Chapter 13
Solving Markov Decision Processes via Simulation

Abhijit Gosavi

Abstract This chapter presents an overview of simulation-based techniques useful for solving Markov decision processes (MDPs). MDPs model problems of sequential decision-making under uncertainty, in which decisions made in each state collectively affect the trajectory of the states visited by the system over a time horizon of interest. Traditionally, MDPs have been solved via dynamic programming (DP), which requires the transition probability model that is difficult to derive in many realistic settings. The use of simulation for solving MDPs allows us to bypass the transition probability model and solve large-scale MDPs considered intractable to solve by traditional DP. The simulation-based methodology for solving MDPs, which like DP is also rooted in the Bellman equations, goes by names such as reinforcement learning, neuro-DP, and approximate or adaptive DP. We begin with a description of algorithms for infinite-horizon discounted reward MDPs, followed by the same for infinite-horizon average reward MDPs. Then we present a discussion on finite-horizon MDPs. For each problem considered, we present a step-by-step description of a selected group of algorithms. In making this selection, we have attempted to blend the old and the classical with more recent developments. Finally, after touching upon extensions and convergence theory, we conclude with a brief summary of some applications and directions for future research.

13.1 Introduction

Reinforcement learning (RL) and approximate dynamic programming (ADP), also called adaptive DP by some authors, are closely related research fields that have been successfully applied to many practical problems addressing sequential decision-making under uncertainty. The central ideas in these fields are closely tied to solving control problems in discrete-event dynamic systems where the underlying problem revolves around finding the optimal control (action) in each state visited by the system. These problems were initially studied by Richard Bellman [10], who also formulated what is now known as the Bellman equation. Much of the

A. Gosavi (✉)
Missouri University of Science and Technology, Rolla, MO, USA
e-mail: gosavia@mst.edu

M.C. Fu (ed.), *Handbook of Simulation Optimization*, International Series in Operations Research & Management Science 216, DOI 10.1007/978-1-4939-1384-8_13

methodology in RL and ADP is tied to solving some version of this equation. These problems are often called Markov decision processes/problems (MDPs). The solution methods invented by Bellman [11] and Howard [45] are called value iteration and policy iteration, respectively; collectively, they are called dynamic programming (DP) methods.

When the number of states and/or the number of actions is very large, DP suffers from the curse of dimensionality, i.e., it becomes difficult to apply DP in an exact sense. This is because DP requires the transition probability matrices, which can become huge for large-scale problems, too large to be stored or manipulated. It is on these problems that simulation can play a major role in solution methods. The key role that simulation plays is in avoiding the transition probabilities. It is well-known that for complex systems, producing simulators is significantly easier than developing exact mathematical models, i.e., the transition probabilities.

RL algorithms can run in simulators and have the potential to break the curse of dimensionality to produce optimal or near-optimal solutions. In particular, we will focus on methods in which the number of actions is finite and relatively small, e.g., a dozen. The algorithms will be based on the Bellman equation but will *not* need the transition probability model.

A significant body of literature in the area of RL and ADP appears to be divided into two branches. The first seeks to use algorithms on a real-time basis within the system, i.e., while the system is running (on-line). This branch is more closely associated with the name RL and is popular within the computer science and artificial intelligence (robotics) community. The other branch works primarily in an off-line sense and seeks to solve large-scale MDPs where the transition probabilities can be estimated but a naïve application of DP does not work. This branch is more closely associated with the name ADP and finds applications in electrical, industrial, and mechanical engineering. Another somewhat synthetic approach to study the differences between these two branches is to consider the function used: RL algorithms for the most part work with the Q-function defined by Eq. (13.1) or (13.2), whereas most ADP algorithms work with the value function of DP (e.g., Definition 13.1).

In the simulation community, one is usually interested in the algorithms belonging to the RL variety, because the Q-function can help avoid the transition probabilities. Note, however, that there are important exceptions to this, e.g., evolutionary policy iteration algorithm [25] and many model-building algorithms (see e.g., [84]). Regardless of whether the Q-function or the value function is used, in the simulation community, the interest lies in problems where the transition probability model is not easy to generate. As such, in this chapter, we limit ourselves to discussing algorithms that can bypass the transition probability model. As stated above, the algorithms we discuss will be limited to finite action spaces. It is also important to note that we will employ the RL algorithm in an off-line sense within the simulator. Hence, one assumes that the distributions of the random variables that form inputs to the system are available.

Some of the earlier books that cover the topics of RL and simulation-based algorithms include [12, 24, 30, 79, 82]. Numerous books have also appeared that primarily focus on the ADP aspects, and some of these include [66, 74]. The latest edition of the second volume of [15] has an entire chapter dedicated to ADP (and also to RL). See also [21] for policy search via simulation optimization. RL has also been surveyed in journal articles [35, 49, 52].

In writing this chapter, we have had to make some conscious choices regarding the selection of algorithms from the vast array now available in the literature. We have sought to blend our discussion of classical algorithms based on Q-functions and value iteration with that of the more recent developments in the area of policy iteration. We study the three main classes of objective functions commonly studied in this area: the infinite-horizon discounted reward, the infinite-horizon average reward, and the finite-horizon total reward. A highlight of our presentation is a step-by-step description of combining function approximators with algorithms of the Q-Learning type—an important topic from the perspective of attacking large-scale problems within simulators. Another highlight is the discussion on the finite horizon and average reward problems that are of significant interest in the operations research community.

The rest of this chapter is organized as follows. In Sect. 13.2, we present some background material including notation. Algorithms related to discounted reward, average reward, and total reward (finite horizon) are presented in Sects. 13.3, 13.4, and 13.5, respectively. Some simple numerical results are presented in Sect. 13.6. Sects. 13.7 and 13.8 present short discussions on extensions and convergence theory, respectively. Sect. 13.9 concludes this chapter with a discussion of applications and topics for future research and open problems.

13.2 Background

In this section, we present some background for simulation-based optimization of MDPs. In an MDP, the system transitions from one state to another in a dynamic fashion. The decision-maker is required to select an action from a set of actions (where the set contains at least two actions) in a subset of states. Those states in which actions have to be chosen are called decision-making states. Henceforth, by states, we will mean decision-making states, since for analyzing MDPs, it is sufficient to observe the transitions that occur from one decision-making state to another. Thus, as a result of selecting an action, the system transitions to another state (which can be the same state)—usually in a probabilistic manner. (In this chapter, we will confine our discussion to those MDPs in which the transitions are probabilistic, since they are more interesting in a simulation-based context.) The probability of transitioning (moving/jumping) from one state to another under the influence of an action is called the one-step transition probability, or simply the transition probability. The transition probabilities are collectively referred to as the transition probability model.

During a transition from one state to another, the system receives a (one-step) *immediate reward*, which is either zero, positive, or negative. Negative rewards can be viewed as costs. The decision-maker's goal is to choose actions in each state in a fashion that optimizes the value of some performance metric of interest. The actions chosen in the different states visited by the system are stored in a *policy*, and the decision-maker is interested in the policy that optimizes the performance metric of interest, so the solution to an MDP is an optimal policy. As stated above, our interest in this chapter lies in simulation-based methods that can help determine the optimal policy without generating the transition probabilities, and thereby solve complex large-scale MDPs with finite action sets, considered intractable via traditional DP methods.

The performance metric's value generally depends on the actions chosen in each state, the immediate rewards, the time horizon of interest and whether the time value of money is considered. When the time value of money is taken into account in the calculations, we have a discounted performance metric, while when it is ignored, we have an undiscounted performance metric. The time value of money will be discussed later in more detail. We now present some of the fundamental notation needed in this chapter.

13.2.1 Notation and Assumptions

A deterministic policy is one in which the decision-maker selects a fixed (deterministic) action in each decision-making state, in contrast to a *stochastic* policy in which in each decision-making state, each action is chosen with some fixed probability (such that the probability of selecting all actions sums to one for every state). A *stationary* policy is one in which the action selected in a state does not change with time. In general, we will be interested in finding stationary deterministic policies, so henceforth when we refer to a policy, we mean a stationary deterministic policy unless explicitly specified otherwise.

Let $\mathscr{S}$ denote the finite set of states visited by the system, $\mathscr{A}(i)$ the finite set of actions permitted in state i, and $\mu(i)$ the action chosen in state i when policy μ is pursued. We define $\mathscr{A} \equiv \cup_{i \in \mathscr{S}} \mathscr{A}(i)$. Further let $r(.,.,.) : \mathscr{S} \times \mathscr{A} \times \mathscr{S} \rightarrow \mathbb{R}$ denote the immediate reward and $p(.,.,.) : \mathscr{S} \times \mathscr{A} \times \mathscr{S} \rightarrow [0,1]$ denote the associated transition probability. Then the *expected* immediate reward earned in state i when action a is chosen in it can be expressed as:

$$\bar{r}(i,a) = \sum_{j=1}^{|\mathscr{S}|} p(i,a,j) r(i,a,j).$$

We will make the following two assumptions about the problems considered in this chapter.

- Both the state-space, $\mathscr{S}$, and the action-space, $\mathscr{A}$, are finite.
- The Markov chain of every policy is regular, i.e., the transition probability of the Markov chain can be raised to some finite power such that all elements in the resulting matrix become strictly positive [42].

13.2.2 Performance Metrics

We now define the three metrics of interest to us. The first metric is the expected total *discounted* reward over an infinitely long time horizon. The discount factor is a well-known mechanism used in MDP theory to capture the time value of money. It is to be interpreted as follows. If one earns z dollars at a time τ time periods after the current time, then the current value of those z dollars will be

$$z\left(\frac{1}{1+\kappa}\right)^{\tau},$$

where κ is the rate of interest. We denote $1/(1+\kappa)$ by γ. For the MDP, the assumption is that transition from one state to another requires one time period, i.e., $\tau = 1$. Hence, the discount factor that will be used after one state transition will be γ.

Definition 13.1. The expected total discounted reward of a policy μ starting at state i in an MDP over an infinitely long time horizon is:

$$V_{\mu}(i) \equiv \liminf_{k\to\infty} \mathsf{E}_{\mu}\left[\sum_{s=1}^{k} \gamma^{s-1} r(x_s, \mu(x_s), x_{s+1}) \middle| x_1 = i\right],$$

where γ is the (one-step) discount factor, E_{μ} denotes the expectation operator over the trajectory induced by policy μ, and x_s denotes the state occupied by the system before the sth transition (jump) in the trajectory occurs.

In a discounted reward MDP, the goal is to maximize this performance metric for all values of $i \in \mathscr{S}$, i.e., it appears that there are multiple objective functions, but fortunately, it can be shown that when all policies have regular Markov chains, there exists a stationary, deterministic optimal policy that maximizes the value of the above metric for all starting states simultaneously [15].

We now define the other popular performance metric for infinite time horizons: the expected reward per transition over an infinitely long time horizon, commonly known as the "average reward." In this metric, the time value of money is ignored.

Definition 13.2. The average (expected) reward of a policy μ per transition in an MDP, starting at state i over an infinitely long time horizon, is defined as:

$$\rho_\mu(i) \equiv \liminf_{k\to\infty} \frac{1}{k} \mathsf{E}_\mu \left[\sum_{s=1}^{k} r(x_s, \mu(x_s), x_{s+1}) | x_1 = i \right].$$

If the Markov chain of the policy is regular, the average reward does not depend on the starting state, i.e., $\rho_\mu(i) = \rho_\mu$ for all $i \in \mathscr{S}$. The goal thus becomes to maximize the average reward.

The third performance metric of interest here is that of the expected *total* reward over a *finite* time (i.e., number of stages) horizon, which is sought to be maximized. It depends on the starting state in the problem, which is assumed to be known, and is defined as follows:

Definition 13.3. The expected total reward over a finite horizon of T time periods for a policy μ when the starting state is i is defined as:

$$\phi_\mu(i) \equiv \mathsf{E}_\mu \left[\sum_{s=1}^{T} r(x_s, \mu(x_s), x_{s+1}) \middle| x_1 = i \right],$$

where s is generally known as the stage, and the starting state i is fixed for the problem.

