Course Notes

Reinforcement Learning II: Learning Values

Richard S. Sutton

In this lecture we examine the reinforcement learning problem in more detail and survey the major classes of methods for solving it: dynamic programming, Monte Carlo methods, and temporal-difference learning.

1 The Reinforcement Learning Problem

Reinforcement learning, like most other kinds of learning, is not really defined by a set of learning methods, but by a learning *problem.* In this section we describe the reinforcement learning problem in a broad sense. Through many informal examples we try to convey the wide range of possible applications that can be framed as reinforcement learning problems. We also describe mathematically idealized forms of the reinforcement learning problem for which precise theoretical statements can be made. We introduce key elements of the problem's mathematical structure, such as value functions and the Bellman equation. As in all of artificial intelligence, there is a tension between breadth of applicability and mathematical tractability. In this section we begin to understand this tension and the tradeoffs and challenges that it implies. In short, we lay out the space of problems that we attempt to solve in reinforcement learning.

1.1 The Agent-Environment Interface

Reinforcement learning is about learning a mapping from situations to actions to achieve a goal. It is a simplified abstraction of a decision-maker, or *agent*, interacting with its environment. At each discrete time step, $t = 1, 2, 3, \ldots$, the agent finds itself in a *situation*, $s_t \in \mathcal{S}$, and on that basis chooses an *action*, $a_t \in \mathcal{A}(s_t)$. One time step later, in part as a consequence of its action, the agent receives a *reward*, $r_{t+1} \in \Re$, and finds itself in a new situation, s_{t+1}. Reinforcement learning methods specify how such experiences result in changes in the agent's mapping

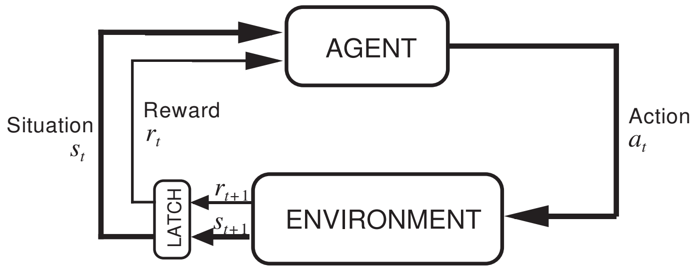


Figure 1: The Reinforcement Learning Framework

from situations to actions, called it's *policy*. Roughly speaking, the agent's objective is to find a policy—a way of behaving—that maximizes the reward it receives in the long run.

This framework is an abstract and flexible one that can be applied in many different ways to many different problems. The actions, for example, could be low level controls like the voltages applied to the motors of a robot arm, or high level decisions like whether or not to have lunch or to go to graduate school. The situations too can take on a wide variety of forms. They may be low-level, like direct sensor readings, or high level, like symbolic descriptions of the objects in a room. Some of the things making up a situation could even be entirely mental or subjective. For example, the agent could be in "the situation" of not being sure where an object is, or of have just been "surprised" in some clearly defined sense. Similarly, some actions could also be be totally mental or computational. For example, some actions may control what the agent chooses to think about, or where it focuses its attention. In general, actions can be any decisions we want to learn how to make, and the situations can be anything we can measure that might be useful in making them.

In particular, it is a mistake to think of the interface between the agent and the environment as the physical boundary of a robot's or animal's body. Usually, the boundary is drawn closer to the agent than that. For example, the motors and mechanical linkages of a robot and its sensing hardware should usually be considered part of the environment rather than part of the agent. Similarly, if we apply the framework to a person or animal, the muscles, skeleton, and sensory organs, such as eyes and ears, should all be considered part of the environment. Rewards too are presumably computed inside the physical bodies of natural and artificial learning systems but are considered external to the reinforcement learning agent.

The general rule we follow is that anything which cannot be changed arbitrarily by the reinforcement learning agent is considered outside of it and thus part of the environment. Note that we do not assume that everything in the environment is unknown to the agent, only that it is incompletely controllable. For example, the agent will often know quite a bit about how its rewards are computed as a function of its actions and the situations in which they are taken.

But the reward computation is always considered external to the agent because it defines the problem facing the agent and thus must be beyond its ability to change arbitrarily. In some cases in fact the agent may know *everything* about its environment and still face a difficult reinforcement learning problem, just as I may know exactly how a puzzle like Rubik's cube works, but still be unable to solve it. The agent-environment boundary represents the limit of the agent's *control*, not of its knowledge.

Example 1 Suppose reinforcement learning is being applied to determine moment-by-moment temperatures and stirring rates for a bioreactor (a large vat of nutrients and bacteria used to produce useful chemicals). The actions in such an application might be *target* temperatures and *target* stirring rates that are passed to lower-level control systems which in turn directly activate heating elements and motors to attain the targets. The sensations are likely to be thermocouple and other sensory readings, perhaps filtered and delayed, plus symbolic inputs representing the ingredients in the vat and the target chemical. Finally, the reward might be a moment-by-moment measure of the rate at which useful chemical is produced by the bioreactor.

Example 2 Consider using reinforcement learning to control the motion of a robot arm in a repetitive pick and place task. If we want to learn movements that are fast and smooth, the reinforcement learning agent will have to directly control the motors and have low-latency information about the current positions and velocities of the mechanical linkages. The actions in this case will be the currents applied to each motor at each joint, and the situations will be the latest readings of joint angle, velocity, and force. The reward might be simply -1 for each time step that elapses. To encourage smooth movements, each time step's reward can be lowered still further by a moment-to-moment measure of the "jerkiness" of the motion.

1.2 Goals, Rewards, and Returns

In reinforcement learning, the concept of goal is modeled by a special scalar signal called *the reward* that passes from the environment to the agent. Informally, the agent's goal is to maximize the total reward it receives. This means not just immediate reward, but reward in the long run.

The use of a scalar reward signal to formalize the idea of a goal is one of the most distinctive features of reinforcement learning. Although this way of formulating goals might at first appear limiting, in practice it has proven very flexible and widely applicable. The best way to see this is just to consider examples of how it has been or could be used. For example, to teach a robot to walk, researchers have provided reward on each time step proportional to the robot's forward motion. In maze-running applications, the reward is often zero except upon reaching the goal, when it becomes +1. Another common approach in maze learning is to give a reward of -1 for every time step that passes prior to reaching the goal; this encourages the agent to reach the goal as quickly as possible. To teach a robot to find and collect empty soda cans for recycling, one might give it a reward of +1 for each can collected (and identified as empty). One might also want to give it negative rewards for bumping into things, or for having people yell at it! To teach an agent to play chess, the natural rewards are +1, -1, and 0 for winning, losing, and drawing.

It is important to remember that the rewards define the ultimate goal of the learning process. The rewards that you arrange to be delivered to a reinforcement learning agent should represent what you really want it to do. In particular, the reward signal is not the place to impart to the agent prior knowledge about *how* to achieve what you want it to do. For example, a chess playing agent should be rewarded only for actually winning, not for achieving subgoals such taking its opponent's pieces or gaining control of the center of the board. If these sort of subgoals are included in the rewards, the agent may find a way to achieve the subgoals without achieving the real goal, e.g., to take the opponent's pieces even at the cost of losing the game. We will see later how subgoal knowledge can be safely given to reinforcement learning agents in ways other than in the reward signal. The reward signal is your way of communicating to the robot *what* you want it to achieve, not how you want it achieved.

Newcomers to the reinforcement learning framework are sometimes surprised that the rewards—the definition of the goal of learning—are computed in the environment rather than in the agent. Certainly most ultimate goals for animals are recognized by computations occurring inside their bodies, e.g., in sensors for recognizing food and hunger, or for pain and pleasure. However, this is not a problem because, as we discussed in the previous section, we can simply redraw the agent-environment interface such that these parts of the body are considered outside the agent (and thus part of the environment). For example, if the goal concerns a robot's internal energy reservoirs, then these are considered part of the environment; if the goal concerns the positions of the robot's limbs, then these too can be considered part of the environment—the agent's boundary is drawn at the interface between the limbs and their control systems. For our purposes it is convenient to place the boundary of agent not at the limit of its physical body, but at the limit of its control. Roughly speaking, things are considered part of the agent if they are completely, directly, and with certainty controllable; things are considered part of the environment if they are not. Since the goal is always something over which we have imperfect control, it is placed outside the agent.

