresponds to the activity of dopamine neurons decreasing below baseline when this happens. If the reward arrives later than expected, it is then an unexpected reward and generates a positive prediction error. This happens with both TD errors and dopamine neuron responses. But when reward arrives earlier than expected, dopamine neurons do not do what the TD error does—at least with the CSC representation used by Montague et al. (1996) and by us in our example. Dopamine neurons do respond to the early reward, which is consistent with a positive TD error because the reward is not predicted to occur then. However, at the later time when the reward is expected but omitted, the TD error is negative whereas, in contrast to this prediction, dopamine neuron activity does not drop below baseline in the way the TD model predicts (Hollerman and Schultz, 1998). Something more complicated is going on in the animal's brain than simply TD learning with a CSC representation.

Some of the mismatches between the TD error and dopamine neuron activity can be addressed by selecting suitable parameter values for the TD algorithm and by using stimulus representations other than the CSC representation. For instance, to address the early-reward mismatch just described, Suri and Schultz (1999) proposed a CSC representation in which the sequences of internal signals initiated by earlier stimuli are cancelled by the occurrence of a reward. Another proposal by Daw, Courville, and Touretzky (2006) is that the brain's TD system uses representations produced by statistical modeling carried out in sensory cortex rather than simpler representations based on raw sensory input. Ludvig, Sutton, and Kehoe (2008) found that TD learning with a microstimulus (MS) representation (Figure 14.2) fits the activity of dopamine neurons in the early-reward and other situations better than when a CSC representation is used. Pan, Schmidt, Wickens, and Hyland (2005) found that even with the CSC representation, prolonged eligibility traces improve the fit of the TD error to some aspects of dopamine neuron activity. In general, many fine details of TD-error behavior depend on subtle interactions between eligibility traces, discounting, and stimulus representations. Findings like these elaborate and refine the reward prediction error hypothesis without refuting its core claim that the phasic activity of dopamine neurons is well characterized as signaling TD errors.

On the other hand, there are other discrepancies between the TD theory and experimental data that are not so easily accommodated by selecting parameter values and stimulus representations (we mention some of these discrepancies in the Bibliographical and Historical Remarks section at the end of this chapter), and more mismatches are likely to be discovered as neuroscientists conduct ever more refined experiments. But the reward prediction error hypothesis has been functioning very effectively as a catalyst for improving our understanding of how the brain's reward system works. Intricate experiments have been designed to validate or refute predictions derived from the hypothesis, and experimental results have, in turn, led to refinement and elaboration of the TD error/dopamine hypothesis.

A remarkable aspect of these developments is that the reinforcement learning algorithms and theory that connect so well with properties of the dopamine system were developed from a computational perspective in total absence of any knowledge about the relevant properties of dopamine neurons—remember, TD learning and its connections to optimal control and dynamic programming were developed many years before any of the experiments were conducted that revealed the TD-like nature of dopamine neuron activity. This unplanned correspondence, despite not being perfect, suggests that the TD error/dopamine parallel captures something significant about brain reward processes.

In addition to accounting for many features of the phasic activity of dopamine neurons, the reward prediction error hypothesis links neuroscience to other aspects of reinforcement learning, in particular, to learning algorithms that use TD errors as reinforcement signals. Neuroscience is still far from reaching complete understanding of the circuits, molecular mechanisms, and functions of the phasic activity of dopamine neurons, but evidence supporting the reward prediction error hypothesis, along with evidence that phasic dopamine responses are reinforcement signals for learning, suggest that the brain might implement something like an actor–critic algorithm in which TD errors play critical roles. Other reinforcement learning algorithms are plausible candidates too, but actor–critic algorithms fit the anatomy and physiology of the mammalian brain particularly well, as we describe in the following two sections.

15.7 Neural Actor–Critic

Actor–critic algorithms learn both policies and value functions. The 'actor' is the component that learns policies, and the 'critic' is the component that learns about whatever policy is currently being followed by the actor in order to 'criticize' the actor's action choices. The critic uses a TD algorithm to learn the state-value function for the actor's current policy. The value function allows the critic to critique the actor's action choices by sending TD errors, δ, to the actor. A positive δ means that the action was 'good' because it led to a state with a better-than-expected value; a negative δ means that the action was 'bad' because it led to a state with a worse-than-expected value. Based on these critiques, the actor continually updates its policy.

Two distinctive features of actor–critic algorithms are responsible for thinking that the brain might implement an algorithm like this. First, the two components of an actor–critic algorithm—the actor and the critic—suggest that two parts of the striatum—the dorsal and ventral subdivisions (Section 15.4), both critical for reward-based learning—may function respectively something like an actor and a critic. A second property of actor–critic algorithms that suggests a brain implementation is that the TD error has the dual role of being the reinforcement signal for both the actor and the critic, though it has a different influence on learning in each of these components. This fits well with several properties of the neural circuitry: axons of dopamine neurons target both the dorsal and ventral subdivisions of the striatum; dopamine appears to be critical for modulating synaptic plasticity in both structures; and how a neuromodulator such as dopamine acts on a target structure depends on properties of the target structure and not just on properties of the neuromodulator.

Section 13.5 presents actor–critic algorithms as policy gradient methods, but the actor–critic algorithm of Barto, Sutton, and Anderson (1983) was simpler and was presented as an artificial neural network. Here we describe an artificial neural network implementation something like that of Barto et al., and we follow Takahashi, Schoenbaum, and Niv (2008) in giving a schematic proposal for how this artificial neural network might be implemented by real neural networks in the brain. We postpone discussion of the actor and critic learning rules until Section 15.8, where we present them as special cases of the policy-gradient formulation and discuss what they suggest about how dopamine might modulate synaptic plasticity.

Figure 15.6a shows an implementation of an actor–critic algorithm as an artificial neural network with component networks implementing the actor and the critic. The critic consists of a single neuron-like unit, V, whose output activity represents state values, and a component shown as the diamond labeled TD that computes TD errors by combining V's output with reward signals and with previous state values (as suggested by the loop from the TD diamond to itself). The actor network has a single layer of k actor units labeled A_i, $i = 1, \ldots, k$. The output of each actor unit is a component of a k-dimensional action vector. An alternative is that there are k separate actions, one commanded by each actor unit, that compete with one another to be executed, but here we will think of the entire A-vector as an action.

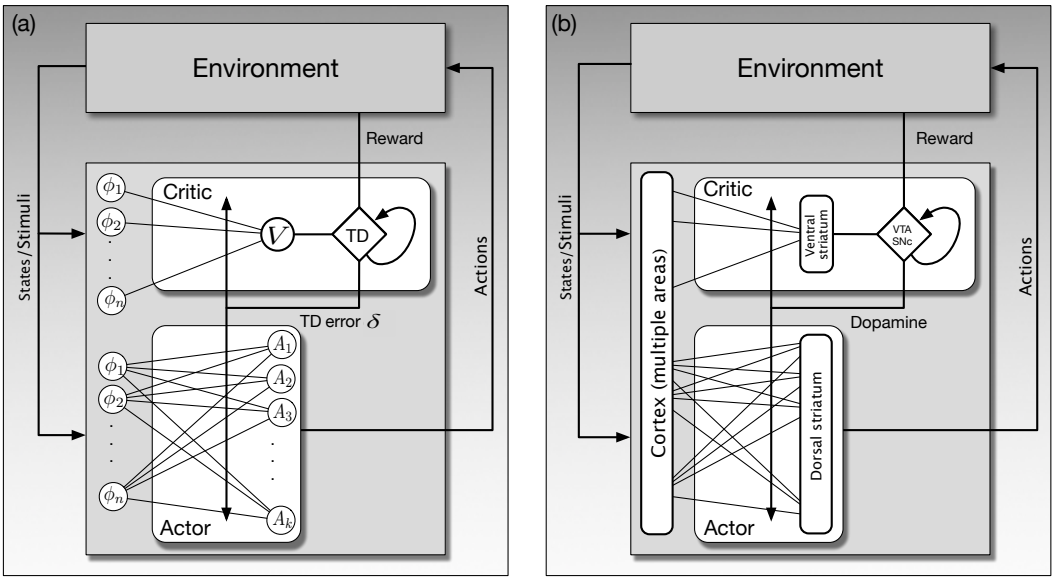


Figure 15.6: Actor–critic artificial neural network and a hypothetical neural implementation. a) Actor–critic algorithm as an artificial neural network. The actor adjusts a policy based on the TD error δ it receives from the critic; the critic adjusts state-value parameters using the same δ. The critic produces a TD error from the reward signal, R, and the current change in its estimate of state values. The actor does not have direct access to the reward signal, and the critic does not have direct access to the action. b) Hypothetical neural implementation of an actor–critic algorithm. The actor and the value-learning part of the critic are respectively placed in the ventral and dorsal subdivisions of the striatum. The TD error is transmitted by dopamine neurons located in the VTA and SNpc to modulate changes in synaptic efficacies of input from cortical areas to the ventral and dorsal striatum. Adapted from Frontiers in Neuroscience, vol. 2(1), 2008, Y. Takahashi, G. Schoenbaum, and Y. Niv, Silencing the critics: Understanding the effects of cocaine sensitization on dorsolateral and ventral striatum in the context of an Actor/Critic model.

Both the critic and actor networks receive input consisting of multiple features representing the state of the agent's environment. (Recall from Chapter 1 that the environment of a reinforcement learning agent includes components both inside and outside of the 'organism' containing the agent.) The figure shows these features as the circles labeled $x_1, x_2, \ldots, x_n$, shown twice just to keep the figure simple. A weight representing the efficacy of a synapse is associated with each connection from each feature x_i to the critic unit, V, and to each of the action units, A_i. The weights in the critic network parameterize the value function, and the weights in the actor network parameterize the policy. The networks learn as these weights change according to the critic and actor learning rules that we describe in the following section.

The TD error produced by circuitry in the critic is the reinforcement signal for changing the weights in both the critic and the actor networks. This is shown in Figure 15.6a by the line labeled 'TD error δ' extending across all of the connections in the critic and actor networks. This aspect of the network implementation, together with the reward prediction error hypothesis and the fact that the activity of dopamine neurons is so widely distributed by the extensive axonal arbors of these neurons, suggests that an actor–critic network something like this may not be too farfetched as a hypothesis about how reward-related learning might happen in the brain.

Figure 15.6b suggests—very schematically—how the artificial neural network on the figure's left might map onto structures in the brain according to the hypothesis of Takahashi et al. (2008). The

hypothesis puts the actor and the value-learning part of the critic respectively in the dorsal and ventral subdivisions of the striatum, the input structure of the basal ganglia. Recall from Section 15.4 that the dorsal striatum is primarily implicated in influencing action selection, and the ventral striatum is thought to be critical for different aspects of reward processing, including the assignment of affective value to sensations. The cerebral cortex, along with other structures, sends input to the striatum conveying information about stimuli, internal states, and motor activity.

In this hypothetical actor–critic brain implementation, the ventral striatum sends value information to the VTA and SNpc, where dopamine neurons in these nuclei combine it with information about reward to generate activity corresponding to TD errors (though exactly how dopaminergic neurons calculate these errors is not yet understood). The 'TD error δ' line in Figure 15.6a becomes the line labeled 'Dopamine' in Figure 15.6b, which represents the widely branching axons of dopamine neurons whose cell bodies are in the VTA and SNpc. Referring back to Figure 15.2, these axons make synaptic contact with the spines on the dendrites of medium spiny neurons, the main input/output neurons of both the dorsal and ventral divisions of the striatum. Axons of the cortical neurons that send input to the striatum make synaptic contact on the tips of these spines. According to the hypothesis, it is at these spines where changes in the efficacies of the synapses from cortical regions to the stratum are governed by learning rules that critically depend on a reinforcement signal supplied by dopamine.

An important implication of the hypothesis illustrated in Figure 15.6b is that the dopamine signal is not the 'master' reward signal like the scalar R_t of reinforcement learning. In fact, the hypothesis implies that one should not necessarily be able to probe the brain and record any signal like R_t in the activity of any single neuron. Many interconnected neural systems generate reward-related information, with different structures being recruited depending on different types of rewards. Dopamine neurons receive information from many different brain areas, so the input to the SNpc and VTA labeled 'Reward' in Figure 15.6b should be thought of as vector of reward-related information arriving to neurons in these nuclei along multiple input channels. What the theoretical scalar reward signal R_t might correspond to, then, is the net contribution of all reward-related information to dopamine neuron activity. It is the result of a pattern of activity across many neurons in different areas of the brain.

Although the actor–critic neural implementation illustrated in Figure 15.6b may be correct on some counts, it clearly needs to be refined, extended, and modified to qualify as a full-fledged model of the function of the phasic activity of dopamine neurons. The Historical and Bibliographic Remarks section at the end of this chapter cites publications that discuss in more detail both empirical support for this hypothesis and places where it falls short. We now look in detail at what the actor and critic learning algorithms suggest about the rules governing changes in synaptic efficacies of corticostriatal synapses.

15.8 Actor and Critic Learning Rules

If the brain does implement something like the actor–critic algorithm—and assuming populations of dopamine neurons broadcast a common reinforcement signal to the corticostriatal synapses of both the dorsal and ventral striatum as illustrated in Figure 15.6b (which is likely an oversimplification as we mentioned above)—then this reinforcement signal affects the synapses of these two structures in different ways. The learning rules for the critic and the actor use the same reinforcement signal, the TD error δ, but its effect on learning is different for these two components. The TD error (combined with eligibility traces) tells the actor how to update action probabilities in order to reach higher-valued states. Learning by the actor is like instrumental conditioning using a Law-of-Effect-type learning rule (Section 1.7): the actor works to keep δ as positive as possible. On the other hand, the TD error (when combined with eligibility traces) tells the critic the direction and magnitude in which to change the parameters of the value function in order to improve its predictive accuracy. The critic works to reduce δ's magnitude to be as close to zero as possible using a learning rule like the TD model of classical conditioning (Section 14.2). The difference between the critic and actor learning rules is relatively

simple, but this difference has a profound effect on learning and is essential to how the actor–critic algorithm works. The difference lies solely in the eligibility traces each type of learning rule uses.

More than one set of learning rules can be used in actor–critic neural networks like those in Figure 15.6b but, to be specific, here we focus on the actor–critic algorithm for continuing problems with eligibility traces presented in Section 13.6. On each transition from state S_t to state S_{t+1}, taking action A_t and receiving action R_{t+1}, that algorithm computes the TD error (δ) and then updates the eligibility trace vectors ($\mathbf{z}_t^{\mathbf{w}}$ and $\mathbf{z}_t^{\boldsymbol{\theta}}$) and the parameters for the critic and actor ($\mathbf{w}$ and $\boldsymbol{\theta}$), according to

$$\begin{aligned}
\delta_t &= R_{t+1} + \gamma \hat{v}(S_{t+1},\mathbf{w}) - \hat{v}(S_t,\mathbf{w}),\\
\mathbf{z}_t^{\mathbf{w}} &= \lambda^{\mathbf{w}} \mathbf{z}_{t-1}^{\mathbf{w}} + \nabla_{\mathbf{w}} \hat{v}(S_t,\mathbf{w}),\\
\mathbf{z}_t^{\boldsymbol{\theta}} &= \lambda^{\boldsymbol{\theta}} \mathbf{z}_{t-1}^{\boldsymbol{\theta}} + \nabla_{\boldsymbol{\theta}} \ln \pi(A_t|S_t,\boldsymbol{\theta}),\\
\mathbf{w} &\leftarrow \mathbf{w} + \alpha^{\mathbf{w}} \delta_t \mathbf{z}_t^{\mathbf{w}},\\
\boldsymbol{\theta} &\leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} \delta \mathbf{z}_t^{\boldsymbol{\theta}},
\end{aligned}$$

where $\gamma \in [0,1)$ is a discount-rate parameter, $\lambda^w c \in [0,1]$ and $\lambda^w a \in [0,1]$ are bootstrapping parameters for the critic and the actor respectively, and $\alpha^{\mathbf{w}} > 0$ and $\alpha^{\boldsymbol{\theta}} > 0$ are analogous step-size parameters.