13.2.3 Bellman Equations

The theory of DP is rooted in the famous Bellman equations, which were originally presented in the form of the value functions (see [11]). For the simulation-based context, it is the Q-function that is more useful, and hence we present the Bellman equations in the Q-format. We present the first equation for the discounted reward MDP.

Theorem 13.1. *For a discounted reward MDP, there exists a function $Q : \mathscr{S} \times \mathscr{A} \to \mathbb{R}$ such that the following set of equations have a unique solution*

$$Q(i,a) = \sum_{j=1}^{|\mathscr{S}|} p(i,a,j) \left[r(i,a,j) + \gamma \max_{b \in \mathscr{A}(j)} Q(j,b) \right] \quad \forall (i,a) \tag{13.1}$$

and the policy μ, defined by $\mu(i) \in \arg\max_{a\in\mathscr{A}(i)} Q(i,a)$ for all $i \in \mathscr{S}$, is an optimal policy for the MDP.

The associated result for the average reward MDP is:

Theorem 13.2. *For an average reward MDP, there exists a function $Q : \mathscr{S} \times \mathscr{A} \to \mathbb{R}$ and a scalar $\rho^* \in \mathbb{R}$ such that a solution exists for the following equations:*

$$Q(i,a) = \sum_{j \in \mathscr{S}} p(i,a,j) \left[r(i,a,j) + \max_{b \in \mathscr{A}(j)} Q(j,b) - \rho^* \right] \quad \forall (i,a). \tag{13.2}$$

Further ρ^* equals the optimal average reward of the MDP, and the policy μ, defined by $\mu(i) \in \arg\max_{a \in \mathscr{A}(i)} Q(i,a)$ for all $i \in \mathscr{S}$, is an optimal policy for the MDP.

We note that $Q(\cdot,\cdot)$ is commonly called the Q-factor, Q-value, or the state-action value in the literature. Finding the optimal values of these quantities holds the key to solving the MDP.

For the finite-horizon problem, we need a somewhat enhanced style of notation to account for the stage. In a finite-horizon problem, the state-action pair, (i,a), will be replaced by the state-stage-action triple, (i,s,a), where s is the stage index and takes values in the set $\mathscr{T} = \{1,2,\ldots,T\}$. The notation for the immediate reward, the transition probability, and the Q-function will need to account for this triple. In such problems, we will assume that there is no decision-making to be performed in stage $T+1$ and that the Q-value in that stage, regardless of the state or action, will be zero. *The starting state, i.e., when $s = 1$, will be assumed to be known with certainty in finite-horizon problems.* The following is the Bellman equation for a finite-horizon undiscounted problem.

Theorem 13.3. *There exists a function $Q : \mathscr{S} \times \mathscr{T} \times \mathscr{A} \to \mathbb{R}$ such that a solution exists for the following equations: For all $i \in \mathscr{S}$, all $s \in \mathscr{T}$, and all $a \in \mathscr{A}(i,s)$:*

$$Q(i,s,a) = \sum_{j \in \mathscr{S}} p(i,s,a,j,s+1) \left[r(i,s,a,j,s+1) + \max_{b \in \mathscr{A}(j,s+1)} Q(j,s+1,b) \right], \tag{13.3}$$

where $Q(j,T+1,b) = 0$ for all $j \in \mathscr{S}$ and $b \in \mathscr{A}(j,T+1)$. The policy μ, defined by $\mu(i,s) \in \arg\max_{a \in \mathscr{A}(i,s)} Q(i,s,a)$ for all $i \in \mathscr{S}$ and all $s \in \mathscr{T}$, is an optimal policy for the MDP.

13.3 Discounted Reward MDPs

In this section, we present some of the key (simulation-based) RL algorithms for solving discounted reward MDPs. We begin with the classical Q-Learning algorithm, which is based on value iteration, along with a discussion on how it can be combined with function approximation. This is followed by a popular, but heuristic, algorithm called SARSA(λ), based on the notion of the temporal difference learning algorithm TD(λ). Thereafter, we present two algorithms based on policy iteration: one is based on the classical modified policy iteration approach and the other based on the actor-critic algorithm. We conclude this section with a relatively new algorithm that combines ideas of genetic algorithms within policy iteration.

13.3.1 Q-Learning

The idea of Q-Learning [88] is based on iteratively solving the Bellman equation presented in Eq. (13.1) while avoiding the transition probabilities. We will first present the intuition underlying the derivation of this algorithm and then present the steps more formally.

Clearly, Eq. (13.1) contains the transition probabilities that we seek to avoid in simulators. The main idea underlying Q-Learning is to use a Robbins–Monro stochastic approximation algorithm [70] to estimate the mean via samples without *directly* summing the samples. If x^k denotes the kth sample and $\hat{x}^k$ denotes the estimate after k samples, then the update is as follows:

$$\hat{x}^{k+1} \leftarrow (1-\alpha^k)\hat{x}^k + \alpha^k x^k,$$

where α^k denotes the step size in the kth iteration, which is a small positive scalar less than 1 that must satisfy the following conditions:

$$\sum_{k=1}^{\infty} \alpha^k = \infty; \quad \sum_{k=1}^{\infty} \left(\alpha^k\right)^2 < \infty.$$

Examples of step-size rules that satisfy the conditions above include

$$\alpha^k = \frac{A}{B+k}(B \geq A \geq 0), \quad \alpha^k = \frac{\log k}{k}(k \geq 2).$$

In practice, finding the right step-size often requires some experimentation; see Chaps. 6 and 7 for further discussion.

To apply the Robbins–Monro update for estimating the Q-factors in Eq. (13.1), it is necessary to express the right hand side of the Bellman equation as an expectation as follows:

$$Q(i,a) = \mathsf{E}_{i,a}\left[r(i,a,j) + \gamma \max_{b \in \mathscr{A}(j)} Q(j,b)\right] \quad \forall (i,a),$$

where the expectation operator $\mathsf{E}_{i,a}[\cdot]$ is over the random state transitions that can occur from state i under the influence of action a. Then the Q-Learning algorithm is as follows:

$$Q^{k+1}(i,a) = (1-\alpha^k)Q^k(i,a) + \alpha^k\left[r(i,a,j) + \gamma \max_{b \in \mathscr{A}(j)} Q^k(j,b)\right], \tag{13.4}$$

where the terms in the square brackets represent the sample. Note that (13.4) does not contain the transition probabilities. In a simulator, where every action is selected in each state with the same probability, it can be shown that as k tends to infinity, the algorithm converges to the unique solution of Eq. (13.1), i.e., the Bellman equation, thereby solving the problem.

The above presentation ignores subtleties that we now note. First, there are multiple Q-factors being estimated simultaneously here. Furthermore, in any given update, the terms within the square brackets potentially contain Q-factors of *other* state-action pairs. In general, the Q-factors of the other state-action pairs will not have been updated with the same frequency, which is true of synchronous updating used in DP. Updating of this nature is called asynchronous updating, and the differences in the frequencies arise from the fact that the trajectory pursued in the simulator is random. In the simulation-based setting, the haphazard order of updating can rarely be avoided, but in practice, it is necessary to maintain rough equality in the frequencies with which state-action pairs are visited. The convergence analysis of the algorithm must take all of this into account.

Not surprisingly, all convergence proofs require that all state-action pairs be visited infinitely often in the limit. Also, since we are dealing with simulation noise, all convergence proofs ensure convergence only with probability (w.p.) 1. Fortunately, a number of proofs for convergence have been worked out under some rather mild conditions [12, 16] that can be ensured within simulators for the case in which the Q-factor for each state-action pair is stored separately—a scenario generally referred to as the "look-up table" case. When the state-action space is very large, e.g., of the order of hundreds of thousands or millions, it is impossible to store each Q-factor separately, and one must then use a function-approximation scheme. When function approximation schemes are used, showing convergence becomes significantly more challenging, although some progress has been made even in this direction recently (see the chapter on ADP in [15]). The general Q-Learning algorithm is presented below.

Basic Q-Learning Algorithm

Step 0. Input $k_{\max}$= total number of iterations. Initialize iteration count $k = 0$ and all Q-values to 0, i.e., for all (l,u), where $l \in \mathscr{S}$ and $u \in \mathscr{A}(l)$, set $Q^k(l,u) = 0$. Start system simulation at any arbitrary state.

Step 1. For current state i, select action a w.p. $1/|\mathscr{A}(i)|$, and simulate to reach next state j, receiving reward $r(i,a,j)$.

Step 2. Update the Q-value of (i,a) via Eq. (13.4). Increment k by 1.
If $k < k_{max}$, then set $i \leftarrow j$ and return to Step 1; otherwise, go to Step 3.

Step 3. For each $l \in \mathscr{S}$, select $d(l) \in \arg\max_{b \in \mathscr{A}(l)} Q^k(l,b)$.
The policy (solution) generated by the algorithm is d.
Stop.

Alternative action–selection strategies for Step 1 will be discussed later.

13.3.2 Q-Learning with Function Approximation

As stated above, function approximation is necessary when the state-action space is large. The function approximation approach hinges on using what is known as a basis-function representation of the Q-function, along with the steepest-descent rule over the Bellman error, which is essentially the sum of squared errors in regression theory applied to the Bellman equation. We now present the key ideas. To simply exposition, we drop the superscript k in this subsection.

Basis Functions

The Q-factor can be represented via basis-functions and their weights using a *linear architecture* as follows:

$$Q_w(i,a) = \sum_{m=1}^{n} w(m,a)\phi(m,a), \tag{13.5}$$

where $\{\phi(\cdot,\cdot)\}$ denote the basis functions and $\{w(\cdot,\cdot)\}$ the weight functions, and n should be much smaller than the size of the state space. The actual set of basis functions is problem dependent. We now provide a simple example where the state has a single dimension.

Example with a Linear Architecture

Consider an MDP with a single-dimensional state, i, and two actions. Let $n = 2$, where for both values of a, the analyst chooses to use the following architecture:

$$\phi(1,a) = 1; \quad \phi(2,a) = i.$$

Thus, $Q_w(i,1) = w(1,1) + w(2,1)i; \quad Q_w(i,2) = w(1,2) + w(2,2)i.$

Bellman Error

The following model is used to represent the Q-factor:

$$Q(i,a) = \sum_{j=1}^{|\mathscr{S}|} p(i,a,j)\left[r(i,a,j) + \gamma \max_{b\in\mathscr{A}(j)} Q_w(j,b)\right] \quad \forall (i,a),$$

where $Q_w(\cdot,\cdot)$ denotes an estimate of the true Q-factor. The Bellman error, BE, is then defined as follows:

$$BE \equiv \frac{1}{2} \sum_{i \in \mathscr{S}, a \in \mathscr{A}(i)} [Q(i,a) - Q_w(i,a)]^2 .$$

Thus, the Bellman error denotes one-half times the sum of the squared difference between the hypothesized model of the Q-factor and that given by the function approximator. This is the classical sum of squared errors of regression and the coefficient of $1/2$ is popular in the machine learning community because it simplifies the resulting algorithm. The following expression, the detailed derivation of which can be found in [12], is commonly used to minimize the Bellman error (the first use was seen in [90] and now extensively used in the RL literature):

$$\frac{\partial BE}{\partial w(m,a)} = -\frac{\partial Q_w(i,a)}{\partial w(m,a)} \left[r(i,a,j) + \gamma \max_{b \in \mathscr{A}(j)} Q_w(j,b) - Q_w(i,a) \right] \text{ for all } (m,a). \tag{13.6}$$

The above is generally combined with the following steepest-descent algorithm:

$$w(m,a) \leftarrow w(m,a) - \alpha \frac{\partial BE}{\partial w(m,a)} \text{ for all } (m,a), \tag{13.7}$$

where α is the step size. For using the above, we must determine the expressions for $\frac{\partial Q_w(i,a)}{\partial w(m,a)}$, which can be done easily from the architecture defined in Eq. (13.5). In general, it is easy to see from Eq. (13.5) that for linear architectures:

$$\frac{\partial Q_w(i,a)}{\partial w(m,a)} = \phi(m,a).$$

Then, for the example MDP considered above, we obtain the following definitions for the partial derivatives:

$$\frac{\partial Q_w(i,a)}{\partial w(1,a)} = 1; \quad \frac{\partial Q_w(i,a)}{\partial w(2,a)} = i. \tag{13.8}$$

We now describe the algorithm using the above example as a specific case. We note the basis-function representation is conceptual; we do not store $Q_w(i,1)$ or $Q_w(i,2)$ in the computer's memory. Rather, only the following four scalars are stored: $w(1,1)$, $w(2,1)$, $w(1,2)$, and $w(2,2)$.