However, so far we are still being imprecise when we speak of the goal of learning as maximizing future reward. How might this be formally defined? If the sequence of rewards received after t is denoted $r_{t+1}, r_{t+2}, r_{t+3}, \ldots$, then what precise aspect of this sequence do we wish to maximize? There are several different useful answers to this question. The simplest is to maximize the *total reward*:

$$r_t = r_{t+1} + r_{t+2} + r_{t+3} + \cdots + r_T \tag{1}$$

where T is a final time step. This approach makes sense in applications in which there is a natural notion of final time step, that is, when the agent-environment interaction breaks naturally into subsequences, such as plays of a game, trips through a maze, or any sort of repeated attempt each ending with a reset to a standard state. In these cases (1) is defined as the *return* at time t.

On the other hand, suppose that the agent-environment interaction does not naturally break into identifiable subsequences, but simply goes on and on without limit. For example, this would be the natural way to characterize a continuous process-control application, or an application to a robot with a long expected lifespan. The total-reward formulation then becomes problematic because the final time step becomes $T = \infty$, and the return as given by (1) becomes a sum of an infinite number of terms. Thus the return—the thing which we are trying to maximize—could itself easily be infinite! (For example, suppose the agent receives a reward of $+1$ on all time steps.) With care this mathematical embarrassment can be avoided without totally giving up the informal idea of (1), by using an *average-reward* formulation of return. But for now, we will use a definition of return which is slightly more complex conceptually but which is much simpler mathematically.

The addition concept that we need is that of *discounted* reward. According to this approach, the agent's objective is to learn how to select actions so that at every time step $t \geq 0$ the discounted sum of the rewards it receives over the future is maximized. That is, the objective is to learn how to maximize the following definition of return for each time step t:

$$r_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots \quad = \quad \sum_{k=1}^{\infty} \gamma^{k-1} r_{t+k}, \tag{2}$$

where γ is a scalar, called the *discount factor*.

The discount factor determines the present value of future rewards: a reward received k time steps in the future is worth γ^{k-1} times what it would be worth if it were received now. If $0 \leq \gamma < 1$, this infinite discounted sum is finite as long as the rewards r_k are bounded. If $\gamma = 0$, the agent is "myopic" in being only concerned with maximizing immediate rewards: its objective in this case would be to learn how to act at each time step t so as maximize only r_{t+1}. If the agent's actions do not influence future rewards as well as immediate rewards, then it could maximize (2) by separately maximizing each immediate reward. But acting to maximize immediate reward can reduce access to future rewards so that the total reward is actually reduced. As γ approaches one, the objective explicitly takes future rewards into account: the agent becomes more far–sighted.

Example 3 An early application of reinforcement learning was to the *pole-balancing* problem (see Figure 2). The object here is to apply forces to a cart moving along a track so as to keep a pole hinged to the cart from falling over. We define a *balancing failure* as the pole falling past a given angle from vertical or the cart exceeding the limits of the track. The pole is reset to vertical after each balancing failure. This problem could be treated as a total-reward problem where the natural subsequences are the repeated attempts to balance the pole. The reward in this case would be $+1$ for every time step on which failure did not occur, so that the return at each time would be the number of steps before failure. Alternatively, a discounted-reward approach could be used, giving a reward of -1 on each failure and zero reward at all other times. The return at each time would then be related to $-\gamma^k$, where k is the number of time steps before failure. In either case, the return is maximized by keeping the pole balanced for as long as possible.

1.3 States and Situations

The *state* of a system with respect to an external observer is a summary of all the observer's past experience with the system. The summary need not be a complete history of every observed input and output, but it must retain all information that makes a difference to the system's future behavior. In particular, the observer must be able (in principle) to predict the system's future behavior just as well from knowledge of the state as would be possible from knowledge of the complete history. For example, a chess position is the state of a chess game because it summarizes everything important about the moves that produced it. Much of the information about the individual moves is lost, but all is retained that really matters for the future of the game. For a cannonball in flight, the state may be just its current position and velocity. It doesn't matter how the position and velocity came to be; they alone determine the projectile's future trajectory.

In reinforcement learning, the external observer is the agent, and the state of interest is the state of the environment. In fact, the notion of "situation" is meant to be an approximation to the environment's state. If an MDP has a finite

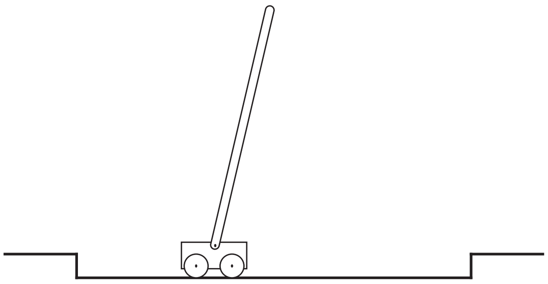


Figure 2: The Pole–Balancing Problem

number of states, and each state a finite number of actions, then the problem is a *finite MDP*. Finite MDPs are particularly easy to think about and to prove theoretical results about. They are an important special case in the theory of reinforcement learning that we will return to again and again.

For the case of finite MDPs it is particularly useful to talk about two particular aspects of the dynamics. These are the transition probabilities,

$$P^a_{ss'} = Pr\{s_{t+1} = s' \mid s_t = s, a_t = a\}, \qquad \forall s, s' \in \mathcal{S}, a \in \mathcal{A}(s),$$

and the expected rewards,

$$R^a_{ss'} = E\{r_{t+1} \mid s_t = s, a_t = a, s_{t+1} = s'\}, \qquad \forall s, s' \in \mathcal{S}, a \in \mathcal{A}(s).$$

These quantities are almost a complete specification of the environment's dynamics (a complete specification would include the entire probability distribution of rewards, not just the expected value). They are sufficient to completely determine the optimal behavior of the agent and are all we will need to deal with. If these quantities are completely known, then we say we have a reinforcement learning problem under conditions of *complete information*.

Although most reinforcement learning algorithms can be applied even when the situations are not states, often with good results, almost all of theory assumes that the situations are states. Accordingly, the theory presented in this course will focus on MDPs. Through the course we will downplay the potential differences between situations and states. When we make theoretical statements, we will usually talk of states, and when we present algorithms, we will usually talk of situations. Of course this means that not all the theory will strictly apply for the case in which the situations are not states. However, even when the situations are not states, they are meant to approximate states, and the theory developed for the state case will still help us to understand the behavior of the algorithms. Furthermore, a full understanding of the theory of the MDP case is an essential foundation for extending it to the more complex and realistic case in which states and situations are distinguished.

Example 4 In the pole-balancing problem introduced in the previous section, the situations might be states if they provided complete information about the location and velocity of the cart along the track and about the angle and angular velocity of the pole. In an idealized cart-pole system, these would be sufficient (along with the future actions, the forces applied to the cart) to exactly predict all the future behavior of the cart and pole. In practice of course it is never possible to know this information exactly. For example, any real sensor would introduce delay between measurement of the positions and reporting the measurement to whatever hardware makes up the agent. And then in any real cart-pole system there are always other state variables, such as the bending of the pole, the temperatures of the axles of the wheels, and various forms of backlash, all of which will slightly affect the behavior of the system in ways that would be unpredictable if only the positions and velocities were used as states.

However, often the positions and velocities serve quite well as approximate states. In several of the early studies of learning to solve the pole-balancing problem, in fact something much cruder was successfully used as a state model. These studies used a coarse quantization of the positions and velocities. For example, the possible cart positions were divided into three regions: right, left, and middle. The situations indicated only in which of these three large regions the cart was located (and similar rough quantizations of the other three intrinsic state variables). For this task, such a rough approximation to the state was sufficient to easily solve the problem using reinforcement learning methods.

Example 5 In draw poker, each player is dealt a hand of five cards. There is a round of betting, each player exchanges some number of his cards for new ones, and then there is a final round of betting. At each round of betting, each player's must match the highest bets of the other players or else drop out (fold). After the second round of betting, the player with the best hand who has not folded is the winner and collects all the bets.

The relevant state in poker is different for each player. Each player knows the cards in his own hand, but can only guess at those in the other player's hands. A common mistake is to think of the state as including the contents of all the player's hands and the cards remaining in the deck. In a fair game we assume that the players are in principle unable to determine these things from their past observations. This then is more than the state: knowing it you would be able to predict some future events (such as the cards one could exchange for) *better* than remembering all past observations.

