

very bad decisions on board configurations that never occur in games against experts. In fact, it is possible that TD-Gammon makes bad decisions for a large fraction of the game's state set. The online nature of reinforcement learning makes it possible to approximate optimal policies in ways that put more effort into learning to make good decisions for frequently encountered states, at the expense of less effort for infrequently encountered states. This is one key property that distinguishes reinforcement learning from other approaches to approximately solving MDPs.

3.8 Summary

Let us summarize the elements of the reinforcement learning problem that we have presented in this chapter. Reinforcement learning is about learning from interaction how to behave in order to achieve a goal. The reinforcement learning *agent* and its *environment* interact over a sequence of discrete time steps. The specification of their interface defines a particular task: the *actions* are the choices made by the agent; the *states* are the basis for making the choices; and the *rewards* are the basis for evaluating the choices. Everything inside the agent is known and controllable. Its environment, on the other hand, is incompletely controllable and may or may not be completely known. A *policy* is a stochastic rule by which the agent selects actions as a function of states. The agent's objective is to maximize the amount of reward it receives over time.

When the reinforcement learning setup described above is formulated with well defined transition probabilities it constitutes a *Markov decision process* (MDP). A *finite MDP* is an MDP with finite state, action, and (as we formulate it here) reward sets. Much of the current theory of reinforcement learning is restricted to finite MDPs, but the methods and ideas apply more generally.

The *return* is the function of future rewards that the agent seeks to maximize (in expected value). It has several different definitions depending upon the nature of the task and whether one wishes to *discount* delayed reward. The undiscounted formulation is appropriate for *episodic tasks*, in which the agent–environment interaction breaks naturally into *episodes*; the discounted formulation is appropriate for *continuing tasks*, in which the interaction does not naturally break into episodes but continues without limit. We try to define the returns for the two kinds of tasks such that one set of equations can apply to both the episodic and continuing cases.

A policy's *value functions* (v_π and q_π) assign to each state, or state–action pair, the expected return from that state, or state–action pair, given that the agent uses the policy. The *optimal value functions* (v_* and q_*) assign to each state, or state–action pair, the largest expected return achievable by any policy. A policy whose value functions are optimal is an *optimal policy*. Whereas the optimal value functions for states and state–action pairs are unique for a given MDP, there can be many optimal policies. Any policy that is *greedy* with respect to the optimal value functions must be an optimal policy. The *Bellman optimality equations* are special consistency conditions that the optimal value functions must satisfy and that can, in principle, be solved for the optimal value functions, from which an optimal policy can be determined with relative ease.

A reinforcement learning problem can be posed in a variety of different ways depending on assumptions about the level of knowledge initially available to the agent. In problems of *complete knowledge*, the agent has a complete and accurate model of the environment's dynamics. If the environment is an MDP, then such a model consists of the complete four-argument dynamics function p (3.2). In problems of *incomplete knowledge*, a complete and perfect model of the environment is not available.

Even if the agent had a complete and accurate environment model, the agent would typically be unable to fully use it because of limitations on its memory and computation per time step. In particular, extensive memory may be required to build up accurate approximations of value functions, policies, and models. In most cases of practical interest there are far more states than could possibly be entries in a table, and approximations must be made.

A well-defined notion of optimality organizes the approach to learning we describe in this book and provides a way to understand the theoretical properties of various learning algorithms, but it is an ideal that reinforcement learning agents can only approximate to varying degrees. In reinforcement learning we are very much concerned with cases in which optimal solutions cannot be found but must be approximated in some way.

Bibliographical and Historical Remarks

The reinforcement learning problem is deeply indebted to the idea of Markov decision processes (MDPs) from the field of optimal control. These historical influences and other major influences from psychology are described in the brief history given in Chapter 1. Reinforcement learning adds to MDPs a focus on approximation and incomplete information for realistically large problems. MDPs and the reinforcement learning problem are only weakly linked to traditional learning and decision-making problems in artificial intelligence. However, artificial intelligence is now vigorously exploring MDP formulations for planning and decision making from a variety of perspectives. MDPs are more general than previous formulations used in artificial intelligence in that they permit more general kinds of goals and uncertainty.

