Reinforcement Learning for Neuroscientists

Seoul National University College of Medicine
Younghoon Kim (B.S.)
aktivhoon.github.io

1 Markov Decision Process

1.1 Agent-Environment Interface

MDPs are meant to be straight forward framing of the problem of learning from interaction to achieve a goal. The learner and decision maker is called the *agent*. The thing it interacts with, comprising everything outside the agent, is called the *environment*. These interact continually, the agent selecting actions and the environment responding to these actions and presenting new situations to the agent. The environment also gives rise to rewards, special numerical values that the agent seeks to maximize over time through its choice of actions.
Being more specific, the agent and environment interact at each sequence of discrete time steps, $t = 0, 1, 2, 3, \ldots, N$. At each time step t, the agent receives some representation of the environment's *state*, $S_t \in \mathcal{S}$, and on that basis selects an *action*, $A_t \in \mathcal{A}$. One time step later, in part as a consequence of its action, the agent receives a numerical *reward*, $R_{t+1} \in \mathcal{R} \subset \mathbb{R}$, and finds itself in a new state, S_{t+1}. The MDP and agent together thereby give rise to a sequence or *trajectory* that begins like this:

$$S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3, \ldots \tag{1}$$

1.2 Goals and Rewards

In reinforcement learning, the purpose or goal of the agent is formalized in terms of a special signal, called the *reward*, passing from the environment to the agent. At each time step, the reward is a simple number, $R_t \in \mathbb{R}$. Informally, the agent's goal is to **maximize the total amount of reward it receives**. This means maximizing not immediate reward, but cumulative reward in the long run.

1.3 Returns and episodes

So far we have said that the agent's goal is to maximize the cumulative reward it receives in the long run. How might this be defined formally? Let's say the sequence of rewards received after time step t is denoted R_{t+1}, R_{t+2}, R_{t+3}, ... , then what precise aspect of this sequence do we wish to maximize? In general, we week to maximize the *expected return*, where the return, denoted G_t, is defined as below:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{T} \gamma^k R_{t+k+1} = R_{t+1} + \gamma G_{t+1} \tag{2}$$

where γ is a parameter, $0 \leq \gamma \leq 1$, called the *discount rate*. Case where T is ∞, the task is a continuing task. Otherwise, we can consider the task to be a set of episodes.

1.4 Policies and Value Functions

Almost all reinforcement learning algorithms involve estimating *value functions* - functions of states that estimate **how good it is for the agent to be in a given state**. the notion of "how good" here is defined in terms of future rewards that can be expected, or, to be precise, in terms of expected return. Of course the rewards the agent can expect to receive in the future **depend on what actions it will take**. Accordingly, value functions are defined with respect to particular ways of acting, called policies.
A *policy* is a mapping from states to probabilities of selecting each possible action. If the agent is following policy π at time t, the $\pi(a|s)$ is the probability that $A_t = a$ if $S_t = s$. Reinforcement learning methods specify how the agent's policy is changed as a result of its experience.
The *value* of a state s under a policy π, denoted $v_\pi(s)$, is the expected return when starting in s and following π thereafter. For MDPs, we can define v_π formally by

$$v_\pi(s) \doteq \mathbb{E}_\pi \left[G_t | S_t = s\right] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \middle| S_t = s\right], \text{for all s} \in \mathcal{S} \tag{3}$$

where $\mathbb{E}_\pi[\cdot]$ denotes the expected value of a random variable given that the agent follows policy π, and t is any time step. Here, we call the function v_π the ***state-value function for policy*** π.
Similarly, we define the 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:

$$q_\pi(s, a) \doteq \mathbb{E}_\pi \left[G_t | S_t = s, A_t = a\right] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \middle| S_t = s, A_t = a\right] \tag{4}$$

We call the function q_π the ***action-value function for policy*** π.
Using property of Markov Decision Process, we can re-express state-value function as below:

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

We call this equation as the *Bellman equation for* v_π.

This expresses a relationship between the value of a state and the values of its successor states. Think of looking ahead from a state to its possible successor states, as suggested by the diagram above. Each open circle represents a state and each solid circle represents a state-action pair. Starting from state s, the root node at the top, the agent could take any of some set of actions– three are shown in the diagram–based on its policy π. From each of these, the environment could respond with one of several next states, s' (two are shown in the figure), along with a reward, r, depending on its dynamics given by the function p. **The Bellman equation averages over all the possibilities, weighting each by its probability of occurring.** It states that the value of

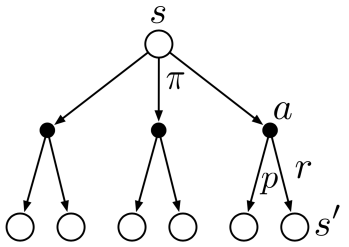


Figure 1: Backup diagram form v_π

the start state must equal the discounted value of the expected next state, plus the reward expected along the way.
The value function v_π is the unique to its Bellman equation. We show in subsequent chapters how this Bellman equation forms the basis of a number of ways to compute, approximate, and learn v_π. We call diagrams like that above *back up diagrams* because they diagram relationships that form the basis of the update or backup operations that are the heart of reinforcement learning methods.

1.5 Optimal Policies and Optimal Value Functions

Solving a reinforcement learning task means, roughly, finding a policy that achieves a lot of reward over the long run. For finite MDPs, we can precisely define an optimal policy in the following way. Value functions define a partial ordering over policies. A policy π is defined to be 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***. For optimal policy π_*, we can define *optimal state-value function*, denoted v_*, and defined as

$$v_*(s) \doteq \max_\pi v_\pi(s), \tag{5}$$

for all $s \in \mathcal{S}$.
Optimal policies also share the same ***optimal action-value function***, denoted q_*, and defined as

$$q_*(s,a) \doteq \max_\pi q_\pi(s,a), \tag{6}$$

for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$. For the state-action pair (s, a), this function gives the expected return for taking action a in state s and thereafter folowing an optimal policy. Thus, we can write q_* in terms of v_* as follows:

$$q_*(s,a) = \mathbb{E}\left[R_{t+1} + \gamma v_*(S_{t+1}) | S_t = s, A_t = a\right] \tag{7}$$

Remember v_* is a value function, thus must satisfy the self-consistency condition given by the Bellman equation for state values. This is the Bellman equation for v_*, or the ***Bellman optimality***

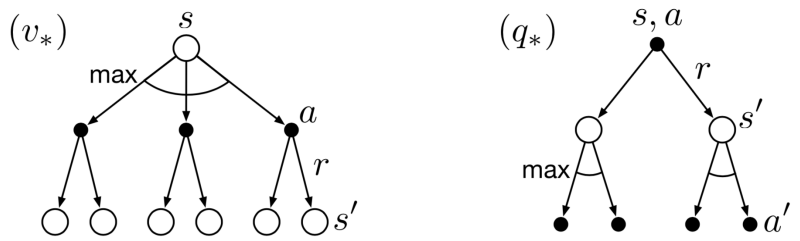


Figure 2: Backup diagram for v_* and q_*

equation.

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

The last two equations are two forms of the Bellman optimality equation for v_*. Likewise, the Bellman optimality equation for q_* is

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

The backup diagrams in Figure 2 show graphically the spans of future states and actions considered in the Bellman optimality equations for v_* and q_*. These are the same as the backup diagrams for v_π and q_π presented earlier except that arcs have been added at the agent's choice points to represent that the maximum over the choice is taken rather than the expected value Given some policy. The backup diagram on the left graphically represents the Bellman optimality equation v_* and the backup diagram on the right graphically represents q_*.

For finite MDPs, the Bellman equation for v_π has a unique solution independent of the policy. The Bellman optimality equation is a system of equations, one for each state, so if there are n states, then there are n equations in n unknowns. If the dynamics p of the of the environment are known, then in principle one can solve this system of equations for v_* using any one of a variety of methods for solving systems of nonlinear equations. 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 optimatility equation. Any policy that assigns nonzero probability only to these actions in an optimal policy. You can think of this as one-step search. If you have the optimal value function, v_*, then the actions that appear best after a one-step search will be optimal actions. Another way of saying this is that any policy that is *greedy* with respect to the optimal evaluation v_* is an optimal policy. The term greedy is used in computer science to describe any search of decision procedure that selects alternatives based only on local or immediate considerations, without considering the possibility that such a

selection may prevent future access to even better alternatives. Consequently, it describes policies that select actions based only on their short-term consequences. The beauty of v_* is that if one uses it to evaluate the short-term consequences of actions – specifically, the one-step consequences – then a greedy policy is actually optimal in the long-term sense in which we are interested because v_* already takes into account the reward consequences of all possible future behavior. By means of v_*, the optimal expected long-term return is turned into a quantity that is locally and immediately available for each state. Hence, a one-step-ahead search yields the long-term optimal actions.
Having q_* makes choosing optimal actions even 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 that maximized $q_*(s, a)$. 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 sates and their values, that is, **without having to know anything about the environment's dynamics**.