In addition to knowledge of one's own cards, the state in draw poker includes knowledge of the bets and the numbers of cards drawn by the other players. For example, if a player draws three new cards you may suspect he retained a pair, and adjust your guess at the strength of his hand accordingly. The player's bets will also influence your assessment of their hands. In principle, in fact all your past history with these particular players may also be part of the state. Does Joe like to bluff or does he play conservatively? Does his face or demeanor provide clues to the strength of his hand? How does Ellen's play change when it is late at night or when she has already won a lot of money?

Although in principle everything ever observed about the other players may have an effect on the probabilities that they are holding various kinds of hands, in practice this is far too much to remember and analyze, and most of it will have no clear effect on one's decisions or predictions. Very good poker players are adept as remembering just the key clues, and at sizing up new players quickly, but no one remembers everything that may be relevant. As a result, the situations people use to make their decisions must be imperfect state models, and the decisions themselves are presumably imperfect. Nevertheless, people can still make very good decisions in such problems. The inability to *perfectly* model the environment's state is probably not in and of itself a severe problem for an AI agent.

1.4 Value Functions

Almost all reinforcement learning algorithms are based on estimating *value functions*—functions of situations, or of situation–action pairs, that estimate *how good* it is for the agent to be in that situation. The notion of "how good" here is defined in terms of future rewards that can be expected, or, to be precise, in terms of the expected return. Of course the rewards the agent can expect to receive in the future will depend on what actions it takes. Accordingly, value functions are defined with respect to a particular policy.

Recall that a policy, π, is a mapping from situations $s \in \mathcal{S}$ to probability distributions over possible actions $a \in \mathcal{A}(s)$. Informally, the value of a situation s under a policy π, denoted $V^\pi(s)$, is the expected return when starting in s and following π. For MDPs, we can define $V^\pi(s)$ formally as:

$$V^\pi(s) = E_\pi\{r_t \mid s_t = s\} = E_\pi\left\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \;\middle|\; s_t = s\right\}, \tag{3}$$

where $E_\pi\{\}$ denotes the expected value given that the agent follows policy π. Similarly, we define the *action value* of taking action a in state s under a policy π, denoted $Q^\pi(s, a)$, as the expected return starting from s, taking the action a, and thereafter following policy π:

$$Q^\pi(s, a) = E_\pi\{r_t \mid s_t = s, a_t = a\} = E_\pi\left\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \;\middle|\; s_t = s, a_t = a\right\}. \tag{4}$$

Exercise 1 Give an equation for the action value, $Q^\pi(s, a)$, in terms of the expected next reward, r_{t+1}, and the expected next state value, $V^\pi(s_{t+1})$, given that $s_t = s$ and $a_t = a$. Then write out the expected value explicitly in terms of $P^a_{ss'}$ and $R^a_{ss'}$ such that no expected value notation appears in the equation.

Exercise 2 Give an equation for the state value $V^\pi(s)$ in terms of the expected action value, $Q^\pi(s_t, a_t)$, given that $s_t = s$. This expectation will depend on the policy, π. Then write out the expected value explicitly in terms of $\pi(s, a)$ such that no expected value notation appears in the equation.

For MDPs, the value functions $V^\pi(s)$ and $Q^\pi(s, a)$ can be estimated from experience. For example, if an agent follows policy π and maintains an average, for each situation encountered, of the actual returns that have followed that situation, then the averages will converge to the situation's value, $V^\pi(s)$, as the number of times that situation is encountered approaches infinity. If separate averages are kept for each action taken in a situation, then these averages will similarly converge to the action values, $Q^\pi(s, a)$. Estimation methods of this kind are often called *Monte Carlo methods* because they involve averaging over many random samples of actual returns. Of course if there are very many situations, or very many actions possible in each situation, then it may not be practical to keep separate averages for each situation individually. Instead, the agent must maintain V^π and Q^π as parameterized functions and adjust the parameters to better match the observed returns. This can also produce accurate estimates, although much depends on the nature of the parameterized function approximation.

A fundamental property of value functions used throughout reinforcement learning and dynamic programming is that they satisfy particular recursive relationships if the situations are genuine states. For any policy π and any state s the following consistency condition holds between the value of s and the value of its possible successor states:

$$
\begin{aligned}
V^\pi(s) &= E_\pi\{r_t \mid s_t = s\} \\
&= E_\pi\left\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \,\middle|\, s_t = s\right\} \\
&= E_\pi\left\{r_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k r_{t+k+2} \,\middle|\, s_t = s\right\} \\
&= \sum_a \pi(s, a) \sum_{s'} P^a_{ss'} \left[R^a_{ss'} + \gamma E_\pi\left\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+2} \,\middle|\, s_{t+1} = s'\right\}\right] \\
&= \sum_a \pi(s, a) \sum_{s'} P^a_{ss'} \left[R^a_{ss'} + \gamma V^\pi(s')\right] \qquad (5)
\end{aligned}
$$

This is known as the *Bellman equation* for state values. It expresses the intuitive idea of looking ahead from one state to its possible successors, as suggested by Figure 1a. Each open circle here represents a state, and each solid circle an action, or, rather, a state-action pair. Starting from state s we could take any of some set of actions—three are shown in Figure 1a. From each of these the environment could respond with one of several next states, s', along with a reward, r. The Bellman equation (5) simply averages over all the possibilities, weighting each by its probability of occurring. The value of the start state s must equal the value of the expected next state (discounted) plus the expected reward along the way.

The value function V^π is the unique solution to its Bellman equation (5). We will show in subsequent chapters a number of ways in which the Bellman equation can be used to compute, approximate, and learn V^π.

Exercise 3 What is the Bellman equation for *action values*? It must give the value of $Q^\pi(s, a)$ in terms of the action values, $Q^\pi(s', a')$, of possible successors to the state-action pair s, a. As a hint, the diagram corresponding to this equation

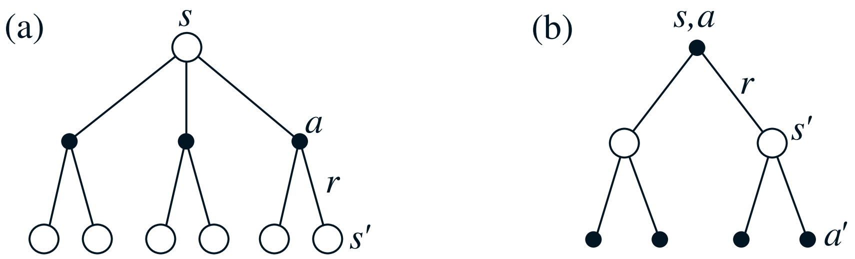
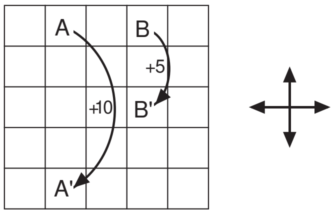


Figure 3: Backup diagrams for a) V^π and b) Q^π

is given in Figure 1b. Show the sequence of equations analogous to (5), but for action values.

Example 6 Figure 2a uses a rectangular grid to illustrate a simple finite MDP. The cells of the grid correspond to the states and situations of the problem. At each cell, four actions are possible: NORTH, SOUTH, EAST, and WEST, which deterministically cause the agent to move one cell in the respective direction in the grid. Actions that would take the agent off the grid leave its location unchanged, but also result in a reward of -1. Other actions result in a reward of 0, except those out of the special states A and B. From state A, all four actions yield a reward of $+10$ and take the system to A'. From state B, all actions yield a reward of $+5$ and take the system to B'.



3.3	8.8	4.4	5.3	1.5
1.5	3.0	2.3	1.9	0.5
0.1	0.7	0.7	0.4	-0.4
-1.0	-0.4	-0.4	-0.6	-1.2
-1.9	-1.3	-1.2	-1.4	-2.0

(a) (b)

Figure 4: Grid Example

Suppose the agent selects all four actions with equal probability in all states. Figure 2b shows the value function, V^π, for this policy, for the discounted–reward case with $\gamma = 0.9$. This value function was computed by solving the system of equations given by (5). Notice the negative values near the lower edge; these are the result of the high probability of hitting the edge of the grid there under the random policy. Notice that A is the best state to be in under this policy, but that its expected return is less than 10, its immediate reward, because from A the agent is taken to A', from which it is likely to run into the edge of the grid. State B, on the other hand, is valued more than 5, its immediate reward, because from B the agent is taken to B', which has a positive value. From B'

the expected penalty (negative reward) for possibly running into an edge is more than compensated for by the expected gain for possibly stumbling onto A or B.