Think of the approximate value function $\hat{v}$ as the output of a single linear neuron-like unit, called the *critic unit* and labeled V in Figure 15.6a. Then the value function is a linear function of the feature-vector representation of state s, $\mathbf{x}(s) = (x_1(s),\ldots,x_n(s))^\top$, parameterized by a weight vector $\mathbf{w} = (w_1,\ldots,w_n)^\top$:

$$\hat{v}(s,\mathbf{w}) = \mathbf{w}^\top \mathbf{x}(s). \tag{15.1}$$

Each $x_i(s)$ is like the presynaptic signal to a neuron's synapse whose efficacy is w_i. The weights of the critic are incremented according to the rule above by $\alpha^{\mathbf{w}}\delta_t \mathbf{z}_t^{\mathbf{w}}$, where the reinforcement signal, δ_t, corresponds to a dopamine signal being broadcast to all of the critic unit's synapses. The eligibility trace vector, $\mathbf{z}_t^{\mathbf{w}}$, for the critic unit is a trace (average of recent values) of $\nabla_{\mathbf{w}}\hat{v}(S_t,\mathbf{w})$. Because $\hat{v}(s,\mathbf{w})$ is linear in the weights, $\nabla_{\mathbf{w}}\hat{v}(S_t,\mathbf{w}) = \mathbf{x}(S_t)$.

In neural terms, this means that each synapse has its own eligibility trace, which is one component of the vector $\mathbf{z}_t^{\mathbf{w}}$. A synapse's eligibility trace accumulates according to the level of activity arriving at that synapse, that is, the level of presynaptic activity, represented here by the component of the feature vector $\mathbf{x}(S_t)$ arriving at that synapse. The trace otherwise decays toward zero at a rate governed by the fraction $\lambda^{\mathbf{w}}$. A synapse is *eligible for modification* as long as its eligibility trace is non-zero. How the synapse's efficacy is actually modified depends on the reinforcement signals that arrive while the synapse is eligible. We call eligibility traces like these of the critic unit's synapses *non-contingent eligibility traces* because they only depend on presynaptic activity and are not contingent in any way on postsynaptic activity.

The non-contingent eligibility traces of the critic unit's synapses mean that the critic unit's learning rule is essentially the TD model of classical conditioning described in Section 14.2. With the definition we have given above of the critic unit and its learning rule, the critic in Figure 15.6a is the same as the critic in the neural network actor–critic of Barto et al. (1983). Clearly, a critic like this consisting of just one linear neuron-like unit is the simplest starting point; this critic unit is a proxy for a more complicated neural network able to learn value functions of greater complexity.

The actor in Figure 15.6a is a one-layer network of k neuron-like actor units, each receiving at time t the same feature vector, $\mathbf{x}(S_t)$, that the critic unit receives. Each actor unit j, $j = 1,\ldots,k$, has its own weight vector, $\boldsymbol{\theta}_j$, but since the actor units are all identical, we describe just one of the units and omit the subscript. One way for these units to follow the actor–critic algorithm given in the equations above is for each to be a *Bernoulli-logistic unit*. This means that the output of each actor unit at each time is a random variable, A_t, taking value 0 or 1. Think of value 1 as the neuron firing, that is, emitting an action potential. The weighted sum, $\boldsymbol{\theta}^\top \mathbf{x}(S_t)$, of a unit's input vector determines the

unit's action probabilities via the exponential softmax distribution (13.2), which for two actions is the logistic function:

$$\pi(1|s,\boldsymbol{\theta}) = 1 - \pi(0|s,\boldsymbol{\theta}) = \frac{1}{1+\exp(-\boldsymbol{\theta}^\top \mathbf{x}(s))}. \tag{15.2}$$

The weights of each actor unit are incremented, as above, by: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} \delta_t \mathbf{z}_t^{\boldsymbol{\theta}}$, where δ again corresponds to the dopamine signal: the same reinforcement signal that is sent to all the critic unit's synapses. Figure 15.6a shows δ_t being broadcast to all the synapses of all the actor units (which makes this actor network a *team* of reinforcement learning agents, something we discuss in Section 15.10 below). The actor eligibility trace vector $\mathbf{z}_t^{\boldsymbol{\theta}}$ is a trace (average of recent values) of $\nabla_{\boldsymbol{\theta}} \ln \pi(A_t|S_t,\boldsymbol{\theta})$. To understand this eligibility trace refer to Exercise 13.3, which defines this kind of unit and asks you to give a learning rule for it. That exercise asked you to express $\nabla_{\boldsymbol{\theta}} \ln \pi(a|s,\boldsymbol{\theta})$ in terms of a, $\mathbf{x}(s)$, and $\pi(a|s,\boldsymbol{\theta})$ (for arbitrary state s and action a) by calculating the gradient. For the action and state actually occurring at time t, the answer is The answer we were looking for is:

$$\nabla_{\boldsymbol{\theta}} \pi(A_t|S_t,\boldsymbol{\theta}) = \big(A_t - \pi(A_t|S_t,\boldsymbol{\theta})\big)\mathbf{x}(S_t). \tag{15.3}$$

Unlike the non-contingent eligibility trace of a critic synapse that only accumulates the presynaptic activity $\mathbf{x}(S_t)$, the eligibility trace of an actor unit's synapse in addition depends on the activity of the actor unit itself. We call this a *contingent eligibility trace* because it is contingent on this postsynaptic activity. The eligibility trace at each synapse continually decays, but increments or decrements depending on the activity of the presynaptic neuron *and* whether or not the postsynaptic neuron fires. The factor $A_t - \pi(A_t|S_t,\boldsymbol{\theta})$ in (15.3) is positive when $A_t = 1$ and negative otherwise. *The postsynaptic contingency in the eligibility traces of actor units is the only difference between the critic and actor learning rules.* By keeping information about what actions were taken in what states, contingent eligibility traces allow credit for reward (positive δ), or blame for punishment (negative δ), to be apportioned among the policy parameters (the efficacies of the actor units' synapses) according to the contributions these parameters made to the units' outputs that could have influenced later values of δ. Contingent eligibility traces mark the synapses as to how they should be modified to alter the units' future responses to favor positive values of δ.

What do the critic and actor learning rules suggest about how efficacies of corticostriatal synapses change? Both learning rules are related to Donald Hebb's classic proposal that whenever a presynaptic signal participates in activating the postsynaptic neuron, the synapse's efficacy increases (Hebb, 1949). The critic and actor learning rules share with Hebb's proposal the idea that changes in a synapse's efficacy depend on the interaction of several factors. In the critic learning rule the interaction is between the reinforcement signal δ and eligibility traces that depend only on presynaptic signals. Neuroscientists call this a *two-factor learning rule* because the interaction is between two signals or quantities. The actor learning rule, on the other hand, is a *three-factor learning rule* because, in addition to depending on δ, its eligibility traces depend on both presynaptic and postsynaptic activity. Unlike Hebb's proposal, however, the relative timing of the factors is critical to how synaptic efficacies change, with eligibility traces intervening to allow the reinforcement signal to affect synapses that were active in the recent past.

Some subtleties about signal timing for the actor and critic learning rules deserve closer attention. In defining the neuron-like actor and critic units, we ignored the small amount of time it takes synaptic input to effect the firing of a real neuron. When an action potential from the presynaptic neuron arrives at a synapse, neurotransmitter molecules are released that diffuse across the synaptic cleft to the postsynaptic neuron, where they bind to receptors on the postsynaptic neuron's surface; this activates molecular machinery that causes the postsynaptic neuron to fire (or to inhibit its firing in the case of inhibitory synaptic input). This process can take several tens of milliseconds. According to (15.1) and (15.2), though, the input to a critic and actor unit instantaneously produces the unit's

output. Ignoring activation time like this is common in abstract models of Hebbian-style plasticity in which synaptic efficacies change according to a simple product of simultaneous pre- and postsynaptic activity. More realistic models must take activation time into account.

Activation time is especially important for a more realistic actor unit because it influences how contingent eligibility traces have to work in order to properly apportion credit for reinforcement to the appropriate synapses. The expression $\big(A_t - \pi(A_t|S_t, \boldsymbol{\theta})\big)\mathbf{x}(S_t)$ defining contingent eligibility traces for the actor unit's learning rule given above includes the postsynaptic factor $\big(A_t - \pi(A_t|S_t, \boldsymbol{\theta})\big)$ and the presynaptic factor $\mathbf{x}(S_t)$. This works because by ignoring activation time, the presynaptic activity $\mathbf{x}(S_t)$ participates in *causing* the postsynaptic activity appearing in $\big(A_t - \pi(A_t|S_t, \boldsymbol{\theta})\big)$. To assign credit for reinforcement correctly, the presynaptic factor defining the eligibility trace must be a cause of the postsynaptic factor that also defines the trace. Contingent eligibility traces for a more realistic actor unit would have to take activation time into account. (Activation time should not be confused with the time required for a neuron to receive a reinforcement signal influenced by that neuron's activity. The function of eligibility traces is to span this time interval which is generally much longer than the activation time. We discuss this further in the following section.)

There are hints from neuroscience for how this process might work in the brain. Neuroscientists have discovered a form of Hebbian plasticity called *spike-timing-dependent plasticity* (STDP) that lends plausibility to the existence of actor-like synaptic plasticity in the brain. STDP is a Hebbian-style plasticity, but changes in a synapse's efficacy depend on the relative timing of presynaptic and postsynaptic action potentials. The dependence can take different forms, but in the one most studied, a synapse increases in strength if spikes incoming via that synapse arrive shortly before the postsynaptic neuron fires. If the timing relation is reversed, with a presynaptic spike arriving shortly after the postsynaptic neuron fires, then the strength of the synapse decreases. STDP is a type of Hebbian plasticity that takes the activation time of a neuron into account, which is one of the ingredients needed for actor-like learning.

The discovery of STDP has led neuroscientists to investigate the possibility of a three-factor form of STDP in which neuromodulatory input must follow appropriately-timed pre- and postsynaptic spikes. This form of synaptic plasticity, called *reward-modulated STDP*, is much like the actor learning rule discussed here. Synaptic changes that would be produced by regular STDP only occur if there is neuromodulatory input within a time window after a presynaptic spike is closely followed by a postsynaptic spike. Evidence is accumulating that reward-modulated STDP occurs at the spines of medium spiny neurons of the dorsal striatum, with dopamine providing the neuromodulatory factor—the sites where actor learning takes place in the hypothetical neural implementation of an actor–critic algorithm illustrated in Figure 15.6b. Experiments have demonstrated reward-modulated STDP in which lasting changes in the efficacies of corticostriatal synapses occur only if a neuromodulatory pulse arrives within a time window that can last up to 10 seconds after a presynaptic spike is closely followed by a postsynaptic spike (Yagishita et al. 2014). Although the evidence is indirect, these experiments point to the existence of contingent eligibility traces having prolonged time courses. The molecular mechanisms producing these traces, as well as the much shorter traces that likely underly STDP, are not yet understood, but research focusing on time-dependent and neuromodulator-dependent synaptic plasticity is continuing.

The neuron-like actor unit that we have described here, with its Law-of-Effect-style learning rule, appeared in somewhat simpler form in the actor–critic network of Barto et al. (1983). That network was inspired by the "hedonistic neuron" hypothesis proposed by physiologist A. H. Klopf (1972, 1982). Not all the details of Klopf's hypothesis are consistent with what has been learned about synaptic plasticity, but the discovery of STDP and the growing evidence for a reward-modulated form of STDP suggest that Klopf's ideas may not have been far off the mark. We discuss Klopf's hedonistic neuron hypothesis next.

15.9 Hedonistic Neurons

In his hedonistic neuron hypothesis, Klopf (1972, 1982) conjectured that individual neurons seek to maximize the difference between synaptic input treated as rewarding and synaptic input treated as punishing by adjusting the efficacies of their synapses on the basis of rewarding or punishing consequences of their own action potentials. In other words, individual neurons can be trained with response-contingent reinforcement like an animal can be trained in an instrumental conditioning task. His hypothesis included the idea that rewards and punishments are conveyed to a neuron via the same synaptic input that excites or inhibits the neuron's spike-generating activity. (Had Klopf known what we know today about neuromodulatory systems, he might have assigned the reinforcing role to neuromodulatory input, but he wanted to avoid any centralized source of training information.) Synaptically-local traces of past pre- and postsynaptic activity had the key function in Klopf's hypothesis of making synapses *eligible*—the term he introduced—for modification by later reward or punishment. He conjectured that these traces are implemented by molecular mechanisms local to each synapse and therefore different from the electrical activity of both the pre- and the postsynaptic neurons. In the Bibliographical and Historical Remarks section of this chapter we bring attention to some similar proposals made by others.

Klopf specifically conjectured that synaptic efficacies change in the following way. When a neuron fires an action potential, all of its synapses that were active in contributing to that action potential become eligible to undergo changes in their efficacies. If the action potential is followed within an appropriate time period by an increase of reward, the efficacies of all the eligible synapses increase. Symmetrically, if the action potential is followed within an appropriate time period by an increase of punishment, the efficacies of eligible synapses decrease. This is implemented by triggering an eligibility trace at a synapse upon a coincidence of presynaptic and postsynaptic activity (or more exactly, upon pairing of presynaptic activity with the postsynaptic activity that that presynaptic activity participates in causing)—what we call a contingent eligibility trace. This is essentially the three-factor learning rule of an actor unit described in the previous section.

The shape and time course of an eligibility trace in Klopf's theory reflects the durations of the many feedback loops in which the neuron is embedded, some of which lie entirely within the brain and body of the organism, while others extend out through the organism's external environment as mediated by its motor and sensory systems. His idea was that the shape of a synaptic eligibility trace is like a histogram of the durations of the feedback loops in which the neuron is embedded. The peak of an eligibility trace would then occur at the duration of the most prevalent feedback loops in which that neuron participates. The eligibility traces used by algorithms described in this book are simplified versions of Klopf's original idea, being exponentially (or geometrically) decreasing functions controlled by the parameters λ and γ. This simplifies simulations as well as theory, but we regard these simple eligibility traces as a placeholders for traces closer to Klopf's original conception, which would have computational advantages in complex reinforcement learning systems by refining the credit-assignment process.

Klopf's hedonistic neuron hypothesis is not as implausible as it may at first appear. A well-studied example of a single cell that seeks some stimuli and avoids others is the bacterium *Escherichia coli*. The movement of this single-cell organism is influenced by chemical stimuli in its environment, behavior known as chemotaxis. It swims in its liquid environment by rotating hairlike structures called flagella attached to its surface. (Yes, it rotates them!) Molecules in the bacterium's environment bind to receptors on its surface. Binding events modulate the frequency with which the bacterium reverses flagellar rotation. Each reversal causes the bacterium to tumble in place and then head off in a random new direction. A little chemical memory and computation causes the frequency of flagellar reversal to decrease when the bacterium swims toward higher concentrations of molecules it needs to survive (attractants) and increase when the bacterium swims toward higher concentrations of molecules that are harmful (repellants). The result is that the bacterium tends to persist in swimming up attractant gradients and tends to avoid swimming up repellant gradients.

The chemotactic behavior just described is called klinokinesis. It is a kind of trial-and-error behavior, although it is unlikely that learning is involved: the bacterium needs a modicum of short-term memory to detect molecular concentration gradients, but it probably does not maintain long-term memories. Artificial intelligence pioneer Oliver Selfridge called this strategy "run and twiddle," pointing out its utility as a basic adaptive strategy: "keep going in the same way if things are getting better, and otherwise move around" (Selfridge, 1978, 1984). Similarly, one might think of a neuron "swimming" (not literally of course) in a medium composed of the complex collection of feedback loops in which it is embedded, acting to obtain one type of input signal and to avoid others. Unlike the bacterium, however, the neuron's synaptic strengths retain information about its past trial-and-error behavior. If this view of the behavior of a neuron (or just one type of neuron) is plausible, then the closed-loop nature of how the neuron interacts with its environment is important for understanding its behavior, where the neuron's environment consists of the rest of the animal together with the environment with which the animal as a whole interacts.

Klopf's hedonistic neuron hypothesis extended beyond the idea that individual neurons are reinforcement learning agents. He argued that many aspects of intelligent behavior can be understood as the result of the collective behavior of a population of self-interested hedonistic neurons interacting with one another in an immense society or economic system making up an animal's nervous system. Whether or not this view of nervous systems is useful, the collective behavior of reinforcement learning agents has implications for neuroscience. We take up this subject next.

15.10 Collective Reinforcement Learning

The behavior of populations of reinforcement learning agents is deeply relevant to the study of social and economic systems, and if anything like Klopf's hedonistic neuron hypothesis is correct, to neuroscience as well. The hypothesis described above about how an actor–critic algorithm might be implemented in the brain only narrowly addresses the implications of the fact that the dorsal and ventral subdivisions of the striatum, the respective locations of the actor and the critic according to the hypothesis, each contain millions of medium spiny neurons whose synapses undergo change modulated by phasic bursts of dopamine neuron activity.

