Lucian Buşoniu, Robert Babuška, Bart De Schutter, and Damien Ernst

Reinforcement learning and dynamic programming using function approximators

Preface

Control systems are making a tremendous impact on our society. Though invisible to most users, they are essential for the operation of nearly all devices – from basic home appliances to aircraft and nuclear power plants. Apart from technical systems, the principles of control are routinely applied and exploited in a variety of disciplines such as economics, medicine, social sciences, and artificial intelligence.

A common denominator in the diverse applications of control is the need to influence or modify the behavior of dynamic systems to attain prespecified goals. One approach to achieve this is to assign a numerical performance index to each state trajectory of the system. The control problem is then solved by searching for a control policy that drives the system along trajectories corresponding to the best value of the performance index. This approach essentially reduces the problem of finding good control policies to the search for solutions of a mathematical optimization problem.

Early work in the field of optimal control dates back to the 1940s with the pioneering research of Pontryagin and Bellman. Dynamic programming (DP), introduced by Bellman, is still among the state-of-the-art tools commonly used to solve optimal control problems when a system model is available. The alternative idea of finding a solution *in the absence* of a model was explored as early as the 1960s. In the 1980s, a revival of interest in this model-free paradigm led to the development of the field of reinforcement learning (RL). The central theme in RL research is the design of algorithms that learn control policies solely from the knowledge of transition samples or trajectories, which are collected beforehand or by online interaction with the system. Most approaches developed to tackle the RL problem are closely related to DP algorithms.

A core obstacle in DP and RL is that solutions cannot be represented exactly for problems with large discrete state-action spaces or continuous spaces. Instead, compact representations relying on function approximators must be used. This challenge was already recognized while the first DP techniques were being developed. However, it has only been in recent years – and largely in correlation with the advance of RL – that approximation-based methods have grown in diversity, maturity, and efficiency, enabling RL and DP to scale up to realistic problems.

This book provides an accessible in-depth treatment of reinforcement learning and dynamic programming methods using function approximators. We start with a concise introduction to classical DP and RL, in order to build the foundation for the remainder of the book. Next, we present an extensive review of state-of-the-art approaches to DP and RL with approximation. Theoretical guarantees are provided on the solutions obtained, and numerical examples and comparisons are used to illustrate the properties of the individual methods. The remaining three chapters are

dedicated to a detailed presentation of representative algorithms from the three major classes of techniques: value iteration, policy iteration, and policy search. The properties and the performance of these algorithms are highlighted in simulation and experimental studies on a range of control applications.

We believe that this balanced combination of practical algorithms, theoretical analysis, and comprehensive examples makes our book suitable not only for researchers, teachers, and graduate students in the fields of optimal and adaptive control, machine learning and artificial intelligence, but also for practitioners seeking novel strategies for solving challenging real-life control problems.

This book can be read in several ways. Readers unfamiliar with the field are advised to start with Chapter 1 for a gentle introduction, and continue with Chapter 2 (which discusses classical DP and RL) and Chapter 3 (which considers approximation-based methods). Those who are familiar with the basic concepts of RL and DP may consult the list of notations given at the end of the book, and then start directly with Chapter 3. This first part of the book is sufficient to get an overview of the field. Thereafter, readers can pick any combination of Chapters 4 to 6, depending on their interests: approximate value iteration (Chapter 4), approximate policy iteration and online learning (Chapter 5), or approximate policy search (Chapter 6).

Supplementary information relevant to this book, including a complete archive of the computer code used in the experimental studies, is available at the Web site:

http://www.dcsc.tudelft.nl/rlbook/

Comments, suggestions, or questions concerning the book or the Web site are welcome. Interested readers are encouraged to get in touch with the authors using the contact information on the Web site.

The authors have been inspired over the years by many scientists who undoubtedly left their mark on this book; in particular by Louis Wehenkel, Pierre Geurts, Guy-Bart Stan, Rémi Munos, Martin Riedmiller, and Michail Lagoudakis. Pierre Geurts also provided the computer program for building ensembles of regression trees, used in several examples in the book. This work would not have been possible without our colleagues, students, and the excellent professional environments at the Delft Center for Systems and Control of the Delft University of Technology, the Netherlands, the Montefiore Institute of the University of Liège, Belgium, and at Supélec Rennes, France. Among our colleagues in Delft, Justin Rice deserves special mention for carefully proofreading the manuscript. To all these people we extend our sincere thanks.

We thank Sam Ge for giving us the opportunity to publish our book with Taylor & Francis CRC Press, and the editorial and production team at Taylor & Francis for their valuable help. We gratefully acknowledge the financial support of the BSIK-ICIS project “Interactive Collaborative Information Systems” (grant no. BSIK03024) and the Dutch funding organizations NWO and STW. Damien Ernst is a Research Associate of the FRS-FNRS, the financial support of which he acknowledges. We appreciate the kind permission offered by the IEEE to reproduce material from our previous works over which they hold copyright.

Finally, we thank our families for their continual understanding, patience, and support.

Lucian Buşoniu
Robert Babuška
Bart De Schutter
Damien Ernst
November 2009

About the authors

Lucian Buşoniu is a postdoctoral fellow at the Delft Center for Systems and Control of Delft University of Technology, in the Netherlands. He received his PhD degree (*cum laude*) in 2009 from the Delft University of Technology, and his MSc degree in 2003 from the Technical University of Cluj-Napoca, Romania. His current research interests include reinforcement learning and dynamic programming with function approximation, intelligent and learning techniques for control problems, and multi-agent learning.

Robert Babuška is a full professor at the Delft Center for Systems and Control of Delft University of Technology in the Netherlands. He received his PhD degree (*cum laude*) in Control in 1997 from the Delft University of Technology, and his MSc degree (with honors) in Electrical Engineering in 1990 from Czech Technical University, Prague. His research interests include fuzzy systems modeling and identification, data-driven construction and adaptation of neuro-fuzzy systems, model-based fuzzy control and learning control. He is active in applying these techniques in robotics, mechatronics, and aerospace.

Bart De Schutter is a full professor at the Delft Center for Systems and Control and at the Marine & Transport Technology department of Delft University of Technology in the Netherlands. He received the PhD degree in Applied Sciences (*summa cum laude* with congratulations of the examination jury) in 1996 from K.U. Leuven, Belgium. His current research interests include multi-agent systems, hybrid systems control, discrete-event systems, and control of intelligent transportation systems.

Damien Ernst received the MSc and PhD degrees from the University of Liège in 1998 and 2003, respectively. He is currently a Research Associate of the Belgian FRS-FNRS and he is affiliated with the Systems and Modeling Research Unit of the University of Liège. Damien Ernst spent the period 2003–2006 with the University of Liège as a Postdoctoral Researcher of the FRS-FNRS and held during this period positions as visiting researcher at CMU, MIT and ETH. He spent the academic year 2006–2007 working at Supélec (France) as professor. His main research interests are in the fields of power system dynamics, optimal control, reinforcement learning, and design of dynamic treatment regimes.

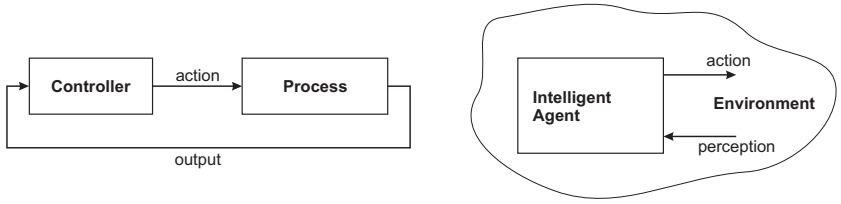
Contents

1

Introduction

Dynamic programming (DP) and reinforcement learning (RL) are algorithmic methods for solving problems in which actions (decisions) are applied to a system over an extended period of time, in order to achieve a desired goal. DP methods require a model of the system's behavior, whereas RL methods do not. The time variable is usually discrete and actions are taken at every discrete time step, leading to a sequential decision-making problem. The actions are taken in closed loop, which means that the outcome of earlier actions is monitored and taken into account when choosing new actions. Rewards are provided that evaluate the one-step decision-making performance, and the goal is to optimize the long-term performance, measured by the total reward accumulated over the course of interaction.

Such decision-making problems appear in a wide variety of fields, including automatic control, artificial intelligence, operations research, economics, and medicine. For instance, in automatic control, as shown in Figure 1.1(a), a controller receives output measurements from a process, and applies actions to this process in order to make its behavior satisfy certain requirements (Levine, 1996). In this context, DP and RL methods can be applied to solve optimal control problems, in which the behavior of the process is evaluated using a cost function that plays a similar role to the rewards. The decision maker is the controller, and the system is the controlled process.



(a) Automatic control. (b) Artificial intelligent agents.

FIGURE 1.1
Two application domains for dynamic programming and reinforcement learning.

In artificial intelligence, DP and RL are useful to obtain optimal behavior for intelligent agents, which, as shown in Figure 1.1(b), monitor their environment through perceptions and influence it by applying actions (Russell and Norvig, 2003). The decision maker is now the agent, and the system is the agent's environment.

If a model of the system is available, DP methods can be applied. A key benefit

of DP methods is that they make few assumptions on the system, which can generally be nonlinear and stochastic (Bertsekas, 2005a, 2007). This is in contrast to, e.g., classical techniques from automatic control, many of which require restrictive assumptions on the system, such as linearity or determinism. Moreover, many DP methods do not require an analytical expression of the model, but are able to work with a simulation model instead. Constructing a simulation model is often easier than deriving an analytical model, especially when the system behavior is stochastic.

However, sometimes a model of the system cannot be obtained at all, e.g., because the system is not fully known beforehand, is insufficiently understood, or obtaining a model is too costly. RL methods are helpful in this case, since they work using only data obtained from the system, without requiring a model of its behavior (Sutton and Barto, 1998). Offline RL methods are applicable if data can be obtained in advance. Online RL algorithms learn a solution by interacting with the system, and can therefore be applied even when data is not available in advance. For instance, intelligent agents are often placed in environments that are not fully known beforehand, which makes it impossible to obtain data in advance. Note that RL methods can, of course, also be applied when a model is available, simply by using the model instead of the real system to generate data.

In this book, we primarily adopt a control-theoretic point of view, and hence employ control-theoretical notation and terminology, and choose control systems as examples to illustrate the behavior of DP and RL algorithms. We nevertheless also exploit results from other fields, in particular the strong body of RL research from the field of artificial intelligence. Moreover, the methodology we describe is applicable to sequential decision problems in many other fields.

The remainder of this introductory chapter is organized as follows. In Section 1.1, an outline of the DP/RL problem and its solution is given. Section 1.2 then introduces the challenge of approximating the solution, which is a central topic of this book. Finally, in Section 1.3, the organization of the book is explained.

1.1 The dynamic programming and reinforcement learning problem

The main elements of the DP and RL problem, together with their flow of interaction, are represented in Figure 1.2: a controller interacts with a process by means of states and actions, and receives rewards according to a reward function. For the DP and RL algorithms considered in this book, an important requirement is the availability of a signal that completely describes the current state of the process (this requirement will be formalized in Chapter 2). This is why the process shown in Figure 1.2 outputs a state signal.

To clarify the meaning of the elements of Figure 1.2, we use a conceptual robotic navigation example. Autonomous mobile robotics is an application domain where automatic control and artificial intelligence meet in a natural way, since a mobile

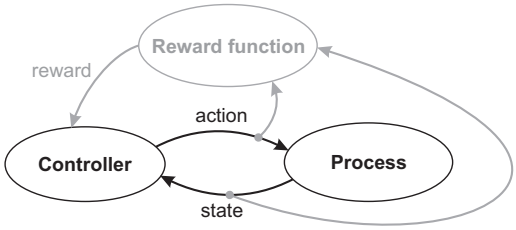


FIGURE 1.2
The elements of DP and RL and their flow of interaction. The elements related to the reward are depicted in gray.

robot and its environment comprise a process that must be controlled, while the robot is also an artificial agent that must accomplish a task in its environment. Figure 1.3 presents the navigation example, in which the robot shown in the bottom region must navigate to the goal on the top-right, while avoiding the obstacle represented by a gray block. (For instance, in the field of rescue robotics, the goal might represent the location of a victim to be rescued.) The controller is the robot's software, and the process consists of the robot's environment (the surface on which it moves, the obstacle, and the goal) together with the body of the robot itself. It should be emphasized that in DP and RL, the physical body of the decision-making entity (if it has one), its sensors and actuators, as well as any fixed lower-level controllers, are all considered to be a part of the process, whereas the controller is taken to be only the decision-making algorithm.

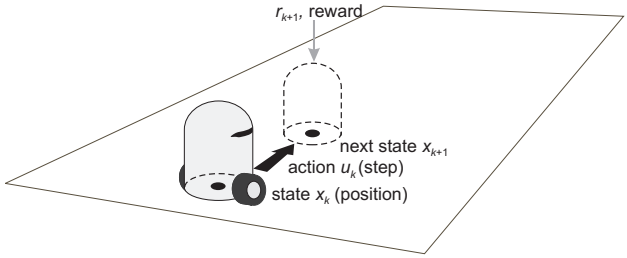


FIGURE 1.3
A robotic navigation example. An example transition is also shown, in which the current and next states are indicated by black dots, the action by a black arrow, and the reward by a gray arrow. The dotted silhouette represents the robot in the next state.

In the navigation example, the state is the position of the robot on the surface, given, e.g., in Cartesian coordinates, and the action is a step taken by the robot, similarly given in Cartesian coordinates. As a result of taking a step from the current

position, the next position is obtained, according to a transition function. In this example, because both the positions and steps are represented in Cartesian coordinates, the transitions are most often additive: the next position is the sum of the current position and the step taken. More complicated transitions are obtained if the robot collides with the obstacle. Note that for simplicity, most of the dynamics of the robot, such as the motion of the wheels, have not been taken into account here. For instance, if the wheels can slip on the surface, the transitions become stochastic, in which case the next state is a random variable.

The quality of every transition is measured by a reward, generated according to the reward function. For instance, the reward could have a positive value such as 10 if the robot reaches the goal, a negative value such as -1, representing a penalty, if the robot collides with the obstacle, and a neutral value of 0 for any other transition. Alternatively, more informative rewards could be constructed, using, e.g., the distances to the goal and to the obstacle.

The behavior of the controller is dictated by its policy: a mapping from states into actions, which indicates what action (step) should be taken in each state (position).