The theory of MDPs is treated by, for example, Bertsekas (2005), White (1969), Whittle (1982, 1983), and Puterman (1994). A particularly compact treatment of the finite case is given by Ross (1983). MDPs are also studied under the heading of stochastic optimal control, where *adaptive* optimal control methods are most closely related to reinforcement learning (e.g., Kumar, 1985; Kumar and Varaiya, 1986).

The theory of MDPs evolved from efforts to understand the problem of making sequences of decisions under uncertainty, where each decision can depend on the previous decisions and their outcomes. It is sometimes called the theory of multistage decision processes, or sequential decision processes, and has roots in the statistical literature on sequential sampling beginning with the papers by Thompson (1933, 1934) and Robbins (1952) that we cited in Chapter 2 in connection with bandit problems (which are prototypical MDPs if formulated as multiple-situation problems).

The earliest instance (that we are aware of) in which reinforcement learning was discussed using the MDP formalism is Andreae's (1969) description of a unified view of

On problems with large state spaces, *asynchronous* DP methods are often preferred. To complete even one sweep of a synchronous method requires computation and memory for every state. For some problems, even this much memory and computation is impractical, yet the problem is still potentially solvable because relatively few states occur along optimal solution trajectories. Asynchronous methods and other variations of GPI can be applied in such cases and may find good or optimal policies much faster than synchronous methods can.

4.8 Summary

In this chapter we have become familiar with the basic ideas and algorithms of dynamic programming as they relate to solving finite MDPs. *Policy evaluation* refers to the (typically) iterative computation of the value function for a given policy. *Policy improvement* refers to the computation of an improved policy given the value function for that policy. Putting these two computations together, we obtain *policy iteration* and *value iteration*, the two most popular DP methods. Either of these can be used to reliably compute optimal policies and value functions for finite MDPs given complete knowledge of the MDP.

Classical DP methods operate in sweeps through the state set, performing an *expected update* operation on each state. Each such operation updates the value of one state based on the values of all possible successor states and their probabilities of occurring. Expected updates are closely related to Bellman equations: they are little more than these equations turned into assignment statements. When the updates no longer result in any changes in value, convergence has occurred to values that satisfy the corresponding Bellman equation. Just as there are four primary value functions (v_π , v_* , q_π , and q_*), there are four corresponding Bellman equations and four corresponding expected updates. An intuitive view of the operation of DP updates is given by their *backup diagrams*.

Insight into DP methods and, in fact, into almost all reinforcement learning methods, can be gained by viewing them as *generalized policy iteration* (GPI). GPI is the general idea of two interacting processes revolving around an approximate policy and an approximate value function. One process takes the policy as given and performs some form of policy evaluation, changing the value function to be more like the true value function for the policy. The other process takes the value function as given and performs some form of policy improvement, changing the policy to make it better, assuming that the value function is its value function. Although each process changes the basis for the other, overall they work together to find a joint solution: a policy and value function that are unchanged by either process and, consequently, are optimal. In some cases, GPI can be proved to converge, most notably for the classical DP methods that we have presented in this chapter. In other cases convergence has not been proved, but still the idea of GPI improves our understanding of the methods.

It is not necessary to perform DP methods in complete sweeps through the state set. *Asynchronous DP* methods are in-place iterative methods that update states in an arbitrary order, perhaps stochastically determined and using out-of-date information. Many of these methods can be viewed as fine-grained forms of GPI.

Finally, we note one last special property of DP methods. All of them update estimates of the values of states based on estimates of the values of successor states. That is, they update estimates on the basis of other estimates. We call this general idea *bootstrapping*. Many reinforcement learning methods perform bootstrapping, even those that do not require, as DP requires, a complete and accurate model of the environment. In the next chapter we explore reinforcement learning methods that do not require a model and do not bootstrap. In the chapter after that we explore methods that do not require a model but do bootstrap. These key features and properties are separable, yet can be mixed in interesting combinations.

Bibliographical and Historical Remarks

The term “dynamic programming” is due to Bellman (1957a), who showed how these methods could be applied to a wide range of problems. Extensive treatments of DP can be found in many texts, including Bertsekas (2005, 2012), Bertsekas and Tsitsiklis (1996), Dreyfus and Law (1977), Ross (1983), White (1969), and Whittle (1982, 1983). Our interest in DP is restricted to its use in solving MDPs, but DP also applies to other types of problems. Kumar and Kanal (1988) provide a more general look at DP.