The actor in Figure 15.6a is a single-layer network of k actor units. The actions produced by this network are vectors $(A_1, A_2, \cdots, A_k)^\top$ presumed to drive the animal's behavior. Changes in the efficacies of the synapses of all of these units depend on the reinforcement signal δ. Because actor units attempt to make δ as large as possible, δ effectively acts as a reward signal for them (so in this case reinforcement is the same as reward). Thus, each actor unit is itself a reinforcement learning agent—a hedonistic neuron if you will. Now, to make the situation as simple as possible, assume that each of these units receives the same reward signal at the same time (although, as indicated above, the assumption that dopamine is released at all the corticostriatal synapses under the same conditions and at the same times is likely an oversimplification).

What can reinforcement learning theory tell us about what happens when all members of a population of reinforcement learning agents learn according to a common reward signal? The field of *multi-agent reinforcement learning* considers many aspects of learning by populations of reinforcement learning agents. Although this field is beyond the scope of this book, we believe that some of its basic concepts and results are relevant to thinking about the brain's diffuse neuromodulatory systems. In multi-agent reinforcement learning (and in game theory), the scenario in which all the agents try to maximize a common reward signal that they simultaneously receive is known as a *cooperative game* or a *team problem*.

What makes a team problem interesting and challenging is that the common reward signal sent to each agent evaluates the *pattern* of activity produced by the entire population, that is, it evaluates the *collective action* of the team members. This means that any individual agent has only limited ability

to affect the reward signal because any single agent contributes just one component of the collective action evaluated by the common reward signal. Effective learning in this scenario requires addressing a *structural credit assignment problem*: which team members, or groups of team members, deserve credit for a favorable reward signal, or blame for an unfavorable reward signal? It is a *cooperative* game, or a team problem, because the agents are united in seeking to increase the same reward signal: there are no conflicts of interest among the agents. The scenario would be a *competitive game* if different agents receive different reward signals, where each reward signal again evaluates the collective action of the population, and the objective of each agent is to increase its own reward signal. In this case there might be conflicts of interest among the agents, meaning that actions that are good for some agents are bad for others. Even deciding what the best collective action should be is a non-trivial aspect of game theory. This competitive setting might be relevant to neuroscience too (for example, to account for heterogeneity of dopamine neuron activity), but here we focus only on the cooperative, or team, case.

How can each reinforcement learning agent in a team learn to "do the right thing" so that the collective action of the team is highly rewarded? An interesting result is that if each agent can learn effectively despite its reward signal being corrupted by a large amount of noise, and despite its lack of access to complete state information, then the population as a whole will learn to produce collective actions that improve as evaluated by the common reward signal, even when the agents cannot communicate with one another. Each agent faces its own reinforcement learning task in which its influence on the reward signal is deeply buried in the noise created by the influences of other agents. In fact, for any agent, all the other agents are part of its environment because its input, both the part conveying state information and the reward part, depends on how all the other agents are behaving. Furthermore, lacking access to the actions of the other agents, indeed lacking access to the parameters determining their policies, each agent can only partially observe the state of its environment. This makes each team member's learning task very difficult, but if each uses a reinforcement learning algorithm able to increase a reward signal even under these difficult conditions, teams of reinforcement learning agents can learn to produce collective actions that improve over time as evaluated by the team's common reward signal.

If the team members are neuron-like units, then each unit has to have the goal of increasing the amount of reward it receives over time, as the actor unit does that we described in Section 15.8. Each unit's learning algorithm has to have two essential features. First, it has to use contingent eligibility traces. Recall that a contingent eligibility trace, in neural terms, is initiated (or increased) at a synapse when its presynaptic input participates in causing the postsynaptic neuron to fire. A non-contingent eligibility trace, in contrast, is initiated or increased by presynaptic input independently of what the postsynaptic neuron does. As explained in Section 15.8, by keeping information about what actions were taken in what states, contingent eligibility traces allow credit for reward, or blame for punishment, to be apportioned to an agent's policy parameters according to the contribution the values of these parameters made in determining the agent's action. By similar reasoning, a team member must remember its recent action so that it can either increase or decrease the likelihood of producing that action according to the reward signal that is subsequently received. The action component of a contingent eligibility trace implements this action memory. Because of the complexity of the learning task, however, contingent eligibility is merely a preliminary step in the credit assignment process: the relationship between a single team member's action and changes in the team's reward signal is a statistical correlation that has to be estimated over many trials. Contingent eligibility is an essential but preliminary step in this process.

Learning with non-contingent eligibility traces does not work at all in the team setting because it does not provide a way to correlate actions with consequent changes in the reward signal. Non-contingent eligibility traces are adequate for learning to predict, as the critic component of the actor–critic algorithm does, but they do not support learning to control, as the actor component must do. The members of a population of critic-like agents may still receive a common reinforcement signal, but they would all learn to predict the same quantity (which in the case of an actor–critic method, would be the expected return for the current policy). How successful each member of the population would

be in learning to predict the expected return would depend on the information it receives, which could be very different for different members of the population. There would be no need for the population to produce differentiated patterns of activity. This is not a team problem as defined here.

A second requirement for collective learning in a team problem is that there has to be variability in the actions of the team members in order for the team to explore the space of collective actions. The simplest way for a team of reinforcement learning agents to do this is for each member to independently explore its own action space through persistent variability in its output. This will cause the team as a whole to vary its collective actions. For example, a team of the actor units described in Section 15.8 explores the space of collective actions because the output of each unit, being a Bernoulli-logistic unit, probabilistically depends on the weighted sum of its input vector's components. The weighted sum biases firing probability up or down, but there is always variability. Because each unit uses a REINFORCE policy gradient algorithm (Chapter 13), each unit adjusts its weights with the goal of maximizing the average reward rate it experiences while stochastically exploring its own action space. One can show, as Williams (1992) did, that a team of Bernoulli-logistic REINFORCE units implements a policy gradient algorithm *as a whole* with respect to average rate of the team's common reward signal, where the actions are the collective actions of the team.

Further, Williams (1992) showed that a team of Bernoulli-logistic units using REINFORCE ascends the average reward gradient when the units in the team are interconnected to form a multilayer neural network. In this case, the reward signal is broadcast to all the units in the network, though reward may depend only on the collective actions of the network's output units. This means that a multilayer team of Bernoulli-logistic REINFORCE units learns like a multilayer network trained by the widely-used error backpropagation method, but in this case the backpropagation process is replaced by the broadcasted reward signal. In practice, the error backpropagation method is considerably faster, but the reinforcement learning team method is more plausible as a neural mechanism, especially in light of what is being learned about reward-modulated STDP as discussed in Section 15.8.

Exploration through independent exploration by team members is only the simplest way for a team to explore; more sophisticated methods are possible if the team members communicate with one another so that they can coordinate their actions to focus on particular parts of the collective action space. There are also mechanisms more sophisticated than contingent eligibility traces for addressing structural credit assignment, which is easier in a team problem when the set of possible collective actions is restricted in some way. An extreme case is a winner-take-all arrangement (for example, the result of lateral inhibition in the brain) that restricts collective actions to those to which only one, or a few, team members contribute. In this case the winners get the credit or blame for resulting reward or punishment.

Details of learning in cooperative games (or team problems) and non-cooperative game problems are beyond the scope of this book. The Bibliographical and Historical Remarks section at the end of this chapter cites a selection of the relevant publications, including extensive references to research on implications for neuroscience of collective reinforcement learning.

15.11 Model-based Methods in the Brain

Reinforcement learning's distinction between model-free and model-based algorithms is proving to be useful for thinking about animal learning and decision processes. Section 14.6 discusses how this distinction aligns with that between habitual and goal-directed animal behavior. The hypothesis discussed above about how the brain might implement an actor–critic algorithm is relevant only to an animal's habitual mode of behavior because the basic actor–critic method is model-free. What neural mechanisms are responsible for producing goal-directed behavior, and how do they interact with those underlying habitual behavior?

One way to investigate questions about the brain structures involved in these modes of behavior is

to inactivate an area of a rat's brain and then observe what the rat does in an outcome-devaluation experiment (Section 14.6). Results from experiments like these indicate that the actor–critic hypothesis described above is too simple in placing the actor in the dorsal striatum. Inactivating one part of the dorsal striatum, the dorsolateral striatum (DLS), impairs habit learning, causing the animal to rely more on goal-directed processes. On the other hand, inactivating the dorsomedial striatum (DMS) impairs goal-directed processes, requiring the animal to rely more on habit learning. Results like these support the view that the DLS in rodents is more involved in model-free processes, whereas their DMS is more involved in model-based processes. Results of studies with human subjects in similar experiments using functional neuroimaging, and with non-human primates, support the view that the analogous structures in the primate brain are differentially involved in habitual and goal-directed modes of behavior.

Other studies identify activity associated with model-based processes in the prefrontal cortex of the human brain, the front-most part of the frontal cortex implicated in executive function, including planning and decision making. Specifically implicated is the orbitofrontal cortex (OFC), the part of the prefrontal cortex immediately above the eyes. Functional neuroimaging in humans, and also recordings of the activities of single neurons in monkeys, reveals strong activity in the OFC related to the subjective reward value of biologically significant stimuli, as well as activity related to the reward expected as a consequence of actions. Although not free of controversy, these results suggest significant involvement of the OFC in goal-directed choice. It may be critical for the reward part of an animal's environment model.

Another structure involved in model-based behavior is the hippocampus, a structure critical for memory and spatial navigation. A rat's hippocampus plays a critical role in the rat's ability to navigate a maze in the goal-directed manner that led Tolman to the idea that animals use models, or cognitive maps, in selecting actions (Section 14.5). The hippocampus may also be a critical component of our human ability to imagine new experiences (Hassabis and Maguire, 2007; Ólafsdóttir, Barry, Saleem, Hassabis, and Spiers, 2105).

The findings that most directly implicate the hippocampus in planning—the process needed to enlist an environment model in making decisions—come from experiments that decode the activity of neurons in the hippocampus to determine what part of space hippocampal activity is representing on a moment-to-moment basis. When a rat pauses at a choice point in a maze, the representation of space in the hippocampus sweeps forward (and not backwards) along the possible paths the animal can take from that point (Johnson and Redish, 2007). Furthermore, the spatial trajectories represented by these sweeps closely correspond to the rat's subsequent navigational behavior (Pfeiffer and Foster, 2013). These results suggest that the hippocampus is critical for the state-transition part of an animal's environment model, and that it is part of a system that uses the model to simulate possible future state sequences to assess the consequences of possible courses of action: a form of planning.

The results described above add to a voluminous literature on neural mechanisms underlying goal-directed, or model-based, learning and decision making, but many questions remain unanswered. For example, how can areas as structurally similar as the DLS and DMS be essential components of modes of learning and behavior that are as different as model-free and model-based algorithms? Are separate structures responsible for (what we call) the transition and reward components of an environment model? Is all planning conducted at decision time via simulations of possible future courses of action as the forward sweeping activity in the hippocampus suggests? In other words, is all planning something like a rollout algorithm (Section 8.10)? Or are models sometimes engaged in the background to refine or recompute value information as illustrated by the Dyna architecture (Section 8.2)? How does the brain arbitrate between the use of the habit and goal-directed systems? Is there, in fact, a clear separation between the neural substrates of these systems?

The evidence is not pointing to a positive answer to this last question. Summarizing the situation, Doll, Simon, and Daw (2012) wrote that "model-based influences appear ubiquitous more or less wherever the brain processes reward information," and this is true even in the regions thought to be critical

for model-free learning. This includes the dopamine signals themselves, which can exhibit the influence of model-based information in addition to the reward prediction errors thought to be the basis of model-free processes.

Continuing neuroscience research informed by reinforcement learning's model-free and model-based distinction has the potential to sharpen our understanding of habitual and goal-directed processes in the brain. A better grasp of these neural mechanisms may lead to algorithms combining model-free and model-based methods in ways that have not yet been explored in computational reinforcement learning.

15.12 Addiction

Understanding the neural basis of drug abuse is a high-priority goal of neuroscience with the potential to produce new treatments for this serious public health problem. One view is that drug craving is the result of the same motivation and learning processes that lead us to seek natural rewarding experiences that serve our biological needs. Addictive substances, by being intensely reinforcing, effectively co-opt our natural mechanisms of learning and decision making. This is plausible given that many—though not all—drugs of abuse increase levels of dopamine either directly or indirectly in regions around terminals of dopamine neuron axons in the striatum, a brain structure firmly implicated in normal reward-based learning (Section 15.7). But the self-destructive behavior associated with drug addiction is not characteristic of normal learning. What is different about dopamine-mediated learning when the reward is the result of an addictive drug? Is addiction the result of normal learning in response to substances that were largely unavailable throughout our evolutionary history, so that evolution could not select against their damaging effects? Or do addictive substances somehow interfere with normal dopamine-mediated learning?

The reward prediction error hypothesis of dopamine neuron activity and its connection to TD learning are the basis of a model due to Redish (2004) of some—but certainly not all—features of addiction. The model is based on the observation that administration of cocaine and some other addictive drugs produces a transient increase in dopamine. In the model, this dopamine surge is assumed to increase the TD error, δ, in a way that cannot be cancelled out by changes in the value function. In other words, whereas δ is reduced to the degree that a normal reward is predicted by antecedent events (Section 15.6), the contribution to δ due to an addictive stimulus does not decrease as the reward signal becomes predicted: drug rewards cannot be "predicted away." The model does this by preventing δ from ever becoming negative when the reward signal is due to an addictive drug, thus eliminating the error-correcting feature of TD learning for states associated with administration of the drug. The result is that the values of these states increase without bound, making actions leading to these states preferred above all others.

Addictive behavior is much more complicated than this result from Redish's model, but the model's main idea may be a piece of the puzzle. Or the model might be misleading. Dopamine appears not to play a critical role in all forms of addiction, and not everyone is equally susceptible to developing addictive behavior. Moreover, the model does not include the changes in many circuits and brain regions that accompany chronic drug taking, for example, changes that lead to a drug's diminishing effect with repeated use. It is also likely that addiction involves model-based processes. Still, Redish's model illustrates how reinforcement learning theory can be enlisted in the effort to understand a major health problem. In a similar manner, reinforcement learning theory has been influential in the development of the new field of computational psychiatry, which aims to improve understanding of mental disorders through mathematical and computational methods.

15.13 Summary

The neural pathways involved in the brain's reward system are complex and incompletely understood, but neuroscience research directed toward understanding these pathways and their roles in behavior is progressing rapidly. This research is revealing striking correspondences between the brain's reward system and the theory of reinforcement learning as presented in this book.

The *reward prediction error hypothesis of dopamine neuron activity* was proposed by scientists who recognized striking parallels between the behavior of TD errors and the activity of neurons that produce dopamine, a neurotransmitter essential in mammals for reward-related learning and behavior. Experiments conducted in the late 1980s and 1990s in the laboratory of neuroscientist Wolfram Schultz showed that dopamine neurons respond to rewarding events with substantial bursts of activity, called phasic responses, only if the animal does not expect those events, suggesting that dopamine neurons are signaling reward prediction errors instead of reward itself. Further, these experiments showed that as an animal learns to predict a rewarding event on the basis of preceding sensory cues, the phasic activity of dopamine neurons shifts to earlier predictive cues while decreasing to later predictive cues. This parallels the backing-up effect of the TD error as a reinforcement learning agent learns to predict reward.

Other experimental results firmly establish that the phasic activity of dopamine neurons is a reinforcement signal for learning that reaches multiple areas of the brain by means of profusely branching axons of dopamine producing neurons. These results are consistent with the distinction we make between a reward signal, R_t, and a reinforcement signal, which is the TD error δ_t in most of the algorithms we present. Phasic responses of dopamine neurons are reinforcement signals, not reward signals.

A prominent hypothesis is that the brain implements something like an actor–critic algorithm. Two structures in the brain (the dorsal and ventral subdivisions of the striatum), both of which play critical roles in reward-based learning, may function respectively like an actor and a critic. That the TD error is the reinforcement signal for both the actor and the critic fits well with the facts that dopamine neuron axons target both the dorsal and ventral subdivisions of the striatum; that dopamine appears to be critical for modulating synaptic plasticity in both structures; and that the effect on a target structure of a neuromodulator such as dopamine depends on properties of the target structure and not just on properties of the neuromodulator.