Explicitly solving the Bellman optimality equation provides one route to 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 of occurrence and their desirabilities in terms of expected rewards. This solution relies on at least **three** assumptions that are rarely true in practice: **(1) we accurately know the dynamics of the environment**; **(2) we have enough computational resources to complete the computation of the solution**; and **(3) the Markov property**. For the kinds of tasks 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 and third assumptions present no problems for the game of backgammon, the second 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 equation for v_*, and the same is true for finding q_*. In reinforcement learning one typically has to settle for approximate solutions.

2 Dynamic Programming* (Optional)

The term dynamic programming (DP) refers to a collection of algorithms that can be used to compute optimal policies given a perfect model of the environment as a Markov decision process (MDP). However, classical DP algorithms are of limited utility in reinforcement learning both because of their assumption of a perfect model and because of their great computational expense. Here, we will assume that the we usually assume that the environment is a finite MDP. That is, we assume that its state, action, and reward sets, $\mathcal{S}$, $\mathcal{A}$, and $\mathcal{R}$, are finite, and that its dynamics are given by a set of probabilities $p(s', r|s, a)$, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$, $r \in \mathcal{R}$, and $s' \in \mathcal{S}^+$ ($\mathcal{S}^+$ is $\mathcal{S}$ plus a terminal state if the problem is episodic). The key idea of DP, and of reinforcement learning generally, is the use of value functions to organize and structure the search for good policies. We can easily obtain policies once we have found the optimal value functions, v_* or q_*, which satisfy the Bellman optimality equations:

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

or

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

for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$, and $s' \in \mathcal{S}^+$. As we shall see, DP algorithms are obtained by turning Bellman equations such as these into assignments, that is, into update rules for improving approximations of the desired value functions.

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* in the DP literature. Recall that, for all $s \in \mathcal{S}$,

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

where $\pi(a|s)$ is the probability of taking action a in state s under poliy π, and the expectations are subscripted by π to indicate they are conditional on π being followed. The existence and uniqueness of v_π are guaranteed as long as either $\gamma < 1$ or eventual termination is guaranteed from all states under the policy π.

If the environment's dynamics are completely known, then the case is a system of $|\mathcal{S}|$ simultaneous linear equations in $|\mathcal{S}|$ unknowns (the $v_\pi(s), s \in \mathcal{S}$). Consider a sequence of approximate value

functions $v_0, v_1, v_2, \cdots$, each mapping $\mathcal{S}^+$ to $\mathbb{R}$ (the real numbers). The inital approximation, v_0, is chosen arbitrarily (except that the terminal state, if any, must be given value 0), and each successive approximation is obtained by using the Bellman equation for v_π as an update rule:

$$\begin{aligned} v_{k+1}(s) &\doteq \mathbb{E}_\pi \left[R_{t+1} + \gamma v_k \left(S_{t+1} \right) | S_t = s \right] \\ &= \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a) \left[r + \gamma v_k \left(s' \right) \right] \end{aligned}$$

for all $s \in \mathcal{S}$. Indeed, the sequence $\{v\}$ can be shown in general to converge to v_π as $k \to \infty$ under the same conditions that guarantee the existence of v_π. This algorithm is called *iterative policy evaluation*.

To produce each successive approximation, v_{k+1} from v_k, iteative policy evaluation applies the same operation to each state s, and the expected immediate rewards, along all the one-step transitions possible under the policy being evaluated. We call this kind of operation an *expected update*. Each iteration of iterative policy evaluation updates the value of every state once to produce the new approximate value function v_{k+1}.

Iterative policy evaluation

Input π, the policy to be evaluated
Initialize an array $V(s) = 0$, for all $s \in \mathcal{S}^+$
Repeat
  $\Delta \leftarrow 0$
  For each $s \in \mathcal{S}$:
    $v \leftarrow V(s)$
    $V(s) \leftarrow \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a) \left[r + \gamma V(s') \right]$
    $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$ (a small positive number)
Output $V \approx v_\pi$

2.2 Policy Improvement

Our reason for computing the value function for a policy is to help find 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 an action $a \neq \pi(s)$. We know how good it is to follow the current policy from s–that is $v_\pi(s)$ – 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 thereafter following the existing policy, π. The value of this way of behaving is

$$q_\pi(s, a) \doteq \mathbb{E} \left[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s, A_t = a \right] \tag{8}$$

$$= \sum_{s',r} p(s', r|s, a) \left[r + \gamma v_\pi(s') \right] \tag{9}$$

The key criterion is whether this is greater than 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. That

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

$$q_\pi(s, \pi'(s)) \geq v_\pi(s) \tag{10}$$

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

$$v_{\pi'}(s) \geq v_\pi(s) \tag{11}$$

For those who are interested in proving equation (11), refer *Reinforcement Learning, Sutton (2017)* for details.[1]

So far we have seen how, given a policy and its value function, we can easily evaluate a change in the policy at a single to a particular action. Obviously, extending this view to consider changes at *all* states and to *all* possible actions, selecting at each state the action that appears best according to $q_\pi(s, a)$. In other words, to consider the greedy policy, π', given by

$$\begin{aligned}
\pi'(s) &\doteq \arg\max_a q_\pi(s, a) && (12)\\
&= \arg\max_a \mathbb{E}\left[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s, A_t = a\right] && (13)\\
&= \arg\max_a \sum_{s', r} p(s', r|s, a)\left[r + \gamma v_\pi(s')\right] && (14)
\end{aligned}$$

where $\arg\max_a$ denotes the value of a at which the expression that follows is maximized. Suppose the new greedy policy, π', is as good as, but not better than, the old policy π. Then $v_\pi = v_{\pi'}$, and from equation (13) it follows that for all $s \in \mathcal{S}$:

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

This looks quite familiar! This is the same as the Bellman optimality equation, and obviously, $v_{\pi'}$ must be v_*, and π' (and π) must be optimal policy. Policy improvement must give us a strictly better policy except when the original policy is already optimal.

2.3 Policy Iteration

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

$$\pi_0 \xrightarrow{E} v_{\pi_0} \xrightarrow{I} \pi_1 \xrightarrow{E} v_{\pi_1} \xrightarrow{I} \pi_2 \xrightarrow{E} \cdots \xrightarrow{I} \pi_* \xrightarrow{E} v_*$$

where $\xrightarrow{E}$ denotes a policy *evaluation* and $\xrightarrow{I}$ denotes a policy *improvement*. Each policy is guaranteed to be a strict improvement over the previous one (unless it is already optimal). This way of finding an optimal policy is called *policy iteration*. A complete algorithm is given in the box on the next page. Note that each policy evaluation, itself an iterative computation, is started with the value function for the previous policy. This typically results in a great increase in the speed of convergence of policy evaluation.

Policy iteration (using iterative policy evaluation)

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

2. Policy Evaluation
   Repeat
      $\Delta \leftarrow 0$
      For each $s \in \mathcal{S}$:
         $v \leftarrow V(s)$
         $V(s) \leftarrow \sum_{s',r} p(s', r|s, \pi(s)) [r + \gamma V(s')]$
         $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
   $\Delta < \theta$ (a small positive number)

3. Policy improvement
   policy-stable $\leftarrow$ *true*
   For each $s \in \mathcal{S}$:
      old-action $\leftarrow \pi(s)$
      $\pi(s) \leftarrow \arg\max_a \sum_{s',r} p(s', r|s, a) [r + \gamma V(s')]$
      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

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 requiring multiple sweeps through the state set. If policy evaluation is done iteratively, then convergence exactly to v_π occurs only in the limit. Is it necessary to wait for exact convergence? The example in Figure 3 certainly suggest that it may be possible to truncate policy evaluation. In the example, policy evaluation iterations beyond the first three have no effect on the corresponding greedy policy.
In fact, the policy evaluation step of policy iteration can be truncated in several ways without losing the convergence guarantees of policy iteration. One important special case is when policy evaluation is stopped after just one sweep (one update of each state). This algorithm is called *value iteration*. It can be written as a particularly simple update operation that combines the policy improvement and truncated policy evaluation steps:

$$v_{k+1}(s) \doteq \max_a \mathbb{E}\left[R_{t+1} + \gamma v_k\left(S_{t+1}\right) | S_t = s, A_t = a\right] \tag{15}$$

$$= \max_a \sum_{s',r} p(s', r|s, a) \left[r + \gamma v_k(s')\right] \tag{16}$$

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 that guarantee the existence of v_*.
Another way of understanding value iteration is by reference to the Bellman optimality equation. Note that **value iteration is obtained simply by turning the Bellman optimality equation into an update rule**. Also note how the value iteration update is identical to the policy evaluation update **except that it requires the maximum to be taken over all actions**. Another way

of seeing this close relationship is to compare the backup diagrams for these algorithms (Figure 4). These two are the natural backup operations for computing v_π and v_*.