In general, the state is denoted by x, the action by u, and the reward by r. These quantities may be subscripted by discrete time indices, where k denotes the current time index (see Figure 1.3). The transition function is denoted by f, the reward function by ρ, and the policy by h.

In DP and RL, the goal is to maximize the return, consisting of the cumulative reward over the course of interaction. We mainly consider discounted infinite-horizon returns, which accumulate rewards obtained along (possibly) infinitely long trajectories starting at the initial time step $k = 0$, and weigh the rewards by a factor that decreases exponentially as the time step increases:

$$\gamma^0 r_1 + \gamma^1 r_2 + \gamma^2 r_3 + \ldots \tag{1.1}$$

The discount factor $\gamma \in [0,1)$ gives rise to the exponential weighting, and can be seen as a measure of how "far-sighted" the controller is in considering its rewards. Figure 1.4 illustrates the computation of the discounted return for the navigation problem of Figure 1.3.

The rewards depend of course on the state-action trajectory followed, which in turn depends on the policy being used:

$$x_0,\, u_0 = h(x_0),\, x_1,\, u_1 = h(x_1),\, x_2,\, u_2 = h(x_2),\, \ldots$$

In particular, each reward r_{k+1} is the result of the transition (x_k, u_k, x_{k+1}). It is convenient to consider the return separately for every initial state x_0, which means the return is a function of the initial state. Note that, if state transitions are stochastic, the goal considered in this book is to maximize the expectation of (1.1) over all the realizations of the stochastic trajectory starting from x_0.

The core challenge of DP and RL is therefore to arrive at a solution that optimizes the long-term performance given by the return, using only reward information that describes the immediate performance. Solving the DP/RL problem boils down to finding an optimal policy, denoted by h^*, that maximizes the return (1.1) for every

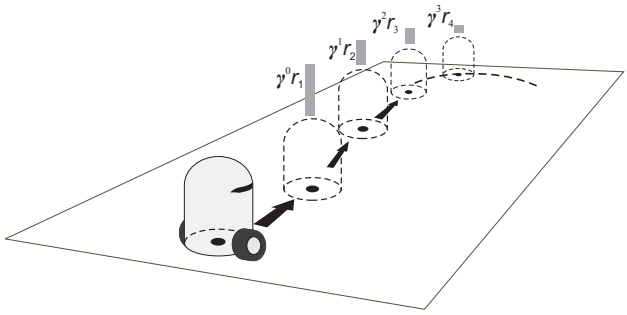


FIGURE 1.4
The discounted return along a trajectory of the robot. The decreasing heights of the gray vertical bars indicate the exponentially diminishing nature of the discounting applied to the rewards.

initial state. One way to obtain an optimal policy is to first compute the maximal returns. For example, the so-called optimal Q-function, denoted by Q^*, contains for each state-action pair (x,u) the return obtained by first taking action u in state x and then choosing optimal actions from the second step onwards:

$$Q^*(x,u) = \gamma^0 r_1 + \gamma^1 r_2 + \gamma^2 r_3 + \ldots$$
$$\text{when } x_0 = x, u_0 = u, \text{ and optimal actions are taken for } x_1, x_2, \ldots \quad (1.2)$$

If transitions are stochastic, the optimal Q-function is defined instead as the expectation of the return on the right-hand side of (1.2) over the trajectory realizations. The optimal Q-function can be found using a suitable DP or RL algorithm. Then, an optimal policy can be obtained by choosing, at each state x, an action $h^*(x)$ that maximizes the optimal Q-function for that state:

$$h^*(x) \in \arg\max_u Q^*(x,u) \quad (1.3)$$

To see that an optimal policy is obtained, recall that the optimal Q-function already contains optimal returns starting from the second step onwards; in (1.3), an action is chosen that additionally maximizes the return over the first step, therefore obtaining a return that is maximal over the entire horizon, i.e., optimal.

1.2 Approximation in dynamic programming and reinforcement learning

Consider the problem of representing a Q-function, not necessarily the optimal one. Since no prior knowledge about the Q-function is available, the only way to guarantee an exact representation is to store distinct values of the Q-function (Q-values)

for every state-action pair. This is schematically depicted in Figure 1.5 for the navigation example of Section 1.1: Q-values must be stored separately for each position of the robot, and for each possible step that it might take from every such position. However, because the position and step variables are continuous, they can both take uncountably many distinct values. Therefore, even in this simple example, storing distinct Q-values for every state-action pair is obviously impossible. The only feasible way to proceed is to use a compact representation of the Q-function.

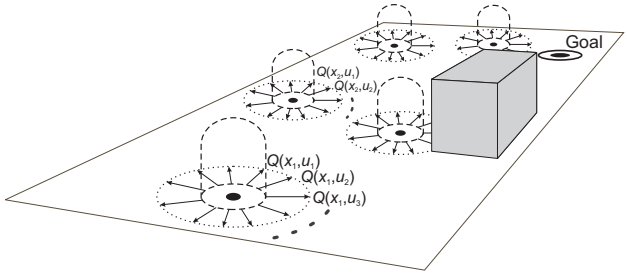


FIGURE 1.5
Illustration of an exact Q-function representation for the navigation example. For every state-action pair, there is a corresponding Q-value. The Q-values are not represented explicitly, but only shown symbolically near corresponding state-action pairs.

One type of compact Q-function representation that will often be used in the sequel relies on state-dependent basis functions (BFs) and action discretization. Such a representation is illustrated in Figure 1.6 for the navigation problem. A finite number of BFs, $\phi_1,\ldots,\phi_N$, are defined over the state space, and the action space is discretized into a finite number of actions, in this case 4: left, right, forward, and back. Instead of storing distinct Q-values for every state-action pair, such a representa-

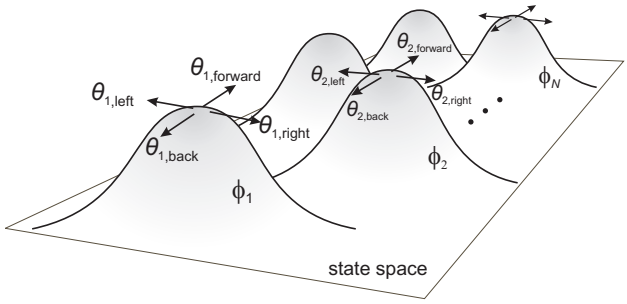


FIGURE 1.6
Illustration of a compact Q-function representation for the navigation example.

tion stores *parameters* θ, one for each combination of a BF and a discrete action. To find the Q-value of a continuous state-action pair (x,u), the action is discretized (e.g., using nearest-neighbor discretization). Assume the result of discretization is the discrete action "forward"; then, the Q-value is computed by adding the parameters $\theta_{1,\text{forward}},\ldots,\theta_{N,\text{forward}}$ corresponding to this discrete action, where the parameters are weighted by the value of their corresponding BFs at x:

$$\widehat{Q}(x,\text{forward}) = \sum_{i=1}^{N} \phi_i(x)\theta_{i,\text{forward}} \tag{1.4}$$

The DP/RL algorithm therefore only needs to remember the N parameters, which can easily be done when N is not too large. Note that this type of Q-function representation generalizes to any DP/RL problem. Even in problems with a finite number of discrete states and actions, compact representations can still be useful by reducing the number of values that must be stored.

While not all DP and RL algorithms employ Q-functions, they all generally require compact representations, so the illustration above extends to the general case. Consider, e.g., the problem of representing a policy h. An exact representation would generally require storing distinct actions for every possible state, which is impossible when the state variables are continuous. Note that continuous actions are not problematic for policy representation.

It should be emphasized at this point that, in general, a compact representation can only represent the target function up to a certain approximation error, which must be accounted for. Hence, in the sequel such representations are called "function approximators," or "approximators" for short.

Approximation in DP and RL is not only a problem of representation. Assume for instance that an approximation of the optimal Q-function is available. To obtain an approximately optimal policy, (1.3) must be applied, which requires maximizing the Q-function over the action variable. In large or continuous action spaces, this is a potentially difficult optimization problem, which can only be solved approximately in general. However, when a discrete-action Q-function of the form (1.4) is employed, it is sufficient to compute the Q-values of all the discrete actions and to find the maximum among these values using enumeration. This provides a motivation for using discretized actions. Besides approximate maximization, other approximation difficulties also arise, such as the estimation of expected values from samples. These additional challenges are outside the scope of this section, and will be discussed in detail in Chapter 3.

The classical DP and RL algorithms are only guaranteed to obtain an optimal solution if they use exact representations. Therefore, the following important questions must be kept in mind when using function approximators:

- If the algorithm is iterative, does it *converge* when approximation is employed? Or, if the algorithm is not iterative, does it obtain a meaningful solution?
- If a meaningful solution is obtained, is it *near optimal*, and more specifically, how far is it from the optimal solution?

- Is the algorithm *consistent*, i.e., does it asymptotically obtain the optimal solution as the approximation power grows?

These questions will be taken into account when discussing algorithms for approximate DP and RL.

Choosing an appropriate function approximator for a given problem is a highly nontrivial task. The complexity of the approximator must be managed, since it directly influences the memory and computational costs of the DP and RL algorithm. This is an important concern in both approximate DP and approximate RL. Equally important in approximate RL are the restrictions imposed by the limited amount of data available, since in general a more complex approximator requires more data to compute an accurate solution. If prior knowledge about the function of interest is available, it can be used in advance to design a lower-complexity, but still accurate, approximator. For instance, BFs with intuitive, relevant meanings could be defined (such as, in the navigation problem, BFs representing the distance between the robot and the goal or the obstacle). However, prior knowledge is often unavailable, especially in the model-free context of RL. In this book, we will therefore pay special attention to techniques that automatically find low-complexity approximators suited to the problem at hand, rather than relying on manual design.

1.3 About this book

This book focuses on approximate dynamic programming (DP) and reinforcement learning (RL) for control problems with continuous variables. The material is aimed at researchers, practitioners, and graduate students in the fields of systems and control (in particular optimal, adaptive, and learning control), computer science (in particular machine learning and artificial intelligence), operations research, and statistics. Although not primarily intended as a textbook, our book can nevertheless be used as support for courses that treat DP and RL methods.

Figure 1.7 presents a road map for the remaining chapters of this book, which we will detail next. Chapters 2 and 3 are prerequisite for the remainder of the book and should be read in sequence. In particular, in Chapter 2 the DP and RL problem and its solution are formalized, representative classical algorithms are introduced, and the behavior of several such algorithms is illustrated in an example with discrete states and actions. Chapter 3 gives an extensive account of DP and RL methods with function approximation, which are applicable to large and continuous-space problems. A comprehensive selection of algorithms is introduced, theoretical guarantees are provided on the approximate solutions obtained, and numerical examples involving the control of a continuous-variable system illustrate the behavior of several representative algorithms.

The material of Chapters 2 and 3 is organized along three major classes of DP and RL algorithms: value iteration, policy iteration, and policy search. In order to strengthen the understanding of these three classes of algorithms, each of the three

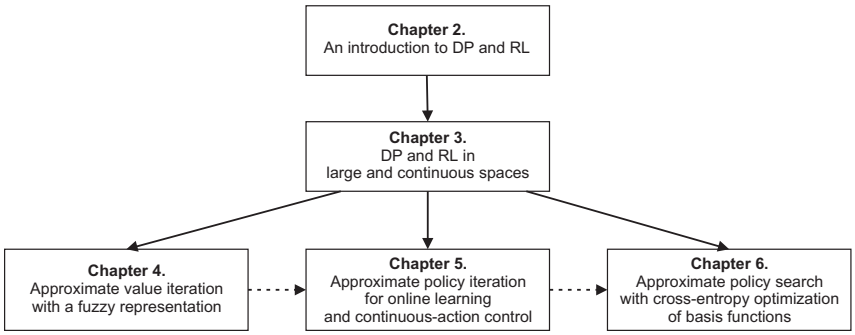


FIGURE 1.7
A road map for the remainder of this book, given in a graphical form. The full arrows indicate the recommended sequence of reading, whereas dashed arrows indicate optional ordering.

final chapters of the book considers in detail an algorithm from one of these classes. Specifically, in Chapter 4, a value iteration algorithm with fuzzy approximation is discussed, and an extensive theoretical analysis of this algorithm illustrates how convergence and consistency guarantees can be developed for approximate DP. In Chapter 5, an algorithm for approximate policy iteration is discussed. In particular, an online variant of this algorithm is developed, and some important issues that appear in online RL are emphasized along the way. In Chapter 6, a policy search approach relying on the cross-entropy method for optimization is described, which highlights one possibility to develop techniques that scale to relatively high-dimensional state spaces, by focusing the computation on important initial states. The final part of each of these three chapters contains an experimental evaluation on a representative selection of control problems.

Chapters 4, 5, and 6 can be read in any order, although, if possible, they should be read in sequence.

Two appendices are included at the end of the book (these are not shown in Figure 1.7). Appendix A outlines the so-called ensemble of extremely randomized trees, which is used as an approximator in Chapters 3 and 4. Appendix B describes the cross-entropy method for optimization, employed in Chapters 4 and 6. Reading Appendix B before these two chapters is not mandatory, since both chapters include a brief, specialized introduction to the cross-entropy method, so that they can more easily be read independently.

Additional information and material concerning this book, including the computer code used in the experimental studies, is available at the Web site:

http://www.dcsc.tudelft.nl/rlbook/

2

An introduction to dynamic programming and reinforcement learning

This chapter introduces dynamic programming and reinforcement learning techniques, and the formal model behind the problem they solve: the Markov decision process. Deterministic and stochastic Markov decision processes are discussed in turn, and their optimal solution is characterized. Three categories of dynamic programming and reinforcement learning algorithms are described: value iteration, policy iteration, and policy search.

2.1 Introduction

In dynamic programming (DP) and reinforcement learning (RL), a controller (agent, decision maker) interacts with a process (environment), by means of three signals: a state signal, which describes the state of the process, an action signal, which allows the controller to influence the process, and a scalar reward signal, which provides the controller with feedback on its immediate performance. At each discrete time step, the controller receives the state measurement and applies an action, which causes the process to transition into a new state. A reward is generated that evaluates the quality of this transition. The controller receives the new state measurement, and the whole cycle repeats. This flow of interaction is represented in Figure 2.1 (repeated from Figure 1.2).