Exercise 4 The Bellman equation (5) must hold for each state for the value function V^π shown in Figure 2b. As an example, show numerically that this equation holds for the center state, valued at $+0.7$, with respect to its four neighboring states, valued at $+2.3$, $+0.4$, -0.4, and 0.7. (These numbers are accurate only to one decimal place.)

Exercise 5 In the gridworld example, rewards are positive for goals, negative for running into the edge of the world, and zero all the rest of the time. Are the signs of these rewards important, or only the intervals between them? Prove using (2) that adding a constant C to all the rewards simply adds a constant K to the values of all states, and thus does not affect the relative value of any policies. What is K in terms of C and γ?

Exercise 6 Do the absolute values of the rewards matter in the total-reward case? Why or why not? Give an example.

1.5 Bellman Optimality Equations

Solving a reinforcement learning problem means, roughly, finding a policy that maximizes the amount of reward the agent receives over the long run. For finite MDPs, we can precisely define an optimal policy in the following simple way. Value functions define a partial ordering over policies. A policy π is better than or equal to a policy π' if its expected return is greater than or equal to that of π' for all states. In other words, $\pi \geq \pi'$ if and only if $V^\pi(s) \geq V^{\pi'}(s)$ for all $s \in \mathcal{S}$. There is always at least one policy that is better than or equal to all other policies. This is an *optimal policy*. Although there may be more than one, we denote all the optimal policies by π^*. They share the same value function, called the *optimal value function* and denoted V^*, defined as

$$V^*(s) = V^{\pi^*}(s) = \max_\pi V^\pi(s), \tag{6}$$

for all $s \in \mathcal{S}$.

Optimal policies also share the same *optimal action–value function*, denoted Q^*, defined as

$$Q^*(s,a) = Q^{\pi^*}(s,a) = \max_\pi Q^\pi(s,a), \tag{7}$$

for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$. For state–action pair (s,a), this function gives the expected return for taking action a in state s and thereafter following an optimal policy. Thus, we can write Q^* in terms of V^* as

$$Q^*(s,a) = E\left\{r_{t+1} + \gamma V^*(s_{t+1}) \mid s_t = s, a_t = a\right\}. \tag{8}$$

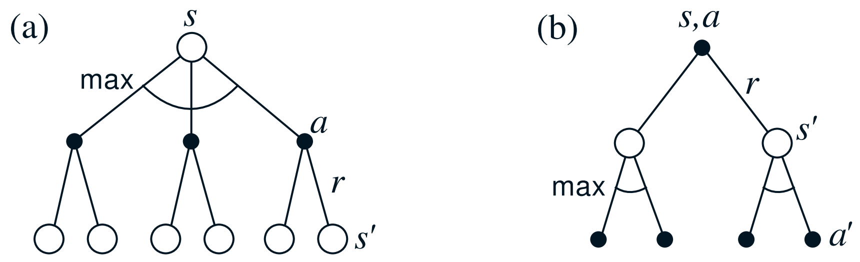


Figure 5: Backup diagrams for a) V^* and b) Q^*

Because V^* is the value function for a policy, it must satisfy the self-consistency condition given by the Bellman equation (5). Because it is the optimal value function, however, V^*'s consistency condition can be written in a special form, called the *Bellman optimality equation*, which is independent of the policy. Intuitively, the Bellman optimality equation is based on the fact that the value of a state under an optimal policy must equal the expected return for the best action from that state:

$$V^*(s) = \max_a E_{\pi^*}\{r_t \mid s_t = s, a_t = a\}. \tag{9}$$

From this follows the Bellman optimality equation for the discounted–reward case:

$$\begin{aligned}
V^*(s) &= \max_a E_{\pi^*}\left\{\sum_{k=0}^{\infty}\gamma^k r_{t+k+1} \,\middle|\, s_t = s, a_t = a\right\} \\
&= \max_a E_{\pi^*}\left\{r_{t+1} + \gamma\sum_{k=0}^{\infty}\gamma^k r_{t+k+2} \,\middle|\, s_t = s, a_t = a\right\} \\
&= \max_a E\left\{r_{t+1} + \gamma V^*(s_{t+1}) \mid s_t = s, a_t = a\right\} \\
&= \max_a \sum_{s'}\left[R^a_{ss'} + \gamma P^a_{ss'} V^*(s')\right] \qquad (10)
\end{aligned}$$

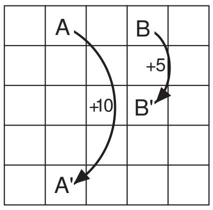
Figure 3 shows graphically the span of future states and actions considered in the Bellman optimality equations for V^* and Q^*. These are the same diagrams as for V^π and Q^π, except that an arc has been added at the agent's choice points to represent that the maximum over that choice is taken rather than the expected value given some policy. Figure 3a represents graphically the Bellman optimality equation (10).

For finite MDPs, the Bellman optimality equation (10) has a unique solution independent of the policy, unlike the ordinary consistency condition (5), whose solution depends on the policy. The Bellman optimality equation is actually a system of equations, one for each state, so that there are $|\mathcal{S}|$ equations in $|\mathcal{S}|$ unknowns. If the dynamics of the environment are known ($R^a_{ss'}$ and $P^a_{ss'}$), then in principle one can solve this system of equations for V^* using, for example, linear programming. One can solve a related set of equations for Q^*.

Once one has V^*, it is relatively easy to determine an optimal policy. For each state s, there will be one or more actions at which the maximum is obtained in the Bellman optimality equation (10): if a is such an action for time step t, then it maximizes $r_{t+1} + \gamma V^*(s_{t+1})$. This means that a is a best action to take *now*, assuming that you also take best actions thereafter. You can think of such an action as the result of a one–step ahead search from s_t in which all the possible successor states, s_{t+1}, are evaluated using V^*. All these maximizing actions are equally good optimal actions. Any policy that selects only from among these is an optimal policy.

Having Q^* makes choosing optimal actions still easier. With Q^*, the agent does not even have to do a one–step ahead search: for any state s, it can simply find any action which maximizes $Q^*(s, a)$. The action–value function effectively caches the results of all one–step ahead searches. It provides the optimal expected long–term return as a value that is locally and immediately available for each state–action pair. Hence, at the cost of representing a function of state–action pairs, instead of just of states, the optimal–action value function allows optimal actions to be selected without having to know anything about possible successor states and their values, that is, without having to know anything about the environment's dynamics.

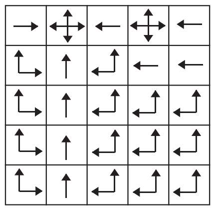
Example 7 Suppose we solve the Bellman optimality equation for the simple grid problem introduced in the previous example and shown again in Figure 4a. Recall that state A is followed by a reward of +10 and transition to state A', while state B is followed by a reward of +5 and transition to state B'. Figure 4b shows the optimal value function, and Figure 4c shows the corresponding optimal policies. Where there are multiple arrows in a cell, either action is optimal.



(a)

22.0	24.4	22.0	19.4	17.5
19.8	22.0	19.8	17.8	16.0
17.8	19.8	17.8	16.0	14.4
16.0	17.8	16.0	14.4	13.0
14.4	16.0	14.4	13.0	11.7

(b)



(c)

Figure 6: a) A gridworld; b) its V^*; and c) the π^*.

Explicitly solving the Bellman optimality equation provides one route for finding an optimal policy and thus to solving the reinforcement learning problem. However, this solution is rarely directly useful. It is akin to an exhaustive search, looking ahead at all possibilities, computing their probability of occurrence, and their desirability in terms of expected rewards. Such a solution relies on at least three assumptions that are never completely true in practice: 1) we accurately know the dynamics of the environment, 2) situations are actual states, i.e., the agent–environment interaction can be modeled as an MDP, and 3) we have enough computational resources to complete the computation of the solution. But for the kinds of problems in which we are interested, one is generally not able to implement this solution exactly because various combinations of these assumptions are violated. For example, although the first two assumptions present no problems for the game of backgammon, the third is a major impediment. Since the game has about 10^{20} states, it would take thousands of years on today's fastest computers to solve the Bellman optimality equation for V^*, and the same is true for finding Q^*. Unless there is some special additional mathematical structure that can be exploited, one has to settle for approximate solutions.