Note that in the above, α uses the step-size rules discussed previously. Furthermore, note that the rule to update weights (i.e., Eqs. (13.9) and (13.10)) is derived from Eqs. (13.6) to (13.8). The above is a popular algorithm, and can be extended easily (using the above equations) to more complex basis functions that can express a non-linear architecture, e.g.,

$$Q_w(i,1) = w(1,1) + w(2,1)i + w(3,1)(i)^2; Q_w(i,2) = w(1,2) + w(2,2)i + w(3,2)(i)^2.$$

Algorithm for the 2-Action Linear Architecture Example

Step 0. Input $k_{\max}$= total number of iterations. Initialize iteration count $k = 0$ and the weights for action 1, i.e., $w(1,1)$ and $w(2,1)$, to small random numbers, and set the corresponding weights for action 2 to the same values. Start system simulation at any arbitrary state.

Step 1. For current state i, select action a w.p. $1/|\mathcal{A}(i)|$, and simulate to reach next state j, receiving reward $r(i,a,j)$.

Step 2. Compute the following:

$$Q_{\text{old}} \leftarrow w(1,a) + w(2,a)i.$$

$$\text{Then, set} \qquad Q_{\text{next}} \leftarrow \max\{Q_{\text{next}}(1), Q_{\text{next}}(2)\}, \text{ where}$$

$$Q_{\text{next}}(1) = w(1,1) + w(2,1)j; \quad Q_{\text{next}}(2) = w(1,2) + w(2,2)j.$$

Then, update the two weights as follows:

$$w(1,a) \leftarrow w(1,a) + \alpha\left(r(i,a,j) + \gamma Q_{\text{next}} - Q_{\text{old}}\right)1; \tag{13.9}$$

$$w(2,a) \leftarrow w(2,a) + \alpha\left(r(i,a,j) + \gamma Q_{\text{next}} - Q_{\text{old}}\right)i. \tag{13.10}$$

Increment k by 1. If $k < k_{\max}$, then set $i \leftarrow j$, go to Step 1; otherwise, go to Step 3.

Step 3. The policy learned, μ, is virtually stored within the weights. To determine the action prescribed in a state i for any $i \in \mathcal{S}$, compute the following:

$$\mu(i) \in \arg\max_{a \in \mathcal{A}(i)} [w(1,a) + w(2,a)i].$$

Before concluding this subsection, we note that the steepest-descent technique, coupled with Bellman error (discussed above), closely resembles the approach adopted in the RL community that uses neurons (also called linear neural networks), via the Widrow–Hoff (adenaline) rule [58, 93], for approximating the Q-function.

13.3.3 SARSA(λ)

We now present a heuristic algorithm which is known to have strong empirical performance. It is based on the concept of TD(λ), popular in RL [80], and the SARSA algorithm [71]. We first explain the concept of TD(λ) and follow that by SARSA.

The notion of TD(λ) is based on the idea that the immediate reward (also called "feedback" in the machine learning community), computed within the simulator after a state transition occurs, can be used to update *all* state-action pairs in

the system. Note that in Q-Learning, only the Q-factor of the most recently visited state-action pair is updated when the immediate reward is obtained from a state transition. This is called a single-step update. In contrast, in TD(λ), the impact of the update for any state-action pair is proportional to how recently it was visited, something that is measured with the "recency traces."

We present this idea somewhat more formally now. If $W^k(i,a)$ denotes the iterate in the kth iteration for the state-action pair, (i,a), the Robbins–Monro update is:

$$W^{k+1}(i,a) \leftarrow W^k(i,a) + \alpha^k \left[\text{feedback}\right], \tag{13.11}$$

where the feedback really depends on the objective function at hand. In the TD(λ) update, where $\lambda \in [0,1]$, one simulates an infinitely long trajectory, and the feedback takes on the following form:

$$\text{feedback } = R_k + \lambda R_{k+1} + \lambda^2 R_{k+2} + \cdots, \tag{13.12}$$

where R_k is a term that depends on the algorithm and the iteration index k. Note that in Q-Learning, $\lambda = 0$, and $R_k = r(i,a,j) + \gamma \max_{b \in \mathscr{A}(j)} Q(j,b) - Q(i,a)$. When $\lambda > 0$ (but less than 1), we have a multi-step update, also called a TD(λ) update. In such an update, *all* states in the system are updated after every state transition in the simulator, and the feedback from multiple state transitions is used in updating all (or as we will see later, all but one) states. In the algorithm that follows, we will use such an updating mechanism.

Within the simulator, the SARSA algorithm uses a policy which is initially fully stochastic but gradually becomes greedy with respect to the Q-factors. This is also called an ε-greedy form of action selection (or policy). Furthermore, in SARSA, the feedback contains the Q-factor of the next state-action pair visited in the simulator; this is different than in Q-Learning, where the feedback contains the *maximum* Q-factor of the next state.

Formally, an ε-greedy form of action selection can be described as follows. In a state i, one selects the greedy action

$$\arg\max_{u \in \mathscr{A}(i)} Q(i,u)$$

w.p. P^k and any one of the remaining actions w.p.

$$\frac{1 - P^k}{|\mathscr{A}(i)| - 1}.$$

Furthermore, the probability of selecting non-greedy actions is gradually diminished to zero. A potential rule for the probability that can achieve this is: $P^k = 1 - B/k$, where for instance $B = 0.5$.

SARSA (λ) Algorithm

Step 0. Input $k_{\max}$= total number of iterations. Initialize iteration count $k = 0$ and all the Q-values, $Q(l,u)$ for all $l \in \mathscr{S}$ and $u \in \mathscr{A}(l)$, to arbitrary values. Set the recency trace, $e(l,u)$, to 0 for all $l \in \mathscr{S}$ and $u \in \mathscr{A}(l)$. Start system simulation at any arbitrary state.

Step 1. For current state i, select action $a \in \mathscr{A}(i)$ via an ε-greedy form of action selection. Simulate action a, and let the next state be j, with $r(i,a,j)$ being the immediate reward. Select an action $b \in \mathscr{A}(j)$ via the ε-greedy form of action selection. Then, compute the feedback, δ, and update $e(i,a)$, the recency trace for (i,a), as follows:

$$\delta \leftarrow r(i,a,j) + \gamma Q(j,b) - Q(i,a);$$

$$e(i,a) \leftarrow e(i,a) + 1. \tag{13.13}$$

Step 2. Update $Q(l,u)$ all $l \in \mathscr{S}$ and $u \in \mathscr{A}(l)$ using

$$Q(l,u) \leftarrow Q(l,u) + \alpha \delta e(l,u). \tag{13.14}$$

Then, update the recency traces for all $l \in \mathscr{S}$ and $u \in \mathscr{A}(l)$ using

$$e(l,u) \leftarrow \lambda \gamma e(l,u).$$

Increment k by 1. If $k < k_{\max}$, then set $i \leftarrow j$, go to Step 1; otherwise, go to Step 3.

Step 3. For each $l \in \mathscr{S}$, select $d(l) \in \arg\max_{b \in \mathscr{A}(l)} Q(l,b)$. The policy (solution) generated by the algorithm is d. Stop.

Original Version of SARSA

SARSA(0), which can be interpreted as a special case of SARSA(λ) with $\lambda = 0$, was in fact the original version of SARSA [71]. It uses one-step updates, and hence no eligibility traces are needed in it. Furthermore, it can be shown to converge to the optimal policy under some mild conditions on how the ε-greedy action selection is performed (see [75] for details).

Steps in SARSA are similar to those for SARSA(λ) above with the following differences: No eligibility traces are required, and in Step 2, only the Q-factor of the current state-action pair, (i,a), is updated as shown below:

$$Q(i,a) \leftarrow Q(i,a) + \alpha \delta.$$

The multi-step updating of TD(λ), used in SARSA(λ), can speed up the updating in the sense that fewer state-transitions may be necessary in the simulator for convergence. But it also has some drawbacks: First, after each transition, all state-action pairs have to be updated, which is computationally intensive (unlike single-step updating in which only one state-action pair is updated after one state-transition). Secondly, and more importantly, SARSA (λ) requires the storage of the recency traces, which can be challenging via function approximation. On the other hand, SARSA, which does not require these traces, can be handily combined with function approximators, via the Bellman error approach (discussed in the previous section). Thus, overall, the usefulness of employing multi-step updating in simulators has not been resolved comprehensively in the literature and remains an open issue for future study.

Finite Trajectories

A version of SARSA (λ) that often performs better in practice uses finite trajectories. In such a version, the trajectory for any state is allowed to end (or is truncated) when that state is revisited. This concept has been discussed extensively in [76] in the context of the "replacing traces." In such a finite trajectory algorithm, the updating of the traces in the steps above is performed as follows: Eq. (13.13) is replaced by

$$e(i,a) = 1 \text{ and for all } u \in \mathscr{A}(i) \setminus \{a\},\ e(i,u) = 0;$$

in all other respects, the algorithm is identical to that described above.

The above mechanism for updating traces ensures that when a state is revisited, the feedback prior to that visit is discarded when that particular state is to be updated in the future. Essentially, the intuition underlying this is that the effect of an action in a state should only be measured by the impact the action produces in terms of the cumulative rewards generated by it, and this impact should terminated when that state is revisited, because a new (different) action will be selected when the state is revisited. Updating of this nature has been used widely in artificial intelligence; see e.g., [91]. This sort of updating can be tied to the original idea of TD(λ), defined in Eq. (13.12), by noting that the trajectory is now a finite one that ends when the state concerned is revisited.

The notion of TD(λ) has been discussed in the context of the value function, but not the Q-factors, in much of the literature [12, 79]. In the simulation-based setting, however, one is interested in the Q-factors, and hence SARSA(λ) holds more appeal to the simulation community. In the next subsection, we will discuss another application of TD(0)—also in the context of Q-factors.

13.3.4 Approximate Policy Iteration

We now present an algorithm based on ideas underlying policy iteration—a well-known DP technique that is based on selecting a policy, evaluating its value function (also called policy evaluation), and then moving on to a better policy. An algorithm called modified policy iteration [86] uses an iterative approach, rooted in value iteration, to evaluate the value function of a given policy. The classical form of policy iteration on the other hand performs the policy evaluation in one step by solving the Bellman equation for the given policy, which is also called the Poisson equation. This step requires the transition probabilities that we seek to avoid here. Modified policy iteration, however, is more relevant in the simulation-based context, since one can invoke the Q-factor version of the Poisson equation, and then use a Q-Learning-like approach to solve the Poisson equation, thereby avoiding the transition probabilities.

In this section, we present a Q-factor version of the modified policy iteration algorithm that can be used within a simulator. Algorithms belonging to this family have also been called Q-P-Learning [30], originally in the average reward context [31]. The algorithm we present is closely related to approximate policy iteration (API), which is based on the value function of DP, rather than Q-factors. API has been discussed extensively in [12]. Unfortunately, API based on the value function cannot be used directly when the transition probability model is not available, which is the case of interest here. When the transition probability model is available, it is unclear why one needs a simulator, since efficient methods for DP are available in the literature. Hence, we restrict our discussion to the Q-factor version here which can be implemented within simulators, bypassing the transition probability model.

The central idea is to start with any randomly selected policy and evaluate its Q-factors via the Poisson equation. In the algorithm, the given policy, whose Q-factors that are being evaluated, will be stored in the form of the P-factors (a name used to distinguish them from the Q-factors that are simultaneously being estimated within the simulator). A clear separation of the two types of Q-factors allows one to perform function approximation. The Q-factor version of the Poisson equation (or the Bellman equation for a given policy) for policy μ is:

$$Q(i,a) = \sum_{j=1}^{|\mathscr{S}|} p(i,a,j)\left[r(i,a,j) + \gamma Q(j,\mu(j))\right] \quad \forall (i,a).$$

Thus, when a given policy, μ, is available, the algorithm will seek to solve the above equation via a form of Q-Learning. As discussed in the context of Q-Learning, one can use the Robbins–Monro algorithm in a simulator to solve this equation without any need for the transition probabilities. When the Q-factors are evaluated, i.e., the above equation is solved, a new policy is generated via the policy improvement step [15]. We now present a step-by-step description of the Q-factor version of API.