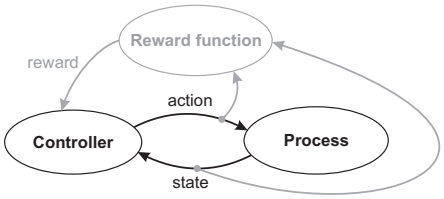


FIGURE 2.1 The flow of interaction in DP and RL.

The behavior of the controller is dictated by its policy, a function from states into actions. The behavior of the process is described by its dynamics, which determine

how the state changes as a result of the controller's actions. State transitions can be deterministic or stochastic. In the deterministic case, taking a given action in a given state always results in the same next state, while in the stochastic case, the next state is a random variable. The rule according to which rewards are generated is described by the reward function. The process dynamics and the reward function, together with the set of possible states and the set of possible actions (respectively called state space and action space), constitute a so-called Markov decision process (MDP).

In the DP/RL setting, the goal is to find an optimal policy that maximizes the (expected) return, consisting of the (expected) cumulative reward over the course of interaction. In this book, we will mainly consider infinite-horizon returns, which accumulate rewards along infinitely long trajectories. This choice is made because infinite-horizon returns have useful theoretical properties. In particular, they lead to stationary optimal policies, which means that for a given state, the optimal action choices will always be the same, regardless of the time when that state is encountered.

The DP/RL framework can be used to address problems from a variety of fields, including, e.g., automatic control, artificial intelligence, operations research, and economics. Automatic control and artificial intelligence are arguably the most important fields of origin for DP and RL. In automatic control, DP can be used to solve nonlinear and stochastic optimal control problems (Bertsekas, 2007), while RL can alternatively be seen as adaptive optimal control (Sutton et al., 1992; Vrabie et al., 2009). In artificial intelligence, RL helps to build an artificial agent that learns how to survive and optimize its behavior in an unknown environment, without requiring prior knowledge (Sutton and Barto, 1998). Because of this mixed inheritance, two sets of equivalent names and notations are used in DP and RL, e.g., "controller" has the same meaning as "agent," and "process" has the same meaning as "environment." In this book, we will use the former, control-theoretical terminology and notation.

A taxonomy of DP and RL algorithms is shown in Figure 2.2 and detailed in the remainder of this section.

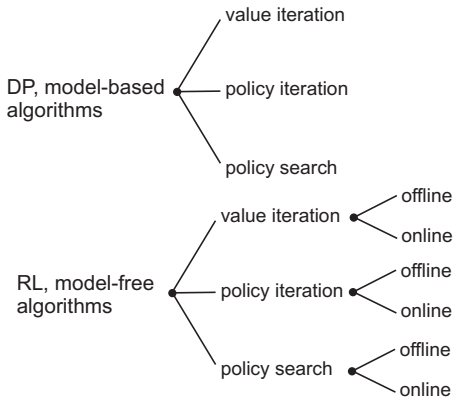


FIGURE 2.2 A taxonomy of DP and RL algorithms.

DP algorithms require a model of the MDP, including the transition dynamics and the reward function, to find an optimal policy (Bertsekas, 2007; Powell, 2007). The model DP algorithms work offline, producing a policy which is then used to control the process.[1] Usually, they do not require an analytical expression of the dynamics. Instead, given a state and an action, the model is only required to generate a next state and the corresponding reward. Constructing such a generative model is often easier than deriving an analytical expression of the dynamics, especially when the dynamics are stochastic.

RL algorithms are model-free (Bertsekas and Tsitsiklis, 1996; Sutton and Barto, 1998), which makes them useful when a model is difficult or costly to construct. RL algorithms use data obtained from the process, in the form of a set of samples, a set of process trajectories, or a single trajectory. So, RL can be seen as model-free, sample-based or trajectory-based DP, and DP can be seen as model-based RL. While DP algorithms can use the model to obtain any number of sample transitions from any state-action pair, RL algorithms must work with the limited data that can be obtained from the process – a greater challenge. Note that some RL algorithms build a model from the data; we call these algorithms "model-learning."

Both the DP and RL classes of algorithms can be broken down into three subclasses, according to the path taken to find an optimal policy. These three subclasses are value iteration, policy iteration, and policy search, and are characterized as follows.

- *Value iteration* algorithms search for the optimal value function, which consists of the maximal returns from every state or from every state-action pair. The optimal value function is used to compute an optimal policy.

- *Policy iteration* algorithms evaluate policies by constructing their value functions (instead of the optimal value function), and use these value functions to find new, improved policies.

- *Policy search* algorithms use optimization techniques to directly search for an optimal policy.

Note that, in this book, we use the name DP to refer to the class of all model-based algorithms that find solutions for MDPs, including model-based policy search. This class is larger than the category of algorithms traditionally called DP, which only includes model-based value iteration and policy iteration (Bertsekas and Tsitsiklis, 1996; Sutton and Barto, 1998; Bertsekas, 2007).

Within each of the three subclasses of RL algorithms, two categories can be further distinguished, namely offline and online algorithms. Offline RL algorithms use data collected in advance, whereas online RL algorithms learn a solution by interacting with the process. Online RL algorithms are typically not provided with any data

[1] There is also a class of model-based, DP-like *online* algorithms called model-predictive control (Maciejowski, 2002; Camacho and Bordons, 2004). In order to restrict the scope of the book, we do not discuss model-predictive control. For details about the relationship of DP/RL with model-predictive control, see, e.g., (Bertsekas, 2005b; Ernst et al., 2009).

in advance, but instead have to rely only on the data they collect while learning, and thus are useful when it is difficult or costly to obtain data in advance. Most online RL algorithms work incrementally. For instance, an incremental, online value iteration algorithm updates its estimate of the optimal value function after each collected sample. Even before this estimate becomes accurate, it is used to derive estimates of an optimal policy, which are then used to collect new data.

Online RL algorithms must balance the need to collect informative data (by *exploring* novel action choices or novel parts of the state space) with the need to control the process well (by *exploiting* the currently available knowledge). This exploration-exploitation trade-off makes online RL more challenging than offline RL. Note that, although online RL algorithms are only guaranteed (under appropriate conditions) to converge to an optimal policy when the process does not change over time, in practice they are sometimes applied also to slowly changing processes, in which case they are expected to adapt the solution so that the changes are taken into account.

The remainder of this chapter is structured as follows. Section 2.2 describes MDPs and characterizes the optimal solution for an MDP, in the deterministic as well as in the stochastic setting. The class of value iteration algorithms is introduced in Section 2.3, policy iteration in Section 2.4, and policy search in Section 2.5. When introducing value iteration and policy iteration, DP and RL algorithms are described in turn, while the introduction of policy search focuses on the model-based, DP setting. Section 2.6 concludes the chapter with a summary and discussion. Throughout the chapter, a simulation example involving a highly abstracted robotic task is employed to illustrate certain theoretical points, as well as the properties of several representative algorithms.

2.2 Markov decision processes

DP and RL problems can be formalized with the help of MDPs (Puterman, 1994). We first present the simpler case of MDPs with deterministic state transitions. Afterwards, we extend the theory to the stochastic case.

2.2.1 Deterministic setting

A deterministic MDP is defined by the state space X of the process, the action space U of the controller, the transition function f of the process (which describes how the state changes as a result of control actions), and the reward function ρ (which evaluates the immediate control performance).[2] As a result of the action u_k applied in the state x_k at the discrete time step k, the state changes to x_{k+1}, according to the

[2] As mentioned earlier, control-theoretic notation is used instead of artificial intelligence notation. For instance, in the artificial intelligence literature on DP and RL, the state space is usually denoted by S, the state by s, the action space by A, the action by a, and the policy by π.

transition function $f : X \times U \to X$:

$$x_{k+1} = f(x_k, u_k)$$

At the same time, the controller receives the scalar reward signal r_{k+1}, according to the *reward function* $\rho : X \times U \to \mathbb{R}$:

$$r_{k+1} = \rho(x_k, u_k)$$

where we assume that $\|\rho\|_\infty = \sup_{x,u} |\rho(x,u)|$ is finite.[3] The reward evaluates the immediate effect of action u_k, namely the transition from x_k to x_{k+1}, but in general does not say anything about its long-term effects.

The controller chooses actions according to its *policy* $h : X \to U$, using:

$$u_k = h(x_k)$$

Given f and ρ, the current state x_k and the current action u_k are sufficient to determine both the next state x_{k+1} and the reward r_{k+1}. This is the Markov property, which is essential in providing theoretical guarantees about DP/RL algorithms.

Some MDPs have terminal states that, once reached, can no longer be left; all the rewards received in terminal states are 0. The RL literature often uses "trials" or "episodes" to refer to trajectories starting from some initial state and ending in a terminal state.

Example 2.1 The deterministic cleaning-robot MDP. Consider the deterministic problem depicted in Figure 2.3: a cleaning robot has to collect a used can and also has to recharge its batteries.

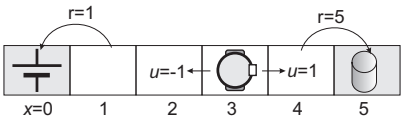


FIGURE 2.3 The cleaning-robot problem.

In this problem, the state x describes the position of the robot, and the action u describes the direction of its motion. The state space is discrete and contains six distinct states, denoted by integers 0 to 5: $X = \{0,1,2,3,4,5\}$. The robot can move to the left ($u = -1$) or to the right ($u = 1$); the discrete action space is therefore $U = \{-1,1\}$. States 0 and 5 are terminal, meaning that once the robot reaches either of them it can no longer leave, regardless of the action. The corresponding transition function is:

$$f(x,u) = \begin{cases} x+u & \text{if } 1 \le x \le 4 \\ x & \text{if } x = 0 \text{ or } x = 5 \text{ (regardless of } u\text{)} \end{cases}$$

[3] To simplify the notation, whenever searching for extrema, performing summations, etc., over variables whose domains are obvious from the context, we omit these domains from the formulas. For instance, in the formula $\sup_{x,u} |\rho(x,u)|$, the domains of x and u are clearly X and U, so they are omitted.

In state 5, the robot finds a can and the transition into this state is rewarded with 5. In state 0, the robot can recharge its batteries and the transition into this state is rewarded with 1. All other rewards are 0. In particular, taking any action while in a terminal state results in a reward of 0, which means that the robot will not accumulate (undeserved) rewards in the terminal states. The corresponding reward function is:

$$\rho(x,u) = \begin{cases} 5 & \text{if } x = 4 \text{ and } u = 1 \\ 1 & \text{if } x = 1 \text{ and } u = -1 \\ 0 & \text{otherwise} \end{cases}$$

□

Optimality in the deterministic setting

In DP and RL, the goal is to find an *optimal policy* that maximizes the return from any initial state x_0. The return is a cumulative aggregation of rewards along a trajectory starting at x_0. It concisely represents the reward obtained by the controller in the long run. Several types of return exist, depending on the way in which the rewards are accumulated (Bertsekas and Tsitsiklis, 1996, Section 2.1; Kaelbling et al., 1996). The *infinite-horizon discounted return* is given by:

$$R^h(x_0) = \sum_{k=0}^{\infty} \gamma^k r_{k+1} = \sum_{k=0}^{\infty} \gamma^k \rho(x_k, h(x_k)) \tag{2.1}$$

where $\gamma \in [0,1)$ is the *discount factor* and $x_{k+1} = f(x_k, h(x_k))$ for $k \geq 0$. The discount factor can be interpreted intuitively as a measure of how "far-sighted" the controller is in considering its rewards, or as a way of taking into account increasing uncertainty about future rewards. From a mathematical point of view, discounting ensures that the return will always be bounded if the rewards are bounded. The goal is therefore to maximize the long-term performance (return), while only using feedback about the immediate, one-step performance (reward). This leads to the so-called challenge of delayed rewards (Sutton and Barto, 1998): actions taken in the present affect the potential to achieve good rewards far in the future, but the immediate reward provides no information about these long-term effects.

Other types of return can also be defined. The undiscounted return, obtained by setting γ equal to 1 in (2.1), simply adds up the rewards, without discounting. Unfortunately, the infinite-horizon undiscounted return is often unbounded. An alternative is to use the infinite-horizon average return:

$$\lim_{K \to \infty} \frac{1}{K} \sum_{k=0}^{K} \rho(x_k, h(x_k))$$

which is bounded in many cases. Finite-horizon returns can be obtained by accumulating rewards along trajectories of a fixed, finite length K (the horizon), instead of along infinitely long trajectories. For instance, the finite-horizon discounted return can be defined as:

$$\sum_{k=0}^{K} \gamma^k \rho(x_k, h(x_k))$$

The undiscounted return ($\gamma = 1$) can be used more easily in the finite-horizon case, as it is bounded when the rewards are bounded.

In this book, we will mainly use the infinite-horizon discounted return (2.1), because it has useful theoretical properties. In particular, for this type of return, under certain technical assumptions, there always exists at least one *stationary*, deterministic optimal policy $h^* : X \to U$ (Bertsekas and Shreve, 1978, Chapter 9). In contrast, in the finite-horizon case, optimal policies depend in general on the time step k, i.e., they are nonstationary (Bertsekas, 2005a, Chapter 1).

While the discount factor γ can theoretically be regarded as a given part of the problem, in practice, a good value of γ has to be chosen. Choosing γ often involves a trade-off between the quality of the solution and the convergence rate of the DP/RL algorithm, for the following reasons. Some important DP/RL algorithms converge faster when γ is smaller (this is the case, e.g., for model-based value iteration, which will be introduced in Section 2.3). However, if γ is too small, the solution may be unsatisfactory because it does not sufficiently take into account rewards obtained after a large number of steps.

There is no generally valid procedure for choosing γ. Consider however, as an example, a typical stabilization problem from automatic control, where from every initial state the process should reach a steady state and remain there. In such a problem, γ should be chosen large enough that the rewards received upon reaching the steady state and remaining there have a detectable influence on the returns from every initial state. For instance, if the number of steps taken by a reasonable policy to stabilize the system from an initial state x is $K(x)$, then γ should be chosen so that $\gamma^{K_{\max}}$ is not too small, where $K_{\max} = \max_x K(x)$. However, finding $K_{\max}$ is a difficult problem in itself, which could be solved using, e.g., domain knowledge, or a suboptimal policy obtained by other means.