The actor and the critic can be implemented by artificial neural networks consisting of neuron-like units having learning rules based on the policy-gradient actor–critic method described in Section 13.5. Each connection in these networks is like a synapse between neurons in the brain, and the learning rules correspond to rules governing how synaptic efficacies change as functions of the activities of the presynaptic and the postsynaptic neurons, together with neuromodulatory input corresponding to input from dopamine neurons. In this setting, each synapse has its own eligibility trace that records past activity involving that synapse. The only difference between the actor and critic learning rules is that they use different kinds of eligibility traces: the critic unit's traces are *non-contingent* because they do not involve the critic unit's output, whereas the actor unit's traces are *contingent* because in addition to the actor unit's input, they depend on the actor unit's output. In the hypothetical implementation of an actor–critic system in the brain, these learning rules respectively correspond to rules governing plasticity of corticostriatal synapses that convey signals from the cortex to the principal neurons in the dorsal and ventral striatal subdivisions, synapses that also receive inputs from dopamine neurons.

The learning rule of an actor unit in the actor–critic network closely corresponds to *reward-modulated spike-timing-dependent plasticity*. In spike-timing-dependent plasticity (STDP), the relative timing of pre- and postsynaptic activity determines the direction of synaptic change. In reward-modulated STDP, changes in synapses in addition depend on a neuromodulator, such as dopamine, arriving within a time window that can last up to 10 seconds after the conditions for STDP are met. Evidence accumulating that reward-modulated STDP occurs at corticostriatal synapses, where the actor's learning takes place

in the hypothetical neural implementation of an actor–critic system, adds to the plausibility of the hypothesis that something like an actor–critic system exists in the brains of some animals.

The idea of synaptic eligibility and basic features of the actor learning rule derive from Klopf's hypothesis of the "hedonistic neuron" (Klopf, 1972, 1981). He conjectured that individual neurons seek to obtain reward and to avoid punishment by adjusting the efficacies of their synapses on the basis of rewarding or punishing consequences of their action potentials. A neuron's activity can affect its later input because the neuron is embedded in many feedback loops, some within the animal's nervous system and body and others passing through the animal's external environment. Klopf's idea of eligibility is that synapses are temporarily marked as eligible for modification if they participated in the neuron's firing (making this the contingent form of eligibility trace). A synapse's efficacy is modified if a reinforcing signal arrives while the synapse is eligible. We alluded to the chemotactic behavior of a bacterium as an example of a single cell that directs its movements in order to seek some molecules and to avoid others.

A conspicuous feature of the dopamine system is that fibers releasing dopamine project widely to multiple parts of the brain. Although it is likely that only some populations of dopamine neurons broadcast the same reinforcement signal, if this signal reaches the synapses of many neurons involved in actor-type learning, then the situation can be modeled as a *team problem*. In this type of problem, each agent in a collection of reinforcement learning agents receives the same reinforcement signal, where that signal depends on the activities of all members of the collection, or team. If each team member uses a sufficiently capable learning algorithm, the team can learn collectively to improve performance of the entire team as evaluated by the globally-broadcast reinforcement signal, even if the team members do not directly communicate with one another. This is consistent with the wide dispersion of dopamine signals in the brain and provides a neurally plausible alternative to the widely-used error-backpropagation method for training multilayer networks.

The distinction between model-free and model-based reinforcement learning is helping neuroscientists investigate the neural bases of habitual and goal-directed learning and decision making. Research so far points to their being some brain regions more involved in one type of process than the other, but the picture remains unclear because model-free and model-based processes do not appear to be neatly separated in the brain. Many questions remain unanswered. Perhaps most intriguing is evidence that the hippocampus, a structure traditionally associated with spatial navigation and memory, appears to be involved in simulating possible future courses of action as part of an animal's decision-making process. This suggests that it is part of a system that uses an environment model for planning.

Reinforcement learning theory is also influencing thinking about neural processes underlying drug abuse. A model of some features of drug addiction is based on the reward prediction error hypothesis. It proposes that an addicting stimulant, such as cocaine, destabilizes TD learning to produce unbounded growth in the values of actions associated with drug intake. This is far from a complete model of addiction, but it illustrates how a computational perspective suggests theories that can be tested with further research. The new field of computational psychiatry similarly focuses on the use of computational models, some derived from reinforcement learning, to better understand mental disorders.

This chapter only touched the surface of how the neuroscience of reinforcement learning and the development of reinforcement learning in computer science and engineering have influenced one another. Most features of reinforcement learning algorithms owe their design to purely computational considerations, but some have been influenced by hypotheses about neural learning mechanisms. Remarkably, as experimental data has accumulated about the brain's reward processes, many of the purely computationally-motivated features of reinforcement learning algorithms are turning out to be consistent with neuroscience data. Other features of computational reinforcement learning, such eligibility traces and the ability of teams of reinforcement learning agents to learn to act collectively under the influence of a globally-broadcast reinforcement signal, may also turn out to parallel experimental data as neuroscientists continue to unravel the neural basis of reward-based animal learning and behavior.

Bibliographical and Historical Remarks

The number of publications treating parallels between the neuroscience of learning and decision making and the approach to reinforcement learning presented in this book is enormous. We can cite only a small selection. Niv (2009), Dayan and Niv (2008), Gimcher (2011), Ludvig, Bellemare, and Pearson (2011), and Shah (2012) are good places to start.

Together with economics, evolutionary biology, and mathematical psychology, reinforcement learning theory is helping to formulate quantitative models of the neural mechanisms of choice in humans and non-human primates. With its focus on learning, this chapter only lightly touches upon the neuroscience of decision making. Glimcher (2003) introduced the field of "neuroeconomics," in which reinforcement learning contributes to the study of the neural basis of decision making from an economics perspective. See also Glimcher and Fehr (2013). The text on computational and mathematical modeling in neuroscience by Dayan and Abbott (2001) includes reinforcement learning's role in these approaches. Sterling and Laughlin (2015) examined the neural basis of learning in terms of general design principles that enable efficient adaptive behavior.

15.1 There are many good expositions of basic neuroscience. Kandel, Schwartz, Jessell, Siegelbaum, and Hudspeth (2013) is an authoritative and very comprehensive source.

15.2 Berridge and Kringelbach (2008) reviewed the neural basis of reward and pleasure, pointing out that reward processing has many dimensions and involves many neural systems. Space prevents discussion of the influential research of Berridge and Robinson (1998), who distinguish between the hedonic impact of a stimulus, which they call "liking," and the motivational effect, which they call "wanting." Hare, O'Doherty, Camerer, Schultz, and Rangel (2008) examined the neural basis of value-related signals from an economic perspective, distinguishing between goal values, decision values, and prediction errors. Decision value is goal value minus action cost. See also Rangel, Camerer, and Montague (2008), Rangel and Hare (2010), and Peters and Büchel (2010).

15.3 The reward prediction error hypothesis of dopamine neuron activity is most prominently discussed by Schultz, Montague, and Dayan (1997). The hypothesis was first explicitly put forward by Montague, Dayan, and Sejnowski (1996). As they stated the hypothesis, it referred to reward prediction errors (RPEs) but not specifically to TD errors; however, their development of the hypothesis made it clear that they were referring to TD errors. The earliest recognition of the TD-error/dopamine connection of which we are aware is that of Montague, Dayan, Nowlan, Pouget, and Sejnowski (1992), who proposed a TD-error-modulated Hebbian learning rule motivated by results on dopamine signaling from Schultz's group. The connection was also pointed out in an abstract by Quartz, Dayan, Montague, and Sejnowski (1992). Montague and Sejnowski (1994) emphasized the importance of prediction in the brain and outlined how predictive Hebbian learning modulated by TD errors could be implemented via a diffuse neuromodulatory system, such as the dopamine system. Friston, Tononi, Reeke, Sporns, and Edelman (1994) presented a model of value-dependent learning in the brain in which synaptic changes are mediated by a TD-like error provided by a global neuromodulatory signal (although they did not single out dopamine). Montague, Dayan, Person, and Sejnowski (1995) presented a model of honeybee foraging using the TD error. The model is based on research by Hammer, Menzel, and colleagues (Hammer and Menzel, 1995; Hammer, 1997) showing that the neuromodulator octopamine acts as a reinforcement signal in the honeybee. Montague et al. (1995) pointed out that dopamine likely plays a similar role in the vertebrate brain. Barto (1995) related the actor–critic architecture to basal-ganglionic circuits and discussed the relationship between TD learning and the main results from Schultz's group. Houk, Adams, and Barto (1995) suggested how TD learning and the actor–critic architecture might map onto

the anatomy, physiology, and molecular mechanism of the basal ganglia. Doya and Sejnowski (1998) extended their earlier paper on a model of birdsong learning (Doya and Sejnowski, 1994) by including a TD-like error identified with dopamine to reinforce the selection of auditory input to be memorized. O'Reilly and Frank (2006) and O'Reilly, Frank, Hazy, and Watz (2007) argued that phasic dopamine signals are RPEs but not TD errors. In support of their theory they cited results with variable interstimulus intervals that do not match predictions of a simple TD model, as well as the observation that higher-order conditioning beyond second-order conditioning is rarely observed, while TD learning is not so limited. Dayan and Niv (2008) discussed "the good, the bad, and the ugly" of how reinforcement learning theory and the reward prediction error hypothesis align with experimental data. Glimcher (2011) reviewed the empirical findings that support the reward prediction error hypothesis and emphasized the significance of the hypothesis for contemporary neuroscience.

15.4 Graybiel (2000) is a brief primer on the basal ganglia. The experiments mentioned that involve optogenetic activation of dopamine neurons were conducted by Tsai, Zhang, Adamantidis, Stuber, Bonci, de Lecea, and Deisseroth (2009), Steinberg, Keiflin, Boivin, Witten, Deisseroth, and Janak (2013), and Claridge-Chang, Roorda, Vrontou, Sjulson, Li, Hirsh, and Miesenböck (2009). Fiorillo, Yun, and Song (2013), Lammel, Lim, and Malenka (2014), and Saddoris, Cacciapaglia, Wightmman, and Carelli (2015) are among studies showing that the signaling properties of dopamine neurons are specialized for different target regions. RPE-signaling neurons may belong to one among multiple populations of dopamine neurons having different targets and subserving different functions. Eshel, Tian, Bukwich, and Uchida (2016) found homogeneity of reward prediction error responses of dopamine neurons in the lateral VTA during classical conditioning in mice, though their results do not rule out response diversity across wider areas. Gershman, Pesaran, and Daw (2009) studied reinforcement learning tasks that can be decomposed into independent components with separate reward signals, finding evidence in human neuroimaging data suggesting that the brain exploits this kind of structure.

15.5 Schultz's 1998 survey article (Schultz, 1998) is a good entrée into the very extensive literature on reward predicting signaling of dopamine neurons. Berns, McClure, Pagnoni, and Montague (2001), Breiter, Aharon, Kahneman, Dale, and Shizgal (2001), Pagnoni, Zink, Montague, and Berns (2002), and O'Doherty, Dayan, Friston, Critchley, and Dolan (2003) described functional brain imaging studies supporting the existence of signals like TD errors in the human brain.

15.6 This section roughly follows Barto (1995) in explaining how TD errors mimic the main results from Schultz's group on the phasic responses of dopamine neurons.

15.7 This section is largely based on Takahashi, Schoenbaum, and Niv (2008) and Niv (2009). To the best of our knowledge, Barto (1995) and Houk, Adams, and Barto (1995) first speculated about possible implementations of actor–critic algorithms in the basal ganglia. On the basis of functional magnetic resonance imaging of human subjects while engaged in instrumental conditioning, O'Doherty, Dayan, Schultz, Deichmann, Friston, and Dolan (2004) suggested that the actor and the critic are most likely located respectively in the dorsal and ventral striatum. Gershman, Moustafa, and Ludvig (2013) focused on how time is represented in reinforcement learning models of the basal ganglia, discussing evidence for, and implications of, various computational approaches to time representation.

The hypothetical neural implementation of the actor–critic architecture described in this section includes very little detail about known basal ganglia anatomy and physiology. In addition to the more detailed hypothesis of Houk, Adams, and Barto (1995), a number of other hypotheses include more specific connections to anatomy and physiology and are claimed to explain additional data. These include hypotheses proposed by Suri and Schultz (1998, 1999), Brown,

Bullock, and Grossberg (1999), Contreras-Vidal and Schultz (1999), Suri, Bargas, and Arbib (2001), O'Reilly and Frank (2006), and O'Reilly, Frank, Hazy, and Watz (2007). Joel, Niv, and Ruppin (2002) critically evaluated the anatomical plausibility of several of these models and present an alternative intended to accommodate some neglected features of basal ganglionic circuitry.

15.8 The actor learning rule discussed here is more complicated than the one in the early actor–critic network of Barto et al. (1983). Actor-unit eligibility traces in that network were traces of just $A_t \times \mathbf{x}(S_t)$ instead of the full $(A_t - \pi(A_t|S_t, \boldsymbol{\theta}))\mathbf{x}(S_t)$. That work did not benefit from the policy-gradient theory presented in Chapter 13 or the contributions of Williams (1986, 1992), who showed how an artificial neural network of Bernoulli-logistic units could implement a policy-gradient method.

Reynolds and Wickens (2002) proposed a three-factor rule for synaptic plasticity in the corticostriatal pathway in which dopamine modulates changes in corticostriatal synaptic efficacy. They discussed the experimental support for this kind of learning rule and its possible molecular basis. The definitive demonstration of spike-timing-dependent plasticity (STDP) is attributed to Markram, Lübke, Frotscher, and Sakmann (1997), with evidence from earlier experiments by Levy and Steward (1983) and others that the relative timing of pre- and postsynaptic spikes is critical for inducing changes in synaptic efficacy. Rao and Sejnowski (2001) suggested how STDP could be the result of a TD-like mechanism at synapses with non-contingent eligibility traces lasting about 10 milliseconds. Dayan (2002) commented that this would require an error as in Sutton and Barto's (1981) early model of classical conditioning and not a true TD error. Representative publications from the extensive literature on reward-modulated STDP are Wickens (1990), Reynolds and Wickens (2002), and Calabresi, Picconi, Tozzi and Di Filippo (2007). Pawlak and Kerr (2008) showed that dopamine is necessary to induce STDP at the corticostriatal synapses of medium spiny neurons. See also Pawlak, Wickens, Kirkwood, and Kerr (2010). Yagishita, Hayashi-Takagi, Ellis-Davies, Urakubo, Ishii, and Kasai (2014) found that dopamine promotes spine enlargement of the medium spiny neurons of mice only during a time window of from 0.3 to 2 seconds after STDP stimulation. Izhikevich (2007) proposed and explored the idea of using STDP timing conditions to trigger contingent eligibility traces.

15.9 Klopf's hedonistic neuron hypothesis (Klopf 1972, 1982) inspired our actor–critic algorithm implemented as an artificial neural network with a single neuron-like unit, called the actor unit, implementing a Law-of-Effect-like learning rule (Barto, Sutton, and Anderson, 1983). Ideas related to Klopf's synaptically-local eligibility have been proposed by others. Crow (1968) proposed that changes in the synapses of cortical neurons are sensitive to the consequences of neural activity. Emphasizing the need to address the time delay between neural activity and its consequences in a reward-modulated form of synaptic plasticity, he proposed a contingent form of eligibility, but associated with entire neurons instead of individual synapses. According to his hypothesis, a wave of neuronal activity

> leads to a short-term change in the cells involved in the wave such that they are picked out from a background of cells not so activated. ... such cells are rendered sensitive by the short-term change to a reward signal ... in such a way that if such a signal occurs before the end of the decay time of the change the synaptic connexions between the cells are made more effective. (Crow, 1968)

Crow argued against previous proposals that reverberating neural circuits play this role by pointing out that the effect of a reward signal on such a circuit would "...establish the synaptic connexions leading to the reverberation (that is to say, those involved in activity at the time of the reward signal) and not those on the path which led to the adaptive motor output."

Crow further postulated that reward signals are delivered via a "distinct neural fiber system," presumably the one into which Olds and Milner (1954) tapped, that would transform synaptic connections "from a short into a long-term form."

In another farsighted hypothesis, Miller (1981) proposed a Law-of-Effect-like learning rule that includes synaptically-local contingent eligibility traces:

> ... it is envisaged that in a particular sensory situation neurone B, by chance, fires a 'meaningful burst' of activity, which is then translated into motor acts, which then change the situation. It must be supposed that the meaningful burst has an influence, *at the neuronal level*, on all of its own synapses which are active at the time ... thereby making a preliminary selection of the synapses to be strengthened, though not yet actually strengthening them. ...The strengthening signal ... makes the final selection ... and accomplishes the definitive change in the appropriate synapses. (Miller, 1981, p. 81)

Miller's hypothesis also included a critic-like mechanism, which he called a "sensory analyzer unit," that worked according to classical conditioning principles to provide reinforcement signals to neurons so that they would learn to move from lower- to higher-valued states, thus anticipating the use of the TD error as a reinforcement signal in the actor–critic architecture. Miller's idea not only parallels Klopf's (with the exception of its explicit invocation of a distinct "strengthening signal"), it also anticipated the general features of reward-modulated STDP.

