

This material will be published by Cambridge University Press as “Markov Decision Processes and Reinforcement Learning” by Martin L. Puterman and Timothy C. Y. Chan. This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale, or use in derivative works. ©Martin L. Puterman and Timothy C. Y. Chan, 2025.

Part III - Reinforcement Learning

This material will be published by Cambridge University Press as “Markov Decision Processes and Reinforcement Learning” by Martin L. Puterman and Timothy C. Y. Chan. This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale, or use in derivative works. ©Martin L. Puterman and Timothy C. Y. Chan, 2025.

With four parameters I can fit an elephant, and with five I can make him wiggle his trunk.¹

John von Neumann, Hungarian-American mathematician, physicist, computer scientist, game theorist and more, 1903-1957.

This part of the book contains three chapters:

- Chapter 9: Value function approximation
- Chapter 10: Simulation in tabular models
- Chapter 11: Simulation with function approximation

The material in Chapter 9 is often referred to as *approximate dynamic programming (ADP)* and that in Chapters 10 and 11 as *reinforcement learning*.

While von Neumann’s quote was relevant for statistical models in the 1950s, things have changed. Many modern reinforcement learning methods, particularly those based on neural networks, use models with millions of parameters and still generalize well to new settings. This progress comes from advances in neural network techniques, access to massive datasets, and powerful optimization algorithms. As a result, the link between model complexity and performance is far more subtle than in von Neumann’s time.

Reinforcement learning can be viewed from two perspectives:

- The *artificial intelligence (AI)* perspective is that reinforcement learning methods seek to develop an “agent which can interpret any environment, and learn a task to superhuman ability, all with minimal user interaction².²”

¹Dyson [2004].

²McKenzie and McDonnell [2022]

- The *operations research (OR)* perspective is that in a model-based environment in which rewards and transition probabilities are known, reinforcement learning provides a toolbox of methods that can be used to find effective policies.

Although these appear to be distinct, the boundaries are far from clear cut in practice. Table 8.2 sheds light on how these perspectives have evolved in practice.

Aspect	OR perspective	AI perspective
Horizon	Infinite or finite and fixed	Finite and random
Model	Specified	Not specified
Rewards	At each decision epoch	At termination
Possible actions	Pre-specified	Pre-specified
Transition probabilities	Known	Not known
Algorithms	Offline	Online

Table 8.2: Distinguishing features of the OR and AI perspectives on reinforcement learning. These are not intended to be exhaustive or mutually exclusive, the boundary between them is often blurred.

Overview

Chapters 4 - 8 provided computational algorithms (e.g., value iteration, policy iteration, linear programming) for finding optimal policies in a fully or partially observable Markov decision process that can be stored on a computer. Since these methods are computationally prohibitive in models with extremely large (or continuous) state spaces, this section of the book and most current research develops and investigates methods to do so.

Motivated by the need to reduce the dimension of the state space, Chapter 9 provides generalizations of value iteration, policy iteration and linear programming for analyzing Markov decision processes in which the underlying value function is approximated by a low-dimensional linear or nonlinear function of features of the state space or state-action space. Such algorithms are applied directly to the Markov decision process model. No simulation is involved.

The penultimate chapter, Chapter 10, represents a major departure from previous chapters. In it, simulation (or real-time experimentation) replaces exact computation. It focuses on models that can be represented in *tabular* form. That is, models that are sufficiently small so that value functions and state-action value functions can be easily stored in memory. In such models, approximating value and state-action value functions is unnecessary. The reason for focusing on tabular models is to compare the performance of simulation to classical methods. Approaches include Monte Carlo methods, temporal differencing and Q-learning. This chapter also distinguishes *model-based* analyses, in which components of the underlying Markov decision process are used in algorithms, and *model-free* analyses, which are based only on the generated or

observed sequence of states, actions and rewards. The underlying mathematical tools are stochastic approximation and stochastic gradient descent.

The concluding chapter, Chapter 11, combines the concepts from the previous two chapters, namely value function or state-action value function approximation and simulation and introduces policy-based approaches as well. It is motivated by applications in which the state space (and/or action space) is so large (or continuous) that tabular representations and exact computation are impractical. The two distinct approaches are:

1. **Value-based methods** use the reward to update the stored state-action value function or an approximation to it. When rewards corresponding to an action are greater than expected, the state-action value function for the specified state-action pair is increased, making the chosen action in that state more attractive. Conversely, when the reward is less than expected, the state-action value function is decreased making the action less attractive. Thus, the rewards lead these algorithms to learn a good approximations to the “true” underlying state-action value function from which actions can be selected greedily in subsequent implementations.
2. **Policy-based methods** use the reward to directly update a randomized policy by gradient ascent. When an action chosen in a state produces a larger than expected reward, the probability of choosing that action in the state is increased. Conversely if the reward is less than expected, the probability of choosing that action is decreased. Repeating this process produces effective policies.

Objectives

This part of the book offers an accessible introduction to reinforcement learning concepts and algorithms, rather than a comprehensive (and soon outdated) overview. By mastering these fundamental methods, readers will establish a solid base from which to delve more deeply into rapidly evolving reinforcement learning research and applications.

So, how should one learn this material? Following the reinforcement learning philosophy, you must learn by doing. This means:

1. Carefully formulating your applications as Markov decision processes.
2. Developing your own code for the algorithms³ in these chapters.

Often your codes will not replicate published results. Unlike the methods in Chapters 4–8, reinforcement learning methods offer few practical guarantees on convergence and have many parameters to tune. Trial-and-error is a natural part of the process, and a necessity for developing the insights needed to create robust and effective algorithms.

³This is not recommended for linear programming and regression, where excellent commercial codes are available.



Figure III.1: Image a Brio Labyrinth owned by one of the authors of this book.

An example: The Brio labyrinth

To provide a flavor of the type of problem that the reinforcement learning community seeks to solve, consider the challenge of optimally guiding a ball through the Brio Labyrinth shown in Figure III.1. Bi and D'Andrea [2023] use reinforcement learning to find and implement a good strategy for playing this game. Details and methods are beyond the scope of this book but hopefully will be understandable after reading these chapters.

In this game, a player places a metal ball at a location designated as the start at the top center of the maze. Using the two knobs on the sides of the labyrinth, the player dynamically tilts the plane of the surface so as to guide the ball to a destination at the right center without falling through any of the 39 numbered holes⁴.

To “solve” this problem using reinforcement learning requires complex engineering, considerable ingenuity and the use of state-of-the-art algorithms. The researchers developed a robotic system that used computer-controlled motors attached to the rotors to adjust the tilt of the surface. A camera mounted above the surface provided real-time information about the ball position, surface angles and the geometry around the ball.

An MDP model

The system was modeled by a discrete-time undiscounted infinite horizon (episodic) MDP. States represent information about the ball, the surface, and the local geometry (position of walls and holes) around the ball. The ball position is given by its (x, y) -

⁴Holes may be numbered twice when they can interact with the ball at two different segments of its path.

coordinates. The surface is summarized in terms of the angle of tilt of the surface in the x and y directions. Rates of change of these quantities are estimated using sequences of images. The local geometry around the ball is captured by $6\text{ cm} \times 6\text{ cm}$ images centered at the ball, taken at discrete time steps and encoded in a 64×64 RGB image. Actions represent the angular velocity of the two motors attached to the knobs controlling the surface tilt.

Camera images also provided a basis for a reward structure that measures progress (in cms) along the directional path represented by the black line from the origin to the destination in Figure III.1. Note in episodic models such as this, it is customary to only receive a reward when achieving a goal or incur a penalty when terminating in an undesirable state. Such a reward structure presents challenges for reinforcement learning methods so the introduction of these intermediate rewards facilitates learning.

Algorithms and results

Good policies are found using a model-based actor-critic algorithm (see Section 11.7.1) called DreamerV3 [Hafner et al., 2025]. Transition probabilities (obtained from “a world model”), policies, and values are approximated by neural networks. Underlying symmetries in the board structure improve algorithmic efficiency.

The system learned to “solve” the game in 5 hours (1 million time steps) using real-world experiences. Running the resulting policy over 50 replicates resulted in a success rate (reaching the target location) of 76% and a completion time of 15.73 ± 0.36 seconds. To put this in perspective, the human record is 15.95 seconds.

Chapter 9

Value Function Approximation

This material will be published by Cambridge University Press as “Markov Decision Processes and Reinforcement Learning” by Martin L. Puterman and Timothy C. Y. Chan. This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale, or use in derivative works. ©Martin L. Puterman and Timothy C. Y. Chan, 2025.

Far better an approximate answer to the right question, which is often vague, than an exact answer to the wrong question . . .¹

John Tukey, American statistician, 1915-2000.

9.1 Introduction

In models with large state spaces or action sets or both, direct evaluation and storage of policy and optimal value functions or state-action value functions may be prohibitive. As Tukey suggests, approximations may be preferable in such instances. To address this issue, this chapter describes methods in which value functions are often approximated by lower-dimensional functions such as polynomials, splines or neural networks. Doing so presents several challenges that are discussed below. Approaches based on function approximations are often referred to as *approximate dynamic programming* (ADP).

As an example, consider the medical appointment scheduling model of Section 3.7, analyzed in Section 9.6 below. In it, the vector-valued state represents the number of booked appointments each day over a multi-day booking horizon and the number of requests for appointments of each type² arriving on a particular day. Actions represent the number of incoming appointment requests of each type to book in available appointment slots on each day in the booking horizon. The objective in this application

¹Tukey [1962].

²In the application that motivated development of these models, type is an urgency level determined by a medical professional. Each urgency type has a different wait time target.

is to find a policy for assigning appointment slots to incoming appointment requests that uses resources efficiently but meets wait time targets.

In a realistic application of this model, the booking horizon $N = 30$, the number of appointment types $K = 3$, the daily capacity $C = 20$ and the maximum number of daily arrivals of each type $M = 10$. Subject to this specification the model will have $(21)^{30} \times (11)^3$ states and as well, a very large number of actions. Clearly in such a model, calculation and storage of exact value functions or state-action value functions in tabular form would be impossible. This chapter offers alternatives.

To motivate the approach, let $\hat{v}_\lambda^*(s)$ denote an approximation to the optimal value function $v_\lambda^*(s)$ in an infinite horizon discounted model. This chapter considers approaches in which the approximation is a lower-dimensional *parametric* function of model characteristics referred to as *basis functions* or *features*. Instead of storing $v_\lambda^*(s)$ in tabular form, the methods herein store only the parameter estimates. Using this information, a “good” action in a specific state s' is chosen as follows. Evaluate $\hat{v}_\lambda^*(j)$ in states j that can be reached from s' in one transition and choose an action $a_{s'}^*$ greedily based on the one-step Bellman update

$$a_{s'}^* \in \arg \max_{a \in A_{s'}} \left\{ r(s', a) + \lambda \sum_{j \in S} p(j|s', a) \hat{v}_\lambda^*(j) \right\}. \quad (9.1)$$

Since in most applications, there are few successor states to s' , this calculation can be very efficient. Recall that this is equivalent³ to solving a one-period problem (see Chapter 2) with terminal reward $\hat{v}_\lambda^*(s)$. Note that such an approach was used in Section 8.4.5 in which the POMDP optimal value function was approximated by supports obtained on a small set of belief points.

As an alternative to approximating value functions, approximations can be applied to state-action value functions to obtain $\hat{q}^*(s, a)$. Then $a_{s'}^*$ can be determined from

$$a_{s'}^* \in \arg \max_{a \in A_{s'}} \{\hat{q}^*(s', a)\}.$$

Moreover, approximations can be also used to directly represent policies in terms of model states such as in the policy gradient methods of Section 11.5.

Note that a policy based on greedy action choice using an approximation to the value function need not be optimal. The quality of the action depends on the accuracy of the approximation.

Returning to the appointment scheduling problem, suppose one has an approximation $\hat{v}_\lambda^*(j)$ to the optimal value function. Instead of a finding and storing the decision rule for all states, an action is chosen as needed as follows:

1. Observe the system state s' , namely the number of appointments previously scheduled for each day in the booking horizon and the number of arrivals of each type.

³Some authors refer to this procedure as *rollout*.

2. Obtain an action to implement by evaluating the expression in braces in (9.1) for each $a \in A_{s'}$ and choosing the maximizing action,, $a_{s'}^*$.
3. Assign patients to future appointment dates according to $a_{s'}^*$.