To the best of our knowledge, the first connection between DP and reinforcement learning was made by Minsky (1961) in commenting on Samuel’s checkers player. In a footnote, Minsky mentioned that it is possible to apply DP to problems in which Samuel’s backing-up process can be handled in closed analytic form. This remark may have misled artificial intelligence researchers into believing that DP was restricted to analytically tractable problems and therefore largely irrelevant to artificial intelligence. Andreae (1969) mentioned DP in the context of reinforcement learning, specifically policy iteration, although he did not make specific connections between DP and learning algorithms. Werbos (1977) suggested an approach to approximating DP called “heuristic dynamic programming” that emphasizes gradient-descent methods for continuous-state problems (Werbos, 1982, 1987, 1988, 1989, 1992). These methods are closely related to the reinforcement learning algorithms that we discuss in this book. Watkins (1989) was explicit in connecting reinforcement learning to DP, characterizing a class of reinforcement learning methods as “incremental dynamic programming.”

4.1–4 These sections describe well-established DP algorithms that are covered in any of the general DP references cited above. The policy improvement theorem and the policy iteration algorithm are due to Bellman (1957a) and Howard (1960). Our presentation was influenced by the local view of policy improvement taken by Watkins (1989). Our discussion of value iteration as a form of truncated policy iteration is based on the approach of Puterman and Shin (1978), who presented a class of algorithms called *modified policy iteration*, which includes policy iteration and value iteration as special cases. An analysis showing how value iteration can be made to find an optimal policy in finite time is given by Bertsekas (1987).

Iterative policy evaluation is an example of a classical successive approximation algorithm for solving a system of linear equations. The version of the algorithm

It follows immediately that there is an alternate importance-sampling estimator, with the same unbiased expectation (in the first-visit case) as the ordinary-importance-sampling estimator (5.5), using \tilde{G}_t :

$$V(s) \doteq \frac{\sum_{t \in \mathcal{T}(s)} \tilde{G}_t}{|\mathcal{T}(s)|}, \quad (5.15)$$

which we might expect to sometimes be of lower variance.

Is there a per-decision version of *weighted* importance sampling? This is less clear. So far, all the estimators that have been proposed for this that we know of are not consistent (that is, they do not converge to the true value with infinite data).

**Exercise 5.13* Show the steps to derive (5.14) from (5.12). \square

**Exercise 5.14* Modify the algorithm for off-policy Monte Carlo control (page 111) to use the idea of the truncated weighted-average estimator (5.10). Note that you will first need to convert this equation to action values. \square

5.10 Summary

The Monte Carlo methods presented in this chapter learn value functions and optimal policies from experience in the form of *sample episodes*. This gives them at least three kinds of advantages over DP methods. First, they can be used to learn optimal behavior directly from interaction with the environment, with no model of the environment's dynamics. Second, they can be used with simulation or *sample models*. For surprisingly many applications it is easy to simulate sample episodes even though it is difficult to construct the kind of explicit model of transition probabilities required by DP methods. Third, it is easy and efficient to *focus* Monte Carlo methods on a small subset of the states. A region of special interest can be accurately evaluated without going to the expense of accurately evaluating the rest of the state set (we explore this further in Chapter 8).

A fourth advantage of Monte Carlo methods, which we discuss later in the book, is that they may be less harmed by violations of the Markov property. This is because they do not update their value estimates on the basis of the value estimates of successor states. In other words, it is because they do not bootstrap.

In designing Monte Carlo control methods we have followed the overall schema of *generalized policy iteration* (GPI) introduced in Chapter 4. GPI involves interacting processes of policy evaluation and policy improvement. Monte Carlo methods provide an alternative policy evaluation process. Rather than use a model to compute the value of each state, they simply average many returns that start in the state. Because a state's value is the expected return, this average can become a good approximation to the value. In control methods we are particularly interested in approximating action-value functions, because these can be used to improve the policy without requiring a model of the environment's transition dynamics. Monte Carlo methods intermix policy evaluation and policy improvement steps on an episode-by-episode basis, and can be incrementally implemented on an episode-by-episode basis.