Value functions and the Bellman equations in the deterministic setting

A convenient way to characterize policies is by using their value functions. Two types of value functions exist: state-action value functions (Q-functions) and state value functions (V-functions). Note that the name "value function" is often used for V-functions in the literature. We will use the names "Q-function" and "V-function" to clearly differentiate between the two types of value functions, and the name "value function" to refer to Q-functions and V-functions collectively. We will first define and characterize Q-functions, and then turn our attention to V-functions.

The Q-function $Q^h : X \times U \to \mathbb{R}$ of a policy h gives the return obtained when starting from a given state, applying a given action, and following h thereafter:

$$Q^h(x,u) = \rho(x,u) + \gamma R^h(f(x,u)) \tag{2.2}$$

Here, $R^h(f(x,u))$ is the return from the next state $f(x,u)$. This concise formula can be obtained by first writing $Q^h(x,u)$ explicitly as the discounted sum of rewards obtained by taking u in x and then following h:

$$Q^h(x,u) = \sum_{k=0}^{\infty} \gamma^k \rho(x_k, u_k)$$

where $(x_0,u_0)=(x,u)$, $x_{k+1}=f(x_k,u_k)$ for $k\geq 0$, and $u_k=h(x_k)$ for $k\geq 1$. Then, the first term is separated from the sum:

$$\begin{aligned} Q^h(x,u) &= \rho(x,u)+\sum_{k=1}^{\infty}\gamma^k\rho(x_k,u_k) \\ &= \rho(x,u)+\gamma\sum_{k=1}^{\infty}\gamma^{k-1}\rho(x_k,h(x_k)) \\ &= \rho(x,u)+\gamma R^h(f(x,u)) \end{aligned} \tag{2.3}$$

where the definition (2.1) of the return was used in the last step. So, (2.2) has been obtained.

The optimal Q-function is defined as the best Q-function that can be obtained by any policy:

$$Q^*(x,u)=\max_h Q^h(x,u) \tag{2.4}$$

Any policy h^* that selects at each state an action with the largest optimal Q-value, i.e., that satisfies:

$$h^*(x)\in\arg\max_u Q^*(x,u) \tag{2.5}$$

is optimal (it maximizes the return). In general, for a given Q-function Q, a policy h that satisfies:

$$h(x)\in\arg\max_u Q(x,u) \tag{2.6}$$

is said to be *greedy* in Q. So, finding an optimal policy can be done by first finding Q^*, and then using (2.5) to compute a greedy policy in Q^*.

Note that, for simplicity of notation, we implicitly assume that the maximum in (2.4) exists, and also in similar equations in the sequel. When the maximum does not exist, the "max" operator should be replaced by the supremum operator. For the computation of greedy actions in (2.5), (2.6), and in similar equations in the sequel, the maximum must exist to ensure the existence of a greedy policy; this can be guaranteed under certain technical assumptions (Bertsekas and Shreve, 1978, Chapter 9).

The Q-functions Q^h and Q^* are recursively characterized by the *Bellman equations*, which are of central importance for value iteration and policy iteration algorithms. The Bellman equation for Q^h states that the value of taking action u in state x under the policy h equals the sum of the immediate reward and the discounted value achieved by h in the next state:

$$Q^h(x,u)=\rho(x,u)+\gamma Q^h(f(x,u),h(f(x,u))) \tag{2.7}$$

This Bellman equation can be derived from the second step in (2.3), as follows:

$$\begin{aligned} Q^h(x,u) &= \rho(x,u)+\gamma\sum_{k=1}^{\infty}\gamma^{k-1}\rho(x_k,h(x_k)) \\ &= \rho(x,u)+\gamma\left[\rho(f(x,u),h(f(x,u)))+\gamma\sum_{k=2}^{\infty}\gamma^{k-2}\rho(x_k,h(x_k))\right] \\ &= \rho(x,u)+\gamma Q^h(f(x,u),h(f(x,u))) \end{aligned}$$

where $(x_0, u_0) = (x, u)$, $x_{k+1} = f(x_k, u_k)$ for $k \geq 0$, and $u_k = h(x_k)$ for $k \geq 1$.

The Bellman optimality equation characterizes Q^*, and states that the optimal value of action u taken in state x equals the sum of the immediate reward and the discounted optimal value obtained by the best action in the next state:

$$Q^*(x,u) = \rho(x,u) + \gamma \max_{u'} Q^*(f(x,u),u') \tag{2.8}$$

The V-function $V^h : X \to \mathbb{R}$ of a policy h is the return obtained by starting from a particular state and following h. This V-function can be computed from the Q-function of policy h:

$$V^h(x) = R^h(x) = Q^h(x, h(x)) \tag{2.9}$$

The optimal V-function is the best V-function that can be obtained by any policy, and can be computed from the optimal Q-function:

$$V^*(x) = \max_h V^h(x) = \max_u Q^*(x,u) \tag{2.10}$$

An optimal policy h^* can be computed from V^*, by using the fact that it satisfies:

$$h^*(x) \in \arg\max_u [\rho(x,u) + \gamma V^*(f(x,u))] \tag{2.11}$$

Using this formula is more difficult than using (2.5); in particular, a model of the MDP is required in the form of the dynamics f and the reward function ρ. Because the Q-function also depends on the action, it already includes information about the quality of transitions. In contrast, the V-function only describes the quality of the states; in order to infer the quality of transitions, they must be explicitly taken into account. This is what happens in (2.11), and this also explains why it is more difficult to compute policies from V-functions. Because of this difference, Q-functions will be preferred to V-functions throughout this book, even though they are more costly to represent than V-functions, as they depend both on x and u.

The V-functions V^h and V^* satisfy the following Bellman equations, which can be interpreted similarly to (2.7) and (2.8):

$$V^h(x) = \rho(x, h(x)) + \gamma V^h(f(x, h(x))) \tag{2.12}$$

$$V^*(x) = \max_u [\rho(x,u) + \gamma V^*(f(x,u))] \tag{2.13}$$

2.2.2 Stochastic setting

In a stochastic MDP, the next state is not deterministically given by the current state and action. Instead, the next state is a random variable, and the current state and action give the probability density of this random variable.

More formally, the deterministic transition function f is replaced by a transition probability function $\tilde{f} : X \times U \times X \to [0, \infty)$. After action u_k is taken in state x_k, the probability that the next state, x_{k+1}, belongs to a region $X_{k+1} \subseteq X$ is:

$$\mathrm{P}(x_{k+1} \in X_{k+1} \,|\, x_k, u_k) = \int_{X_{k+1}} \tilde{f}(x_k, u_k, x') \mathrm{d}x'$$

For any x and u, $\tilde{f}(x,u,\cdot)$ must define a valid probability density function of the argument "$\cdot$", where the dot stands for the random variable x_{k+1}. Because rewards are associated with transitions, and the transitions are no longer fully determined by the current state and action, the reward function also has to depend on the next state, $\tilde{\rho} : X \times U \times X \to \mathbb{R}$. After each transition to a state x_{k+1}, a reward r_{k+1} is received according to:

$$r_{k+1} = \tilde{\rho}(x_k, u_k, x_{k+1})$$

where we assume that $\|\tilde{\rho}\|_\infty = \sup_{x,u,x'} \tilde{\rho}(x,u,x')$ is finite. Note that $\tilde{\rho}$ is a deterministic function of the transition (x_k, u_k, x_{k+1}). This means that, once x_{k+1} has been generated, the reward r_{k+1} is fully determined. In general, the reward can also depend stochastically on the entire transition (x_k, u_k, x_{k+1}). If it does, to simplify notation, we assume that $\tilde{\rho}$ gives the *expected* reward after the transition.

When the state space is countable (e.g., discrete), the transition function can also be given as $\bar{f} : X \times U \times X \to [0,1]$, where the probability of reaching x' after taking u_k in x_k is:

$$\mathrm{P}\left(x_{k+1} = x' \,|\, x_k, u_k\right) = \bar{f}(x_k, u_k, x') \tag{2.14}$$

For any x and u, the function $\bar{f}$ must satisfy $\sum_{x'} \bar{f}(x,u,x') = 1$. The function $\tilde{f}$ is a generalization of $\bar{f}$ to uncountable (e.g., continuous) state spaces; in such spaces, the probability of ending up in a given state x' is generally 0, making a description of the form $\bar{f}$ inappropriate.

In the stochastic case, the Markov property requires that x_k and u_k fully determine the probability density of the next state.

Developing an analytical expression for the transition probability function $\tilde{f}$ is generally a difficult task. Fortunately, as previously noted in Section 2.1, most DP (model-based) algorithms can work with a generative model, which only needs to generate samples of the next state and corresponding rewards for any given pair of current state and action taken.

Example 2.2 The stochastic cleaning-robot MDP. Consider again the cleaning-robot problem of Example 2.1. Assume that, due to uncertainties in the environment, such as a slippery floor, state transitions are no longer deterministic. When trying to move in a certain direction, the robot succeeds with a probability of 0.8. With a probability of 0.15 it remains in the same state, and it may even move in the opposite direction with a probability of 0.05 (see also Figure 2.4).

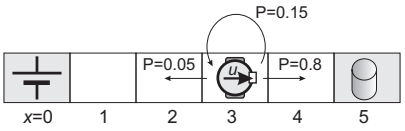


FIGURE 2.4
The stochastic cleaning-robot problem. The robot intends to move right, but it may instead end up standing still or moving left, with different probabilities.

Because the state space is discrete, a transition model of the form (2.14) is appropriate. The transition function $\bar{f}$ that models the probabilistic transitions described above is shown in Table 2.1. In this table, the rows correspond to combinations of current states and actions taken, while the columns correspond to future states. Note that the transitions from any terminal state still lead deterministically to the same terminal state, regardless of the action.

TABLE 2.1 Dynamics of the stochastic, cleaning-robot MDP.

(x,u)	$\bar{f}(x,u,0)$	$\bar{f}(x,u,1)$	$\bar{f}(x,u,2)$	$\bar{f}(x,u,3)$	$\bar{f}(x,u,4)$	$\bar{f}(x,u,5)$
$(0,-1)$	1	0	0	0	0	0
$(1,-1)$	0.8	0.15	0.05	0	0	0
$(2,-1)$	0	0.8	0.15	0.05	0	0
$(3,-1)$	0	0	0.8	0.15	0.05	0
$(4,-1)$	0	0	0	0.8	0.15	0.05
$(5,-1)$	0	0	0	0	0	1
$(0,1)$	1	0	0	0	0	0
$(1,1)$	0.05	0.15	0.8	0	0	0
$(2,1)$	0	0.05	0.15	0.8	0	0
$(3,1)$	0	0	0.05	0.15	0.8	0
$(4,1)$	0	0	0	0.05	0.15	0.8
$(5,1)$	0	0	0	0	0	1

The robot receives rewards as in the deterministic case: upon reaching state 5, it is rewarded with 5, and upon reaching state 0, it is rewarded with 1. The corresponding reward function, in the form $\tilde{\rho} : X \times U \times X \to \mathbb{R}$, is:

$$\tilde{\rho}(x,u,x') = \begin{cases} 5 & \text{if } x \neq 5 \text{ and } x' = 5 \\ 1 & \text{if } x \neq 0 \text{ and } x' = 0 \\ 0 & \text{otherwise} \end{cases}$$

□

Optimality in the stochastic setting

The *expected* infinite-horizon discounted return of an initial state x_0 under a (deterministic) policy h is:[4]

$$\begin{aligned} R^h(x_0) &= \lim_{K\to\infty} \mathrm{E}_{x_{k+1}\sim\tilde{f}(x_k,h(x_k),\cdot)} \left\{ \sum_{k=0}^{K} \gamma^k r_{k+1} \right\} \\ &= \lim_{K\to\infty} \mathrm{E}_{x_{k+1}\sim\tilde{f}(x_k,h(x_k),\cdot)} \left\{ \sum_{k=0}^{K} \gamma^k \tilde{\rho}(x_k,h(x_k),x_{k+1}) \right\} \end{aligned} \tag{2.15}$$

[4]We assume that the MDP and the policies h have suitable properties such that the expected return and the Bellman equations in the remainder of this section are well defined. See, e.g., (Bertsekas and Shreve, 1978, Chapter 9) and (Bertsekas, 2007, Appendix A) for a discussion of these properties.

where E denotes the expectation operator, and the notation $x_{k+1} \sim \tilde{f}(x_k, h(x_k), \cdot)$ means that the random variable x_{k+1} is drawn from the density $\tilde{f}(x_k, h(x_k), \cdot)$ at each step k. The discussion of Section 2.2.1 regarding the interpretation and choice of the discount factor also applies to the stochastic case. For any stochastic or deterministic MDP, when using the infinite-horizon discounted return (2.15) or (2.1), and under certain technical assumptions on the elements of the MDP, there exists at least one stationary deterministic optimal policy (Bertsekas and Shreve, 1978, Chapter 9). Therefore, we will mainly consider stationary deterministic policies in the sequel.

Expected undiscounted, average, and finite-horizon returns (see Section 2.2.1) can be defined analogously to (2.15).

Value functions and the Bellman equations in the stochastic setting

To obtain the Q-function of a policy h, the definition (2.2) is generalized to the stochastic case, as follows. The Q-function is the *expected* return under the stochastic transitions, when starting in a particular state, applying a particular action, and following the policy h thereafter:

$$Q^h(x,u) = \mathrm{E}_{x' \sim \tilde{f}(x,u,\cdot)} \left\{ \tilde{\rho}(x,u,x') + \gamma R^h(x') \right\} \tag{2.16}$$

The definition of the optimal Q-function Q^* remains unchanged from the deterministic case (2.4), and is repeated here for easy reference:

$$Q^*(x,u) = \max_h Q^h(x,u)$$

Similarly, optimal policies can still be computed from Q^* as in the deterministic case, because they satisfy (2.5), also repeated here:

$$h^*(x) \in \arg\max_u Q^*(x,u)$$

The Bellman equations for Q^h and Q^* are given in terms of expectations over the one-step stochastic transitions:

$$Q^h(x,u) = \mathrm{E}_{x' \sim \tilde{f}(x,u,\cdot)} \left\{ \tilde{\rho}(x,u,x') + \gamma Q^h(x', h(x')) \right\} \tag{2.17}$$

$$Q^*(x,u) = \mathrm{E}_{x' \sim \tilde{f}(x,u,\cdot)} \left\{ \tilde{\rho}(x,u,x') + \gamma \max_{u'} Q^*(x',u') \right\} \tag{2.18}$$

The definition of the V-function V^h of a policy h, as well as of the optimal V-function V^*, are the same as for the deterministic case (2.9), (2.10):

$$V^h(x) = R^h(x)$$
$$V^*(x) = \max_h V^h(x)$$

However, the computation of optimal policies from V^* becomes more difficult, involving an expectation that did not appear in the deterministic case:

$$h^*(x) \in \arg\max_u \mathrm{E}_{x' \sim \tilde{f}(x,u,\cdot)} \left\{ \tilde{\rho}(x,u,x') + \gamma V^*(x') \right\} \tag{2.19}$$

In contrast, computing an optimal policy from Q^* is as simple as in the deterministic case, which is yet another reason for using Q-functions in practice.