Finally, let us consider how value iteration terminates. Like policy evaluation, value iteration formally requires an infinite number of iterations to converge exactly to v_*. In practice, we stop once the value function changes by only a small amount in a sweep. The box shows a complete algorithm with this kind of termination condition.

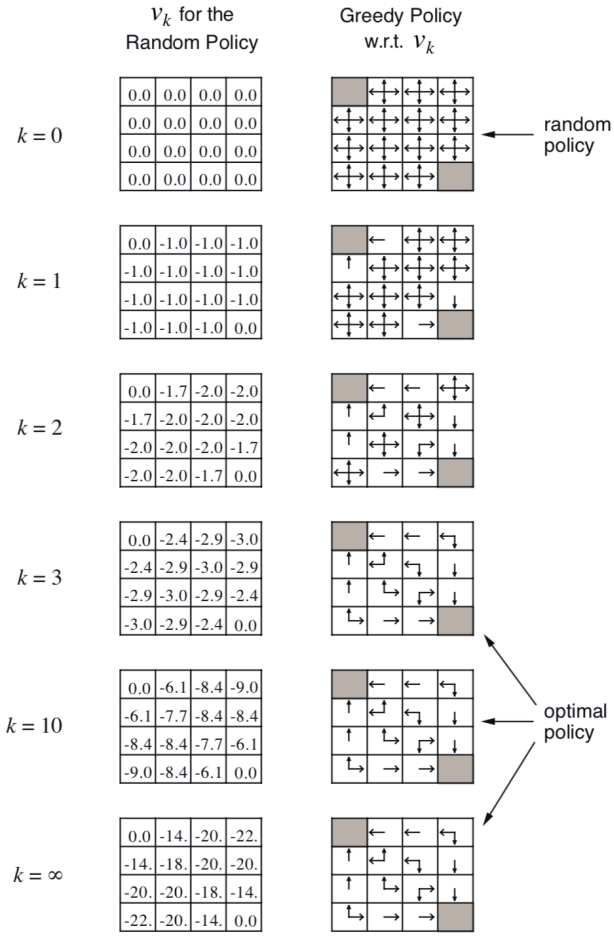


Figure 3: Convergence of iterative policy evaluation on a small grid-world. The left column is the sequence of approximations of the state-value function for the random policy (all actions equal). The right column is the sequence of greedy policies corresponding to the value function estimates (arrows are 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.

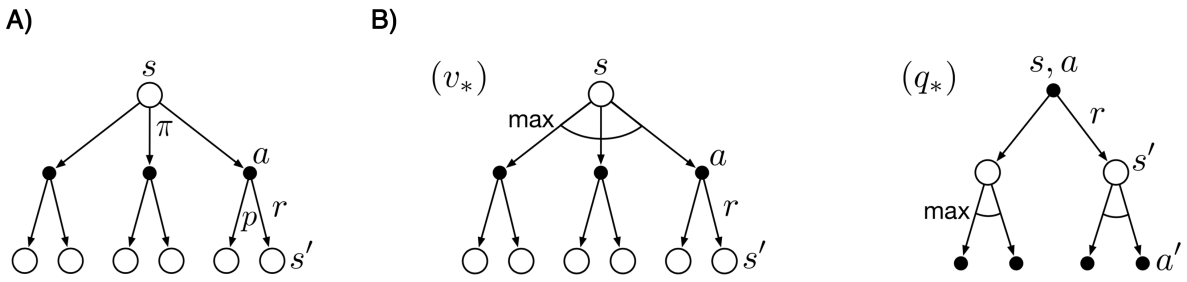


Figure 4: Backup diagram of policy evaluation(A), Backup diagram of value iteration(B).

Value Iteration

Initialize array V arbitrarily (e.g., $V(s) = 0$ for all $s \in \mathcal{S}^+$)

Repeat
  $\Delta \leftarrow 0$
  For each $s \in \mathcal{S}$:
    $v \leftarrow V(s)$
    $V(s) \leftarrow \max_a \sum_{s',r} p(s', r|s, a) [r + \gamma V(s')]$
    $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$ (a small positive number)

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

Value iteration combines, in each of its sweeps, **one sweep of policy evaluation and one sweep of policy improvement**. Faster convergence is often achieved by interposing multiple policy evaluation sweeps between each policy improvement sweep.

3 Monte Carlo Methods

In this chapter we consider our first learning methods for estimating value functions and discovering policies. Unlike the previous chapter, **here we do not assume complete knowledge of the environment**. Monte Carlo methods require only *experience* – sample sequences of states, actions, and rewards from actual or simulated interaction with an environment. Monte Carlo methods are ways of solving the reinforcement learning problem based on **averaging sample returns**. To ensure that well-defined returns are available. That is, we assume experience is divided into episodes, and that all episodes eventually terminate no matter what actions are selected.

3.1 Monte Carlo Prediction

We begin by considering Monte Carlo methods for learning the state-value function for a given policy. Recall that the value of a state is the expected return – expected cumulative future discounted reward – starting from that state. An obvious way to estimate it from experience, then, is simply to average the returns observed after visits to that state. **As more returns are observed, the average should converge to the expected value**. This idea underlies all Monte Carlo methods.
In particular, suppose we wish to estimate $v_\pi(s)$, the value of a state s under policy π, given a set of episodes obtained by following π and passing through s. Each occurrence of state s in an episode is called a *visit* to s. Of course, s may be visited multiple times in the same episode; let us call the first time it is visited in an episode the *first visit* to s The *first-visit MC method* estimates $v_\pi(s)$ as the average of the returns following first visits to s, whereas the *every-visit MC method* averages the returns following all visits to s. First-visit MC is shown in procedural form in the box:

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

Initialize:
  $\pi \leftarrow$ policy to be evaluated
  $V \leftarrow$ an arbitrary state-value function
  $Returns(s) \leftarrow$ an empty list, for all $s \in \mathcal{S}$
Repeat forever:
  Generate an episode using π
  For each state s appearing in the episode:
    $G \leftarrow$ the return that follows the first occurrence of s
    Append G to $Returns(s)$
    $V(s) \leftarrow$ average($Returns(s)$)

By the law of average large numbers the sequence of averages of averages of these estimates converges to their expected value. Each average is itself an **unbiased estimate**, and the standard deviation of its error falls as $1/\sqrt{n}$, where n is the number of returns averaged.

3.2 Monte Carlo Control

We are now ready to consider how Monte Carlo estimation can be used in control, that is, to approximate optimal policies. The overall idea is to proceed according to the same pattern as in the DP chapter, that is, according to the idea of generalized policy iteration (GPI). In GPI **one maintains both an approximate policy and an approximate value function**. The value function is repeatedly altered to more closely approximate the value function for the current policy, and the policy is repeatedly improved with respect to the current value function. These two kinds of changes work against each other to some extent, as each creates a moving target for the other, but together they cause both policy and value function to approach optimality.
To begin, let us consider a Monte Carlo version of classical policy iteration. In this method, we perform alternating complete steps of policy evaluation and policy improvement, beginning with an arbitrary policy π_0 and ending with the optimal policy and optimal action-value function:

$$\pi_0 \xrightarrow{E} q_{\pi_0} \xrightarrow{I} \pi_1 \xrightarrow{E} q_{\pi_1} \xrightarrow{I} \pi_2 \xrightarrow{E} \cdots \xrightarrow{I} \pi_* \xrightarrow{E} q_*$$

where $\xrightarrow{E}$ denotes a complete policy evaluation and $\xrightarrow{I}$ denotes a complete policy improvement, as in policy iteration. For the moment, let us assume that we do indeed observe an infinite number of episodes and that, in addition, the episodes are generated with exploring starts. Under these assumptions, the Monte Carlo methods will compute each q_{π_k} exactly, for arbitrary π_k.
Policy improvement is done by making the policy greedy with respect to the current value function. In this case we have an *action*-value function, and therefore no model is needed to construct the greedy policy. For any action-value function q, the corresponding greedy policy is the one that, for each $s \in \mathcal{S}$, deterministically chooses an action with maximal action-value:

$$\pi(s) \doteq \arg\max_a q(s,a) \tag{17}$$

Policy improvement then can be done by constructing each π_{k+1} as the greedy policy with respect to q_{π_k}. The policy improvement theorem then applies to π_k and π_{k+1} because, for all $s \in \mathcal{S}$,

$$\begin{aligned} q_{\pi_k}(s, \pi_{k+1}(s)) &= q_{\pi_k}(s, \arg\max_a q_{\pi_k}(s,a)) \\ &= \max_a q_{\pi_k}(s,a) \\ &\geq q_{\pi_k}(s, \pi_k(s)) \\ &\geq v_{\pi_k}(s). \end{aligned}$$

To avoid the infinite number of episodes nominally required for policy evaluation, on each evaluation step we move the value function *toward* q_{π_k}, but we do not expect to actually get close except over many steps. Remember that we used this idea when we first introduced the idea of GPI.
For Monte Carlo policy evaluation it is natural to alternate between evaluation and improvement on an episode-by-episode basis. **After each episode, the observed returns are used for policy evaluation, and then the policy is improved at all the states visited in the episode**. A complete simple algorithm along these lines, which we call ***Monte Carlo ES***, for Monte Carlo with Exploring Starts, is given in the box.