In this case, as in most implementations of Markov decision processes, the optimal action in all states is not needed a priori; it can be determined on an as-needed basis using the above approach or one based on an approximation $\hat{q}^*(s, a)$. The advantage of using $\hat{q}^*(s, a)$ will become apparent in model-free implementations described in Chapters 10 and 11.

Questions arising are:

1. How to obtain good value function approximations?
2. How close to optimal is the stationary policy based on decision rule $\hat{d}^*(s) = a_s^*$?

Note that in most of the literature, approximation and simulation are discussed together. To identify issues particular to each, these discussions are separated. This chapter focuses on approximation. Chapter 10 discusses simulation methods. Then, Chapter 11 combines simulation and approximation, which is the cornerstone of *reinforcement learning*. Moreover, the focus in this chapter is on discounted models. Extensions to average and total reward models are left to the reader.

9.2 Approximating policy value functions

This section describes approaches for approximating value functions and subsequently generalizes them to state-action value functions. Moreover, the focus will be on approximating the expected discounted reward of a *fixed* stationary deterministic policy. The following example provides motivation.

Example 9.1. Recall that in the queuing service rate control model (Section 3.4.1) the state represents the number of jobs in the system and the action represents the chosen service probability. In each period a job arrives with probability $b = 0.2$ and the controller can choose among three service probabilities $a_1 = 0.2$, $a_2 = 0.4$ and $a_3 = 0.6$. In any period, there is either an arrival, a service or neither. There is a delay cost of $f(s) = s^2$ and a cost per period of serving at rate a_k is $m(a_k) = 5k^3$.

To illustrate approximations, the unbounded state space is truncated to $S = \{0, 1, \dots, 50\}$ and the discount rate is set to $\lambda = 0.98$. Approximations to the value function of stationary policy $\pi = d^\infty$, which uses the deterministic decision

rule

$$d(s) = \begin{cases} a_1 & \text{for } s < 20 \\ a_3 & \text{for } s \geq 20, \end{cases}$$

are shown.

Since the state space is ordered and interpretable, simple parametric functions of the state can be used as value function approximators. To investigate the quality of approximations, the exact policy value function^a is evaluated using methods of Chapter 5. Approximations to the exact policy value function based on linear, quadratic and exponential functions are considered. Parameter estimates for the polynomials are obtained using least squares linear regression, while those for the exponential function are obtained using nonlinear least squares. Regression is based on the data $\{(s, v(s)) | s \in S\}$ or a subset thereof. In this example, this data was obtained from exact methods. In practice the exact $v(s)$ will not be available.

Figure 9.1a shows the true value function and its linear and quadratic approximations. Clearly the linear fit is poor while the quadratic fit appears to be better. It is left as an exercise to evaluate the quality of cubic polynomial and spline approximations. The equations of the two fitted models are

$$\hat{v}_\lambda^\pi(s) = -7603.3 + 1320.9s$$

and

$$\hat{v}_\lambda^\pi(s) = 2096.3 + 133.2s + 23.78s^2.$$

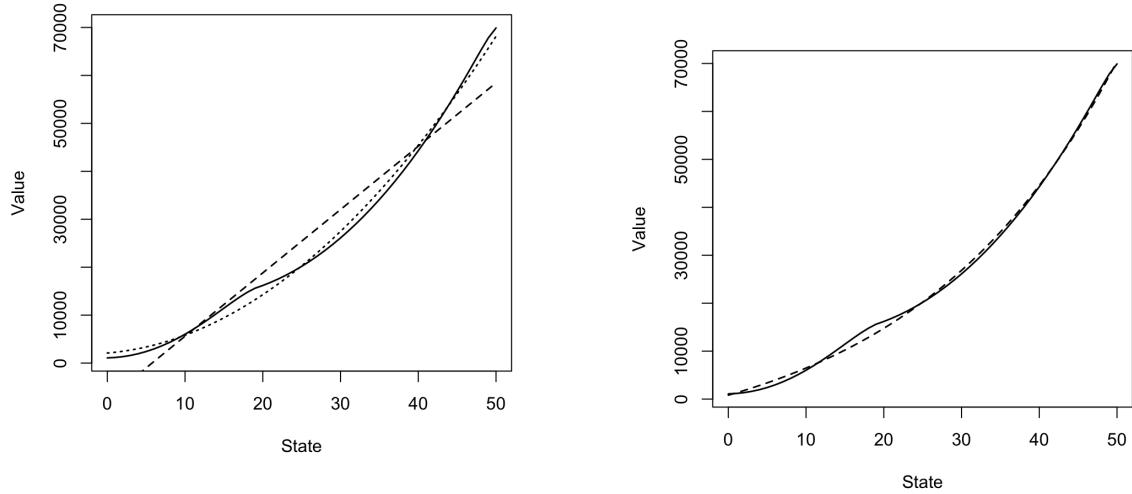
In addition the value function is approximated by the nonlinear function $\beta_0 + \beta_1 e^{\beta_2 s}$, which requires software that implements nonlinear least squares. The analysis here uses the *nls* package in R [R Core Team, 2024], which implements Gauss-Newton iteration to obtain parameter estimates. Given the sensitivity of nonlinear least squares to starting values, experimentation was required to obtain the fitted equation:

$$\hat{v}_\lambda^\pi(s) = -11403.2 + 12222.5e^{0.038s}.$$

Figure 9.1b shows the actual value (solid line) and fitted value (dashed line). It appears that the exponential fit is superior to the quadratic fit.

Of course, value function approximation is not necessary in such a small instance; the above calculations are provided to motivate for the material in this chapter. In practice $v_\lambda^\pi(s)$ would not be known for all $s \in S$, so approximation would be necessary. As an aside, if the focus was the countable state version of this model, the approximation on $s \leq 50$ could be extrapolated to $s > 50$.

^aOf course, if the exact policy value function is available, such as in this small example, there is no need to approximate it.



(a) Value function for d^∞ (solid line) with linear (dashed line) and quadratic (dotted line) approximations.

(b) Value function for d^∞ (solid line) with exponential approximation (dashed line).

Figure 9.1: Example 9.1 value functions and approximations.

9.2.1 Features

It is customary to use features of the state space as variables when approximating value functions. Feature generation is similar to the choice of independent variables in regression models. Features are usually low dimension summaries of key aspects of the state space that can be used to well approximate a value function. More formally:

Definition 9.1. A *feature* or *basis function* is a real-valued function defined on S .

Denote the set of features evaluated at s by the vector⁴ $\mathbf{b}(s) = (b_0(s), b_1(s), \dots, b_I(s))$. Often $b_0(s) := 1$ to correspond to a constant in a regression model.

For $S = \{s_1, s_2, \dots, s_M\}$ define the $M \times (I + 1)$ matrix \mathbf{B} of features by

$$\mathbf{B} := \begin{bmatrix} b_0(s_1) & \dots & b_I(s_1) \\ \vdots & \ddots & \vdots \\ \vdots & \dots & \vdots \\ b_0(s_M) & \dots & b_I(s_M) \end{bmatrix}. \quad (9.2)$$

⁴Recall that Chapter 8 used the vector \mathbf{b} to denote a belief state. Hopefully this will not cause confusion.

Assume that features are constructed so that the columns of \mathbf{B} are linearly independent (in the linear algebra sense⁵) so that the matrix $\mathbf{B}^T \mathbf{B}$ is invertible.

Specifying features

Feature choice depends on the setting:

1. If the state space is a set of **real numbers or ordered integers** possible feature choices include powers of the state, B-spline components, trigonometric functions or exponentials. In Example 9.1 when using a quadratic value function approximation, $I = 2$ and $b_0(s) = 1$, $b_1(s) = s$ and $b_2(s) = s^2$ for all $s \in S$.
2. If the state space is **vector-valued**, possible feature choices include component values, powers of component values, interactions between components or nonlinear functions of the component values. For example in the appointment scheduling model described in the introduction to this chapter when there is a five-day booking window and two appointment types the state space is

$$S = \{(x_1, x_2, \dots, x_5, y_1, y_2) \mid 0 \leq x_i \leq C, i = 1, \dots, 5; 0 \leq y_k \leq L, k = 1, 2\},$$

where C denotes the maximum daily capacity and L denotes the maximum number of appointments of each type that can arrive on a given day.

A possible set of features is:

$$\begin{aligned} b_0((x_1, x_2, \dots, x_5, y_1, y_2)) &= 1; \\ b_i((x_1, x_2, \dots, x_5, y_1, y_2)) &= x_i \quad \text{for } i = 1, \dots, 5; \\ b_{i,j}((x_1, x_2, \dots, x_5, y_1, y_2)) &= x_i x_j \quad \text{for } i = 1, \dots, 5, j = 1, \dots, 5 \quad \text{and} \\ b'_k((x_1, x_2, \dots, x_5, y_1, y_2)) &= y_k \quad \text{for } k = 1, 2. \end{aligned}$$

These features correspond to a quadratic model in the number of booked appointments and a linear function of the number of appointment requests.

3. In **more general settings**, application and subject matter knowledge dictates feature choice. For example, in checkers⁶ (Figure 9.2) there are 32 squares on the board that can be occupied (numbered in some way) and the two players, referred to as “Red” and “Black”, each start with 12 pieces. Thus

$$S = \left\{ (x_1, \dots, x_{32}) \mid x_i \in \{E, B, R, BK, RK\} \text{ for } i = 1, \dots, 32 \text{ and} \right. \\ \left. \sum_{i=1}^{32} I_{\{B, BK\}}(x_i) \leq 12, \sum_{i=1}^{32} I_{\{R, RK\}}(x_i) \leq 12 \right\},$$

⁵This means that the only solution of $\mathbf{B}\mathbf{x} = \mathbf{0}$ is $\mathbf{x} = \mathbf{0}$. Equivalently the rank of \mathbf{B} is $I + 1$ (since it is assumed $I + 1 < M$).

⁶Checkers is a two-person game but it can be analyzed by *self play*, that is, playing against yourself.



Figure 9.2: Checkerboard showing positions of pieces at the start of a game. Credit: gmatsuno/E+ via Getty Images.

where $I_A(x)$ is the indicator function for $x \in A$, E indicates that the square is empty, B denotes it is occupied by a single black checker, R denotes it is occupied by a single red checker, BK denotes it is occupied by a black king, and RK denotes it is occupied by a red king. The summations indicate that there are at most 12 pieces of each color on the board. Since $|S| = 5^{32}$, approximations are necessary to analyze this game.

Possible features include

$$\begin{aligned} b_0((x_1, \dots, x_{32})) &= 1; \\ b_1((x_1, \dots, x_{32})) &= \sum_{i=1}^{32} I_{\{B\}}(x_i); \quad b_2((x_1, \dots, x_{32})) = \sum_{i=1}^{32} I_{\{BK\}}(x_i); \\ b_3((x_1, \dots, x_{32})) &= \sum_{i=1}^{32} I_{\{R\}}(x_i); \quad b_4((x_1, \dots, x_{32})) = \sum_{i=1}^{32} I_{\{RK\}}(x_i). \end{aligned}$$

The above features can be interpreted as follows: $b_1(\cdot)$ and $b_3(\cdot)$ denote the number of single checkers of each color and $b_2(\cdot)$ and $b_4(\cdot)$ denote the number of kings of each color. Other possible features include the number of pieces of each color that are within κ rows of becoming kings for specific choices of κ .

In summary, possible features include:

1. the function 1,

2. powers and cross products of state variables,
3. expressions that interpolate between designated states, and
4. indicator functions of subsets of states.

Choosing the set of features equal to the indicator of each state, that is,

$$b_i(s) = \begin{cases} 1 & \text{if } s = s_i, \\ 0 & \text{otherwise} \end{cases}$$

for $i = 1, \dots, M$, is equivalent to a *tabular* representation of the model. This means that expressing the value function as a linear combination of such indicator functions is equivalent to using the standard Markov decision process formulation in terms of $v(s_i)$ for $i = 1, \dots, M$. In this case the methods of earlier chapters apply directly.

9.2.2 Feature-based value function approximators

This section proposes some value function approximations based on features. These approximations may be linear or nonlinear with neural networks representing an important and widely used class of nonlinear approximations. Some authors refer to the specified functional form as an *architecture*.

Linear architectures

The expression *linear approximation*⁷, refers to a value function approximation of the form:

$$v(s) \approx \beta_0 b_0(s) + \beta_1 b_1(s) + \dots + \beta_I b_I(s), \quad (9.3)$$

where $b_0(s) = 1$ for all $s \in S$ so that β_0 denotes the model constant. With judicious choice of features, such models can represent a wide range of functional forms. Advantages of linear approximations include the ability to estimate parameters using least squares regression (i.e., where estimates are available in closed form; see the chapter appendix) and having a functional form that can be directly used in a linear programming model. Moreover, when meaningful, the parameters can be interpreted as marginal values or costs.

