Contents

0.1 Definitions

- *Stochastic* - randomly determined.
- *Markov chain* - stochastic model describing a sequence of states in which the probability of coming into any state is solely defained and attained from the previous state.
- *Markov property* - a property of a process where he condtional probability distribution of future states is solely dependent on the current state: memorylessness.
- *Markov process* - a stochastic process that satisfies the *Markov property*.
- *Markov decision process* (MDP) - discrete time stochastic control process. Modeling decision making in situations where outcomes are partly random, partly in control of the decision maker. An MDP is essentially a *Markov chain* with added actions and rewards: interaction with an agent.
- *Markov reward process* (MRP) - an *MDP* without actions, the transitions are defined by a probability function.

Chapter 1

Introduction

RL is the optimal control of incompletely-known Markov decision process.

1.1 Basic Elements

- Policy - mapping from state to action built and adjusted through learning.
- Reward - what the environment returns to the agent based on the current state and the action taken. Essentially tells the agent how desireable the current env state is.
- Value function - similar to the reward function, but rather than immediate, it specifies what is good in the long run. Value of a state is the total reward an agent can expect starting from that state. The reward is a predecessor to value and without it value would not exist. Although this is the case, in RL we focus on maximizing value, as it looks into long-term reward maximization. The reward comes from the environment, but the value is approximated by the agent and its experience with the env. The estimation and constant updating of the value function is the core of RL.
- Environment model - essentially models a given env and provides an interface to it that the agent uses to learn.

1.2 Summary

Reinforcement learning is a computational approach to understanding and automating goal-directed learning and decision making. It is distinguished from other computational approaches by its emphasis on learning by an agent from direct interaction with its environment, without requiring exemplary supervision or complete models of the environment. In our opinion, reinforcement learning is the first field to seriously address the computational issues that arise when learning from interaction with an environment in order to achieve long-term goals. Reinforcement learning uses the formal framework of Markov decision processes to define the interaction between a learning agent and its environment in terms of states, actions, and rewards.

This framework is intended to be a simple way of representing essential features of the artificial intelligence problem. These features include a sense of cause and effect, a sense of uncertainty and nondeterminism, and the existence of explicit goals. The concepts of value and value function are key to most of the reinforcement learning methods that we consider in this book. We take the position that value functions are important for efficient search in the space of policies. The use of value functions distinguishes reinforcement learning methods from evolutionary methods that search directly in policy space guided by evaluations of entire policies.

Part I

Tabular Solution Methods

Chapter 2

Multi-armed Bandits

2.1 k-armed Bandit Problem

With every step you take one of the k actions and get a reward. The goal is to maximize the reward in the long run. We attempt to figure out what the value function for a given action is. EXPLOITATION vs. EXPLORATION

2.2 Action-value Methods

Action-value methods are methods that help assess the value of taking a given action.

- ϵ-greedy - making the greedy choice most of the time and every so often, with probability ϵ, making a random choice

2.3 The 10-armed Testbed

Example approach to solving the 10-armed bandit problem using a greedy approach and different ϵ-greedy approaches. The higher the variance the better the ϵ-greedy methods perform. If the variance of the reward distribution is low, or 0, the more greedy the approach the better.

2.4 Incremental Implementation

We define the incremental update formula for Qn as

$$Q_{n+1} = Q_n + \frac{1}{n}\left[R_n - Q_n\right] \tag{2.3}$$

Qn: estimate of the value from action a given $n-1$ experienced steps.

Rn - reward received after nth selection of action a.

$$NewEstimate = OldEstimate + StepSize \cdot (Target - OldEstimate)$$

2.5 Tracking a Nonstationary Problem

This approach is when dealing with rewards that are not constant but change over time. The equation from the incremental implentation is modified to be

$$Q_{n+1} \doteq Q_n + \alpha[R_n - Q_n] \tag{2.5}$$

$\alpha \in (0, 1]$: constant step size parameter.

Now Q_{n+1} can be represented as a weighted average of Q_1:

$$Q_{n+1} = (1-\alpha)^n Q_1 + \alpha \sum_{i=1}^{\infty} (1-\alpha)^{n-i} R_i \tag{2.6}$$

The weight of R_i declines the further in the past it is: $(1-\alpha)^{n-i}$.

2.6 Optimistic Initial Values

The initial values of Q largely influence the algorithms behavior and can help sway it in one or the other direction. For example, if when using the action-value averages we were to be overly optimistic and set Q's initial values to $+5$ (when the actual values are lower than that) the algorithm will do a lot of exploration in the beginnning, not being satisfied with the rewards it was receiving after picking actions. This exploration happens even if ε is set to 0 and the algorithm is greedy.

2.7 Upper-Confidence-Bound (UCB) Action Selection

Attempting to pick exploratory actions based on their potential to be more optimal than the current greedy option. One effective way of doing this is:

$$A_t = \arg\max_a \left[Q_t(a) + c\sqrt{\frac{\ln t}{N_t(a)}} \right] \tag{2.10}$$

N_t: the number of times action a has been picked prior to timestep t.

$\ln t$: the natural algorithm of t.

c: the degree of exploration.

The square root expression represents the level of uncertainty or variance in the estimate for action a.

2.8 Gradient Bandit Algorithms

Calculating a numerical preference for action a denoted with $H_t(a)$. The probabilities only depend on other actions, not rewards.

$$Pr\{A_t = a\} \doteq \frac{e^{H_t(a)}}{\sum_{b=1}^{k} e^{H_t(b)}} \doteq \pi_t(a) \tag{2.11}$$

$\pi_t(a)$: the probability of taking action a at timestep t.

$\forall a, H_1(a) = 0$.

$\alpha > 0$: step size parameter.

After every step the preferences are updated the following way:

$$\begin{aligned} H_{t+1}(a) &\doteq H_t(a) + \alpha(R_t - \bar{R}_t)(1 - \pi_t(a)), \quad a = A_t \\ H_{t+1}(a) &\doteq H_t(a) - \alpha(R_t - \bar{R}_t)\pi_t(a), \qquad\quad a \neq A_t \end{aligned} \tag{2.12}$$

$\bar{R}_t$: the average of all the rewards up to and including timestep t.

The $\bar{R}_t$ is used as a baseline for the current R. If the current reward is higher than the baseline then the probability increases and vice versa. The probabilities for the non-selected actions move in the opposite direction.

2.9 Associative Search (Contextual Bandits)

In this case the optimal action is based on the state the environment is in. Therefore, we learn a policy which actively maps state to optimal action.

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 2.1 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 algorithms performance; all the algorithms perform best at an intermediate value of their parameter, neither too large nor too small.

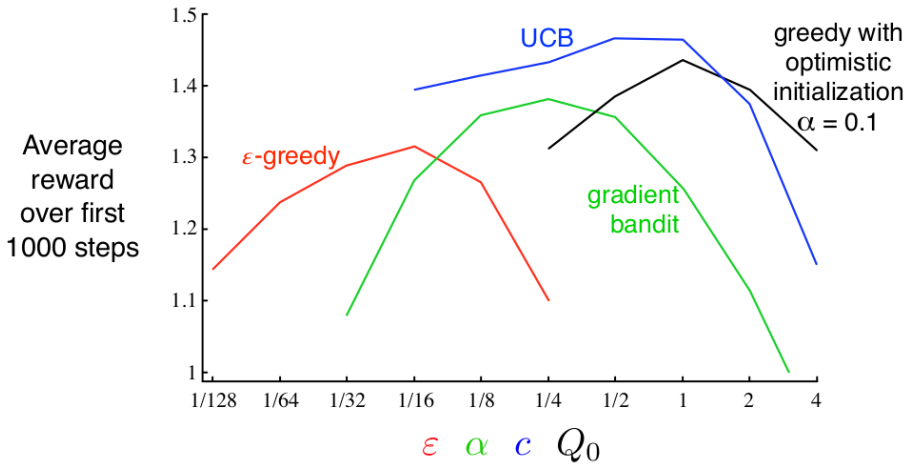


Figure 2.1: Parameter study of the various bandit algorithms presented in this chapter

In assessing 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, and 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.

Chapter 3

Finite Markov Decision Processes

The problem of finite MDPs include evaluative feedback as seen with bandits, but it also includes the associative aspect of linking states to actions. We estimate $q_*(s, a)$ and $v_*(s)$, as the expected reward from taking action a in state s and the maximum possible value from state s respectively.

3.1 The Agent-Environment Interface

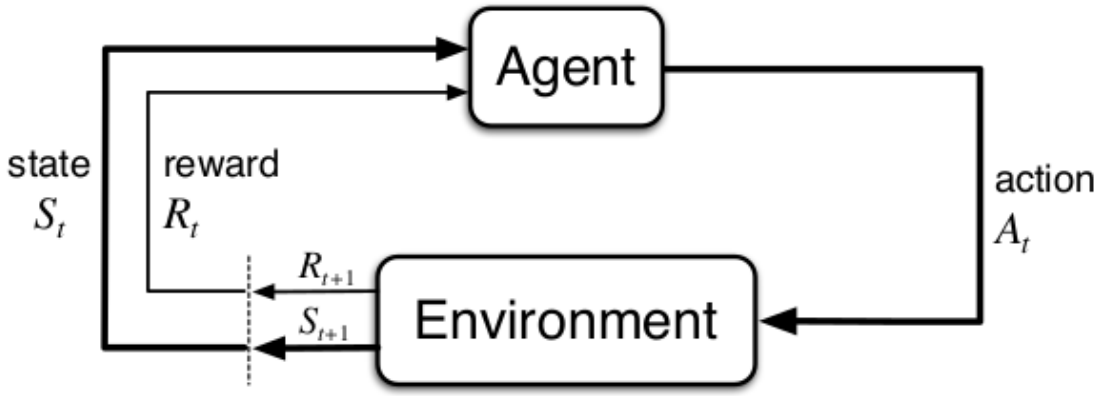


Figure 3.1: The agent-environment interaction in MPDs.

At each time step t the agent receives a state s and then based on its policy takes action a on the environment. Based on this action the agent receives a numerical reward from the environment. With finite MDPs the sets of $\mathcal{S}$, $\mathcal{A}$ and $\mathcal{R}$ are finite.

$$p(s', r \mid s, a) \doteq Pr\{S_t = s', R_t = r \mid S_{t-1} = s, A_{t-1} = a\} \tag{3.2}$$

$$\sum_{s'} \sum_{r} p(s', r \mid s, a) = 1, \forall s \in \mathcal{S}, a \in \mathcal{A}(s) \tag{3.3}$$

The state must have the *Markov Property* which entails that it holds all knowledge about the past agent-environment interaction that has influence on future rewards,

states.

$$p(s' \mid s, a) \doteq Pr\{S_t = s' \mid S_{t-1} = s, A_{t-1} = a\} = \sum_r p(s', r \mid s, a) \tag{3.4}$$

$$r(s, a) \doteq \mathbb{E}\left[R_t \mid S_{t-1} = s, A_{t-1} = a\right] = \sum_r r \sum_{s'} p(s', r \mid s, a) \tag{3.5}$$

$$r(s, a, s') \doteq \mathbb{E}\left[R_t \mid S_{t-1} = s, A_{t-1} = a, St = s'\right] = \sum_r r \cdot p(s', r \mid s, a) \tag{3.6}$$

3.2 Goals and Rewards

The reward signal is your way of communicating to the agent what you want it to achieve, not how you want it achieved.