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

Initialize, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$:
  $Q(s,a) \leftarrow$ arbitrary
  $\pi(s) \leftarrow$ arbitrary
  $Returns(s,a) \leftarrow$ empty list
Repeat forever:
  Choose $S_0 \in \mathcal{S}$ and $A_0 \in \mathcal{A}(S_0)$ s.t. all pairs have probability > 0
  Generate an episode starting from S_0, A_0, following π
  For each pair s, a appearing in the episode:
    $G \leftarrow$ the return that follows the first occurrence of s, a
    Append G to $Returns(s,a)$
    $Q(s,a) \leftarrow$ average($Returns$(s,a))
  For each s in the episode:
    $\pi(s) \leftarrow \arg\max_a Q(s,a)$

It is easy to see that Monte Carlo ES cannot converge to any suboptimal policy. If it did, then the value function would eventually converge to the value function for that policy, and that in turn would cause the policy to change.

3.3 Monte Carlo Control without Exploring Starts

How can we avoid the unlikely assumption of exploring starts? The only general way to ensure that all actions are selected infinitely often is for the agent to continue to select them. There are two approaches to ensuring this, resulting in what we call *on-policy* methods and *off-policy* methods. On-policy methods attempt to evaluate or improve the policy that is used to make decisions, whereas off-policy methods evaluate or improve a policy different from that used to generate the data. Here, we will focus on how an on-policy Monte Carlo control method can be designed that does not use the unrealistic assumption of exploring starts.
In on-policy control methods the policy is generally *soft*, meaning that $\pi(a|s) > 0$ for all $s \in \mathcal{S}$ and all $a \in \mathcal{A}(s)$, **but gradually shifted closer and closer to a deterministic optimal policy**. The on-policy method we present in this section uses $\epsilon - greedy$ policies, meaning that most of the time they choose an action that has maximal estimated action value, but with probability ϵ they instead select an action at random. That is, all non-greedy actions are given the minimal probability of selection, $\frac{\epsilon}{|\mathcal{A}(s)|}$, and the remaining bulk of the probability, $1 - \epsilon + \frac{\epsilon}{|\mathcal{A}(s)|}$, is given to the greedy action. The ϵ-greedy policies are examples of ϵ-*soft* policies, defined as policies for which $\pi(a|s) \geq \frac{\epsilon}{|\mathcal{A}(s)|}$ for all states and actions, for some $\epsilon > 0$.
The overall idea of on-policy Monte Carlo control is still that of GPI. As in Monte Carlo ES, we use first-visit MC methods to estimate the action-value function for the current policy. Without the assumption of exploring starts, however, we cannot simply improve the policy by making it greedy with respect to the current value function, because that would prevent further exploration of non-greedy actions. Fortunately, GPI does not require that the policy be taken all the way to a greedy policy, only that it be moved *toward* a greedy policy. In our on-policy method we will move it only to an ϵ-greedy policy. For any ϵ-soft policy, π, any ϵ-greedy policy with respect to q_π is guaranteed to be better than or equal to π. The complete algorithm is given in the box below.

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

Initialize, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$:
  $Q(s,a) \leftarrow$ arbitrary
  $Returns(s,a) \leftarrow$ empty list
  $\pi(a|s) \leftarrow$ an arbitrary ϵ-soft policy

Repeat forever:
  (a) Generate an episode using π
  (b) For each pair s, a appearing in the episode:
    $G \leftarrow$ the return that follows the first occurrence of s, a
    Append G to $Returns(s,a)$
    $Q(s,a) \leftarrow$ average$(Returns(s,a))$
  (c) For each s in the episode:
    $A^* \leftarrow \arg\max_a Q(s,a)$ (with ties broken arbitrarily)
    For all $a \in \mathcal{A}(s)$:

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

4 Temporal-Difference Learning

If one had to identify one idea as central and novel to reinforcement learning, it would undoubtedly be *temporal-difference* (TD) learning. TD learning is a combination of Monte Carlo ideas and dynamic programming (DP) ideas. Like Monte Carlo methods, **TD methods can learn directly from raw experience without a model of the environment's dynamics**. Like DP, **TD methods update estimates based in part on other learned estimates, without waiting for a final outcome**.

4.1 TD Prediction

Both TD and Monte Carlo Methods use experience to solve the prediction problem. Given some experience following a policy π, both methods update their estimate V of v_π for the nonterminal states S_t occurring in that experience. Roughly speaking, Monte Carlo methods wait until the return following the visit is known, then use that return as a target for $V(S_t)$. A simple every-visit Monte Carlo method suitable for non-stationary environment is

$$V(S_t) \leftarrow V(S_t) + \alpha \left[G_t - V(S_t)\right] \tag{18}$$

where G_t is the actual return following time t, and α is a constant step-size parameter. Let us call this method *constant-α MC*. Whereas Monte Carlo methods must wait until the end of the episode to determine the increment to $V(S_t)$, **TD methods need to wait only until the next time step**. At time $t+1$ they immediately form a target and make a useful update using the observed reward R_{t+1} and the estimate $V(S_{t+1})$. The simplest TD method makes the update

$$V(S_t) \leftarrow V(S_t) + \alpha \left[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)\right] \tag{19}$$

immediately on transition to S_{t+1} and receiving R_{t+1}. In effect, the target for the Monte Carlo update is G_t, whereas the target for the TD update is $R_{t+1} + \gamma V(S_{t+1})$. This TD method is called *TD(0)*, or *one-step TD*. The box below specifies TD(0) completely in procedural form.

Tabular TD(0) for estimating v_π

Input: the policy π to be evaluated
Initialize $V(s)$ arbitrarily (e.g., $V(s) = 0$, for all $s \in \mathcal{S}^+$)
Repeat (for each episode):
  Initialize S
  Repeat (for each step of episode):
    $A \leftarrow$ action given by π for S
    Take action A, observe R, S'
    $V(s) \leftarrow V(s) + \alpha \left[R + \gamma V(S') - V(s)\right]$
    $S \leftarrow S'$
  until S is terminal

We know from Chapter 2 that

$$v_\pi(s) \doteq \mathbb{E}_\pi \left[G_t | S_t = s\right] \tag{20}$$
$$= \mathbb{E}_\pi \left[R_{t+1} + \gamma G_{t+1} | S_t = s\right] \tag{21}$$
$$= \mathbb{E}_\pi \left[R_{t+1} + \gamma v_\pi \left(S_{t+1}\right) | S_t = s\right] \tag{22}$$

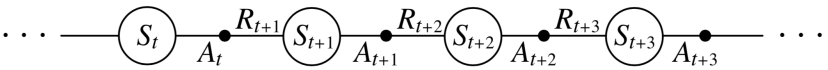
Roughly speaking, Monte Carlo methods use an estimate of Eq. (20) as a target, whereas DP methods use an estimate of Eq. (22) as a target. The TD target, however, combine the sampling of Monte Carlo with the bootstrapping of DP. Here, we refer to TD and Monte Carlo updates as *sample updates* because they involve looking ahead to a sample successor state (or state-action pair), using the value of the successor and the reward along the way to compute a backed-up value, and then updating the value of the original state (or state-action pair) accordingly. *Sample* updates differ from the *expected* updates of DP methods in that **they are based on a single sample successor rather than on a complete distribution of all possible successors**.
Note that the quantity in brackets in the TD(0) update is a sort of error, measuring the difference between the estimated value of S_t and the better estimate $R_{t+1} + \gamma V(S_{t+1})$. This quantity, called the ***TD error***, arises in various forms throughout reinforcement learning:

$$\delta_t \doteq R_{t+1} + \gamma V(S_{t+1}) - V(S_t) \tag{23}$$

Notice that the TD error at each time is the error in the estimate made at that time. Because TD error depends on the next state and next reward, it is not actually available until one time step later. That is, δ_t **is the error in** $V(S_t)$**, available at time** $t+1$.