Linear approximations can be nicely represented using matrix notation as follows:

$$\mathbf{v} \approx \mathbf{B}\boldsymbol{\beta}, \quad (9.4)$$

where

$$\mathbf{B} = \begin{bmatrix} b_0(s_1) & \dots & b_I(s_1) \\ \vdots & \ddots & \vdots \\ \vdots & \dots & \vdots \\ b_0(s_M) & \dots & b_I(s_M) \end{bmatrix} \quad \text{and} \quad \boldsymbol{\beta} = \begin{bmatrix} \beta_0 \\ \vdots \\ \beta_I \end{bmatrix} \quad (9.5)$$

⁷This may also be referred to as an *affine* approximation. Precisely speaking, the expression affine is more accurate when the approximation includes a constant term.

and $S = \{s_1, \dots, s_M\}$. In statistical literature, the matrix \mathbf{B} is often referred to as the *design matrix*⁸.

Note that the above expression does not explicitly include an error term ϵ because the models in this chapter do not involve statistical sampling. The only source of imprecision is the inaccuracy of the approximation.

Nonlinear architectures

Although in many applications, a linear approximation will suffice, a nonlinear function f of features can also be used to represent value functions. In general, nonlinear architectures can be written as

$$v(s) \approx f(b_0(s), b_1(s), \dots, b_I(s)|\boldsymbol{\beta}), \quad (9.6)$$

where the unknown vector of parameters in state s is represented by $\boldsymbol{\beta}$. In nonlinear models, the number of parameters need not equal the number of features used. That is, the dimension of $\boldsymbol{\beta}$ need not be $I + 1$. For example, with two features $b_0(s)$ and $b_1(s)$ a possible choice for $f(\cdot)$ is

$$f(b_0(s), b_1(s)|\boldsymbol{\beta}) = \beta_0 b_0(s) + \beta_1 e^{\beta_2 b_1(s)}$$

where $\boldsymbol{\beta} = (\beta_0, \beta_1, \beta_2)$. In this case there are two features and three parameters. In the special case when $b_0(s) = 1$ and $b_1(s) = s$ for all $s \in S$, the above approximation becomes

$$f(s|\boldsymbol{\beta}) = \beta_0 + \beta_1 e^{\beta_2 s},$$

which is the same as the exponential function from Example 9.1.

Neural networks

Artificial neural networks or neural networks (NNs)⁹ have been widely used in large modern applications of Markov decision processes. In the Markov decision process context they transform states or features into approximate value functions, state-action value functions or even the probability of selecting an action. However they also have been used in numerous significant machine learning implementations including image recognition, recommendation systems, forecasting, and natural language processing.

The most basic type of neural network is a *feedforward neural network* or FNN. A FNN consists of *nodes* organized in *layers*, uni-directional *arcs* connecting nodes from “left” to “right” and *activation functions* corresponding to nodes in interior *hidden* layers. Layers are classified as input layers, hidden layers and output layers. Figure 9.3 depicts a feedforward neural network with an input layer, two hidden layers and an output layer.

⁸The expression *design matrix* originates in the experimental design literature where the analyst chooses the allocation of treatments to experimental units. This allocation can be summarized in terms of a matrix.

⁹Some authors refer to neural networks as *multi-layer perceptrons*.

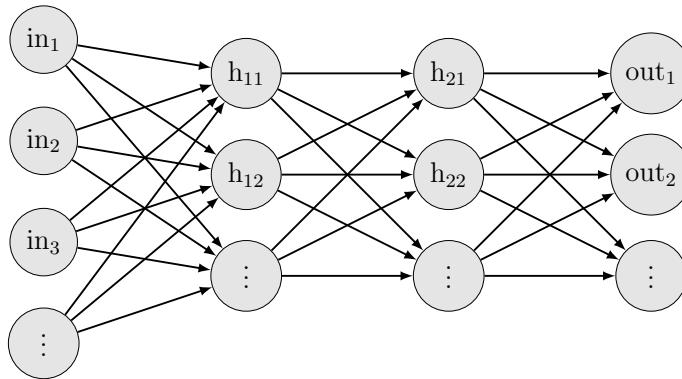


Figure 9.3: A generic feedforward neural network. The nodes labeled in_j denote nodes in the input layer, those labeled $\text{h}_{i,j}$ denote nodes in hidden layer j , and those labeled out_j denote nodes in the output layer.

The FNN processes information as follows: Values in the input layer are weighted by parameters corresponding to each arc and summed to generate inputs to the nodes at next layer where they are transformed nonlinearly by the activation function to produce outputs. These outputs are then weighted and summed and fed to the next layer, and so on.

Activation functions mimic the behavior of neurons that “fire” when the incoming signal reaches a threshold. They can be modeled by a step function that equals 0 for inputs less than a threshold and 1 for inputs greater than the threshold. Alternatively they can be represented by non-decreasing continuous functions $f : \mathbb{R} \rightarrow [0, 1]$ such as logistic or sigmoidal functions.

Other types of neural networks include:

1. **recurrent neural networks**, which generate outputs by combining inputs with previous hidden states, and
2. **convolutional neural networks**, which include filters that extract features from grid-like data such as camera images.

Details of these models are beyond the scope of this book but are important for applications in robotics and autonomous vehicle guidance. From the perspective of this chapter, neural networks are viewed as “black box” nonlinear function approximators.

An example

This example applies a single-layer FNN to the queuing service rate control example analyzed in Example 9.1 and discusses some practical issues arising when applying neural networks.

Example 9.2. Consider the queuing service rate control model analyzed in Example 9.1. This example approximates the value function of the specified stationary policy with $\lambda = 0.98$ using the FNN represented in Figure 9.4. This network consists of an input layer, a hidden layer with two nodes plus a node denoted by 1, which is often referred to as an *offset* or *bias*, and a single node in the output layer. The objective is to estimate parameters of this neural network to accurately approximate the value function. This may be regarded as using a neural network to perform nonlinear regression with features $b_0(s) = 1$ and $b_1(s) = s$.

In the hidden layer, the FNN model transforms a linear combination of inputs using the activation function

$$f(x) = \frac{1}{1 + e^{-x}} \quad (9.7)$$

and outputs the value function approximation using a linear combination of outputs from the hidden layer plus a constant. The model has seven parameters, one corresponding to each arc denoted by $w_{i,j}^k$, where i denotes the node at the start of the arc, j denotes the destination of the arc and $k = 2$ or 3 denotes the layer of the destination.

The FNN may be expressed as a single function as follows. The input to h_1 in the hidden layer is a linear function of the inputs 1 and s

$$w_{11}^1 + w_{21}^1 s$$

and the input to node 2 in the hidden layer is

$$w_{12}^1 + w_{22}^1 s.$$

The activation function transforms these inputs to

$$\frac{1}{1 + e^{-(w_{11}^1 + w_{21}^1 s)}} \quad \text{and} \quad \frac{1}{1 + e^{-(w_{12}^1 + w_{22}^1 s)}}.$$

Finally the input to the output layer can be represented by the linear function of outputs from the hidden layer as follows:

$$w_{01}^2 1 + w_{11}^2 \frac{1}{1 + e^{-(w_{11}^1 + w_{21}^1 s)}} + w_{21}^2 \frac{1}{1 + e^{-(w_{12}^1 + w_{22}^1 s)}}. \quad (9.8)$$

Note that this function contains the constant w_{01}^2 . In light of the von Neumann quote in the Part III introduction, one would expect an accurate approximation of the value function using this architecture.

The *nnet* package in R produced the parameter estimates shown on the arcs in Figure 9.4. Achieving reliable estimates required scaling both the state and the

value function to the interval $[0, 1]$ prior to training, followed by a reverse transformation to the original scale. Furthermore, the inclusion of a small amount of random noise during training proved beneficial. Without scaling, the estimation process produced unreliable results.

Figure 9.5 compares the true value function (solid line) and its FNN approximation (dashed line). The RMSE was 395 and while the fit looks good, the residuals had a sinusoidal pattern suggesting a systematic lack of fit. Ultimately, the quality of the approximation is determined based on the policy derived from the approximate value function.

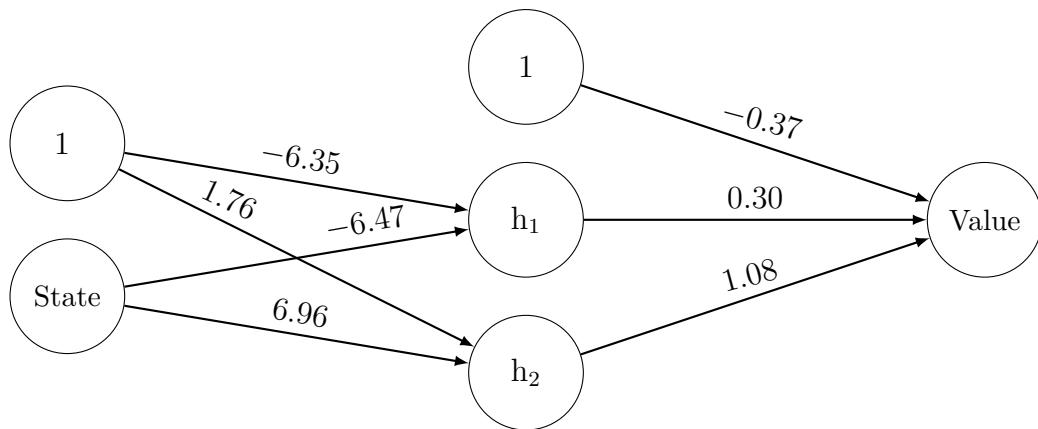


Figure 9.4: Feedforward neural network used in queuing service rate control approximation. Values on the arcs are estimated weights (parameter estimates).

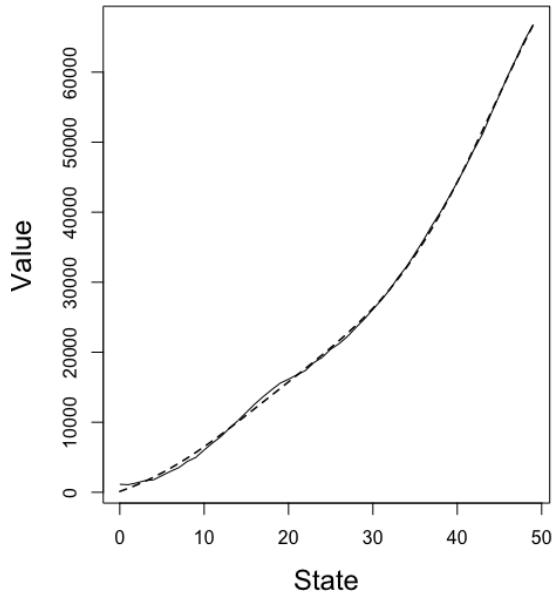


Figure 9.5: FNN approximation to the queuing system service rate control value function for the specified stationary policy. Solid line is the true value function while the dashed line is the approximation based on the FNN.

9.2.3 State-action value functions

The above sections describe how to approximate value functions using basis functions and linear or nonlinear architectures. Alternatively these constructs can be used to approximate state-action value functions $q(s, a)$. While such an approach may be of use in reinforcement learning environments when the model components are unknown, in this chapter, which assumes knowledge of $r(s, a)$ and $p(j|s, a)$, it is preferable to approximate $q(s, a)$ for all $a \in A_s$ by

$$\hat{q}(s, a) := r(s, a) + \lambda \sum_{j \in S} p(j|s, a) \hat{v}(j),$$

where $\hat{v}(s)$ is a value function approximation based on basis functions. Thus, the focus of this chapter will be on value function approximations.

9.3 Parameter estimation for policy value functions: Linear architectures

This section provides methods for finding good approximations to the value function of a *fixed* stationary policy in the form of linear combinations of basis functions. The

proposed methods are based on least squares estimation and linear programming. The next section considers nonlinear approximations. We strongly recommend referring to the Appendix of this chapter, which reviews least squares regression emphasizing matrix methods.