3.3 Returns and Episodes

We define return after step t as

$$G_t \doteq R_{t+1} + R_{t+2} + ... + R_T = \sum_{k=0}^{T} R_{t+k+1} \tag{3.7}$$

T: the terminal state.

R_t: reward at time t.

We define discounted return for non episodic tasks as:

$$G_t \doteq \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{3.8}$$

$$G_t \doteq R_{t+1} + \gamma G_{t+1} \tag{3.9}$$

$\gamma \in [0, 1]$: the discount rate.

If the reward is a constant +1:

$$G_t \doteq \sum_{k=0}^{\infty} \gamma^k = \frac{1}{1-\gamma} \tag{3.10}$$

3.4 Policies and Value Functions

Value functions show how good a state is, or how good of a move would be to take an action a at state s. *Policy* is a mapping from state s to probabilities of taking actions in $\mathcal{A}$.

For MDPs we define v_π as the *state-value function* for policy π:

$$\begin{aligned} v_\pi(s) &\doteq \mathbb{E}_\pi \left[G_t \mid S_t = s \right] \\ &= \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s \right], \forall s \in \mathcal{S} \end{aligned} \tag{3.12}$$

And $q_\pi(s, a)$ as the *action-value function* for policy π:

$$\begin{aligned} q_\pi(s, a) &\doteq \mathbb{E}_\pi \left[G_t \mid S_t = s, A_t = a \right] \\ &= \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s, A_t = a \right] \end{aligned} \tag{3.13}$$

Monte Carlo methods are RL methods that involve averaging over many random samples for the actual returns.

$$\begin{aligned} v_\pi(s) &\doteq \mathbb{E}_\pi \left[G_t \mid S_t = s \right] \\ &= \mathbb{E}_\pi \left[R_{t+1} + \gamma G_{t+1} \mid S_t = s \right] \\ &= \sum_a \pi(a \mid s) \sum_{s'} \sum_r p(s', r \mid s, a) \left[r + \gamma \mathbb{E}_\pi \left[G_{t+1} \mid S_{t+1} = s' \right] \right] \\ &= \sum_a \pi(a \mid s) \sum_{s', r} p(s', r \mid s, a) \left[r + \gamma v_\pi(s') \right], \forall s \in \mathcal{S} \end{aligned} \tag{3.14}$$

This last equation is the so-called *Bellman equation* for v_π and is essentially weighing the values in the brackets on the right given the probability $\pi(a \mid s)p(s', r \mid s, a)$, and iterating through all possibilities.

3.5 Optimal Policies and Optimal Value Functions

A policy π' is better than policy π if $\forall s \in \mathcal{S}, v_{\pi'} \geq v_\pi$. In other words

$$\pi' \geq \pi \iff (\forall s \in \mathcal{S})(v_{\pi'} \geq v_\pi)$$

Optimal policy:

$$\pi_* \doteq \max_{v_\pi(s)} \pi, \forall s \in \mathcal{S} \tag{3.60*}$$

Optimal state-value function:

$$v_*(s) \doteq \max_\pi v_\pi(s), \forall s \in \mathcal{S} \tag{3.15}$$

Optimal action-value function:

$$q_*(s,a) \doteq \max_\pi q_\pi(a,s), \forall s \in \mathcal{S}, \forall a \in \mathcal{A} \tag{3.16}$$

$$q_*(s,a) = \mathbb{E}\left[R_{t+1} + \gamma v_*(S_{t+1}) \in S_t = s, A_t = a\right] \tag{3.17}$$

Bellman optimality equation:

$$\begin{aligned} v_*(s) &= \max_{a \in \mathcal{A}(s)} q_{\pi_*}(s,a) \\ &= \max_a \mathbb{E}_{\pi_*}\left[G_t \mid S_t = s, A_t = a\right] \\ &= \max_a \mathbb{E}\left[R_{t+1} + \gamma v_*(S_{t+1}) \mid S_t = s, A_t = a\right] \\ &= \max_a \sum_{s',r} p(s', r \mid s, a)[r + \gamma v_*(s')] \end{aligned} \tag{3.18}$$

$$\begin{aligned} q_*(s,a) &= \mathbb{E}\left[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a') \mid S_t = s, A_t = a\right] \\ &= \sum_{s',r} p(s', r \mid s, a)\left[r + \gamma \max_{a'} q_*(s', a')\right] \end{aligned} \tag{3.20}$$

Solving the *Bellman optimality equations* relies on having the following three properties that are rarely found in real-world scenarios:

- accurate knowledge of the dynamics of the environment.
- computational resources to complete the computation of the solution.
- the Markov property.

3.6 Optimality and Approximation

Tabular methods can store the optimal policy in a table but a lot of RL problems have a huge number of states that cannot be represented through a table due to memory restrictions. Therefore, we resolve to approximation methods.

3.7 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

- the rewards are the basis for evaluating the choices

Everything inside the agent is completely known and controllable by the agent; everything outside is incompletely controllable but 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 agents 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 policys value functions 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 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 environments 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 has a complete and accurate environment model, the agent is typically unable to perform enough computation per time step to fully use it. The memory available is also an important constraint. 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.

Chapter 4

Dynamic Programming

DP algorithms are obtained by turning the Bellman equations into assignments - update rules for improving approximations in the future.

4.1 Policy Evaluation (Prediction)

Policy evaluation is the process of determining $v_\pi(s), \forall s \in \mathcal{S}$. *Iterative policy evaluation* is the process of arbitrarily initializing the state-value function for all states and then using the Bellman equation as an iterative update rule.

Iterative Policy Evaluation, for estimating $V \approx v_\pi$

Input π, the policy to be evaluated
Algorithm parameter: a small threshold $\theta > 0$ determining accuracy of estimation
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(terminal) = 0$

Loop:
  $\Delta \leftarrow 0$
  Loop for each $s \in \mathcal{S}$:
    $v \leftarrow V(s)$
    $V(s) \leftarrow \sum_a \pi(a|s) \sum_{s',r} p(s', r | s, a)\big[r + \gamma V(s')\big]$
    $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$

4.2 Policy Improvement

If we find policy π' such that $q_{\pi'}(s, \pi'(s)) \geq v_\pi(s)$ then we update the policy π so that $\pi(s) == \pi'(s)$. At each step we select the action that appears best according to $q_\pi(s, a)$, the new *greedy* policy π' is defined as the following:

$$\begin{aligned} \pi'(s) &\doteq \arg\max_a q_\pi(s, a) \\ &= \arg\max_a \sum_{s',r} p(s', r \mid s, a)\,[r + \gamma v_\pi(s')] \end{aligned} \tag{4.9}$$

$\arg\max_a$: value of a for which the expression that follows is maximized.

Policy improvement is the process of making a new policy that improves the original policy by making it greedy with respect to the value function of the original policy.

4.3 Policy Iteration

Policy Iteration (using iterative policy evaluation) for estimating $\pi \approx \pi_*$

1. Initialization
   $V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$

2. Policy Evaluation
   Loop:
   $\Delta \leftarrow 0$
   Loop for each $s \in \mathcal{S}$:
   $v \leftarrow V(s)$
   $V(s) \leftarrow \sum_{s',r} p(s', r \,|\, s, \pi(s)) \big[r + \gamma V(s')\big]$
   $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
   until $\Delta < \theta$ (a small positive number determining the accuracy of estimation)

3. Policy Improvement
   policy-stable $\leftarrow$ *true*
   For each $s \in \mathcal{S}$:
   old-action $\leftarrow \pi(s)$
   $\pi(s) \leftarrow \text{argmax}_a \sum_{s',r} p(s', r \,|\, s, a) \big[r + \gamma V(s')\big]$
   If *old-action* $\neq \pi(s)$, then *policy-stable* $\leftarrow$ *false*
   If *policy-stable*, then stop and return $V \approx v_*$ and $\pi \approx \pi_*$; else go to 2

4.4 Value Iteration

We are truncating *policy evaluation* to get faster computation but still ensure convergence. Value iteration is truncating *policy evaluation* to exactly one step.

$$\begin{aligned} v_{k+1}(s) &\doteq \max_a \mathbb{E}\left[R_{t+1} + \gamma v_k(S_{t+1}) \mid S_t = s, A_t = a\right] \\ &= \max_a \sum_{s',r} p(s', r \mid s, a)\left[r + \gamma v_k(s')\right] \end{aligned} \tag{4.10}$$

Value Iteration, for estimating $\pi \approx \pi_*$

Algorithm parameter: a small threshold $\theta > 0$ determining accuracy of estimation
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(terminal) = 0$

Loop:
| $\Delta \leftarrow 0$
| Loop for each $s \in \mathcal{S}$:
| $\quad v \leftarrow V(s)$
| $\quad V(s) \leftarrow \max_a \sum_{s',r} p(s', r \mid s, a)\big[r + \gamma V(s')\big]$
| $\quad \Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$

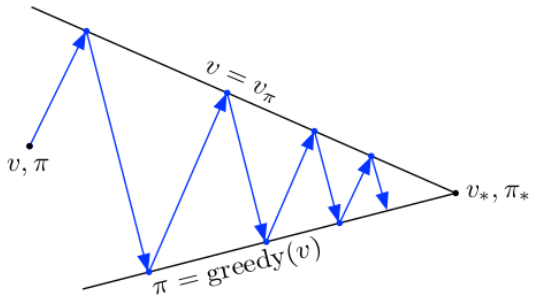
Output a deterministic policy, $\pi \approx \pi_*$, such that
$\pi(s) = \arg\max_a \sum_{s',r} p(s', r \mid s, a)\big[r + \gamma V(s')\big]$

4.5 Asynchronous Dynamic Programming

These algorithms don't update all state values in each sweep but only target some. This targeting can be done using the amount of update done in a sweep on a state, or given preset priorities. They also allow for real-time interaction - while an agent is playing the algorithm is running and learning.

4.6 Generalized Policy Iteration (GPI)

Generalized Policy Iteration (GPI) refers to a branch of algorithms that allow for *policy evaluation* and *policy improvement* to intertwine.



4.7 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 functions 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.

Chapter 5

Monte Carlo (MC) Methods

Unlike the previous chapters here we only require experience, and not a complete knowledge of the environment. The term *Monte Carlo* is often used in a more broad sense for any estimation method that invlolves a significant random component. We are learning the value function based on the sample returns from the MDP. The idea of gaining experience and averaging returns from each state to estimate the value functions is an essential idea of all MC methods. In comparison to DP algorithms, that are breadth-first we can view MCMs as depth-first algorithms. MCMs do not bootstrap - an estimation of the value of one state is not based on estimations of value functions for other states.

5.1 Monte Carlo Prediction

First-visit MC methods average returns following the first visit to state s. *Every-visit MC methods* average returns following every visit to state s.

First-visit MC prediction, for estimating $V \approx v_\pi$

Input: a policy π to be evaluated
Initialize:
    $V(s) \in \mathbb{R}$, arbitrarily, for all $s \in \mathcal{S}$
    $Returns(s) \leftarrow$ an empty list, for all $s \in \mathcal{S}$

Loop forever (for each episode):
    Generate an episode following π: $S_0, A_0, R_1, S_1, A_1, R_2, \ldots, S_{T-1}, A_{T-1}, R_T$
    $G \leftarrow 0$
    Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:
        $G \leftarrow \gamma G + R_{t+1}$
        Unless S_t appears in $S_0, S_1, \ldots, S_{t-1}$:
            Append G to $Returns(S_t)$
            $V(S_t) \leftarrow$ average$(Returns(S_t))$

5.2 Monte Carlo Estimation of Action Values

The problem of lack of exploration. Possible solution with exploring starts, even though this can be used only under specific circumstances.