4.2 Sarsa: On-Policy TD Control

We turn now to use of TD prediction methods for the control problem. As usual, we follow the pattern of generalized policy iteration (GPI), only this time using TD methods for the evaluation or prediction part. As with Monte Carlo methods, we face the need to trade off exploration and exploitation, and again approaches fall into two main classes: **on-policy** and **off-policy**. In this section we present on-policy TD control method.
The first step is to learn an **action-value function** rather than a state-value function. In particular, for an on-policy method we must estimate $q_\pi(s, a)$ for the current behavior policy π and for all states s and actions a. This can be done using essentially the same TD method described above for learning v_π. Recall that an episode consists of an alternating sequence of states and state-action pairs:



In the previous section we considered transitions from state to state and learned the values of states. Now we consider transitions from state-action pair to state-action pair, and learn the values of state-action pairs. The theorems assuring the convergence of state values under TD(0) also apply to the corresponding algorithm for action values:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)\right] \tag{24}$$

This update is done after every transition from a nonterminal state S_t. If S_{t+1} is terminal, then $Q(S_{t+1}, A_{t+1})$ is defined as zero. This rule uses every element of the quintuple of events,

$(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$, that make up a transition from one state-action pair to the next. The quintuple gives rise to the name *Sarsa* for the algorithm. Note that Eq. (24) can be written as

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\delta_t$$

where $\delta_t = R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)$. It is straight forward to design an on-policy control algorithm based on the Sarsa prediction method. As in all on-policy methods, we continually estimate q_π for the behavior policy π, and at the same time change π toward greediness with respect to q_π. The general form of the Sarsa control algorithm is given in the box below.

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

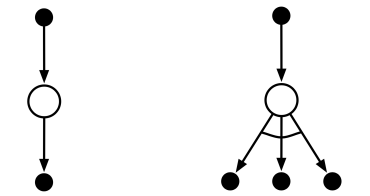
Initialize $Q(s, a)$ for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
  Initialize S
  Choose A from S using policy derived from Q (e.g., ϵ-greedy)
  Repeat (for each step of episode):
    Take action A, observe R, S'
    Choose A' from S' using policy derived from Q (e.g., ϵ-greedy)
    $Q(S, A) \leftarrow Q(S, A) + \alpha\left[R + \gamma Q(S', A') - Q(S, A)\right]$
    $S \leftarrow S'$; $A \leftarrow A'$;
  until S is terminal

4.3 Q-learning: Off-policy TD control

One of the early breakthroughs in reinforcement learning was the development of an off-policy TD control algorithm known as *Q-learning*, defined by

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\right] \tag{25}$$

In this case, the learned action-value function, Q, directly approximates q_*, the optimal action-value function, independent of the policy being followed. This dramatically simplifies the analysis of the algorithm and enabled early convergence proofs. The policy still has an effect in that it determines which state-action pairs are visited and updated. However, all that is required for correct convergence is that all pairs continue to be updated. Under this assumption and a variant of the usual stochastic approximation conditions on the sequence of step-size parameters, Q has been shown to converge with probability 1 to q_*.



The rule (25) updates a state-action pair, so the top node, the root of the update, must be a small, filled action node. The update is also *from* action nodes, maximizing over all those actions possible in the next state. Thus the bottom nodes of the backup diagram should be all these action nodes. Finally, remember that we indicate taking the maximum of these "next actions" nodes with an arc across them.

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

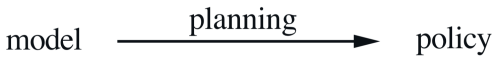
Initialize $Q(s, a)$ for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
  Initialize S
  Repeat (for each step of episode):
    Choose A from S using policy derived from Q (e.g., ϵ-greedy)
    Take action A, observe R, S'
    $Q(S, A) \leftarrow Q(S, A) + \alpha \left[R + \gamma \max_a Q(S', a) - Q(S, A)\right]$
    $S \leftarrow S'$;
  until S is terminal

5 Planning and Learning with Tabular Methods

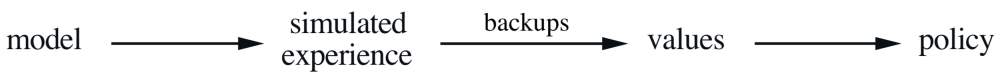
In this chapter we develop a unified view of reinforcement learning methods that require a model of the environment, such as dynamic programming and heuristic search, and methods that can be used without a model such as Monte Carlo and temporal-difference methods. These are respectively called *model-based* and *model-free* reinforcement learning methods. Model-based methods rely on *planning* as their primary component, while model-free methods primarily rely on *learning*. Although there are real differences between these two kinds, of methods, there are also great similarities. In particular, the heart of both kinds of methods is the computation of value functions. Moreover, all the methods are based on looking ahead to future events, computing a backed-up value, and then using it as an update target for an approximate value function.

5.1 Models and planning

By a *model* of the environment we mean anything that an agent can use to predict how the environment will respond to its actions. Given a state and an action, a model produces a prediction of the resultant next state and next reward. If a model is stochastic, then there are several possible next states and next rewards, each with some probability of occurring. Some models produce a description of all possibilities; these we call ***distribution models***. Other models produce just one of the possibilities, sampled according to the probabilities; these we call ***sample models***.
Models can be used to mimic or simulate experience. Given a starting state and action, a sample model produces a possible transition, and a distribution model generates all possible transitions weighted by their probabilities of occurring. Given a starting state and a policy, a sample model could produce an entire episode, and a distribution model could generate all possible episodes and their probabilities. In either case, we say the model is used to *simulate* the environment and produce *simulated experience*.
The word *planning* is used in several different ways in different fields. We use the term to refer to any computational process that takes a model as input and produces or improves a policy for interacting with the modeled environment:



The unified view we present in this chapter is that all planning methods share a common structure, a structure that is also present in the learning methods presented in this book. Here, we have two basic ideas: (1) all (state-space) planning methods involve computing value functions as a key intermediate step toward improving the policy, and (2) they compute value functions by updates or backup operations applied to simulated experience. This common structure can be diagrammed as follows:



Viewing planning methods in this way emphasizes their relationship to the learning methods that we have described in this book. The heart of both learning and planning methods is the **estimation of value functions by backing-up update operations**. The difference is that whereas planning

uses simulated experience generated by a model, learning methods use real experience generated by the environment. Of course this difference leads to a number of other differences, for example, in how performance is assessed and in how flexibly experience can be generated. But the common structure means that many ideas and algorithms can be transferred between planning and learning. In particular, in many cases a learning algorithm can be substituted for the key update step of a planning method. Learning methods require only experience as input, and in many cases they can be applied to simulated experience just as well as to real experience. The box below shows a simple example of a planning method based on one-step tabular Q-learning and on random samples from a sample model. This method, which we call *random-sample one-step tabular Q-planning*, converges to the optimal policy for the model under the same conditions that one-step tabular Q-learning converges to the optimal policy for the real environment.

Random-sample one-step tabular Q-planning

Do 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':
   $Q(S,A) \leftarrow Q(S,A) + \alpha\left[R + \gamma \max_a Q(S',a) - Q(S,A)\right]$

5.2 Dyna: Integrating Planning, Acting, and Learning

When planning is done on-line, while interacting with the environment, new information gained from the interaction may change the model and thereby interact with planning. It may be desirable to customize the planning process in some way to the states or decisions currently under consideration, or expected in the near future. If decision making and model learning are both computation-intensive processes, then the available computational resources may need to be divided between them. In this section we present Dyna-Q, a simple architecture integrating the major functions needed in an on-line planning agent. Within a planning agent, there are least two role for real experience: 1) it can be used to improve the model (to make it more accurately match the real environment) and 2) 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 later we call *direct reinforcement learning* (direct RL). The possible relationships between experience, model, values, and policy are summarized in Figure 5. Each arrow shows a relationship of influence and presumed improvement. Note how experience can improve value functions and policies either directly or indirectly via the model. It is the latter, which is sometimes called indirect reinforcement learning, that is involved in planning.

Both direct and indirect methods have advantages and disadvantages. Indirect methods often make **fuller use of a limited amount of experience and thus achieve a better policy with fewer environmental interactions**. Direct methods, on the other hand, are much simpler and are **not affected by biases in the design of the model**. Some have argued that indirect methods are always superior to direct ones, while others have argued that direct methods are responsible for most human and animal learning. However, recent studies suggest existence of both model-based and model-free learning in human brains: Model-based are mostly encoded in the prefrontal cortex, while model-free learning are mostly encoded in the basal ganglia. We will discuss further

neuroscience details in the last Chapter.

