UniTs - University of Trieste

Faculty of Scientific and Data Intensive Computing

Department of mathematics informatics and geosciences

Reinforcement Learning

Lecturer:
Prof. Antonio Celani

Authors:
Christian Faccio

August 8, 2025

github.com/christianfaccio christianfaccio@outlook.it

This book is a summary of the lectures given in the cource ”Reinforcement Learning” by professor Antonio Celani at the University of Trieste, along with the referenced book ”Reinforcement Learning: An Introduction” by Sutton and Barto. The course is part of the Master’s degree in Data Science and Artificial Intelligence at the University of Trieste, in which I am currently enrolled. This

While these notes were primarily created for my personal study, they may serve as a valuable resource for fellow students and professionals interested in reinforcement learning.

The content focuses on both theoretical foundations and practical implementations, documenting the challenges and solutions I’ve encountered while learning these advanced concepts. My goal in sharing these notes is to provide a practical perspective on reinforcement learning from a student’s point of view, hopefully making these complex topics more approachable for others who are on a similar learning path.

Contents

1 Introduction

Reinforcement Learning (RL) is learning what to do so as to maximize a numerical reward signal. The learner is not told which actions to take, but instead must discover which actions yield the most reward by trying them. It is different from supervised learning, where the agent learns from a dataset of input-output pairs, and from unsupervised learning, where the agent tries to find patterns in data without any explicit feedback. Therefore, it is considered a third machine learning paradigm.
One challenge that arise here is the trade-off between exploration and exploitation. The agent must explore the environment to discover which actions yield the most reward, but it must also exploit its current knowledge to maximize its reward. All reinforcement learning agents have explicit goals, can sense aspects of their environments and can choose actions to influence their environments, making RL an approach that starts with a complete, interactive and goal-seeking agent.
Elements of reinforcement learning include:

- **Policy**: A strategy that the agent employs to determine its actions based on the current state of the environment. Roughly speaking, it is a mapping from perceived states of the environment to actions to be taken when in those states. In some cases the policy may be a simple function or lookup table, whereas in others it may involve extensive computation such as a search process. In general, policies may be stochastic, specifying provabilities for each action.
- **Reward signal**: A feedback signal that the agent receives from the environment after taking an action. The reward signal is used to evaluate the effectiveness of the agent's actions and to guide its learning process. It is the primary basis for alternating the policy.
- **Value function**: A function that estimates the expected cumulative reward that the agent can obtain from a given state or state-action pair. The value function helps the agent to evaluate the long-term consequences of its actions and to make decisions that maximize its expected reward over time. The value of a state is the total amount of reward an agent can expect to accumulate over the future, starting from that state.
- **Model of the environment**: An internal representation of the environment that the agent uses to predict the consequences of its actions. The model can be used to simulate the environment and to plan actions based on the predicted outcomes. It is not always necessary, but it can be useful for planning and decision-making.

Most of the time we move *greedily*, selecting the move that leads to the state with greatest value, i.e., with the highest estimated probability of winning. Occasionally, however, we select randomly from among the other moves instead. These are called *exploratory* moves because they cause us to experience states that we might otherwise never see.
If we let S_t denote the state before the greedy move, and S_{t+1} the state after that move, then the update to the estimated value of S_t, denoted $V(S_t)$, can be written as

$$V(S_t) \leftarrow V(S_t) + \alpha \left(V(S_{t+1}) - V(S_t) \right)$$

where α is a small positive number called the *learning rate*. This equation is known as the *temporal difference* (TD) learning rule, which is a fundamental concept in reinforcement learning. It allows the agent to update its value estimates based on the difference between the predicted value of the next state and the current value estimate.

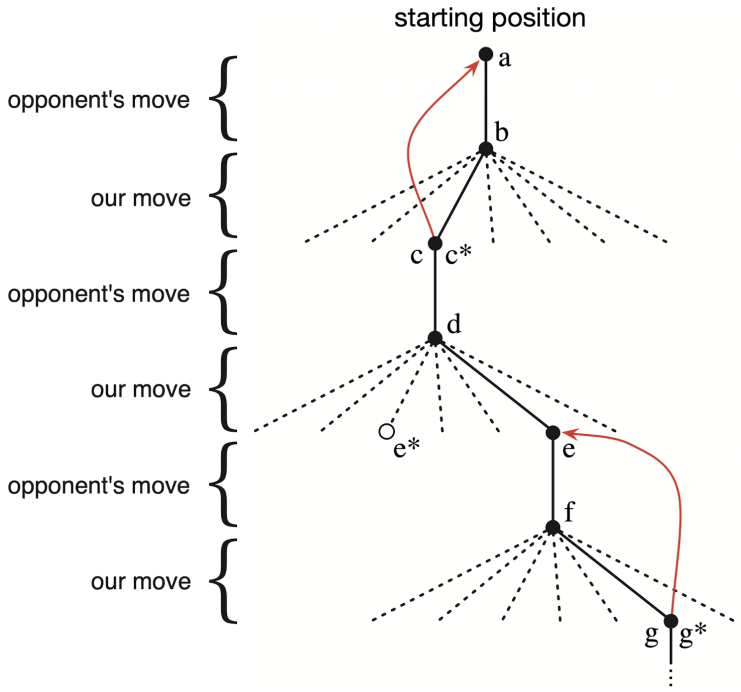


Figure 1.1: Tic Tac Toe example of a reinforcement learning agent. The agent learns to play the game by exploring different moves and receiving rewards based on the outcome of the game.

Multi-Armed Bandit Problem

The most important feature distinguishing RL from other types of learning is that it uses training information that *evaluates* the actions taken rather than *instructs* by giving correct actions. This is what creates the need for active exploration, for an explicit search for good behavior.

Let's say we have a slot machine with multiple arms (k) and we are forced to repeatedly choose one arm to pull. After each choice we receive a numerical reward chosen from a stationary (*strong assumption!*) probability distribution that depends on the action we selected. Our objective is to maximize the expected total reward over some time period, i.e. *time steps*.

In this k-armed bandit problem, each of the k actions has an expected or mean reward given that that action is selected; this is the *value* of that action. We denote the action selected on time step t as A_t and the corresponding reward as R_t. The value then of an arbitrary action a, denoted $q_*(a)$, is the expected reward given that a is selected:

$$q_*(a) \equiv \mathbb{E}[R_t | A_t = a]$$

We assume that we don't know the action values with certainty, although we may have estimates. We denote the estimated value of action a at time step t as $Q_t(a)$. We would like $Q_t(a)$ to be close to $q_*(a)$, but we do not know the true values.

It we maintain estimates of the action values, then at any time step there is at least one action whose estimated value is greatest. We call these the *greedy* actions. When we select one of these actions, we say that we are *exploiting* our current knowledge of the values of the actions. If instead we select one of the nongreedy actions, then we say we are *exploring*, because this enables us to improve our estimate of the nongreedy action's value. Exploitation is the right thing to do to maximize the expected reward on the one step, but exploration may produce the greater total reward in the long run.

However, most of these methods make strong assumptions about stationarity and prior knowledge that are either violated or impossible to verify in most applications and in the full reinforcement learning problem that we consider in subsequent chapters. The guarantees of optimality or bounded

loss for these methods are of little comfort when the assumptions of their theory do not apply.

Action-value Methods

Action-value methods are a class of reinforcement learning algorithms that focus on estimating the value of actions based on the rewards received from the environment. The key idea is to maintain an estimate of the expected reward for each action and to use this information to make decisions about which action to take next.

$$Q_t(a) \equiv \frac{\sum_{i=1}^{t-1} R_i \cdot \mathbb{1}(A_i = a)}{\sum_{i=1}^{t-1} \mathbb{1}(A_i = a)}$$

We call this the *sample-average* method for estimating action values becuase each estimate is an average of the sample of relevant rewards. To select an action, the simplest approach is to select one of the actions with the highest estimated value, i.e., one of the greedy actions as defined in the previous section. We write this *greedy* action selection method as

$$A_t \equiv \arg\max_a Q_t(a)$$

A simple alternative is to behave greedily most of the time, but every once in a while, with a small probability ε, instead select randomly from all the actions with equal probability, independently of the action-value estimates. These methods are named ε-*greedy*.

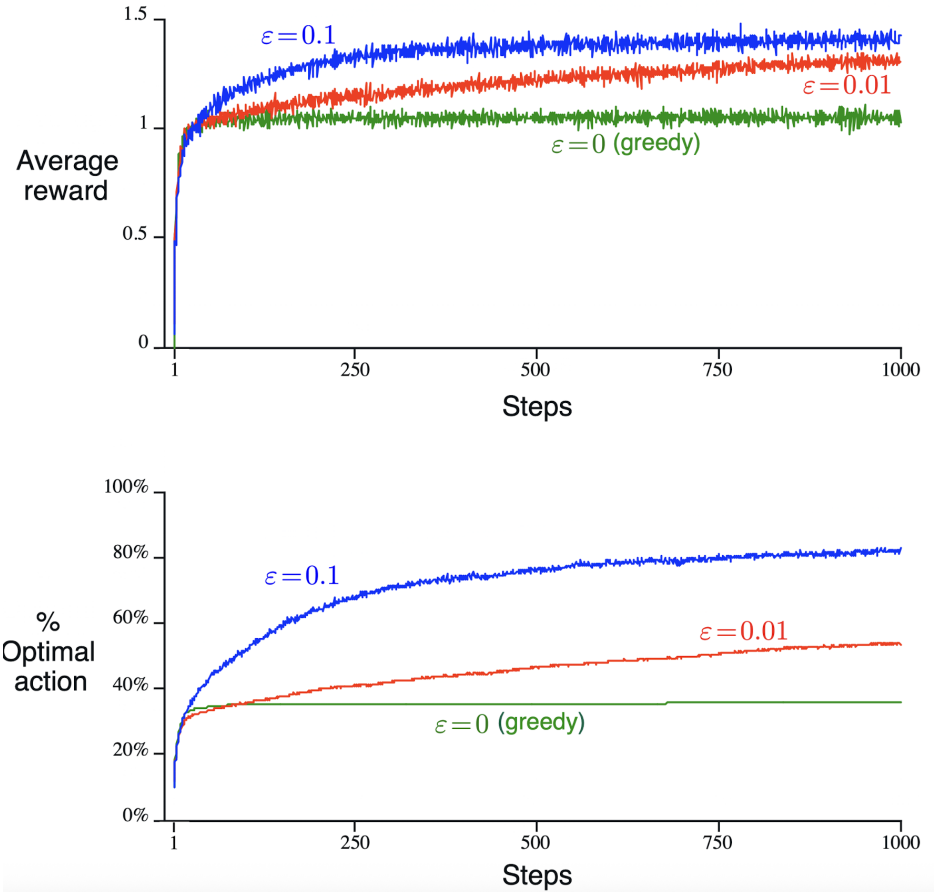


Figure 1.2: Comparison between greedy and ε-greedy action selection methods. The ε-greedy method explores the environment by selecting random actions with a small probability ε, while the greedy method always selects the action with the highest estimated value.

Nonstationary Bandit Problems

Previous methods assume that the action values are stationary, meaning that they do not change over time. However, in many real-world scenarios, the action values can change due to various factors such as changes in the environment or the agent's own actions. This leads to nonstationary bandit problems, where the action values are not fixed and can vary over time. In such cases it makes sense to give more weight to recent rewards than to long-past rewards. One of the most popular ways of doing this is to use a constant step-size parameter.

$$Q_{n+1}(a) = Q_n(a) + \alpha(R_{n+1} - Q_n(a))$$

where $\alpha \in [0,1]$ is a constant step-size parameter that determines how much weight to give to the most recent reward. This approach allows the agent to adapt to changes in the environment and to update its action-value estimates accordingly.

Optimistic Initial Values

All the methods we have discussed so far are dependent to some extent on the initial action-value estimates, $Q_1(a)$. In the language of statistics, these methods are *biased* by their initial estimates. Initial action-values can also be used as a simple way to encourage exploration. Setting the initial action values to be optimistic, i.e., higher than the true expected rewards, can encourage the agent to explore more. This is because the agent will initially prefer actions with higher estimated values, leading to more exploration of those actions until their true values are learned. We regard it as a simple trick that can be quite effective on stationary problems, but it is far from being a generally useful approach to encouraging exploration.

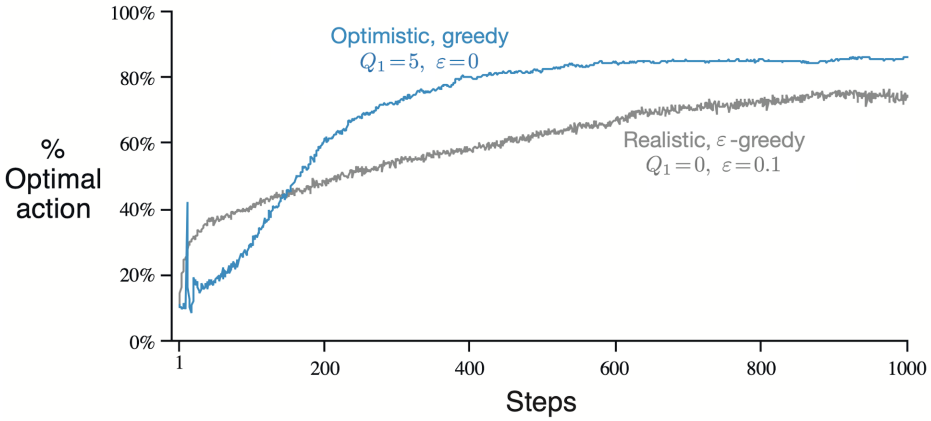


Figure 1.3: Optimistic initial values in action-value methods. The agent starts with high initial estimates for all actions, encouraging exploration until the true values are learned.

Gradient Bandit Algorithms

Here we consider learning a numerical *preference* for each action a, which we denote as $H_t(a) \in \mathbb{R}$. The larger the preference, the more often that action is taken, but the preference has no interpretation in terms of reward. Only the relative preference of one action over another is important.

$$Pr\{A_t = a\} = \frac{e^{H_t(a)}}{\sum_b e^{H_t(b)}} = \pi_t(a)$$

where $\pi_t(a)$ indicates the probability of taking aciton a at time step t. The action probabilities are normalized so that they sum to one.