5.3 Monte Carlo Control

Monte Carlo ES (Exploring Starts), for estimating $\pi \approx \pi_*$

Initialize:
$\pi(s) \in \mathcal{A}(s)$ (arbitrarily), for all $s \in \mathcal{S}$
$Q(s,a) \in \mathbb{R}$ (arbitrarily), for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$
$Returns(s,a) \leftarrow$ empty list, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$

Loop forever (for each episode):
Choose $S_0 \in \mathcal{S}$, $A_0 \in \mathcal{A}(S_0)$ randomly such that all pairs have probability > 0
Generate an episode from S_0, A_0, following π: $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$
$G \leftarrow 0$
Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:
$G \leftarrow \gamma G + R_{t+1}$
Unless the pair S_t, A_t appears in $S_0, A_0, S_1, A_1 \ldots, S_{t-1}, A_{t-1}$:
Append G to $Returns(S_t, A_t)$
$Q(S_t, A_t) \leftarrow \text{average}(Returns(S_t, A_t))$
$\pi(S_t) \leftarrow \arg\max_a Q(S_t, a)$

We can modify the algorithm to only store the mean and the count in *Returns*, rather than the whole list of returns. This way we optimize in memory consumption and remove the time needed to calculate the average when storing $Q(S_t, A_t)$.

5.4 Monte Carlo Control without Exploring Starts

On-policy methods evaluate or improve the policy used to make decisions, whereas *off-policy methods* evaluate or improve a policy different from the one used to make decisions.

On-policy first-visit MC control (for ε-soft policies), estimates $\pi \approx \pi_*$

Algorithm parameter: small $\varepsilon > 0$
Initialize:
$\pi \leftarrow$ an arbitrary ε-soft policy
$Q(s,a) \in \mathbb{R}$ (arbitrarily), for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$
$Returns(s,a) \leftarrow$ empty list, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$

Repeat forever (for each episode):
Generate an episode following π: $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$
$G \leftarrow 0$
Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:
$G \leftarrow \gamma G + R_{t+1}$
Unless the pair S_t, A_t appears in $S_0, A_0, S_1, A_1 \ldots, S_{t-1}, A_{t-1}$:
Append G to $Returns(S_t, A_t)$
$Q(S_t, A_t) \leftarrow \text{average}(Returns(S_t, A_t))$
$A^* \leftarrow \arg\max_a Q(S_t, a)$ (with ties broken arbitrarily)
For all $a \in \mathcal{A}(S_t)$:

$$\pi(a|S_t) \leftarrow \begin{cases} 1 - \varepsilon + \varepsilon/|\mathcal{A}(S_t)| & \text{if } a = A^* \\ \varepsilon/|\mathcal{A}(S_t)| & \text{if } a \neq A^* \end{cases}$$

5.5 Off-policy Prediction via Importance Sampling

Learn the optimal policy while behaving according to an exploratory policy. The policy being learned about is the *target policy*, and the policy used to generate behavior is called the *behavior policy*. *Off-policy* methods are usually of greater variance and slower to converge. If b is the behavior policy and π the target policy then $\pi(a \mid s) > 0 \rightarrow b(a \mid s) > 0$ - this is called the assumption of *coverage*. Meaning that every state-action for which π has a non-zero value (covers it), policy b also has a non-zero value, and technically will eventually cover it too.
Importance sampling ratio given a state-action trajectory:

$$\rho_{t:T-1} \doteq \prod_{k=t}^{T-1} \frac{\pi(A_k \mid S_k)}{b(A_k \mid S_k)} \tag{5.3}$$

We use the importance sampling ratio to transform the estimated $v_b(s)$ to $v_\pi(s)$:

$$v_\pi(s) = \mathbb{E}\left[\rho_{t:T-1} \cdot G_t \mid S = s\right] \tag{5.4}$$

Here the timesteps will surpass episode boundaries. If we define $\mathcal{T}(s)$ as a set of all time steps where s was encountered; only the first times per episode for first-visit methods. Let $T(t)$ denote the first time of termination after t, and G_t denote the return after t following up to $T(t)$, then we can estimate $v_\pi(s)$ as:

$$V(s) \doteq \frac{\sum_{t \in \mathcal{T}(s)} \rho_{t:T(t)-1} G_t}{|\mathcal{T}(s)|} \tag{5.5}$$

When *importance sampling* is done as a simple average in this way it's called *ordinary importance sampling*. An important alternative to that is *weighted importance sampling* that uses weighted averages:

$$V(s) \doteq \frac{\sum_{t \in \mathcal{T}(s)} \rho_{t:T(t)-1} G_t}{\sum_{t \in \mathcal{T}(s)} \rho_{t:T(t)-1}} \tag{5.6}$$

5.6 Incremental Implementation

Off-policy MC prediction (policy evaluation) for estimating $Q \approx q_\pi$

Input: an arbitrary target policy π
Initialize, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$:
    $Q(s,a) \in \mathbb{R}$ (arbitrarily)
    $C(s,a) \leftarrow 0$

Loop forever (for each episode):
    $b \leftarrow$ any policy with coverage of π
    Generate an episode following b: $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$
    $G \leftarrow 0$
    $W \leftarrow 1$
    Loop for each step of episode, $t = T-1, T-2, \ldots, 0$, while $W \neq 0$:
        $G \leftarrow \gamma G + R_{t+1}$
        $C(S_t, A_t) \leftarrow C(S_t, A_t) + W$
        $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{W}{C(S_t, A_t)} [G - Q(S_t, A_t)]$
        $W \leftarrow W \frac{\pi(A_t|S_t)}{b(A_t|S_t)}$

$C(s,a)$: cumulative sum of the weights.

5.7 Off-policy Monte Carlo Control

Off-policy MC control, for estimating $\pi \approx \pi_*$

Initialize, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$:
    $Q(s,a) \in \mathbb{R}$ (arbitrarily)
    $C(s,a) \leftarrow 0$
    $\pi(s) \leftarrow \arg\max_a Q(s,a)$ (with ties broken consistently)

Loop forever (for each episode):
    $b \leftarrow$ any soft policy
    Generate an episode using b: $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$
    $G \leftarrow 0$
    $W \leftarrow 1$
    Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:
        $G \leftarrow \gamma G + R_{t+1}$
        $C(S_t, A_t) \leftarrow C(S_t, A_t) + W$
        $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{W}{C(S_t, A_t)} [G - Q(S_t, A_t)]$
        $\pi(S_t) \leftarrow \arg\max_a Q(S_t, a)$ (with ties broken consistently)
        If $A_t \neq \pi(S_t)$ then exit inner Loop (proceed to next episode)
        $W \leftarrow W \frac{1}{b(A_t|S_t)}$

5.8 *Discounting-aware Importance Sampling

Importance sampling (ρ) takes into account all timesteps leading up to T. What it should preferably be doing is looking at the discount to G which is γ and using that to discount the ρ. We can look into γ as partially terminating G and then define *flat partial returns* ($\bar{G}_{t:h}$) as:

$$\bar{G} \doteq R_{t+1} + R_{t+2} + \cdots + R_h, \qquad 0 \leq t \leq h \leq T$$

They're called *flat* because there's no discounting, and *partial* because rather than going to T they end at h, called the *horizon*.

$$G_t \doteq (1-\gamma) \sum_{h=t+1}^{T-1} \gamma^{h-t-1} \bar{G}_{t:h} + \gamma^{T-t-1} \bar{G}_{t:T}$$

Given this, (5.5) and (5.6) become discount aware importance sampling estimators:

$$V(s) \doteq \frac{\sum_{t \in \mathcal{T}(s)} \left((1-\gamma) \sum_{h=t+1}^{T(t)-1} \gamma^{h-t-1} \rho_{t:h-1} \bar{G}_{t:h} + \gamma^{T(t)-t-1} \rho_{t:T(t)-1} \bar{G}_{t:T(t)} \right)}{|\mathcal{T}(s)|} \tag{5.9}$$

$$V(s) \doteq \frac{\sum_{t \in \mathcal{T}(s)} \left((1-\gamma) \sum_{h=t+1}^{T(t)-1} \gamma^{h-t-1} \rho_{t:h-1} \bar{G}_{t:h} + \gamma^{T(t)-t-1} \rho_{t:T(t)-1} \bar{G}_{t:T(t)} \right)}{\sum_{t \in \mathcal{T}(s)} \left((1-\gamma) \sum_{h=t+1}^{T(t)-1} \gamma^{h-t-1} \rho_{t:h-1} + \gamma^{T(t)-t-1} \rho_{t:T(t)-1} \right)} \tag{5.10}$$

5.9 *Per-decision Importance Sampling

To further minimize the variance, we can multiply a given R at time t with just the $\rho_{t0:t-1}$ because those after do not influence the reward.

$$\mathbb{E}\left[\rho_{t:T-1} R_{t+k}\right] = \mathbb{E}\left[\rho_{t:t-k-1} R_{t+k}\right] \Rightarrow \mathbb{E}\left[\rho_{t:T-1} G_t\right] = \mathbb{E}\left[\tilde{G}_t\right]$$

$$\tilde{G}_t = \rho_{t:t} R_{t+1} + \gamma \rho_{t:t+1} R_{t+2} + \cdots + \gamma^{k-1} \rho_{t:t+k-1} R_{t+k} + \cdots + \gamma^{T-1} \rho_{t:T-1} R_T$$

Now (5.5) becomes:

$$V(s) = \frac{\sum_{t \in \mathcal{T}(s)} \tilde{G}_t}{|\mathcal{T}(s)|} \tag{5.15}$$

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.

1. They can be used to learn optimal behavior directly from interaction with the environment, with no model of the environments dynamics.

2. 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.

3. 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).

4. 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 *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 states 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 environments 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 stateaction 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.

1. They operate on sample experience, and thus can be used for direct learning without a model.

2. 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.

Chapter 6

Temporal-Difference Learning

Temporal-difference learning is a combination of *MC* and *DP* methods, it generates experience from which it learns, like *MC*, and it bootstraps like *DP*.

6.1 TD Prediction

Similar to how *MC* methdods predict but a stop is made right after the first step and then a approximated value is used. Basically, the target is $R_{t+1} + \gamma V(S_{t+1})$ instead of G_t.

$$V(S_t) \leftarrow V(S_t) + \alpha \left[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)\right] \tag{6.2}$$

This is called $TD(0)$ or *one-step TD*.

Tabular TD(0) for estimating v_π

Input: the policy π to be evaluated
Algorithm parameter: step size $\alpha \in (0, 1]$
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(terminal) = 0$

Loop for each episode:
  Initialize S
  Loop for each step of episode:
    $A \leftarrow$ action given by π for S
    Take action A, observe R, S'
    $V(S) \leftarrow V(S) + \alpha\big[R + \gamma V(S') - V(S)\big]$
    $S \leftarrow S'$
  until S is terminal

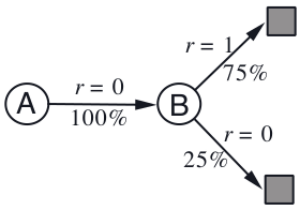
We can notice that the value in the brackets in (6.2) is a sort of error. That error is called *TD error* (δ) and it is the difference from the estimated value of S_t and the better estimate after the next step $R_{t+1} + \gamma V(S_{t+1})$.

$$\delta_t \doteq R_{t+1} + \gamma V(S_{t+1}) - V(S_t) \tag{6.5}$$

6.2 Optimality of TD(0)

Consider an example where we use *batch updating*. *TD(0)* and constant α *MCM* converge to different values. Observe the following episodes:

1. $A, 0, B, 0$
2. $B, 1$
3. $B, 1$
4. $B, 1$
5. $B, 1$
6. $B, 1$
7. $B, 1$
8. $B, 0$



Both batch TD(0) and batch MC would conclude that $V(B) = \frac{3}{4}$ but TD(0) would approximate $V(A) = V(B) = \frac{3}{4}$ whereas MC would approximate $V(A) = 0$ because the one episode we go through the state we get reward 0.