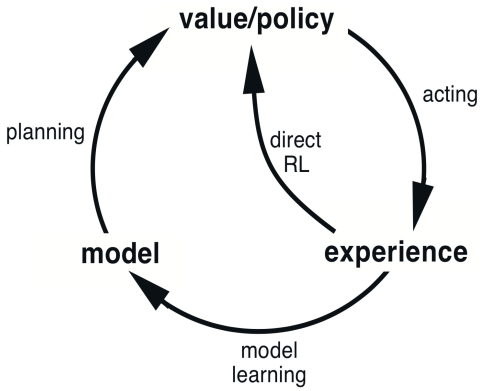


Figure 5: Diagram of Dyna-Q

Dyna-Q includes all of the processes shown in Figure 5–planning, acting, model-learning, and direct RL–all occurring continuously. The planning method is the random-sample one-step tabular Q-planning method given above. The direct RL method is **one-step tabular Q-learning**. The model-learning method is also **table-based and assumes the environment is deterministic**. After each transition $S_t, A_t \to R_{t+1}, S_{t+1}$, the model records in its table entry for S_t, A_t the prediction that R_{t+1}, S_{t+1} will deterministically follow. Thus, if the model is queried with a state-action pair that has been experience before, it simply returns the last-observed next state and next reward as its prediction. During planning, the Q-learning algorithm randomly samples only from state-action pairs that have previously been experienced (in Step 1), so the model never queried with a pair about which it has no information.

The overall architecture of Dyna agents, of which the Dyna-Q algorithm is one example, shown

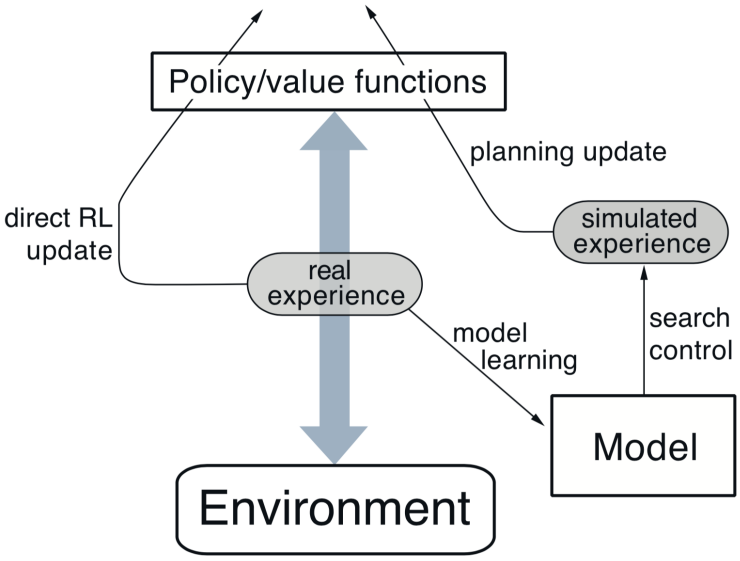


Figure 6: The general Dyna Architecture. Real experience, passing back and forth between the environment and the policy, affects policy and value functions in much the same way as does simulated experience generated by the model of the environment.

in Figure 6. The central column represents the basic interaction between agent and environment, giving rise to a trajectory of real experience. The arrow on the left of the figure represents direct reinforcement learning operating operating on real experience to improve the value function and the policy. On the right are model-based processes. The model is learned from real experience and gives rise to simulated experience. We use the term search control to refer to the process that selects the starting states and actions for the simulated experiences generated by the model. Finally, planning is achieved by applying reinforcement learning methods to the simulated experiences just as if they had really happened. Typically, as in Dyna-Q, the same reinforcement learning method is used both for learning from real experience and for planning from simulated experience. The reinforcement learning method is thus the "final common path" for both learning and planning. Learning and planning are deeply integrated in the sense that they share almost all the same machinery, differing only in the source of their experience.

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) $A \leftarrow \epsilon$-greedy(S,Q)
- (c) Take action A; observe resultant, reward, R, and state S'
- (d) $Q(S,A)\ Q(S,A) + \alpha\left[R + \gamma \max_a Q(S',a) - Q(S,A)\right]$
- (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
  - $R, S' \leftarrow Model(S,A)$
  - $Q(S,A) \leftarrow Q(S,A) + \alpha\left[R + \gamma \max_a Q(S',a) - Q(S,A)\right]$

6 Function Approximation

Reinforcement learning methods previously discussed assume discrete, well defined states. However, in reality, for most cases states are continuous, thus should be approximated. In this chapter, we will learn methods estimating the state-value function from on-policy data, that is, in approximating v_π from experience generated using a known policy π. The approximate value function is represented as a parameterized functional form with weight vector $\mathbf{w} \in \mathbb{R}^d$. We will write $\hat{v}(s, \mathbf{w}) \approx v_\pi(s)$ for the approximate value of state s given weight vector $\mathbf{w}$. For example, $\hat{v}$ might be a linear function in features of the state, with $\mathbf{w}$ the vector of feature weights. More generally, $\hat{v}$ might be the function of computed by a multi-layer artificial neural network, with $\mathbf{w}$ the vector of connection weights in all the layers. Typically, the number of weights (the dimensionality of $\mathbf{w}$) is much less than the number of states ($d \ll |\mathcal{S}|$), and changing one weight changes the estimated value of many states. Consequently, when a single state is updated, the change generalizes from that state to affect the values of many other states. Such ***generalization*** makes the learning potentially more powerful but also potentially more difficult to manage and understand.

6.1 Value-Function Approximation

All of the prediction methods in this book updates to an estimated value function that shift its value at particular states toward a "backed-up value", or *update target*, for that state. Let's refer to an individual update by the notation $s \mapsto u$, where s is the state updated and u is the update target that s's estimated value is shifted toward. For example, Monte Carlo update for value prediction is $S_t \mapsto G_t$, and the TD(0) update is $S_t \mapsto R_{t+1} + \gamma\hat{v}(S_{t+1}, \mathbf{w}_t)$.
The update $s \mapsto u$ means that the estimated value for state s should be more like the update target u. Up to now, the actual update has been trivial: the table entry for s's estimated value has simply been shifted a fraction of the way toward u, and the estimated values of all other states were left unchanged. However, arbitrarily complex sophisticated methods discussed in this chapter works in another way around: updating at s generalizes so that the estimated values of many other states are changed as well. Machine learning methods that learn to mimic input-output examples in this way are called *supervised learning* methods, and when the outputs are numbers, like u, the process is often called *function approximation*.
In principle, we can use any method for supervised learning from examples, including artificial neural networks, decision trees, and various kinds of multivariate regression. However, not all function approximation methods are equally well suited for use in reinforcement learning. The most sophisticated artificial neural network and statistical methods all assume a static training set over which multiple passes are made. In reinforcment learning, however, it is important that **learning be able to occur online, while the agent interacts with its environment or with a model of its environment**. Thus, methods that cannot easily handle such nonstationarity are less suitable for reinforcement learning.

6.2 The Prediction Objective ($\overline{\text{VE}}$)

Up to now we have not specified an explicit objective for prediction. In the tabular case a continuous measure of prediction quality was not necessary because the learned value function could come to equal the true value function exactly. Moreover, the learned values at each state were decoupled–an update at one state affected no other. But with genuine approximation, an update at one state affects many others, and it is not possible to get the values of all states exactly correct. We have far more states than weights, so making one state's estimate more accurate invariably means making others' less accurate. We are obligated then to say which states we care most about. We must specify a state distribution $\mu(s) \geq 0$, $\sum_s \mu(s) = 1$, representing how much we care about the error in each state s. By the error in state s we mean the square of the difference between the approximate value $\hat{v}(s, \mathbf{w})$ and the true value $v_\pi(s)$. Weighting this over the state space by μ, we obtain a natural objective function, the *Mean Squared Value Error*, denoted $\overline{\text{VE}}$:

$$\overline{\text{VE}} \doteq \sum_{s \in \mathcal{S}} \mu(s) \left[v_\pi(s) - \hat{v}(s, \mathbf{w}) \right]^2 \tag{26}$$

The square root of this measure, the root $\overline{\text{VE}}$, gives a rough measure of how much the approximate values differ from the true values and is often used in plots.
An ideal goal in terms of $\overline{\text{VE}}$ would be to find a global optimum, a weight vector $\mathbf{w}^*$ for which $\overline{\text{VE}}(\mathbf{w}^*) \leq \overline{\text{VE}}(\mathbf{w})$ for all $\mathbf{w}$ in some neighborhood of $\mathbf{w}^*$.