There is a natural learning algorithm for soft-max action preferences based on the idea of stochastic gradient ascent. On each step, after selecting action A_t and receiving the reward R_t, the action preferences are updated by:

$$H_{t+1}(a) = H_t(a) + \alpha(R_t - \bar{R}_t)(1 - \pi_t(A_t)), \quad \text{and}$$

$$H_{t+1}(A_t) = H_t(A_t) - \alpha(R_t - \bar{R}_t)\pi_t(a) \quad \text{for all } a \neq A_t$$

where $\bar{R}_t$ is the average reward received up to time step t. This update rule adjusts the preferences based on the received reward, encouraging actions that yield higher rewards and discouraging those that yield lower rewards.

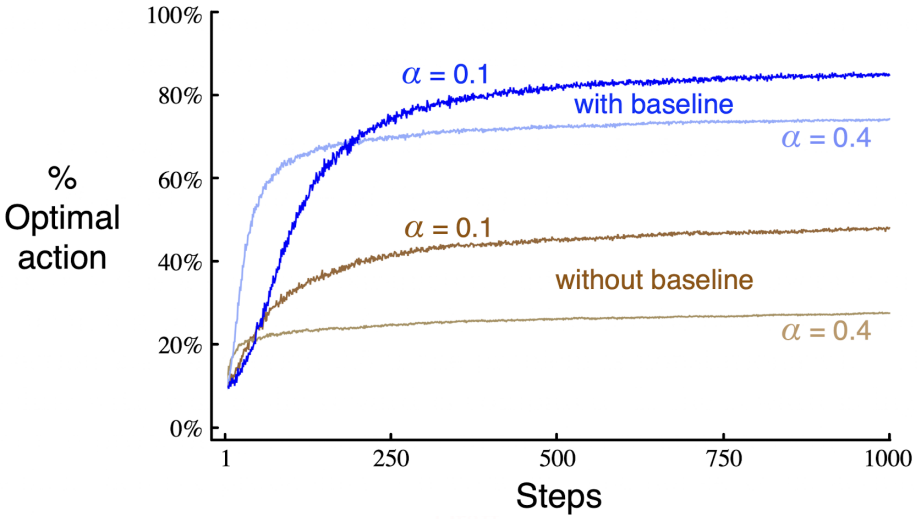


Figure 1.4: Gradient bandit algorithms for action selection. The agent learns preferences for actions based on the received rewards, using a soft-max policy to select actions probabilistically.

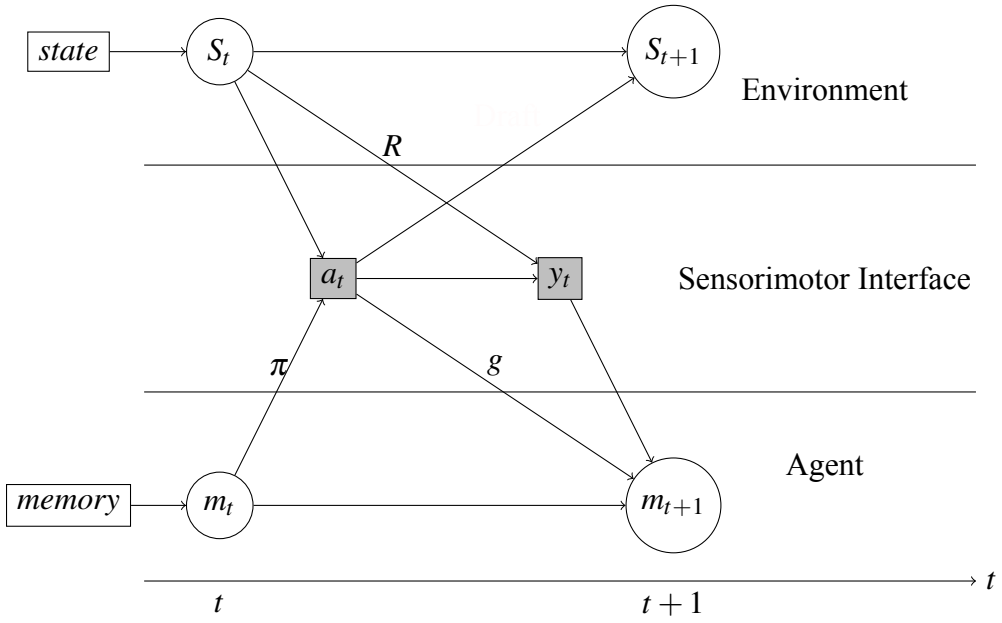
2 Markov Decision Processes

MDPs are a classical formalization of sequential decision making, where actions influence not just immediate rewards but also subsequent situations or states, and through those future rewards. Whereas in bandit problems we estimated the value $q_*(a)$ of each action a, in MDPs we estimate the value $q_*(s,a)$ of each action a in each state s, or we estimate the value $v_*(s)$ of each state given optimal action selections.

2.1 Interface

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.

General scheme of a *Decision Process*:



- $\pi(a|m) \rightarrow$ policy
- $R(y) \rightarrow$ reward function
- $p(s',y|s,a) \rightarrow$ model of the environment
- $g(m'|m,a,y) \rightarrow$ memory update

The goal is to find the optimal policy π^* that maximizes the expected return:

$$\arg\max_{\pi} \underbrace{\mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t R(y_t)\right]}_{ExpectedReturn} \qquad 0 \leq \gamma < 1$$

with γ survival probability.
The expected survival time is:

$$\frac{1}{1-\gamma}$$

Specifications:

- **Perfect observability** $\rightarrow$ the agent knows the state of the environment ($y = S$) and $p(y|s,a,s') = \infty(y = s')$

Observation:

$$p(s',y|s,a) = p(s'|s,a)p(y|s,a,s')$$

- **Memory update** $\rightarrow$ the agent knows the state of the environment and the memory ($M = y$) and $g(m'|m,a,y) = \infty(m' = y)$

At each time step t, the agent receives some representation of the environment's *state* $S_t \in \mathscr{S}$, and on that basis selects an *action* $A_t \in \mathscr{A}(s)$. One time step later, in part as a consequence of its action, the agent receives a numerical *reward* $R_{t+1} \in \mathscr{R} \subset \mathbb{R}$, and finds itself in a new state S_{t+1}.

Definition: ***Markov Decision Process***

A **Markov Decision Process (MDP)** is a fully observable set of tuples (S,A,R,P,γ) where:

- $s \in S$ is a finite set of states
- $a \in A$ is a finite set of actions
- $R : S \times A \rightarrow \mathbb{R}$ is the reward function
- $P : S \times A \times S \rightarrow [0,1]$ is the transition probability function
- $\gamma \in [0,1]$ is the discount factor
- $p(s',y|s,a)$ is the model of the environment
- $p_0(s)$ is the initial state distribution
- $\pi(a|s)$ is the policy

In a *finite* MDP, the sets of states, actions and rewards all have a finite number of elements.

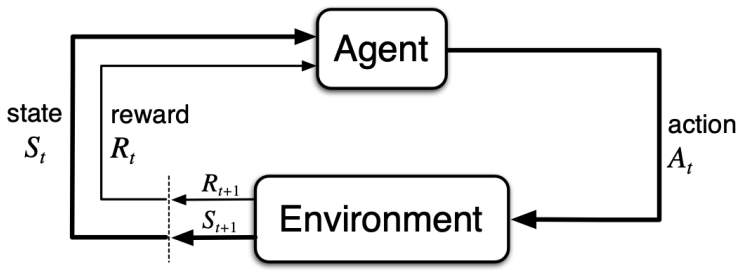


Figure 2.1: Markov Decision Process

In this case, the random variables R_t and S_t have well defined discrete probability distributions dependent only on the preceding state and action.

$$p(s',r|s,a) = Pr(S_t = s', R_t = r|S_{t-1} = s, A_{t-1} = a)$$

This function defines the dynamics of the MDP, it specifies a probability distribution for each choice of s and a and has four arguments.

$$\sum_{s' \in \mathscr{S}} \sum_{r \in \mathscr{R}} p(s', r|s, a) = 1 \text{ for all } s \in \mathscr{S}, a \in \mathscr{A}$$

The state must include information about all aspects of the past agent-environment interaction that make a difference for the future. If it does, then the state is said to have the *Markov property*.
From the four-argument dynamics function, one can obtain:

- the *state-transition probabilities* $p : \mathscr{S} \times \mathscr{S} \times \mathscr{A} \to [0,1]$

$$p(s'|s,a) = Pr\{S_t = s' | S_{t-1} = s, A_{t-1} = a\} = \sum_{r \in \mathscr{R}} p(s', r|s, a)$$

- the expected rewards for state-action pairs $r : \mathscr{S} \times \mathscr{A} \to \mathbb{R}$:

$$r(s,a) = \mathbb{E}[R_t | S_{t-1} = s, A_{t-1} = a] = \sum_{r \in \mathscr{R}} r \sum_{s' \in \mathscr{S}} p(s', r|s, a)$$

- the expected rewards for state-action-next-state triples $r : \mathscr{S} \times \mathscr{A} \times \mathscr{S} \to \mathbb{R}$:

$$r(s,a,s') = \mathbb{E}[R_t | S_{t-1} = s, A_{t-1} = a, S_t = s'] = \sum_{r \in \mathscr{R}} r \frac{p(s', r|s, a)}{p(s'|s, a)}$$

In general, actions can be any decisions we want to learn how to make, and states can be anything we can know that might be useful in making them. In particular, the boundary between agent and environment is typically not the same as the physical boundary of a robot's or an animal's body. Usually, the boundary is drawn closer to the agent for that.
Generally, anything that cannot be changed arbitrarily by the agent is considered to be outside of it and thus part of its environment. We do not assume that everything in the environment is unknown to the agent.
The MDP framework proposes that whatever the details of the sensory, memory and control apparatus, and whatever objective one is trying to achieve, any problem of learning goal-directed behavior can be reduced to three signals passing back and forth between the agent and its environment:

- **actions:** choices made by the agent
- **states:** basis on which the choices are made
- **rewards:** the agent's goal

2.2 Goals and Rewards

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. We can clearly state this informal idea as the *reward hypothesis*:

> *That all of what we mean by goals and purposes can be well thought of as the maximization of the expected value of the cumulative sum of a received scalar signal (called reward).*

In particular, the reward signal is not the place to impart to the agent prior knowledge about *how* to achieve what we want it to do. For example, a chess-playing agent should be rewarded only for actually winning, not for achieving subgoals such as taking its opponent's pieces or gaining control

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

2.3 Returns and Episodes

If the sequence of rewards received after time step 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? In general, we seek to maximize the *expected return*, where the return, denoted as G_{t+1} is defined as some specific function of the reward sequence (most of the cases the sum of the rewards).

This approach makes sense in applications in which there is a natural notion of final time step, i.e., when the agent-environment interaction breaks naturally into sequencces called *episodes*. Each one ends in a special state called the *terminal state*, followed by a reset to a standard starting state or to a sample from a standard distribution of starting states. The next episode begins independently of how the previous one ended. Tasks with episodes of this kind are called *episodic tasks* and in them we need to distinguish the set of nonterminal states ($\mathscr{S}$), from the set of all states plus the terminal state ($\mathscr{S}^+$).

On the other hand, in many cases the agent-environment interaction does not break naturally into identifiable episodes, but goes on continually without limit. These are the *continuing tasks*. The return formulation is problematic for these because the final time step would be $T = \infty$, such as the the return.

The additional concept that we need is that of *discounting*. According to this approach, the agent tries to select actions so that the sum of the discounted rewards it receives over the future is maximized. In particular, it chooses A_t to maximize the expected *discounted return*:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} R_{t+k+1},$$

where γ is a parameter defined in $[0,1]$ called the *discount rate*.

It determines the present value of future rewards: a reward received k time steps in the future is worth only γ^{k-1} times what it would be worth if it were received immediately.

Returns at successive time steps are related to each other in a way that is important for the theory and algorithms of RL:

$$\begin{aligned} G_t &= R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} + \ldots \\ &= R_{t+1} + \gamma G_{t+1} \end{aligned}$$

where $G_{t+1} = R_{t+2} + \gamma R_{t+3} + \gamma^2 R_{t+4} + \ldots$ and if the reward is a constant $+1$, the return is finite

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

2.4 Unified Notation for Episodic and Continuing Tasks

We number the time steps of each episode starting anew from zero, referring thus not to S_t but to $S_{t,i}$ at time t of episode i. The same for the other parameters. However, it turns out that when we discuss episodic tasks we almost never have to distinguish between different episodes. In practice we almost always abuse notation slightly by dropping the explicit reference to episode number, i.e., we write S_t to refer to $S_{t,i}$ and so on.

The sums for both episodic and continuing tasks can be unified considering episode termination to be the entering of a special *absorbing state* that transitions only to itself and that generates only rewards of zero. Consider the following transition diagram:

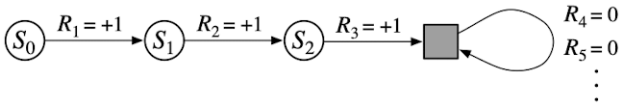


Figure 2.2: Transition diagram for an absorbing state

We can alternatively write

$$G_t = \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k,$$

including the possibility that $T = \infty$ or $\gamma = 1$, but not both.

2.5 Policies and Value Functions

Almost all RL algorithms involve estimating *value functions*, i.e., 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 deifned in terms of future rewards that can be expected or in terms of expected return. They 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, then $\pi(a|s)$ is the probability that $A_t = a$ if $S_t = s$.

The *value function* 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

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

where $\mathbb{E}_\pi [.]$ denotes the expected value of a random variable given that the agent follows policy π and t is any time step.

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:

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

We call q_π the *action-value function* for policy π.

The value functions v_π and q_π can be estimated from experience by averaging the returns G_t observed after each time step t when the agent is following policy π. This leads to the so called *Monte Carlo methods*.

Observation:

A fundamental property of value functions used throughout RL and dynamic programming is that they satisfy recursive relationships similar to that which we have already estabilished for the return. 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 state:

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

Note how in the last equation we have merged the two sums, one over all the values of s' and the other over all the values of r, into one sum over all possible values of both. For each triple (a,s',r) we compute its probability $\pi(a|s)p(s',r|s,a)$, weight the quantity in brackets by that probability and then sum over all possibilities to get an expected value.

Note that the last equation in the observation is the so called **Bellman equation** for $v\pi$. It expresses a relationship between the value of a state and the values of its successor states, averaging over all the possibilities, weighting each by its probability of occurring.