Many different decision–making methods can be viewed as ways of approximately solving the Bellman optimality equation. For example, heuristic search methods can be viewed as expanding the right–hand side of (10) several times, up to some depth, forming a "tree" of possibilities, and then using a heuristic evaluation function to approximate V^* at the "leaf" nodes. (Heuristic search methods such as A^* are almost always based on the total–reward case, e.g., when the rewards are negative costs.) The methods of dynamic programming can be related even more closely to the Bellman optimality equation. Many reinforcement learning methods can be clearly understood as approximately solving the Bellman optimality equation, using actual experienced transitions in place of knowledge of the expected transitions. We consider a variety of such methods in the following chapters.

Exercise 7 What is the Bellman equation—the analog of (10)—for the action-value function Q^*? The equation you come up with should correspond to Figure 3b.

Exercise 8 Figure 4 gives the optimal value of the best state of the gridworld as 24.4, to one decimal place. Use your knowledge of the optimal policy and (2) to compute this value symbolically, and then to three decimal places.

2 Dynamic Programming

Dynamic Programming is the name given to a collection of methods for computing optimal policies given a perfect model of the environment as a Markov decision task. These methods are not directly useful to us, as our primary goal is to find good policies without assuming a complete and accurate model, but they are still a useful guide, a theoretical foundation and touchstone. All reinforcement learning algorithms can in fact be viewed as the attempt to achieve much the same effects as dynamic programming only without assuming the availability of a perfect model.

A second problem with classical dynamic programming methods is that they involve exhaustive sweeps through the state space, making them intractable for large problems. Other reinforcement learning methods can sometimes greatly reduce the computation required before useful approximate solutions can be found.

2.1 Policy Evaluation

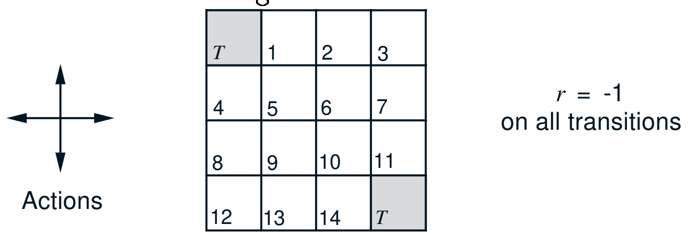
First we consider how to compute the state-value function V^π for an arbitrary policy, π. This is called *policy evaluation.*

As the environment is assumed known, (3) is a system of $|\mathcal{S}|$ simultaneous linear equations in $|\mathcal{S}|$ unknowns. In principle, its solution is a straightforward, if tedious, computation. For our purposes, iterative solutions are most suitable. Consider a sequence of approximate value functions $V_0, V_1, V_2, \ldots$, each mapping from $\mathcal{S}$ to $\Re$. The initial approximation, V_0, is chosen arbitrarily, and each successive approximation is obtained by using the Bellman equation for V^π as an update rule:

$$\begin{aligned} V_{k+1}(s) &= E_\pi\{r_{t+1} + \gamma V_k(s_{t+1}) \mid s_t = s\} \\ &= \sum_a \pi(s,a) \sum_{s'} P^a_{ss'}\left[R^a_{ss'} + \gamma V^\pi(s')\right] \end{aligned} \tag{11}$$

for all $s \in \mathcal{S}$. Clearly, $V_k = V^\pi$ is a fixed point for this update rule, as the Bellman equation (5) assures us of equality here. Indeed, the sequence $\{V_k\}$ can be shown in general to converge to V^π as $k \to \infty$ under the same conditions for the existence of V^π. This algorithm is called *iterative policy evaluation.*

Example 8 Consider the 4×4 gridworld shown below.



The nonterminal states are $\mathcal{S} = \{1, 2, \ldots, 14\}$. There are 4 actions possible in each state, $\mathcal{A} = \{up, down, right, left\}$, which deterministically cause the corresponding state transitions, except that actions that would take the agent off the grid have no effect. Thus, e.g., $P^{right}_{5,6} = 1$, $P^{right}_{5,10} = 0$, and $P^{right}_{7,7} = 1$. This is an undiscounted, finite-horizon task. The reward is -1 on all transitions until the terminal state is reached. The terminal state is labeled T in the figure (although it is shown in two places it is formally one state). The expected reward function is thus $R^a_{ss'} = -1$, $\forall a, s, s'$.

Suppose the agent follows the equiprobable random policy (all actions equally likely). The left side of Figure 2.1 shows the sequence of value functions $\{V_k\}$ computed by the iterative policy evaluation procedure (11). The final estimate is in fact V^π, which in this case is minus the expected number of steps until termination.

Exercise 9 If π is the random policy, what is $Q^\pi(11, down)$? What is $Q^\pi(7, down)$?

Exercise 10 Suppose a new state 15 is added to the gridworld just below state 13 whose actions, **`left`**, **`up`**, **`right`**, and **`down`**, take the agent to states 12, 13, 14, and 15 respectively. What then is $V^\pi(15)$ for the random policy?

2.2 Policy Improvement

Our reason for computing the value function for a policy is to help in finding better policies. Suppose we have determined the value function V^π for an arbitrary deterministic policy π. For some state s we would like to know whether or not we should change the policy to deterministically choose action $a \neq \pi(s)$. We know how good it is to follow the current policy in s—that is just $V^\pi(s)$—but would it be better or worse to change to the new policy? One way to answer this question is to consider selecting a in s and then thereafter following the existing policy π. The value of that way of behaving is

$$\begin{aligned} Q^\pi(s,a) &= E_\pi\{r_{t+1} + \gamma V^\pi(s_{t+1}) \mid s_t = s, a_t = a\} \\ &= \sum_{s'} P^a_{ss'}\left[R^a_{ss'} + \gamma V^\pi(s')\right] \end{aligned} \tag{12}$$

The key criterion is whether this is greater or less than $V^\pi(s)$. If it is greater, that is, if it is better to select a once in s and thereafter follow π than it would be to follow π all the time, then one would expect it to be better still to select a every time s is encountered, and that the new policy would in fact be a better one overall.

That this is true is a special case of a general result, the *policy improvement theorem.* Let π and π' be any pair of deterministic policies such that, for all $s \in \mathcal{S}$,

$$Q^\pi(s, \pi'(s)) \geq V^\pi(s). \tag{13}$$

Then the policy π' must be uniformly better than π. That is, it must obtain greater expected reward from all states $s \in \mathcal{S}$:

$$V^{\pi'}(s) \geq V^\pi(s).$$

This result applies in particular to the two policies that we considered in the previous paragraph, which differed at only one state, s. At s, one policy or the other satisfies the inequality (13), and at all other states equality holds. Thus, if $Q^\pi(s,a) \geq V^\pi(s)$, then the changed policy is indeed better.