Motivation

Let d^∞ denote a stationary randomized policy. To determine its value, $\mathbf{v}_\lambda^{d^\infty}$, involves solving the following linear system of $|S|$ equations, which appeared previously in vector form in (5.31) as

$$\mathbf{v} = \mathbf{r}_d + \lambda \mathbf{P}_d \mathbf{v} = L_d \mathbf{v}. \quad (9.9)$$

Recall that when $\lambda < 1$, this system has a unique solution that can be represented by

$$\mathbf{v}_\lambda^{d^\infty} = (\mathbf{I} - \lambda \mathbf{P}_d)^{-1} \mathbf{r}_d. \quad (9.10)$$

Using matrix notation, the general form of a policy value function estimator¹⁰ based on a linear combination of basis functions can be written as $\mathbf{v} = \mathbf{B}\boldsymbol{\beta}$, where the columns of \mathbf{B} are vectors of basis functions evaluated for each $s \in S$. That is, a row of \mathbf{B} is of the form $(b_0(s), \dots, b_I(s))$ for some $s \in S$. Let $\text{col}(\mathbf{B})$ denote the subspace of \mathbb{R}^M spanned by the columns of \mathbf{B} , that is, it contains all real-valued functions that can be written as linear combinations of basis functions.

The benefit of seeking approximations in $\text{col}(\mathbf{B})$ is that there are $I + 1$ parameters to determine, as opposed to $|S|$. In applications where approximation is necessary, I will be orders of magnitude smaller than $|S|$. But replacing exact calculation by an approximation introduces *approximation error*. Moreover the methods below seek to do this without first computing $\mathbf{v}_\lambda^{d^\infty}$.

Least squares estimation: $\mathbf{v}_\lambda^{d^\infty}$ known

To motivate approximation methods and introduce some notation, assume first that $\mathbf{v}_\lambda^{d^\infty}$ has been evaluated¹¹ and one seeks a good approximation $\text{col}(\mathbf{B})$. Thus one wants to find

$$\hat{\boldsymbol{\beta}} \in \arg \min_{\boldsymbol{\beta} \in \mathbb{R}^{I+1}} \|\mathbf{B}\boldsymbol{\beta} - \mathbf{v}_\lambda^{d^\infty}\|, \quad (9.11)$$

where $\|\cdot\|$ is either the Euclidean norm

$$\|\mathbf{v}\|_2 := \left(\sum_{s \in S} v(s)^2 \right)^{\frac{1}{2}} = (\mathbf{v}^\top \mathbf{v})^{\frac{1}{2}}$$

¹⁰The statistical expression *estimator* is used, even though there is no randomness in the model.

¹¹Of course if this quantity were known, there would no need to approximate it unless one intended to interpolate or extrapolate, as is the case when truncating a countable state model.

or the weighted Euclidean norm

$$\|\mathbf{v}\|_2^w := \left(\sum_{s \in S} w(s) v(s)^2 \right)^{\frac{1}{2}} = (\mathbf{v}^\top \mathbf{W} \mathbf{v})^{\frac{1}{2}}, \quad (9.12)$$

where $w(s) > 0$ for all $s \in S$ and \mathbf{W} is a diagonal matrix with entries $w(s)$ on the diagonal.

The ordinary least squares (OLS) parameter and policy value function estimates (equations (9.68) and (9.70) in the chapter Appendix) become

$$\hat{\boldsymbol{\beta}}_{\text{OLS}} = (\mathbf{B}^\top \mathbf{B})^{-1} \mathbf{B}^\top \mathbf{v}_\lambda^{d^\infty} = \boldsymbol{\Gamma} \mathbf{v}_\lambda^{d^\infty} = \boldsymbol{\Gamma} (\mathbf{I} - \lambda \mathbf{P}_d)^{-1} \mathbf{r}_d \quad (9.13)$$

and

$$\hat{\mathbf{v}}_{\text{OLS}} = \mathbf{B} (\mathbf{B}^\top \mathbf{B})^{-1} \mathbf{B}^\top \mathbf{v}_\lambda^{d^\infty} = \boldsymbol{\Pi} \mathbf{v}_\lambda^{d^\infty} = \boldsymbol{\Pi} (\mathbf{I} - \lambda \mathbf{P}_d)^{-1} \mathbf{r}_d, \quad (9.14)$$

respectively, where

$$\boldsymbol{\Gamma} := (\mathbf{B}^\top \mathbf{B})^{-1} \mathbf{B}^\top \quad \text{and} \quad \boldsymbol{\Pi} := \mathbf{B} \boldsymbol{\Gamma} = \mathbf{B} (\mathbf{B}^\top \mathbf{B})^{-1} \mathbf{B}^\top. \quad (9.15)$$

The matrix $\boldsymbol{\Pi}$ is the projection of \mathbb{R}^M onto the $(I + 1)$ -dimensional subspace $\text{col}(\mathbf{B})$. Note that $\boldsymbol{\Gamma} \mathbf{v}$ is a vector of parameters and $\boldsymbol{\Pi} \mathbf{v}$ is a vector of values.

Note that this approach was used to illustrate concepts earlier in Example 9.1. It is often referred to as a *direct method*.

Direct methods will not be used in practice since the goal of approximate dynamic programming is to approximate the value function without first computing it.

Least squares estimation: $\mathbf{v}_\lambda^{d^\infty}$ unknown

In contrast to the direct method described above, *indirect methods* approximate a value function by appealing to the Bellman equation and a variant. Suppose \mathbf{v}' is an estimate of $\mathbf{v}_\lambda^{d^\infty}$. Then its accuracy can be measured with respect to the Bellman equation according to

$$\|\mathbf{v}' - L_d \mathbf{v}'\| \quad (9.16)$$

or with respect to the *projected Bellman evaluation equation*

$$\mathbf{v} = \boldsymbol{\Pi} L_d \mathbf{v} \quad (9.17)$$

and the corresponding error

$$\|\mathbf{v}' - \boldsymbol{\Pi} L_d \mathbf{v}'\|, \quad (9.18)$$

where as above, the norm can be either the Euclidean or weighted Euclidean norm.

Thus, setting $\mathbf{v}' = \mathbf{B} \boldsymbol{\beta}'$, the two above approaches for choosing a value of $\boldsymbol{\beta}'$ lead to:

1. Bellman¹² residual minimization: Choose $\hat{\beta}_{\text{BR}}$ according to

$$\hat{\beta}_{\text{BR}} \in \arg \min_{\beta \in \mathbb{R}^{I+1}} \|\mathbf{B}\beta - L_d \mathbf{B}\beta\| \quad (9.19)$$

2. Projection error minimization: Choose $\hat{\beta}_{\text{PE}}$ according to

$$\hat{\beta}_{\text{PE}} \in \arg \min_{\beta \in \mathbb{R}^{I+1}} \|\mathbf{B}\beta - \Pi L_d \mathbf{B}\beta\|, \quad (9.20)$$

where as above, the norms may be either the Euclidean or weighted Euclidean norm.

Geometric interpretation

Geometrically these two approaches minimize different quantities as shown in Figure 9.6. The Bellman residual minimization approach finds a value for β that minimizes the distance between $L_d \mathbf{B}\beta$ and $\mathbf{B}\beta$. In this case $L_d \mathbf{B}\beta$ need not be in $\text{col}(\mathbf{B})$. On the other hand the projection error minimization approach minimizes the difference between $\mathbf{B}\beta$ and $\Pi L_d \mathbf{B}\beta$, where both expressions lie in $\text{col}(\mathbf{B})$. Note that least squares regression theory (see chapter appendix) establishes that $\Pi L_d \mathbf{B}\beta$ represents the element of $\text{col}(\mathbf{B})$ closest to $L_d \mathbf{B}\beta$ in the least squares sense.

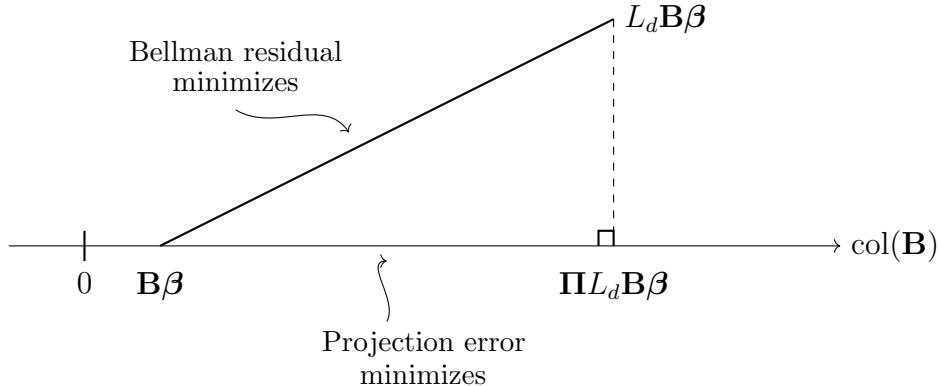


Figure 9.6: Graphical comparison of two approaches for estimating β . This figure assumes that \mathbf{B} contains a single column (corresponding to a single basis function) so that $\text{col}(\mathbf{B})$ is a one-dimensional subspace of \mathbb{R}^2 .

In principle, these two estimates can be found by varying β so as to make each of the distances as small as possible. In the one-dimensional example shown in Figure 9.6, this would correspond to altering the value of the scalar β , or equivalently sliding $\mathbf{B}\beta$ along the horizontal axis and noting how $\|\mathbf{B}\beta - L_d \mathbf{B}\beta\|$ and $\|\mathbf{B}\beta - \Pi L_d \mathbf{B}\beta\|$ change.

To distinguish these methods and gain familiarity with this notation, consider the following example.

¹²Note that as stated here, this approach corresponds to minimizing the error using the policy evaluation equation, which is a special case of the Bellman equation.

Example 9.3. Let $S = \{1, 2\}$,

$$\mathbf{r}_d = \begin{bmatrix} 2 \\ 8 \end{bmatrix} \quad \text{and} \quad \mathbf{P}_d = \begin{bmatrix} 0.25 & 0.75 \\ 0.10 & 0.90 \end{bmatrix}.$$

Choose a single basis function $b(s) = s$ so that $\mathbf{B} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$. Consequently β is a scalar denoted by β so that approximation of $v_\lambda^{d\infty}(s)$ is of the form

$$\mathbf{B}\beta = \begin{bmatrix} \beta \\ 2\beta \end{bmatrix}.$$

The Bellman residual error is given by

$$\|\mathbf{B}\beta - L_d \mathbf{B}\beta\| = \left\| \begin{bmatrix} \beta \\ 2\beta \end{bmatrix} - \left(\begin{bmatrix} 2 \\ 8 \end{bmatrix} + \lambda \begin{bmatrix} 1.75\beta \\ 1.9\beta \end{bmatrix} \right) \right\| = \left\| \begin{bmatrix} (1 - 1.75\lambda)\beta - 2 \\ (2 - 1.90\lambda)\beta - 8 \end{bmatrix} \right\|.$$

On the other hand noting that

$$\boldsymbol{\Pi} = \mathbf{B}(\mathbf{B}^\top \mathbf{B})^{-1} \mathbf{B}^\top = \frac{1}{5} \begin{bmatrix} 1 & 2 \\ 2 & 4 \end{bmatrix},$$

the least squares projection error is given by

$$\left\| \begin{bmatrix} \beta \\ 2\beta \end{bmatrix} - \frac{1}{5} \begin{bmatrix} 1 & 2 \\ 2 & 4 \end{bmatrix} \left(\begin{bmatrix} 2 \\ 8 \end{bmatrix} + \lambda \begin{bmatrix} 1.75\beta \\ 1.90\beta \end{bmatrix} \right) \right\| = \left\| \begin{bmatrix} (1 - 1.11\lambda)\beta - 2.4 \\ (2 - 2.22\lambda)\beta - 4.8 \end{bmatrix} \right\|.$$

Setting $\lambda = 0.6$ and using the Euclidean norm gives

$$\mathbf{v}_\lambda^{d\infty} = \begin{bmatrix} 18.40 \\ 25.10 \end{bmatrix}, \quad \boldsymbol{\Pi} \mathbf{v}_\lambda^{d\infty} = \begin{bmatrix} 10.09 \\ 20.18 \end{bmatrix}, \quad \mathbf{B}\hat{\beta}_{\text{BR}} = \begin{bmatrix} 9.14 \\ 18.27 \end{bmatrix} \quad \text{and} \quad \mathbf{B}\hat{\beta}_{\text{PE}} = \begin{bmatrix} 10.78 \\ 21.55 \end{bmatrix}.$$

Because the value function is **not** in $\text{col}(\mathbf{B})$ neither value function estimate accurately approximates $\mathbf{v}_\lambda^{d\infty}$. However, the estimate based on projection error minimization provides a better estimate of the projection of the value function on $\text{col}(\mathbf{B})$ as represented by $\boldsymbol{\Pi} \mathbf{v}_\lambda^{d\infty}$.