6.3 Stochastic-gradient and Semi-gradient Methods

In gradient-descent methods, the weight vector is a column vector with a fixed number of real valued components, $\mathbf{w} \doteq (w_1, w_2, \ldots, w_d)^\top$, and the approximate value function $\hat{v}(s, \mathbf{w})$ is a differentiable function of $\mathbf{w}$ for all $s \in \mathcal{S}$. We will be updating $\mathbf{w}$ at each of a series of discrete time steps, $t = 0, 1, 2, 3, \ldots$, so we will need a notation $\mathbf{w}_t$ for the weight vector at each step. For now, let us assume that, on each step, we observe a new example $S_t \mapsto v_\pi(S_t)$ consisting of a (possibly randomly selected) state S_t and its true value under the policy. Even though we are given the exact, correct values, $v_\pi(S_t)$ for each S_t, there is still a difficult problem because our function approximator has limited resources and thus limited resolution. In particular, there is generally no $\mathbf{w}$ that gets all states, or even all the examples, exactly correct. In addition, we must generalize to all the other states that have not appeared in examples.
We assume that states appear in examples with the same distribution, μ, over which we are trying to minimize the $\overline{\text{VE}}$. A good strategy in this case is to try to minimize error on the observed examples. ***Stochastic gradient-descent*** (SGD) methods do this by adjusting the weight vector after each example by a small amount in the direction that would most reduce the error on that example:

$$\begin{aligned} \mathbf{w}_{t+1} &\doteq \mathbf{w}_t - \frac{1}{2}\alpha \nabla \left[v_\pi(S_t) - \hat{v}(S_t, \mathbf{w}_t) \right]^2 && (27) \\ &= \mathbf{w}_t + \alpha \left[v_\pi(S_t) - \hat{v}(S_t, \mathbf{w}_t) \right] \nabla \hat{v}(S_t, \mathbf{w}_t), && (28) \end{aligned}$$

where $\alpha > 0$, and $\nabla f(\mathbf{w})$, for any scalar expression $f(\mathbf{w})$ that is a function of a vector, denotes the column vector of partial derivatives of the expression with respect to the components of the vector:

$$\nabla f(\mathbf{w}) \doteq \left(\frac{\partial f(\mathbf{w})}{\partial w_1}, \frac{\partial f(\mathbf{w})}{\partial w_2}, \ldots, \frac{\partial f(\mathbf{w})}{\partial w_d} \right)^\top \tag{29}$$

This derivative vector is the *gradient* of f with respect to $\mathbf{w}$. SGD methods are "gradient descent" methods because the overall step in $\mathbf{w}_t$ is proportional to the negative gradient of the example's squared error, equation (26).

We turn now to the case in which the target output, here denoted $U_t \in \mathbb{R}$, of the tth training example, $S_t \mapsto U_t$, is not the true value, $v_\pi(S_t)$, but some possibly random, approximation to it. For example, U_t might be a noise-corrupted version of $v_\pi(S_t)$, or it might be one of the bootstrapping targets using $\hat{v}$ mentioned in the previous section. In these cases we cannot perform the exact update as in equation (28) because $v_\pi(S_t)$ is unknown, but we can approximate it by substituting U_t in place of $v_\pi(S_t)$. This yields the following general SGD method for state-value prediction:

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha \left[U_t - \hat{v}(S_t, \mathbf{w}_t)\right] \nabla \hat{v}(S_t, \mathbf{w}_t) \tag{30}$$

If U_t is an *unbiased estimate*, that is, if $\mathbb{E}\left[U_t | S_t = s\right] = v_\pi(S_t)$, for each t, when $\mathbf{w}_t$ is guaranteed to converge to a local optimum under the usual stochastic approximation conditions for decreasing α. For example, suppose the states in the examples are the states generated by interaction with the environment using policy π. Because the true value of a state is the expected value of the return following it, the Monte Carlo target $U_t \doteq G_t$ is by definition an unbiased estimate of $v_\pi(S_t)$. Thus, the gradient descent version of Monte Carlo state-value prediction is guaranteed to find a locally optimal solution. Pseudocode for a complete algorithm is shown in the box below.

Gradient Monte Carlo Algorithm for Estimating $\hat{v} \approx v_\pi$

Input: the policy π to be evaluated
Input: a differentiable function $\hat{v} : \mathcal{S} \times \mathbb{R}^d \to \mathbb{R}$
Algorithm parameter: step size $\alpha > 0$
Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\mathbf{w} = 0$)

Loop forever (for each episode):
  Generate an episode $S_0, A_0, R_1, S_1, A_{1,...,R_T,S_T}$ using π
  Loop for each step of episode, $t = 0, 1, ..., T-1$:
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha \left[G_t - \hat{v}(S_t, \mathbf{w})\right] \nabla \hat{v}(S_t, \mathbf{w})$

One does not obtain the same guarantees if a bootstrapping estimate of $v_\pi(S_t)$ is used as the target U_t. Bootstrapping target such as DP target $\sum_{a,s',r} \pi(a|S_t)p(s', r|S_t, a)\left[r + \gamma \hat{v}(s', \mathbf{w}_t)\right]$ all depend on the current value of the weight vector $\mathbf{w}_t$, which implies that they will be biased ant that they will not produce a true gradient-descent method. One way to look at this is that the key step from equation (28) relies on the target being independent of $\mathbf{w}_t$. Thus bootstrapping methods are not instances of true gradient descent, and we call them *semi-gradient methods*. A prototypical semi-gradient method is semi-gradient TD(0), which uses $U_t \doteq R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w})$ as its target. Complete pseudocode for this method is given in the box in the next page.

Here, we will not discuss further details for the value function approximation methods and n-policy control. Great examples using various function approximation methods such as linear methods, Fourier basis, artificial neural networks are well discussed and explained in Sutton's Reinforcement Learning. (2017; Chapter 9.4 $\sim$ 9.1) These chapters are highly recommended for those who are interested in implementing methods mentioned above.

Semi-gradient TD(0) for estimating $\hat{v} \approx v_\pi$

Input: the policy π to be evaluated
Input: a differentiable function $\hat{v} : \mathcal{S}^+ \times \mathbb{R}^d \to \mathbb{R}$ such that $\hat{v}(\text{terminal}, \cdot) = 0$
Algorithm parameter: step size $\alpha > 0$
Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\mathbf{w} = 0$)

Loop forever (for each episode):
    Initialize S
    Loop for each step of episode:
        Choose A $\pi(\cdot|S)$
        Take action A, observe R, S'
        $\mathbf{w} \leftarrow \mathbf{w} + \alpha \left[R + \gamma \hat{v}(S', \mathbf{w}) - \hat{v}(S, \mathbf{w})\right] \nabla \hat{v}(S, \mathbf{w})$
        $S \leftarrow S'$
    Until S is terminal

7 Policy Gradient Methods

So far almost all methods have been *action-value methods*; they learned the values of actions and then selected actions based on their estimated action values; their policies would not even exist without the action-value estimates. In this chapter we consider methods that instead learn a *parameterized policy* that can select actions without consulting a value function. A value function may still be used to *learn* the policy parameter, but is not required for action selection. We use the notation $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ for the policy's parameter vector. Thus we write $\pi(a|s, \boldsymbol{\theta}) = Pr\{A_t = a|S_t = s, \boldsymbol{\theta}_t = \boldsymbol{\theta}\}$ for the probability that action a is taken at time t given that the environment is in state s at time t with parameter $\boldsymbol{\theta}$.
In this chapter we consider methods for learning the policy parameter based on the gradient of some scalar performance measure $J(\boldsymbol{\theta})$ with respect to the policy parameter. These methods seek to *maximize* performance, so their updates approximate gradient *ascent* in J:

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

where $\widehat{\nabla J(\boldsymbol{\theta}_t)} \in \mathbb{R}^d$ is a stochastic estimate whose expectation approximates the gradient of the performance measure with respect to its argument $\boldsymbol{\theta}_t$.

7.1 Policy Approximation and its Advantages

In policy gradient methods, the policy can be parameterized in any way, as long as $\pi(a|s, \boldsymbol{\theta})$ is differentiable with respect to its parameters, that is, as long as $\nabla\pi(a|s, \boldsymbol{\theta})$ exists and is finite for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$, and $\boldsymbol{\theta} \in \mathbb{R}^{d'}$. In practice, to ensure exploration we generally require that the policy never becomes deterministic.
If the action space is discrete and not too large, then a natural and common kind of parameterization is to form parameterized numerical preferences $h(s, a, \boldsymbol{\theta}) \in \mathbb{R}$ for each state-action pair. The actions with highest preferences in each state are given the highest probabilities of being selected, for example, according to an exponential softmax distribution:

$$\pi(a|s, \boldsymbol{\theta}) \doteq \frac{e^{h(s,a,\boldsymbol{\theta})}}{\sum_b e^{h(s,b,\boldsymbol{\theta})}} \tag{31}$$

Note that the denominator here is just what is required so that the action probabilities in each state sum to one. We call this kind of policy parameterization *softmax in action preferences*.
One advantage of parameterizing policies according to the softmax in action preferences is that the approximate policy can approach a deterministic policy, whereas with ϵ-greedy action selection over action values there is always an ϵ probability of selecting a random action. Of course, one could select according to a softmax distribution based on action values, but this alone would not allow the policy to approach a deterministic policy. Instead, the action-value estimates would converge to their corresponding true values, which would differ by a finite amount, translating to specific probabilities other than 0 and 1. Action preferences are different because they do not approach specific values; instead they are driven to produce the optimal stochastic policy. If the optimal policy is deterministic, then the preferences of the optimal actions will be driven infinitely higher than all suboptimal actions.
A second advantage of parameterizing policies according to the sofmax in action preferences is that it enables the selection of actions with arbitrary probabilities. In problems with significant function

approximation, the best approximate policy may be stochastic. For example, in card games with imperfect information the optimal play is often to do two different things with specific probabilities, such as when bluffing in Poker.