Batch MC methods always find the estimates that minimize mean-squared error on the training set, whereas batch TD(0) always finds the estimates that would be exactly correct for the maximum-likelihood model of the Markov process. In general, the *maximum-likelihood estimate* of a parameter is the parameter value whose probability of generating the data is greatest. In this case, the maximum-likelihood estimate is the model of the Markov process formed in the obvious way from the observed episodes: the estimated transition probability from i to j is the fraction of observed transitions from i that went to j, and the associated expected reward is the average of the rewards observed on those transitions. Given this model, we can compute the estimate of the value function that would be exactly correct if the model were exactly correct. This is called the *certainty-equivalence estimate* because it is equivalent to assuming that the estimate of the underlying process was known with certainty rather than being approximated.

6.3 Sarsa: On-policy TD Control

Instead of state values we use action values:

$$\begin{aligned} \delta_t &\doteq R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t) \\ Q(S_t, A_t) &\leftarrow Q(S_t, A_t) + \alpha \cdot \delta_t \end{aligned} \tag{6.7}$$

Sarsa (on-policy TD control) for estimating $Q \approx q_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(\mathit{terminal}, \cdot) = 0$

Loop for each episode:
  Initialize S
  Choose A from S using policy derived from Q (e.g., ε-greedy)
  Loop for each step of episode:
    Take action A, observe R, S'
    Choose A' from S' using policy derived from Q (e.g., ε-greedy)
    $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma Q(S', A') - Q(S, A)\big]$
    $S \leftarrow S'$; $A \leftarrow A'$;
  until S is terminal

6.4 Q-Learning: Off-policy TD Control

Instead of using $Q(S_{t+1}, A_{t+1})$ for action A_{t+1} taken by the *behavior policy* we use $\max_a Q(S_{t+1}, a)$.

$$\begin{aligned} \delta_t &\doteq R_{t+1} + \max_a Q(S_{t+1}, a) - Q(S_t, A_t) \\ Q(S_t, A_t) &\leftarrow Q(S_t, A_t) + \alpha \cdot \delta_t \end{aligned} \tag{6.8}$$

All that is required for correct convergence is that all pairs continue to be updated.

Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(\mathit{terminal}, \cdot) = 0$

Loop for each episode:
  Initialize S
  Loop for each step of episode:
    Choose A from S using policy derived from Q (e.g., ε-greedy)
    Take action A, observe R, S'
    $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma \max_a Q(S', a) - Q(S, A)\big]$
    $S \leftarrow S'$
  until S is terminal

6.5 Expected SARSA

$$\begin{aligned} \delta_t &\doteq R_{t+1} + \gamma \mathbb{E}\left[Q(S_{t+1}, A_{t+1} \mid S_{t+1})\right] - Q(S_t, A_t) \\ &\doteq R_{t+1} + \gamma \sum_a \pi(a \mid S_{t+1}) Q(S_{t+1}, a) - Q(S_t, A_t) \\ Q(S_t, A_t) &\leftarrow Q(S_t, A_t) + \alpha \cdot \delta_t \end{aligned} \tag{6.9}$$

6.6 Maximization Bias and Double Learning

A maximum over estimated values is used implicitly as an estimate of the maximum value, which can lead to a significant positive bias. To see why, consider a single state s where there are many actions a whose true values, $q(s,a)$, are all zero but whose estimated values, $Q(s,a)$, are uncertain and thus distributed some above and some below zero. The maximum of the true values is zero, but the maximum of the estimates is positive - a positive bias. We call this *maximization bias*.

The reason this bias occurs is because we are using the same state-action value function to approximate $q(s,a)$ and then using the maximum of the estimation of $q(s,a)$ to estimate the maximum of $q(s,a)$ - the optimal action. To attempt and mitigate this issue, we can learn two independent estimates of the state-action value (Q_1, Q_2) and then using them generate an unbiased estimate of the maximum of $q(s,a)$. We take the value of the next state-action pair given one of the estimators for the action and update of value for the other estimator.

$$\begin{aligned} \delta_t &\doteq R_{t+1} + \gamma Q_2(S_t, \arg\max_a Q_1(S_{t+1}, a)) - Q_1(S_t, A_t) \\ Q_1(S_t, A_t) &\leftarrow Q_1(S_t, A_t) + \alpha\delta_t \end{aligned} \tag{6.10}$$

Double Q-learning, for estimating $Q_1 \approx Q_2 \approx q_*$

Algorithm parameters: step size $\alpha \in (0,1]$, small $\varepsilon > 0$
Initialize $Q_1(s,a)$ and $Q_2(s,a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, such that $Q(\textit{terminal}, \cdot) = 0$

Loop for each episode:
  Initialize S
  Loop for each step of episode:
    Choose A from S using the policy ε-greedy in $Q_1 + Q_2$
    Take action A, observe R, S'
    With 0.5 probabilility:
      $Q_1(S,A) \leftarrow Q_1(S,A) + \alpha\Big(R + \gamma Q_2\big(S', \arg\max_a Q_1(S',a)\big) - Q_1(S,A)\Big)$
    else:
      $Q_2(S,A) \leftarrow Q_2(S,A) + \alpha\Big(R + \gamma Q_1\big(S', \arg\max_a Q_2(S',a)\big) - Q_2(S,A)\Big)$
    $S \leftarrow S'$
  until S is terminal

6.7 Games, Afterstates, and Other Special Cases

A conventional state-value function evaluates states in which the agent has the option of selecting an action, but the state-value function used in tic-tac-toe evaluates board positions after the agent has made its move. Let us call these *afterstates*, and value functions over these, *afterstate value functions*.

A conventional action-value function would have to separately assess both pairs, whereas an afterstate value function would immediately assess both equally.

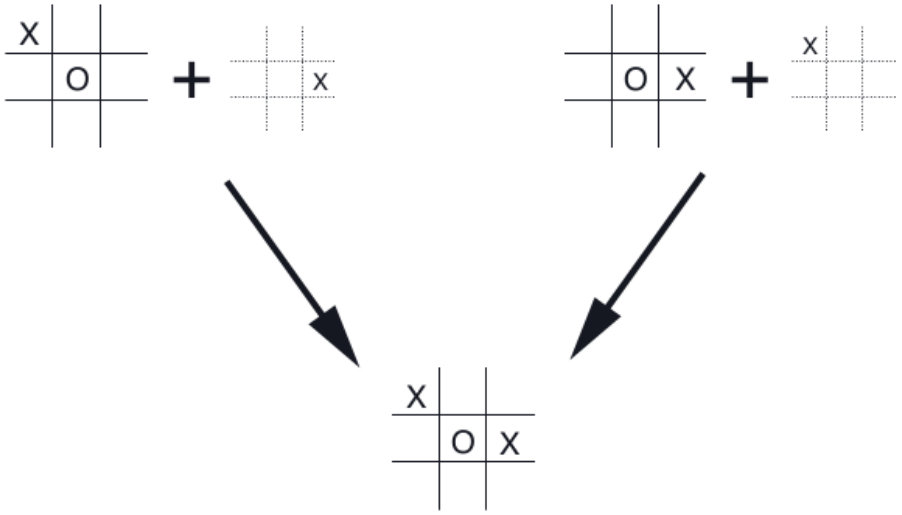


Figure 6.1: Two different state-action pairs that result in the same afterstate and therefore should have the same Q.

6.8 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 *MCM* for solving the prediction problem. In both cases, the extension to the control problem is via the idea of *GPI* that we abstracted from *DP*. 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 actorcritic 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 rela-

tively 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 MCM) and forms that include a model of the environment (a link to planning and DP). 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.

Chapter 7

n-step Bootstrapping

Preferably *bootstrapping* should be done over multiple steps, and that is what these methods allow us to do. The idea of n-step methods is usually used as an introduction to the algorithmic idea of *eligibility traces* (Chapter 12), which enable bootstrapping over multiple time intervals simultaneously.

7.1 n-step TD Prediction

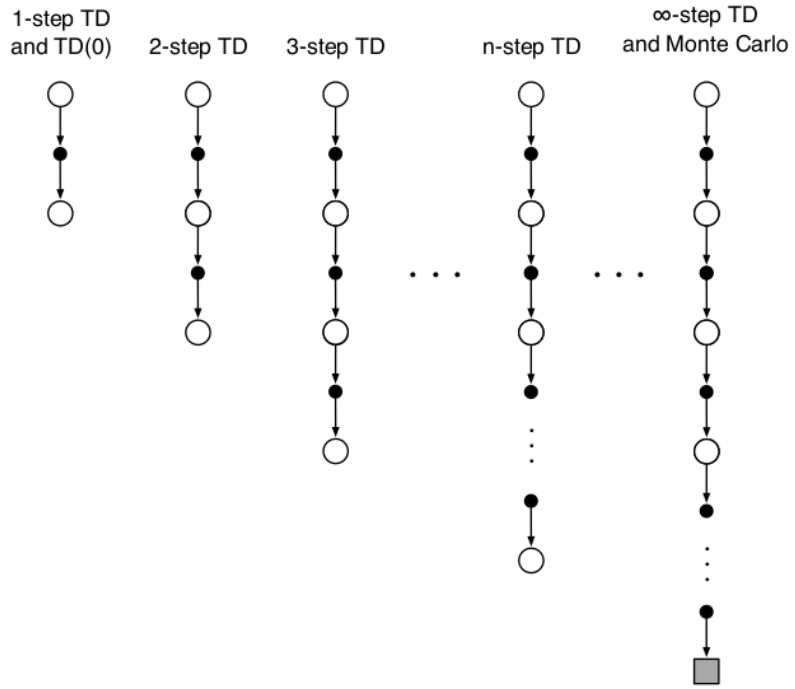


Figure 7.1: The backup diagrams of n-step methods. These methods form a spectrum ranging from one-step TD methods to Monte Carlo methods.

$$G_{t:t+n} \doteq R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n V_{t+n+1}(S_{t+n}) \tag{7.1}$$

$$V_{t+n}(S_t) \doteq V_{t+n-1} + \alpha \left(G_{t:t+n} - V_{t+n-1}(S_t) \right) \tag{7.2}$$

n-step TD for estimating $V \approx v_\pi$

Input: a policy π
Algorithm parameters: step size $\alpha \in (0, 1]$, a positive integer n
Initialize $V(s)$ arbitrarily, for all $s \in \mathcal{S}$
All store and access operations (for S_t and R_t) can take their index mod $n + 1$