9.3.1 Least squares policy evaluation

This section describes a flexible iterative algorithm for projection error minimization referred to as *least squares policy evaluation (LSPE)*. It is based on iteratively solving the *projected* policy evaluation equation $\mathbf{v} = \boldsymbol{\Pi} L_d \mathbf{v}$. Its convergence depends on contraction properties of $\boldsymbol{\Pi} L_d$. Bellman residual minimization will be briefly discussed in a subsequent starred section and not be explored beyond that.

The following iterative algorithm (stated in matrix form) corresponds to applying value iteration to a fixed decision rule. Its beauty is that it:

1. is intuitive,
2. avoids computation of $\mathbf{v}_\lambda^{d\infty}$,
3. can be applied on a fixed or randomly generated subset of S as described below, and
4. easily generalizes to nonlinear architectures and optimization problems.

Note that the algorithm is structured so that value function approximations remain in $\text{col}(\mathbf{B})$.

Algorithm 9.1. Least squares policy evaluation: linear architectures

1. **Initialize:** Specify $\boldsymbol{\beta}$ arbitrary, $\epsilon > 0$, and $\Delta > \epsilon$.
2. **Iterate:** While $\Delta \geq \epsilon$:
 - (a) $\mathbf{v}' \leftarrow \mathbf{B}\boldsymbol{\beta}$.
 - (b) $\mathbf{v} \leftarrow \mathbf{r}_d + \lambda \mathbf{P}_d \mathbf{v}'$.
 - (c) $\boldsymbol{\beta}' \leftarrow \boldsymbol{\Gamma} \mathbf{v}^a$ ^a
 - (d) $\Delta \leftarrow \|\boldsymbol{\beta}' - \boldsymbol{\beta}\|_2$.
 - (e) $\boldsymbol{\beta} \leftarrow \boldsymbol{\beta}'$.
3. **Terminate:** Return $\boldsymbol{\beta}$.

^aIn practice this step would use a regression routine such as *lm* in R to implement.

Closed form representation of the LSPE estimate

The parameter estimate generated by LSPE can be represented in closed form as follows. Starting with $\boldsymbol{\beta}$, 2(a) generates the approximate value function $\mathbf{v}' = \mathbf{B}\boldsymbol{\beta}$. Then step 2(b) performs a one-step update on the approximation to obtain

$$\mathbf{v} = \mathbf{r}_d + \lambda \mathbf{P}_d \mathbf{v}' = \mathbf{r}_d + \lambda \mathbf{P}_d \mathbf{B}\boldsymbol{\beta}.$$

Step 2(c) represents projection (least squares regression of \mathbf{v} on the basis vectors) on $\text{col}(\mathbf{B})$ to obtain

$$\boldsymbol{\beta}' = (\mathbf{B}^\top \mathbf{B})^{-1} \mathbf{B}^\top \mathbf{v} = \boldsymbol{\Gamma} \mathbf{v}$$

Combining the above two expressions it follows that LSPE generates β' from β according to

$$\beta' = \Gamma \left(\mathbf{r}_d + \lambda \mathbf{P}_d \mathbf{B} \beta \right) = \Gamma \mathbf{r}_d + \lambda \Gamma \mathbf{P}_d \mathbf{B} \beta. \quad (9.21)$$

As a result of (9.21), if $\lambda \Gamma \mathbf{P}_d \mathbf{B}$ is a contraction, LSPE converges to the solution of

$$(\mathbf{I} - \lambda \Gamma \mathbf{P}_d \mathbf{B}) \beta = \Gamma \mathbf{r}_d, \quad (9.22)$$

where \mathbf{I} is the $(I+1) \times (I+1)$ identity matrix. In this case, $(\mathbf{I} - \lambda \Gamma \mathbf{P}_d \mathbf{B})$ is invertible so that the fixed point of the projection equation can be represented by

$$\hat{\beta} = (\mathbf{I} - \lambda \Gamma \mathbf{P}_d \mathbf{B})^{-1} \Gamma \mathbf{r}_d. \quad (9.23)$$

Thus, in general, the updated approximation at the subsequent application of step 2(a) becomes

$$\mathbf{v}' = \mathbf{B} \Gamma (\mathbf{r}_d + \lambda \mathbf{P}_d \mathbf{v}) = \Pi (\mathbf{r}_d + \lambda \mathbf{P}_d \mathbf{v}) = \Pi L_d \mathbf{v} \quad (9.24)$$

in agreement with Figure 9.6 where $\mathbf{v} = \mathbf{B} \beta$ and $\mathbf{v}' = \Pi L_d \mathbf{B} \beta$. As a result of this observation, the above algorithm may also be referred to as “projected value iteration”.

Examples

The following example¹³ shows that the inverse in (9.23) need not exist.

Example 9.4. Let $S = \{s_1, s_2\}$, \mathbf{r}_d be arbitrary, $\lambda = 5/5.4 = 0.926$,

$$\mathbf{P}_d = \begin{bmatrix} 0.2 & 0.8 \\ 0.2 & 0.8 \end{bmatrix} \quad \text{and} \quad \mathbf{B} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}.$$

Hence

$$\Gamma = \left([1 \ 2] \begin{bmatrix} 1 \\ 2 \end{bmatrix} \right)^{-1} [1 \ 2] = \frac{1}{5} [1 \ 2]$$

so that

$$\lambda \Gamma \mathbf{P}_d \mathbf{B} = \frac{5}{5.4} \left(\frac{1}{5} [1 \ 2] \begin{bmatrix} 0.2 & 0.8 \\ 0.2 & 0.8 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \end{bmatrix} \right) = 1.$$

This implies that $\lambda \Gamma \mathbf{P}_d \mathbf{B}$ is not a contraction so that $\mathbf{I} - \lambda \Gamma \mathbf{P}_d \mathbf{B}$ is not invertible.

In this example, the iterative scheme implicit in (9.21) reduces to

$$\beta' \leftarrow \Gamma \mathbf{r}_d + \beta.$$

Hence, if $\Gamma \mathbf{r}_d \neq \mathbf{0}$, LSPE diverges and if $\Gamma \mathbf{r}_d = \mathbf{0}$, it does not uniquely determine β .

¹³This example appears in de Farias and Roy [2000].

This example is an *edge case* that is unlikely to occur in practice. In most settings, LSPE will converge as shown in the example below. Note that under the conditions that \mathbf{P}_d is recurrent and that the basis vectors are linearly independent, ΠL_d is a contraction mapping¹⁴ with respect to the weighted Euclidean norm (9.12) with weight function equal to the (positive) steady state distribution of \mathbf{P}_d .

Example 9.5. This example illustrates the calculations described above in the context of the queuing service rate control model analyzed in Example 9.1. It approximates the value function of the previously specified stationary policy d^∞ by linear and quadratic functions of the state and compares the estimates based on LSPE (or its fixed point representation based on (9.23)) to the exact value function and its direct OLS estimate.

Set $c(s, a) := f(s) + m(a)$. Step 2(b) of LSPE is implemented component-wise using the expressions:

$$v(0) = c(0, d(0)) + \lambda((1 - b)v'(0) + bv'(1)),$$

$$v(50) = c(50, d(50)) + \lambda(d(50)v'(49) + (1 - d(50))v'(50)) \quad (9.25)$$

and

$$v(s) = c(s, d(s)) + \lambda((1 - b)v'(s - 1) + (1 - b - d(s))v'(s) + d(s)v'(s)),$$

for $s = 1, \dots, 49$ where, $d(s)$ denotes the service rate in state s .

Note that there is no need to distinguish the truncation state 50 as in (9.25) since the approximation $v'(51)$ is available. Choosing such an approach may be regarding as evaluating the infinite state model with $S = \{0, 1, \dots\}$ by an approximation of β based on the reduced set of states $S' = \{0, \dots, 50\}$.

The analysis below chooses $v'(s)$ to be either the linear approximation

$$v'(s) = \beta_0 + \beta_1 s$$

or the quadratic approximation

$$v'(s) = \beta_0 + \beta_1 s + \beta_2 s^2.$$

Applying LSPE for the linear approximation yields $\hat{\beta}_0 = -15825.3$ and $\hat{\beta}_1 = 1682.6$ compared to OLS values $\hat{\beta}_0 = -7603.3$ and $\hat{\beta}_1 = 1320.9$. Graphical comparisons are provided in Figure 9.7a. The main takeaways are that these two estimates are quite different and neither approximates $v_\lambda^{d^\infty}(s)$ well.

Using the quadratic approximation, LSPE gives $\hat{\beta}_0 = 3371.2$, $\hat{\beta}_1 = -216.1$ and $\hat{\beta}_2 = -30.6$ in contrast to the OLS values $\hat{\beta}_0 = 2096.3$, $\hat{\beta}_1 = 133.2$ and $\hat{\beta}_2 = 23.8$.

¹⁴See Bertsekas [2012], pp. 423-427.

Graphical comparisons of fitted values appear in Figure 9.7b. Observe that the quadratic fit is superior to the linear fit. Moreover, both approximations are close to the true value function for $s \geq 30$ but differ at lower values. Note also that the LSPE approximation is not monotone in s for low occupancy levels.

Note that improved fits may be possible by taking into account that $d(s)$ has a step change at $s = 20$. However letting the form of a particular policy determine the approximation will have downsides when turning to optimization. It is left to the reader to compare approximations that use cubic polynomials or a splines with knots at 20.

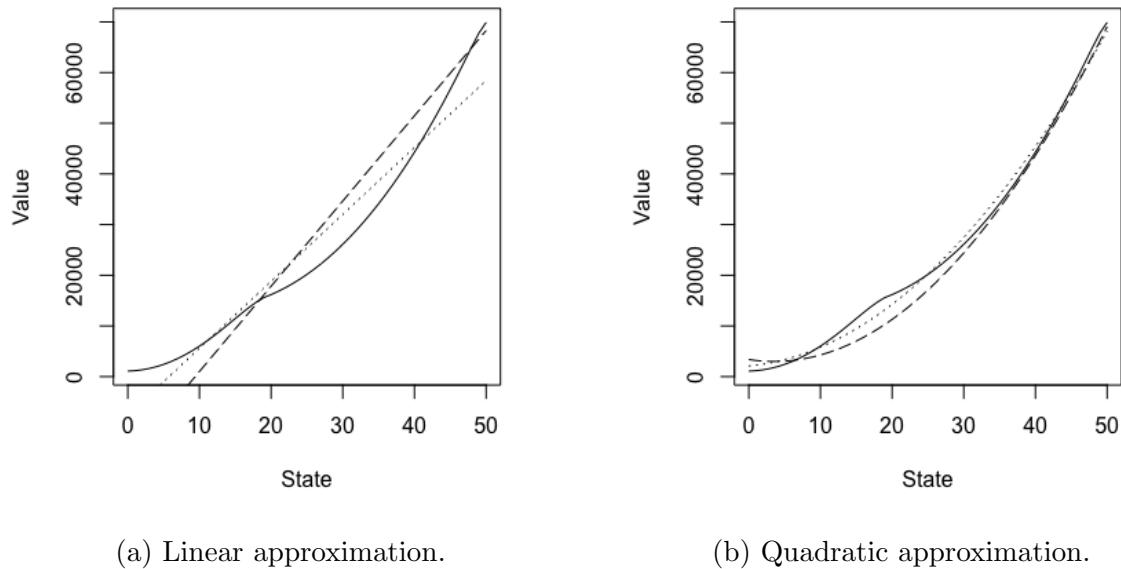


Figure 9.7: Linear and quadratic approximations to the exact value function for the specified policy in Example 9.5. The solid line indicates the exact value function, the dotted line indicates the OLS estimate of the exact value function and the dashed line indicates the LSPE approximation.

Using subsets of S in LSPE and other iterative algorithms

The numerical examples above applied LSPE using basis functions and updates defined for **all** $s \in S$. But in practice, especially when $S = \{s_1, \dots, s_M\}$ is large, one would seek to obtain estimates based on basis functions and updates using a small subset $S' = \{s'_1, \dots, s'_n\} \subset S$. This affects Algorithm 9.1 and all other iterative methods in this chapter as follows.

To begin with, \mathbf{B} has n rows with entries in each column corresponding to the values of basis functions on S' . Then instead of applying steps 2(a) and 2(b) for all $s \in S$, the algorithm generates $v(s')$ for $s' \in S'$ according to

$$v(s') \leftarrow r(s', d(s')) + \lambda \sum_{j \in S} \left(p(j|s', d(s')) \sum_{i=0}^I \beta_i b_i(j) \right). \quad (9.26)$$

Note that summation over j involves states that are not in S' . Therefore in such states the update uses the current value of $\boldsymbol{\beta}$ to approximate the value in these states by $\sum_{i=0}^I \beta_i b_i(j)$. Fortunately in most practical examples, such as queuing control or Gridworld navigation, $p(j|s', d(s')) > 0$ is positive for only a small number of successor states so computing this sum is not burdensome.