Approximate Policy Iteration (API) Algorithm Using Q-Factors

Step 0. Input $k_{\max}$= number of iterations per policy evaluation and $E_{\max}$ = total number of policy evaluations, $\alpha \in (0,1)$. Initialize policy evaluations count $E = 0$ and all the P-values, $P(l,u)$ for all $l \in \mathscr{S}$ and $u \in \mathscr{A}(l)$, to arbitrary values.

Step 1 (Policy Evaluation). Start fresh simulation at any initial state. Initialize all the Q-values, $Q(l,u)$, to 0, and k, the number of iterations within a policy evaluation, to 0.

Step 1a. For current state i, select action a w.p. $1/|\mathscr{A}(i)|$, and simulate to reach next state j, receiving reward $r(i,a,j)$.

Step 1b. Update $Q(i,a)$ via

$$Q(i,a) \leftarrow (1-\alpha)Q(i,a) + \alpha \left[r(i,a,j) + \gamma Q\left(j, \arg\max_{b \in \mathscr{A}(j)} P(j,b) \right) \right]. \quad (13.15)$$

Step 1c. Set $k \leftarrow k+1$. If $k < k_{\max}$, set $i \leftarrow j$ and go to Step 1a; else go to Step 2.

Step 2 (Policy Improvement). Set for all $l \in \mathscr{S}$ and all $u \in \mathscr{A}(l)$,

$$P(l,u) \leftarrow Q(l,u); \quad E \leftarrow E+1.$$

If E equals $E_{\max}$, then go to Step 3; otherwise, go to Step 1.

Step 3. For each $l \in \mathscr{S}$, select $d(l) \in \arg\max_{b \in \mathscr{A}(l)} Q(l,b)$. The policy (solution) generated by the algorithm is d. Stop.

In practice, the algorithm exhibits robust behavior but may be time-consuming, since a number of policies are generally evaluated before the algorithm converges, and each policy evaluation requires numerous iterations. The convergence of this algorithm, which is a special case of TD(0) adapted to Q-factors, can be shown along the lines of the convergence of Q-Learning [39]. Furthermore, as in Q-Learning, one can use function approximation; separate approximators would be needed for the Q- and the P-function.

13.3.5 Actor-Critic Algorithm

We now present an API algorithm that is based on policy iteration but is much faster because it performs the policy evaluation via only one iteration. The algorithm has been called the actor-critic or the adaptive critic in the literature. This algorithm has evolved over time with ideas from [8, 53, 89, 95]. We present the most modern version, which has some proven convergence properties.

The algorithm stores the value function and a substitute (proxy) for the action–selection probability. Here $J(i)$ will denote the value function for state i, and $H(i,a)$

for all $i \in \mathcal{S}$ and all $a \in \mathcal{A}(i)$ will denote the quantities (proxies) used to select actions in the states visited. We present the step-by-step algorithm.

Approximate Policy Iteration (API) via Actor-Critic

Step 0. Input $k_{\max}$= total number of iterations, $\alpha \in (0,1)$. Initialize iteration count $k = 0$ and all J-values and H-values to 0, i.e., for all l, where $l \in \mathcal{S}$, and $u \in \mathcal{A}(l)$, set $J(l) \leftarrow 0$ and $H(l,u) \leftarrow 0$. Initialize a scalar, $\bar{H}$, to the largest possible value such that $\exp(\bar{H})$ can be stored in the computer's memory without overflow. Start system simulation at any arbitrary state.

Step 1. For current state i, select action a w.p.

$$\frac{\exp(H(i,a))}{\sum_{b \in \mathcal{A}(i)} \exp(H(i,b))},$$

and simulate to reach next state j, receiving reward $r(i,a,j)$. The above style of action selection is called the Gibbs-softmax method of action selection.

Step 2. (Critic update) Increment k by 1. Update $J(i)$ via

$$J(i) \leftarrow (1-\alpha)J(i) + \alpha\left[r(i,a,j) + \gamma J(j)\right].$$

Step 3. (Actor update) Update $H(i,a)$ using a step size, β, that shares a special relationship with α (discussed below):

$$H(i,a) \leftarrow H(i,a) + \beta\left[r(i,a,j) + \gamma J(j) - J(i)\right].$$

If $H(i,a) < -\bar{H}$, set $H(i,a) = -\bar{H}$; if $H(i,a) > \bar{H}$, set $H(i,a) = \bar{H}$.

Step 4. If $k < k_{max}$, then set $i \leftarrow j$, go to Step 1; otherwise, go to Step 5.

Step 5. For each $l \in \mathcal{S}$, select $d(l) \in \arg\max_{b \in \mathcal{A}(l)} H(l,b)$. The policy (solution) generated by the algorithm is d. Stop.

The algorithm's ε-convergence to optimality can be shown when the step-sizes share the following relationship in addition to the usual conditions of stochastic approximation and some other conditions [53]:

$$\lim_{k \to \infty} \frac{\beta^k}{\alpha^k} = 0.$$

The above requires that β converge to 0 faster than α. It is not difficult to find step sizes that satisfy these conditions. One example that satisfies all of these conditions required in [53] is: $\alpha^k = log(k)/k$ and $\beta^k = A/(B+k)$. Although the initial policy is a stochastic policy, under ideal conditions of convergence, the algorithm converges to a deterministic stationary optimal policy in the limit as $k \to \infty$.

Note that the update in Step 4 ensures that the $H(\cdot,\cdot)$ values are projected onto the interval $[-\bar{H},\bar{H}]$. This is necessary because in the simulator the values of one or more elements of the matrix $H(\cdot,\cdot)$ become unbounded.

The algorithm can be combined with function approximation, using ideas related to Bellman error, by using separate approximators for $J(\cdot)$ and $H(\cdot,\cdot)$. One difficulty with this algorithm is that convergence to optimality depends on finding a sufficiently large value for $\bar{H}$. Note that the algorithm requires computation of $\exp(H(i,a))$. If the theoretical value for $\bar{H}$ that leads to an optimal solution is so large that the computer overflows in trying to compute $\exp(\bar{H})$, one obtains sub-optimal policies.

13.3.6 Evolutionary Policy Iteration

We conclude this section with a relatively recent development that combines ideas from genetic algorithms (GAs) and policy iteration. The GA is a widely used meta-heuristic [44] based on the principles of genetic evolution. It is typically used in discrete optimization when the number of solutions is large, but a mechanism is available to estimate the objective function at each solution. The GA is a very popular algorithm known to have the ability of generating good solutions when the number of solutions is very large. The other attractive feature of GA is that it is simple to code.

The algorithm of interest here is Evolutionary Policy Iteration (EPI) due to [25]. It uses simulation to evaluate the value function of each state for a given policy. The overall scheme is similar to that used in a typical GA, and in what follows, we present an informal explanation.

One starts with an arbitrarily selected population (collection) of n policies. Thereafter, an *elite* policy is generated from the population. This is the best policy in the current population; the elite policy and how it is generated is a special feature of this algorithm. Via mutation, $(n-1)$ policies are generated from the existing population; this is called offspring generation in GAs. The elite policy and the $(n-1)$ newly generated policies are considered to be the new population, and the algorithm repeats the steps described above performing another iteration. The algorithm terminates when the last K (a large number, e.g., 20) iterations have not produced any change in the elite policy.

Estimating the value function of a given policy, μ, for a given state, i, can be performed via one simulation trajectory as shown below (it is also possible to use a one long trajectory to update all states, but we restrict our discussion to the case of a given state). The output generated by the rth trajectory will be denoted by $\hat{V}_{\mu}^{r}(i)$. In practice, one needs to simulate numerous trajectories, and obtain the average of the values generated by each trajectory. Thus, if R trajectories are simulated, the average will be:

$$\tilde{V}_{\mu}(i) = \frac{\sum_{r=1}^{R} \hat{V}_{\mu}^{r}(i)}{R}.$$

The routine described below must be applied for each state in the system separately.

Inputs: Given a state i, a policy μ, and the trajectory index r.
Initialize TR, the total reward, to 0. Set k, the number of steps, to 0.
Loop until $k = k_{\max}$, where $k_{\max}$ denotes the number of state transitions in the trajectory:

- Start simulation in a given state i. Selection action $\mu(i)$ in state i. Let the next state be j. Observe the immediate reward, $r(i,\mu(i),j)$. Update TR as follows:

$$TR \leftarrow TR + \gamma^k r(i,\mu(i),j).$$

- Set $i \leftarrow j$ and $k \leftarrow k+1$.

Output: $\hat{V}^r_\mu(i) \leftarrow TR$.

Note that in the above, the value of $k_{\max}$ depends on γ. Clearly, when γ^k is very small, there is no point in continuing with the trajectory any further. Thus, in practice, $k_{\max}$ can be quite small; however, the above routine needs to be performed for numerous (R) trajectories and for each state separately. In what follows, we will assume that the above routine can be called whenever the value function of a state for a given policy is to be estimated.

Evolutionary Policy Iteration (EPI) Algorithm

Initialization: Set the iteration count, k, to 0. The population (collection) of policies in the kth iteration will be denoted by $\mathscr{L}^k$. Populate $\mathscr{L}^0$ with n arbitrarily selected policies. Set θ, the global/local selection probability, to an arbitrarily selected value between 0 and 1. Set a and b, the local and the global mutation probabilities, also to values in (0,1) such that $a < b$. Initialize action–selection probabilities, $P(i,a)$ for all $i \in \mathscr{S}$ and all $a \in \mathscr{A}(i)$, to some values such that $\sum_b P(i,b) = 1$ for every $i \in \mathscr{S}$. One example is to set $P(i,a) = 1/|\mathscr{A}(i)|$ for every $i \in \mathscr{S}$.

Loop until a termination criterion is met:

- Select an elite policy, denoted by π_*, from $\mathscr{L}^k$ via the following computation. For every state $i \in \mathscr{S}$, select an action a_i as follows:

$$a_i \in \left\{ \arg\max_{\pi \in \mathscr{L}^k} \tilde{V}_\pi(i) \right\} \text{ and set } \pi_*(i) = a_i.$$

- **Offspring generation:**

  1. Generate $(n-1)$ subsets of $\mathscr{L}^k$, denoted by $\mathscr{Y}(t)$ for $t = 1,2,\ldots,n-1$. Each of these subsets will contain m policies where m itself is a random number from the discrete uniform distribution, $DU(2,n-1)$. The m policies that will be selected for $\mathscr{Y}(\cdot)$ will be selected with equal probability from the set $\mathscr{L}^k$.

2. Generate $(n-1)$ offspring (policies), denoted by π^t for $t = 1,2,\ldots,n-1$, as follows: For each state $i \in \mathscr{S}$ and each value of t, select an action u_i^t as follows:

$$u_i^t \in \left\{ \arg\max_{\pi \in \mathscr{Y}^k(t)} \tilde{V}_\pi(i) \right\} \text{ and set } \pi^t(i) = u_i^t.$$

3. For each π^t for $t = 1,2,\ldots,n-1$, generate a mutated policy, denoted by $\hat{\pi}^t$, from π^t as follows. With probability θ, use a as the mutation probability and with the remaining probability $(1-\theta)$, use b as the mutation probability. The mutation probability is the probability with which each state's action in the policy undergoes a change (mutation). If a state's action is to be changed, it is changed to an action selected via the action–selection probabilities, $P(\cdot,\cdot)$, defined above.

- Generating the new population: The new population $\mathscr{Q}^{k+1}$ is now defined to be the following set:

$$\{\pi_*, \hat{\pi}^1, \hat{\pi}^2, \ldots, \hat{\pi}^{n-1}\}.$$

- Set $k \leftarrow k+1$.

Output: The policy π_* is the best policy generated by EPI.