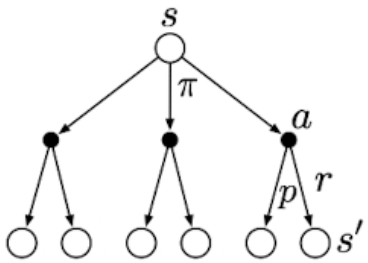


Figure 2.3: Backup diagram for v_π

2.6 Optimal Policies and Value Functions

Solving a RL task means, roughly, finding a policy that achieves a lot of reward over the long run. 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. There is always at least one policy that is better than or equal to all other policies, and that is the *optimal policy* π_*. The optimal policies share the same state-value function, called the *optimal state-value function*, denoted v_* and defined as

$$v_*(s) = \max_\pi v_\pi(s), \text{ for all } s\in\mathcal{S}.$$

Optimal policies also share the same *optimal action-value function*, denoted q_* and defined as

$$q_*(s,a) = \max_\pi q_\pi(s,a), \text{ for all } s\in\mathcal{S}, a\in\mathcal{A}(s).$$

Thus, we can write q_* in terms of v_* as follows:

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

Because v_* is the value function for a policy, it must satisfy the self-consistency condition given by the Bellman equation for state values. It can be written in a special form without reference to any specific policy. This is the Bellman equation for v_*, or the *Bellman optimality equation*, and expresses the fact that the value of a state under an optimal policy must equal the expected return for the best action from that state.

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

The Bellman optimality equation for q_* is

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

For finite MDPs, the Bellman equations for v_* and q_* have unique solutions. The optimal policies can be derived from these value functions.

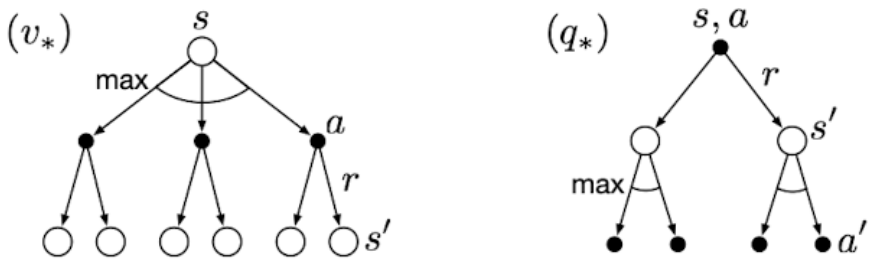


Figure 2.4: Backup diagram for v_* and q_*

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. Any policy that assigns nonzero probability only to these actions is an optimal policy. 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 simpli find any action that maximizes $q_*(s,a)$, and that will be an optimal action.

Warning:

Explicitly solving the Bellman optimality equation provides one route to finding an optimal policy, and thus to solving the RL problem. However, this solution is rarely direcly usegful. This solution relies on at least three assumptions that are rarely true in practice:

- the dynamics of the environment are accurately known
- computational resources are sufficient to complete the calculation
- the states have the Markov property

3 Dynamic Programming

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). 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, but they are still important theoretically.

We usually assume that the environment is a finite MDP. That is, we assume that its state, action and reward sets are finite and that its dynamics are given by a set of probabilities $p(s',r|s,a)$, for all $s \in \mathscr{S}, a \in \mathscr{A}, r \in \mathscr{R}, and s' \in \mathscr{S}$. A common way of obtaining approximate solutions for tasks with continuous states and actions is to quantize the state and action spaces and then apply finite-state DP methods. The key idea of DP is the value functions to organize and structure the search for good policies.

We can easily obtain optimal policies once we found the optimal value functions v_* or q_* which satisfy the Bellman optimality equations:

$$v_*(s) = \max_a \sum_{s',r} p(s',r|s,a)\left[r + \gamma v_*(s')\right]$$
$$q_*(s,a) = \sum_{s',r} p(s',r|s,a)\left[r + \gamma v_*(s')\right]$$

As we shall see, DP algorithms are obtained by turning Bellman equations such as these into assignments, i.e., into update rules for improving approximations of the desired value functions.

3.1 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.

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

where $\pi(a|s)$ is the probability of taking action a in state s under policy π.

If the environment's dynamics are completely known, then the last equation above is a system of

$$|S|$$

linear equations in the

$$|S|$$

unknowns $v_\pi(s)$, one for each state $s \in \mathscr{S}$. Iterative solutions here are most suitable. The initial approximation v_0 is chosen arbitrarily, while the terminal state must be 0. Each successive

approximation is obtained by using the Bellman equation for v_π as an updated rule:

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

The sequence 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 the **iterative policy evaluation**.
It essentially replaces the old value of s with a new value obtained from the old values of the successor states of 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*. All the updates done in DP algorithms are called *expected* updates because they are based on an expectation over all possible next states rather than on a sample next state.
A complete in-place version of iterative policy evaluation is shown in pseudocode in the box below.

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

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

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

Figure 3.1: Iterative Policy Evaluation Algorithm

3.2 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 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

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

If it is greater than $v_\pi(s)$ 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 called the *policy improvement theorem*. The policy π' must be as good as, or better than, π, i.e., it must obtain greater or equal expected return from all states $s \in \mathcal{S}$.
It is a natural extension to consider changes at all states, selecting at each state the action that appears best according to $q_\pi(s,a)$. In other words, to consider the new greedy policy π' given by:

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

The greedy policy takes the action that looks best in the short term, according to v_π. The process of making a new policy that improves on an original policy, by making it greedy with respect to the value function of the original policy, is called **policy improvement**.

3.3 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:

$$\pi_0 \to^E v_{\pi_0} \to^I \pi_1 \to^E v_{\pi_1} \to^I \pi_2 \to^E \ldots \to^I \pi_* \to^E v_*,$$

where the superscripts E and I indicate the expected update of the value function and the improvement of the policy, respectively. Because a finite MDP has only a finite number of deterministic policies, this process must converge to an optimal policy and the optimal value function in a finite number of iterations. This way of finding an optimal policy is called **policy iteration**.

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

1. Initialization
   $V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$; $V(terminal) \doteq 0$

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

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

3.4 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. 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, and 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:

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

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.

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

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

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

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

Figure 3.2: Value Iteration Algorithm

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

3.5 Asynchronous DP

A major drawback to the DP methods that we have discussed so far is that they involve operations over the entire state set of the MDP, i.e., they require sweeps of the state set. If the state set is very large, then even a single sweep can be prohibitively expensive.

Asynchronous DP algorithms are in-place iterative DP algorithms that are not organized in terms of systematic sweeps of the state set. These algorithms update the values of states in any order whatsoever, using whatever values of other states happen to be available. The values of some states may be updated several times before the values of others are updated once. To converge correctly, however, an asynchronous algorithm must continue to update the values of all the states: it cannot ignore any state after some point in the computation. Asynchronous DP algorithms allow great flexibility in selecting states to update.

Avoiding sweeps does not necessarily mean that we can get away with less computation. It just means that an algorithm does not need to get locked into any hopelessly long sweep before it can make progress improving a policy. We can try to take advantage of this flexibility by selecting the states to which we apply updates so as to improve the algorithm's rate of progress. We can try to order the updates to let value information propagate from state to state in an efficient way. Some states may not need their values updated as often as others.

3.6 Generalized Policy Iteration

Policy iteration consists of two simultaneous, interacting processes: one making the value function consistent with the current policy (policy evaluation), and the other making the policy greedy with respect to the current value function (policy improvement). In policy iteration, these two processes alternate, each completing before the other begins, even though this is not really necessary.
We use the term **generalized policy iteration (GPI)** to refer to the general idea of letting policy-evaluation and policy-improvement processes interact, independent of the granularity and other details of the two processes. Almost all RL methods are well described as GPI. That is, all have identifiable policies and value functions, with the policy always being improved with respect to the value function for the policy. If both the evaluation process and the improvement process stabilize, then the value function and policy must be optimal.

3.7 Linear Programming for MDP

Primal: optimize a function $f(u)$ where $u \in \mathbb{R}^{|S|}$

$$f(u) = \sum_s p_0(s)u(s)$$

$\Omega : \left\{ u \in \mathbb{R}^{|S|} : \sum_s p(s'|sa)(r(sas') + \gamma u(s')) - u(s) \leq 0 \quad \forall sa \right\}$
Ω is a polyhedron and a convex set. The objective is to minimize the function $f(u)$ over the set Ω.

Observation:
If $u \in \Omega$ we call it a **Feasible Point**.

The solution of the primal MDP linear programming = optimal solution of the MDP

$$\max_\pi G_\pi(p_0) = f(V^*) = \min_{u \in \Omega} f(u)$$

- V^* is feasible : $V^* \in \Omega \rightarrow f(V^*) \geq \min_{u \in \Omega} f(u)$ Since:

$$\sum_{s'} p(s'|sa)(r(sas') + \gamma V^*(s')) \leq \max_a \left[\sum_{s'} p(s'|sa)(r(sas') + \gamma V^*(s')) \right] = V^*(s) \quad \forall sa$$

- if u is feasible, then $u \geq V^*$ component wise (meaning $\forall s \quad u(s) \geq V^*(s)$).

$$\begin{gathered} if \quad u \in \Omega \\ u(s) - V^*(s) \geq \sum_{s'} p(s'|sa^*)(f(sa^*s') + \gamma u(s')) - \sum_{s'} p(s'sa^*)(r(sa^*s') + \gamma V^*(s')) \\ = \gamma \sum_{s'} p(s'|sa^*s')(u(s') - V^*(s')) \geq \gamma \sum_{s'} p(s'|sa^*) \min(u - V^*) \\ u(s) - V^*(s) \geq \gamma \min(u - V^*) \rightarrow (1-\gamma) \min(u - V^* \geq 0) \rightarrow f(u) \geq f(V^*) \end{gathered}$$

Primal problem:

$$\min_{u \in \Omega} f(u)$$

Can we go from this constraint problem to an unconstrained problem? Yes.

$$\min_{u \in \Omega} f(u) \rightarrow \min_{u \in \mathbb{R}^{|S|}} [f(u) + h(u)] \quad h(u) = \begin{cases} +\infty & \text{if} \quad u \notin \Omega \\ 0 & \text{if} \quad u \in \Omega \end{cases}$$

$$h(u) = \max_{Z \geq 0} \left[\sum_{sa} z(sa) \left(\sum_{s'} p(s'|sa)(r(sas') + \gamma u(s')) - u(s) \right) \right]$$

thus:

$$\min_{u\in\Omega} f(u) = \min_{u}\max_{Z\geq 0} \mathbb{L}(u,Z)$$

$$\begin{aligned}
\mathbb{L}(u,Z) &= \textstyle\sum_{sa} Z(sa)p(s'|sa)r(sas') + \sum_{s'} u(s')(p(s) + \gamma\sum_{s'} p(s'|sa)Z(sa) - \sum_{a'} Z(s'a)) \\
&= \max_{Z\geq 0}\left[\textstyle\sum_{sa} z(sa)p(s'|sa)r(sas') + \min_u\left[\sum_{s'} u(s')\left(p(s') + \gamma\sum_{s'} p(s'|sa)Z(sa) - \sum_{a'} Z(s'a)\right)\right]\right] \\
&= \max_{Z\geq 0}\left[\textstyle\sum_{sa} Z(sa)p(s'|sa)r(sas')\right] \rightarrow \textbf{Dual MDP Problem} \\
&= \left\{Z\in\mathbb{R}^{|s|} \quad : \quad Z\geq 0, \quad p_0(s') + \gamma\textstyle\sum_{sa} p(s'|sa)Z(sa) - \sum_{a'} z(s'a') = 0 \quad \forall s'\right\}
\end{aligned}$$

Then:

$$\begin{aligned}
&\max_\pi G_\pi(p_0) \rightarrow G_\pi(p_0) = \textstyle\sum_{t=0}^{\infty}\gamma^t\sum_{sas'} p_t(s)\pi(a|s)p(s'|sa)r(sas') \\
&\eta_\pi(s'a) \cong \textstyle\sum_{t=0}^{\infty}\gamma^t p_t(s)\pi(a|s) = \mathbb{E}(\sum_{t=0}^{\infty}\gamma^t\mathbb{1}(s_tA_t = sa)) \\
&= \textstyle\sum_{sas'}\eta_\pi(sa)p(s'|sa)r(sas') \\
&z\rightarrow\eta_\pi
\end{aligned}$$

4 Monte Carlo Methods

Here we do not assume complete knowledge of the environment. Monte Carlo methods require only **experience**, meaning sample sequences of states, actions and rewards from actual or simulated interaction with an environment. Learning from *actual* experience is striking because it requires no prior knowledge of the environment's dynamics, yet can still attain optimal behavior. Learning from *simulated* experience is also powerful. Altough a morel is required, the model need only generate sample transitions, not the complete probability distributions of all possible transitions that is required for dynamic programming (DP).

Monte Carlo methods (MC) are ways of solving the RL problem based on averaging sample returns. Here are defined only for episodic tasks. Only on the completion of an episode are value estimates and policies changed.

Essentially, MC methods sample and average *returns* for each state-action pair much like the bandit methods which sample and average rewards for each action. The main difference is that now there are multiple states, each acting like a different bandit problem and the different bandit problems are interrelated. That is, the return after taking an action in one state depends on the actions taken in later states in the same episode. Because all the action selections are undergoing learning, the problem becomes nonstationary from the point of view of the earlier state.

To handle the nonstationarity, we adapt the idea of general policy iteration (GPI) developed earlier for DP. Whereas there we computed value functions from knowledge of the MDP, here we *learn* value functions from sample returns with the MDP.

4.1 Monte Carlo Prediction

We begin by considering MC methods for learning the state-valiue function for a given policy. An obvious way to estimate it from experience 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. 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. The *first-visit MC method* estimates $v_\pi(s)$ as the average of the returns following the first visits to s, whereas the *every-visit MC method* averages the returns following all visits to s.

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

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

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

Both first-visit and every-visit MC converge to $v_\pi(s)$ as the number of visits to s goes to infinity.
Can we generalize the idea of backup diagram to MC algorithms? The general idea of a backup diagram is to show at the top the root node to be updated and to show below all the transitions and leaf nodes whose rewards and estimated values contribute to the update. For MC estimation of v_π, the root is a state node, and below it is the entire trakectory of transitions along a particular single episode, ending at the terminal state. An important fact about MC methods is that the estimates for each state are independent. MC methods do not bootstrap as we defined it in the previous chapter.
Note that the computational expense of estimating the value of a single state is independent of the number of states. This can make MC methods particularly attractive when one requires the value of only one or a subset of states. One can generate many sample episodes starting from the states of interest, averaging returns from only these states, ignoring all others.