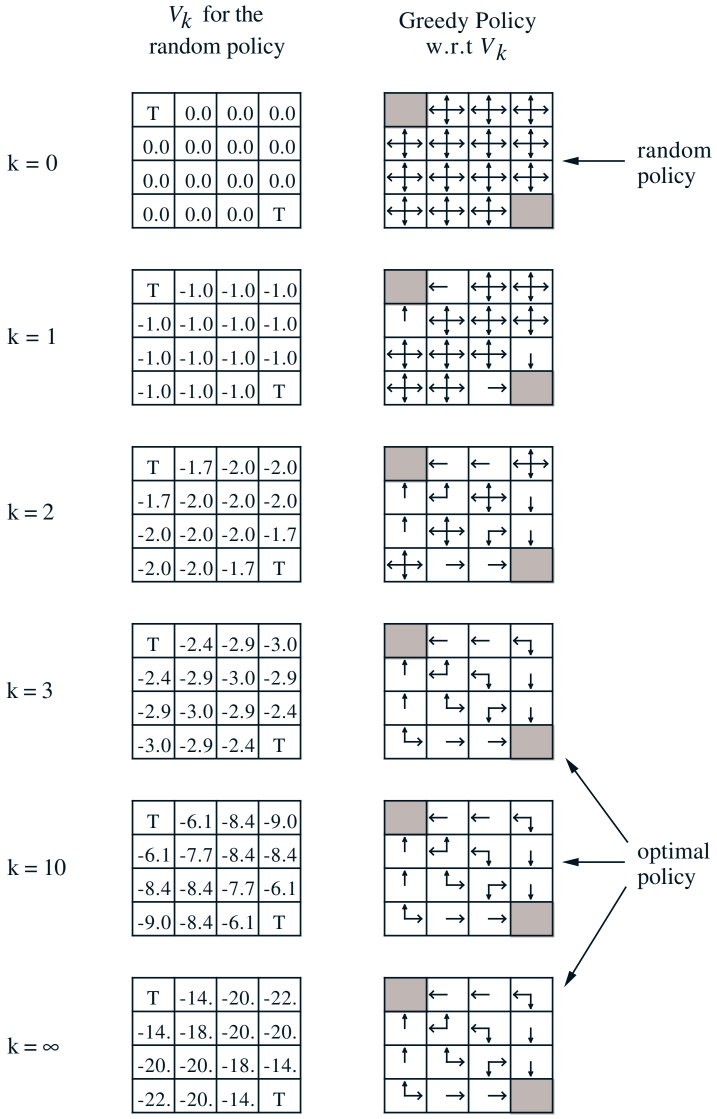


Figure 7: Convergence of iterative policy evaluation on a small gridworld. The lefthand column is the sequence of approximations to the value function for the random policy (all actions equal). The righthand side is the sequence of greedy policies corresponding to the value function estimates (arrows shown for all actions achieving the maximum). The last policy is guaranteed only to be an improvement over the random policy, but in this case it and all policies after the third iteration are optimal.

The idea behind the proof of the policy improvement theorem is easy to understand. Let $s_1, s_2, s_3, \ldots$ denote the random variables corresponding to the states following an arbitrary state $s \in \mathcal{S}$, assuming actions are selected according to policy π'. Let $r_1, r_2, \ldots$ similarly denote the random variables corresponding to the rewards received upon arrival in those states. Then

$$
\begin{aligned}
V^\pi(s) &\leq Q^\pi(s, \pi'(s)) \\
&= E\left\{r_1 + \gamma V^\pi(s_1)\right\} \\
&\leq E\Big\{r_1 + \gamma E\left\{r_2 + \gamma V^\pi(s_2) \mid s_1\right\}\Big\} \\
&\leq E\Big\{r_1 + \gamma r_2 + \gamma^2 V^\pi(s_2)\Big\} \\
&\vdots \\
&\leq E\Big\{r_1 + \gamma r_2 + \gamma^2 r_3 + \cdots + \gamma^k V^\pi(s_k)\Big\} \\
&\vdots \\
&\leq V^{\pi'}(s)
\end{aligned}
$$

Q.E.D.

2.3 Policy Iteration

In the preceding section we have seen how, given a policy and its value function, we can easily evaluate a change in the policy at a single state to a particular action. It is a natural extension to consider changes at *all* states and *all* possible actions, selecting at each state the action that appears best according to $Q^\pi(s, a)$ (12). In other words, to consider the new deterministic policy π' given by

$$
\begin{aligned}
\pi'(s) &= \arg\max_a Q^\pi(s, a) \\
&= \arg\max_a E\left\{r_{t+1} + \gamma V^\pi(s_{t+1}) \mid s_t = s, a_t = a\right\} \\
&= \arg\max_a \sum_{s'} P^a_{ss'} \left[R^a_{ss'} + \gamma V^\pi(s')\right]
\end{aligned}
\tag{14}
$$

where the "$\arg\max$" notation denotes the value of a at which the expression is maximized (with ties broken arbitrarily). The new policy π' is called the *greedy* policy with respect to V^π because it takes the action that looks best in the short term—after one step of lookahead—according to V^π. By construction, the new greedy policy meets the conditions of the policy improvement theorem (13), so we know that it is as good or better than the original policy.

Suppose the new policy π' is only as good as the old policy π. Then from (14) it follows that for all $s \in \mathcal{S}$:

$$
V^{\pi'}(s) = \max_a E\left\{r_{t+1} + \gamma V^\pi(s_{t+1}) \mid s_t = s, a_t = a\right\}
$$

$$= \max_a \sum_{s'} P^a_{ss'} \left[R^a_{ss'} + \gamma V^\pi(s') \right]$$

But $V^\pi = V^{\pi'}$, and therefore this equation is the Bellman equation (10), and therefore V^π must be V^*, and π and π' must be optimal policies. Thus, policy improvement must give us a strictly better policy except in the case that the policy is already optimal.

Once π has been improved using V^π to yield π', we can then compute $V^{\pi'}$ and improve again to yield an even better π''. We can thus obtain an alternating sequence of monotonically improving policies and value functions:

$$\pi_0 \xrightarrow{\text{PE}} V^{\pi_0} \xrightarrow{\text{PI}} \pi_1 \xrightarrow{\text{PE}} V^{\pi_1} \xrightarrow{\text{PI}} \pi_2 \xrightarrow{\text{PE}} \cdots \xrightarrow{\text{PI}} \pi^* \xrightarrow{\text{PE}} V^*$$

where $\xrightarrow{\text{PE}}$ denotes a policy evaluation and $\xrightarrow{\text{PI}}$ denotes a policy improvement. If $\mathcal{S}$ is finite, then the sequence must converge to the optimal policy and value function in a finite number of iterations. The computation needed for each policy evaluation is greatly reduced if $V^{\pi_{k-1}}$ is used as the initial estimate V_0 of V^{π_k} (in (11)). A version of the complete algorithm is given in Figure 6.

Policy iteration often converges in surprisingly few steps. This is illustrated by the simple example introduced in the preceding section. The last diagram on the left in Figure 2.1 shows the value function for the random policy, and the last column on the right shows the greedy policy for this value function. The policy improvement theorem assures us only that this policy will be better than the original random policy. In this case, however, this policy is actually optimal, proceeding to the terminal states in the minimum number of steps. In this example policy iteration finds the optimal policy after only one iteration.

Exercise 11 How would policy iteration be defined for action values? Give a complete algorithm for computing Q^*, analogous to Figure 6 for computing V^*.

Exercise 12 Suppose you were restricted to consideration only of policies that are *ϵ-soft*, meaning that the probability of selecting each action in each state, s, was at least $\frac{\epsilon}{|\mathcal{A}(s)|}$. Describe qualitatively the changes that would be required in each of the steps 3, 2, and 1, in that order, of the policy iteration algorithm for V^* (Figure 6).

2.4 Value Iteration

One drawback to policy iteration is that each of its iterations involves policy evaluation, which may itself be a protracted iterative computation. If policy evaluation is done iteratively, as in (11), then convergence exactly to V^π occurs only in the limit. Must we wait until policy evaluation is complete, or can we stop short of that? The example in Figure 2.1 certainly suggests that it may be possible to truncate policy evaluation: after just three iterations of policy evaluation the corresponding greedy policy no longer changes.

1. Initialization

    $\pi :=$ an arbitrary deterministic policy
    $V :=$ an arbitrary function: $\mathcal{S} \mapsto \Re$
    $\theta :=$ small positive number

2. Policy Evaluation

    Repeat
        $\Delta := 0$
        For each $s \in \mathcal{S}$:
            $v := V(s)$
            $V(s) := \sum_{s'} P_{ss'}^{\pi(s)} \left[R_{ss'}^{\pi(s)} + \gamma V(s') \right]$
            $\Delta := \max(\Delta, |v - V(s)|)$
    until $\Delta < \theta$

3. Policy Improvement

    policy-stable := *true*
    For each $s \in \mathcal{S}$:
        $b := \pi(s)$
        $\pi(s) := \arg\max_a \sum_{s'} P_{ss'}^{a} \left[R_{ss'}^{a} + \gamma V(s') \right]$
        If $b \neq \pi(s)$ then *policy-stable* := *false*
    If *policy-stable*, then stop
                else go to 2

Figure 8: Policy Iteration (in place) for V^*

$V :=$ an arbitrary function:$\mathcal{S} \mapsto \Re$
$\pi :=$ an arbitrary deterministic policy
Repeat
  policy-stable $:=$ *true*
  For each $s \in \mathcal{S}$:
    $b := \pi(s)$
    $V(s) := \max_a \sum_{s'} P^a_{ss'} \left[R^a_{ss'} + \gamma V(s') \right]$
    $\pi(s) :=$ the a at which the above max is obtained
    If $b \neq \pi(s)$ then *policy-stable* $:=$ *false*
until *policy-stable*

Figure 9: Value Iteration (in place) for V^*

An important special case of truncated policy evaluation is that in which it proceeds for only one iteration. This algorithm is known as *value iteration.* It can be written as a particularly simple update rule:

$$\begin{aligned} V_{k+1}(s) &= \max_a E\left\{ r_{t+1} + \gamma V_k(s_{t+1}) \mid s_t = s, a_t = a \right\} \\ &= \max_a \sum_{s'} P^a_{ss'} \left[R^a_{ss'} + \gamma V_k(s') \right] \end{aligned} \tag{15}$$

for all $s \in \mathcal{S}$. For arbitrary V_0, the sequence $\{V_k\}$ can be shown to converge to V^* under the same conditions for the existence of V^* (e.g., see Ross, 1983). Note that this algorithm for computing V^* is obtained by literally converting the Bellman equation for V^* (10) into an update rule. A complete explicit algorithm for a version of value iteration is given in Figure 7.