The idea of mutating policies can be explained via a simple example. Assume $a = 0.2$ and $b = 0.9$. Further assume that $\theta = 0.4$. Then, a random number from the distribution $U(0,1)$ is first generated. If this number is below $\theta = 0.4$, then the mutation probability will be a; otherwise, the mutation probability will be b. When it comes to mutating a given policy, consider the following example. Assume that there are three states and four actions in each state. Furthermore, for the sake of exposition, the policy, μ, will now be expressed as $(\mu(1), \mu(2), \mu(3))$. Consider the following policy which is to be mutated w.p. 0.9:

$$(3,4,1).$$

A random number, y, from the distribution $U(0,1)$ will be generated for each state. We illustrate these ideas for state 2 in which the action prescribed by the policy is 4. If for state 2, $y < 0.9$, then the state's action will be altered to one dictated by $P(\cdot,\cdot)$, defined above. Then an action v is selected for state 3 w.p. $P(3,v)$. On the other hand, if $y \geq 0.9$, the state's action will be unaltered. These mutations are performed for every state in the system for every policy that is to be mutated.

13.4 Average Reward MDPs

Average reward MDPs (infinite horizon) are commonly associated to problems where discounting does not appear to be appropriate. This is usually the case when the discount factor is very close to 1. Also, finite-horizon problems are often converted into infinite-horizon problems to make them tractable, and if the horizon is short, money does not lose value appreciably to factor that into the calculations. Under these conditions also, the average reward is a more appropriate performance metric. Finally, there are scenarios, e.g., in queueing networks, electrical systems and production systems, where the rewards/costs are often calculated in terms of non-monetary measures such as waiting time, utilization, inventory, etc. Again, in such instances, the average reward is a more suitable performance metric.

Surprisingly, simple extensions of the discounted MDP algorithms do not work in the average reward domain. Thus, for instance, setting $\gamma = 1$ in Q-Learning leads to an unstable algorithm in which the Q-factors become unbounded. For average reward, we cover two different algorithms: one based on relative value iteration and the other based on updating the value of average reward, alongside the Q-factors. Note that the Bellman optimality equation for average reward, Eq. (13.2), contains ρ^* as an unknown in addition to the Q-factors. Hence, a straightforward extension of Q-Learning is ruled out.

13.4.1 Relative Q-Learning

The algorithm we now describe employs the concept of relative value iteration in combination with Q-Learning and originates from [2]. The steps differ from those in Q-Learning as follows:

- In Step 0, any state-action pair in the system is selected, and henceforth termed the distinguished state-action pair. It will be denoted by (i^*, a^*).
- In Step 2, the update is performed as follows:

$$Q^{k+1}(i,a) = (1-\alpha^k)Q^k(i,a) + \alpha^k \left[r(i,a,j) + \max_{b \in \mathscr{A}(j)} Q^k(j,b) - Q^k(i^*,a^*) \right].$$

The algorithm can be shown to converge w.p.1 to the optimal solution [2], under conditions identical to those required for Q-Learning. Furthermore, it can be shown that w.p.1,

$$\lim_{k \to \infty} Q^k(i^*,a^*) = \rho^*,$$

which implies that in the limit the Bellman optimality equation for average reward is solved.

13.4.2 R-SMART

We now present another algorithm for average reward MDPs [38], which is a refined version of algorithms originally presented for semi-MDPs (a more general version of the MDP) in [26, 32]. We restrict the discussion here to the MDP, which is a special case of the semi-MDP. The algorithm is called R-SMART (short for Relaxed-Semi-Markov Average Reward Technique). Like SARSA, this algorithm relies on using ε-greedy action selection, and further it uses a separate update for the average reward term. There are two versions of this algorithm. One uses connection to the stochastic shortest path (SSP) problem [15], and the other uses an artificial contraction factor. We now present details.

SSP-Version

In this version, the average reward problem is essentially transformed into an SSP. Since we are not interested in the original SSP, we do not explain the SSP in detail, but refer the reader to [15]. What is important to note is that the SSP transformation creates a contractive mapping for the Q-factors enabling their convergence. The steps in the resulting algorithm will have the following differences with those of Q-Learning:

- In Step 0, select any state in the system, and call it the distinguished state i^*.
- In this algorithm, some parameters related to the average reward calculation will be needed. They are TR, the total reward, TT, the total time, and ρ^k, the estimate of the average reward in the kth iteration. In Step 0, initialize each of TR, TT, and ρ^1, to 0.
- In Step 1, select action a using an ε-greedy form of action selection (as discussed in the context of SARSA).
- In Step 2, update $Q(i,a)$ as follows:

$$Q^{k+1}(i,a) = (1-\alpha^k)Q^k(i,a) + \alpha^k \left[r(i,a,j) + \mathbf{1}\{j \neq i^*\} \max_{b \in \mathscr{A}(j)} Q^k(j,b) - \rho^k \right], \tag{13.16}$$

where $\mathbf{1}\{\cdot\}$ is the indicator function that returns a 1 if the condition inside the brackets is satisfied and zero otherwise. This is followed by an update of ρ. If a greedy action was selected in Step 1, i.e., if $a \in \arg\max_{b \in \mathscr{A}(i)} Q(i,b)$, then update TR, TT and ρ^k in the order shown below:

$$TR \leftarrow TR + r(i,a,j);$$

$$TT \leftarrow TT + 1;$$

$$\rho^{k+1} \leftarrow (1-\beta^k)\rho^k + \beta^k (TR/TT).$$

If a greedy action is not selected, set $\rho^{k+1} \leftarrow \rho^k$.

In the update for ρ^k, the step size β^k is chosen in a manner discussed in the actor-critic algorithm. The indicator function within the update in Eq. (13.16) ensures that the problem is essentially transformed into an SSP. Further note that the steps above ensure that Q-factors for state-action pair (i^*, a) are updated like every other Q-factor. Under some conditions shown in [38], the algorithm is shown to converge to the optimal solution and ρ^k to ρ^* w.p.1. It is also worthwhile pointing out that the state i^* plays the role of the (fictitious) absorbing state in the SSP when it is encountered as the next state (j) in a transition.

Contraction-Factor Version

In this version of R-SMART, an artificial contraction factor, $\bar{\gamma}$, will be used. This artificial factor will make the Q-factor transformation contractive, thereby enabling convergence. The average reward will be updated as done above in the SSP-version. The steps in the contraction-factor version of R-SMART will have the following differences with those of Q-Learning:

- In Step 0, select a suitable value for $\bar{\gamma} \in (0, 1)$.
- As in the SSP-version above, in Step 0, initialize each of TR, TT, and ρ^1, to 0.
- In Step 1, select action a using an ε-greedy form of action selection.
- In Step 2, update $Q(i, a)$ as follows:

$$Q^{k+1}(i,a) = (1-\alpha^k)Q^k(i,a) + \alpha^k \left[r(i,a,j) + \bar{\gamma} \max_{b \in \mathscr{A}(j)} Q^k(j,b) - \rho^k \right].$$

Thereafter, update ρ^k as shown in the steps of the SSP-version.

In the above algorithm, the value of $\bar{\gamma}$ must be guesstimated. Convergence can be assured under certain conditions that require the knowledge of the transition probabilities. In practice, a value close to 1 often generates optimal solutions. Although, it is not possible to guess the correct value of $\bar{\gamma}$ and one must use trial and error, on large-scale problems, this version often outperforms the SSP-version.

13.5 Finite Horizon MDPs

As noted earlier, the finite-horizon problem includes stages, which further adds to the curse of dimensionality posed by the state-action space of infinite-horizon problems. Nonetheless, these problems are important in their own right. They find many applications in operations research. Many inventory control problems and problems in revenue management belong to the finite time horizon. In machine learning, these problems are called *episodic tasks*. Sometimes finite-horizon problems can

be converted into infinite-horizon problems by introducing an artificial transition from the absorbing state to a starting state; the reason for this transformation is that it can make the problem tractable [40].

In this section, we will first discuss an SSP algorithm for solving the finite-horizon problem. Thereafter, we will consider a learning automata algorithm that has been developed more recently.

13.5.1 Special Case of Stochastic Shortest Path (SSP)

The finite-horizon problem can be studied as a special case of the stochastic shortest path (SSP) problem in which the state-action pair is replaced by the state-stage-action triple [15]. Notation for this problem has been introduced earlier, and the reader should review it at this time.

We will assume that there are T stages in which decision-making is to be performed. When the system reaches the $(T+1)$th stage, there is no decision making to be done, and the simulator will return to the starting state, which is assumed to be known. The following description is based on [37].

Simulation-Based Algorithm for Finite-Horizon MDPs

Step 0. Input $k_{\max}$= total number of iterations. Initialize iteration count $k=0$, stage $s=1$, and all the Q-factors, $Q^k(i,s,a)$ for all $i \in \mathscr{S}$, all $s \in \mathscr{T}$ and all $a \in \mathscr{A}(i,s)$, to 0. Also set $Q^k(j,T+1,b)=0$ for all k, all $j \in \mathscr{S}$ and $b \in \mathscr{A}(j,T+1)$. Start system simulation at the starting state, which is assumed to be known with certainty.

Step 1. For current state i and current stage s, select action a w.p. $1/|\mathscr{A}(i,s)|$, and simulate to reach next state j in stage $(s+1)$, receiving reward $r(i,s,a,j,s+1)$.

Step 2. Update the Q-value of (i,s,a) as follows:

$$Q^{k+1}(i,s,a) \leftarrow (1-\alpha^k)Q^k(i,s,a)+\alpha\left[r(i,s,a,j,s+1)+\max_{b\in\mathscr{A}(j,s+1)} Q^k(j,s+1,b)\right].$$

Increment k by 1 and s by 1, and if the new value of s equals $(T+1)$, set $s=1$. If $k<k_{max}$, then set $i \leftarrow j$ and return to Step 1; otherwise, go to Step 3.

Step 3. For each $l \in \mathscr{S}$ and each $s \in \mathscr{T}$, select $d(l,s) \in \arg\max_{b\in\mathscr{A}(l,s)} Q^k(l,s,b)$. The policy (solution) generated by the algorithm is d.
Stop.

13.5.2 Pursuit Learning Automata (PLA) Sampling

We now present the pursuit learning automata (PLA) sampling algorithm due to [23]. The theory of learning automata for MDPs has been covered extensively in [60]. This particular algorithm is based on the concept of PLA [67, 85]. The idea here is to apply a simulation-based sampling algorithm at each state-stage pair in the system. It is possible to start the simulation at any given state-stage pair, and simulate different actions. The algorithm is remarkably robust, and can converge quickly to the optimal solution. In algorithms considered previously, one simulates a long trajectory in which states are visited in an asynchronous manner. This algorithm is also simulation-based, thereby avoiding the transition probability model, but requires that the system be simulated in each state-stage pair separately; also, a long simulation trajectory is not needed here. In many ways, it uses the power of simulation and at the same time does not leave the convergence to count on visiting each state-stage pair infinitely often. Thus, it appears to combine the graceful and systematic synchronous updating of dynamic programming with the power of simulation.

The algorithm will be presented for a given state-stage pair (i,s). Since the value function of the next state-stage pair will be required, a recursive call will be necessary to that pair if a forward pass is done. Alternatively, one could start at the last decision-making stage (T) in any state, update all the states in that stage, and then move backwards, one stage at a time, as is done in backward dynamic programming. Furthermore, in a backward pass style of updating, all states for that stage must be updated before moving to the previous stage. Remember that for stage T, the value function of the next stage is zero. Thus, if $V(l,s)$ denotes the value function for state l and stage s, then $V(l,T+1) = 0$ for all l.

The steps below present a routine (function) that must be performed for each state-stage pair separately. A step-size α, which could be state-dependent, will be used in updating the probabilities of the learning automaton (LA). In each state, the LA will store an action–selection probability of $P(i,a)$ such that $\sum_a P(i,a) = 1$. Some other counters will be needed: $TR(l,s,u)$ will measure the total reward accumulated thus far within the routine when action u is tried in state l when encountered in stage s, and $N(l,s,u)$ will measure the number of times action u has been tried in the state-stage pair, (l,s).

The initialization step must be performed each time this routine is called. For each call, we have a given state, i, and a given stage, s.