Applying (9.26) for all $s' \in S'$ results in a “data set” suitable for regression. This data, in matrix form, consists of

$$\mathbf{v} = \begin{bmatrix} v(s'_1) \\ \vdots \\ v(s'_n) \end{bmatrix} \quad \text{and} \quad \mathbf{B} = \begin{bmatrix} b_0(s'_1) & \dots & b_I(s'_1) \\ \vdots & \vdots & \vdots \\ b_0(s'_n) & \dots & b_I(s'_n) \end{bmatrix}.$$

Thus estimates of $\boldsymbol{\beta}'$ can be obtained in closed form as

$$\boldsymbol{\beta}' = \mathbf{\Gamma}\mathbf{v} = (\mathbf{B}^\top \mathbf{B})^{-1} \mathbf{B}^\top \mathbf{v}$$

or by using a regression package. In the nonlinear setting, $f(b_0(j), b_1(j), \dots, b_I(j); \boldsymbol{\beta})$ replaces $\sum_{i=0}^I \beta_i b_i(j)$ in the update.

Bellman residual minimization*

As noted above, Bellman residual minimization obtains a linear approximation to the value function by substituting $\mathbf{v} = \mathbf{B}\boldsymbol{\beta}$ into $(\mathbf{I} - \lambda \mathbf{P}_d)\mathbf{v} = \mathbf{r}_d$ to obtain

$$(\mathbf{I} - \lambda \mathbf{P}_d)\mathbf{B}\boldsymbol{\beta} = (\mathbf{B} - \lambda \mathbf{P}_d \mathbf{B})\boldsymbol{\beta} \approx \mathbf{r}_d. \quad (9.27)$$

This approach regresses \mathbf{r}_d on the columns of $(\mathbf{I} - \lambda \mathbf{P}_d)\mathbf{B}$ directly. Defining $\mathbf{G}_d := \mathbf{B} - \lambda \mathbf{P}_d \mathbf{B}$, the least squares approximations based on (9.27) become

$$\hat{\mathbf{v}} = \mathbf{G}_d(\mathbf{G}_d^\top \mathbf{G}_d)^{-1} \mathbf{G}_d^\top \mathbf{r}_d \quad (9.28)$$

and

$$\hat{\boldsymbol{\beta}}_{\text{BR}} = (\mathbf{G}_d^\top \mathbf{G}_d)^{-1} \mathbf{G}_d^\top \mathbf{r}_d. \quad (9.29)$$

Note that in the above, the matrix $\mathbf{G}_d = \mathbf{B} - \lambda \mathbf{P}_d \mathbf{B}$ is $M \times (I+1)$ so that $\mathbf{G}_d^\top \mathbf{G}_d$ is $(I+1) \times (I+1)$.

To avoid matrix inversion, one can instead solve the normal equations

$$\mathbf{G}_d^\top \mathbf{G}_d \hat{\boldsymbol{\beta}} = \mathbf{G}_d^\top \mathbf{r}_d. \quad (9.30)$$

directly.

Analysis of the quality of this approximation is left to the reader.

9.3.2 Linear programming approximations for policy value functions

Linear programming offers another way of finding approximations for linear architectures.

Approximate linear programming – primal model

Chapter 5 formulated the following (primal) LP to determine optimal value functions:

$$\begin{aligned} & \text{minimize} \quad \sum_{s \in S} \alpha(s)v(s) \\ & \text{subject to} \quad v(s) - \lambda \sum_{j \in S} p(j|s, a)v(j) \geq r(s, a), \quad a \in A_s, s \in S. \end{aligned}$$

When there is **only one** deterministic stationary policy $d^\infty(s)$ to evaluate, this primal LP reduces to

$$\text{minimize} \quad \sum_{s \in S} \alpha(s)v(s) \tag{9.31a}$$

$$\text{subject to} \quad v(s) - \lambda \sum_{j \in S} p(j|s, d(s))v(j) \geq r(s, d(s)), \quad s \in S. \tag{9.31b}$$

Observe that when restricted to a single stationary policy, this LP has $|S|$ variables and $|S|$ constraints. Now suppose $v(s)$ is approximated by a linear function of pre-specified basis functions $(b_0(s), b_1(s), \dots, b_I(s))$ of the form

$$v(s) = \beta_0 b_0(s) + \beta_1 b_1(s) + \dots + \beta_I b_I(s).$$

Substituting this expression into (9.31a) and (9.31b) and rearranging terms gives the approximate primal linear program:

Approximate primal linear program (APLP) – fixed decision rule

$$\text{minimize} \quad \sum_{i=0}^I \beta_i \sum_{s \in S} \alpha(s)b_i(s) \tag{9.32a}$$

$$\text{subject to} \quad \sum_{i=0}^I \beta_i b_i(s) - \lambda \left(\sum_{i=0}^I \beta_i \sum_{j \in S} p(j|s, d(s))b_i(j) \right) \geq r(s, d(s)), \quad s \in S. \tag{9.32b}$$

Observe that this LP has $I + 1$ variables $\beta_0, \beta_1, \dots, \beta_I$ and $|S|$ constraints. Since in most applications $|S| \gg I$, this system has many more constraints than variables. The following example illustrates simple cases of this approximation and provides insight into the geometry underlying the LP approximation.

Example 9.6. Suppose there is a single feature $b_0(s) = 1$ for all $s \in S$ so that

$$\mathbf{B} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}.$$

Consequently the value function will be estimated^a by a constant β_0 . Assuming $\sum_{s \in S} \alpha(s) = 1$, the APLP becomes

$$\begin{aligned} & \text{minimize} && \beta_0 \\ & \text{subject to} && (1 - \lambda)\beta_0 \geq r(s, d(s)), \quad s \in S. \end{aligned} \tag{9.33}$$

Moreover this easy to solve LP has one variable and $|S|$ constraints. Its optimal solution is

$$\hat{\beta}_0^{\text{LP}} = \max_{s \in S} \left\{ \frac{r(s, d(s))}{1 - \lambda} \right\}.$$

Now consider as a special case the two-state ($S = \{0, 1\}$) version of the queuing service rate control model previously analyzed in this chapter. Assume an arrival rate of 0.2 and service rate of $d(0) = a_1 = 0.2$ and $d(1) = a_2 = 0.4$. To be consistent with the LP formulation, represent it as a reward maximization problem with $r(0, d(0)) = -5$ and $r(1, d(1)) = -41$.

Therefore the approximate linear program (9.33) when $\lambda = 0.5$ becomes

$$\begin{aligned} & \text{minimize} && \beta_0 \\ & \text{subject to} && 0.5\beta_0 \geq -5 \\ & && 0.5\beta_0 \geq -41 \end{aligned}$$

and its optimal solution is $\hat{\beta}_0^{\text{LP}} = -10$. Consequently the LP approximation to the value function is $\hat{v}_{\text{LP}}(0) = \hat{v}_{\text{LP}}(1) = -10$. Note that if S were instead truncated at $N = 50$, there would be 51 inequalities of a similar form to those above.

The exact LP for this model is

$$\begin{aligned} & \text{minimize} && \alpha_0 v(0) + \alpha_1 v(1) \\ & \text{subject to} && 0.6v(0) - 0.1v(1) \geq -5 \\ & && -0.2v(0) + 0.7v(1) \geq -41 \end{aligned}$$

The solution of the exact LP is $v_{\lambda}^{d^\infty}(0) = -19$ and $v_{\lambda}^{d^\infty}(1) = -64$. Note that the least squares approximation to this solution would be the average of these

two values, $\hat{\beta}_0^{\text{OLS}} = -41.5$, so that the least squares approximation is $\hat{v}_{\text{OLS}}(0) = \hat{v}_{\text{OLS}}(1) = -41.5$.

Figure 9.8 illustrates these estimates. Its caption explains what is shown in considerable detail. The takeaway is that when the subspace spanned by the basis functions intersects the feasible region near the optimal vertex, then the LP approach will provide good estimates of the optimal value function. Unfortunately, this cannot be known priori, so care must be taken when choosing basis functions.

^aIn a regression model with only a constant, the parameter estimate is the mean of the observations.

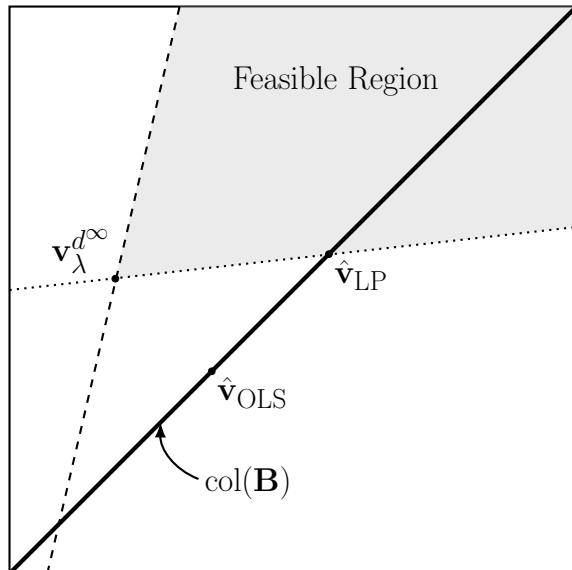


Figure 9.8: Graphical representation of the value function and two approximations for Example 9.6. It shows the feasible region for the exact LP, its solution $\mathbf{v}_\lambda^{d^\infty}$, and the subspace $\text{col}(\mathbf{B})$. The quantities $\hat{\mathbf{v}}_{\text{LP}}$ and $\hat{\mathbf{v}}_{\text{OLS}}$ denote the LP and OLS estimates in that subspace. Since the LP minimizes β_0 , its value is the minimum value of the intersection of $\text{col}(\mathbf{B})$ and the feasible region of the LP. The OLS estimate is the orthogonal projection of the exact value function on to that subspace.

The approximate LP model will be discussed further below when it is used to find approximate optimal values.

9.4 Parameter estimation for a fixed policy: Non-linear architectures

This section modifies Algorithm 9.1 to use nonlinear architectures. This requires replacing computation of \mathbf{v}' in step 2(a) by

$$v'(s) = f(s, \boldsymbol{\beta}),$$

written as $\mathbf{v}' \leftarrow \mathbf{f}(\boldsymbol{\beta})$. Step 2(c), replaces the computation of \mathbf{v}' by its least squares estimate obtained using Gauss-Newton iteration or some other method expressed as $\boldsymbol{\beta}' \in \arg \min_{\boldsymbol{\beta}} \|\mathbf{f}(\boldsymbol{\beta}) - \mathbf{v}\|_2$ since a closed form representation for $\hat{\boldsymbol{\beta}}$ is unavailable for nonlinear architectures.

Algorithm 9.2. Least squares policy evaluation: nonlinear architectures

1. **Initialize:** Specify $\boldsymbol{\beta}$ arbitrary, $\epsilon > 0$, and $\Delta > \epsilon$.
2. **Iterate:** While $\Delta \geq \epsilon$:
 - (a) $\mathbf{v}' \leftarrow \mathbf{f}(\boldsymbol{\beta})$.
 - (b) $\mathbf{v} \leftarrow \mathbf{r}_d + \lambda \mathbf{P}_d \mathbf{v}'$.
 - (c) $\boldsymbol{\beta}' \in \arg \min_{\boldsymbol{\beta}} \|\mathbf{f}(\boldsymbol{\beta}) - \mathbf{v}\|_2$.
 - (d) $\Delta \leftarrow \|\boldsymbol{\beta}' - \boldsymbol{\beta}\|_2$.
 - (e) $\boldsymbol{\beta} \leftarrow \boldsymbol{\beta}'$.
3. **Terminate:** Return $\boldsymbol{\beta}$.

The following example applies Algorithm 9.2 to fit a nonlinear exponential approximation to the value function in the queuing service rate control model.

Example 9.7. This example illustrates the nonlinear LSPE algorithm above in the queuing service rate control model in Example 9.1. In it, the value function for the specified stationary policy d^∞ is approximated by an exponential function. It compares the estimate based on LSPE to the exponential fit when the value function is known. To implement the algorithm, chose $\boldsymbol{\beta} = \mathbf{0}$ so that the corresponding initial value function $\tilde{\mathbf{v}} = \mathbf{0}$ and \mathbf{v} in step 2(b) becomes $\mathbf{v} = \mathbf{r}_d$. Step 2(c) used the *nls* function in R. Estimation required some adjustment of starting values to ensure convergence at every iteration of step 2(c).