Loop for each episode:
  Initialize and store $S_0 \neq$ terminal
  $T \leftarrow \infty$
  Loop for $t = 0, 1, 2, \ldots$:
  | If $t < T$, then:
  | Take an action according to $\pi(\cdot|S_t)$
  | Observe and store the next reward as R_{t+1} and the next state as S_{t+1}
  | If S_{t+1} is terminal, then $T \leftarrow t + 1$
  | $\tau \leftarrow t - n + 1$ (τ is the time whose state's estimate is being updated)
  | If $\tau \geq 0$:
  | $G \leftarrow \sum_{i=\tau+1}^{\min(\tau+n,T)} \gamma^{i-\tau-1} R_i$
  | If $\tau + n < T$, then: $G \leftarrow G + \gamma^n V(S_{\tau+n})$ $(G_{\tau:\tau+n})$
  | $V(S_\tau) \leftarrow V(S_\tau) + \alpha \left[G - V(S_\tau) \right]$
  Until $\tau = T - 1$

7.2 n-step Sarsa

$$\begin{aligned} G_{t:t+n} &\doteq \sum_{k=1}^{n} \gamma^{k-1} R_{t+k} + \gamma^n Q_{t+n-1}(S_{t+n}, A_{t+n}) \\ Q_{t+n}(S_t, A_t) &\doteq Q_{t+n-1}(S_t, A_t) + \alpha \left[G_{t:t+n} - Q_{t+n-1}(S_t, A_t) \right] \end{aligned} \tag{7.5}$$

n-step Sarsa for estimating $Q \approx q_*$ or q_π

Initialize $Q(s,a)$ arbitrarily, for all $s \in \mathcal{S}, a \in \mathcal{A}$
Initialize π to be ε-greedy with respect to Q, or to a fixed given policy
Algorithm parameters: step size $\alpha \in (0,1]$, small $\varepsilon > 0$, a positive integer n
All store and access operations (for S_t, A_t, and R_t) can take their index mod $n+1$

Loop for each episode:
  Initialize and store $S_0 \neq$ terminal
  Select and store an action $A_0 \sim \pi(\cdot|S_0)$
  $T \leftarrow \infty$
  Loop for $t = 0, 1, 2, \ldots$:
  | If $t < T$, then:
  | Take action A_t
  | Observe and store the next reward as R_{t+1} and the next state as S_{t+1}
  | If S_{t+1} is terminal, then:
  | $T \leftarrow t+1$
  | else:
  | Select and store an action $A_{t+1} \sim \pi(\cdot|S_{t+1})$
  | $\tau \leftarrow t-n+1$ (τ is the time whose estimate is being updated)
  | If $\tau \geq 0$:
  | $G \leftarrow \sum_{i=\tau+1}^{\min(\tau+n,T)} \gamma^{i-\tau-1} R_i$
  | If $\tau + n < T$, then $G \leftarrow G + \gamma^n Q(S_{\tau+n}, A_{\tau+n})$ $(G_{\tau:\tau+n})$
  | $Q(S_\tau, A_\tau) \leftarrow Q(S_\tau, A_\tau) + \alpha\left[G - Q(S_\tau, A_\tau)\right]$
  | If π is being learned, then ensure that $\pi(\cdot|S_\tau)$ is ε-greedy wrt Q
  Until $\tau = T-1$

7.3 n-step Off-policy Learning

Other than the α, the difference of gain and current value function value is weighted by $\rho_{t:t+n-1}$.

$$V_{t+n}(S_t) \doteq V_{t+n-1}(S_t) + \alpha\rho_{t:t+n-1}\left[G_{t:t+n} - V_{t+n-1}(S_t)\right] \tag{7.9}$$

$$\rho_{t:h} \doteq \sum_{k=t}^{min(h,T-1)} \frac{\pi(A_k \mid S_k)}{b(A_k \mid S_k)} \tag{7.10}$$

$$Q_{t+n}(S_t, A_t) \doteq Q_{t+n-1}(S_t, A_t) + \alpha\rho_{t+1:t+n}\left[G_{t:t+n} - Q_{t+n-1}(S_t, A_t)\right] \tag{7.11}$$

Note that here the *importance sampling ratio* (ρ) starts and ends one step later in comparison to the state-value function. This is because we are not bothered with the current action taken, but rather only looking at future actions.

Off-policy n-step Sarsa for estimating $Q \approx q_*$ or q_π

Input: an arbitrary behavior policy b such that $b(a|s) > 0$, for all $s \in \mathcal{S}, a \in \mathcal{A}$
Initialize $Q(s,a)$ arbitrarily, for all $s \in \mathcal{S}, a \in \mathcal{A}$
Initialize π to be greedy with respect to Q, or as a fixed given policy
Algorithm parameters: step size $\alpha \in (0,1]$, a positive integer n
All store and access operations (for S_t, A_t, and R_t) can take their index mod $n+1$

Loop for each episode:
  Initialize and store $S_0 \neq$ terminal
  Select and store an action $A_0 \sim b(\cdot|S_0)$
  $T \leftarrow \infty$
  Loop for $t = 0, 1, 2, \ldots$:
    If $t < T$, then:
      Take action A_t
      Observe and store the next reward as R_{t+1} and the next state as S_{t+1}
      If S_{t+1} is terminal, then:
        $T \leftarrow t+1$
      else:
        Select and store an action $A_{t+1} \sim b(\cdot|S_{t+1})$
    $\tau \leftarrow t - n + 1$ (τ is the time whose estimate is being updated)
    If $\tau \geq 0$:
      $\rho \leftarrow \prod_{i=\tau+1}^{\min(\tau+n-1, T-1)} \frac{\pi(A_i|S_i)}{b(A_i|S_i)}$ $(\rho_{\tau+1:t+n-1})$
      $G \leftarrow \sum_{i=\tau+1}^{\min(\tau+n, T)} \gamma^{i-\tau-1} R_i$
      If $\tau + n < T$, then: $G \leftarrow G + \gamma^n Q(S_{\tau+n}, A_{\tau+n})$ $(G_{\tau:\tau+n})$
      $Q(S_\tau, A_\tau) \leftarrow Q(S_\tau, A_\tau) + \alpha\rho\left[G - Q(S_\tau, A_\tau)\right]$
      If π is being learned, then ensure that $\pi(\cdot|S_\tau)$ is greedy wrt Q
  Until $\tau = T - 1$

7.4 *Per-decision Methods with Control Variates

This approach takes a similar path to that in section 5.9.

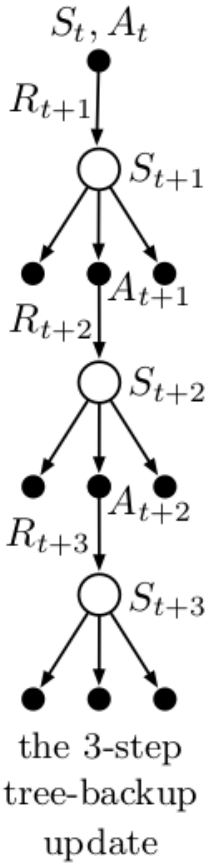
$$\begin{aligned} G_{t:h} &\doteq \rho_t(R_{t+1} + \gamma G_{t+1:h}) + (1-\rho_t)V_{h-1}(S_t) \\ \rho_t &\doteq \frac{\pi(A_t \mid S_t)}{b(A_t \mid S_t)} \qquad\qquad G_{h:h} \doteq V_{h-1}(S_h) \end{aligned} \tag{7.13}$$

The $(1-\rho_t)V_{h-1}(S_t)$ term is called the *control variate*. It is used to take the value of the resulting state as the approximation of the gain following it. This term has an effect on $G_{t:h}$ only when b takes actions that are improbable in π.

For a conventional n-step method, the learning rule to use in conjunction with (7.13) is the n-step TD update (7.2), which has no explicit importance sampling ratios other than those embedded in the return.

$$
\begin{aligned}
G_{t:h} &\doteq R_{t+1} + \gamma\rho_{t+1}(G_{t+1:h} - Q_{h-1}(S_{t+1}, A_{t+1})) + \gamma\bar{V}_{h-1}(S_t + 1) \\
\bar{V}_t(s) &\doteq \sum_a \pi(a \mid s) Q_t(s, a) \\
h < T &\Rightarrow G_{h:h} \doteq Q_{h-1}(S_h, A_h) \\
h \geq T &\Rightarrow G_{T-1:h} \doteq R_T
\end{aligned}
\tag{7.14}
$$

7.5 Off-policy Learning Without Importance Sampling: The n-step Tree Backup Algorithm



the 3-step tree-backup update

This algorithm takes into consideration actions not taken. We go down the taken action to calculate the G but add to it the possibility of taking and the value of actions **not taken**. The last node's gain is calculated just like for *expected sarsa*.

$$
\begin{aligned}
G_{t:t+1} &\doteq R_{t+1} + \gamma \sum_a \pi(a \mid S_{t+1}) Q_t(S_{t+1}, a) \\
G_{t:t+2} &\doteq R_{t+1} + \gamma \sum_{a \neq A_{t+1}} \pi(a \mid S_{t+1}) Q_{t+1}(S_{t+1}, a) \\
&\quad + \gamma\pi(A_{t+1} \mid S_{t+1}) G_{t+1:t+2}
\end{aligned}
\tag{7.15}
$$

Generalizing to:

$$
\begin{aligned}
G_{t:t+n} &\doteq R_{t+1} + \gamma \sum_{a \neq A_{t+1}} \pi(a \mid S_{t+1}) Q_{t+1}(S_{t+1}, a) \\
&\quad + \gamma\pi(A_{t+1} \mid S_{t+1}) G_{t+1:t+n} \\
G_{T-1:t+n} &\doteq R_T
\end{aligned}
\tag{7.16}
$$

The update rule is the same as in the n-step Sarsa:

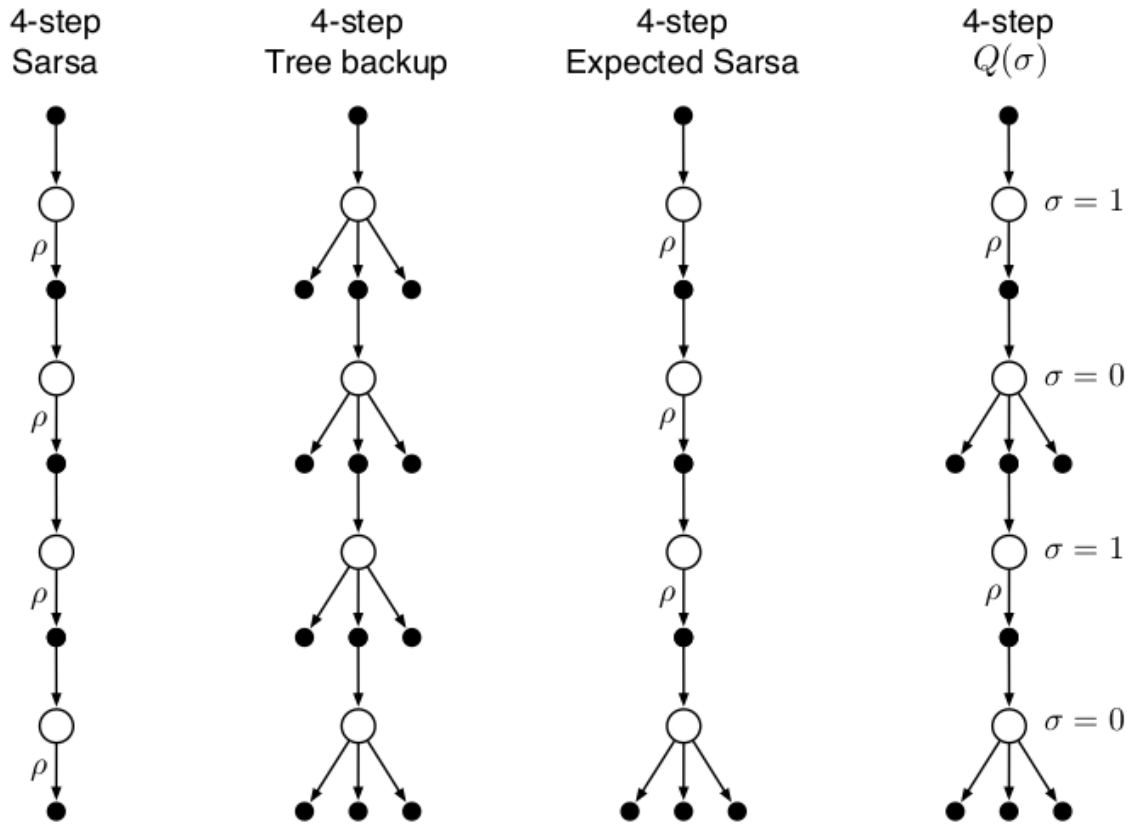
$$
Q_{t+n} \doteq Q_{t+n-1}(S_t, A_t) + \alpha \left[G_{t:t+n} - Q_{t+n-1}(S_t, A_t) \right]
$$