Pursuit Learning Automata (PLA) Sampling Algorithm

Initialization: Input $k_{\max}(i,s)$ = number of iterations (simulations) allowed for each state-stage pair (i,s) (simulation budget), $\alpha \in (0,1)$. Initialize the LA probabilities as follows: For any $l \in \mathscr{S}$ and all $a \in \mathscr{A}(l)$, $P(l,a) = 1/|\mathscr{A}(l)|$. Initialize the counters, $TR(l,s,u)$ and $N(l,s,u)$, for all $l \in \mathscr{S}$, all $s \in \mathscr{T}$, and all $u \in \mathscr{A}(l,s)$, to 0; the iteration count $k = 0$; and $V(l,T+1) = 0$ for all l.

Step 1. For current state i and current stage s, select action a w.p. $P(i,a)$, and simulate to reach next state j and next stage $(s+1)$, receiving reward $r(i,s,a,j,s+1)$.

Step 2. Update the following quantities:

$$TR(i,s,a) \leftarrow TR(i,s,a) + r(i,s,a,j,s+1) + V(j,s+1); \tag{13.17}$$

$$N(i,s,a) \leftarrow N(i,s,a) + 1.$$

Then compute

$$Q(i,s,a) \leftarrow \frac{TR(i,s,a)}{N(i,s,a)}. \tag{13.18}$$

Step 3. Determine the greedy action as follows:

$$a^* \in \arg\max_{b \in \mathscr{A}(l,s)} Q(l,s,b).$$

Step 4. Update the action–selection probabilities: For all $u \in \mathscr{A}(i,s)$,

$$P(i,u) \leftarrow (1-\alpha)P(i,u) + \alpha \mathbf{1}\{a^* = u\}.$$

Step 5. Increment k by 1. If $k < k_{\max}(i,s)$, then return to Step 1; otherwise, go to Step 6.

Step 6. Set

$$V(i,s) = Q(i,s,a^*).$$

Stop.

Note that the update in Eq. (13.17) requires the value function of the next state-stage pair. In a backward pass application of the above routine, this value will already have been computed and will be available for use in the update. In case one does not use a backward pass, estimating this value will have to be done via a recursive call to the next state-stage combination $(j,s+1)$. It is also interesting to note that the computation of the Q-factor in Eq. (13.18) is based on *direct* averaging, a concept used in model-building (also called model-based) RL [79].

13.6 Numerical Results

In this section, we present numerical results on some small problems in order to illustrate the use of RL algorithms. The case studies covered here are for problems whose transition probabilities can be estimated; thus, it is possible to determine whether the optimal solution was reached. Furthermore, these problems can also be used as testbeds by other researchers for empirical investigation of their own algorithms. In practice, a simulation analyst is generally also interested in testing an algorithm on large-scale problems, whose transition probabilities are difficult to estimate and the only benchmarks available are problem-specific heuristics. Large-scale versions of both case studies that we cover here are also candidates for such tests. We begin with the infinite-horizon problem, and then discuss the finite horizon case.

13.6.1 Infinite Horizon

The case study we cover here is on preventive maintenance of machines and is primarily drawn from [34]. It is well-known that systems whose probabilities of failures increase as they age can benefit from preventive maintenance. Examples of such systems include production lines, bridges, roads, and electric power plants.

We consider the case of a production line which deteriorates with time and the deterioration can be captured by a function. After a preventive maintenance, generally, the system has a lower probability of failure than when it failed. Since it generally costs much lower to preventively maintain a line than to repair it after a failure, a significant volume of literature has appeared in the area of preventive maintenance. Toyota Motors have popularized the use of preventive maintenance in many automobile firms. A large chunk of the literature studies the problem of determining the time interval after which maintenance should be performed.

We make the following assumptions about the system:

- The production line is needed every day.
- If the line fails during the day, the repair takes the remainder of the day, and the line is available only the next morning. After a repair, the line is as good as new.
- When a line is shut down for preventive maintenance, it is down for the entire day. After a preventive maintenance, the line is as good as new.
- If σ denotes the number of days elapsed since the last preventive maintenance or repair (subsequent to a failure), the probability of failure during the σth day can be modeled as $(1-\xi\psi^{\sigma+2})$, where ξ and ψ are scalars in the interval $(0,1)$, whose values can be estimated from the data for time between successive failures of the system.
- For any given positive value of $\varepsilon \in \mathbb{R}$, we define $\bar{\sigma}_\varepsilon$ as the minimum integer value of σ such that the probability of failure on the $\bar{\sigma}$th day is less than or equal to $(1-\varepsilon)$. Since a fixed value of ε will be used, we will drop ε from the

notation. The definition of $\bar{\sigma}$ will allow us to truncate the countably infinite state space to a finite one. The resulting state space of the system will be assumed to be $\mathscr{S} = \{0, 1, 2, \ldots, \bar{\sigma}\}$, i.e., the probability of failure on the $\bar{\sigma}$th day will be assumed to equal 1. Furthermore, note that rounding of this nature is necessary to ensure that the probabilities in the last row of the finite Markov chain for one of the actions (the production action in particular) add up to 1.
- The costs of maintenance and repair are known with certainty, and will equal C_m and C_r respectively.

The underlying system transitions can be modeled via Markov chains as follows. Let the state of the system be defined by σ, the number of days elapsed since the last preventive maintenance or repair. Clearly, when a maintenance or repair is performed, σ is set to 0. If a successful day of production occurs, i.e., no failure occurs during the day, the state of the system is increased by 1. Each morning, the manager has to choose from two actions: $\{produce, maintain\}$. Then, we have the following transition probabilities for the system. We first consider the action *produce*. For $\sigma = 0, 1, 2, \ldots, \bar{\sigma} - 1$,

$$p(\sigma, produce, \sigma + 1) = \xi \psi^{\sigma+2}; \quad p(\sigma, produce, 0) = 1 - \xi \psi^{\sigma+2}.$$

For $\sigma = \bar{\sigma}$, $p(\sigma, produce, 0) = 1$. For all other cases not specified above,

$$p(\cdot, produce, \cdot) = 0.$$

For the action *produce* and all values of σ,

$$r(\sigma, produce, 0) = -C_r; \quad r(\sigma, produce, \sigma') = 0 \text{ when } \sigma' \neq 0.$$

For the action *maintain*, the mathematical dynamics will be defined as follows. For all values of σ, $p(\sigma, maintain, 0) = 1$ and $r(\sigma, maintain, 0) = -C_m$. For all other cases not specified above, $p(., maintain, .) = 0$ and $r(., maintain, .) = 0$.

We set $\xi = 0.99$, $\psi = 0.96$, $C_m = 4$, $C_r = 2$, and $\bar{\sigma} = 30$. Thus, we have 31 states and 2 actions. Our objective function is average reward, and the optimal policy, which is determined via policy iteration, is of a threshold nature in which the action is to produce for $\sigma = 0, 1, \ldots, 5$ and to maintain from $\sigma = 6$ onwards. The contracting factor version of the algorithm R-SMART was used in a simulator with the following specifications:

- The artificial contraction factor, $\bar{\gamma}$, was set to 0.99.
- The learning rates used were:

$$\alpha^k = \frac{1000}{5000 + k}; \quad \beta^k = \frac{1000}{k(5000 + k)} \text{ where } k \geq 1.$$

- The exploration probability was defined as shown below:

$$P^k = 0.5\frac{log(k+1)}{k+1} \text{ where } k \geq 1.$$

- The system was simulated for 200,000 days.

The algorithm always generated the optimal solution in every replication. At least 20 different replications were performed.

The above problem can be studied for a much larger state space. If the time between failures is significantly higher and as a result $\bar{\sigma}$ is closer to 1,000, the transition probability matrix contains a million elements, which is not easy to handle. In other words, dynamic programming breaks down. However, it is not difficult to simulate this system, allowing us to use the simulation-based algorithm—if necessary in conjunction with some function approximation. It is also very critical to note here that unlike the problem considered above, usually the transition probability structure is not available, and the system can still be simulated as long as the distributions of the input random variables are available. Thus, for example, in an $M/G/1$ queue, if one observes the system at the instants when arrivals occur, one can formulate a Markov chain. However, the transition probabilities of this Markov chain are not necessary to simulate the system; rather one can simulate the queue using the distributions of the inter-arrival time and the service time. In preventive maintenance problems also, simulation of the system is often possible without generating the transition probabilities (see e.g., [26]).

In general, look-up tables work with up to maybe 2,000 Q-factors in regular computers, but for a state-action space larger than that, some sort of function approximation becomes necessary. Thus, for large-scale MDPs, it is imperative that the analyst either seeks to reduce the state-action space to a reasonable number or alternatively uses a function approximation scheme. Using function approximation introduces additional computational issues with regards to how to capture the state-action space in terms of the basis functions (architectures). It is not uncommon in practice to experiment with a large number of candidate architectures before the algorithm starts outperforming a heuristic or a set of heuristics known to generate reasonable results. It is this aspect of the large-scale problem that makes it necessary to study, or generate if necessary, some problem-specific heuristics. In summary, one can conclude that a successful implementation of RL on a large-scale problem is a significant computational exercise that requires patience.

13.6.2 Finite Horizon

We now consider a finite-horizon MDP where the objective is to maximize the expected value of the total undiscounted reward over the time horizon. We use an example, drawn from [24], on inventory control. Inventory control problems are

ubiquitous in supply chain management, and indeed, most modern supply-chain software seek to solve problems of the nature considered here. Although the transition probabilities will be computed to determine the optimal solution, as in the previous subsection, these problems can be solved for any given distribution of the input random variable—the size of the demand (in one period/stage) in this case.

In many inventory control problems, the manager seeks to order raw material in a fashion so as to minimize the fixed costs of ordering (also called set-up costs), the costs of lost sales, and the costs of holding inventory. In general, large order quantities increase holding costs, but reduce the costs of lost sales and the ordering costs; on the other hand, small order quantities increase the risk of lost sales, the ordering costs, but minimize the inventory holding costs. Clearly, without holding costs, the problem has a trivial solution, which is: order the maximum possible quantity. A goal of modern inventory control [5], however, is to maximize inventory turns and minimize inventory. As such, the holding costs play a critical role in this problem. Moreover, this is a multiple-period problem in which decisions for ordering quantities have to be made in every period (stage) separately and unused inventory from a previous time period is carried over into the next time period.

Let D_s denote the demand during the sth stage (period), x_s denote the inventory at the start of the sth stage, and u_s denote the action chosen in the sth stage. The action u_s will equal the amount ordered at the start of the sth stage. The following assumptions will be made about the problem:

- There is no backordering, and a demand lost is lost forever.
- The size of demand in a given time period is a discrete random variable whose distribution is known.
- The holding costs per unit per unit time, h, the cost per order, A, and the lost sales cost per unit (opportunity cost), p, are known with certainty.
- We will assume that the demand will be realized at the end of the stage, and the order placed at the start of a stage will arrive at the end of the stage. This will simplify the computation of the inventory holding costs.
- There is an upper limit, M, on the amount of inventory that can be held (dictated by storage requirements). This implies that an ordering amount that causes inventory to exceed M will not be allowed.
- The number of stages, T, is deterministic and known.
- The starting inventory, x_1, is known with certainty.

Under these conditions, the inventory levels will then change from one stage to the next as follows:

$$x_{s+1} = (x_s + u_s - D_s)^+, \;\; x_1 \geq 0,$$

where $x^+ \equiv \max(x, 0)$, which reflects the condition that inventory cannot be negative due to the no backordering assumption. The goal is to minimize the expected total costs of operating the system, where costs can arise out of holding inventory, lost sales, and the set-up (fixed) cost per order. The total cost in one trajectory can be computed as follows:

$$\sum_{s=1}^{T} \left[A \cdot \mathbf{1}\{u_s > 0\} + hx_s + py_s^- \right],$$

where $x^- \equiv \max(-x, 0)$ and $y_{s+1} = x_s + u_s - D_s$ for $s = 1, 2, \ldots, T$, and $y_1 = 0$.