4.2 Monte Carlo Estimatino of Action Values

If a model is not available, then it is particularly useful to estiamte action values rather than state values. With a model, state values alone are sufficient to determine a policy; one simply looks ahead one step and chooses whichever action leads to the best combination of reward and next state. Without a model, however, state values alone are not sufficient. One must explicitly estimate the value of each action in order for the values to be useful in suggesting a policy.
The policy evaluation problem for action values is to estimate $q_\pi(s,a)$, the expected return when starting in state s, taking action a and thereafter following policy π. A state-action pair s,a is said to be visited in an episode if ever the state s is visited and action a is taken in it.
The only complication is that many state-action pairs may never be visited. If π is a deterministic policy, then following π one will observe returns only for one of the actions from each state. With no return to average, the MC estimates of the other actions will not improve with experience. This is a serious problem because the purpose of learning action values is to help in choosing among the actions available in each state. This is the general problem of **maintaininig exploration**.
For policy evaluation to work for action values, we must assure continual exploration. One way to do this is by specifying the episodes start in a state-action pair., and that every pair has a nonzero probability of being selected as the start. This guarantees that all state-action pairs will be visited an infinite number of times in the limit of an infinite number of episodes. We call this the assumption of **exploring starts**.

4.3 Monte Carlo Control

Let's consider a MC version of classical policy iteration. In this method, we perform alternating complete steps of policy evaluaton and policy improvement, beginning with an arbitrary policy π_0 and ending with the optimal policy and optimal action-value function:

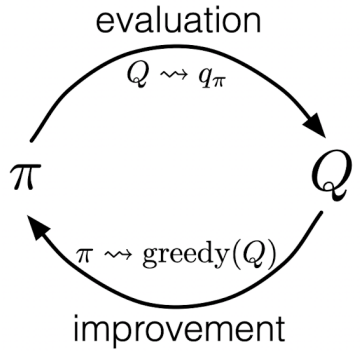
$$\pi_0 \to^E q_{\pi_0} \to^I \pi_1 \to^E q_{\pi_1} \to^I \pi_2 \to^E \ldots \to^I \pi_* \to^E q_*,$$

where $\to^E$ denotes a complete policy evaluation and $\to^I$ denotes a complete policy improvement.
Policy evaluation is done exactly as described in the preceding section. Many episodes are experienced, with the approximate action-value function approaching the true function asymptotically. 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 MC methods will compute each q_{π_k} exactly, for arbitraty π_k.
Policy improvement is done by making the policy greedy with respect to the current value function. In this case we have an acion-value function, and therefore no model is needed to construct the

greedy policy. For any given 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:

$$\pi(s) = \arg\max_a q(s,a)$$

Policy improvement then can be done by constructing each π_{k+1} as the greedy policy with respect to q_{π_k}.



The first strong assumption we made (exploring starts) is easy to remove. In both DP and MC cases there are two ways to solve the problem. One is to hold firm to the idea of approximating q_{π_k} in each policy evaluation. Measurements and assumptions are made to obtain bounds on the magnitude and probability of error in the estimates, and then sufficient steps are taken during each policy evaluation to assure that these bounds are sufficiently small.
The second approach is to give up trying to complete policy evaluation before returning to policy improvement. On each evaluation step we move the value function toward q_{π_k}, but we don't expect to actually get close except over many steps. Below is the pseudocode of the so called **Monte Carlo with Exploring Starts (MCES)**.

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

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

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

Figure 4.1: Monte Carlo with Exploring Starts (MCES)

4.4 Monte Carlo Control without Exploring Starts

How can we avoid the unlikely assumption of exploring starts? The only general way to ensure that all acitons are selected infinitely often is for the agent to continue to select them. There are two approaches:

- **On-policy methods** - attempt to evaluate or improve the policy that is used to make decisions
- **Off-policy methods** - evaluate or improve a policy different from the one used to generate the data

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 control method we present in this section uses ε-greedy policies, meaning that mist of the time they choose an action that has maximal estimated action value, but with probability ε they instead select an action at random.
As in MCES, 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 nongreedy 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.

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

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

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

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

Figure 4.2: Monte Carlo Control with ε-greedy policies

So, the policy ieration workds for ε-soft policies. Using the natural notion of greedy policy for ε-soft policies, one is assured of improvement on every step, except when the best policy has been found among the ε-soft policies. This analysis is independent of how the action-value functions are determined at each stage, but it does assume that they are computed exactly. Now we only achieve the best policy among the ε-soft policies, but on the other hand, we have eliminated the assumption of exploring starts.

4.5 Off-policy Prediction via Importance Sampling

All learning control methods face a dilemma: they seek to learn action values conditional on subsequent optimal behavior, but they need to behave non-optimally in order to explore all actions. How can they learn about the optimal policy while behaving according to an exploratory policy? The on-policy approach in the preceding section is actually a compromise, i.e., it learns action values not for the optimal policy but for a near-optimal policy that still explores. A more straightforward

approach is to use two policies, one that is learned about and that becomes the optimal policy, and one that is more exploratory and is used to generate behavior. The first is the **target policy**, while the second is the **behavior policy**. In this case we say that learning is from data "off" the target policy, and the overall process is termed *off-policy learning*.

Off-policy methods are often of greater variance and are slower to converge. On the other hand, off-policy methods are more powerful in general.

We begin by considering the *prediction* problem, where both target and behavior policies are fixed. Suppose we wish to estimate v_π or q_π, but all we have are episodes following another policy b, where $b \neq \pi$. In this case, π is the target policy, b is the behavior policy, and both policies are considered fixed and given.

We require that every action taken under π is also taken, at least occasionally, under b. This is called the assumption of *coverage*. It follows from coverage that b must be stochastic in states where it is not identical to π. The target policy π, on the other hand, may be deterministic.

Almost all off-policy methods utilize **importance sampling**, a general technique for estimating expected values under one distribution given samples from another. We apply importance sampling to off-policy learning by weighting returns according to the relative probability of their trajectories occurring under the target and behavior policies, called the *importance-sampling ratio*. Given a starting state S_t, the probability of the subsequent state-action trajectory occurring under any policy π is:

$$\begin{aligned} &Pr\{A_t, S_{t+1}, A_{t+1}, \dots, S_T | S_t, A_{t:T-1} \approx \pi\} \\ &\quad = \pi(A_t|S_t)\, p(S_{t+1}|S_t, A_t)\, \pi(A_{t+1}|S_{t+1}) \dots p(S_T|S_{T-1}, A_{T-1}) \\ &\quad = \prod_{k=t}^{T-1} \pi(A_k|S_k)\, p(S_{k+1}|S_k, A_k) \end{aligned}$$

where p here is the state-transition probability function.

The importance-sampling ratio is:

$$\rho_{t:T-1} = \frac{\prod_{k=T}^{T-1} \pi(A_k|S_k)\, p(S_{k+1}|S_k, A_k)}{\prod_{k=t}^{T-1} b(A_k|S_k)\, p(S_{k+1}|S_k, A_k)} = \prod_{k=t}^{T-1} \frac{\pi(A_k|S_k)}{b(A_k|S_k)}$$

Recall that we wish to estimate the expected returns (values) under the target policy, but all we have are returns G_t due to the behavior policy. These returns have the wrong expectation $\mathbb{E}[G_t|S_t = s] = v_b(s)$ and so cannot be averaged to obtain v_π This is where importance sampling comes in. The ratio $\rho_{t:T-1}$ transforms the returns to have the right expected value:

$$\mathbb{E}[\rho_{t:T-1} G_t | S_t = s] = v_\pi(s)$$

We can define the set of all time steps in which state s is visited, denoted $\mathcal{T}(s)$. Also, let $T(t)$ denote the first time of termination following time t, and G_t denote the return after t up through $T(t)$. Then $\{G_t\}_{t \in \mathcal{T}(s)}$ are the returns that pertain to state s, and $\left\{\rho_{t:T(t)-1}\right\}_{t \in \mathcal{T}(s)}$ are the corresponding importance-sampling rations. To estimate $v_\pi(s)$, we simply scale the returns by the ratios and average the results:

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

When importance sampling is done as a simple average in this way it is called *ordinary importance sampling*.

An important alternative is *weighted importance sampling*, which uses a weighted average, defined as:

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

4.6 Incremental Implementation

MC prediction methods can be implemented incrementally, on an episode-by-episode basis, using extensions of the techniques described in Chapter 3. In MC methods we average returns. In all other aspects the same methods of Chapter 3 can be used for on-policy MC methods. For off-policy MC methods, we need to separately consider those that use ordinary importance sampling and those that use weighted importance sampling.

In ordinary importance sampling, the returns are scaled by the importance sampling ratio $\rho_{t:T(t)-1}$, then simply averaged. This leaves the case of off policy methods using weighted importance sampling. Here we have to form a weighted average of the returns, and a slightly different incremental algorithm is required. We let the definition and demonstration of this algorithm be left in the book [1].

4.7 Off-policy Monte Carlo Control

Recall that the distinguishing feature of on-policy methods is that they estimate the value of a policy while using it for control. In off-policy methods these two functions are separated. The policy used to generate behavior, called the *behavior policy*, may in fact be unrelated to the policy that is evaluated and improved, called the *target policy*. An advantage of this separation is that the target policy may be deterministic (e.g. greedy), while the behavior policy can continue to sample all possible actions.

These techniques require that the behavior policy has a nonzero probability of selecting all actions that might be selected by the target policy (coverage). To explore all possibilities, we require that the behavior policy be soft.

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

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

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

Figure 4.3: Off-policy Monte Carlo Control

A potential problem is that this method learns only from the tails of episodes, when all of the remaining actions in the episode are greedy. If nongreedy actions are common, then learning will be slow, particularly for states appearing in the early portions of long episodes.

5 Temporal-Difference 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 estimated, without waiting for a final outcome (they bootstrap).
As usual, we start by focusing on the policy evaluation or *prediction* problem, the problem of esimating the value function v_π for a given policy π. For the *control* problem, DP, TD and Monte Carlo methods all use some variation of generalized policy iteration (GPI).

5.1 TD Prediction

Both TD and Monte Carlo methdos 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.

$$V(S_t) \leftarrow V(S_t) + \alpha \left[G_t - V(S_t) \right] \tag{5.1}$$

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

$$V(S_t) \leftarrow V(S_t) + \alpha \left[R_{t+1} + \gamma V(S_{t+1} - V(S_t)) \right] \tag{5.2}$$

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

Tabular TD(0) for estimating v_π

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

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

Figure 5.1: Pseudocode for one-step TD learning.

Because TD(0) methods bases its update in part on an existing estimate, we say that it is a *bootstrapping* method, like DP.

$$v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s] \tag{5.3}$$
$$= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1}|S_t = s]$$
$$= \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1})|S_t = s] \tag{5.4}$$

Monte Carlo methods use an estimate of (5.3) as a target, whereas DP methods use an estimate of (5.4) as a target. The Monte Carlo target is an estimate because the expected value in (5.3) is not known; a sample return is used in place of the real expected return. The DP target is an estimate not because of the expected values, which are assumed to be completely provided by a model of the environment, but because $v_\pi(S_{t+1})$ is not known and the current estimate, $V(S_{t+1})$, is used instead. The TD target is an estimate for both reasons: it samples the expected values in (5.4) and it uses the current estimate V instead of the true v_π. Thus, TD methods combine the sampling Monte Carlo with the bootstrapping of DP.

We refer to TD and Monte Carlo updates as *sample updates* bacause 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 originale 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.

Finally, note that the quantity in brackets in $TD(0)$ 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 RL:

$$\delta_t = R_{t+1} + \gamma V(S_{t+1}) - V(S_t) \tag{5.5}$$

5.2 Advantages of TD Prediction Methods

TD methods update their estimates based in part on other estimates. They learn a guess from a guess., i.e., they **bootstrap**.

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 next 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 an espisode, because only then is the return known, whereas with TD methods one need wait only one time step. Certaily 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.

5.3 Optimality of TD(0)

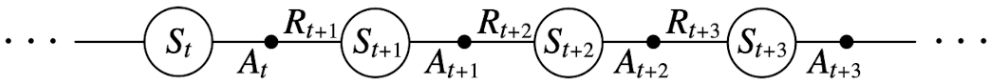
Suppose there is available only a finite amount of experience. In this case, a common approach with incremental learning methods is to present the experience repeatedly until the method converges upon an answer. Given an approximate value function, V, the increments specified by 5.1 or 5.2 are computed for every time step t at which a nonterminal state is visited, but the value function is changed only once, by the sum of all the increments. Then all the available experience is processed again with the new value function to produce a new overall increment, and so on, until the value function converges. We call this **batch updating** because updates are made only after processing each complete *batch* of training data.

Batch Monte Carlo methods always find the estimates that minimize the mean square error on the training set, whereas batch TD(0) always finds the estimates that would be exactly correct for the maximum-likelihood model of the Markov process. In general, the maximum-likelihood estimate of a parameter is the parameter value whose probability of generating the data is greatest.

5.4 Sarsa: On-policy TD Control

We turn now to the use of TD prediction methods for the control problem. 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 an on-policy TD control method.

The first step is to learn an action-value function rather than a state-value function. We must estimate $q_\pi(S,a)$ for the current behavior policy π for all states s and actions a. This can be done using essentially the same TD method described above for learning v_π.



In the previous section we considered transitions from state to state and learned values of states. Now we consider transitions from state-action pair to state-action pair, and learn the values of state-action pairs. Formally these cases are identical: they are both Markov chains with a reward process. The theorems assuring the convergence of state values under TD(0) also apply to the corresponding algorith, for action values:

$$Q(S_t,A_t) \leftarrow Q(S_t,A_t) + \alpha\left[R_{t+1} + \gamma Q(S_{t+1},A_{t+1}) - Q(S_t,A_t)\right]. \tag{5.6}$$

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. This quintuple gives rise to the name **Sarsa** for the algorithm.

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 tio q_π.

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

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

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

Figure 5.2: Pseudocode for Sarsa.

5.5 Q-learning: Off-policy TD Control

$$Q(S_t,A_t) \leftarrow Q(S_t,A_t) + \alpha\left[R_{t+1} + \gamma \max_a Q(S_{t+1},a) - Q(S_t,A_t)\right] \tag{5.7}$$

This is an off-policy TD control algorithm known as **Q-learning**. Here, the learnied action-value function, Q, direcly approximated q_*, the optimal action-value function, independent of the policy being followed. The policy still has an effect in that it determines which state-action pairs are visited and updated. However, all that is required for the correct convergence is that all pairs continue to be updated . This is a minimal requirement in the sense that any method guaranteed to find optimal behavior in the general case must require it. Q has been shown to converge with probability 1 to q_*. The Q-learning algorithm is shown below:

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

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

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

Figure 5.3: Pseudocode for Q-learning.