n-step Tree Backup for estimating $Q \approx q_*$ or q_π

Initialize $Q(s,a)$ arbitrarily, for all $s \in \mathcal{S}, a \in \mathcal{A}$
Initialize π to be greedy with respect to Q, or as a fixed given policy
Algorithm parameters: step size $\alpha \in (0,1]$, a positive integer n
All store and access operations can take their index mod $n+1$

Loop for each episode:
  Initialize and store $S_0 \neq$ terminal
  Choose an action A_0 arbitrarily as a function of S_0; Store A_0
  $T \leftarrow \infty$
  Loop for $t = 0, 1, 2, \ldots$:
  | If $t < T$:
  | Take action A_t; observe and store the next reward and state as R_{t+1}, S_{t+1}
  | If S_{t+1} is terminal:
  | $T \leftarrow t+1$
  | else:
  | Choose an action A_{t+1} arbitrarily as a function of S_{t+1}; Store A_{t+1}
  | $\tau \leftarrow t+1-n$ (τ is the time whose estimate is being updated)
  | If $\tau \geq 0$:
  | If $t+1 \geq T$:
  | $G \leftarrow R_T$
  | else
  | $G \leftarrow R_{t+1} + \gamma \sum_a \pi(a|S_{t+1}) Q(S_{t+1}, a)$
  | Loop for $k = \min(t, T-1)$ down through $\tau+1$:
  | $G \leftarrow R_k + \gamma \sum_{a \neq A_k} \pi(a|S_k) Q(S_k, a) + \gamma \pi(A_k|S_k) G$
  | $Q(S_\tau, A_\tau) \leftarrow Q(S_\tau, A_\tau) + \alpha \left[G - Q(S_\tau, A_\tau) \right]$
  | If π is being learned, then ensure that $\pi(\cdot|S_\tau)$ is greedy wrt Q
  Until $\tau = T-1$

7.6 *A Unifying Algorithm: n-step $Q(\sigma)$



Depending on the *random* variable $\sigma \in [0, 1]$ the algorithm either samples with *importance sampling* or calculates via expectations and π. σ can be a function of t, s, (s, a),...

$$\begin{aligned} G_{t:h} \doteq R_{t+1} &+ \gamma \left(\sigma_{t+1}\rho_{t+1} + (1 - \sigma_{t+1})\pi(A_{t+1} \mid S_{t+1})\right)\left(G_{t+1:h} - Q_{h-1}(S_{t+1}, A_{t+1})\right) \\ &+ \gamma \bar{V}_{h-1}(S_{t+1}) \\ G_{h:h} &\doteq Q_{h-1}(S_h, A_h), \qquad h < T \\ G_{T-1:h} &\doteq R_T, \qquad\qquad h == T \end{aligned} \tag{7.17}$$

The update rule is the same as in the n-step Sarsa:

$$Q_{t+n} \doteq Q_{t+n-1}(S_t, A_t) + \alpha \left[G_{t:t+n} - Q_{t+n-1}(S_t, A_t)\right]$$

Off-policy n-step $Q(\sigma)$ for estimating $Q \approx q_*$ or q_π

Input: an arbitrary behavior policy b such that $b(a|s) > 0$, for all $s \in \mathcal{S}, a \in \mathcal{A}$
Initialize $Q(s,a)$ arbitrarily, for all $s \in \mathcal{S}, a \in \mathcal{A}$
Initialize π to be ε-greedy with respect to Q, or as a fixed given policy
Algorithm parameters: step size $\alpha \in (0,1]$, small $\varepsilon > 0$, a positive integer n
All store and access operations can take their index mod $n+1$

Loop for each episode:
  Initialize and store $S_0 \neq$ terminal
  Choose and store an action $A_0 \sim b(\cdot|S_0)$
  $T \leftarrow \infty$
  Loop for $t = 0, 1, 2, \ldots$:
    If $t < T$:
      Take action A_t; observe and store the next reward and state as R_{t+1}, S_{t+1}
      If S_{t+1} is terminal:
        $T \leftarrow t+1$
      else:
        Choose and store an action $A_{t+1} \sim b(\cdot|S_{t+1})$
        Select and store σ_{t+1}
        Store $\frac{\pi(A_{t+1}|S_{t+1})}{b(A_{t+1}|S_{t+1})}$ as ρ_{t+1}
    $\tau \leftarrow t - n + 1$ (τ is the time whose estimate is being updated)
    If $\tau \geq 0$:
      $G \leftarrow 0$:
      Loop for $k = \min(t+1, T)$ down through $\tau + 1$:
        if $k = T$:
          $G \leftarrow R_T$
        else:
          $\bar{V} \leftarrow \sum_a \pi(a|S_k) Q(S_k, a)$
          $G \leftarrow R_k + \gamma\big(\sigma_k \rho_k + (1 - \sigma_k)\pi(A_k|S_k)\big)\big(G - Q(S_k, A_k)\big) + \gamma \bar{V}$
      $Q(S_\tau, A_\tau) \leftarrow Q(S_\tau, A_\tau) + \alpha\,[G - Q(S_\tau, A_\tau)]$
      If π is being learned, then ensure that $\pi(\cdot|S_\tau)$ is greedy wrt Q
  Until $\tau = T - 1$

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 *MCMs* 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. All n-step methods involve a delay of n timesteps before updating, as only then are all the required future events known. A further drawback is that they involve more computation per timestep 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.

Chapter 8

Planning and Learning with Tabular Methods

Model-based methods rely on *planning* as their primary component, while model-free methods primarily rely on *learning*.

8.1 Models and Planning

Distribution models produce a description of all possibilities after a state-action pair and the corresponding probabilities. *Sample models* produce just one of the possibilities after a state-action pair, sampled according to the probabilities. Obviously distribution models are more useful, and if needed they can be used to create samples and have the output of the sample models. The opposite is not possible. The model is used to *simulate* the environment and produce *simulated experience*. *Planning* is a computational process that takes a model as input and produces or improves a policy for interacting with the modeled environment:

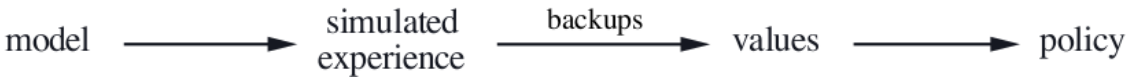
Two distinct approaches to planning:

- *State-space planning*: a search through the state space for an optimal policy or an optimal path to a goal. Value functions are computed for states or state-action pairs and actions cause transitions from state to state.
- *Plan-space planning*: a search through the space of plans. Operators transform one plan into another, and value functions, if any, are defined over the space of plans. These methods are difficult to apply efficiently to the stochastic sequential decision problems that are the focus in reinforcement learning, and we do not consider them further.

All *state-space planning methods* share a common structure:

1. They all involve computing value functions as a key intermediate step to improving the policy.

2. They compute value functions by updates or backup operations applied to simulated experience.



Planning uses simulated experience generated by a model, learning methods use real experience generated by the environment.

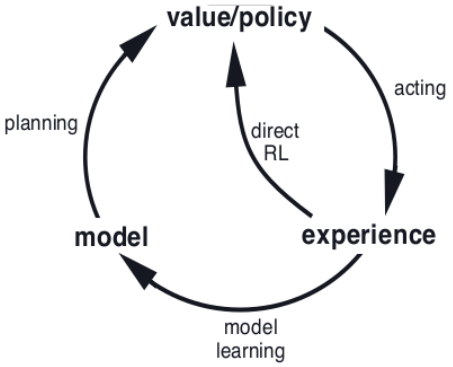
Random-sample one-step tabular Q-planning

Loop forever:

1. Select a state, $S \in \mathcal{S}$, and an action, $A \in \mathcal{A}(S)$, at random
2. Send S, A to a sample model, and obtain
   a sample next reward, R, and a sample next state, S'
3. Apply one-step tabular Q-learning to S, A, R, S':
   $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$

8.2 Dyna: Integrated Planning, Acting, and Learning

Within a planning agent, there are at least two roles for real experience: it can be used to improve the model (to make it more accurately match the real environment) and it can be used to directly improve the value function and policy using the kinds of reinforcement learning methods we have discussed in previous chapters. The former we call *model-learning*, and the latter we call *direct reinforcement learning* (direct RL). Dyna-Q includes all of the processes shown in the diagram above - planning, acting, model-learning, and direct RL - all occurring continually.



- planning: one-step tabular Q-planning
- direct RL: one-step tabular Q-learning
- model learning: stores transitions deterministically in the table

During planning the random sampling from the model is done only for (s, a) that have been experienced.

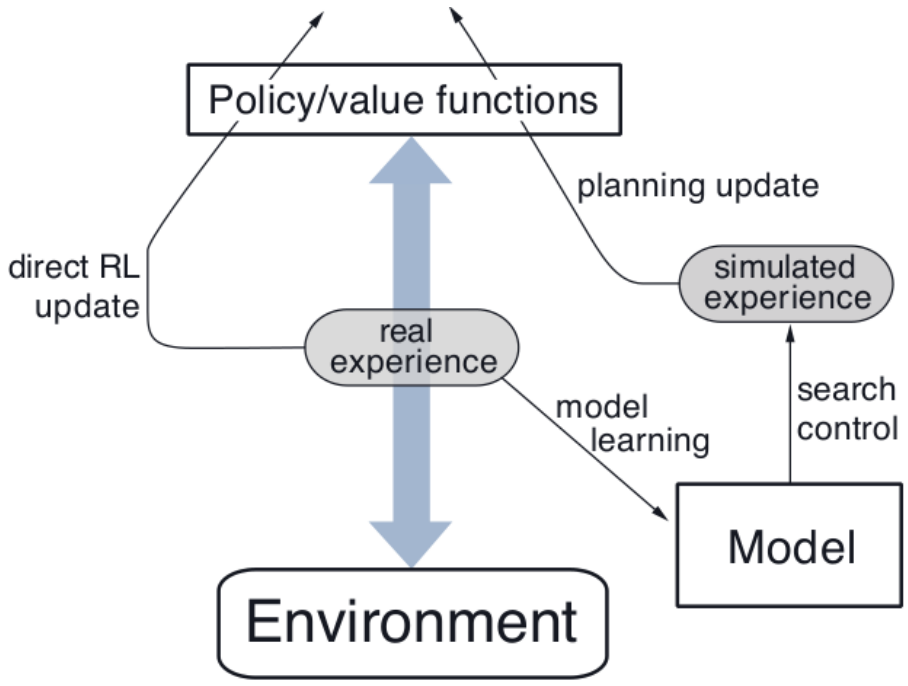


Figure 8.1: The general Dyna Architecture. Real experience, passing back and forth between the environment and the policy, affects policy and value functions in much the same way as does simulated experience generated by the model of the environment.