The following values were used for the inputs in the experiments reported in [24]: $p = 10$, $A = 5$, $h = 1$, $M = 20$, $T = 3$, $x_1 = 5$, and $k_{\max}(i, s) = 4$ for all state-stage pairs. The demand was assumed to have a discrete uniform distribution, $DU(0, 9)$. Backward dynamic programming was used to determine the optimal solution, which was also delivered by the PLA sampling technique. Then, via simulation, the value function for the starting state, $V_{PLA}(x_1, 1)$, was computed for the optimal policy, using 30 replications and a large number of samples. This value was found to be 24.48 with a standard error of 0.51, which compares well with the true value, $V_*(x_1, 1) = 27.322$, determined via backward dynamic programming (which needed the exact transition probabilities). Numerous results with other values for the inputs can be found in [24].

13.7 Extensions

The ideas underlying MDPs can be extended to at least three other domains: semi-MDPs (SMDPs), stochastic games, also called Competitive Markov decision processes (CMDPs), and Partially Observable MDPs (POMDPs).

In an SMDP, the time of transition from one state to another is not the same for every transition. In the most general case, this time is a generally distributed random variable. Although the SMDP is sometimes loosely referred to as a continuous time MDP (CTMDP), the latter name is usually reserved for the SMDP in which the transition times have the exponential distribution. The theory of SMDPs can be studied for both discounted and average reward objective functions [15]. RL for SMDPs has been studied in [19] (discounted reward) and [31, 38] (average reward).

In a CMDP, there are multiple decision-makers, and the transition probabilities and rewards depend on the actions of all or a subset of all decision-makers. The problem becomes significantly more complex and has been studied in detail in [29]. These problems are of considerable interest to economists, and both of the early contributors [61, 73] have been awarded Nobel prizes in economics. The work in [61] provides a critical idea, called Nash equilibrium, needed for solving a CMDP, while the work of Shapley [73] provides the first attempt at value iteration and solving the CMDP computationally.

In a POMDP, the underlying state is only partially observable to the decision maker via signals, and the decision-maker is required to choose the best possible action. The POMDP has been used in robotics and pattern recognition problems.

Not surprisingly, attempts have been made to use simulation to solve POMDPs and CMDPs via simulation. Some noteworthy works in CMDPs include the algorithms in [46, 69]. Some special forms of sequential games in which the actions

of the different decision-makers are not concurrently taken but are sequential are studied in [12]. For POMDPs, the body of literature is somewhat evolved. See [49] for a survey of early algorithms and [24] for some more recent simulation-based algorithms.

A large number of computational extensions can be found in computer science, e.g., hierarchical RL, although convergence guarantees for the underlying theory are somewhat sparse. Another topic that we did not cover is that of policy gradients [9, 77], which is rooted in the idea of using simulation and derivatives of the objective function to generate an optimal policy in MDPs. These algorithms suffer from large variance.

13.8 Convergence Theory

The convergence theory for RL algorithms has become quite rigorous. Here, we provide a brief account. At least three different lines of convergence arguments have been worked out for classical Q-Learning-type algorithms:

1. theory developed in [12], which is based primarily on the idea of "reducing cube sizes" (see Prop. 4.5 of [12]);
2. theory developed in [48, 75, 83], which is based on some remarkably simple arguments and draws on some basic results in stochastic approximation (see [48] in particular), and
3. theory based on ordinary differential equations, for which the reader is referred to [16].

Although one finds three distinct strands of convergence arguments, many of the proofs rest on showing some basic properties, e.g., the underlying transformation is contractive (shown for Q-Learning and the SSP-version of Q-Learning in [12]) and the iterates remain bounded (shown for the SSP-version of Q-Learning in [12] and for classical Q-Learning in [33]). In other words, by exploiting these fundamental convergence arguments, generally showing convergence boils down to stability and contraction arguments, which are much simpler.

More recently, convergence arguments when the algorithm is combined with function approximation have been developed; [15] provides an up-to-date account and presents some of the open problems. The algorithms based on evolutionary search and learning automata have an independent convergence theory, which has been developed in depth in texts such as [24, 60].

13.9 Concluding Remarks

This chapter presented selected topics on RL relevant to simulation optimization. We began with Q-Learning, along with a discussion on how to combine it with a simple linear-basis function approximator, which is critical for large-scale problems

that simulation-based optimization seeks to solve. We also covered actor-critics, but only the most modern version. SARSA(λ) was covered in some detail, because although a heuristic, it remains popular and can be implemented in simulators. Thereafter, we presented algorithms belonging to the class called API. Our discussion included average reward and the finite-horizon problems, which are not covered in most texts in much detail. Extensions and convergence theory were briefly described. Here, we summarize some applications and some of the open problems that should be exciting topics for future research.

- *Applications*: Since this area has origins in machine learning, some of the initial applications were naturally in the area of robotics and computer science. Even today, these algorithms find applications in exciting areas in machine learning, e.g., autonomous helicopter control [1, 63] and fMRI studies [97]. However, the body of literature that applies these algorithms to industrial tasks is expanding. Some of the early applications in the area of operations management include jobshop scheduling [99], AGV routing [84], preventive maintenance [26], and airline revenue management [31, 40]. Some more recent work includes the beer game of supply chain management [22], wireless communication [3, 96], irrigation control [72], and reentrant semiconductor manufacturing [68].
- *Function approximation and convergence*: While many advances have been made in combining RL with function approximators, this is an area that needs significant additional research without which function approximation will remain the Achilles' heel of RL. Not surprisingly, recent coverage of RL seems to stress the function approximation aspects, many of which are heuristically applied without convergence analysis. The Bellman error development is also rather heuristic, and apart from the traditional techniques used in conjunction with Bellman error [6, 17, 18, 27, 89], recently other techniques have been sought to be used: see [64] (kernel-based function approximation), [92] (evolutionary function approximation), and [56] (Laplacian methods). Regardless of the nature of the function approximation technique used, a preliminary step in this process involves transforming the state space into the feature space, and some important references in this context include [43] (coarse coding), [4] (CMAC coding), and [50] (Kanerva coding). Another promising line of investigation includes the LSTD (least-squares temporal differences) algorithm [62, 98]. Much of this work has occurred along the lines of API, which also happens to be an active area of research [14]. See also [55] where a Q-function-based LSTD algorithm, which is suitable for a simulation-based setting, is presented, although the algorithm's convergence has never been proved.
- *TD(λ) in combination with Q-Learning*: Although the concept of TD(λ) has been around for a long time in this field, its convergence properties are known well for the case when it is combined with policy evaluation within an API algorithm. In API, whether $\lambda > 0$ provides any advantages over $\lambda = 0$ has not been tested empirically in a comprehensive manner; one clear disadvantage is that it requires eligibility traces that are not easily combined with function approximation. When combined with value iteration algorithms, such as Q-Learning (see Q(λ) learning

in [65]), the worth of TD(λ) becomes even more doubtful, since no convergence guarantees to optimality are available. However, in practice, Q(λ) appears to converge significantly faster than Q-Learning [65]. See [83] for an interesting result which shows that Q(λ) does converge but not necessarily to the optimal solution. It is clear that issues with respect to combining TD(λ) with algorithms such as Q-Learning and SARSA remain important open problems that require further investigation.

- *Optimistic API*: Although much research has occurred in classical versions of API, it is well-known that it is slow, since it requires numerous simulations to evaluate just one policy. As such, there is great interest in optimistic API in which the policy is evaluated via just one (or a few) state transition. The interest in API also stems from the empirical evidence suggesting that function approximation works better in conjunction with API rather than with value-iteration-based methods. Interestingly, the actor-critic, one of the earliest algorithms in RL [8,95], sought to attain the same objective. One recent algorithm in this direction is [13].
- *Model-building algorithms*: Model-building (also called model-based in the RL literature, but not to be confused with the same term in optimization) algorithms do not need the transition probability model, but build it within the simulator while simultaneously solving the Bellman equation. The earliest model-building algorithms for discounted and average reward are [7, 84], respectively. Other works on model building include [20, 28, 51, 59, 78, 81, 87] While most of these algorithms seek to generate the value function and are rooted in DP, some recent algorithms are based on Q-Learning and actor-critic frameworks [36, 41]. These algorithms have attracted significant interest recently in applications: robotic soccer [94], helicopter control [1, 54, 63], function magnetic imaging resonance (fMRI) studies of brain [47, 97], and vision [57].

References

1. P. Abbeel, A. Coates, T. Hunter, and A. Y. Ng. Autonomous autorotation of an RC helicopter. In *International Symposium on Robotics*, 2008.
2. J. Abounadi, D. Bertsekas, and V. S. Borkar. Learning algorithms for Markov decision processes with average cost. *SIAM Journal of Control and Optimization*, 40(3):681–698, 2001.
3. N. Akar and S. Sahin. Reinforcement learning as a means of dynamic aggregate qos provisioning. In *Lecture Notes in Computer Science, Volume 2698/2003*. Springer, Berlin/Heidelberg, 2003.
4. J. S. Albus. *Brain, Behavior and Robotics*. Byte Books, Peterborough, NH, 1981.
5. R. Askin and J. Goldberg. *Design and Analysis of Lean Production Systems*. Wiley, NY, 2002.
6. L. Baird. Residual algorithms: Reinforcement learning with function approximation. In *Proceedings of the Twelfth International Conference on Machine Learning*, pages 30–37. Morgan Kaufmann, 1995.
7. A. G. Barto, S. J. Bradtke, and S. P. Singh. Learning to act using real-time dynamic programming. *Artificial Intelligence*, 72:81–138, 1995.

8. A. G. Barto, R. S. Sutton, and C. W. Anderson. Neuronlike elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man, and Cybernetics*, 13:835–846, 1983.
9. J. Baxter and P. Bartlett. Infinite-horizon policy-gradient estimation. *Journal of Artificial Intelligence*, 15:319–350, 2001.
10. R. E. Bellman. The theory of dynamic programming. *Bull. Amer. Math. Soc*, 60:503–516, 1954.
11. R. E. Bellman. *Dynamic Programming*. Princeton University Press, Princeton, NJ, 1957.
12. D. Bertsekas and J. Tsitsiklis. *Neuro-Dynamic Programming*. Athena Scientific, Belmont, MA, USA, 1996.
13. D. Bertsekas and H. Yu. Q-learning and enhanced policy iteration in discounted dynamic programming. In *Proceedings of the 49th IEEE Conference on Decision and Control*, pages 1409–1416, 2010.
14. D. P. Bertsekas. Approximate policy iteration: A survey and some new methods. *Journal of Control Theory and Applications*, 9(3):310–335, 2011.
15. D. P. Bertsekas. *Dynamic Programming and Optimal Control*. Athena Scientific, Belmont, MA, USA, 4th edition, 2012.
16. V. S. Borkar. *Stochastic Approximation: A Dynamical Systems Viewpoint*. Hindusthan Book Agency, New Delhi, India, 2008.
17. J. A. Boyan and A. W. Moore. Generalization in Reinforcement Learning: Safely Approximating the Value Function. *Advances in Neural Information Processing Systems*, pages 369–376, 1995.
18. S. Bradtke and A. G. Barto. Linear least squares learning for temporal differences learning. *Machine Learning*, 22:33–57, 1996.
19. S. Bradtke and M. Duff. Reinforcement learning methods for continuous-time Markov decision problems. In *Advances in Neural Information Processing Systems 7*. MIT Press, Cambridge, MA, USA, 1995.
20. R. I. Brafman and M. Tennenholtz. R-max: A general polynomial time algorithm for near-optimal reinforcement learning. *Journal of Machine Learning Research*, 3:213–231, 2002.
21. X. R. Cao. *Stochastic Learning and Optimization: A Sensitivity-Based View*. Springer, New York, 2007.
22. S. K. Chaharsooghi, J. Heydari, and S. H. Zegordi. A reinforcement learning model for supply chain ordering management: An application to the beer game. *Decision Support Systems*, 45, 2008.
23. H. S. Chang, M. C. Fu, J. Hu, and S. I. Marcus. Recursive learning automata approach to Markov decision processes. *IEEE Transactions on Automatic Control*, 52(7):1349–1355, 2007.
24. H. S. Chang, M. C. Fu, J. Hu, and S. I. Marcus. *Simulation-based Algorithms for Markov Decision Processes*. Springer, 2007.
25. H. S. Chang, H. G. Lee, M. C. Fu, and S. I. Marcus. Evolutionary policy iteration for solving Markov decision processes. *IEEE Transactions on Automatic Control*, 50(11):1804–1808, 2005.
26. T. K. Das, A. Gosavi, S. Mahadevan, and N. Marchalleck. Solving semi-Markov decision problems using average reward reinforcement learning. *Management Science*, 45(4):560–574, 1999.
27. S. Davies. Multi-dimensional interpolation and triangulation for reinforcement learning. *Advances in Neural Information and Processing Systems*, 1996.
28. C. Diuk, L. Li, and B. Leffler. The adaptive k-meteorologists problem and its application to structure learning and feature selection in reinforcement learning. In *Proceedings of the 26th Annual International Conference on Machine Learning*, 2009.
29. J. Filar and K. Vrieze. *Competitive Markov Decision Processes*. Springer-Verlag, New York, NY, USA, 1997.
30. A. Gosavi. *Simulation-Based Optimization: Parametric Optimization Techniques and Reinforcement Learning*. Springer, Boston, 2003.