With $\epsilon = 0.001$, the algorithm terminated in 386 iterations with the approxi-

mation

$$\hat{v}_\lambda^{d^\infty}(s) = -13427.3 + 13004.1e^{0.037s}$$

which is shown in Figure 9.9. Observe that $\hat{\beta}_1$ and $\hat{\beta}_2$ differ from that in Example 9.1 but $\hat{\beta}_3$ is almost identical to that obtained when the value function was known. Moreover the approximation was considerably more accurate than those based on linear and quadratic approximations.

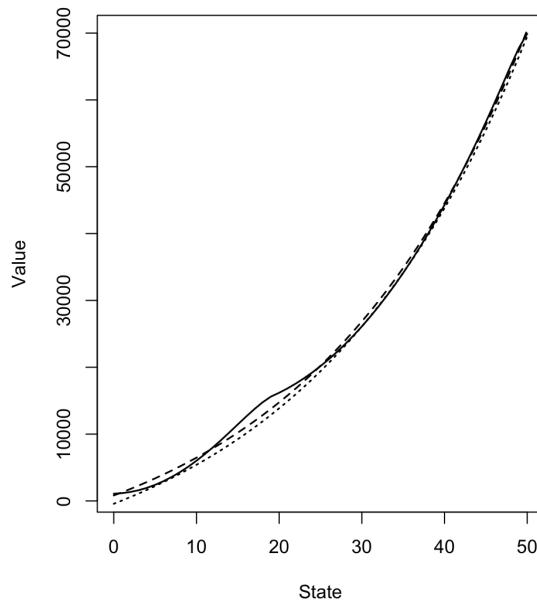


Figure 9.9: Value function (solid line) for queuing service rate control model with nonlinear least squares approximation (dashed line) and nonlinear LSPE approximation (dotted line).

This example suggest that applying nonlinear architectures requires considerably more care than in the linear case because of the sensitivity of nonlinear algorithms to starting values and other algorithmic parameters. As in the case of a linear architecture, this method can be applied to a subset of states that are pre-determined or sampled.

Note that Algorithm 9.2 can also be used to fit a neural network iteratively. In this case step 2(b) would require repeated training of the neural network.

9.5 Combining approximation and optimization

This section generalizes the previous section by providing methods that seek good policies in a Markov decision process based on approximations to the optimal value

function. Such methods are usually referred to as approximate dynamic programming (ADP). It presents value iteration, policy iteration and modified policy iteration algorithms based on linear, nonlinear and neural network architectures and a linear programming approach, which by necessity applies only to linear architectures.

In practice these iterative methods can be applied to pre-specified or randomly selected subsets of the state space, but most of our examples implement them on the entire state space to illustrate the most optimistic outcomes.

9.5.1 Iterative methods for linear architectures

The following algorithm is an optimization version of LSPE. It is then generalized to obtain policy iteration and modified policy iteration algorithms. The approaches all assumes a pre-specified set of basis functions and a set of states at which to evaluate them so that the matrix \mathbf{B} can be specified a priori.

Least squares value iteration (LSVI) – linear architectures

The following algorithm generalizes Algorithm 9.1 to incorporate optimization. The only change is in step 2(b) where the Bellman operator replaces the one-step policy update $\mathbf{v} \leftarrow \mathbf{r}_d + \lambda \mathbf{P}_d \mathbf{v}'$.

Algorithm 9.3. Least squares value iteration: linear architectures

1. **Initialize:** Specify $\boldsymbol{\beta}$ arbitrary, $\epsilon > 0$, and $\Delta > \epsilon$.
2. **Iterate:** While $\Delta \geq \epsilon$:
 - (a) $\mathbf{v}' \leftarrow \mathbf{B}\boldsymbol{\beta}$.
 - (b) $\mathbf{v} \leftarrow \text{c-max}_{d \in D^{\text{MD}}} \{ \mathbf{r}_d + \lambda \mathbf{P}_d \mathbf{v}' \}$.
 - (c) $\boldsymbol{\beta}' \leftarrow \boldsymbol{\Gamma}\mathbf{v}$.
 - (d) $\Delta \leftarrow \|\boldsymbol{\beta}' - \boldsymbol{\beta}\|_2$.
 - (e) $\boldsymbol{\beta} \leftarrow \boldsymbol{\beta}'$.
3. **Terminate:** Return $\boldsymbol{\beta}$.

As noted in Example 9.4 above, the algorithm cannot be guaranteed to converge because the operator implicitly defined by the algorithm need not be a contraction mapping. However, empirically, this algorithm performs well as illustrated by the examples below.

Least squares policy iteration (LSPI) – linear architectures

This section describes a policy iteration method for finding “good” policies. Instead of choosing the value in step 2(b) of Algorithm 9.3, it obtains the arg c-max, evaluates the resulting policy and then finds the best approximation. The algorithm can be initiated with either:

1. a policy,
2. a value, or
3. a parameter vector.

The order of steps in the first pass through the algorithm depends on how it is initialized. With linear architectures it appeared best to begin with a parameter or value; with nonlinear architectures, methods based on initialization with a value seemed to work best. In the latter case, the first pass through step 2 begins with an improvement.

Algorithm 9.4. Least squares policy iteration: linear architectures

1. **Initialize:** Specify $d' \in D^{\text{MD}}$, β arbitrary, $\epsilon > 0$, and $\Delta > \epsilon$.
2. **Iterate:** While $\Delta \geq \epsilon$:
 - (a) **Parameter evaluation:** $\beta' \leftarrow (\mathbf{I} - \lambda \Gamma \mathbf{P}_{d'} \mathbf{B})^{-1} \Gamma \mathbf{r}_{d'}$.
 - (b) **Value function approximation:** $\mathbf{v}' \leftarrow \mathbf{B}\beta'$.
 - (c) **Improvement:**

$$d' \leftarrow \arg \underset{d \in D^{\text{MD}}}{\text{c-max}} \{ \mathbf{r}_d + \lambda \mathbf{P}_d \mathbf{v}' \}.$$

- (d) $\Delta \leftarrow \|\beta' - \beta\|_2$.
 - (e) $\beta \leftarrow \beta'$.
3. **Terminate:** Return d' and β .

Some comments about this algorithm follow.

1. This algorithm can be applied to a subset of S using methods described in the section on LSPE.
2. The stopping criterion in step 2 is stated in terms of parameter estimates. It can be replaced by either: “Stop when $d' = d$ ” or stop when the change in \mathbf{v} is

sufficiently small. Our calculations found no difference in algorithmic behavior between these criteria in examples. Note if a stopping criterion is based on decision rules, the specification “set $d' = d$ if possible” to avoid cycling needs to be added.

3. Note that the only reason to carry β' forward in step 2(e) is to evaluate the stopping criterion at the next iteration, it is not used in steps 2(a)-2(c).
4. Step 2(a) provides the closed-form representation for β corresponding to decision rule d . This representation is valid for linear architectures. In practice, it might be easier to implement it using LSPE. When using nonlinear architectures, this step would be replaced by LSPE.
5. Recall that the matrix $\Gamma = (\mathbf{B}^\top \mathbf{B})^{-1} \mathbf{B}^\top$.
6. When the algorithm is initialized with a starting value for β , step 2(a) can be skipped at the first pass through the algorithm.
7. This algorithm can be modified easily to use approximations of state-action value functions instead of value functions.

Example 9.8. This example applies LSPI to the queuing service rate control model with $S = \{0, \dots, 50\}$. It compares linear, quadratic and cubic polynomial approximations. The algorithm was initiated with the previously considered decision rule and used $\epsilon = 0.0001$ for the stopping criterion. All instances required 3 iterations for convergence.

Figure 9.10 shows the optimal policy and the policies determined by the algorithm. In all cases the approximations resulted in monotone non-decreasing decision rules. Clearly the linear approximation was inadequate. The quadratic and cubic approximations produced similar policies differing only in states 11, 12, 13 and 30. The policy identified by the cubic approximation was optimal.

A larger model: Additional calculations revealed that the policies identified by the cubic approximation agreed with the optimal policy for $S = \{0, \dots, 500\}$ and $\lambda = 0.9^{\text{#}}$. When $\lambda = 0.98$, the policy obtained from the cubic approximation d_{cub}^∞ and the optimal policy d_{opt}^∞ differ as follows:

$$d_{\text{cub}}(s) = \begin{cases} a_1 & s \in \{0, \dots, 20\} \\ a_2 & s \in \{21, 22, 23\} \\ a_3 & s \in \{24, \dots, 500\} \end{cases} \quad \text{and} \quad d_{\text{opt}}(s) = \begin{cases} a_1 & s \in \{0, \dots, 8\} \\ a_2 & s \in \{10, \dots, 15\} \\ a_3 & s \in \{16, \dots, 500\}. \end{cases}$$

To gain further insight into the effect of approximation, the values of these two policies were compared. Note that LSPI does not provide the value of d_{cub}^∞ ; this requires computing $(\mathbf{I} - \lambda \mathbf{P}_{d_{\text{cub}}})^{-1} \mathbf{r}_{d_{\text{cub}}}$. These calculations show that values

differ most at low occupancy levels. Bounds in the next subsection will provide further insight. However, it appears that more accurate approximations may be needed when λ is close to 1.

^aWhen solving the larger instance, the approximating polynomials were centered at 250 to avoid numerical instability in step 2(a) when computing $(\mathbf{B}^\top \mathbf{B})^{-1}$. That is, the components in the \mathbf{B} matrix were of the form $s - 250$, $(s - 250)^2$ and $(s - 250)^3$. Alternatively, this step could be replaced by iterative policy evaluation such as in Algorithm 9.5 below.

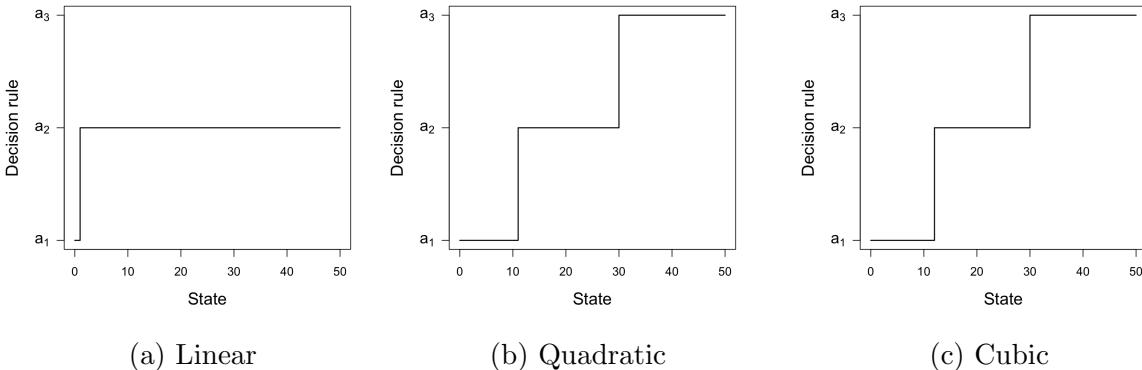


Figure 9.10: Stationary policies obtained using LSPI with linear, quadratic and cubic approximations. The policy obtained using a cubic approximation was optimal. In these figures, $S = \{0, \dots, 50\}$ and $\lambda = 0.9$.

The following example shows that LSPI may not converge when using linear approximations.

Example 9.9. This example applies LSPI to the two-state example from Section 2.5 using the single basis function $b(s_1) = 0.05$ and $b(s_2) = 1$, and setting $\lambda = 0.9$. In this case,

$$\mathbf{B} = \begin{bmatrix} 0.05 \\ 1 \end{bmatrix} \quad \text{and} \quad \mathbf{B}\beta = \begin{bmatrix} 0.05\beta \\ \beta \end{bmatrix}.$$

The Bellman update (in component notation) applied to the approximation becomes:

$$\begin{aligned} v(s_1) &\leftarrow \max(3 + \lambda(0.8 \cdot 0.05\beta + 0.2\beta), 5 + \lambda\beta) \\ v(s_2) &\leftarrow \max(-5 + \lambda\beta, 3 + \lambda(0.4 \cdot 0.05\beta) + 0.6\beta) \end{aligned}$$

It is left as an exercise to show that for initial values $\beta = 0$ and $\beta = -50$, LSVI converges quickly and identifies an optimal policy. However, for both

initial values, LSPI does not converge. Initially it chooses the decision rule $(a_{1,2}, a_{2,2})$ but then it **cycles** between the parameter estimates -51.87 and 26.92 , and decision rules $(a_{1,2}, a_{2,1})$ and $(a_{1,1}, a_{2,2})$ indefinitely.