5.6 Expected Sarsa

Consider the learning algorithm that is just like Q-learning except that instead of the maximum over the next state-action paris it uses the expected value, taking into account how likely each action is under the current policy. That is, consider the algorith, with the update rule:

$$Q(S_t,A_t) \leftarrow Q(S_t,A_t) + \alpha\left[R_{t+1} + \gamma \mathbb{E}_\pi\left[Q(S_t,A_t)|S_{t+1}\right] - Q(S_t,A_t)\right] \tag{5.8}$$

$$= Q(S_t,A_t) + \alpha\left[R_{t+1} + \gamma \sum_a (a|S_{t+1}) Q(S_{t+1},a) - Q(S_t,A_t)\right] \tag{5.9}$$

but that otherwise follows the schema of Q-learning. Given the next state, S_{t+1}, this algorithm moves *deterministically* in the same direction as Sarsa moves in *expectation*, and accordingly it is called **Expected Sarsa**.
Expected Sarsa is more complex computationally than Sarsa but, in return, it eliminates the variance due to the random selection of A_{t+1}.

Figure 5.4: The backup diagrams for Q-learning (left) and Expected Sarsa (right).

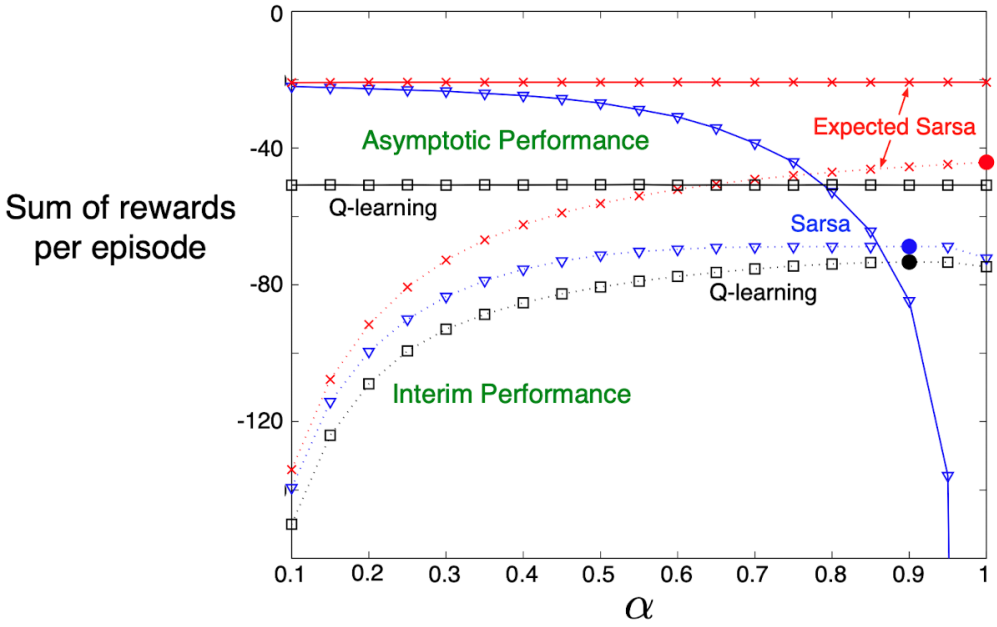


Figure 5.5: Interim and asymptotic performance of TD control methods on the cliff-walking task as a function of α. All algorithms used an ε-greedy policy with $\varepsilon = 0.1$. Asymptotic performance is an average over 100,000 episodes whereas interim performance is an average over the first 100 episodes.

5.7 Maximization Bias and Double Learning

All the control algorithms that we have discussed so far involve a maximization inthe construction of their target policies. In these algorithms, a maximum over estimated values is used implicitly as an estimate of the maximum value, which can lead to a significant positive bias. To see why, consider a single state s where there are many actions a whose true values, $q(s,a)$, are all zero but whose estimated values, $Q(s,a)$, are uncertain and thus distributed some above and some below zero. The maximum of the true values is zero, but the maximum of the estimates is positive, a positive bias. We call this *maximization bias*.

Are there algorithms that avoid this bias? Consider a bandit case in which we have noisy estimates of the value of each of many actions, obtained as simple averages of the rewards received on all the plays with each action. There will be a positive maximization bias if we use the maximum of the estimates as an estimate of the maximum of the true values.

Suppose we divided the plays in two sets and used them to learn two independent estimates, call them $Q_1(a)$ and $Q_2(a)$, each estimate of the true value $q(a)$, for all $a \in \mathcal{A}$. We could then use one estimate to determine the maximizing action $A^* = \arg\max_a Q_1(a)$ and the other, Q_2 in this case, to provide the estimate of its value, $Q_2(A^*) = Q_2(\arg\max_a Q_1(a))$. This estimate will then be unbiased in the sense that $\mathbb{E}[Q_2(A^*)] = q(A^*)$.

This is the idea of **double learning**. Note that although we learn two estimates, only one estimate is updated on each play; double learning doubles the memory requirements, but does not increase the amount of computation per step.

The idea of double learning extends naturally to algorithms for full MDPs. As an example, the double learning algorithm analogous to Q-learning, called Double Q-learning, divides the time

steps in two, perhaps by flipping a coin on each step. If the coin comes up heads, the update is

$$Q_1(S_t,A_t) \leftarrow Q_1(S_t,A_t) + \alpha \left[R_{t+1} + \gamma Q_2(S_{t+1}, \arg\max_a Q_1(S_{t+1},a)) - Q_1(S_t,A_t) \right]. \quad (5.10)$$

If the coin comes up tails, then the same update is done with $Q-1$ and Q_2 switched, so that Q_2 is updated.

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

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

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

Figure 5.6: The Double Learning algorithm.

6 n-step Bootstrapping

Neither MC methods nor one-step TD methods are always the best. In this chapter we present **n-step TD methods** that generalize both approaches, so that one can shift from one to the other smoothly as needed to meed the demands of a particular task. *n*-step methods span a spectrum with MC methods at one end and one-step TD methods at the other. The best methods are often intermediate between the two extremes.

As usual, we first consider the prediction problem and then the control problem. That is, we first consider how *n*-step methods can help in predicting returns as a function of state for a fixed policy (i.e., in estimating v_π). Then we extend the ideas to action values and control methods.

6.1 *n*-step TD Prediction

Consider estimating v_π from sample episodes generated using π. Monte Carlo methods perform an update for each state based on the entire sequence of observed rewards from that state until the end of the episode. The update of one-step TD methods, on the other hand, is based on just the one next reward, bootstrapping from the value of the state one step later as a proxy for the remaining rewards. One kind of intermediate method, then, would perform an update based on an intermediate number of rewards.

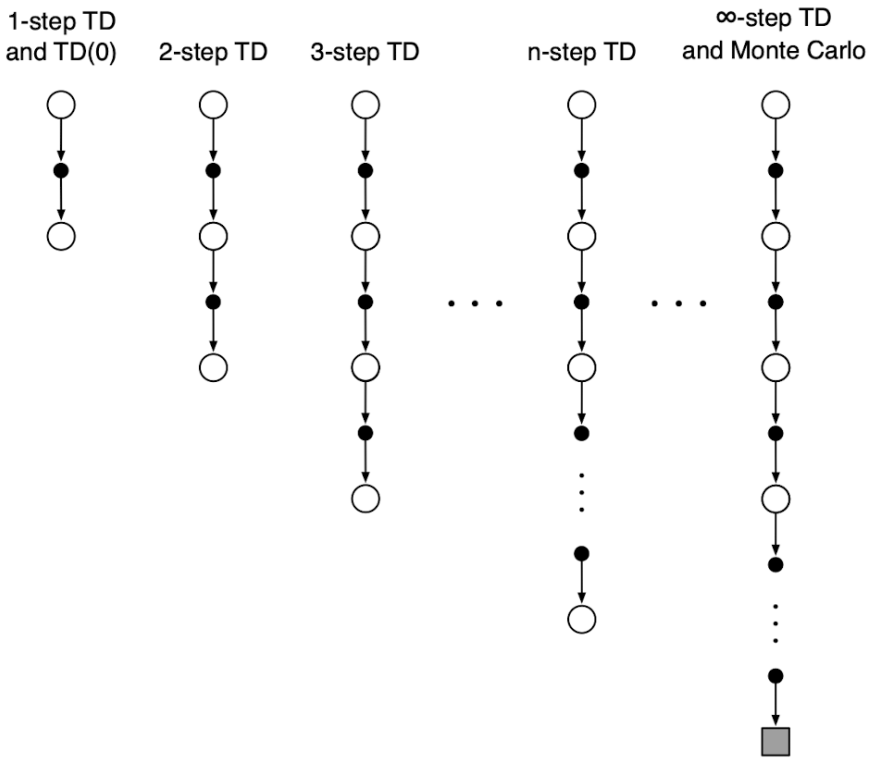


Figure 6.1: The backup diagram of *n*-step methods. These methods form a spectrum ranging from one-step TD methods to Monte Carlo methods.

Figure 6.1 shows the backup diagram of the spectrum of n-step updates for v_π, with the one-step TD update on the left and the up-until-termination Monte Carlo update on the right.
Methods in which the TD extends over n steos are called **n-step TD methods**. We know that in Monte Carlo updates the estimate of $v_\pi(S_t)$ is updated in the direction of the complete return:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots + \gamma^{T-t-1} R_T \tag{6.1}$$

where T is the last time step of the episode. Let us call this quantity the *target* of the update. Whereas in Monte Carlo updates the target is the return, in one-step updates the target is the first reward plus the discounted estimated value of the next state, which we call the *one-step return*:

$$G_t^{t:t+1} = R_{t+1} + \gamma v_t(S_{t+1}) \tag{6.2}$$

where $V_t : \mathcal{S} \to \mathbb{R}$ here is the estimate at time t of v_π. Our point now is that this idea makes just as much sense after two steps as it does after one. The target for a two-step update is the *two-step return*:

$$G_t^{t:t+2} = R_{t+1} + \gamma R_{t+2} + \gamma^2 v_t(S_{t+2}) \tag{6.3}$$

Similarly, the target for an arbitrary n-step update is the *n-step return*:

$$G_t^{t:t+n} = R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n v_t(S_{t+n}) \tag{6.4}$$

The natural state-value learning algorithm for using n-step returns is thus:

$$V_{t+n}(S_t) = V_{t+n-1}(S_t) + \alpha \left[G_{t:t+n} - V_{t+n-1}(S_t) \right], \quad 0 \le t < T, \tag{6.5}$$

while the values of all other states remain unchanged: $V_{t+n}(S) = V_{t+n-1}(s)$, for all $s \neq S_t$. We call this algorithm n-step TD. Note that no changes at all are made during the first $n-1$ steps of each episode. To make up for that, an equal number of additional updates are made at the end of the episode, after termination and before starting the next episode.

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

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

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

The n-step return uses the value funtion V_{t+n-1} to correct for the missing rewards beyond R_{t+n}. An important property of n-step returns is that their expectation is guaranteed to be a better estimate of v_π than V_{t+n-1} is, in a worst-state sense. That is, the worst error of the expected n-step return is guaranteed to be less than or equal to γ^n times the worst error under V_{t+n-1}:

$$\max_s |\mathbb{E}_\pi [G_{t:t+n}|S_t = s] - v_\pi(s)| \leq \gamma^n \max_s |V_{t+n-1}(s) - v_\pi(s)|, \tag{6.6}$$

for all $n \geq 1$. This is called the **error reduction property of n-step returns**.

6.2 n-step Sarsa

How can n-step methods be used not just for prediction, but for control? The main idea is to simply switch states for actions (state-action pairs) and then use an ε-greedy policy. This is the n-step version of Sarsa and is called **n-step Sarsa**. We redefine n-step returns (update targets) in terms of estimated action values:

$$G_{t:t+n} = R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n Q(t+n-1)(S_{t+n}, A_{t+n}), n \geq 1, 0 \leq t < T-n, \tag{6.7}$$

with $G_{t:t+n} = G_t$ if $t+n \geq T$. The natural algorithm is then:

$$Q_{t+n}(S_t, A_t) = Q_{t+n-1}(S_t, A_t) + \alpha [G_{t:t+n} - Q_{t+n-1}(S_t, A_t)], \quad 0 \leq t < T, \tag{6.8}$$

while the values of all other states remain unchanged: $Q_{t+n}(s,a) = Q_{t+n-1}(s,a)$, for all s, a such that $s \neq S_t$ or $a \neq A_t$.

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

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

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

Figure 6.2: The n-step Sarsa algorithm. The n-step return is used to update the action value function. The policy is ε-greedy with respect to the action value function.

6.3 *n*-step Off-policy Learning

In *n*-step methods, returns are constructed over *n* steps, so we are interested in the relative probability of just those *n* actions. To make a simple off-policy version of *n*-step TD, the update for time t (actually made at time $t+n$) can simply be weighted by $\rho_{t:t+n-1}$:

$$V_{t+n}(S_t) = V_{t+n-1}(S_t) + \alpha\rho_{t:t+n-1}\left[G_{t:t+n} - V_{t+n-1}(S_t)\right], \quad 0 \le t < T, \tag{6.9}$$

where $\rho_{t:t+n-1}$, called the *importance sampling ratio*, is the relative probability under the two policies of taking the *n* actions from A_t to A_{t+n-1}:

$$\rho_{t:h} = \prod_{k=t}^{\min(h,T-1)} \frac{\pi(A_k|S_k)}{b(A_k|S_k)} \tag{6.10}$$

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

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

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

Figure 6.3: The *n*-step off-policy learning algorithm. The importance sampling ratio $\rho_{t:t+n-1}$ is used to weight the update. The policy is ε-greedy with respect to the action value function.

7 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.

7.1 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 the 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 and their probabilities; these we call **distribution models**. Other models produce just one of the possibilities, sampled according to the probabilities; these we call **sample models**.