A related though different idea, which Seung (2003) called the "hedonistic synapse," is that synapses individually adjust the probability that they release neurotransmitter in the manner of the Law of Effect: if reward follows release, the release probability increases, and decreases if reward follows failure to release. This is essentially the same as the learning scheme Minsky used in his 1954 Princeton Ph.D. dissertation (Minsky, 1954), where he called the synapse-like learning element a SNARC (Stochastic Neural-Analog Reinforcement Calculator). Contingent eligibility is involved in these ideas too, although it is contingent on the activity of an individual synapse instead of the postsynaptic neuron.

Frey and Morris (1997) proposed the idea of a "synaptic tag" for the induction of long-lasting strengthening of synaptic efficacy. Though not unlike Klopf's eligibility, their tag was hypothesized to consist of a temporary strengthening of a synapse that could be transformed into a long-lasting strengthening by subsequent neuron activation. The model of O'Reilly and Frank (2006) and O'Reilly, Frank, Hazy, and Watz (2007) uses working memory to bridge temporal intervals instead of eligibility traces. Wickens and Kotter (1995) discuss possible mechanisms for synaptic eligibility. He, Huertas, Hong, Tie, Hell, Shouval, Kirkwood (2015) provide evidence supporting the existence of contingent eligibility traces in synapses of cortical neurons with time courses like those of the eligibility traces Klopf postulated.

The metaphor of a neuron using a learning rule related to bacterial chemotaxis was discussed by Barto (1989). Koshland's extensive study of bacterial chemotaxis was in part motivated by similarities between features of bacteria and features of neurons (Koshland, 1980). See also Berg (1975). Shimansky (2009) proposed a synaptic learning rule somewhat similar to Seung's mentioned above in which each synapse individually acts like a chemotactic bacterium. In this case a collection of synapses "swims" toward attractants in the high-dimensional space of synaptic weight values. Montague, Dayan, Person, and Sejnowski (1995) proposed a chemotactic-like model of the bee's foraging behavior involving the neuromodulator octopamine.

15.10 Research on the behavior of reinforcement learning agents in team and game problems has a long history roughly occurring in three phases. To the best of our knowledge, the first phase began with investigations by the Russian mathematician and physicist M. L. Tsetlin. A collection of

his work was published as Tsetlin (1973) after his death in 1966. Our Sections 1.7 and 4.8 refer to his study of learning automata in connection to bandit problems. The Tsetlin collection also includes studies of learning automata in team and game problems, which led to later work in this area using stochastic learning automata as described by Narendra and Thathachar (1974), Viswanathan and Narendra (1974), Lakshmivarahan and Narendra (1982), Narendra and Wheeler (1983), Narendra (1989), and Thathachar and Sastry (2002). Thathachar and Sastry (2011) is a more recent comprehensive account. These studies were mostly restricted to non-associative learning automata, meaning that they did not address associative, or contextual, bandit problems (Section 2.9).

The second phase began with the extension of learning automata to the associative, or contextual, case. Barto, Sutton, and Brouwer (1981) and Barto and Sutton (1981) experimented with associative stochastic learning automata in single-layer artificial neural networks to which a global reinforcement signal was broadcast. They called neuron-like elements implementing this kind of learning *associative search elements* (ASEs). Barto and Anandan (1985) introduced a more sophisticated associative reinforcement learning algorithm called the *associative reward-penalty* (A_{R-P}) algorithm. They proved a convergence result by combining theory of stochastic learning automata with theory of pattern classification. Barto (1985, 1986) and Barto and Jordan (1987) described results with teams of A_{R-P} units connected into multi-layer neural networks, showing that they could learn nonlinear functions, such as XOR and others, with a globally-broadcast reinforcement signal. Barto (1985) extensively discussed this approach to artificial neural networks and how this type of learning rule is related to others in the literature at that time. Williams (1992) mathematically analyzed and broadened this class of learning rules and related their use to the error backpropagation method for training multilayer artificial neural networks. Williams (1988) described several ways that backpropagation and reinforcement learning can be combined for training artificial neural networks. Williams (1992) showed that a special case of the A_{R-P} algorithm is a REINFORCE algorithm, although better results were obtained with the general A_{R-P} algorithm (Barto,1985).

The third phase of interest in teams of reinforcement learning agents was influenced by increased understanding of the role of dopamine as a widely broadcast neuromodulator and speculation about the existence of reward-modulated STDP. Much more so than earlier research, this research considers details of synaptic plasticity and other constraints from neuroscience. Publications include the following (chronologically and alphabetically): Bartlett and Baxter (1999, 2000), Xie and Seung (2004), Baras and Meir (2007), Farries and Fairhall (2007), Florian (2007), Izhikevich (2007), Pecevski, Maass, and Legenstein (2007), Legenstein, Pecevski, and Maass (2008), Kolodziejski, Porr, and Wörgötter (2009), Urbanczik and Senn (2009), and Vasilaki, Frémaux, Urbanczik, Senn, and Gerstner (2009). Nowé, Vrancx, and De Hauwere (2012) reviewed more recent developments in the wider field of multi-agent reinforcement learning

15.11 Yin and Knowlton (2006) reviewed findings from outcome-devaluation experiments with rodents supporting the view that habitual and goal-directed behavior (as psychologists use the phrase) are respectively most associated with processing in the dorsolateral striatum (DLS) and the dorsomedial striatum (DMS). Results of functional imaging experiments with human subjects in the outcome-devaluation setting by Valentin, Dickinson, and O'Doherty (2007) suggest that the orbitofrontal cortex (OFC) is an important component of goal-directed choice. Single unit recordings in monkeys by Padoa-Schioppa and Assad (2006) support the role of the OFC in encoding values guiding choice behavior. Rangel, Camerer, and Montague (2008) and Rangel and Hare (2010) reviewed findings from the perspective of neuroeconomics about how the brain makes goal-directed decisions. Pezzulo, van der Meer, Lansink, and Pennartz (2014) reviewed the neuroscience of internally generated sequences and presented a model of how these mechanisms might be components of model-based planning. Daw and Shohamy (2008)

proposed that while dopamine signaling connects well to habitual, or model-free, behavior, other processes are involved in goal-directed, or model-based, behavior. Data from experiments by Bromberg-Martin, Matsumoto, Hong, and Hikosaka (2010) indicate that dopamine signals contain information pertinent to both habitual and goal-directed behavior. Doll, Simon, and Daw (2012) argued that there may not a clear separation in the brain between mechanisms that subserve habitual and goal-directed learning and choice.

15.12 Keiflin and Janak (2015) reviewed connections between TD errors and addiction. Nutt, Lingford-Hughes, Erritzoe, and Stokes (2015) critically evaluated the hypothesis that addiction is due to a disorder of the dopamine system. Montague, Dolan, Friston, and Dayan (2012) outlined the goals and early efforts in the field of computational psychiatry, and Adams, Huys, and Roiser (2015) reviewed more recent progress.

16.6.2 AlphaGo Zero

Building upon the experience with *AlphaGo*, a DeepMind team developed *AlphaGo Zero* (Silver et al. 2017). In contrast to *AlphaGo*, this program used *no human data or guidance beyond the basic rules of the game* (hence the *Zero* in its name). It learned exclusively from self-play reinforcement learning, with input giving just "raw" descriptions of the placements of stones on the Go board. *AlphaGo Zero* implemented a form of policy iteration (Section 4.3), interleaving policy evaluation with policy improvement. Figure 16.8 is an overview of *AlphaGo Zero*'s algorithm. A significant difference between *AlphaGo Zero* and *AlphaGo* is that *AlphaGo Zero* used MCTS to select moves throughout self-play reinforcement learning, whereas *AlphaGo* used MCTS for live play after—but not during—learning. Other differences besides not using any human data or human-crafted features are that *AlphaGo Zero* used only one deep convolutional ANN and used a simpler version of MCTS.

AlphaGo Zero's MCTS was simpler than the version used by *AlphaGo* in that it did not include rollouts of complete games, and therefore did not need a rollout policy. Each iteration of *AlphaGo Zero*'s MCTS ran a simulation that ended at a leaf node of the current search tree instead of at the terminal position of a complete game simulation. But as in *AlphaGo*, each iteration of MCTS in *AlphaGo Zero* was guided by the output of a deep convolutional network, labeled f_θ in Figure 16.7, were θ is the network's weight vector. The input to the network, whose architecture we describe below, consisted of raw representations of board positions, and its output had two parts: a scaler value, v, an estimate of the probability that the current player will win from from the current board position, and a vector, $\mathbf{p}$, of move probabilities, one for each possible stone placement on the current board, plus the pass, or resign, move.

Instead of selecting self-play actions according to the probabilities $\mathbf{p}$, however, *AlphaGo Zero* used these probabilities, together with the network's value output, to direct each execution of MCTS, which returned new move probabilities, shown in Figure 16.7 as the policies π_i. These policies benefitted from the many simulations that MCTS conducted each time it executed. The result was that the policy actually followed by *AlphaGo Zero* was an improvement over the policy given by the network's outputs $\mathbf{p}$. Silver et al. (2017) wrote that "MCTS may therefore be viewed as a powerful *policy improvement* operator."

Here is more detail about *AlphaGo Zero*'s ANN and how it was trained. The network took as input a $19 \times 19 \times 17$ image stack consisting of 17 binary feature planes. The first 8 feature planes were raw representations of the positions of the current player's stones in the current and seven past board configurations: a feature value was 1 if a player's stone was on the corresponding point, and was 0 otherwise. The next 8 feature planes similarly coded the positions of the opponent's stones. A final input feature plane had a constant value indicating the color of the current play: 1 for black; 0 for white. Because repetition is not allowed in Go and one player is given some number of "compensation

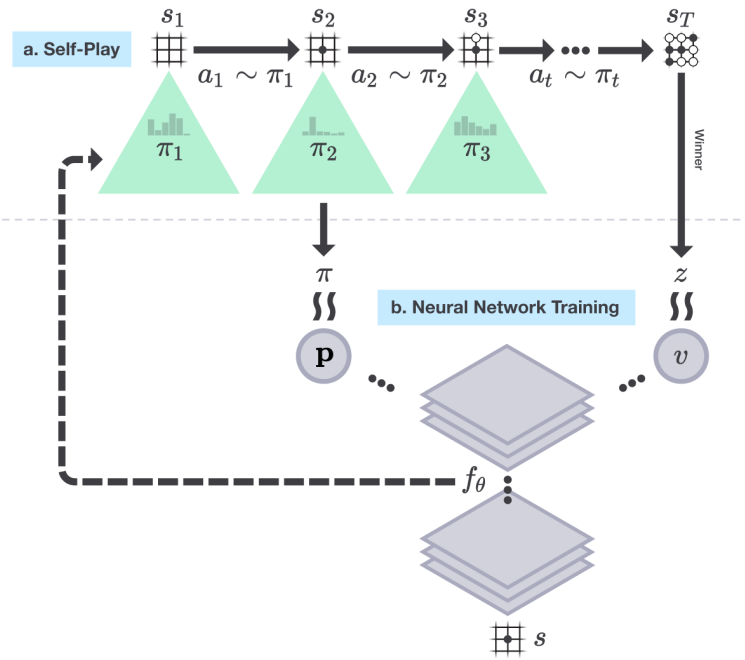


Figure 16.8: *AlphaGo Zero* self-play reinforcement learning. a) The program played many games against itself, one shown here as a sequence of board positions s_i, $i = 1, 2, \ldots, T$, with moves a_i, $i = 1, 2, \ldots, T$, and winner z. Each move a_i was determined by action probabilities π_i returned by MCTS executed from root node s_i and guided by a deep convolutional network, here labeled f_θ, with latest weights θ. Shown here for just one position s but repeated for all s_i, the network's inputs were raw representations of board positions s_i (together with several past position, though not shown here), and its outputs were vectors $\mathbf{p}$ of move probabilities that guided MCTS's forward searches, and scalar values v that estimated the probability of the current player winning from each position s_i. b) Deep convolutional network training. Training examples were randomly sampled steps from recent self-play games. Weights θ were updated to move the policy vector $\mathbf{p}$ toward the probabilities π returned by MCTS, and to include the winners z in the estimated win probability v. Reprinted from draft of Silver et al. (2017) with permission of the authors and DeepMind.

points" for not getting the first move, the current board position is not a Markov state of Go. This is why features describing past board positions and the color feature were needed.

The network was "two-headed," meaning that after a number of initial layers, the network split into two separate "heads" of additional layers that separately fed into two sets of output units. In this case, one head fed 362 output units producing $19^2 + 1$ move probabilities $\mathbf{p}$, one for each possible stone placement plus pass; the other head fed just one output unit producing the scalar v, an estimate of the probability that the current player will win from the current board position. The network before the split consisted of 41 convolutional layers, each followed by batch normalization, and with skip connections added to implement residual learning by pairs of layers (see Section 9.6). Overall, move probabilities and values were computed by 43 and 44 layers respectively.

Starting with random weights, the network was trained by stochastic gradient descent (with momentum, regularization, and step-size parameter decreasing as training continues) using batches of examples sampled uniformly at random from all the steps of the most recent 500,000 games of self-play with the current best policy. Extra noise was added to the network's output $\mathbf{p}$ to encourage exploration of all possible moves. At periodic checkpoints during training, which Silver et al. (2017) chose to be at every

1,000 training steps, the policy output by the ANN with the latest weights was evaluated by simulating 400 games (using MCTS with 1,600 iterations to select each move) against the current best policy. If the new policy won (by a margin set to reduce noise in the outcome), then it became the best policy to be used in subsequent self-play. The network's weights were updated to make the network's policy output $\mathbf{p}$ more closely match the policy returned by MCTS, and to make its value output, v, more closely match the probability that the current best policy wins from the board position represented by the network's input.

The DeepMind team trained *AlphaGo Zero* over 4.9 million games of self-play, which took about 3 days. Each move of each game was selected by running MCTS for 1,600 iterations, taking approximately 0.4 second per move. Network weights were updated over 700,000 batches each consisting of 2,048 board configurations. They then ran tournaments with the trained *AlphaGo Zero* playing against the version of *AlphaGo* that defeated Fan Hui by 5 games to 0, and against the version that defeated Lee Sedol by 4 games to 1. They used the Elo rating system to evaluate the relative performances of the programs. The difference between two Elo ratings is meant to predict the outcome of games between the players. The Elo ratings of *AlphaGo Zero*, the version of *AlphaGo* that played against Fan Hui, and the version that played against Lee Sedol were respectively 4,308, 3,144, and 3,739. The gaps in these Elo ratings translate into predictions that *AlphaGo Zero* would defeat these other programs with probabilities very close to one. In a match of 100 games between *AlphaGo Zero*, trained as described, and the exact version of *AlphaGo* that defeated Lee Sedol held under the same conditions that were used in that match, *AlphaGo Zero* defeated *AlphaGo* in all 100 games.

The DeepMind team also compared *AlphaGo Zero* with a program using an ANN with the same architecture but trained by supervised learning to predict human moves in a data set containing nearly 30 million positions from 160,000 games. They found that the supervised-learning player initially played better than *AlphaGo Zero*, and was better at predicting human expert moves, but played less well after *AlphaGo Zero* was trained for a day. This suggested that *AlphaGo Zero* had discovered a strategy for playing that was different from how humans play. In fact, *AlphaGo Zero* discovered, and came to prefer, some novel variations of classical move sequences.

Final tests of *AlphaGo Zero*'s algorithm were conducted with a version having a larger ANN and trained over 29 million self-play games, which took about 40 days, again starting with random weights. This version achieved an Elo rating of 5,185. The team pitted this version of *AlphaGo Zero* against a program called *AlphaGo Master*, the strongest program at the time, that was identical to *AlphaGo Zero* but, like *AlphaGo*, used human data and features. *AlphaGo Master*'s Elo rating was 4,858, and it had defeated the strongest human professional players 60 to 0 in online games. In a 100 game match, *AlphaGo Zero* with the larger network and more extensive learning defeated *AlphaGo Master* 89 games to 11, thus providing a convincing demonstration of the problem-solving power of *AlphaGo Zero*'s algorithm.

AlphaGo Zero soundly demonstrated that superhuman performance can be achieved by pure reinforcement learning, augmented by a simple version of MCTS, and deep ANNs with very minimal knowledge of the domain and no reliance on human data or guidance. We will surely see systems inspired by the DeepMind accomplishments of both *AlphaGo* and *AlphaGo Zero* applied to challenging problems in other domains.