Tabular Dyna-Q

Initialize $Q(s,a)$ and $Model(s,a)$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$
Loop forever:
- (a) $S \leftarrow$ current (nonterminal) state
- (b) $A \leftarrow \varepsilon$-greedy(S,Q)
- (c) Take action A; observe resultant reward, R, and state, S'
- (d) $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$
- (e) $Model(S,A) \leftarrow R, S'$ (assuming deterministic environment)
- (f) Loop repeat n times:
  - $S \leftarrow$ random previously observed state
  - $A \leftarrow$ random action previously taken in S
  - $R, S' \leftarrow Model(S,A)$
  - $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$

8.3 When the Model Is Wrong

Models may be incorrect because the environment is stochastic and only a limited number of samples have been observed, or because the model was learned using function approximation that has generalized imperfectly, or simply because the environment has changed and its new behavior has not yet been observed. The suboptimal policy computed by planning quickly leads to the discovery and correction of

the modeling error. This tends to happen when the model is optimistic in the sense of predicting greater reward or better state transitions than are actually possible. The planned policy attempts to exploit these opportunities and in doing so discovers that they do not exist.

The Dyna-Q+ agent uses a heruistic where it tracks the last time a particular state-action pair has been last tried in a real interaction with the environment. The more time passes the greater the possibility that the dynamics of the environment under such a state-action pair have changed. To stimulate the agent to take these exploratory steps a special *bonus reward* is given in simulated experiences involving these actions. In particular, if the modeled reward for a transition is r, and the transition has not been tried in τ time steps, then planning updates are done as if that transition produces a reward of $r + \kappa\sqrt{\tau}$, for some small κ.

8.4 Prioritized Sweeping

Pick transitions to learn from using a kind of priority. Realize that some transitions will not provide any added knowledge, while others might. It would be good to work *backwards* from goal states, but this notion can be extended to working back from any state whose value has changed. One can work backward from arbitrary states that have changed in value, either performing useful updates or terminating the propagation. This general idea might be termed backward focusing of planning computations. A queue is maintained of every state-action pair whose estimated value would change nontrivially if updated, prioritized by the size of the change. When the top pair in the queue is updated, the effect on each of its predecessor pairs is computed. If the effect is greater than some small threshold, then the pair is inserted in the queue with the new priority (if there is a previous entry of the pair in the queue, then insertion results in only the higher priority entry remaining in the queue). Priority is assigned using *td error for Q-learning*.

Prioritized sweeping for a deterministic environment

Initialize $Q(s,a)$, $Model(s,a)$, for all s, a, and $PQueue$ to empty
Loop forever:
- (a) $S \leftarrow$ current (nonterminal) state
- (b) $A \leftarrow policy(S, Q)$
- (c) Take action A; observe resultant reward, R, and state, S'
- (d) $Model(S, A) \leftarrow R, S'$
- (e) $P \leftarrow |R + \gamma \max_a Q(S', a) - Q(S, A)|$.
- (f) if $P > \theta$, then insert S, A into $PQueue$ with priority P
- (g) Loop repeat n times, while $PQueue$ is not empty:
  - $S, A \leftarrow first(PQueue)$
  - $R, S' \leftarrow Model(S, A)$
  - $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma \max_a Q(S', a) - Q(S, A)\big]$
  - Loop for all $\bar{S}, \bar{A}$ predicted to lead to S:
    - $\bar{R} \leftarrow$ predicted reward for $\bar{S}, \bar{A}, S$
    - $P \leftarrow |\bar{R} + \gamma \max_a Q(S, a) - Q(\bar{S}, \bar{A})|$.
    - if $P > \theta$ then insert $\bar{S}, \bar{A}$ into $PQueue$ with priority P

(g) - backpropagate the update to previous states as long as δ is over θ.

8.5 Expected vs. Sample Updates

In favor of the expected update is that it is an exact computation, resulting in a new $Q(s, a)$ whose correctness is limited only by the correctness of the $Q(s_0, a_0)$ at successor states. The sample update is in addition affected by sampling error. On the other hand, the sample update is cheaper computationally because it considers only one next state, not all possible next states. For a particular starting pair (s, a), let b be the *branching factor* (i.e., the number of possible next states (s') for which $\hat{p}(s' \mid s, a) > 0$). Then an expected update of this pair requires roughly b times as much computation as a sample update.

8.6 Trajectory Sampling

Attempt to distribute updates according to the on-policy distribution, that is, according to the distribution observed when following the current policy. Sample state transitions and rewards are given by the model, and sample actions are given by the behavior policy. Basically, one simulates explicit individual trajectories and performs updates at the state or state-action pairs encountered along the way.

The bigger the state space and the lower the *branching factor* (b) the bigger the benefit of *trajectory sampling*. Conversely, in the long run, uniform sampling from the model does better when the state space is small and b big. Furthermore, *trajectory sampling* shows better performance early in learning. This makes sense because in the short term, sampling according to the on-policy distribution helps by focusing on states that are near descendants of the start state. If there are many states and a small branching factor, this effect will be large and long-lasting. In the long run, focusing on the on-policy distribution may hurt because the commonly occurring states all already have their correct values. Sampling them is useless, whereas sampling other states may actually perform some useful work.

8.7 Real-time Dynamic Programming (RTDP)

Like *asynchronous DP*, we update the value function asynchronously. In the case of *RTDP* the states for which the value function is updated are chosen by the agent's experience with the environment.

This algorithm can successfully find an *optimal partial policy* - a policy that is optimal for the relevant states but can specify arbitrary actions, or even be undefined, for the irrelevant states. Given some reasonable conditions, *RTDP* is guaranteed to find a policy that is optimal on the relevant states without visiting every state infinitely often, or even without visiting some states at all. The conditions are the following:

1. the initial value of every goal state is zero

2. there exists at least one policy that ensures that a goal state will be reached with probability 1 from any given start state
3. all rewards for transitions from non-goal states are strictly negative
4. all the initial values for states are equal to, or greater than, their optimal values

8.8 Planning at Decision Time

Background planning is a type of planning that improves table entries regardless of current state S_t. *Decision-time planning* is a type of planning where we plan the action for the given state S_t when we encounter it. These two ways of thinking about planning - using simulated experience to gradually improve a policy or value function, or using simulated experience to select an action for the current state - can blend together in natural and interesting ways. In general, one may want to do a mix of both: focus planning on the current state and store the results of planning so as to be that much farther along should one return to the same state later. If low latency action selection is the priority, then one is generally better off doing planning in the background to compute a policy that can then be rapidly applied to each newly encountered state.

8.9 Heuristic Search

In heuristic search, for each state encountered, a large tree of possible continuations is considered. The approximate value function is applied to the leaf nodes and then backed up toward the current state at the root. If one has a perfect model and an imperfect action-value function, then in fact deeper search will usually yield better policies. Certainly, if the search is all the way to the end of the episode, then the effect of the imperfect value function is eliminated, and the action determined in this way must be optimal. Focusing of memory and computational resources on the current decision is presumably the reason why heuristic search can be so effective.

8.10 Rollout Algorithms

Rollout algorithms are decision-time planning algorithms based on Monte Carlo control applied to simulated trajectories that all begin at the current environment state. They produce Monte Carlo estimates of action values only for the current state and for a given policy usually called the *rollout policy*. Averaging the returns of the simulated trajectories produces estimates of $q_\pi(s, a')$ for each action $a' \in \mathcal{A}(s)$. Then the policy that selects an action in s that maximizes these estimates and thereafter follows π is a good candidate for a policy that improves over π. The computation time needed by a rollout algorithm depends on:

- the number of actions that have to be evaluated for each decision

- the number of time steps in the simulated trajectories needed to obtain useful sample returns
- the time it takes the rollout policy to make decisions
- the number of simulated trajectories needed to obtain good MC action-value estimates

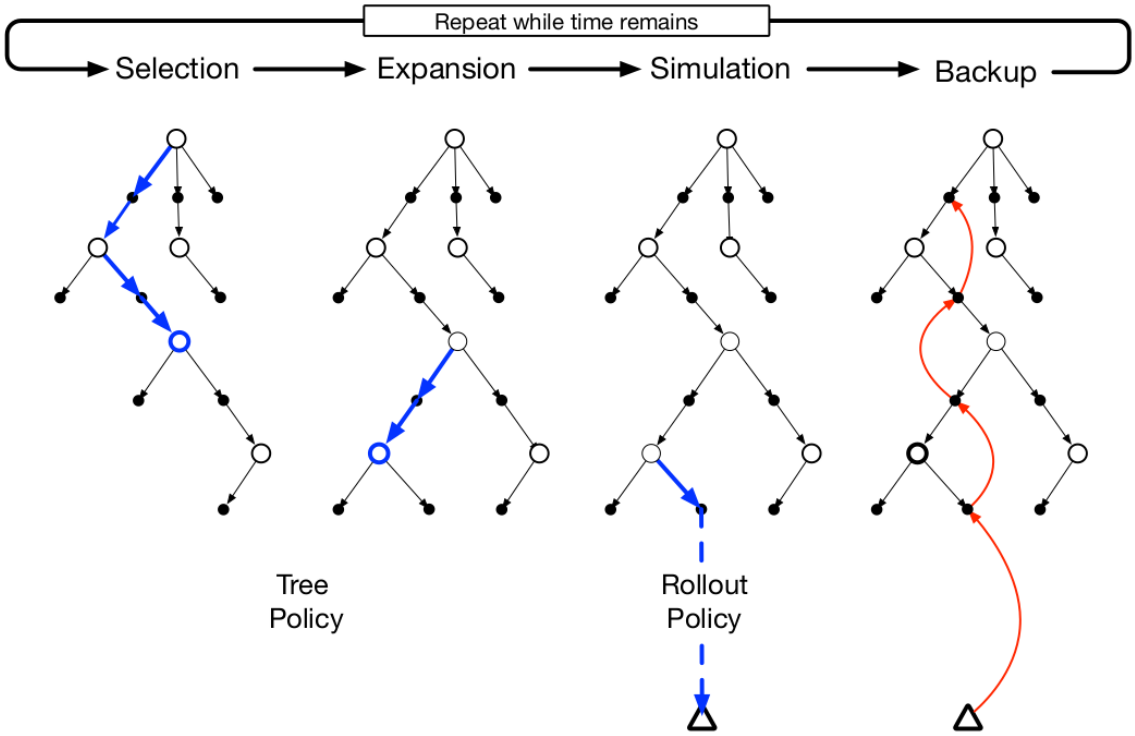
As instances of Monte Carlo control, they estimate action values by averaging the returns of a collection of sample trajectories, in this case trajectories of simulated interactions with a sample model of the environment.