The kind of model assumed in dynamic programming—estimates of the MDP's dynamics, $p(s', r|s, a)$—is a distribution model. The kind of model used in the blackjack example in Chapter 5 is a sample model. Distribution models are stronger than sample models in that they can always be used to produce samples. However, in many applications it is much easier to obtain sample models than distribution 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:

$$\text{model} \rightarrow \text{planning} \rightarrow \text{policy}$$

State-space planning, which includes the approach we take in this book, is viewed primarily as a search through the state space for an optimal policy or an optimal path to a goal. Actions cause transitions from state to state, and value functions are computed over states. In what we call plan-space planning, planning is instead a search through the space of plans. Operators transform

one plan into another, and value functions, if any, are defined over the space of plans.

The unified view we present in this chapter is that all state-space planning methods share a common structure, a structure that is also present in the learning methods presented in this book. It takes the rest of the chapter to develop this view, but there are 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.

Dynamic programming methods clearly fit this structure: they make sweeps through the space of states, generating for each state the distribution of possible transitions. Each distribution is then used to compute a backed-up value (update target) and update the state's estimated value. 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.

Random-sample one-step tabular Q-planning

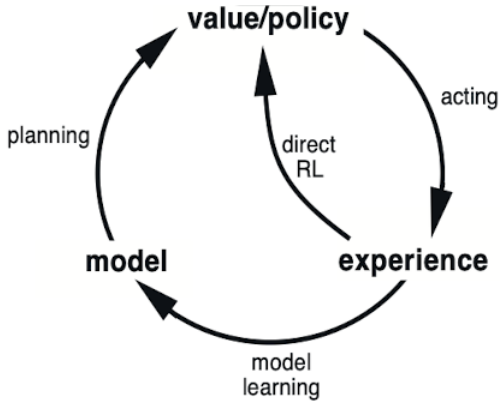
Loop forever:

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

7.2 Dyna: Integrated Planning, Acting, and Learning

When planning is done online, while interacting with the environment, a number of interesting issues arise. 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. To begin exploring these issues, in this section we present Dyna-Q, a simple architecture integrating the major functions needed in an online planning agent.

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



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. On the other hand, direct methods 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.

Dyna-Q includes all of the processes shown in the diagram above—planning, acting, model-learning, and direct RL—all occurring continually. The planning method is the random-sample one-step tabular Q-planning method. 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 $S_t, A_t \rightarrow R_{t+1}, S_{t+1}$, 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 experienced before, it simply returns the last-observed next state and next reward as its prediction.

During planning, the Q-planning algorithm randomly samples only from state–action pairs that have previously been experienced (in Step 1), so the model is never queried with a pair about which it has no information.

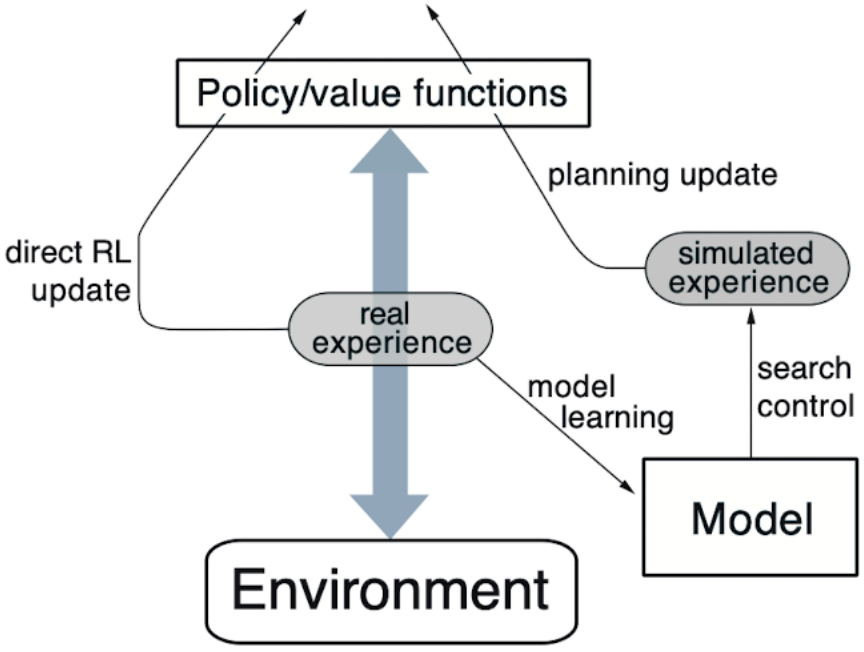


Figure 7.1: The general Dyna Architecture.

Conceptually, planning, acting, model-learning, and direct RL occur simultaneously and in parallel in Dyna agents. For concreteness and implementation on a serial computer, however, we fully specify the order in which they occur within a time step. In Dyna-Q, the acting, model-learning, and direct RL processes require little computation, and we assume they consume just a fraction of the time. The remaining time in each step can be devoted to the planning process, which is inherently computation-intensive.

Tabular Dyna-Q

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

7.3 When the Model Is Wrong

In the maze example presented in the previous section, the changes in the model were relatively modest. The model started out empty, and was then filled only with exactly correct information. In general, we cannot expect to be so fortunate. Models may be incorrect because the environment is stochastic and only a limited number of samples have been observed, or because the model was learned using function approximation that has generalized imperfectly, or simply because the environment has changed and its new behavior has not yet been observed. When the model is incorrect, the planning process is likely to compute a suboptimal policy.

In some cases, the suboptimal policy computed by planning quickly leads to the discovery and correction of the modeling error. This tends to happen when the model is optimistic in the sense of predicting greater reward or better state transitions than are actually possible. The planned policy attempts to exploit these opportunities and in doing so discovers that they do not exist.

Greater difficulties arise when the environment changes to become better than it was before, and yet the formerly correct policy does not reveal the improvement. In these cases the modeling error may not be detected for a long time, if ever.

The general problem here is another version of the conflict between exploration and exploitation. In a planning context, exploration means trying actions that improve the model, whereas exploitation means behaving in the optimal way given the current model.

We want the agent to explore to find changes in the environment, but not so much that performance is greatly degraded. As in the earlier exploration/exploitation conflict, there probably is no solution that is both perfect and practical, but simple heuristics are often effective.

7.4 Prioritized Sweeping

In the Dyna agents presented in the preceding sections, simulated transitions are started in state–action pairs selected uniformly at random from all previously experienced pairs. But a uniform selection is usually not the best; planning can be much more efficient if simulated transitions and updates are focused on particular state–action pairs.

This example suggests that search might be usefully focused by working backward from goal states. Of course, we do not really want to use any methods specific to the idea of "goal state." We want methods that work for general reward functions. Goal states are just a special case, convenient for stimulating intuition. In general, we want to work back not just from goal states but from any state whose value has changed. Suppose that the values are initially correct given the model. Suppose now that the agent discovers a change in the environment and changes its estimated value of one state, either up or down. Typically, this will imply that the values of many other states should also be changed, but the only useful one-step updates are those of actions that lead directly into the one state whose value has been changed.

This general idea might be termed **backward focusing of planning computations**.

It is natural to prioritize the updates according to a measure of their urgency, and perform them in order of priority. This is the idea behind **prioritized sweeping**. A queue is maintained of every state–action pair whose estimated value would change nontrivially if updated, prioritized by the size of the change. When the top pair in the queue is updated, the effect on each of its predecessor pairs is computed. If the effect is greater than some small threshold, then the pair is inserted in the queue with the new priority. In this way the effects of changes are efficiently propagated backward until quiescence.

Prioritized sweeping is just one way of distributing computations to improve planning efficiency, and probably not the best way. One of prioritized sweeping's limitations is that it uses expected updates, which in stochastic environments may waste lots of computation on low-probability transitions.

Prioritized sweeping for a deterministic environment

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

We have suggested in this chapter that all kinds of state-space planning can be viewed as sequences of value updates, varying only in the type of update, expected or sample, large or small, and in the order in which the updates are done.

7.5 Expected vs. Sample Updates

Much of this book has been about different kinds of value-function updates, and we have considered a great many varieties. Focusing for the moment on one-step updates, they vary primarily along three binary dimensions. The first two dimensions are whether they update state values or action values and whether they estimate the value for the optimal policy or for an arbitrary given policy. These

two dimensions give rise to four classes of updates for approximating the four value functions, q_*, v_*, q_π, and v_π. The other binary dimension is whether the updates are **expected updates**, considering all possible events that might happen, or **sample updates**, considering a single sample of what might happen.

When we introduced one-step sample updates in Chapter 6, we presented them as substitutes for expected updates. In the absence of a distribution model, expected updates are not possible, but sample updates can be done using sample transitions from the environment or a sample model. Implicit in that point of view is that expected updates, if possible, are preferable to sample updates. But are they? Expected updates certainly yield a better estimate because they are uncorrupted by sampling error, but they also require more computation, and computation is often the limiting resource in planning.

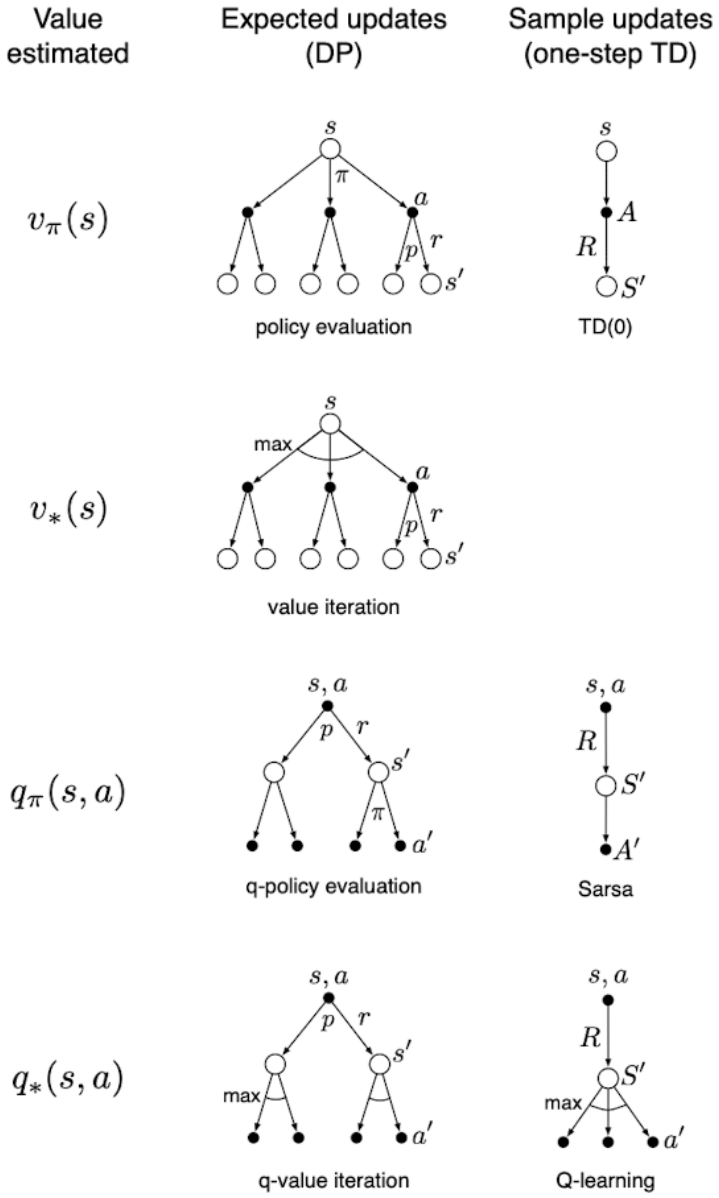


Figure 7.2: Backup diagrams for all the one-step updates considered in this chapter.

For concreteness, consider the expected and sample updates for approximating q_*, and the special case of discrete states and actions, a table-lookup representation of the approximate value function, Q,

and a model in the form of estimated dynamics, $\hat{p}(s',r|s,a)$. The expected update for a state–action pair, s,a, is:

$$Q(s,a) \leftarrow \sum_{s',r} \hat{p}(s',r|s,a) \left[r + \gamma \max_{a'} Q(s',a') \right] \tag{7.1}$$

The corresponding sample update for s,a, given a sample next state and reward, S' and R (from the model), is the Q-learning-like update:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[R + \gamma \max_{a'} Q(S',a') - Q(s,a) \right] \tag{7.2}$$

If there is enough time to complete an expected update, then the resulting estimate is generally better than that of b sample updates because of the absence of sampling error. But if there is insufficient time to complete an expected update, then sample updates are always preferable because they at least make some improvement in the value estimate with fewer than b updates.

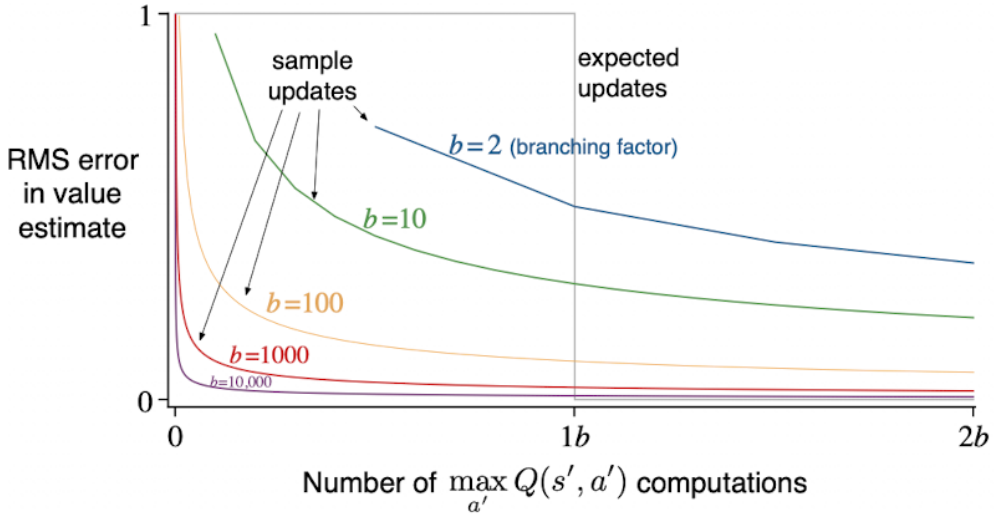


Figure 7.3: Comparison of efficiency of expected and sample updates.

7.6 Trajectory Sampling

In this section we compare two ways of distributing updates. The classical approach, from dynamic programming, is to perform sweeps through the entire state (or state–action) space, updating each state (or state–action pair) once per sweep. This is problematic on large tasks because there may not be time to complete even one sweep. In many tasks the vast majority of the states are irrelevant because they are visited only under very poor policies or with very low probability.