16.7 Personalized Web Services

Personalizing web services such as the delivery of news articles or advertisements is one approach to increasing users' satisfaction with a website or to increase the yield of a marketing campaign. A policy can recommend content considered to be the best for each particular user based on a profile of that user's interests and preferences inferred from their history of online activity. This is a natural domain for machine learning, and in particular, for reinforcement learning. A reinforcement learning system

Chapter 17

Frontiers

In this final chapter we touch on some topics that are beyond the scope of this book but that we see as particularly important for the future of reinforcement learning. Many of these topics bring us beyond what is reliably known, and some bring us beyond the MDP framework.

17.1 General Value Functions and Auxiliary Tasks

Over the course of this book, our notion of value function has become quite general. With off-policy learning we allowed a value function to be conditional on an arbitrary target policy. Then in Section 12.8 we generalized discounting to a *termination function* $\gamma : \mathcal{S} \mapsto [0,1]$, so that a different discount rate could be applied at each time step in determining the return (12.24). This allowed us to express predictions about how much reward we will get over an arbitrary, state-dependent horizon. The next, and perhaps final, step is to generalize beyond rewards to permit predictions about arbitrary other signals. Rather than predicting the sum of future rewards, we might predict the sum of the future values of a sound or color sensation, or of an internal, highly processed signal such as another prediction. Whatever signal is added up in this way in a value-function-like prediction, we call it the *cumulant* of that prediction. We formalize it in a *cumulant signal* $C_t \in \mathbb{R}$. Using this, a *general value function*, or GVF, is written

$$v_{\pi,\gamma,C}(s) = \mathbb{E}\left[\sum_{k=t}^{\infty} C_{t+1} \prod_{i=t+1}^{k} \gamma(S_i) \;\middle|\; S_t = s, A_{t:\infty} \sim \pi\right]. \tag{17.1}$$

As with conventional value functions (such as v_π or q_*) this is an ideal function that we seek to approximate with a parameterized form, which we might continue to denote $\hat{v}(s,\mathbf{w})$, although of course there would have to be a different $\mathbf{w}$ for each prediction, that is, for each choice of π, γ, and C_t. Because a GVF has no necessory connection to reward, it is perhaps a misnomer to call it a *value* function. One could call it simply a prediction or, to make it more distinctive, a *forecast* (Ring, in preparation). Whatever it is called, it is in the form of a value function and thus can be learned in the usual ways using the methods developed in this book for learning approximate value functions. Along with the learned predictions, we might also learn policies to maximize the predictions in the usual GPI ways by greedification, or by actor–critic methods. In this way an agent could learn to predict and control great numbers of signals, not just long-term reward.

Why might it be useful to predict and control signals other than long-term reward? These are *auxiliary* tasks in that they are extra, in-addition-to, the main task of maximizing reward. One answer is that the ability to predict and control a diverse multitude of signals can constitute a powerful kind of environmental model. As we saw in Chapter 8, a good model can enable the agent to get reward more

efficiently. It takes a couple of further concepts to develop this answer clearly, so we postpone it to the next section. First let's consider two simpler ways in which a multitude of diverse predictions can be helpful to a reinforcement learning agent.

One simple way in which auxiliary tasks can help on the main task is that they may require some of the same representations as are needed on the main task. Some of the auxiliary tasks may be easier, with less delay and a clearer connection between actions and outcomes. If good features can be found early on easy auxilary tasks, then those features may significantly speed learning on the main task. There is no necessary reason why this has to be true, but in many cases it seems plausible. For example, if you learn to predict and control your sensors over short time scales, say seconds, then you might plausibly come up with part of the idea of objects, which would then greatly help with the prediction and control of long-term reward.

We might imagine an artificial neural network in which the last layer is split into multiple parts, or *heads*, each working on a different task. One head might produce the approximate value function for the main task (with reward as its cumulant) whereas the others would produce solutions to various auxilary tasks. All heads could propagate errors by stochastic gradient descent into the same body—the shared preceding part of the network—which would then try to form representations, in its next-to-last layer, to support all the heads. Researchers have experimented with auxiliary tasks such as predicting change in pixels, predicting the next-time-step's reward, and predicting the distribution of the return. In many cases this approach has been shown to greatly accelerate learning on the main task (Jaderberg et al., 2017). Multiple predictions have similarly been repeatedly proposed as a way of directing the construction of state estimates (see Section 17.3).

Another simple way in which the learning of auxiliary tasks can improve performance is best explained by analogy to the psychological phenomena of classical conditioning (Section 14.2). One way of understanding classical conditioning is that evolution has built in a reflexive (non-learned) association to a particular action from the prediction of a particular signal. For example, humans and many other animals appear to have a built-in reflex to blink whenever their prediction of being poked in the eye exceeds some threshold. The prediction is learned, but the association from prediction to blinking is built in, and thus the animal is saved many pokes in its eye. Similarly, the association from fear to increased heart rate, or to freezing, can be built in. Agent designers can do something similar, connecting by design (without learning) predictions of specific events to predetermined actions. For example, a self-driving car that learns to predict whether going forward will produce a collision could be given a built-in reflex to stop, or to turn away, whenever the prediction is above some threshold. Or consider a vacuum-cleaning robot that learned to predict whether it might run out of battery power before returning to the charger, and reflexively headed back to the charger whenever the prediction became non-zero. The correct prediction would depend on the size of the house, the room the robot was in, and the age of the battery, all of which would be hard for the robot designer to know. It would be difficult for the designer to build in a reliable algorithm for deciding whether to head back to the charger in sensory terms, but it might be easy to do this in terms of the learned prediction. We foresee many possible ways like this in which learned predictions might combine usefully with built-in algorithms for controlling behavior.

Finally, perhaps the most important role for auxiliary tasks is in moving beyond the assumption we have made throughout this book that the state representation is fixed and given to the agent. To explain this role, we first have to take a few steps back to appreciate the magnitude of this assumption and the implications of removing it. We do that in Section 17.3.

17.2 Temporal Abstraction via Options

An appealing aspect of the MDP formalism is that it can be applied usefully to tasks at many different time scales. One can use it to formalize the task of deciding which muscles to twitch to grasp an object,

which airplane flight to take to arrive conveniently at a distant city, and which job to take to lead a satisfying life. These tasks differ greatly in their time scales, yet each can be usefully formulated as an MDP that can be solved by planning or learning processes as described in this book. All involve interaction with the world, sequential decision making, and a goal usefully conceived of as accumulating rewards over time, and so all can be formulated as MDPs.

Although all these tasks can be formulated as MDPs, one might think that they cannot be formulated as a *single* MDP. They involve such different time scales, such different notions of choice and action! It would be no good, for example, to plan a flight across a continent at the level of muscle twitches. Yet for other tasks, grasping, throwing darts, or hitting a baseball, low-level muscle twitches may be just the right level. People do all these things seamlessly without appearing to switch between levels. Can the MDP framework be stretched to cover all the levels simultaneously?

Perhaps it can. One popular idea is to formalize an MDP at a detailed level, with a small time step, yet enable planning at higher levels using extended courses of action that correspond to many base-level time steps. To do this we need a notion of course of action that extends over many time steps and includes a notion of termination. A general way to formulate these two ideas are as a policy, π, and a state-dependent termination function, γ, as in GVFs. We define a pair of these as a generalized notion of action termed an *option*. To execute an option $\omega = \langle \pi_\omega, \gamma_\omega \rangle$ at time t is to obtain the action to take, A_t, from $\pi_\omega(\cdot|S_t)$, then terminate at time $t+1$ with probability $\gamma_\omega(S_{t+1})$. If the option does not terminate, then A_{t+1} is selected from $\pi_\omega(\cdot|S_{t+1})$, the option terminates at $t+2$ with probability $\gamma_\omega(S_{t+2})$, and so on until eventual termination. Such options are a strict generalization of low-level actions, which correspond to options $\langle \pi_\omega, \gamma_\omega \rangle$ in which the policy always picks the same action ($\pi_\omega(s) = a$ for all $s \in \mathcal{S}$) and in which the termination condition always terminates ($\gamma_\omega(s) = 0$ for all $s \in \mathcal{S}^+$). Options effectively extend the action space. The agent can either select a primitive action, terminating after one time step, or select an extended option that might execute for many time steps before terminating.

Options are designed so that they are interchangable with (primitive) actions. For example, the notion of an action-value function q_π naturally generalizes to an *option*-value function that takes a state and option as input and returns the expected return starting from that state, executing that option to termination, and thereafter following the policy, π. We can also generalize the notion of policy to a *hierarchical policy* that selects from options rather than actions, where options, when selected, execute until termination. With these ideas, many of the algorithms in this book can be generalized to learn approximate option-value functions and hierarchical policies. In the simplest case, the learning process 'jumps' from option initiation to option termination, with an update only occurring when an option terminates. More subtly, updates can be made on each time step, using *intra-option* learning algorithms, which in general require off-policy learning.

Perhaps the most important generalization made possible by option ideas is that of the environmental model as developed in Chapters 3, 4 and 8. The conventional model of an action is the state-transition probabilities and the expected immediate reward for taking the action in each state. How do conventional action models generalize to *option models*? For options, the appropriate model is again of two parts, one corresponding to the state transition resulting from executing the option and one corresponding to the expected cumulative reward along the way. The reward part of an option model, analogous to the expected reward for state–action pairs (3.5), is

$$r(s,\omega) \doteq \mathbb{E}\big[R_1 + \gamma R_2 + \gamma^2 R_3 + \cdots + \gamma^{\tau-1} R_\tau \;\big|\; S_0 = s, A_{0:\tau-1} \sim \pi_\omega, \tau \sim \gamma_\omega\big], \tag{17.2}$$

for all options ω and all states $s \in \mathcal{S}$, where τ is the random time step at which the option terminates according to γ_ω. Note the role of the overall discounting parameter γ in this equation—discounting is according to γ, but termination of the option is according to γ_ω. The state-transition part of an option model is a little more subtle. This part of the model characterizes the probability of each possible resulting state (as in (3.4)), but now this state may result after various numbers of time steps, each of which must be discounted differently. The model for option ω specifies, for each state s that ω might

start executing in, and for each state s' that ω might terminate in,

$$p(s' \,|\, s, \omega) \doteq \sum_{k=1}^{\infty} \Pr\{S_k = s', \tau = k \mid S_0 = s, A_{0:k-1} \sim \pi_\omega, \tau \sim \gamma_\omega\} \gamma^k . \tag{17.3}$$

Note that this $p(s'|s,\omega)$ is no longer a transition probability and no longer sums to one over all values of s'. (Nevertheless, we continue to use the '|' notation in p.)

The above definition of the transition part of an option model allows us to formulate Bellman equations and dynamic programming algorithms that apply to all options, including primitive actions as a special case. For example, the general Bellman equation for the state values of a hierarchical policy π is

$$v_\pi(s) = \sum_{\omega \in \Omega(s)} \pi(\omega|s) \left[r(s,\omega) + \sum_{s'} p(s' \,|\, s, \omega) v_\pi(s') \right], \tag{17.4}$$

where $\Omega(s)$ denotes the set of options available in state s. If $\Omega(s)$ includes only the primitive actions, then this equation reduces to a version of the usual Bellman equation (3.14), except of course γ is included in the new p (17.3) and thus does not appear. Similarly, the corresponding planning algorithms also have no γ. For example, the value iteration algorithm with options, analogous to (4.10), is

$$v_{k+1}(s) \doteq \max_{\omega \in \Omega(s)} \left[r(s,\omega) + \sum_{s'} p(s' \,|\, s, \omega) v_k(s') \right], \text{ for all } s \in \mathcal{S}. \tag{17.5}$$

If $\Omega(s)$ includes all the primitive actions available in each s, then this algorithm converges to the conventional v_*, from which the optimal policy can be computed. However, it is particularly useful to plan with options when only a subset of the possible options are considered (in $\Omega(s)$) in each state. Value iteration will then converge to the best hierarchical policy limited to the restricted set of options. Although this policy may be sub-optimal, convergence can be much faster because fewer options are considered and because each option can jump over many time steps.

To plan with options, one must either be given the option models, or learn them. One natural way to learn an option model is to formulate it as a collection of GVFs (as defined in the preceding section) and then learn the GVFs using the methods presented in this book. It is not difficult to see how this could be done for the reward part of the option model. One merely chooses one GVF's cumulant to be the reward ($C_t = R_t$), its policy to be the the option's policy ($\pi = \pi_\omega$), and its termination function to be the discount rate times the option's termination function ($\gamma(s) = \gamma \cdot \gamma_\omega(s)$). The true GVF then equals the reward part of the option model, $v_{\pi,\gamma,C}(s) = r(s,\omega)$, and the learning methods described in this book can be used to approximate it. The state-transition part of the option model is only a little more complicated. One needs to allocate one GVF for each state that the option might terminate in. We don't want these GVFs to accumulate anything except when the option terminates, and then only when the termination is in the appropriate state. This can be achieved by choosing the cumulant of the GVF that predicts transition to state s' to be $C_t = \gamma(S_t) \cdot \mathbb{1}_{S_t = s'}$. The GVF's policy and termination functions are chosen the same as for the reward part of the option model. The true GVF then equals the s' portion of the option's state-transition model, $v_{\pi,\gamma,C}(s) = p(s' \,|\, s, \omega)$, and again this book's methods could be employed to learn it. Although each of these steps is seemingly natural, putting them all together (including function approximation and other essential components) is quite challenging and beyond the current state of the art.

Exercise 17.1 This section has presented options for the discounted case, but discounting is arguably inappropriate for control when using function approximation (Section 10.4). What is the natural Bellman equation for a hierarchical policy, analogous to (17.4), but for the average reward setting (Section 10.3)? What are the two parts of the option model, analogous to (17.2) and (17.3), for the average reward setting?

17.3 Observations and State

Throughout this book we have written the learned approximate value functions (and the policies in Chapter 13) as functions of the environment's state. This is a significant limitation of the methods presented in Part I, in which the learned value function was implemented as a table such that any value function could be exactly approximated; that case is tantamount to assuming that the state of the environment is completely observed by the agent. But in many cases of interest, and certainly in the lives of all natural intelligences, the sensory input gives only partial information about the state of the world. Some objects may be ocluded by others, or behind the agent, or miles away. In these cases, potentially important aspects of the environment's state are not directly observable, and it is a strong, unrealistic, and limiting assumption to assume that the learned value function is implemented as a table over the environment's state space.

On the other hand, the framework of parametric function approximation that we developed in Part II is far less restrictive and, arguably, is no limitation at all. In Part II we retained the assumption that the learned value functions (and policies) are functions of the environment's state, but allowed these functions to be arbitrarily restricted by the parameterization. It is somewhat surprising and not widely recognized, but function approximation includes important aspects of partial observability. For example, if there is some state variable that is not observable, then the parameterization can be chosen such that the approximate value does not depend on that state variable. The effect is just as if that state variable was not observable. Because of this, all the results obtained for the parameterized case apply to partial observability without change. In this sense, the case of parameterized function approximation includes the case of partial observability.

Nevertheless, there are many issues that cannot be investigated without a more explicit treatment of partial observability. Although we cannot give them a full treatment here, we can outline the changes that would be needed to do so. There are four steps.

First, we would change the problem. The environment would emit not its states, but only *observations*—signals that depend on its state but, like a robot's sensors, provide only partial information about it. For convenience, without loss of generality, we assume that the reward is a direct, known function of the observation (perhaps the observation is a vector, and the reward is one of is components). The environmental interaction would then have no explicit states or rewards, but would simply be an alternating sequence of actions $A_t \in \mathcal{A}$ and observations $O_t \in \mathcal{O}$:

$$A_0, O_1, A_1, O_2, A_2, O_3, A_3, O_4, \ldots,$$

going on forever (cf. Equation 3.1) or forming episodes each ending with a special terminal observation.