The Bellman equations for V^h and V^* are obtained from (2.12) and (2.13), by considering expectations over the one-step stochastic transitions:

$$V^h(x) = \mathrm{E}_{x' \sim \tilde{f}(x,h(x),\cdot)} \left\{ \tilde{\rho}(x, h(x), x') + \gamma V^h(x') \right\} \tag{2.20}$$

$$V^*(x) = \max_u \mathrm{E}_{x' \sim \tilde{f}(x,u,\cdot)} \left\{ \tilde{\rho}(x, u, x') + \gamma V^*(x') \right\} \tag{2.21}$$

Note that in the Bellman equation for V^* (2.21), the maximization is outside the expectation operator, whereas in the Bellman equation for Q^* (2.18), the order of the expectation and maximization is reversed.

Clearly, all the equations for deterministic MDPs are a special case of the equations for stochastic MDPs. The deterministic case is obtained by using a degenerate density $\tilde{f}(x,u,\cdot)$ that assigns all the probability mass to $f(x,u)$. The deterministic reward function is obtained as $\rho(x,u) = \tilde{\rho}(x,u,f(x,u))$.

The entire class of value iteration algorithms, introduced in Section 2.3, revolves around solving the Bellman optimality equations (2.18) or (2.21) to find, respectively, the optimal Q-function or the optimal V-function (in the deterministic case, (2.8) or (2.13) are solved instead). Similarly, policy evaluation, which is a core component of the policy iteration algorithms introduced in Section 2.4, revolves around solving (2.17) or (2.20) to find, respectively, Q^h or V^h (in the deterministic case (2.7) or (2.12) are solved instead).

2.3 Value iteration

Value iteration techniques use the Bellman optimality equation to iteratively compute an optimal value function, from which an optimal policy is derived. We first present DP (model-based) algorithms for value iteration, followed by RL (model-free) algorithms. DP algorithms like V-iteration (Bertsekas, 2007, Section 1.3) solve the Bellman optimality equation by using knowledge of the transition and reward functions. RL techniques either learn a model, e.g., Dyna (Sutton, 1990), or do not use an explicit model at all, e.g., Q-learning (Watkins and Dayan, 1992).

2.3.1 Model-based value iteration

We will next introduce the model-based *Q-iteration* algorithm, as an illustrative example from the class of model-based value iteration algorithms. Let the set of all the Q-functions be denoted by $\mathscr{Q}$. Then, the Q-iteration mapping $T : \mathscr{Q} \to \mathscr{Q}$, computes the right-hand side of the Bellman optimality equation (2.8) or (2.18) for any

Q-function.[5] In the deterministic case, this mapping is:

$$[T(Q)](x,u) = \rho(x,u) + \gamma \max_{u'} Q(f(x,u),u') \tag{2.22}$$

and in the stochastic case, it is:

$$[T(Q)](x,u) = \mathrm{E}_{x' \sim \tilde{f}(x,u,\cdot)} \left\{ \tilde{\rho}(x,u,x') + \gamma \max_{u'} Q(x',u') \right\} \tag{2.23}$$

Note that if the state space is countable (e.g., finite), a transition model of the form (2.14) is appropriate, and the Q-iteration mapping for the stochastic case (2.23) can be written as the simpler summation:

$$[T(Q)](x,u) = \sum_{x'} \bar{f}(x,u,x') \left[\tilde{\rho}(x,u,x') + \gamma \max_{u'} Q(x',u') \right] \tag{2.24}$$

The same notation is used for the Q-iteration mapping both in the deterministic case and in the stochastic case, because the analysis given below applies to both cases, and the definition (2.22) of T is a special case of (2.23) (or of (2.24) for countable state spaces).

The Q-iteration algorithm starts from an arbitrary Q-function Q_0 and at each iteration ℓ updates the Q-function using:

$$Q_{\ell+1} = T(Q_\ell) \tag{2.25}$$

It can be shown that T is a contraction with factor $\gamma < 1$ in the infinity norm, i.e., for any pair of functions Q and Q', it is true that:

$$\|T(Q) - T(Q')\|_\infty \leq \gamma \|Q - Q'\|_\infty$$

Because T is a contraction, it has a unique fixed point (Istratescu, 2002). Additionally, when rewritten using the Q-iteration mapping, the Bellman optimality equation (2.8) or (2.18) states that Q^* is a fixed point of T, i.e.:

$$Q^* = T(Q^*) \tag{2.26}$$

Hence, the unique fixed point of T is actually Q^*, and Q-iteration asymptotically converges to Q^* as $\ell \to \infty$. Moreover, Q-iteration converges to Q^* at a rate of γ, in the sense that $\|Q_{\ell+1} - Q^*\|_\infty \leq \gamma \|Q_\ell - Q^*\|_\infty$. An optimal policy can be computed from Q^* with (2.5).

Algorithm 2.1 presents Q-iteration for deterministic MDPs in an explicit, procedural form, wherein T is computed using (2.22). Similarly, Algorithm 2.2 presents Q-iteration for stochastic MDPs with countable state spaces, using the expression (2.24) for T.

[5] The term "mapping" is used to refer to functions that work with other functions as inputs and/or outputs; as well as to compositions of such functions. The term is used to differentiate mappings from ordinary functions, which only have numerical scalars, vectors, or matrices as inputs and/or outputs.

ALGORITHM 2.1 Q-iteration for deterministic MDPs.

Input: dynamics f, reward function ρ, discount factor γ

initialize Q-function, e.g., $Q_0 \leftarrow 0$
repeat at every iteration $\ell = 0, 1, 2, \ldots$
  for every (x,u) **do**
    $Q_{\ell+1}(x,u) \leftarrow \rho(x,u) + \gamma \max_{u'} Q_\ell(f(x,u),u')$
  end for
until $Q_{\ell+1} = Q_\ell$

Output: $Q^* = Q_\ell$

ALGORITHM 2.2 Q-iteration for stochastic MDPs with countable state spaces.

Input: dynamics $\bar{f}$, reward function $\tilde{\rho}$, discount factor γ

initialize Q-function, e.g., $Q_0 \leftarrow 0$
repeat at every iteration $\ell = 0, 1, 2, \ldots$
  for every (x,u) **do**
    $Q_{\ell+1}(x,u) \leftarrow \sum_{x'} \bar{f}(x,u,x') [\tilde{\rho}(x,u,x') + \gamma \max_{u'} Q_\ell(x',u')]$
  end for
until $Q_{\ell+1} = Q_\ell$

Output: $Q^* = Q_\ell$

The results given above only guarantee the asymptotic convergence of Q-iteration, hence the stopping criterion of Algorithms 2.1 and 2.2 may only be satisfied asymptotically. In practice, it is also important to guarantee the performance of Q-iteration when the algorithm is stopped after a finite number of iterations. The following result holds both in the deterministic case and in the stochastic case. Given a suboptimality bound $\varsigma_{\text{QI}} > 0$, where the subscript "QI" stands for "Q-iteration," a finite number L of iterations can be (conservatively) chosen with:

$$L = \left\lceil \log_\gamma \frac{\varsigma_{\text{QI}}(1-\gamma)^2}{2\|\rho\|_\infty} \right\rceil \tag{2.27}$$

so that the suboptimality of a policy h_L that is greedy in Q_L is guaranteed to be at most ς_{QI}, in the sense that $\|V^{h_L} - V^*\|_\infty \leq \varsigma_{\text{QI}}$. Here, $\lceil \cdot \rceil$ is the smallest integer larger than or equal to the argument (ceiling). Equation (2.27) follows from the bound (Ernst et al., 2005):

$$\|V^{h_L} - V^*\|_\infty \leq \frac{2\gamma^L \|\rho\|_\infty}{(1-\gamma)^2}$$

on the suboptimality of h_L, by requiring that $2\frac{\gamma^L \|\rho\|_\infty}{(1-\gamma)^2} \leq \varsigma_{\text{QI}}$.

Alternatively, Q-iteration could be stopped when the difference between two consecutive Q-functions decreases below a given threshold $\varepsilon_{\text{QI}} > 0$, i.e., when $\|Q_{\ell+1} - Q_\ell\|_\infty \leq \varepsilon_{\text{QI}}$. This can also be guaranteed to happen after a finite number of iterations, due to the contracting nature of the Q-iteration updates.

A V-iteration algorithm that computes the optimal V-function can be developed along similar lines, using the Bellman optimality equation (2.13) in the deterministic case, or (2.21) in the stochastic case. Note that the name "value iteration" is typically used for the V-iteration algorithm in the literature, whereas we use it to refer more generally to the entire class of algorithms that use the Bellman optimality equations to compute optimal value functions. (Recall that we similarly use "value function" to refer to Q-functions and V-functions collectively.)

Computational cost of Q-iteration for finite MDPs

Next, we investigate the computational cost of Q-iteration when applied to an MDP with a finite number of states and actions. Denote by $|\cdot|$ the cardinality of the argument set "$\cdot$", so that $|X|$ denotes the finite number of states and $|U|$ denotes the finite number of actions.

Consider first the deterministic case, for which Algorithm 2.1 can be used. Assume that, when updating the Q-value for a given state-action pair (x,u), the maximization over the action space U is solved by enumeration over its $|U|$ elements, and $f(x,u)$ is computed once and then stored and reused. Updating the Q-value then requires $2+|U|$ function evaluations, where the functions being evaluated are f, ρ, and the current Q-function Q_ℓ. Since at every iteration, the Q-values of $|X|\,|U|$ state-action pairs have to be updated, the cost per iteration is $|X|\,|U|\,(2+|U|)$. So, the total cost of L Q-iterations for a deterministic, finite MDP is:

$$L\,|X|\,|U|\,(2+|U|) \tag{2.28}$$

The number L of iterations can be chosen, e.g., by imposing a suboptimality bound ς_{QI} and using (2.27).

In the stochastic case, because the state space is finite, Algorithm 2.2 can be used. Assuming that the maximization over u' is implemented using enumeration, the cost of updating the Q-value for a given pair (x,u) is $|X|\,(2+|U|)$, where the functions being evaluated are $\bar{f}$, $\tilde{\rho}$, and Q_ℓ. The cost per iteration is $|X|^2\,|U|\,(2+|U|)$, and the total cost of L Q-iterations for a stochastic, finite MDP is thus:

$$L\,|X|^2\,|U|\,(2+|U|) \tag{2.29}$$

which is larger by a factor $|X|$ than the cost (2.28) for the deterministic case.

Example 2.3 Q-iteration for the cleaning robot. In this example, we apply Q-iteration to the cleaning-robot problem of Examples 2.1 and 2.2. The discount factor γ is set to 0.5.

Consider first the deterministic variant of Example 2.1. For this variant, Q-iteration is implemented as Algorithm 2.1. Starting from an identically zero initial Q-function, $Q_0 = 0$, this algorithm produces the sequence of Q-functions given in the first part of Table 2.2 (above the dashed line), where each cell shows the Q-values of the two actions in a certain state, separated by a semicolon. For instance:

$$Q_3(2,1) = \rho(2,1) + \gamma \max_u Q_2(f(2,1),u) = 0 + 0.5 \max_u Q_2(3,u) = 0 + 0.5 \cdot 2.5 = 1.25$$

TABLE 2.2
Q-iteration results for the deterministic cleaning-robot problem.

	$x=0$	$x=1$	$x=2$	$x=3$	$x=4$	$x=5$
Q_0	0; 0	0; 0	0; 0	0; 0	0; 0	0; 0
Q_1	0; 0	1; 0	0.5; 0	0.25; 0	0.125; 5	0; 0
Q_2	0; 0	1; 0.25	0.5; 0.125	0.25; 2.5	1.25; 5	0; 0
Q_3	0; 0	1; 0.25	0.5; 1.25	0.625; 2.5	1.25; 5	0; 0
Q_4	0; 0	1; 0.625	0.5; 1.25	0.625; 2.5	1.25; 5	0; 0
Q_5	0; 0	1; 0.625	0.5; 1.25	0.625; 2.5	1.25; 5	0; 0
h^*	$*$	-1	1	1	1	$*$
V^*	0	1	1.25	2.5	5	0

The algorithm converges after 5 iterations; $Q_5 = Q_4 = Q^*$. The last two rows of the table (below the dashed line) also give the optimal policies, computed from Q^* with (2.5), and the optimal V-function V^*, computed from Q^* with (2.10). In the policy representation, the symbol "$*$" means that any action can be taken in that state without changing the quality of the policy. The total number of function evaluations required by the algorithm in the deterministic case is:

$$5\,|X|\,|U|\,(2+|U|) = 5 \cdot 6 \cdot 2 \cdot 4 = 240$$

Consider next the stochastic variant of the cleaning-robot problem, introduced in Example 2.2. For this variant, Q-iteration is implemented by using Algorithm 2.2, and produces the sequence of Q-functions illustrated in the first part of Table 2.3 (not all the iterations are shown). The algorithm fully converges after 22 iterations.

TABLE 2.3
Q-iteration results for the stochastic cleaning-robot problem. Q-function and V-function values are rounded to 3 decimal places.