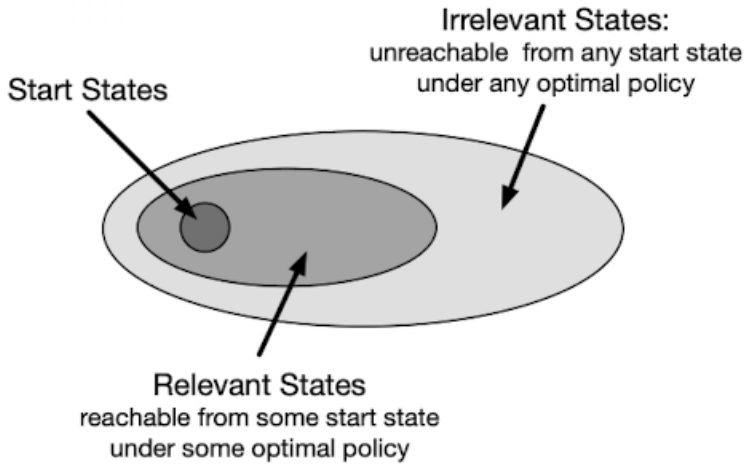
The second approach is to sample from the state or state–action space according to some distribution. One could sample uniformly, as in the Dyna-Q agent, but this would suffer from some of the same problems as exhaustive sweeps. More appealing is to distribute updates according to the **on-policy distribution**, that is, according to the distribution observed when following the current policy. One advantage of this distribution is that it is easily generated; one simply interacts with the model, following the current policy. We call this way of generating experience and updates **trajectory sampling**.

Focusing on the on-policy distribution could be beneficial because it causes vast, uninteresting parts of the space to be ignored, or it could be detrimental because it causes the same old parts of the space to be updated over and over.

7.7 Real-time Dynamic Programming

Real-time dynamic programming, or RTDP, is an on-policy trajectory-sampling version of the value-iteration algorithm of dynamic programming (DP). Because it is closely related to conventional sweep-based policy iteration, RTDP illustrates in a particularly clear way some of the advantages that on-policy trajectory sampling can provide. RTDP updates the values of states visited in actual or simulated trajectories by means of expected tabular value-iteration updates.

If trajectories can start only from a designated set of start states, and if you are interested in the prediction problem for a given policy, then on-policy trajectory sampling allows the algorithm to completely skip states that cannot be reached by the given policy from any of the start states: such states are irrelevant to the prediction problem. For a control problem, where the goal is to find an optimal policy instead of evaluating a given policy, there might well be states that cannot be reached by any optimal policy from any of the start states, and there is no need to specify optimal actions for these irrelevant states. What is needed is an **optimal partial policy**, meaning a policy that is optimal for the relevant states but can specify arbitrary actions, or even be undefined, for the irrelevant states.



The most interesting result for RTDP is that for certain types of problems satisfying reasonable conditions, RTDP is guaranteed to find a policy that is optimal on the relevant states without visiting every state infinitely often, or even without visiting some states at all. Indeed, in some problems, only a small fraction of the states need to be visited. This can be a great advantage for problems with very large state sets, where even a single sweep may not be feasible.

7.8 Planning at Decision Time

Planning can be used in at least two ways. The one we have considered so far in this chapter, typified by dynamic programming and Dyna, is to use planning to gradually improve a policy or value function on the basis of simulated experience obtained from a model. We call planning used in this way **background planning**.

The other way to use planning is to begin and complete it after encountering each new state S_t, as a computation whose output is the selection of a single action A_t; on the next step planning begins anew with S_{t+1} to produce A_{t+1}, and so on. Unlike the first use of planning, here planning focuses on a particular state. We call this **decision-time planning**.

7.9 Heuristic Search

The classical state-space planning methods in artificial intelligence are decision-time planning methods collectively known as **heuristic search**. In heuristic search, for each state encountered, a large tree of possible continuations is considered. The approximate value function is applied to the leaf nodes and then backed up toward the current state at the root. The backing up within the search tree is just the same as in the expected updates with maxes discussed throughout this book. The backing up stops at the state–action nodes for the current state. Once the backed-up values of these nodes are computed, the best of them is chosen as the current action, and then all backed-up values are discarded.

In conventional heuristic search no effort is made to save the backed-up values by changing the approximate value function. In fact, the value function is generally designed by people and never changed as a result of search.

We should not overlook the most obvious way in which heuristic search focuses updates: on the current state. Much of the effectiveness of heuristic search is due to its search tree being tightly focused on the states and actions that might immediately follow the current state.

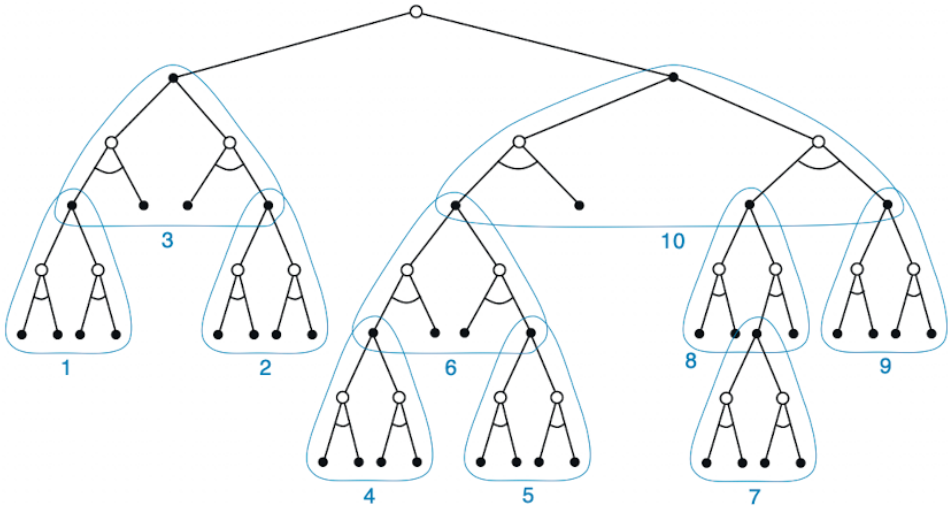


Figure 7.4: Heuristic search can be implemented as a sequence of one-step updates (blu lines) backing up values from the leaf nodes toward the root. The ordering shown here is for a selective depth-first search.

7.10 Rollout Algorithms

Rollout algorithms are decision-time planning algorithms based on Monte Carlo control applied to simulated trajectories that all begin at the current environment state. They estimate action values for a given policy by averaging the returns of many simulated trajectories that start with each possible action and then follow the given policy. When the action-value estimates are considered to be accurate enough, the action (or one of the actions) having the highest estimated value is executed, after which the process is carried out anew from the resulting next state.

What then do rollout algorithms accomplish? The policy improvement theorem tells us that given any two policies π and π' that are identical except that $\pi'(s) = a \neq \pi(s)$ for some state s, if $q_\pi(s,a) \geq v_\pi(s)$, then policy π' is as good as, or better, than π. This applies to rollout algorithms where s is the current state and π is the rollout policy.

In other words, the aim of a rollout algorithm is to improve upon the rollout policy; not to find an

optimal policy. Experience has shown that rollout algorithms can be surprisingly effective.

7.11 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) is a recent and strikingly successful example of decision-time planning. At its base, MCTS is a rollout algorithm as described above, but enhanced by the addition of a means for accumulating value estimates obtained from the Monte Carlo simulations in order to successively direct simulations toward more highly-rewarding trajectories. MCTS is largely responsible for the improvement in computer Go from a weak amateur level in 2005 to a grandmaster level (6 dan or more) in 2015.

MCTS is executed after encountering each new state to select the agent's action for that state; it is executed again to select the action for the next state, and so on. The core idea of MCTS is to successively focus multiple simulations starting at the current state by extending the initial portions of trajectories that have received high evaluations from earlier simulations.

For the most part, the actions in the simulated trajectories are generated using a simple policy, usually called a rollout policy as it is for simpler rollout algorithms. Monte Carlo value estimates are maintained only for the subset of state–action pairs that are most likely to be reached in a few steps, which form a tree rooted at the current state. MCTS incrementally extends the tree by adding nodes representing states that look promising based on the results of the simulated trajectories.

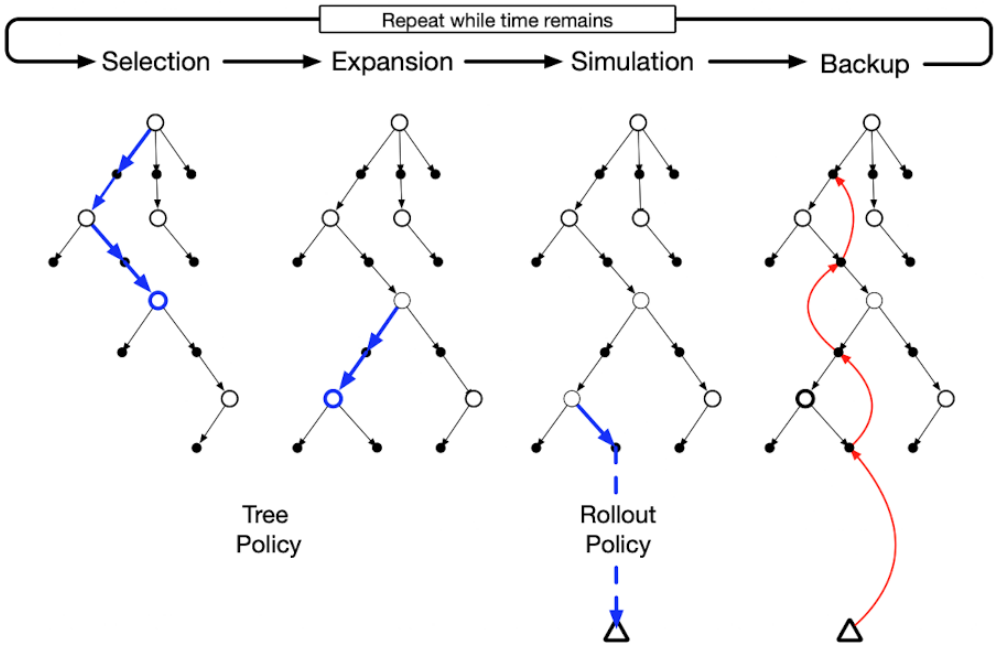


Figure 7.5: Monte Carlo Tree Search. When the environment changes to a new state, MCTS executes as many iterations as possible before an action needs to be selected, incrementally building a tree whose root node represents the current state. Each iteration consists of the four operations **Selection**, **Expansion**, **Simulation** and **Backup**.

Each iteration of a basica version of MCTS consists of the following four steps:

1. **Selection.** Starting at the root node, a tree policy base on the action values attached to the edges of the tree traverses the tree to select a leaf node.
2. **Expansion.** On some iterations, the tree is expanded from the selected leaf node by adding one or ore child nodes reached from the selected node via unexplored actions.

3. **Simulation.** From the selected node or from one of its newly-added child nodes, simulation of a complete episode is run with actions selected by the rolloud policy. The result is a MC trial with actions selected first by the tree policy and beyond the tree by the rollout policy.
4. **Backup.** The return generated by the simulated episode is backed up to update or to initialize, the action values attached to the edges of the tree traversed by the tree policy in this iteration of MCTS. No value are saved for the states and actions visited by the rollout policy beyond the tree.

8 Approximate Solution Methods

The novelty in this chapter is that the approximate value function is represented not as a table but as a parameterized functional form with weight vector $\mathbf{w} \in \mathbb{R}$. We will write $\hat{v}(s, w) \approx v_\pi(s)$ for the approximate value of state s given weight vector $\mathbf{w}$.

Typically, the number of weights is much less than the number of 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 byt also potentially more difficult to manage and understand. Extending RL to function approximation also makes it applicable to partially observable problems, in which the full state is not available to the agent.

What function approximation can't do, however, is augment the state representation with memories of past observations.

8.1 Value-function Approximation

All of the prediction methods covered in this book have been described as updates to an estimated value function that shifts its value at particular states toward a "backed-up value", or *update target*, for that state. Let us refer to an individual update by the notation $s \to u$, where s is the state updated and u is the update target that s's estimated value is shifted toward.

It is natural to interpret each update as specifying an example of the desired input-output behavior of the value function. In a sense, the update $s \to u$ 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. Now we permit arbitrarily complex and sophisticated methods to implement the update, and updating at s generalizer so that the estimated values of many other states are changed as well.

Function approximation methods expect to receive examples of the desired input-output behavior of the function they are trying to approximate. We use these methods for value prediction simply by passing to them the $s \to u$ of each update as a training example. We then interpret the approximate function they produce as an estimated value function. In RL, however, it is imoirtant that learning be able to occur online, while the agent interacts with its environment or with a model of its environment. To do this requires methods that are able to learn efficiently from incrementally acquired data. In addition, RL generally requires function approximation methods able to handle nonstationary target functions.

8.2 The Prediction Objective ($\overline{VE}$)

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. By assumption 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 a state s we mean the square of the difference between the approximate value $\hat{v}(s,w)$ and the true value $v_\pi(s)$. Weighting this over the state space by μ, we obtain a natural objective function, the mean square value error, denoted $\overline{VE}$:

$$\overline{VE}(w) \doteq \sum_{s \in S} \mu(s)[v_\pi(s) - \hat{v}(s,w)]^2$$

The square root of this measure, the root $\overline{VE}$, gives a rough measure of how much the approximate values differ from the true values and is often used in plots. Often $\mu(s)$ is chosen to be the fraction of time spent in s. Under on-policy training this is called the on-policy distribution; we focus entirely on this case in this chapter. In continuing tasks, the on-policy distribution is the stationary distribution under π.

8.3 Stochastic-gradient and Semi-gradient Methods

We now develop in detail one class of learning methods for function approximation in value prediction, those based on stochastic gradient descent (SGD). SGD methods are among the most widely used of all function approximation methods and are particularly well suited to online reinforcement learning.