Second, we can recover the idea of state as used in this book from the sequence of observations and actions. Let us use the word *history*, and the notation H_t, for an initial portion of the trajectory up to an observation: $H_t \doteq A_0, O_1, \ldots, A_{t-1}, O_t$. The history represents the most that we can know about the past without looking outside of the data stream (because the history is the whole past data stream). Of course, the history grows with t and can become large and unwieldy. The idea of state is that of some compact summary of the history that is as useful as the actual history for predicting the future. Let us be clear about exactly what this means. To be a summary of the history, the state must be a function of history, $S_t = f(H_t)$, and to be as useful for predicting the future as the whole history is known as the *Markov property*. Formally, this is a property of the function f. A function f has the Markov property if and only if any two histories h and h' that are mapped by f to the same state ($f(h) = f(h')$) also have the same probabilities for their next observation,

$$f(h) = f(h') \quad \Rightarrow \quad \Pr\{O_t = o | H_t = h, A_t = a\} = \Pr\{O_{t+1} = o | H_t = h', A_t = a\}, \tag{17.6}$$

for all $o \in \mathcal{O}$ and $a \in \mathcal{A}$. If f is Markov, then $S_t = f(H_t)$ is a state as we have used the term in this book. Let us henceforth call it a *Markov state* to distinguish it from states that are summaries of the history but fall short of the Markov property (which we will consider shortly).

A Markov state is a good basis for predicting the next observation (17.6) but, more importantly, it is also a good basis for predicting or controlling *anything*. For example, let a *test* be any specific sequence of alternating actions and observations that might occur in the future. For example, a three-step test is denoted $\tau = a_1\, o_1\, a_2, o_2, a_3, o_3$. The probability of this test given a specific history h is defined as

$$p(\tau|h) \doteq \Pr\{O_{t+1}{=}o_1, O_{t+2}{=}o_2, O_{t+3}{=}o_3 \mid H_t{=}h, A_t{=}a_1, A_{t+1}{=}a_2, A_{t+2}{=}a_3\}. \tag{17.7}$$

If f is Markov and h and h' are any two histories that map to the same state under f, then for any test τ of any length, its probabilities given the two histories must also be the same:

$$f(h) = f(h') \quad \Rightarrow \quad p(\tau|h) = p(\tau|h'). \tag{17.8}$$

In other words, a Markov state summarizes all the information in the history necessary for determining any test's probability. In fact, it summarizes all that is necessary for making *any prediction*, including any GVF, and for behaving optimally (if f is Markov, then there is always a deterministic function π such that choosing $A_t \doteq \pi(f(H_t))$ is optimal).

The third step in extending reinforcement learning to partial observability is to deal with certain computational considerations. In particular, we want the state to be a *compact* summary of the history. For example, the identity function completely satisfies the conditions for a Markov f, but would nevertheless be of little use because the corresponding state $S_t{=}H_t$ would grow with time and becomes unwieldy, as mentioned earlier, but more fundamentally because it would never recur; the agent would never encounter the same state twice (in a continuing task) and thus could never benefit from a tabular learning method. We want our states to be compact as well as Markov. There is a similar issue regarding how state is obtained and updated. We don't really want a function f that takes whole histories. Instead, for computational reasons we prefer to obtain the same effect as f with an incremental, recursive update that computes S_{t+1} from S_t, incorporating the next increment of data, A_t and O_{t+1}:

$$S_{t+1} = u(S_t, A_t, O_{t+1}), \text{ for all } t \geq 0, \tag{17.9}$$

with the first state S_0 given. The function u is called the *state-update* function. For example, if f were the identity ($S_t{=}H_t$), then u would merely extend S_t by appending A_t and O_{t+1} to it. Given f, it is always possible to construct a corresponding u, but it may not be computationally convenient and, as in the identity example, it may not produce a compact state. The state-update function is a central part of any agent architecture that handles partial observability. It must be efficiently computatible, as no actions or predictions can be made until the state is available.

An example of obtaining Markov states through a state-update function is provided by the popular Bayesian approach known as *Partially Observable MDPs*, or *POMDPs*. In this approach the environment is assumed to have a well defined *latent state* X_t that underlies and produces the environment's observations, but is never available to the agent (and is not to be confused with the state S_t used by the agent to make predictions and decisions). The natural Markov state S_t for a POMDP is the *distribution* over the latent states given the history, called the *belief state*. For concreteness, assume the usual case in which there are a finite number of hidden states, $X_t \in \{1, 2, \ldots, d\}$. Then the belief state is the vector $S_t \doteq \mathbf{s}_t \in \mathbb{R}^d$ with components

$$\mathbf{s}_t[i] \doteq \Pr\{X_t{=}i \mid H_t\}, \text{ for all possible latent states } i \in \{1, 2, \ldots, d\}. \tag{17.10}$$

The belief state remains the same size (same number of components) however t grows. It can also be incrementally updated by Bayes rule, assuming one has complete knowledge of the internal workings of the environment. Specifically, the ith component of the belief-state update function is

$$u(\mathbf{s}, a, o)[i] = \frac{\sum_{x=1}^{d} \mathbf{s}[x]\, p(i, o|x, a)}{\sum_{x=1}^{d} \sum_{x'=1}^{d} \mathbf{s}[x]\, p(x', o|x, a)}, \tag{17.11}$$

for all $a \in \mathcal{A}$, $o \in \mathcal{O}$, and belief states $\mathbf{s} \in \mathbb{R}^d$ with components $\mathbf{s}[x]$, where the four-argument p function here is not the usual one for MDPs (as in Chapter 3), but the analogous one for POMDPs, in terms of the *latent* state: $p(x', o|x, a) \doteq \Pr\{X_t = x', O_t = o \mid X_{t-1} = x, A_{t-1} = a\}$. This approach is popular in theoretical work and has many significant applications, but its assumptions and computational complexity scale poorly and we do not recommend it as an approach to artificial intelligence.

Another example of Markov states is provided by *Predictive State Representations*, or *PSRs*. PSRs address a weakness of the POMDP approach that the semantics of its agent state S_t are grounded in the environment state, X_t, which is never observed and thus is difficult to learn about. In PSRs and related approaches, the semantics of the agent state is instead grounded in predictions about future observations and actions, which are readily observable. In PSRs, a Markov state is defined as a d-vector of the probabilities of d specially chosen "core" tests as defined above (17.7). The vector is then updated by a state-update function u that is analogous to Bayes rule, but with a semantics grounded in observable data, which arguably makes it easier to learn. This approach has been extended in many ways, including end-tests, compositional tests, powerful "spectral" methods, and closed-loop and temporally abstract tests learned by TD methods. Some of the best theoretical developments are for systems known as *Observable Operator Models* (OOMs) and Sequential Systems (Thon, 2017).

The fourth and final step in our brief outline of how to handle partial observability in reinforcement learning is to re-introduce approximation. As discussed in the introduction to Part II, to approach artificial intelligence ambitiously one must embrace approximation. This is just as true for states as it is for value functions. We must accept and work with an approximate notion of state. The approximate state will play the same role in our algorithms as before, so we continue use the notation S_t for the state used by the agent, even though it may not be Markov.

Perhaps the simplest example of an approximate state is just the latest observation, $S_t \doteq O_t$. Of course this approach cannot handle any hidden state information. Better is to use the last k observations and actions, $S_t \doteq O_t, A_{t-1}, O_{t-1}, \ldots, O_{t-k}$, for some $k \geq 1$, which can be achieved by a state-update function that just shifts the new data in and the oldest data out. This *kth-order history* approach is still very simple, but can greatly increase the agent's capabilities compared to trying to use the single immediate observation directly as the state.

What happens when the Markov property (17.6) is only approximately satisfied? Prediction performance can degrade dramatically. Longer-term tests, GVFs, and state-update functions may all approximate poorly with an approximate state even if the one-step predictions (17.6) defining the Markov property are well approximated with it. There are essentially no useful theoretical guarantees at present.

Nevertheless, there are still reasons to think that the general idea outlined in this section applies to the approximate case. The general idea is that a state good for some predictions is also good for others (in particular, that a Markov state, sufficient for one-step predictions, is also sufficient for all others). If we step back from that specific result for the Markov case, the general idea is similar to what we discussed in Section 17.1 with multi-headed learning and auxiliary tasks. We discussed how representations that were good for the auxiliary tasks were often also good for the main task. Taken together, these suggest an approach to both partial observability and representation learning in which multiple predictions are pursued and used to direct the contruction of state features. The guarantee provided by the perfect-but-impractical Markov property is replaced by the heuristic that what's good for some predictions may be good for others. This approach scales well with computational resources. With a large machine one could experiment with large numbers of predictions, perhaps favoring those that are most similar to the ones of ultimate interest, that are easiest to learn reliably, or by other criteria. Key here is to move beyond selecting the predictions manually. The agent should do it. This would require a general language for predictions, so that the agent can systematically explore a large space of possible predictions, sifting through them for the ones that are most useful.

17.4 Designing Reward Signals

A major advantage of reinforcement learning over supervised learning is that reinforcement learning does not rely on detailed instructional information: designing reward signals does not depend on knowing what the correct actions should be. But not all reward signals are created equal. The success of a reinforcement learning application often strongly depends on how well the reward signal frames the problem and how well it assesses progress in solving it. Future applications can benefit from a better understanding of how reward signals affect learning and from improved methods for designing them.

The usual way to use reinforcement learning to solve a problem is to reward the agent according to its success in solving the problem. This is relatively easy for many problems, but some problems have goals that are difficult to translate into reward signals. This is especially true when the problem is to get an agent to skillfully perform a complex task. Even when there is a simple goal that is easy to identify, the problem of sparse rewards often arises. Delivering non-zero reward frequently enough to allow the agent to achieve the goal once, let alone to learn to achieve it efficiently from multiple initial conditions, can be a daunting challenge. Further, reinforcement learning agents can discover unexpected ways to make their environments deliver reward, some of which might be undesirable, or even dangerous. For these reasons, designing reward signals is a critical part of any reinforcement learning application.

In practice, designing reward signals is often left to an informal trial-and-error search for a signal that produces acceptable results. If the agent fails to learn, learns too slowly, or learns the wrong thing, the designer tweaks the reward signal and tries again. To do this, the designer judges the agent's performance by criteria that he or she is attempting to translate into reward signals so that the the agent's goal matches his or her own. Some more sophisticated ways to find good reward signals have been proposed, but the subject has interesting and relatively unexplored dimensions.

It is tempting to address the sparse reward problem by rewarding the agent for achieving subgoals that the designer thinks are important way stations to the overall goal. But augmenting the reward signal with well-intentioned supplemental rewards may lead the agent to behave very differently from what is intended; the agent may end up not achieving the overall goal at all. A better way to provide such guidance is to leave the reward signal alone and instead augment the value-function approximation with an initial guess of what it should ultimately be. For example, suppose one wants to offer $v_0 : \mathcal{S} \to \mathbb{R}$ as an initial guess at the true optimal value function v_*, and that one is using linear function approximation with feature $\mathbf{x} : \mathcal{S} \to \mathbb{R}^d$. Then one would define the approximate value function as

$$\hat{v}(s,\mathbf{w}) \doteq \mathbf{w}^\top \mathbf{x}(s) + v_0(s), \tag{17.12}$$

and update the weights $\mathbf{w}$ as usual. If the initial weight vector is $\mathbf{0}$, then the initial value function will be v_0, but the asymptotic solution quality will be determined by the feature vectors as usual. This initialization works for arbitrary nonlinear approximators and arbitrary forms of v_0. Wiewiora (2003) showed that this initialization is equivalent to the more complex "potential-based shaping" technique for changing rewards described by Ng, Harada, and Russell (1999).

What if one has no idea what the rewards should be but there is another agent, perhaps a person, who is already expert at the task and whose behavior can be observed? In this case one can use a variety of methods known variously as "imitation learning," "learning from demonstration," and "apprenticeship learning." The idea here is to benefit from the expert agent but leave open the possibility of eventually performing even better. Learning from an expert's behavior can be done either by learning directly by supervised learning or by extracting a reward signal using what is known as "inverse reinforcement learning" and then using a reinforcement learning algorithm with that reward function to learn a policy. The task of inverse reinforcement learning as explored by Ng and Russell (2000) is to recover the expert's reward signal (within a scalar constant) from the expert's behavior alone. This cannot be done exactly because a policy can be optimal with respect to many different reward signals (always including any reward signal that gives the same reward for all states and actions), but it is possible to find plausible reward-signal candidates. However, some strong assumptions are required, including

knowledge of the feature vectors in which the reward signal is linear and complete knowledge of the environment's dynamics. The method also requires completely solving the problem (e.g., by dynamic programming methods) multiple times. These difficulties notwithstanding, Abbeel and Ng (2004) argue that the inverse reinforcement approach can sometimes be more effective than supervised learning for benefiting from the behavior of an expert.

Another approach to finding a good reward signal is based on automating the trial-and-error search for such a signal that we mentioned above. From an engineering perspective, the reward signal is a parameter of the learning algorithm. As is true for other algorithm parameters, the search for a good reward signal can be automated by defining a space of feasible candidates and applying an optimization algorithm. The optimization algorithm evaluates each candidate reward signal by running the reinforcement learning system with that signal for some number of steps, and then scoring the overall result by a "high-level" objective function intended to encode the designer's true goal, ignoring the limitations of the agent. In some cases, reward signals can be improved via online gradient ascent, again as evaluated by a high-level objective function (Sorg, Lewis, and Singh, 2010). Relating this to the natural world, the the algorithm for optimizing the high-level objective function is analogous to evolution, where the high-level objective function is like an animal's evolutionary fitness determined by the number of its offspring that survive to reproductive age.

Experiments with this bilevel optimization approach (Singh, Lewis, and Barto, 2009) confirmed that intuition alone is not always adequate to devise good reward signals. The performance of a reinforcement learning agent as evaluated by the high-level objective function can be very sensitive to details of the agent's reward signal in subtle ways determined by the agent's limitations and the environments in which it acts and learns. These experiments also demonstrated that an agent's goal should not always be the same as the goal of the agent's designer.

At first this seems counterintuitive, but it may be impossible for the agent to achieve the designer's goal no matter what its reward signal is because the agent has to learn under various kinds of constraints, such as having limited computational power, limited access to information about its environment, or limited time to learn. When there are constraints like these, learning to achieve a goal that is different from the designer's goal can sometimes end up getting closer to the designer's goal than if that goal were pursued directly (Sorg, Singh, and Lewis, 2010; Sorg, 2011). There are also abundant examples of this in the natural world. Since we cannot directly detect the nutritional value of most foods, evolution—the designer of our reward signal—produced a reward signal that makes us seek certain tastes. Though certainly not infallible (indeed, possibly detrimental in environments that differ in certain ways from the ancestral environments), this compensates for many of our limitations: our limited sensory abilities, the limited time over which we can learn, and the risks involved in finding a diet through personal reinforcement learning. Similarly, since an animal cannot observe its own evolutionary fitness, that evaluation function does not work as a reward signal for learning (although predictors of evolutionary fitness certainly can be observed and figure prominently in animals' reward signals).

Another dimension to devising reward signals is whether the agent is to learn to solve a specific problem, or if instead, it is to learn skills that can be useful across many different problems that the agent is likely to face in the future. Pursuing the latter goal has led to the idea of implementing in reinforcement learning something like what psychologists call "intrinsic motivation." Where "extrinsic motivation" means doing something because of some specific rewarding outcome, "intrinsic motivation" refers to doing something "for its own sake." Intrinsic motivation leads animals to engage in exploration, play, and other behavior driven by curiosity in the absence of problem-specific rewards. The true value of what is learned via intrinsically-motivated behavior (which for an animal would be the value of the evolutionary advantage it confers) emerges over long-term experience with many different specific tasks.

Giving an agent something analogous to intrinsic motivation can be done by devising a reward signal that helps an agent learn widely useful skills, including skills that aid the learning process itself. Reward signals can depend on such things as a general ability to cause changes in the environment, assessments

of general progress in learning, or other measures that do not depend on a goal of performing a specific task. An example is the "bonus reward" described in Section 8.3. Instead of being tied to a specific task, this reward signal encourages exploration in general, which benefits the learning of many specific tasks. Another example is the proposal by Schmidhuber (1991a, b) for how something like curiosity would result if reward signals were a function of how quickly an agent's environment model was improving in predicting state transitions. Many preliminary studies of such computational curiosity have been conducted and are exciting topics of ongoing research.

17.5 Remaining Issues

In this book we have presented the foundations of a reinforcement learning approach to artificial intelligence. Roughly speaking, that approach is based on model-free and model-based methods working together, as in the Dyna architecture of Chapter 8, combined with function approximation as developed in Part II. The focus has been on online and incremental algorithms, which we see as fundamental even to model-based methods, and on how these can be applied in off-policy training situations. The full rationale for the latter has been presented only in this last chapter. That is, we have all along presented off-policy learning as an appealing way to deal with the explore/exploit dilemma, but only in this chapter have we talked about learning about many diverse auxiliary tasks simultaneously with GVFs, and about understanding the world hierarchically in terms of temporally-abstract option models, both of which seem to ineluctably involve off-policy learning. Much remains to be worked out, as we have indicated throughout the book and as evidenced by the directions for additional research discussed in this chapter. But suppose we are generous and grant the broad outlines of everything that we have done in the book *and* everthing that has been outlined in this chapter. What would remain even after that? Of course, we can't know for sure what will be required, but we can make some guesses. In this section we highlight four further issues which it seems to us will still need to be addressed by future research.