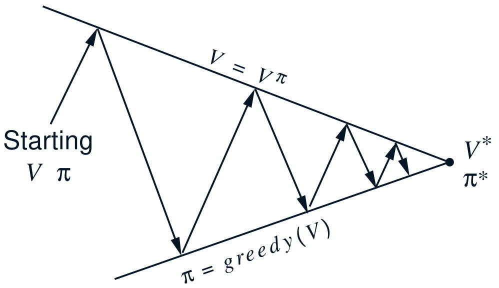
Exercise 13 What is the analog of the value iteration equation (15) for action values, $Q_{k+1}(s, a)$?

2.5 Bootstrapping and Co-Convergence

Policy iteration and value iteration are the two most important methods of dynamic programming. What are the key ideas underlying these methods? We would identify two. One is the idea of looking ahead along a state transition and using the current value estimates at the ending states to improve the value estimate for the starting state. Thus, estimates are built on the basis of other estimates. Let us call this the *bootstrapping* property. This property will play a key role in temporal-difference learning as well.

The second key idea underlying both policy iteration and value iteration is that of two simultaneous, interacting processes, one computing the value function for the current policy and one trying to make the policy greedy with respect to the current value function. In policy iteration these two steps are done alternately, completing each before going on to the other, whereas in value iteration only a single iteration of policy evaluation is performed between each policy improvement step. Over many such steps, however, the overall result is the same—convergence onto the optimal value function and an optimal policy. It is easy to see that if both goals have been achieved, then the value function and policy must be optimal. The consistency of the value function means that (3) holds, and the greediness of the policy implies that the expected value in that equation is in fact the maximum in the Bellman equation (10). Since the optimal value function is unique as a solution to the Bellman equation, the value function must be optimal, and thus the policy must be optimal as well.

There is both competition and cooperation in the convergence of the value function to consistency with the policy and of the policy to greediness with respect to the value function. They compete in the sense that in the short term they pull in opposing directions. Making the policy greedy with respect to the value function typically makes the value function incorrect for the changed policy, and making the value function consistent with the policy typically causes that policy to no longer be greedy. In the long run, however, these two convergence processes interact to find a single joint solution at the optimal policy and optimal value function. We like to think of the interaction in terms of two constraints, e.g., as two lines in 2-space:



where at each step the system is driven towards one of the lines representing a solution to one of the two goals. The two goals interact because the two lines are not orthogonal. Thus, driving directly towards one causes the other to be violated. Inevitably, however, the joint process is brought closer and closer to the overall goal. (This diagram is most suggestive of policy iteration in the way each step takes the system all the way to achieving one of the two goals completely on each step.) Each of the two processes is trying to achieve its own goal, neither of which is finding optimality. Only together do they achieve this. We use the term *co-convergence* for this way of converging onto an overall goal via the interaction of two simultaneous convergence processes.

3 Monte Carlo Methods

In this section we consider our first *learning* methods for estimating value functions and discovering optimal policies. Unlike the dynamic programming section, in this section we do not assume that we have a complete and accurate model of the environment. Instead, we only observe the sequence of states, actions and rewards generated by the environment and agent over some period of time. If a model is used, it is only as a substitute way of generating such sequences.

Recall that the value of a state is defined as the expected return (expected total future discounted reward) from that state. An obvious way to estimate it is to average the returns observed after visits to that state. As more and more returns are observed, the average should converge to the expected value. This is the idea of what we call *Monte Carlo* (MC) methods.

The name "Monte Carlo" is sometimes used more broadly for any estimation method whose operation involves a significant random component. Here we use it specifically for methods based on averaging complete, observed returns (as opposed to methods that learn from partial returns, considered in the next section). To ensure that complete returns are always available, we define Monte Carlo methods only for the finite-horizon (trial-based) case. That is, we assume experience is divided into trials and that all trials eventually terminate no matter what actions are selected. It is only upon the completion of a trial that value estimates and policies are changed.

4 Temporal-Difference Learning

If one had to identify one idea as central and novel to reinforcement learning, it would undoubtedly be that which we introduce in this section—*temporal difference* (TD) learning. TD learning is a combination of Monte Carlo (MC) ideas and dynamic programming (DP) ideas. Like MC, TD methods can learn directly from raw experience without requiring a model. Like DP, TD methods update their estimates based in part on other learned estimates, without waiting for a final outcome.

The relationships between TD, MC, and DP approaches is a recurring theme throughout reinforcement learning. This section is just the beginning of our exploration of this theme. Before we are done, we will see that these ideas and methods blend into each other and can be combined in many ways. In particular, TD and Monte Carlo methods can be interrelated and seamlessly combined in the TD(λ) algorithm.

4.1 TD Prediction vis-à-vis MC and TD

We start by considering simple TD methods for policy evaluation, that is, for predicting $V^{\pi}(s)$, the expected return after visiting state s when following policy π. Both TD and Monte Carlo methods use actual experience to form the estimates. That is, if state s is visited at time t, then both estimates update their estimate $V_t(s)$ based on what actually happens after that visit. Roughly speaking, Monte Carlo methods wait until the return following the visit is known, then use that return as a target for $V_t(s_t)$. A simple Monte Carlo update rule for nonstationary environments is

$$\Delta V_t(s_t) = \alpha\Big[R_t - V_t(s_t)\Big], \tag{16}$$

where $\Delta V_t(s_t)$ is the increment to $V_t(s_t)$ due to visiting s_t at time t, R_t is the actual return following time t and α is a step-size parameter. Whereas Monte Carlo methods wait until the end of the trial to determine the increment, TD methods wait only until the next time step. At time $t+1$ they immediately attempt to form a target and make a useful update, using the observed reward r_{t+1} and the estimate $V_t(s_{t+1})$. The simplest TD update rule is

$$\Delta V_t(s_t) = \alpha\Big[r_{t+1} + \gamma V_t(s_{t+1}) - V_t(s_t)\Big]. \tag{17}$$

In effect, the target for the Monte Carlo update is R_t, whereas for the TD update it is $r_{t+1} + \gamma V_t(s_{t+1})$.

Because the TD method bases its update in part on an existing estimate, we say that it is a *bootstrapping* method, like DP. We know from sections 2 and 4 that

$$\begin{aligned}
V^{\pi}(s) &= E_{\pi}\left\{R_t \mid s_t = s\right\} && (18)\\
&= E_{\pi}\left\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \,\middle|\, s_t = s\right\} \\
&= E_{\pi}\left\{r_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k r_{t+k+2} \,\middle|\, s_t = s\right\} \\
&= E_{\pi}\left\{r_{t+1} + \gamma V^{\pi}(s_{t+1}) \,\middle|\, s_t = s\right\} && (19)
\end{aligned}$$

Roughly speaking, MC methods use an estimate of (18) as a target, whereas DP methods use an estimate of (19) as a target. The MC target is an estimate because the expected value in (18) is not known; a sample return is used in place of the real expected return. The DP target is not an estimate because of the expected values, as these are assuming completely provided by a model of the environment. Rather, the DP target is an estimate because $V^{\pi}(s_{t+1})$ is not known and the current estimate, $V_t(s_{t+1})$, is used instead. The TD target is an estimate for both reasons: it samples the expected values in (19) *and* it uses the current

Figure 10: The backup diagram for the TD(0) algorithm.

estimate V_t instead of the true V^π. Thus TD methods combine the sampling of Monte Carlo methods with the bootstrapping of DP methods. As we shall see, with care and imagination this can take us a long way toward obtaining all the advantages of both MC and DP methods.