In gradient-descent methods, the weight vector is a column vector with a fixed number of real valued components, $w \doteq (w_1, w_2, \ldots, w_d)^T$, and the approximate value function $\hat{v}(s,w)$ is a differentiable function of w for all $s \in S$. We will be updating w at each of a series of discrete time steps, $t = 0, 1, 2, 3, \ldots$, so we will need a notation w_t for the weight vector at each step. For now, let us assume that, on each step, we observe a new example $S_t \to v_\pi(S_t)$ consisting of a (possibly randomly selected) state S_t and its true value under the policy. These states might be successive states from an interaction with the environment, but for now we do not assume so. 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 w that gets all the 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 $\overline{VE}$ as given by (9.1). 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:

$$w_{t+1} \doteq w_t - \frac{1}{2}\alpha \nabla [v_\pi(S_t) - \hat{v}(S_t, w_t)]^2$$

$$= w_t + \alpha [v_\pi(S_t) - \hat{v}(S_t, w_t)] \nabla \hat{v}(S_t, w_t)$$

where α is a positive step-size parameter, and $\nabla f(w)$, for any scalar expression $f(w)$ that is a function of a vector (here w), denotes the column vector of partial derivatives of the expression with respect to the components of the vector:

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

This derivative vector is the gradient of f with respect to w. SGD methods are "gradient descent" methods because the overall step in w_t is proportional to the negative gradient of the example's squared error (9.4). This is the direction in which the error falls most rapidly. Gradient descent methods are called "stochastic" when the update is done, as here, on only a single example, which might have been selected stochastically. Over many examples, making small steps, the overall effect is to minimize an average performance measure such as the $\overline{VE}$.

We turn now to the case in which the target output, here denoted $U_t \in \mathbb{R}$, of the tth training example, $S_t \to U_t$, 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 (9.5) 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:

$$w_{t+1} \doteq w_t + \alpha[U_t - \hat{v}(S_t, w_t)]\nabla\hat{v}(S_t, w_t)$$

If U_t is an unbiased estimate, that is, if $\mathbb{E}[U_t|S_t = s] = v_\pi(s)$, for each t, then w_t is guaranteed to converge to a local optimum under the usual stochastic approximation conditions (2.7) for decreasing α.

One does not obtain the same guarantees if a bootstrapping estimate of $v_\pi(S_t)$ is used as the target U_t in (9.7). Bootstrapping targets such as n-step returns $G_{t:t+n}$ or the DP target $\sum_{a,s',r}\pi(a|S_t)p(s',r|S_t,a)[r+\hat{v}(s',w_t)]$ all depend on the current value of the weight vector w_t, which implies that they will be biased and that they will not produce a true gradient-descent method. One way to look at this is that the key step from (9.4) to (9.5) relies on the target being independent of w_t. This step would not be valid if a bootstrapping estimate were used in place of $v_\pi(S_t)$. Bootstrapping methods are not in fact instances of true gradient descent (Barnard, 1993). They take into account the effect of changing the weight vector w_t on the estimate, but ignore its effect on the target. They include only a part of the gradient and, accordingly, we call them semi-gradient methods.

Although semi-gradient (bootstrapping) methods do not converge as robustly as gradient methods, they do converge reliably in important cases such as the linear case discussed in the next section. Moreover, they offer important advantages that make them often clearly preferred. One reason for this is that they typically enable significantly faster learning, as we have seen in Chapters 6 and 7. Another is that they enable learning to be continual and online, without waiting for the end of an episode. This enables them to be used on continuing problems and provides computational advantages. A prototypical semi-gradient method is semi-gradient TD(0), which uses $U_t \doteq R_{t+1} + \hat{v}(S_{t+1}, w)$ as its target.

8.4 Linear Methods

One of the most important special cases of function approximation is that in which the approximate function, $\hat{v}(\cdot, w)$, is a linear function of the weight vector, w. Corresponding to every state s, there is a real-valued vector $x(s) \doteq (x_1(s), x_2(s), \ldots, x_d(s))^T$, with the same number of components as w. Linear methods approximate the state-value function by the inner product between w and $x(s)$:

$$\hat{v}(s, w) \doteq w^T x(s) \doteq \sum_{i=1}^{d} w_i x_i(s)$$

In this case the approximate value function is said to be linear in the weights, or simply linear.

The vector $x(s)$ is called a feature vector representing state s. Each component $x_i(s)$ of $x(s)$ is the value of a function $x_i : S \to \mathbb{R}$. We think of a feature as the entirety of one of these functions, and we call its value for a state s a feature of s. For linear methods, features are basis functions because they form a linear basis for the set of approximate functions. Constructing d-dimensional feature vectors to represent states is the same as selecting a set of d basis functions. Features may be defined in many different ways; we cover a few possibilities in the next sections.

It is natural to use SGD updates with linear function approximation. The gradient of the approximate value function with respect to w in this case is

$$\nabla \hat{v}(s,w) = x(s)$$

Thus, in the linear case the general SGD update (9.7) reduces to a particularly simple form:

$$w_{t+1} \doteq w_t + \alpha[U_t - \hat{v}(S_t, w_t)]x(S_t)$$

Because it is so simple, the linear SGD case is one of the most favorable for mathematical analysis. Almost all useful convergence results for learning systems of all kinds are for linear (or simpler) function approximation methods.

In particular, in the linear case there is only one optimum (or, in degenerate cases, one set of equally good optima), and thus any method that is guaranteed to converge to or near a local optimum is automatically guaranteed to converge to or near the global optimum. For example, the gradient Monte Carlo algorithm presented in the previous section converges to the global optimum of the $\overline{VE}$ under linear function approximation if α is reduced over time according to the usual conditions. The semi-gradient TD(0) algorithm presented in the previous section also converges under linear function approximation, but this does not follow from general results on SGD; a separate theorem is necessary. The weight vector converged to is also not the global optimum, but rather a point near the local optimum. It is useful to consider this important case in more detail, specifically for the continuing case. The update at each time t is

$$w_{t+1} \doteq w_t + \alpha(R_{t+1} + w_t^T x_{t+1} - w_t^T x_t)x_t$$

$$= w_t + \alpha(R_{t+1}x_t - x_t(x_t - x_{t+1})^T w_t)$$

where here we have used the notational shorthand $x_t = x(S_t)$. Once the system has reached steady state, for any given w_t, the expected next weight vector can be written

$$\mathbb{E}[w_{t+1}|w_t] = w_t + \alpha(b - Aw_t)$$

where

$$b \doteq \mathbb{E}[R_{t+1}x_t] \in \mathbb{R}^d \quad \text{and} \quad A \doteq \mathbb{E}[x_t(x_t - \gamma x_{t+1})^T] \in \mathbb{R}^{d\times d}$$

From (9.10) it is clear that, if the system converges, it must converge to the weight vector w_{TD} at which

$$b - Aw_{TD} = 0$$
$$\Rightarrow b = Aw_{TD}$$
$$\Rightarrow w_{TD} \doteq A^{-1}b$$

This quantity is called the TD fixed point. In fact linear semi-gradient TD(0) converges to this point. Some of the theory proving its convergence, and the existence of the inverse above, is given in the box.

At the TD fixed point, it has also been proven (in the continuing case) that the $\overline{VE}$ is within a bounded expansion of the lowest possible error:

$$\overline{VE}(w_{TD}) \leq \frac{1}{1-\gamma} \min_w \overline{VE}(w)$$

That is, the asymptotic error of the TD method is no more than $\frac{1}{1-\gamma}$ times the smallest possible error, that attained in the limit by the Monte Carlo method. Because γ is often near one, this expansion factor can be quite large, so there is substantial potential loss in asymptotic performance with the TD method. On the other hand, recall that the TD methods are often of vastly reduced variance compared to Monte Carlo methods, and thus faster, as we saw in Chapters 6 and 7. Which method will be best depends on the nature of the approximation and problem, and on how long learning continues.

8.5 Feature Construction for Linear Methods

Linear methods are interesting because of their convergence guarantees, but also because in practice they can be very efficient in terms of both data and computation. Whether or not this is so depends critically on how the states are represented in terms of features, which we investigate in this large section. Choosing features appropriate to the task is an important way of adding prior domain knowledge to reinforcement learning systems.

Intuitively, the features should correspond to the aspects of the state space along which generalization may be appropriate. If we are valuing geometric objects, for example, we might want to have features for each possible shape, color, size, or function. If we are valuing states of a mobile robot, then we might want to have features for locations, degrees of remaining battery power, recent sonar readings, and so on.

A limitation of the linear form is that it cannot take into account any interactions between features, such as the presence of feature i being good only in the absence of feature j. For example, in the pole-balancing task (Example 3.4), high angular velocity can be either good or bad depending on the angle. If the angle is high, then high angular velocity means an imminent danger of falling—a bad state—whereas if the angle is low, then high angular velocity means the pole is righting itself—a good state. A linear value function could not represent this if its features coded separately for the angle and the angular velocity. It needs instead, or in addition, features for combinations of these two underlying state dimensions. In the following subsections we consider a variety of general ways of doing this.

- **Polynomials.** You can represent a state with n dimensions, so that $x(s) = (s_1, s_2, \ldots, s_n)^T$, maybe with interactions or appending a constant value (1) at the beginning to represent affine functions in the original state numbers. Thus, each order-n polynomial-basis feature x_i can be written as

$$x_i(s) = \prod_{j=1}^{n} s_j^{c_{i,j}} \tag{8.1}$$

  where $c_{i,j}$ is an integer in the set $\{0, 1, \ldots, n\}$ for an integer $n \geq 0$.

- **Fourier Basis.** The Fourier basis represents functions as a sum of sinusoids. For a state s, we can define Fourier features as

$$x_i(s) = \cos(\omega_i^T s + \phi_i) \tag{8.2}$$

where ω_i is a frequency vector and ϕ_i is a phase shift. This allows for capturing periodic patterns in the state space.

- **Radial Basis Functions (RBFs).** RBFs are a popular choice for function approximation, defined as

$$x_i(s) = \exp\left(-\frac{1}{2\sigma^2}\|s - \mu_i\|^2\right) \tag{8.3}$$

where μ_i is the center of the RBF and σ controls the width. RBFs are particularly useful for capturing localized features in the state space.

- **ANN.** Artificial Neural Networks (ANNs) can be used to learn complex feature representations from raw state inputs. They consist of multiple layers of interconnected nodes, where each node applies a non-linear transformation to its inputs. The learned features can capture intricate patterns and interactions in the state space.

9 Policy Gradient Methods

So far in this book almost all the methods have been *action-value methods*; they learned the values of actions and then selected actions based on their estimated action values. 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 it is not required for action selection. We will use the notation $\theta \in \mathbb{R}^{d'}$ for the policy's parameter vector. In this chapter we consider methods for learning the policy parameter based on the gradient of some scalar performance mearure $J(\theta)$ with respect to the policy parameter. These methods seek to *maximize* the performance, so their updates approximate gradient *ascent* in J:

$$\theta_{t+1} = \theta_t + \alpha \nabla \hat{J}(\theta_t),$$

All methods that follow this general schema are called *policy gradient methods*.

9.1 Policy Approximation and its Advantages

In policy gradient methods, the policy can be parameterized in any way, as long as $\pi(a|s,\theta)$ is differentiable with respect to its parameters, i.e., as long as $\nabla\pi(a|s,\theta)$ exists and is finite for all $s \in \mathcal{S}, a \in \mathcal{A}(s), \theta \in \mathbb{R}^{d'}$.

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

$$\pi(a|s,\theta) = \frac{e^{h(s,a,\theta)}}{\sum_{a'} e^{h(s,a',\theta)}}$$

The action preferences themselves can be parameterized arbitrarily. As an example, they might be computed by a deep artificial neural network (ANN), where θ is the vector of all the connection weights of the network. Or the preferences could be simply linear in features:

$$h(s,a,\theta) = \theta^\top \phi(s,a)$$

9.2 Policy Gradient Theorem

We can simplify the notation without losing any meaninful generality by assuming that every episode starts in sime particular (non-random) state s_0. Then, in the episodic case we define performance as:

$$J(\theta) = v_{\pi_\theta}(s_0)$$

where v_{π_θ} is the true value function for π_θ, the policy determined by θ. From here on in our discussion we will assume no discounting ($\gamma = 1$) for the episodic case.

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 is which those selections are made, and that both of these are affected by the polict parameter. Given s state, the effect of the policy parameter on the actions, and thus on the 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?
Here comes the *policy gradient theorem*, which provides an analytical expression for the gradient of performance with respect to the policy parameter. The policy gradient theorem for the episodic case estabilishes that

$$\nabla J(\theta) \propto \sum_s \mu(s) \sum_a q_\pi(s,a) \nabla \pi(a|s,\theta),$$

The distribution μ here (seen in Chapters 8 and 9) is the on-policy distribution under π.

9.3 REINFORCE: Monte Carlo Policy Gradient

Recall out stratgy for stochastic gradient ascent, which requires a way to obtain samples such that the expectation of the sample gradient is proportional to the actual gradient of the performance measure as a function of the parameter. The sample gradient need only be proportional to the gradient because any constant of proportionality can be absorbed into the step size α, which is otherwise arbitrary.
Notice that the right-hand side of the policy gradient theorem is a sum over states weighted by how often the staes occyr under the target policy π; if π is followed, then states will be encountered in these proportions. Thus:

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

We could stop here and instantiate our stochastic gradient-ascent algorithm as

$$\theta_{t+1} = \theta_t + \alpha \sum_a \hat{q}(S_t,a,w) \nabla \pi(a|S_t,\theta)$$

but our current interest is the classical REINFORCE algorithm whose update at time t involves just A_t, the one action actually taken at time t.
We continue our derivation of REINFORCE by introducing A_t in the same way as we introduced S_t, i.e., by replacing a sum over the random variable's possible values by an expectation under π, and then sampling the expectation.

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

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

$$\theta_{t+1} = \theta_t + \alpha G_t \frac{\nabla \pi(A_t|S_t,\theta)}{\pi(A_t|S_t,\theta)}$$

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 favour actions that yield the highest return. The latter makes sense because otherwise actions that are selected frequently are at an advantage and might win out even if they do not yield the highest return.

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 $t = 0, 1, \ldots, T-1$:
    $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$ $\qquad (G_t)$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \gamma^t G \nabla \ln \pi(A_t|S_t,\boldsymbol{\theta})$

Figure 9.1: Monte-Carlo Policy-Gradient Control (episodic) for π_*

As a stochastic gradient method, REINFORCE has good theoretical convergence properties. By construction, the expected update over an episode is in the same direction as the performance gradient. This assures an improvement in expected performance for sufficiently small α, and convergence to a local optimum under standard stochastic approximation conditions for decreasing α. However, as Monte Carlo method REINFORCE may be of high variance and thus produce slow learning.

Bibliography

[1] Andrew G Barto. “Reinforcement learning: An introduction. by richard’s sutton”. In: *SIAM Rev* 6.2 (2021), p. 423.