7.2 The Policy Gradient Theorem

The episodic and continuing cases define the performance measure, $J(\boldsymbol{\theta})$, differently and thus have to be treated separately to some extent. Nevertheless, we will try to present both cases uniformly, and we develop a notation so that the major theoretical results can be described with a single set of equations.
In this section we treat the episodic case, for which we define the performance measure as the value of the start state of the episode. We can simplify the notation without losing any meaningful generality by assuming that every episode starts in some particular state s_0. Then, in episodic case we define performance as

$$J(\boldsymbol{\theta}) \doteq v_{\pi_{\boldsymbol{\theta}}}(s_0) \tag{32}$$

where $v_{\pi_{\boldsymbol{\theta}}}$ is the true value for $\pi_{\boldsymbol{\theta}}$, the policy determined by $\boldsymbol{\theta}$. From here on in our discussion we will assume no discounting (that is, $\gamma = 1$) for the episodic case, although for completeness we do include the possibility of discounting in the boxed algorithms. With function approximation, it may seem challenging to change the policy parameter in a way that ensures improvement. The problem is that performance depends on both the action selections and the distribution of states in which those selections are made, and that both of these are affected by the policy parameter. Given a state, the effect of the policy parameter on the actions, and thus on reward, can be computed in a relatively straightforward way from knowledge of the parameterization. But the effect of the policy on the state distribution is a function of the environment and is typically unknown. How can we estimate the performance gradient with respect to the policy parameter when the gradient depends on the unknown effect of policy changes on the state distribution?
Fortunately, there is an excellent theoretical answer to this challenge in the form of the *policy gradient theorem*, which provides an analytic expression for the gradient of performance with respect to the policy parameter that does *not* involve the derivative of the state distribution. The policy gradient theorem for the episodic case establishes that

$$\nabla J(\boldsymbol{\theta}) \propto \sum_s \mu(s) \sum_a q_\pi(s,a) \nabla \pi(a|s,\boldsymbol{\theta}) \tag{33}$$

where gradient are column vectors of partial derivatives with respect to the components of $\boldsymbol{\theta}$, and π denotes the policy. For those who are interested in proving policy gradient theorem in episodic case, refer to Chapter 13.2 of Sutton's Reinforcement Learning(2017).

7.3 REINFORCE: Monte Carlo Policy Gradient

Recall our overall strategy of stochastic gradient ascent. The sample gradients need only be proportional to the gradient because any constant of proportionality can be absorbed into the step size α, which is otherwise arbitrary. The policy gradient theorem gives an exact expression proportional to the gradient; all that is needed is some way of sampling whose expectation equals or approximates this expression. Notice that the right-hand side of the policy gradient theorem is a sum over states weighted by how often the states occur under the target policy π; if π is followed, then states will be encountered in these proportions. Thus

$$\nabla J(\boldsymbol{\theta}) \propto \sum_s \mu(s) \sum_a q_\pi(s,a) \nabla \pi(a|s,\boldsymbol{\theta}) \tag{34}$$

$$= \mathbb{E}_\pi \left[\sum_a q_\pi(S_t,a) \nabla \pi(a|S_t,\boldsymbol{\theta}) \right] \tag{35}$$

We could stop here and instantiate our SGA algorithm as

$$\boldsymbol{\theta}_{t+1} \doteq \boldsymbol{\theta}_t + \alpha \sum_a \hat{q}(S_t,a,\mathbf{w}) \nabla \pi(a|S_t,\boldsymbol{\theta})$$

where $\hat{q}$ is some learned approximation to q_π. Now assume update at time t involves jsut A_t, the one action actually taken at time t. To introduce A_t in the equation, we will use the same way as we introduced S_t in equation (35)—by replacing a sum over the random variable's by an expectation under π, then sampling the expectation. Equation (35) involves an appropriate sum over actions, but each term is not weighted by $\pi(a|S_t,\boldsymbol{\theta})$ as is needed for an expectation under π. So we introduce such a weighting, without changing the quality, by multiplying and then dividing the summed terms by $\pi(a|S_t,\boldsymbol{\theta})$):

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

where G_t is the return as usual. Thus, the REINFORCE update would be

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

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

Notice that we can express fractional vector $\frac{\nabla\pi(A_t|S_t,\boldsymbol{\theta})}{\pi(A_t|S_t,\boldsymbol{\theta})}$ in a simpler form: $\nabla \ln \pi(A_t|S_t,\boldsymbol{\theta})$. As we discussed early in this section, we assumed all cases are non-discounted case($\gamma = 1$). However, we can generalize REINFORCE algorithm to cover discounted case as well. Pseudocode for REINFORCE is as below.

REINFORCE: Monte-Carlo Policy-Gradient Control (episodic) for π_*

Input: a differentiable policy parameterization $\pi(a|s,\boldsymbol{\theta})$
Algorithm parameter: step size $\alpha > 0$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ (e.g., to $\mathbf{0}$)

Loop forever (for each episode):
  Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot,\boldsymbol{\theta})$
  Loop for each step of the episode $t = 0, 1, \ldots, T-1$:
    $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha\gamma^t G \nabla \ln \pi(A_t|S_t,\boldsymbol{\theta})$

Remember REINFORCE algorithm lies on Monte Carlo method. Thus as well as Monte Carlo method, REINFORCE may be of high variance and thus produce slow learning.

7.4 Actor-Critic Methods

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

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

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

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

The policy gradient theorem with baseline can be used to derive an update rule using similar steps as in the previous section:

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

One natural choice for the baseline is an estimate of the state value $\hat{v}(S_t, \mathbf{w})$, where $\mathbf{w} \in \mathbb{R}^m$ is a weight vector learned by one of the methods presented in previous chapters. However, still this method–called REINFORCE with Baseline–uses G_t as its target, thus has high variance making the learning time longer and inconvenient to implement online or for continuing problems. As we have seen earlier, with temporal-difference methods we can eliminate these inconveniences. In order to gain these advantages in the case of policy gradient methods we use **actor-critic** methods with a bootstrapping critic.

First consider one-step actor-critic methods, the analog of the TD methods introduced in Chapter 4. The main appeal of one-step methods is that they are fully online and incremental. One-step actor-critic methods replace the full return of REINFORCE with the one-step return (and use a learned state-value function as the baseline) as follows:

$$\begin{aligned}\boldsymbol{\theta}_{t+1} &\doteq \boldsymbol{\theta}_t + \alpha\Big(G_{t:t+1} - \hat{v}(S_t, \mathbf{w})\Big)\frac{\nabla\pi(A_t|S_t,\boldsymbol{\theta})}{\pi(A_t|S_t,\boldsymbol{\theta})} \\ &= \boldsymbol{\theta}_t + \alpha\Big(R_{t+1} + \gamma\hat{v}(S_{t+1}, \mathbf{w}) - \hat{v}(S_t, \mathbf{w})\Big)\frac{\nabla\pi(A_t|S_t,\boldsymbol{\theta})}{\pi(A_t|S_t,\boldsymbol{\theta})} \\ &= \boldsymbol{\theta}_t + \alpha\delta_t\frac{\nabla\pi(A_t|S_t,\boldsymbol{\theta})}{\pi(A_t|S_t,\boldsymbol{\theta})}\end{aligned}$$

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

One-step Actor-Critic (episodic) for estimating $\pi_{\boldsymbol{\theta}} \approx \pi_*$

Input: a differentiable policy parameterization $\pi(a|s,\boldsymbol{\theta})$
Input: a differentiable state-value function parametirzation $\hat{v}(s,\mathbf{w})$
Parameters: step sizes $\alpha^{\boldsymbol{\theta}} > 0, \alpha^{\mathbf{w}} > 0$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ and state-value weight $\mathbf{w} \in \mathbb{R}^d$ (e.g., to $\mathbf{0}$)
Loop forever (for each episode):
  Initialize $\mathcal{S}$ (first state of episode)
  $I \leftarrow 1$
  Loop while $\mathcal{S}$ is not terminal (for each time step):
    $A \sim \pi(\cdot|S,\boldsymbol{\theta})$
    Take action A, observe S', R
    $\delta \leftarrow R + \gamma\hat{v}(S',\mathbf{w}) - \hat{v}(S,\mathbf{w})$ (if S' is terminal, $\hat{v}(S',\mathbf{w}) \doteq 0$)
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}}\delta\nabla\hat{v}(S,\mathbf{w})$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} I\delta\nabla\ln\pi(A|S,\boldsymbol{\theta})$
    $I \leftarrow \gamma I$
    $S \leftarrow S'$