	$x=0$	$x=1$	$x=2$	$x=3$	$x=4$	$x=5$
Q_0	0; 0	0; 0	0; 0	0; 0	0; 0	0; 0
Q_1	0; 0	0.800; 0.110	0.320; 0.044	0.128; 0.018	0.301; 4.026	0; 0
Q_2	0; 0	0.868; 0.243	0.374; 0.101	0.260; 1.639	1.208; 4.343	0; 0
Q_3	0; 0	0.874; 0.265	0.419; 0.709	0.515; 1.878	1.327; 4.373	0; 0
Q_4	0; 0	0.883; 0.400	0.453; 0.826	0.581; 1.911	1.342; 4.376	0; 0
...	...	...	...	...	...	...
Q_{12}	0; 0	0.888; 0.458	0.467; 0.852	0.594; 1.915	1.344; 4.376	0; 0
...	...	...	...	...	...	...
Q_{22}	0; 0	0.888; 0.458	0.467; 0.852	0.594; 1.915	1.344; 4.376	0; 0
h^*	$*$	-1	1	1	1	$*$
V^*	0	0.888	0.852	1.915	4.376	0

The optimal policies and the optimal V-function obtained are also shown in Table 2.3 (below the dashed line). While the optimal Q-function and the optimal V-function are different from those obtained in the deterministic case, the optimal policies remain the same. The total number of function evaluations required by the algorithm in the stochastic case is:

$$22\,|X|^2\,|U|\,(2+|U|) = 22 \cdot 6^2 \cdot 2 \cdot 4 = 6336$$

which is considerably greater than in the deterministic case.

If we impose a suboptimality bound $\varsigma_{\text{QI}} = 0.01$ and apply (2.27), we find that Q-iteration should run for $L = 12$ iterations in order to guarantee this bound, where the maximum absolute reward $\|\rho\|_\infty = 5$ and the discount factor $\gamma = 0.5$ have also been used. So, in the deterministic case, the algorithm fully converges to its fixed point in fewer iterations than the conservative number given by (2.27). In the stochastic case, even though the algorithm does not fully converge after 12 iterations (instead requiring 22 iterations), the Q-function at iteration 12 (shown in Table 2.3) is already very accurate, and a policy that is greedy in this Q-function is fully optimal. The suboptimality of such a policy is 0, which is smaller than the imposed bound ς_{QI}. In fact, for any iteration $\ell \geq 3$, the Q-function Q_ℓ would produce an optimal policy, which means that $L = 12$ is also conservative in the stochastic case. □

2.3.2 Model-free value iteration and the need for exploration

We have discussed until now model-based value iteration. We next consider RL, model-free value iteration algorithms, and discuss *Q-learning*, the most widely used algorithm from this class. Q-learning starts from an arbitrary initial Q-function Q_0 and updates it without requiring a model, using instead observed state transitions and rewards, i.e., data tuples of the form $(x_k, u_k, x_{k+1}, r_{k+1})$ (Watkins, 1989; Watkins and Dayan, 1992). After each transition, the Q-function is updated using such a data tuple, as follows:

$$Q_{k+1}(x_k,u_k) = Q_k(x_k,u_k) + \alpha_k[r_{k+1} + \gamma \max_{u'} Q_k(x_{k+1},u') - Q_k(x_k,u_k)] \qquad (2.30)$$

where $\alpha_k \in (0,1]$ is the learning rate. The term between square brackets is the temporal difference, i.e., the difference between the updated estimate $r_{k+1} + \gamma \max_{u'} Q_k(x_{k+1},u')$ of the optimal Q-value of (x_k,u_k), and the current estimate $Q_k(x_k,u_k)$. In the deterministic case, the new estimate is actually the Q-iteration mapping (2.22) applied to Q_k in the state-action pair (x_k,u_k), where $\rho(x_k,u_k)$ has been replaced by the observed reward r_{k+1}, and $f(x_k,u_k)$ by the observed next-state x_{k+1}. In the stochastic case, these replacements provide a single sample of the random quantity whose expectation is computed by the Q-iteration mapping (2.23), and thus Q-learning can be seen as a sample-based, stochastic approximation procedure based on this mapping (Singh et al., 1995; Bertsekas and Tsitsiklis, 1996, Section 5.6).

As the number of transitions k approaches infinity, Q-learning asymptotically converges to Q^* if the state and action spaces are discrete and finite, and under the following conditions (Watkins and Dayan, 1992; Tsitsiklis, 1994; Jaakkola et al., 1994):

- The sum $\sum_{k=0}^{\infty} \alpha_k^2$ produces a finite value, whereas the sum $\sum_{k=0}^{\infty} \alpha_k$ produces an infinite value.
- All the state-action pairs are (asymptotically) visited infinitely often.

The first condition is not difficult to satisfy. For instance, a satisfactory standard choice is:

$$\alpha_k = \frac{1}{k} \tag{2.31}$$

In practice, the learning rate schedule may require tuning, because it influences the number of transitions required by Q-learning to obtain a good solution. A good choice for the learning rate schedule depends on the problem at hand.

The second condition can be satisfied if, among other things, the controller has a nonzero probability of selecting any action in every encountered state; this is called exploration. The controller also has to exploit its current knowledge in order to obtain good performance, e.g., by selecting greedy actions in the current Q-function. This is a typical illustration of the exploration-exploitation trade-off in online RL. A classical way to balance exploration with exploitation in Q-learning is ε-greedy exploration (Sutton and Barto, 1998, Section 2.2), which selects actions according to:

$$u_k = \begin{cases} u \in \arg\max_{\bar{u}} Q_k(x_k, \bar{u}) & \text{with probability } 1 - \varepsilon_k \\ \text{a uniformly random action in } U & \text{with probability } \varepsilon_k \end{cases} \tag{2.32}$$

where $\varepsilon_k \in (0,1)$ is the exploration probability at step k. Another option is to use Boltzmann exploration (Sutton and Barto, 1998, Section 2.3), which at step k selects an action u with probability:

$$\mathrm{P}(u \,|\, x_k) = \frac{e^{Q_k(x_k,u)/\tau_k}}{\sum_{\bar{u}} e^{Q_k(x_k,\bar{u})/\tau_k}} \tag{2.33}$$

where the temperature $\tau_k \geq 0$ controls the randomness of the exploration. When $\tau_k \to 0$, (2.33) is equivalent to greedy action selection, while for $\tau_k \to \infty$, action selection is uniformly random. For nonzero, finite values of τ_k, higher-valued actions have a greater chance of being selected than lower-valued ones.

Usually, the exploration diminishes over time, so that the policy used asymptotically becomes greedy and therefore (as $Q_k \to Q^*$) optimal. This can be achieved by making ε_k or τ_k approach 0 as k grows. For instance, an ε-greedy exploration schedule of the form $\varepsilon_k = 1/k$ diminishes to 0 as $k \to \infty$, while still satisfying the second convergence condition of Q-learning, i.e., while allowing infinitely many visits to all the state-action pairs (Singh et al., 2000). Notice the similarity of this exploration schedule with the learning rate schedule (2.31). For a schedule of the Boltzmann exploration temperature τ_k that decreases to 0 while satisfying the convergence conditions, see (Singh et al., 2000). Like the learning rate schedule, the exploration schedule has a significant effect on the performance of Q-learning.

Algorithm 2.3 presents Q-learning with ε-greedy exploration. Note that an idealized, infinite-time online setting is considered for this algorithm, in which no termination condition is specified and no explicit output is produced. Instead, the result of

the algorithm is the improvement of the control performance achieved while interacting with the process. A similar setting will be considered for other online learning algorithms described in this book, with the implicit understanding that, in practice, the algorithms will of course be stopped after a finite number of steps. When Q-learning is stopped, the resulting Q-function and the corresponding greedy policy can be interpreted as outputs and reused.

ALGORITHM 2.3 Q-learning with ε-greedy exploration.

Input: discount factor γ,
exploration schedule $\{\varepsilon_k\}_{k=0}^{\infty}$, learning rate schedule $\{\alpha_k\}_{k=0}^{\infty}$

1: initialize Q-function, e.g., $Q_0 \leftarrow 0$
2: measure initial state x_0
3: **for** every time step $k = 0, 1, 2, \ldots$ **do**
4: $u_k \leftarrow \begin{cases} u \in \arg\max_{\bar{u}} Q_k(x_k, \bar{u}) & \text{with probability } 1-\varepsilon_k \text{ (exploit)} \\ \text{a uniformly random action in } U & \text{with probability } \varepsilon_k \text{ (explore)} \end{cases}$
5: apply u_k, measure next state x_{k+1} and reward r_{k+1}
6: $Q_{k+1}(x_k, u_k) \leftarrow Q_k(x_k, u_k) + \alpha_k[r_{k+1} + \gamma \max_{u'} Q_k(x_{k+1}, u') - Q_k(x_k, u_k)]$
7: **end for**

Note that this discussion has not been all-encompassing; the ε-greedy and Boltzmann exploration procedures can also be used in other online RL algorithms besides Q-learning, and a variety of other exploration procedures exist. For instance, the policy can be biased towards actions that have not recently been taken, or that may lead the system towards rarely visited areas of the state space (Thrun, 1992). The value function can also be initialized to be larger than the true returns, in a method known as "optimism in the face of uncertainty" (Sutton and Barto, 1998, Section 2.7). Because the return estimates have been adjusted downwards for any actions already taken, greedy action selection leads to exploring novel actions. Confidence intervals for the returns can be estimated, and the action with largest upper confidence bound, i.e., with the best potential for good returns, can be chosen (Kaelbling, 1993). Many authors have also studied the exploration-exploitation trade-off for specific types of problems, such as problems with linear transition dynamics (Feldbaum, 1961), and problems without any dynamics, for which the state space reduces to a single element (Auer et al., 2002; Audibert et al., 2007).

2.4 Policy iteration

Having introduced value iteration in Section 2.3, we now consider *policy iteration*, the second major class of DP/RL algorithms. Policy iteration algorithms evaluate policies by constructing their value functions, and use these value functions to find new, improved policies (Bertsekas, 2007, Section 1.3). As a representative example

of policy iteration, consider an offline algorithm that evaluates policies using their Q-functions. This algorithm starts with an arbitrary policy h_0. At every iteration ℓ, the Q-function Q^{h_ℓ} of the current policy h_ℓ is determined; this step is called *policy evaluation*. Policy evaluation is performed by solving the Bellman equation (2.7) in the deterministic case, or (2.17) in the stochastic case. When policy evaluation is complete, a new policy $h_{\ell+1}$ that is greedy in Q^h is found:

$$h_{\ell+1}(x) \in \arg\max_u Q^{h_\ell}(x,u) \tag{2.34}$$

This step is called *policy improvement*. The entire procedure is summarized in Algorithm 2.4. The sequence of Q-functions produced by policy iteration asymptotically converges to Q^* as $\ell \to \infty$. Simultaneously, an optimal policy h^* is obtained.

ALGORITHM 2.4 Policy iteration with Q-functions.

initialize policy h_0
repeat at every iteration $\ell = 0, 1, 2, \ldots$
  find Q^{h_ℓ}, the Q-function of h_ℓ ▷ policy evaluation
  $h_{\ell+1}(x) \in \arg\max_u Q^{h_\ell}(x,u)$ ▷ policy improvement
until $h_{\ell+1} = h_\ell$
Output: $h^* = h_\ell$, $Q^* = Q^{h_\ell}$

The crucial component of policy iteration is policy evaluation. Policy improvement can be performed by solving static optimization problems, e.g., of the form (2.34) when Q-functions are used – often an easier challenge.

In the remainder of this section, we first discuss DP (model-based) policy iteration, followed by RL (model-free) policy iteration. We pay special attention to the policy evaluation component.

2.4.1 Model-based policy iteration

In the model-based setting, the policy evaluation step employs knowledge of the transition and reward functions. A model-based iterative algorithm for policy evaluation can be given that is similar to Q-iteration, which will be called *policy evaluation for Q-functions*. Analogously to the Q-iteration mapping T (2.22), a policy evaluation mapping $T^h : \mathscr{Q} \to \mathscr{Q}$ is defined, which computes the right-hand side of the Bellman equation for an arbitrary Q-function. In the deterministic case, this mapping is:

$$[T^h(Q)](x,u) = \rho(x,u) + \gamma Q(f(x,u), h(f(x,u))) \tag{2.35}$$

and in the stochastic case, it is:

$$[T^h(Q)](x,u) = \mathrm{E}_{x' \sim \tilde{f}(x,u,\cdot)} \left\{ \tilde{\rho}(x,u,x') + \gamma Q(x', h(x')) \right\} \tag{2.36}$$

Note that when the state space is countable, the transition model (2.14) is appropriate, and the policy evaluation mapping for the stochastic case (2.36) can be written as the

simpler summation:

$$[T^h(Q)](x,u) = \sum_{x'} \bar{f}(x,u,x') \left[\tilde{\rho}(x,u,x') + \gamma Q(x', h(x'))\right] \tag{2.37}$$

Policy evaluation for Q-functions starts from an arbitrary Q-function Q_0^h and at each iteration τ updates the Q-function using:[6]

$$Q_{\tau+1}^h = T^h(Q_\tau^h) \tag{2.38}$$

Like the Q-iteration mapping T, the policy evaluation mapping T^h is a contraction with a factor $\gamma < 1$ in the infinity norm, i.e., for any pair of functions Q and Q':

$$\|T^h(Q) - T^h(Q')\|_\infty \leq \gamma \|Q - Q'\|_\infty$$

So, T^h has a unique fixed point. Written in terms of the mapping T^h, the Bellman equation (2.7) or (2.17) states that this unique fixed point is actually Q^h:

$$Q^h = T^h(Q^h) \tag{2.39}$$

Therefore, policy evaluation for Q-functions (2.38) asymptotically converges to Q^h. Moreover, also because T^h is a contraction with factor γ, this variant of policy evaluation converges to Q^h at a rate of γ, in the sense that $\|Q_{\tau+1}^h - Q^h\|_\infty \leq \gamma \|Q_\tau^h - Q^h\|_\infty$.

Algorithm 2.5 presents policy evaluation for Q-functions in deterministic MDPs, while Algorithm 2.6 is used for stochastic MDPs with countable state spaces. In Algorithm 2.5, T^h is computed with (2.35), while in Algorithm 2.6, (2.37) is employed. Since the convergence condition of these algorithms is only guaranteed to be satisfied asymptotically, in practice they can be stopped, e.g., when the difference between consecutive Q-functions decreases below a given threshold, i.e., when $\|Q_{\tau+1}^h - Q_\tau^h\|_\infty \leq \varepsilon_{\text{PE}}$, where $\varepsilon_{\text{PE}} > 0$. Here, the subscript "PE" stands for "policy evaluation."

ALGORITHM 2.5 Policy evaluation for Q-functions in deterministic MDPs.