8.11 Monte Carlo Tree Search (MCTS)

MCTS is a recent and strikingly successful example of decision-time planning. At its base, *MCTS* is a rollout algorithm as described above, but enhanced by the addition of a means for accumulating value estimates obtained from the Monte Carlo simulations in order to successively direct simulations toward more highly-rewarding trajectories. As in any tabular Monte Carlo method, the value of a state-action pair is estimated as the average of the (simulated) returns from that pair. MC value estimates are maintained only for the subset of state-action pairs that are most likely to be reached in a few steps, which form a tree rooted at the current state. Outside the tree and at the leaf nodes the rollout policy is used for action selections, but at the states inside the tree we have value estimates of at least some of the actions, so we can pick among them using an informed policy, called the *tree policy*, that balances exploration and exploitation (ϵ-greedy or *UCB*).

Four steps of *MCTS*:

1. **Selection:** starting from the root node, the *tree policy* selects a leaf node.
2. **Expansion:** on some iterations the leaf node is extended by attaching one or more nodes to it as unexplored actions.
3. **Simulation:** continuing from the selected leaf node or one of its children (if such are added) we simulate a trajectory following the *rollout policy*.
4. **Backup:** back up the update from the simulated trajectory and update the edges of the tree traversed by the *tree policy*.



This process is repeated for as long as there is time to make a decision in the given timepoint/state. When time terminates or some other resource is exhausted the algorithm selects an action from the root given some mechanism that depends on the accumulated statistics in the tree (the node with the greatest value, the node visited the most times,...). After the environment transitions to a new state, *MCTS* is run again, sometimes starting with a tree of a single root node representing the new state, but often starting with a tree containing any descendants of this node left over from the tree constructed by the previous execution of *MCTS*; all the remaining nodes are discarded.

By incrementally expanding the tree, *MCTS* effectively grows a lookup table to store a partial action-value function, with memory allocated to the estimated values of state-action pairs visited in the initial segments of high-yielding sample trajectories. *MCTS* thus avoids the problem of globally approximating an action-value function while it retains the benefit of using past experience to guide exploration.

8.12 Summary

Planning requires a model of the environment. A *distribution model* consists of the probabilities of next states and rewards for possible actions; a *sample model* produces single transitions and rewards generated according to these probabilities. *DP* requires a distribution model because it uses expected updates, which involve computing expectations over all possible next states and rewards. A sample model, on the other hand, is what is needed to simulate interacting with the environment during which sample updates, like those used by many reinforcement learning algorithms, can be used. Sample models are generally much easier to obtain than distribution models.

We have presented a perspective emphasizing the surprisingly close relationships between planning optimal behavior and learning optimal behavior. Both involve estimating the same value functions, and in both cases it is natural to update the estimates incrementally, in a long series of small backing-up operations. This makes it straightforward to integrate learning and planning processes simply by allowing both to update the same estimated value function. In addition, any of the learning methods can be converted into planning methods simply by applying them to simulated (model-generated) experience rather than to real experience. In this case learning and planning become even more similar; they are possibly identical algorithms operating on two different sources of experience.

It is straightforward to integrate incremental planning methods with acting and model-learning. Planning, acting, and model-learning interact in a circular fashion, each producing what the other needs to improve; no other interaction among them is either required or prohibited. The most natural approach is for all processes to proceed asynchronously and in parallel. If the processes must share computational resources, then the division can be handled almost arbitrarily - by whatever organization is most convenient and efficient for the task at hand.

In this chapter we have touched upon a number of dimensions of variation among state-space planning methods. One dimension is the variation in the size of updates. The smaller the updates, the more incremental the planning methods can be. Among the smallest updates are one-step sample updates, as in Dyna. Another important dimension is the distribution of updates, that is, of the focus of search. Prioritized sweeping focuses backward on the predecessors of states whose values have recently changed. On-policy trajectory sampling focuses on states or state-action pairs that the agent is likely to encounter when controlling its environment. This can allow computation to skip over parts of the state space that are irrelevant to the prediction or control problem. *RTDP*, an on-policy trajectory sampling version of value iteration, illustrates some of the advantages this strategy has over conventional sweep-based policy iteration.

Planning can also focus forward from pertinent states, such as states actually encountered during an agent-environment interaction. The most important form of this is when planning is done at decision time, that is, as part of the action-selection process. Classical heuristic search as studied in artificial intelligence is an example of this. Other examples are *rollout algorithms* and *MCTS* that benefit from online, incremental, sample-based value estimation and policy improvement.

Summary of Tabular Solution Methods

This chapter concludes Part I of this book. In it we have tried to present reinforcement learning not as a collection of individual methods, but as a coherent set of ideas cutting across methods. Each idea can be viewed as a dimension along which methods vary. The set of such dimensions spans a large space of possible methods. By exploring this space at the level of dimensions we hope to obtain the broadest and most lasting understanding. In this section we use the concept of dimensions in method space to recapitulate the view of reinforcement learning developed so far in this book.

All of the methods we have explored so far in this book have three key ideas in common:

1. they all seek to estimate value functions
2. they all operate by backing up values along actual or possible state trajectories
3. they all follow the general strategy of *GPI*

These three ideas are central to the subjects covered in this book. We suggest that value functions, backing up value updates, and GPI are powerful organizing principles potentially relevant to any model of intelligence, whether artificial or natural.

Two of the most important dimensions along which the methods vary are shown in Figure 8.2. These dimensions have to do with the kind of update used to improve the value function. The horizontal dimension is whether they are sample updates (based on a sample trajectory) or expected updates (based on a distribution of possible trajectories). Expected updates require a distribution model, whereas sample updates need only a sample model, or can be done from actual experience with no model at all (another dimension of variation). The vertical dimension corresponds to the depth of updates, that is, to the degree of bootstrapping. At three of the four corners of the space are the three primary methods for estimating values: *DP*, *TD*, and *Monte Carlo*. Along the left edge of the space are the sample-update methods, ranging from one-step TD updates to full-return Monte Carlo updates. Between these is a spectrum including methods based on n-step updates (and in Chapter 12 we will extend this to mixtures of n-step updates such as the λ-updates implemented by eligibility traces).

DP methods are shown in the extreme upper-right corner of the space because they involve one-step expected updates. The lower-right corner is the extreme case

of expected updates so deep that they run all the way to terminal states (or, in a continuing task, until discounting has reduced the contribution of any further rewards to a negligible level). This is the case of exhaustive search. Intermediate methods along this dimension include heuristic search and related methods that search and update up to a limited depth, perhaps selectively. There are also methods that are intermediate along the horizontal dimension. These include methods that mix expected and sample updates, as well as the possibility of methods that mix samples and distributions within a single update. The interior of the square is filled in to represent the space of all such intermediate methods.

A third dimension that we have emphasized in this book is the binary distinction between on-policy and off-policy methods. In the former case, the agent learns the value function for the policy it is currently following, whereas in the latter case it learns the value function for the policy for a different policy, often the one that the agent currently thinks is best. The policy generating behavior is typically different from what is currently thought best because of the need to explore. This third dimension might be visualized as perpendicular to the plane of the page in Figure 8.2.

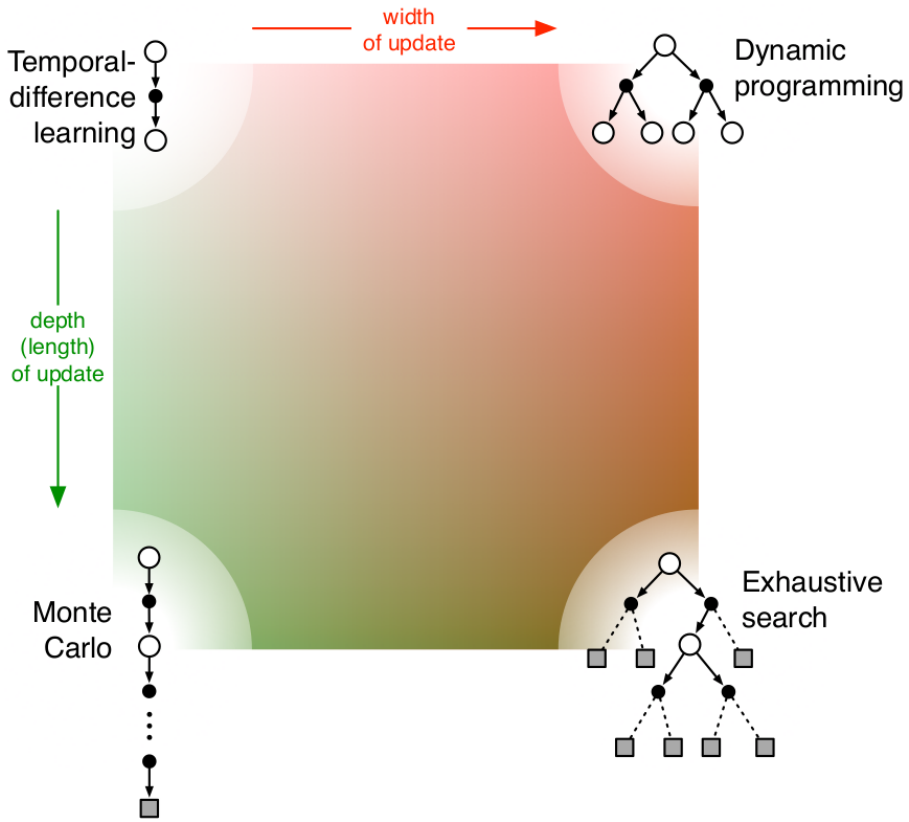


Figure 8.2: A slice through the space of reinforcement learning methods, highlighting the two of the most important dimensions explored in Part I of this book: the depth and width of the updates.

In addition to the three dimensions just discussed, we have identified a number of others throughout the book:

- **Definition of return.** Is the task episodic or continuing, discounted or undiscounted?
- **Action values vs. state values vs. afterstate values.** What kind of values should be estimated? If only state values are estimated, then either a model or a separate policy (as in actorcritic methods) is required for action selection.
- **Action selection/exploration.** How are actions selected to ensure a suitable trade-off between exploration and exploitation? We have considered only the simplest ways to do this: ϵ-greedy, optimistic initialization of values, soft-max, and upper confidence bound.
- **Synchronous vs. asynchronous.** Are the updates for all states performed simultaneously or one by one in some order?
- **Real vs. simulated.** Should one update based on real experience or simulated experience? If both, how much of each?
- **Location of updates.** What states or state-action pairs should be updated? Model-free methods can choose only among the states and state-action pairs actually encountered, but model-based methods can choose arbitrarily. There are many possibilities here.
- **Timing of updates.** Should updates be done as part of selecting actions, or only afterward?
- **Memory for updates.** How long should updated values be retained? Should they be retained permanently, or only while computing an action selection, as in heuristic search?

Of course, these dimensions are neither exhaustive nor mutually exclusive. Individual algorithms differ in many other ways as well, and many algorithms lie in several places along several dimensions. For example, Dyna methods use both real and simulated experience to affect the same value function. It is also perfectly sensible to maintain multiple value functions computed in different ways or over different state and action representations. These dimensions do, however, constitute a coherent set of ideas for describing and exploring a wide space of possible methods.

The most important dimension not mentioned here, and not covered in Part I of this book, is that of function approximation. Function approximation can be viewed as an orthogonal spectrum of possibilities ranging from tabular methods at one extreme through state aggregation, a variety of linear methods, and then a diverse set of nonlinear methods. This dimension is explored in Part II.

Part II

Approximate Solution Methods