31. A. Gosavi. A reinforcement learning algorithm based on policy iteration for average reward: Empirical results with yield management and convergence analysis. *Machine Learning*, 55:5–29, 2004.
32. A. Gosavi. Reinforcement learning for long-run average cost. *European Journal of Operational Research*, 155:654–674, 2004.
33. A. Gosavi. Boundedness of iterates in *Q*-learning. *Systems and Control Letters*, 55:347–349, 2006.
34. A. Gosavi. A risk-sensitive approach to total productive maintenance. *Automatica*, 42:1321–1330, 2006.
35. A. Gosavi. Reinforcement learning: A tutorial survey and recent advances. *INFORMS Journal on Computing*, 21(2):178–192, 2009.
36. A. Gosavi. Reinforcement learning for model building and variance-penalized control. In *Proceedings of the 2009 Winter Simulation Conference*. IEEE, Piscataway, NJ, 2009.
37. A. Gosavi. Finite horizon Markov control with one-step variance penalties. In *Conference Proceedings of the Allerton Conference*. University of Illinois, USA, 2010.
38. A. Gosavi. Target-sensitive control of Markov and semi-Markov processes. *International Journal of Control, Automation, and Systems*, 9(5):1–11, 2011.
39. A. Gosavi. Approximate policy iteration for Markov control revisited. In *Procedia Computer Science, Complex Adaptive Systems, Chicago*. Elsevier, 2012.
40. A. Gosavi, N. Bandla, and T. K. Das. A reinforcement learning approach to a single leg airline revenue management problem with multiple fare classes and overbooking. *IIE Transactions*, 34(9):729–742, 2002.
41. A. Gosavi, S. Murray, J. Hu, and S. Ghosh. Model-building adaptive critics for semi-Markov control. *Journal of Artificial Intelligence and Soft Computing Research*, 2(1), 2012.
42. C. M. Grinstead and J. L. Snell. *Introduction to Probability*. American Mathematical Society, Providence, RI, 1997.
43. G. E. Hinton. Distributed representations. Technical Report, CMU-CS-84-157, Carnegie Mellon University, Pittsburgh, PA, USA, 1984.
44. J. H. Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, MI, USA, 1975.
45. R. Howard. *Dynamic Programming and Markov Processes*. MIT Press, Cambridge, MA, 1960.
46. J. Hu and M. P. Wellman. Nash Q-Learning for general-sum stochastic games. *Journal of Machine Learning Research*, 4:1039–1069, 2003.
47. S. Ishii, W. Yoshida, and J. Yoshimoto. Control of exploitation-exploration meta-parameter in reinforcement learning. *Neural Networks*, 15:665–687, 2002.
48. T. Jaakkola, M. Jordan, and S. Singh. On the convergence of stochastic iterative dynamic programming algorithms. *Neural Computation*, 6(6):1185–1201, 1994.
49. L. P. Kaelbling, M. L. Littman, and A. W. Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996.
50. P. Kanerva. *Sparse Distributed Memory*. MIT Press, Cambridge, MA, USA, 1988.
51. M. Kearns and S. P. Singh. Near-optimal reinforcement learning in polynomial time. *Machine Learning*, 49(2):209–232, 2002.
52. S. S. Keerthi and B. Ravindran. A tutorial survey of reinforcement learning. *Sadhana*, 19(6):851–889, 1994.
53. V. Konda and V. S. Borkar. Actor-critic type learning algorithms for Markov decision processes. *SIAM Journal on Control and Optimization*, 38(1):94–123, 1999.
54. R. Koppejan and S. Whiteson. Neuroevolutionary reinforcement learning for generalized helicopter control. In *GECCO: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 145–152, 2009.
55. M. Lagoudakis and R. Parr. Least-squares policy iteration. *Journal of Machine Learning Research*, 4:1107–1149, 2003.
56. S. Mahadevan. Learning representation and control in Markov decision processes: New frontiers. In *Foundations and Trends in Machine Learning, Vol I(4)*, pages 403–565. Now Publishers, 2009.

57. J. Michels, A. Saxena, and A. Y. Ng. High speed obstacle avoidance using monocular vision and reinforcement learning. In *Proceedings of the 22nd International Conference on Machine Learning, Bonn, Germany*, 2005.
58. T. M. Mitchell. *Machine Learning*. McGraw Hill, Boston, MA, USA, 1997.
59. A. Moore and C. Atkeson. Prioritized sweeping: Reinforcement learning with less data and less real time. *Machine Learning*, 13:103–130, 1993.
60. K. Narendra and M. Thathachar. *Learning Automata: An Introduction*. Prentice Hall, Englewood Cliffs, NJ, USA, 1989.
61. J. F. Nash. Equilibrium points in n-person games. *Proceedings, Nat. Acad. of Science, USA*, 36:48–49, 1950.
62. A. Nedić and D. P. Bertsekas. Least-squares policy evaluation with linear function approximation. *Discret-event Dynamic Systems: Theory and Applications*, 13:79–110, 2003.
63. A. Y. Ng, H. J. Kim, M. I. Jordan, and S. Sastry. Autonomous helicopter flight via reinforcement learning. In *Advances in Neural Information Processing Systems 17*. MIT Press, 2004.
64. D. Ormoneit and S. Sen. Kernel-based reinforcement learning. *Machine Learning*, 49(2–3):161–178, 2002.
65. J. Peng and R. J. Williams. Incremental multi-step Q-learning. In *Machine Learning*, pages 226–232. Morgan Kaufmann, 1996.
66. W. Powell. *Approximate Dynamic Programming: Solving the curses of dimensionality*. Wiley-Interscience, NJ, USA, 2007.
67. K. Rajaraman and P. Sastry. Finite time analysis of the pursuit algorithm for learning automata. *IEEE Transactions on Systems, Man, and Cybernetics: Part B*, 26(4):590–598, 1996.
68. J. A. Ramirez-Hernandez and E. Fernandez. A case study in scheduling re-entrant manufacturing lines: Optimal and simulation-based approaches. In *Proceedings of 44th IEEE Conference on Decision and Control*, pages 2158–2163. IEEE, 2005.
69. K. Ravulapati, J. Rao, and T. Das. A reinforcement learning approach to stochastic business games. *IIE Transactions*, 36:373–385, 2004.
70. H. Robbins and S. Monro. A stochastic approximation method. *Annals of Mathematical Statistics*, 22:400–407, 1951.
71. G. Rummery and M. Niranjan. On-line Q-learning using connectionist systems. Technical Report CUED/F-INFENG/TR 166. Engineering Department, Cambridge University, 1994.
72. N. Schutze and G.H.Schmitz. Neuro-dynamic programming as a new framework for decision support for deficit irrigation sytems. In *International Congress on Modelling and Simulation, Christchurch, New Zealand*, pages 2271–2277, 2007.
73. L. Shapley. Stochastic games. *Proceedings of the National Academy of Sciences*, 39:1095–1100, 1953.
74. J. Si, A. G. Barto, W. B. Powell, and D. Wunsch, editors. *Handbook of Learning and Approximate Dynamic Programming*. IEEE Press, Wiley, Hoboken, NJ, 2004.
75. S. Singh, T. Jaakkola, M. Littman, and C. Szepesvári. Convergence results for single-step on-policy reinforcement-learning algorithms. *Machine Learning*, 39:287–308, 2000.
76. S. Singh and R. Sutton. Reinforcement learning with replacing eligibility traces. *Machine Learning*, 22:123–158, 1996.
77. S. Singh, V. Tadic, and A. Doucet. A policy-gradient method for semi-Markov decision processes with application to call admission control. *European Journal of Operational Research*, 178(3):808–818, 2007.
78. A. Strehl and M. Littman. A theoretical analysis of model-based interval estimation. In *Proceedings of the 22th International Conference on Machine Learning*, pages 856–863, 2005.
79. R. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, Cambridge, MA, USA, 1998.
80. R. S. Sutton. Learning to predict by the method of temporal differences. *Machine Learning*, 3:9–44, 1988.
81. R. S. Sutton. Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In *Proceedings of the 7th International Workshop on Machine Learning*, pages 216–224. Morgan Kaufmann, San Mateo, CA, 1990.

82. C. Szepesvári. *Algorithms for Reinforcement Learning*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2010.
83. C. Szepesvári and M. Littman. A unified analysis of value-function-based reinforcement-learning algorithms. *Neural Computation*, 11(8):2017–2060, 1999.
84. P. Tadepalli and D. Ok. Model-based average reward reinforcement learning algorithms. *Artificial Intelligence*, 100:177–224, 1998.
85. M. Thathachar and P. Sastry. A class of rapidly converging algorithms for learning automata. *IEEE Transactions on Systems, Man, and Cybernetics*, 15:168–175, 1985.
86. J. E. E. van Nunen. A set of successive approximation methods for discounted Markovian decision problems. *Z. Operations Research*, 20:203–208, 1976.
87. H. van Seijen, S. Whiteson, H. van Hasselt, and M. Wiering. Exploiting best-match equations for efficient reinforcement learning. *Journal of Machine Learning Research*, 12:2045–2094, 2011.
88. C. J. Watkins. *Learning from Delayed Rewards*. PhD thesis, Kings College, Cambridge, England, 1989.
89. P. J. Werbös. Building and understanding adaptive systems: A statistical/numerical approach to factory automation and brain research. *IEEE Transactions on Systems, Man., and Cybernetics*, 17:7–20, 1987.
90. P. J. Werbös. Consistency of HDP applied to a simple reinforcement learning problem. *Neural Networks*, 3:179–189, 1990.
91. R. M. Wheeler and K. S. Narenda. Decentralized learning in finite Markov chains. *IEEE Transactions on Automatic Control*, 31(6):373–376, 1986.
92. S. Whiteson and P. Stone. Evolutionary function approximation for reinforcement learning. *Journal of Machine Learning Research*, 7:877–917, 2006.
93. B. Widrow and M. E. Hoff. Adaptive Switching Circuits. In *Institute of Radio Engineers, Western Electronic Show and Convention, Convention Record, Part 4*, pages 96–104. 1960.
94. M. A. Wiering, R. P. Salustowicz, and J. Schmidhuber. Model-based reinforcement learning for evolving soccer strategies. In *Computational Intelligence in Games*. Springer Verlag, 2001.
95. I. H. Witten. An adaptive optimal controller for discrete time Markov environments. *Information and Control*, 34:286–295, 1977.
96. W. Yeow, C. Tham, and W. Wong. Energy efficient multiple target tracking in wireless sensor networks. *IEEE Transactions on Vehicular Technology*, 56(2):918–928, 2007.
97. W. Yoshida and S. Ishii. Model-based reinforcement learning: A computational model and an fMRI study. *Neurocomputing*, 63:253–269, 2005.
98. H. Yu and D. P. Bertsekas. Convergence results on some temporal difference methods based on least squares. *IEEE Transactions on Automatic Control*, 54(7):1515–1531, 2009.
99. W. Zhang and T. G. Dietterich. A reinforcement learning approach to job-shop scheduling. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pages 1114–1120. Morgan Kaufmann, 1995.