First, we still need powerful parametric function approximation methods that work well in fully incremental and online settings. Methods based on deep learning and artificial neural networks are a major step in this direction, but rely on batch training with large data sets, extensive off-line self play, or learning asynchronously from multiple simultaneous streams agent-environment interaction. These and other techniques are ways of working around a basic limitation of today's deep learning methods, which struggle to learn rapidly in the incremental, online settings that are most natural for reinforcement learning settings and that we have emphasized in this book. The problem is sometimes described as one of "correlated data" or "catastrophic interference". When something new is learned it tends to replace what has previously been learned rather than adding to it, such that the benefit of the older learning is lost. Techniques such as "replay buffers" are often used to retain and replay old data so that its benefits are not permanently lost. An honest assessment has to be that current deep learning methods just don't learn well online. We don't see any reason why they couldn't, but the learning algorithms to do this have not yet been devised, and the bulk of current research is directed toward working around this limitation of current algorithms rather than to removing it.

Second, and perhaps closely related, we still need methods for learning features such that subsequent learning generalizes well. This issue is an instance of a general problem variously called "representation learning," "constructive induction," and "meta-learning"—how can we use experience not just to learn a given a desired function, but to learn inductive biases such that future learning generalizes better and is thus faster? This is an old problem, dating back to the origins of artificial intelligence and pattern recognition in the 1950s and 1960s. (Some would claim that deep learning solves this problem, but we consider it still unsolved.) Such age should give one pause. Perhaps there is no solution? But just as likely the time has not previously been ripe for any solution being found and being shown effective. Today machine learning is conducted at a far larger scale and the benefits of a good representation

learning method are potentially much more apparent. We note that a new annual conference—the International Conference on Learning Representations—has been exploring this and related topics every year since 2013. It is also new to explore representation learning in a reinforcement learning context. Reinforcement learning brings some new possibilities to this old issue, such as the auxiliary tasks discussed in Section 17.1. In reinforcement learning, the problem of representation learning can be identified with the problem of learning the state-update function discussed in Section 17.3.

Third, we still need scalable methods for planning with learned models. Planning methods have proven extremely effective in applications such as AlphaGo Zero and computer chess in which the model of the environment is known from the rules of the game or can otherwise be designed in by people. But cases of full model-based reinforcement learning, in which the environmental model is learned from data and then used for planning, are rare. The Dyna system described in Chapter 8 is one example, but as described there and in most subsequent work it used a tabular model without function approximation of any sort, which greatly limits its applicability. There have been only a few studies with learned linear models, and even fewer that have also tried to incorporate temporally abstract models using options as discussed in Section 17.2.

These limitations are a problem because they greatly limit the effectiveness of planning. In particular, model making needs to be selective because the contents of the model strongly affect planning efficiency. If the model is focused on the key consequences of the most important possible options, then planning can be efficient and rapid, but if the model details the unimportant consequences of options that are unlikely to be chosen, then planning may be almost useless. Environmental models must be constructed judiciously in both their states and dynamics so as to optimize the planning process. The various parts of the model will have to be continually monitored for the degree to which they contribute to or detract from planning efficiency. The field has not yet come to grips with this complex of issues or designed model-learning methods that take into account their implications. To make a good model that supports planning is analogous to obtaining a true understanding of the environment that enables reasoning to obtain a goal. As such it would be a significant milestone in artificial intelligence.

The fourth and final issue that strikes us as as needing to be addressed in future research is that of automating the choice of subproblems on which an agent works and which it uses to structure its developing mind. In machine learning, designers are used to setting the problems or tasks for the learning agent. Because these tasks are fixed, we build them into the code for the learning algorithm. However, looking ahead we will want the agent to make its own choices about what tasks to work on. These tasks may be like the auxiliary tasks or the GVFs discussed in Section 17.1. In forming a GVF, for example, what should the cumulant, the policy, and the termination function be? The current state of the art is to select these manually, but far greater power and generality would come from making these task choices automatically, particularly when they are from things previously constructed by the agent as a result of representation learning or previous subproblems. If GVF design is automated, then the design choices themselves will have to be explicitly represented. Rather than the task choices being in the mind of the designer and built into the code, they will have to be in the machine itself in such a way that they can be set and changed, monitored, filtered, and searched among automatically. Tasks could then be built hierarchically upon one each other much like features are in a neural network. The tasks are the questions and the contents of the neural network the answers to those questions. We expect there will need to be a full hierarchy of questions to match the hierarchy of the answers in modern deep learning.

17.6 Reinforcement Learning and the Future of Artificial Intelligence

When we were writing the first edition of this book in the mid-1990s, artificial intelligence was making significant progress and was having an impact on society, though it was mostly still the *promise* of artificial intelligence that was inspiring developments. Machine learning was part of that outlook, but it had not yet become indispensable to artificial intelligence. By today that promise has transitioned to applications that are changing the lives of millions of people, and machine learning has come into its own as a key technology. As we write this second edition, some of the most remarkable developments in artificial intelligence have involved reinforcement learning, most notably "deep reinforcement learning"—reinforcement learning with function approximation by deep neural networks. We are at the beginning of a wave of real-world applications of artificial intelligence, many of which will include reinforcement learning, deep and otherwise, that will impact our lives in ways that are hard to predict.

But an abundance of successful real-world applications does not mean that true artificial intelligence has arrived. Despite great progress in many areas, the gulf between artificial intelligence and the intelligence of humans, and even of other animals, remains great. Superhuman performance can be achieved in some domains, even formidable domains like Go, but it remains a significant challenge to develop systems that are like us in being complete, interactive agents having general adaptability and problem-solving skills, emotional sophistication, creativity, and the ability to learn quickly from experience. With its focus on learning by interacting with dynamic environments, reinforcement learning, as it develops over the future, will be a critical component of agents with these abilities.

Reinforcement learning's connections to psychology and neuroscience (Chapters 14 and 15) underscore its relevance to another longstanding goal of artificial intelligence: shedding light on fundamental questions about the mind and how it emerges from the brain. Reinforcement learning theory is already contributing to our understanding of natural reward, motivation, and decision-making systems, understanding that can contribute to improving human abilities to learn, to remain motivated, and to make decisions. There is also good reason to believe that through its links to computational psychiatry, reinforcement learning theory will contribute to methods for treating mental disorders, including drug abuse and addiction.

Another contribution that reinforcement learning can make over the future is as an aid to human decision making. Policies derived by reinforcement learning in simulated environments can advise human decision makers in such areas as education, healthcare, transportation, energy, and public-sector resource allocation. Particularly relevant is the key feature of reinforcement learning that it takes long-term consequences of decisions into account. This is very clear in games like backgammon and Go, where some of the most impressive results of reinforcement learning have been demonstrated, but it is also a property of many high-stakes decisions that affect our lives and our planet. Reinforcement learning follows related methods for advising human decision making that have been developed in the past by decision analysts in many disciplines. With advanced function approximation methods and massive computational power, reinforcement learning methods have the potential to overcome some of the difficulties of scaling up traditional decision-support methods to larger and more complex problems.

The rapid pace of advances in artificial intelligence has led to warnings that artificial intelligence poses serious threats to our societies, even to humanity itself. The renowned scientist and artificial intelligence pioneer Herbert Simon anticipated the warnings we are hearing today in a presentation at the Earthware Symposium at CMU in 2000 (Simon, 2000). He spoke of the eternal conflict between the promise and perils of any new knowledge, reminding us of the Greek myths of Prometheus, the hero of modern science, who stole fire from the gods for the benefit of mankind, and Pandora, whose box could be opened by a small and innocent action to release untold perils on the world. While accepting that this conflict is inevitable, Simon urged us to recognize that as designers of our future and not simply as spectators, the decisions **we** make can tilt the scale in Prometheus' favor. This is certainly

true for reinforcement learning, which can benefit society but can also produce undesirable outcomes if it is carelessly deployed. Thus, the *safety* of artificial intelligence applications involving reinforcement learning is a topic that deserves careful attention.

A reinforcement learning agent can learn by interacting with either the real world or with a simulation of some piece of the real world, or by a mixture of these two sources of experience. Simulators provide safe environments in which an agent can explore and learn without risking real damage to itself or to its environment. In most current applications, policies are learned from simulated experience instead of direct interaction with the real world. In addition to avoiding undesirable real-world consequences, learning from simulated experience can make virtually unlimited data available for learning, generally at less cost than needed to obtain real experience, and since simulations typically run much faster than real time, learning can often occur more quickly than if it relied on real experience.

Nevertheless, the full potential of reinforcement learning requires reinforcement learning agents to be embedded into the flow of real-world experience, where they act, explore, and learn in *our* world, and not just in *their* worlds. After all, reinforcement learning algorithms—at least those upon which we focus in this book—are designed to learn online, and they emulate many aspects of how animals are able to survive in nonstationary and hostile environments. Embedding reinforcement learning agents in the real world can be transformative in realizing the promises of artificial intelligence to amplify and extend human abilities.

A major reason for wanting a reinforcement learning agent to act and learn in the real world is that it is often difficult, sometimes impossible, to simulate real-world experience with enough fidelity to make the resulting policies, whether derived by reinforcement learning or by other methods, work well—and safely—when directing real actions. This is especially true for environments whose dynamics depend on the behavior of humans, such as in education, healthcare, transportation, and public policy, domains that can surely benefit from improved decision making. However, it is for real-world embedded agents that warnings about potential dangers of artificial intelligence need to be heeded.

Reinforcement learning is a collection of optimization methods, so it inherits the pluses and minuses of all optimization methods. On the minus side is the problem we mentioned at several places above: how do you devise objective functions, or reward signals in the case of reinforcement learning, so that optimization produces the desired results while avoiding undesirable results? This is hardly a new problem with reinforcement learning; recognition of it has a long history in literature and engineering. The founder of cybernetics, Norbert Weiner, for one, warned of this more than half a century ago by relating the supernatural story of "The Monkey's Paw" (Weiner, 1964): wishes are granted but come with unacceptable cost. The problem has also been discussed at length in a modern context by Nick Bostrom (2014). Anyone having experience with reinforcement learning has likely seen their systems discover unexpected ways to obtain a lot of reward. Sometimes the unexpected behavior is good: it solves a problem in a nice new way. In other instances, what the agent learns violates considerations that the system designer may never have thought about. Careful design of reward signals is essential if an agent is to act in the real world with no opportunity for human vetting of its actions or means to easily interrupt its behavior.

Despite the possibility of unintended negative consequences, optimization has been used for hundreds of years by engineers, architects, and others whose designs have positively impacted the world. Many approaches have been developed to mitigate the risk of optimization, such as adding hard and soft constraints, restricting optimization to robust and risk-sensitive policies, and optimizing with multiple objective functions. Some of these have been adapted to reinforcement learning. We owe much that is good in our environment to the application of optimization methods. Still, the problem of ensuring that a reinforcement learning agent's goal is attuned to our own remains a challenge.

Another challenge if reinforcement learning agents are to act and learn in the real world is not just about what they might eventually learn, but about how they will behave while they are learning. How do you make sure that an agent gets enough experience to learn a high-performing policy, all the while

not harming its environment, other agents, or itself (or more realistically, while keeping the probability of harm acceptably low)? This problem is also not novel or unique to reinforcement learning. Risk management and mitigation for embedded reinforcement learning is similar to what control engineers have had to confront from the beginning of using automatic control in situations where a controller's behavior can have unacceptable, possibly catastrophic, consequences, as in the control of an aircraft or a delicate chemical process. Control applications rely on careful system modeling, model validation, and extensive testing, and there is a highly-developed body of theory aimed at ensuring convergence and stability of adaptive controllers designed for use when the dynamics of the system to be controlled are not fully known. Theoretical guarantees are never iron-clad because they depend on the validity of the assumptions underlying the mathematics, but without this theory, combined with risk-management and mitigation practices, automatic control—adaptive and otherwise—would not be as beneficial as it is today in improving the quality, efficiency, and cost-effectiveness of processes on which we have come to rely. Some of this theory has been adapted to reinforcement learning to help prevent unwanted behavior during, and after, learning, but many future applications of reinforcement learning are likely to be in domains that are less constrained than those to which control theory and practice readily apply. Developing methods to make it acceptably safe to fully embed reinforcement learning agents into physical environments is one of the most pressing areas for future research.

In closing, we return to Simon's call for us to recognize that we are designers of our future and not simply spectators. By decisions we make as individuals, and by the influence we can exert on how our societies are governed, we can work toward ensuring that the benefits made possible by a new technology outweigh the harm it can cause. There is ample opportunity to do this in the case of reinforcement learning, which can help improve the quality, fairness, and sustainability of life on our planet, but which can also release new perils. A threat already here is the displacement of jobs caused by applications of artificial intelligence. Still there are good reasons to believe that the benefits of artificial intelligence can outweigh the disruption it causes. As to safety, hazards possible with reinforcement learning are not completely different from those that have been managed successfully for related applications of optimization and control methods. As reinforcement learning moves out into the real world in future applications, developers have an obligation to follow best practices that have evolved for similar technologies, while at the same time extending them to make sure that Prometheus keeps the upper hand.

Bibliographical and Historical Remarks

17.1 General value functions were first explicitly identified by Sutton and colleagues (Sutton, 1995; Sutton et al., 2011; Modayil, White and Sutton, 2013). Ring (in preparation) developed an extensive thought experiment with GVFs ("forecasts") that has been influential despite not yet having been published.

The first demonstrations of multi-headed learning in reinforcement learning were by Jaderberg et alia (2017). Bellemare, Dabney and Munos (2017) showed that predicting more things about the distribution of reward could significantly speed learning to optimize its expectation, an instance of auxiliary tasks. Many others have since taken up this line of research.

The view of classical conditioning as learned predictions together with built-in, reflexive reactions to the predictions has not to our knowledge been clearly articulated in the psychological literature. Modayil and Sutton (2014) describe it as an approach to the engineering of robots and other agents, calling it "Pavlovian control" to allude to its roots in classical conditioning.

17.2 The formalization of temporally abstract courses of action as options was introduced by Sutton, Precup, and Singh (1999), building on prior work by Parr (1998) and Sutton (1995a), and on classical work on Semi-MDPs (e.g., see Puterman, 1994). Precup's (2000) PhD thesis developed

option ideas fully. An important limitation of these early works is that they treated either the tabular case or the on-policy case. The general case of intra-option learning involves off-policy learning, which could not be done reliably with function approximation at that time. Although now we have a variety of stable off-policy learning methods using function approximation, their combination with option ideas had not been significantly explored at the time of publication of this book.

Using GVFs to implement option models has not previously been described. Our presentation uses the trick introduced by Modayil, White and Sutton (2014) for predicting signals at the termination of policies.

Among the few works that have learned option models with function approximation are those by Bacon, Harb, and Precup (2017).

The extension of options and option models to the average-reward setting has not yet been developed in the literature.

17.3 For a good intuitive discussion of the system-theoretic concept of state, see Minsky (1967). A good presentation of the POMDP approach is given by Monahan (1982). PSRs and tests were introduced by Littman, Sutton and Singh (2002). OOMs were introduced by Jaeger (1997, 1998, 2000). Sequential Systems, which unify PSRs, OOMs, and many other works, were introduced in the PhD thesis of Michael Thon (2017; Thon and Jaeger, 2015).

The theory of reinforcement learning with a non-Markov state representation was developed explicitly by Singh, Jaakkola, and Jordan (1994).

17.5 The problem of catastrophic interference in artificial neural networks was developed by McCloskey and Cohen (1989), Ratcliff (1990), and French (1999). The idea of a replay buffer was introduced by Lin (1992) and used prominently in deep learning in the Atari game playing system (Section 16.5, Minh et al., 2013, 2015).

Minsky (1961) was one of the first to identify the problem of representation learning.

Among the few works to consider planning with learned, approximate models are those by Kuvayev and Sutton (1998), Sutton, Szepesvari, Geramifard, and Bowling (2008), and Nouri and Littman (2009).

The need to be selective in model construction to avoid slowing planning is well known in artificial intelligence. Some of the classic work is by Minton (1990) and Tambe, Newell, and Rosenbloom (1990). Hauskrecht, Mieuleau, Boutilier, Kaelbling, and Dean (1998) showed this effect in MDPs with deterministic options.