Maintaining *sufficient exploration* is an issue in Monte Carlo control methods. It is not enough just to select the actions currently estimated to be best, because then no returns will be obtained for alternative actions, and it may never be learned that they are actually better. One approach is to ignore this problem by assuming that episodes begin with state-action pairs randomly selected to cover all possibilities. Such *exploring starts* can sometimes be arranged in applications with simulated episodes, but are unlikely in learning from real experience. In *on-policy* methods, the agent commits to always exploring and tries to find the best policy that still explores. In *off-policy* methods, the agent also explores, but learns a deterministic optimal policy that may be unrelated to the policy followed.

Off-policy prediction refers to learning the value function of a *target policy* from data generated by a different *behavior policy*. Such learning methods are based on some form of *importance sampling*, that is, on weighting returns by the ratio of the probabilities of taking the observed actions under the two policies, thereby transforming their expectations from the behavior policy to the target policy. *Ordinary importance sampling* uses a simple average of the weighted returns, whereas *weighted importance sampling* uses a weighted average. Ordinary importance sampling produces unbiased estimates, but has larger, possibly infinite, variance, whereas weighted importance sampling always has finite variance and is preferred in practice. Despite their conceptual simplicity, off-policy Monte Carlo methods for both prediction and control remain unsettled and are a subject of ongoing research.

The Monte Carlo methods treated in this chapter differ from the DP methods treated in the previous chapter in two major ways. First, they operate on sample experience, and thus can be used for direct learning without a model. Second, they do not bootstrap. That is, they do not update their value estimates on the basis of other value estimates. These two differences are not tightly linked, and can be separated. In the next chapter we consider methods that learn from experience, like Monte Carlo methods, but also bootstrap, like DP methods.

Bibliographical and Historical Remarks

The term “Monte Carlo” dates from the 1940s, when physicists at Los Alamos devised games of chance that they could study to help understand complex physical phenomena relating to the atom bomb. Coverage of Monte Carlo methods in this sense can be found in several textbooks (e.g., Kalos and Whitlock, 1986; Rubinstein, 1981).

5.1–2 Singh and Sutton (1996) distinguished between every-visit and first-visit MC methods and proved results relating these methods to reinforcement learning algorithms. The blackjack example is based on an example used by Widrow, Gupta, and Maitra (1973). The soap bubble example is a classical Dirichlet problem whose Monte Carlo solution was first proposed by Kakutani (1945; see Hersh and Griego, 1969; Doyle and Snell, 1984).

Barto and Duff (1994) discussed policy evaluation in the context of classical Monte Carlo algorithms for solving systems of linear equations. They used the

6.9 Summary

In this chapter we introduced a new kind of learning method, temporal-difference (TD) learning, and showed how it can be applied to the reinforcement learning problem. As usual, we divided the overall problem into a prediction problem and a control problem. TD methods are alternatives to Monte Carlo methods for solving the prediction problem. In both cases, the extension to the control problem is via the idea of generalized policy iteration (GPI) that we abstracted from dynamic programming. This is the idea that approximate policy and value functions should interact in such a way that they both move toward their optimal values.

One of the two processes making up GPI drives the value function to accurately predict returns for the current policy; this is the prediction problem. The other process drives the policy to improve locally (e.g., to be ε -greedy) with respect to the current value function. When the first process is based on experience, a complication arises concerning maintaining sufficient exploration. We can classify TD control methods according to whether they deal with this complication by using an on-policy or off-policy approach. Sarsa is an on-policy method, and Q-learning is an off-policy method. Expected Sarsa is also an off-policy method as we present it here. There is a third way in which TD methods can be extended to control which we did not include in this chapter, called actor-critic methods. These methods are covered in full in Chapter 13.

The methods presented in this chapter are today the most widely used reinforcement learning methods. This is probably due to their great simplicity: they can be applied online, with a minimal amount of computation, to experience generated from interaction with an environment; they can be expressed nearly completely by single equations that can be implemented with small computer programs. In the next few chapters we extend these algorithms, making them slightly more complicated and significantly more powerful. All the new algorithms will retain the essence of those introduced here: they will be able to process experience online, with relatively little computation, and they will be driven by TD errors. The special cases of TD methods introduced in the present chapter should rightly be called *one-step, tabular, model-free* TD methods. In the next two chapters we extend them to n -step forms (a link to Monte Carlo methods) and forms that include a model of the environment (a link to planning and dynamic programming). Then, in the second part of the book we extend them to various forms of function approximation rather than tables (a link to deep learning and artificial neural networks).