Figure 8 shows the backup diagram for the simple TD rule given above. The value estimate for the state node at the top is updated from the one sample transition from it to the immediately following state.

Example 9 *Driving Home.* Each day you drive home from work and you try to predict how long it will take you to get home. When you first leave your office you note the time and the day of week and anything else that might be relevant. Say on this friday you are leaving at exactly 6 o'clock, and you estimate it will take you 30 minutes to get home. As you reach your car it is 6:05 and you notice it is starting to rain. Traffic is often slower in the rain so you re-estimate that it will take 35 minutes from then, or a total of 40 minutes. Fifteen minutes later you have completed the highway portion of your journey in good time. As you exit onto a secondary road you cut your estimate of total travel time to 35 minutes. Unfortunately, at this point you get stuck behind a slow truck, and the road is too narrow to pass. You end up having to follow the truck until you turn off onto the side street where you live at 6:40. Three minutes later you are home. The sequence of situations, times, and predictions is thus something like this:

situation	*elapsed time (minutes)*	*predicted time-to-go*	*predicted total time*
leaving office, friday at 6	0	30	30
reach car, raining	5	35	40
exiting highway	20	15	35
2ndary road, behind truck	30	10	40
entering home street	40	3	43
arrive home	43	0	43

The rewards in this example are the elapsed times on each leg of the journey. We are not discounting ($\gamma = 1$) and thus the return for each situation is the actual time-to-go from that situation. The value of each situation is the *expected* time-to-go. The second column of numbers gives the current estimated value for each situation encountered.

A simple way to view the operation of Monte Carlo methods is to plot the

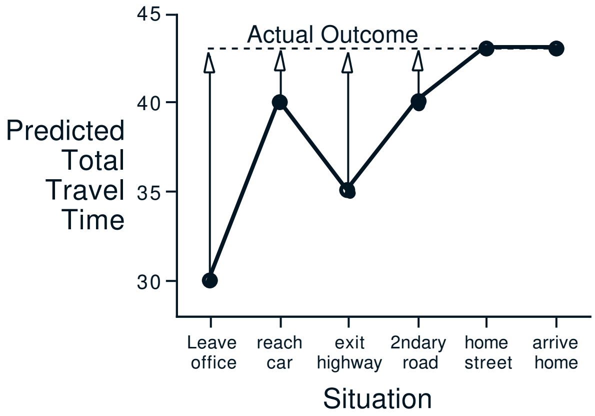


Figure 11: Changes recommended by Monte Carlo methods in the driving-home example.

predicted total time (the last column) over the sequence, as in Figure 9. The arrows show the changes in predictions recommended by the Monte Carlo rule (16). These are exactly the errors between the estimated value (predicted time-to-go) in each situation and the actual return (actual time-to-go). For example, when we exited the highway we thought it would take only 15 minutes more to get home, but in fact it took 23. Equation 9 applies at this point and determines an increment in our estimate of time-to-go after exiting the highway. The error, $R_t - V_t(s_t)$, at this time is 8 minutes. Suppose the step size α is 1/2. Then the predicted time-to-go after exiting the highway would be revised upward by 4 minutes as a result of this experience. This is probably too large of a change in this case; the truck was probably just an unlucky break. In any event, the change can only be made offline, that is, after you have reached home. Only at this point do you know any of the actual returns.

Is it necessary to wait until the final outcome is known before learning can begin? Suppose on another day you again estimate when leaving your office that it will take you 30 minutes to drive home, but then you become stuck in a massive traffic jam. Twenty-five minutes after leaving the office you are still bumper-to-bumper on the highway. You now estimate that it will take you another 25 minutes, for a total of 50 minutes, to get home. As you wait in traffic you already know that your initial estimate of 30 minutes was too optimistic. Must you wait until you get home before increasing your estimate for the initial situation? According to the Monte Carlo approach you must, because you don't yet know the true return.

According to a TD approach, on the other hand, you would learn immediately, shifting your initial estimate from 30 minutes toward 50. In fact, each estimate would be shifted toward the estimate that immediately follows it. Returning to our first day of driving, Figure 10 shows the same predictions as Figure 9

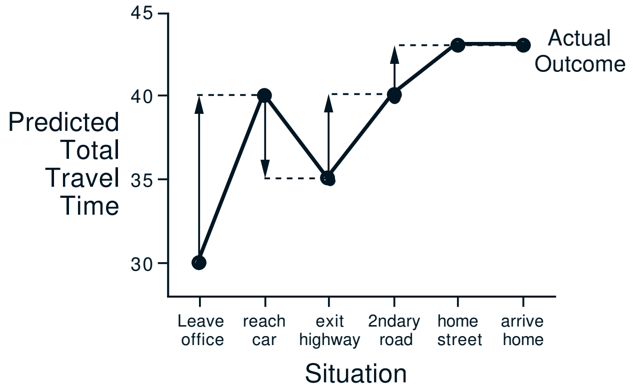


Figure 12: Changes recommended by TD methods in the driving-home example.

except here showing the changes recommended by the TD rule (17). Each error is proportional to the change over time of the prediction, i.e., learning is driven by the *temporal differences* in prediction.

Besides giving you something to do while waiting in traffic, there are several computational reasons why it is advantageous to learn based in part on your current predictions rather than waiting until termination when you know the actual return. We briefly discuss some of these next.

4.2 Advantages of TD Prediction Methods

TD methods learn their estimates in part on the basis of other estimates, it learns a guess from a guess. Is this a good thing to do? Developing and answering such questions is beyond the scope of these notes, but we can anticipate at least some of the answers.

Obviously, TD methods have an advantage over DP methods in that they do not require a model of the environment, of its reward and next-state probability distributions.

The most obvious advantage of TD methods over Monte Carlo methods is that they are naturally implemented in an online, fully incremental fashion. With Monte Carlo methods one must wait until the end of a trial, as only then is the return known, whereas with TD methods one need only wait one time step. Surprisingly often this turns out to be a critical consideration. Some applications have very long trials, such that delaying all learning until the trial ends is just too slow. Other applications have no natural sense of trial at all, but just continue without end. Finally, as we saw in the previous section, with MC methods one must often ignore or discount trials on which experimental actions are taken, which can greatly slow learning. We will see that TD methods, because they learn fully from each transition regardless of what subsequent actions are taken, are much less susceptible to this problem.

But are TD methods sound? Certainly it is convenient to learn one guess from the next, without waiting for an actual outcome, but can we still guarantee convergence to the correct answer? Happily, the answer is yes. For any fixed policy π, the TD algorithm described above has been proven convergent to V^π, in the mean for sufficiently small step size, and with probability one for step size decreasing according to the usual stochastic approximation conditions. Most convergence proofs apply only to the tabular case of the algorithm presented above (17), but some also apply to the case of general linear function approximators.

If both TD and MC methods converge asymptotically to the same predictions, then a natural next question is "Which gets there first?" In other words, which method learns fastest, makes the most efficient use of limited data? At the current time this is an open question in that no one has been able to mathematically prove one method converges faster than the other. In is not even clear what is the most appropriate formal way to phrase this question! In practice, however, TD methods have usually been found to converge must faster than MC methods.

Exercise 14 This is a exercise to help develop your intuitions about why TD methods are often more efficient than MC methods. Consider the driving home example and how it is addressed by TD and MC methods. Can you imagine a scenario in which a TD update would be better on average than an MC update? Give an example scenario—a description of past experience and a current situation—in which you would expect the TD update to be better. Here's a hint: Suppose you have lots and lots of experience driving home from work. Then you move to a new building and a new parking lot (but you still enter the highway at the same place). Now you are starting to learn predictions for the new building. Can you see why TD updates are likely to be much better, at least initially, in this case? Might the same sort of thing happen in the original problem?

5 Conclusion: A Unified View

So far we have discussed three classes of methods for solving the reinforcement learning problem: dynamic programming, Monte Carlo methods, and temporal-difference learning. Although each of these are different, they are not really alternatives in the sense that one must pick one or the other. It is perfectly sensible and often desirable to apply several of the approaches at once, to intermix and interrelate them. In fact, in our view that it is best not even to see them as alternative computations, but simply as different components of one algorithm. For different problems or different parts of one problem one may want to emphasize kind of computing over another, but ideally all should be present.