Input: policy h to be evaluated, dynamics f, reward function ρ, discount factor γ

initialize Q-function, e.g., $Q_0^h \leftarrow 0$
repeat at every iteration $\tau = 0, 1, 2, \ldots$
  for every (x,u) **do**
    $Q_{\tau+1}^h(x,u) \leftarrow \rho(x,u) + \gamma Q_\tau^h(f(x,u), h(f(x,u)))$
  end for
until $Q_{\tau+1}^h = Q_\tau^h$

Output: $Q^h = Q_\tau^h$

[6] A different iteration index τ is used for policy evaluation, because it runs in the inner loop of every (offline) policy iteration ℓ.

ALGORITHM 2.6
Policy evaluation for Q-functions in stochastic MDPs with countable state spaces.

Input: policy h to be evaluated, dynamics $\bar{f}$, reward function $\tilde{\rho}$, discount factor γ
1: initialize Q-function, e.g., $Q^h_0 \leftarrow 0$
2: **repeat** at every iteration $\tau = 0, 1, 2, \ldots$
3: **for** every (x,u) **do**
4: $Q^h_{\tau+1}(x,u) \leftarrow \sum_{x'} \bar{f}(x,u,x') \left[\tilde{\rho}(x,u,x') + \gamma Q^h_\tau(x', h(x'))\right]$
5: **end for**
6: **until** $Q^h_{\tau+1} = Q^h_\tau$
Output: $Q^h = Q^h_\tau$

There are also other ways to compute Q^h. For example, in the deterministic case, the mapping T^h (2.35) and equivalently the Bellman equation (2.7) are obviously linear in the Q-values. In the stochastic case, because X has a finite cardinality, the policy evaluation mapping T^h and equivalently the Bellman equation (2.39) can be written by using the summation (2.37), and are therefore also linear. Hence, when the state and action spaces are finite and the cardinality of $X \times U$ is not too large (e.g., up to several thousands), Q^h can be found by directly solving the linear system of equations given by the Bellman equation.

The entire derivation can be repeated and similar algorithms can be given for V-functions instead of Q-functions. Such algorithms are more popular in the literature, see, e.g., (Sutton and Barto, 1998, Section 4.1) and (Bertsekas, 2007, Section 1.3). Recall however, that policy improvements are more problematic when V-functions are used, because a model is required to find greedy policies, as seen in (2.11). Additionally, in the stochastic case, expectations over the one-step stochastic transitions have to be computed to find greedy policies, as seen in (2.19).

An important advantage of policy iteration over value iteration stems from the linearity of the Bellman equation for Q^h in the Q-values. In contrast, the Bellman optimality equation (for Q^*) is highly nonlinear due to the maximization at the right-hand side. This makes policy evaluation generally easier to solve than value iteration. Moreover, in practice, offline policy iteration algorithms often converge in a small number of iterations (Madani, 2002; Sutton and Barto, 1998, Section 4.3), possibly smaller than the number of iterations taken by offline value iteration algorithms. However, this does not mean that policy iteration is computationally less costly than value iteration. For instance, even though policy evaluation using Q-functions is generally less costly than Q-iteration, every single policy iteration requires a complete policy evaluation.

Computational cost of policy evaluation for Q-functions in finite MDPs

We next investigate the computational cost of policy evaluation for Q-functions (2.38) for an MDP with a finite number of states and actions. We also provide a comparison with the computational cost of Q-iteration. Note again that policy evaluation

is only one component of policy iteration; for an illustration of the computational cost of the entire policy iteration algorithm, see the upcoming Example 2.4.

In the deterministic case, policy evaluation for Q-functions can be implemented as in Algorithm 2.5. The computational cost of one iteration of this algorithm, measured by the number of function evaluations, is:

$$4\,|X|\,|U|$$

where the functions being evaluated are ρ, f, h, and the current Q-function Q_τ^h. In the stochastic case, Algorithm 2.6 can be used, which requires at each iteration the following number of function evaluations:

$$4\,|X|^2\,|U|$$

where the functions being evaluated are $\tilde{\rho}, \bar{f}, h$, and Q_τ^h. The cost in the stochastic case is thus larger by a factor $|X|$ than the cost in the deterministic case.

Table 2.4 collects the computational cost of policy evaluation for Q-functions, and compares it with the computational cost of Q-iteration (Section 2.3.1). A single Q-iteration requires $|X|\,|U|\,(2+|U|)$ function evaluations in the deterministic case (2.28), and $|X|^2\,|U|\,(2+|U|)$ function evaluations in the stochastic case (2.29). Whenever $|U|>2$, the cost of a single Q-iteration is therefore larger than the cost of a policy evaluation iteration.

TABLE 2.4
Computational cost of policy evaluation for Q-functions and of Q-iteration, measured by the number of function evaluations. The cost for a single iteration is shown.

	Deterministic case	Stochastic case
Policy evaluation	$4\,\|X\|\,\|U\|$	$4\,\|X\|^2\,\|U\|$
Q-iteration	$\|X\|\,\|U\|\,(2+\|U\|)$	$\|X\|^2\,\|U\|\,(2+\|U\|)$

Note also that evaluating the policy by directly solving the linear system given by the Bellman equation typically requires $O(|X|^3\,|U|^3)$ computation. This is an asymptotic measure of computational complexity (Knuth, 1976), and is no longer directly related to the number of function evaluations. By comparison, the complexity of the complete iterative policy evaluation algorithm is $O(L\,|X|\,|U|)$ in the deterministic case and $O(L\,|X|^2\,|U|)$ in the stochastic case, where L is the number of iterations.

Example 2.4 Model-based policy iteration for the cleaning robot. In this example, we apply a policy iteration algorithm to the cleaning-robot problem introduced in Examples 2.1 and 2.2. Recall that *every* single policy iteration requires a *complete execution* of policy evaluation for the current policy, together with a policy improvement. The (model-based) policy evaluation for Q-functions (2.38) is employed, starting from identically zero Q-functions. Each policy evaluation is run until the

Q-function fully converges. The same discount factor is used as for Q-iteration in Example 2.3, namely $\gamma = 0.5$.

Consider first the deterministic variant of Example 2.1, in which policy evaluation for Q-functions takes the form shown in Algorithm 2.5. Starting from a policy that always moves right ($h_0(x) = 1$ for all x), policy iteration produces the sequence of Q-functions and policies given in Table 2.5. In this table, the sequence of Q-functions produced by a given execution of policy evaluation is separated by dashed lines from the policy being evaluated (shown above the sequence of Q-functions) and from the improved policy (shown below the sequence). The policy iteration algorithm converges after 2 iterations. In fact, the policy is already optimal after the first policy improvement: $h_2 = h_1 = h^*$.

TABLE 2.5

Policy iteration results for the deterministic cleaning-robot problem. Q-values are rounded to 3 decimal places.

	$x=0$	$x=1$	$x=2$	$x=3$	$x=4$	$x=5$
h_0	$*$	1	1	1	1	$*$
Q_0	0; 0	0; 0	0; 0	0; 0	0; 0	0; 0
Q_1	0; 0	1; 0	0; 0	0; 0	0; 5	0; 0
Q_2	0; 0	1; 0	0; 0	0; 2.5	1.25; 5	0; 0
Q_3	0; 0	1; 0	0; 1.25	0.625; 2.5	1.25; 5	0; 0
Q_4	0; 0	1; 0.625	0.313; 1.25	0.625; 2.5	1.25; 5	0; 0
Q_5	0; 0	1; 0.625	0.313; 1.25	0.625; 2.5	1.25; 5	0; 0
h_1	$*$	-1	1	1	1	$*$
Q_0	0; 0	0; 0	0; 0	0; 0	0; 0	0; 0
Q_1	0; 0	1; 0	0.5; 0	0; 0	0; 5	0; 0
Q_2	0; 0	1; 0	0.5; 0	0; 2.5	1.25; 5	0; 0
Q_3	0; 0	1; 0	0.5; 1.25	0.625; 2.5	1.25; 5	0; 0
Q_4	0; 0	1; 0.625	0.5; 1.25	0.625; 2.5	1.25; 5	0; 0
Q_5	0; 0	1; 0.625	0.5; 1.25	0.625; 2.5	1.25; 5	0; 0
h_2	$*$	-1	1	1	1	$*$

Five iterations of the policy evaluation algorithm are required for the first policy, and the same number of iterations are required for the second policy. Recall that the computational cost of every iteration of the policy evaluation algorithm, measured by the number of function evaluations, is $4|X||U|$, leading to a total cost of $5 \cdot 4 \cdot |X||U|$ for each of the two policy evaluations. Assuming that the maximization over U in the policy improvement is solved by enumeration, the computational cost of every policy improvement is $|X||U|$. Each of the two policy iterations consists of a policy evaluation and a policy improvement, requiring:

$$5 \cdot 4 \cdot |X||U| + |X||U| = 21|X||U|$$

function evaluations, and thus the entire policy iteration algorithm has a cost of:

$$2 \cdot 21 \cdot |X|\,|U| = 2 \cdot 21 \cdot 6 \cdot 2 = 504$$

Compared to the cost 240 of Q-iteration in Example 2.3, policy iteration is in this case more computationally expensive. This is true even though the cost of any single policy evaluation, $5 \cdot 4 \cdot |X|\,|U| = 240$, is the same as the cost of Q-iteration. The latter fact is expected from the theory (Table 2.4), which indicated that policy evaluation for Q-functions and Q-iteration have similar costs when $|U| = 2$, as is the case here.

Consider now the stochastic case of Example 2.2. For this case, policy evaluation for Q-functions takes the form shown in Algorithm 2.6. Starting from the same policy as in the deterministic case (always going right), policy iteration produces the sequence of Q-functions and policies illustrated in Table 2.6 (not all Q-functions are shown). Although the Q-functions are different from those in the deterministic case, the same sequence of policies is produced.

TABLE 2.6
Policy iteration results for the stochastic cleaning-robot problem. Q-values are rounded to 3 decimal places.

	$x=0$	$x=1$	$x=2$	$x=3$	$x=4$	$x=5$
h_0	$*$	1	1	1	1	$*$
Q_0	0; 0	0; 0	0; 0	0; 0	0; 0	0; 0
Q_1	0; 0	0.800; 0.050	0.020; 0.001	0.001; 0	0.250; 4	0; 0
Q_2	0; 0	0.804; 0.054	0.022; 0.001	0.101; 1.600	1.190; 4.340	0; 0
Q_3	0; 0	0.804; 0.055	0.062; 0.641	0.485; 1.872	1.324; 4.372	0; 0
Q_4	0; 0	0.820; 0.311	0.219; 0.805	0.572; 1.909	1.342; 4.376	0; 0
...	...	...	...	...	...	...
Q_{24}	0; 0	0.852; 0.417	0.278; 0.839	0.589; 1.915	1.344; 4.376	0; 0
h_1	$*$	-1	1	1	1	$*$
Q_0	0; 0	0; 0	0; 0	0; 0	0; 0	0; 0
Q_1	0; 0	0.800; 0.110	0.320; 0.020	0.008; 0.001	0.250; 4	0; 0
Q_2	0; 0	0.861; 0.123	0.346; 0.023	0.109; 1.601	1.190; 4.340	0; 0
Q_3	0; 0	0.865; 0.124	0.388; 0.664	0.494; 1.873	1.325; 4.372	0; 0
Q_4	0; 0	0.881; 0.382	0.449; 0.821	0.578; 1.910	1.342; 4.376	0; 0
...	...	...	...	...	...	...
Q_{22}	0; 0	0.888; 0.458	0.467; 0.852	0.594; 1.915	1.344; 4.376	0; 0
h_2	$*$	-1	1	1	1	$*$

Twenty-four iterations of the policy evaluation algorithm are required to evaluate the first policy, and 22 iterations are required for the second. Recall that the cost of every iteration of the policy evaluation algorithm, measured by the number of function evaluations, is $4|X|^2|U|$ in the stochastic case, while the cost for policy improvement is the same as in the deterministic case: $|X|\,|U|$. So, the first policy

iteration requires $24 \cdot 4 \cdot |X|^2 |U| + |X||U|$ function evaluations, and the second requires $22 \cdot 4 \cdot |X|^2 |U| + |X||U|$ function evaluations. The total cost of policy iteration is obtained by adding these two costs:

$$46 \cdot 4 \cdot |X|^2 |U| + 2|X||U| = 46 \cdot 4 \cdot 6^2 \cdot 2 + 2 \cdot 6 \cdot 2 = 13272$$

Comparing this to the 6336 function evaluations necessary for Q-iteration in the stochastic problem (see Example 2.3), it appears that policy iteration is also more computationally expensive in the stochastic case. Moreover, policy iteration is more computationally costly in the stochastic case than in the deterministic case; in the latter case, policy iteration required only 504 function evaluations. □

2.4.2 Model-free policy iteration

After having discussed above model-based policy iteration, we now turn our attention to the class of class of RL, model-free policy iteration algorithms, and within this class, we focus on *SARSA*, an online algorithm proposed by Rummery and Niranjan (1994) as an alternative to the value-iteration based Q-learning. The name SARSA is obtained by joining together the initials of every element in the data tuples employed by the algorithm, namely: state, action, reward, (next) state, (next) action. Formally, such a tuple is denoted by $(x_k, u_k, r_{k+1}, x_{k+1}, u_{k+1})$. SARSA starts with an arbitrary initial Q-function Q_0 and updates it at each step using tuples of this form, as follows:

$$Q_{k+1}(x_k, u_k) = Q_k(x_k, u_k) + \alpha_k [r_{k+1} + \gamma Q_k(x_{k+1}, u_{k+1}) - Q_k(x_k, u_k)] \tag{2.40}$$

where $\alpha_k \in (0, 1]$ is the learning rate. The term between square brackets is the temporal difference, obtained as the difference between the updated estimate $r_{k+1} + \gamma Q_k(x_{k+1}, u_{k+1})$ of the Q-value for (x_k, u_k), and the current estimate $Q_k(x_k, u_k)$. This is not the same as the temporal difference used in Q-learning (2.30). While the Q-learning temporal difference includes the maximal Q-value in the next state, the SARSA temporal difference includes the Q-value of the action actually taken in this next state. This means that SARSA performs online, model-free policy evaluation of the policy that is currently being followed. In the deterministic case, the new estimate $r_{k+1} + \gamma Q_k(x_{k+1}, u_{k+1})$ of the Q-value for (x_k, u_k) is actually the policy evaluation mapping (2.35) applied to Q_k in the state-action pair (x_k, u_k). Here, $\rho(x_k, u_k)$ has been replaced by the observed reward r_{k+1}, and $f(x_k, u_k)$ by the observed next state x_{k+1}. In the stochastic case, these replacements provide a single sample of the random quantity whose expectation is found by the policy evaluation mapping (2.36).