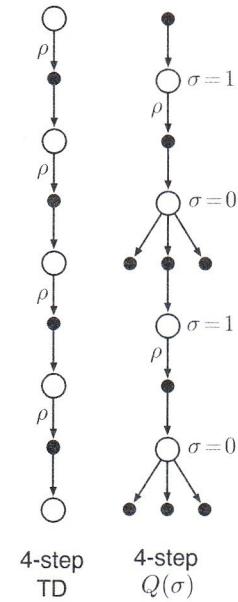
Finally, in this chapter we have discussed TD methods entirely within the context of reinforcement learning problems, but TD methods are actually more general than this. They are general methods for learning to make long-term predictions about dynamical systems. For example, TD methods may be relevant to predicting financial data, life spans, election outcomes, weather patterns, animal behavior, demands on power stations, or customer purchases. It was only when TD methods were analyzed as pure prediction methods, independent of their use in reinforcement learning, that their theoretical properties first came to be well understood. Even so, these other potential applications of TD learning methods have not yet been extensively explored.

7.7 Summary

In this chapter we have developed a range of temporal-difference learning methods that lie in between the one-step TD methods of the previous chapter and the Monte Carlo methods of the chapter before. Methods that involve an intermediate amount of bootstrapping are important because they will typically perform better than either extreme.

Our focus in this chapter has been on n -step methods, which look ahead to the next n rewards, states, and actions. The two 4-step backup diagrams to the right together summarize most of the methods introduced. The state-value update shown is for n -step TD with importance sampling, and the action-value update is for n -step $Q(\sigma)$, which generalizes Expected Sarsa and Q-learning. All n -step methods involve a delay of n time steps before updating, as only then are all the required future events known. A further drawback is that they involve more computation per time step than previous methods. Compared to one-step methods, n -step methods also require more memory to record the states, actions, rewards, and sometimes other variables over the last n time steps. Eventually, in Chapter 12, we will see how multi-step TD methods can be implemented with minimal memory and computational complexity using eligibility traces, but there will always be some additional computation beyond one-step methods. Such costs can be well worth paying to escape the tyranny of the single time step.

Although n -step methods are more complex than those using eligibility traces, they have the great benefit of being conceptually clear. We have sought to take advantage of this by developing two approaches to off-policy learning in the n -step case. One, based on importance sampling is conceptually simple but can be of high variance. If the target and behavior policies are very different it probably needs some new algorithmic ideas before it can be efficient and practical. The other, based on tree-backup updates, is the natural extension of Q-learning to the multi-step case with stochastic target policies. It involves no importance sampling but, again if the target and behavior policies are substantially different, the bootstrapping may span only a few steps even if n is large.



2.10 Summary

We have presented in this chapter several simple ways of balancing exploration and exploitation. The ϵ -greedy methods choose randomly a small fraction of the time, whereas UCB methods choose deterministically but achieve exploration by subtly favoring at each step the actions that have so far received fewer samples. Gradient bandit algorithms estimate not action values, but action preferences, and favor the more preferred actions in a graded, probabilistic manner using a soft-max distribution. The simple expedient of initializing estimates optimistically causes even greedy methods to explore significantly.

It is natural to ask which of these methods is best. Although this is a difficult question to answer in general, we can certainly run them all on the 10-armed testbed that we have used throughout this chapter and compare their performances. A complication is that they all have a parameter; to get a meaningful comparison we have to consider their performance as a function of their parameter. Our graphs so far have shown the course of learning over time for each algorithm and parameter setting, to produce a *learning curve* for that algorithm and parameter setting. If we plotted learning curves for all algorithms and all parameter settings, then the graph would be too complex and crowded to make clear comparisons. Instead we summarize a complete learning curve by its average value over the 1000 steps; this value is proportional to the area under the learning curve. Figure 2.6 shows this measure for the various bandit algorithms from this chapter, each as a function of its own parameter shown on a single scale on the x-axis. This kind of graph is called a *parameter study*. Note that the parameter values are varied by factors of two and presented on a log scale. Note also the characteristic inverted-U shapes of each algorithm's performance; all the algorithms perform best at an intermediate value of their parameter, neither too large nor too small. In assessing

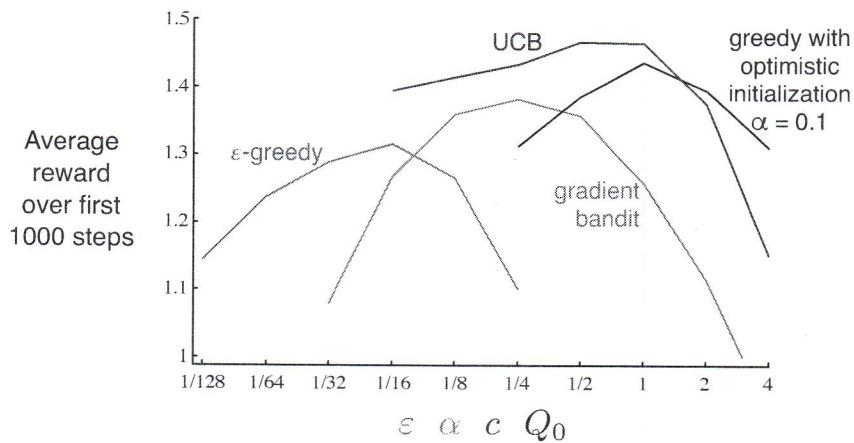


Figure 2.6: A parameter study of the various bandit algorithms presented in this chapter. Each point is the average reward obtained over 1000 steps with a particular algorithm at a particular setting of its parameter.

a method, we should attend not just to how well it does at its best parameter setting, but also to how sensitive it is to its parameter value. All of these algorithms are fairly insensitive, performing well over a range of parameter values varying by about an order of magnitude. Overall, on this problem, UCB seems to perform best.

Despite their simplicity, in our opinion the methods presented in this chapter can fairly be considered the state of the art. There are more sophisticated methods, but their complexity and assumptions make them impractical for the full reinforcement learning problem that is our real focus. Starting in Chapter 5 we present learning methods for solving the full reinforcement learning problem that use in part the simple methods explored in this chapter.

Although the simple methods explored in this chapter may be the best we can do at present, they are far from a fully satisfactory solution to the problem of balancing exploration and exploitation.

One well-studied approach to balancing exploration and exploitation in k -armed bandit problems is to compute a special kind of action value called a *Gittins index*. In certain important special cases, this computation is tractable and leads directly to optimal solutions, although it does require complete knowledge of the prior distribution of possible problems, which we generally assume is not available. In addition, neither the theory nor the computational tractability of this approach appear to generalize to the full reinforcement learning problem that we consider in the rest of the book.

The Gittins-index approach is an instance of *Bayesian* methods, which assume a known initial distribution over the action values and then update the distribution exactly after each step (assuming that the true action values are stationary). In general, the update computations can be very complex, but for certain special distributions (called *conjugate priors*) they are easy. One possibility is to then select actions at each step according to their posterior probability of being the best action. This method, sometimes called *posterior sampling* or *Thompson sampling*, often performs similarly to the best of the distribution-free methods we have presented in this chapter.

In the Bayesian setting it is even conceivable to compute the *optimal* balance between exploration and exploitation. One can compute for any possible action the probability of each possible immediate reward and the resultant posterior distributions over action values. This evolving distribution becomes the *information state* of the problem. Given a horizon, say of 1000 steps, one can consider all possible actions, all possible resulting rewards, all possible next actions, all next rewards, and so on for all 1000 steps. Given the assumptions, the rewards and probabilities of each possible chain of events can be determined; one need only pick the best. But the tree of possibilities grows extremely rapidly; even if there were only two actions and two rewards, the tree would have 2^{2000} leaves. It is generally not feasible to perform this immense computation exactly, but perhaps it could be approximated efficiently. This approach would effectively turn the bandit problem into an instance of the full reinforcement learning problem. In the end, we may be able to use approximate reinforcement learning methods such as those presented in Part II of this book to approach this optimal solution. But that is a topic for research and beyond the scope of this introductory book.