Next, the policy employed by SARSA is considered. Unlike offline policy iteration, SARSA cannot afford to wait until the Q-function has (almost) converged before it improves the policy. This is because convergence may take a long time, during which the unimproved (and possibly bad) policy would be used. Instead, to select actions, SARSA combines a greedy policy in the current Q-function with exploration, using, e.g., ε-greedy (2.32) or Boltzmann (2.33) exploration. Because of the greedy component, SARSA implicitly performs a policy improvement at every time step, and is therefore a type of online policy iteration. Such a policy iteration

algorithm, which improves the policy after every sample, is sometimes called fully optimistic (Bertsekas and Tsitsiklis, 1996, Section 6.4).

Algorithm 2.3 presents SARSA with ε-greedy exploration. In this algorithm, because the update at step k involves the action u_{k+1}, this action has to be chosen prior to updating the Q-function.

ALGORITHM 2.7 SARSA with ε-greedy exploration.

Input: discount factor γ,
exploration schedule $\{\varepsilon_k\}_{k=0}^{\infty}$, learning rate schedule $\{\alpha_k\}_{k=0}^{\infty}$

initialize Q-function, e.g., $Q_0 \leftarrow 0$
measure initial state x_0
$u_0 \leftarrow \begin{cases} u \in \arg\max_{\bar{u}} Q_0(x_0,\bar{u}) & \text{with probability } 1-\varepsilon_0 \text{ (exploit)} \\ \text{a uniformly random action in } U & \text{with probability } \varepsilon_0 \text{ (explore)} \end{cases}$
for every time step $k = 0,1,2,\ldots$ **do**
    apply u_k, measure next state x_{k+1} and reward r_{k+1}
    $u_{k+1} \leftarrow \begin{cases} u \in \arg\max_{\bar{u}} Q_k(x_{k+1},\bar{u}) & \text{with probability } 1-\varepsilon_{k+1} \\ \text{a uniformly random action in } U & \text{with probability } \varepsilon_{k+1} \end{cases}$
    $Q_{k+1}(x_k,u_k) \leftarrow Q_k(x_k,u_k) + \alpha_k[r_{k+1} + \gamma Q_k(x_{k+1},u_{k+1}) - Q_k(x_k,u_k)]$
end for

In order to converge to the optimal Q-function Q^*, SARSA requires conditions similar to those of Q-learning, which demand exploration, and *additionally* that the exploratory policy being followed asymptotically becomes greedy (Singh et al., 2000). Such a policy can be obtained by using, e.g., ε-greedy (2.32) exploration with an exploration probability ε_k that asymptotically decreases to 0, or Boltzmann (2.33) exploration with an exploration temperature τ_k that asymptotically decreases to 0. Note that, as already explained in Section 2.3.2, the exploratory policy used by Q-learning can also be made greedy asymptotically, even though the convergence of Q-learning does not rely on this condition.

Algorithms like SARSA, which evaluate the policy they are currently using to control the process, are also called "on-policy" in the RL literature (Sutton and Barto, 1998). In contrast, algorithms like Q-learning, which act on the process using one policy and evaluate another policy, are called "off-policy." In Q-learning, the policy used to control the system typically includes exploration, whereas the algorithm implicitly evaluates a policy that is greedy in the current Q-function, since maximal Q-values are used in the Q-function updates (2.30).

2.5 Policy search

The previous two sections have introduced value iteration and policy iteration. In this section, we consider the third major class of DP/RL methods, namely *policy search*

algorithms. These algorithms use optimization techniques to directly search for an optimal policy, which maximizes the return from every initial state. The optimization criterion should therefore be a combination (e.g., average) of the returns from every initial state. In principle, any optimization technique can be used to search for an optimal policy. For a general problem, however, the optimization criterion may be a nondifferentiable function with multiple local optima. This means that global, gradient-free optimization techniques are more appropriate than local, gradient-based techniques. Particular examples of global, gradient-free techniques include genetic algorithms (Goldberg, 1989), tabu search (Glover and Laguna, 1997), pattern search (Torczon, 1997; Lewis and Torczon, 2000), cross-entropy optimization (Rubinstein and Kroese, 2004), etc.

Consider the return estimation procedure of a model-based policy search algorithm. The returns are infinite sums of discounted rewards (2.1), (2.15). However, in practice, the returns have to be estimated in a finite time. To this end, the infinite sum in the return can be approximated with a finite sum over the first K steps. To guarantee that the approximation obtained in this way is within a bound $\varepsilon_{\text{MC}} > 0$ of the infinite sum, K can be chosen with (e.g., Mannor et al., 2003):

$$K = \left\lceil \log_\gamma \frac{\varepsilon_{\text{MC}}(1-\gamma)}{\|\rho\|_\infty} \right\rceil \tag{2.41}$$

Note that, in the stochastic case, usually many sample trajectories need to be simulated to obtain an accurate estimate of the expected return.

Evaluating the optimization criterion of policy search requires the accurate estimation of returns from all the initial states. This procedure is likely to be computationally expensive, especially in the stochastic case. Since optimization algorithms typically require many evaluations of the criterion, policy search algorithms are therefore computationally expensive, usually more so than value iteration and policy iteration.

Computational cost of exhaustive policy search for finite MDPs

We next investigate the computational cost of a policy search algorithm for deterministic MDPs with a finite number of states and actions. Since the state and action spaces are finite and therefore discrete, any combinatorial optimization technique could be used to look for an optimal policy. However, for simplicity, we consider an algorithm that exhaustively searches the entire policy space.

In the deterministic case, a single trajectory consisting of K simulation steps suffices to estimate the return from a given initial state. The number of possible policies is $|U|^{|X|}$ and the return has to be evaluated for all the $|X|$ initial states. It follows that the total number of simulation steps that have to be performed to find an optimal policy is at most $K|U|^{|X|}|X|$. Since f, ρ, and h are each evaluated once at every simulation step, the computational cost, measured by the number of function evaluations, is:

$$3K|U|^{|X|}|X|$$

Compared to the cost $L|X||U|(2+|U|)$ of Q-iteration for deterministic systems (2.28), this implementation of policy search is, in most cases, clearly more costly.

In the stochastic case, when computing the expected return from a given initial state x_0, the exhaustive search algorithm considers all the possible realizations of a trajectory of length K. Starting from initial state x_0 and taking action $h(x_0)$, there are $|X|$ possible values of x_1, the state at step 1 of the trajectory. The algorithm considers all these possible values, together with their respective probabilities of being reached, namely $\bar{f}(x_0,h(x_0),x_1)$. Then, for each of these values of x_1, given the respective actions $h(x_1)$, there are again $|X|$ possible values of x_2, each reachable with a certain probability, and so on until K steps have been considered. With a recursive implementation, a total number of $|X|+|X|^2+\cdots+|X|^K$ steps have to be considered. Each such step requires 3 function evaluations, where the functions being evaluated are $\bar{f}$, $\tilde{\rho}$, and h. Moreover, $|U|^{|X|}$ policies have to be evaluated for $|X|$ initial states, so the total cost of exhaustive policy search in the stochastic case is:

$$3\left(\sum_{k=1}^{K}|X|^k\right)|U|^{|X|}|X| = 3\frac{|X|^{K+1}-|X|}{|X|-1}|U|^{|X|}|X|$$

Unsurprisingly, this cost grows roughly exponentially with K, rather than linearly as in the deterministic case, so exhaustive policy search is more computationally expensive in the stochastic case than in the deterministic case. In most problems, the cost of exhaustive policy search in the stochastic case is also greater than the cost $L|X|^2|U|(2+|U|)$ of Q-iteration (2.29).

Of course, much more efficient optimization techniques than exhaustive search are available, and the estimation of the expected returns can also be accelerated. For instance, after the return of a state has been estimated, this estimate can be reused at every occurrence of that state along subsequent trajectories, thereby reducing the computational cost. Nevertheless, the costs derived above can be seen as worst-case values that illustrate the inherently large complexity of policy search.

Example 2.5 Exhaustive policy search for the cleaning robot. Consider again the cleaning-robot problem introduced in Examples 2.1 and 2.2, and assume that the exhaustive policy search described above is applied. Take the approximation tolerance in the evaluation of the return to be $\varepsilon_{\text{MC}} = 0.01$, which is equal to the suboptimality bound ς_{QI} for Q-iteration in Example 2.3. Using ς_{QI}, maximum absolute reward $\|\rho\|_\infty = 5$, and discount factor $\gamma = 0.5$ in (2.41), a time horizon of $K = 10$ steps is obtained. Therefore, in the deterministic case, the computational cost of the algorithm, measured by the number of function evaluations, is:

$$3K|U|^{|X|}|X| = 3\cdot 10\cdot 2^6\cdot 6 = 11520$$

whereas in the stochastic case, it is:

$$3\frac{|X|^{K+1}-|X|}{|X|-1}|U|^{|X|}|X| = 3\cdot\frac{6^{11}-6}{6-1}\cdot 2^6\cdot 6 \approx 8\cdot 10^{10}$$

By observing that it is unnecessary to look for optimal actions and to evaluate returns in the terminal states, the cost can further be reduced to $3 \cdot 10 \cdot 2^4 \cdot 4 = 1920$ in the deterministic case, and to $3 \cdot \frac{6^{11}-6}{6-1} \cdot 2^4 \cdot 4 \approx 1 \cdot 10^{10}$ in the stochastic case. Additional reductions in cost can be obtained by stopping the simulation of trajectories as soon as they reach a terminal state, which will often happen in fewer than 10 steps.

Table 2.7 compares the computational cost of exhaustive policy search with the cost of Q-iteration from Example 2.3 and of policy iteration from Example 2.4. For the cleaning-robot problem, the exhaustive implementation of direct policy search is very likely to be more expensive than both Q-iteration and policy iteration. □

TABLE 2.7
Computational cost of exhaustive policy search for the cleaning robot, compared with the cost of Q-iteration and of policy iteration. The cost is measured by the number of function evaluations.

	Deterministic case	Stochastic case
Exhaustive policy search	11520	$8 \cdot 10^{10}$
Exhaustive policy search, no optimization in terminal states	1920	$1 \cdot 10^{10}$
Q-iteration	240	6336
Policy iteration	504	13272

2.6 Summary and discussion

In this chapter, deterministic and stochastic MDPs have been introduced, and their optimal solution has been characterized. Three classes of DP and RL algorithms have been described: value iteration, policy iteration, and direct search for control policies. This presentation provides the necessary background for the remainder of this book, but is by no means exhaustive. For the reader interested in more details about classical DP and RL, we recommend the textbook of Bertsekas (2007) on DP, and that of Sutton and Barto (1998) on RL.

A central challenge in the DP and RL fields is that, in their original form, DP and RL algorithms cannot be implemented for general problems. They can only be implemented when the state and action spaces consist of a finite number of discrete elements, because (among other reasons) they require the exact representation of value functions or policies, which is generally impossible for state spaces with an infinite number of elements. In the case of Q-functions, an infinite number of actions also prohibits an exact representation. For instance, most problems in automatic control have continuous states and actions, which can take infinitely many distinct values. Even when the states and actions take finitely many values, the cost of representing value functions and policies grows exponentially with the number of state variables (and action variables, for Q-functions). This problem is called the curse of dimen-

sionality, and makes the classical DP and RL algorithms impractical when there are many state and action variables.

To cope with these problems, versions of the classical algorithms that *approximately* represent value functions and/or policies must be used. Such algorithms for approximate DP and RL form the subject of the remainder of this book.

In practice, it is essential to provide more comprehensive performance guarantees than simply the asymptotical maximization of the return. For instance, online RL algorithms should guarantee an increase in performance over time. Note that the performance cannot increase monotonically, since exploration is necessary, which can cause temporary degradations of the performance. In order to use DP and RL algorithms in industrial applications of automatic control, it should be guaranteed that they can never destabilize the process. For instance, Perkins and Barto (2002); Balakrishnan et al. (2008) discuss DP and RL approaches that guarantee stability using the Lyapunov framework (Khalil, 2002, Chapters 4 and 14).

Designing a good reward function is an important and nontrivial step of applying DP and RL. Classical texts on RL recommend making the reward function as simple as possible; it should only reward the achievement of the final goal (Sutton and Barto, 1998). However, a simple reward function often makes online RL slow, and including more information may be required for successful learning. Moreover, other high-level requirements on the behavior of the controller often have to be considered in addition to achieving the final goal. For instance, in automatic control the controlled state trajectories often have to satisfy requirements on overshoot and the rate of convergence to an equilibrium, etc. Translating such requirements into the language of rewards can be very challenging.

DP and RL algorithms can also be greatly helped by domain knowledge. Although RL is usually envisioned as purely model-free, it can be very beneficial to use prior knowledge about the problem, if such knowledge is available. If a partial model is available, a DP algorithm can be run with this partial model, in order to obtain a rough initial solution for the RL algorithm. Prior knowledge about the policy can also be used to restrict the class of policies considered by (model-based or model-free) policy iteration and policy search. A good way to provide domain knowledge to any DP or RL algorithm is to encode it in the reward function (Dorigo and Colombetti, 1994; Matarić, 1997; Randløv and Alstrøm, 1998; Ng et al., 1999). This procedure is related to the problem of reward function design discussed above. For instance, prior knowledge about promising control actions can be exploited by associating these actions with high rewards. Encoding prior knowledge in the reward function should be done with care, because doing so incorrectly can lead to unexpected and possibly undesirable behavior.

Other work aiming to expand the boundaries of DP and RL includes: problems in which the state is not fully measurable, called partially observable MDPs (Lovejoy, 1991; Kaelbling et al., 1998; Singh et al., 2004; Pineau et al., 2006; Porta et al., 2006), exploiting modular and hierarchical task decompositions (Dietterich, 2000; Hengst, 2002; Russell and Zimdars, 2003; Barto and Mahadevan, 2003; Ghavamzadeh and Mahadevan, 2007), and applying DP and RL to distributed, multi-agent problems (Panait and Luke, 2005; Shoham et al., 2007; Buşoniu et al., 2008a).