On the other hand, for other basis functions and starting values of β both LSPI and LSVI converge to either an optimal policy or a sub-optimal policy¹⁴.

This simple example shows that it is difficult to predict the behavior of LSPI and LSVI. This cycling phenomenon is explored further in Bertsekas [2011] and Bertsekas [2012].

¹⁴In LSVI, the resulting policy is defined as the greedy decision rule based on the value approximation, once a convergence criterion is met.

LSPI bounds

This section applies the bounds from Section 5.5.3 to LSPI. Step 2(c) of the algorithm implicitly computes the quantity

$$L\mathbf{v}' = \underset{d \in D^{\text{MD}}}{\text{c-max}} \{ \mathbf{r}_d + \lambda \mathbf{P}_d \mathbf{v}' \}.$$

Therefore with minimal additional computation the bounds from Theorem 5.12 can be applied to obtain

$$\begin{aligned} \mathbf{v}' + \frac{1}{(1-\lambda)} \overline{(L\mathbf{v}' - \mathbf{v}')} \mathbf{e} &\leq L\mathbf{v}' + \frac{\lambda}{(1-\lambda)} \overline{(L\mathbf{v}' - \mathbf{v}') \mathbf{e}} \leq \mathbf{v}_\lambda^{(d')^\infty} \leq \mathbf{v}_\lambda^* \\ &\leq L\mathbf{v}' + \frac{\lambda}{(1-\lambda)} \overline{(L\mathbf{v}' - \mathbf{v}')} \mathbf{e} \leq \mathbf{v}' + \frac{1}{(1-\lambda)} \overline{(L\mathbf{v}' - \mathbf{v}') \mathbf{e}}, \end{aligned} \tag{9.34}$$

where as before $\bar{\mathbf{u}} = \max_{s \in S} u(s)$, $\underline{\mathbf{u}} = \min_{s \in S} u(s)$ and

$$d' \in \arg \underset{d \in D^{\text{MD}}}{\text{c-max}} \{ \mathbf{r}_d + \lambda \mathbf{P}_d \mathbf{v}' \}.$$

Most importantly, since $\mathbf{v}_\lambda^{(d')^\infty}$ is never computed using LSPI¹⁵, the following bound provides a measure of how close it is to the optimal value:

$$\| \mathbf{v}_\lambda^{(d')^\infty} - \mathbf{v}_\lambda^* \| \leq \frac{\lambda}{(1-\lambda)} \text{sp}(L\mathbf{v}' - \mathbf{v}'), \tag{9.35}$$

where as before $\text{sp}(\mathbf{v}) = \bar{\mathbf{v}} - \underline{\mathbf{v}}$. Moreover, averaging the inner upper and lower bounds gives the approximation

$$L\mathbf{v}' + \frac{\lambda}{2(1-\lambda)} \text{sp}(L\mathbf{v}' - \mathbf{v}') \mathbf{e} \approx \mathbf{v}_\lambda^*. \tag{9.36}$$

¹⁵It only computes the approximation $\mathbf{v}' = \mathbf{B}\beta'$.

The beauty of these bounds is that they can be applied to any \mathbf{v}' and $L\mathbf{v}'$ regardless of how they have been obtained. Moreover, (9.35) provides an alternative stopping criterion for LSPI (or LSVI). However, if the stopping criterion is too strict or the basis functions are inappropriate, it need not be attained. Also, the factor of $(1 - \lambda)^{-1}$ may result in the bounds not being very tight.

The following example applies these bounds to the queuing control example.

Example 9.10. This example computes the bounds in (9.34) for the queuing service rate control model with capacity 50.

Since \mathbf{v}' is computed in step 2(b) and $L\mathbf{v}'$ can be easily obtained from step 2(c), the quantity $L\mathbf{v}' - \mathbf{v}'$ and all of the bounds can be easily computed. Using a cubic approximation, at termination of LSPI, $L\mathbf{v}' - \mathbf{v}' = -113.9$ and $\overline{L\mathbf{v}' - \mathbf{v}'} = 38.2$. Using these values gives the inner bounds in (9.34), which are shown in Figure 9.11b.

Since $\text{sp}(L\mathbf{v}' - \mathbf{v}') = 152.1$, (9.35) yields the estimate

$$\|\mathbf{v}_\lambda^{(d')\infty} - \mathbf{v}_\lambda^*\| \leq 1369.5.$$

Since in fact $(d')^\infty$ is optimal, this bound is not very accurate. However, the average absolute error of the approximation in (9.36) is 38.8 indicating that the approximation based on (9.36) is very accurate.

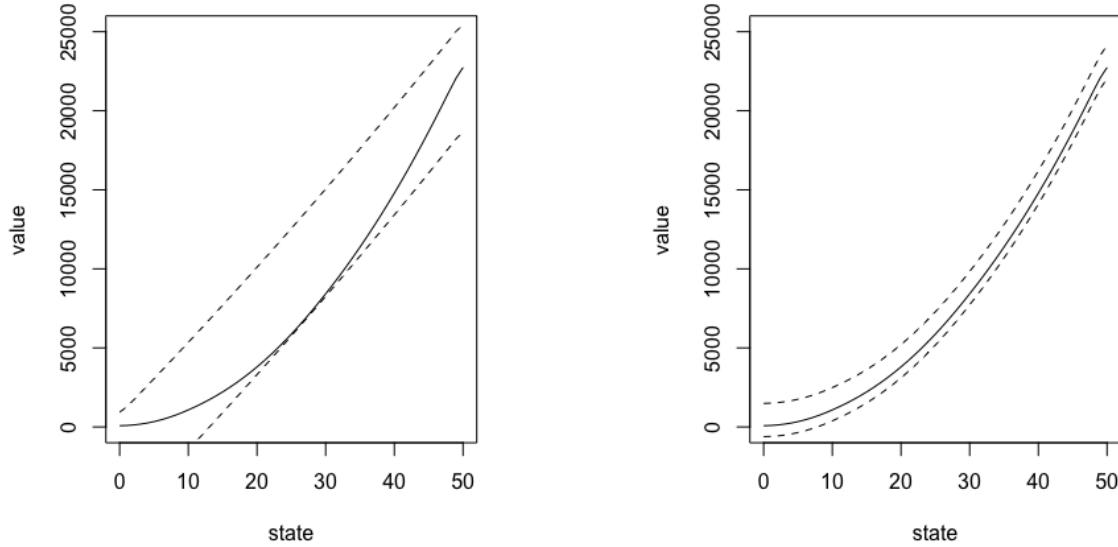
Further calculations show that bounds based on a quadratic approximation are only slightly less tight than those based on the cubic approximation but, those based on a linear approximation are considerably less accurate (Figure 9.11a).

Least squares modified policy iteration – linear architectures

In large applications, it might be challenging to construct and solve the linear system implicit in step 2(a) of the LSPI iteration algorithm. The following generalization of MPI, referred to as *least squares modified policy iteration* and denoted by LSMPI addresses this issue.

Algorithm 9.5. Least squares modified policy iteration: linear architectures

1. **Initialize:** Specify $M \in \mathbb{Z}_+$, β arbitrary, $\epsilon > 0$, and $\Delta > \epsilon$.
2. **Iterate:** While $\Delta \geq \epsilon$:
 - (a) $\mathbf{v} \leftarrow \mathbf{B}\beta$.



(a) Bounds based on a linear approximation to the value function.

(b) Bounds based on a cubic approximation to the value function.

Figure 9.11: Bounds (dashed lines) for value function (solid line) derived from the LSPI solution using inner inequalities in (9.34).

- (b) $\mathbf{v}' \leftarrow \text{c-max}_{d \in D^{\text{MD}}} \{ \mathbf{r}_d + \lambda \mathbf{P}_d \mathbf{v} \}.$
- (c) $d' \leftarrow \arg \text{c-max}_{d \in D^{\text{MD}}} \{ \mathbf{r}_d + \lambda \mathbf{P}_d \mathbf{v} \}.$
- (d) $m \leftarrow 1.$
- (e) While $m \leq M:$
 - i. $\boldsymbol{\beta}' \leftarrow \Gamma \mathbf{v}'.$
 - ii. $\mathbf{v}' \leftarrow \mathbf{r}_{d'} + \lambda \mathbf{P}_{d'} \mathbf{B} \boldsymbol{\beta}'.$
 - iii. $m \leftarrow m + 1.$
- (f) $\boldsymbol{\beta}' \leftarrow \Gamma \mathbf{v}'.$
- (g) $\Delta \leftarrow \|\boldsymbol{\beta}' - \boldsymbol{\beta}\|_2.$
- (h) $\boldsymbol{\beta} \leftarrow \boldsymbol{\beta}'.$

3. **Terminate:** Return d' and $\boldsymbol{\beta}.$

Some comments about using this algorithm follow:

1. The algorithm is presented in a way that enables its implementation on a *subset* of $S.$

2. Step 2(e) corresponds to terminating LSPE after $M + 1$ iterations. If $M = 0$, step 2(e) is skipped and the algorithm is identical LSVI and if $M = \infty$, it is equivalent to LSPI.
3. The integer M denotes the order of the algorithm. It can vary from iteration to iteration. See the discussion in Chapter 5 for guidance on selection of M . Choosing M adaptively to ensure that the norm of successive values of β' computed in 2(e)i is less than some pre-specified small tolerance is equivalent to LSPI.
4. It may be easier (especially when applying to LSMPI to nonlinear architectures¹⁶) to initialize the algorithm with some $\tilde{\mathbf{v}}$ instead of β . In this case step 2(a) is not necessary at the first pass through the step 2.
5. Steps 2(b) and 2(c) are implemented together avoiding double computation.
6. As an alternative to a parameter-based stopping criterion, one could use the value based stopping criterion $\|\mathbf{v}' - \tilde{\mathbf{v}}\| < \epsilon$. Such a modification was necessary when using neural network approximations in which parameter estimates were highly unstable.
7. The concluding observations in Example 9.9 apply as well to LSMPI.

9.5.2 Iterative methods for nonlinear architectures

This brief section generalizes LSVI and LSPI by developing similar methods for finding good policies using nonlinear approximations based on features.

Least squares value iteration – nonlinear architectures

This section modifies the LSVI algorithm (Algorithm 9.3) to allow for nonlinear approximation architectures $\mathbf{v} \approx \mathbf{f}(\beta)$. The main difference with the linear architecture variant is that the estimate of β cannot be obtained in closed form. That is, the closed form representation in step 2(c) of Algorithm 9.3 is replaced by an estimate derived by nonlinear least squares. This step is implemented using appropriate code.

Algorithm 9.6. Least squares value iteration: nonlinear architectures

1. **Initialize:** Specify β arbitrary, $\epsilon > 0$, and $\Delta > \epsilon$.
2. **Iterate:** While $\Delta \geq \epsilon$:
 - (a) $\mathbf{v}' \leftarrow \mathbf{f}(\beta)$.

¹⁶In complicated nonlinear models such as neural networks, it may be challenging to accurately estimate parameter values.

- (b) $\mathbf{v} \leftarrow \text{c-max}_{d \in D^{\text{MD}}} \{ \mathbf{r}_d + \lambda \mathbf{P}_d \mathbf{v}' \}$.
 - (c) $\boldsymbol{\beta}' \in \arg \min_{\boldsymbol{\beta}} \| \mathbf{f}(\boldsymbol{\beta}) - \mathbf{v} \|_2$.
 - (d) $\Delta \leftarrow \| \boldsymbol{\beta}' - \boldsymbol{\beta} \|_2$.
 - (e) $\boldsymbol{\beta} \leftarrow \boldsymbol{\beta}'$.
3. **Terminate:** Return $\boldsymbol{\beta}$.

Least squares policy iteration – nonlinear architectures

The following generalizes LSPI (Algorithm 9.4) to nonlinear architectures.

Algorithm 9.7. Least squares policy iteration: nonlinear architectures

1. **Initialize:** Specify $d \in D^{\text{MD}}$, $\boldsymbol{\beta}$ arbitrary, $\epsilon > 0$, and $\Delta > \epsilon$.
2. **Iterate:** While $\Delta \geq \epsilon$:
 - (a) $\mathbf{v}' \leftarrow \mathbf{f}(\boldsymbol{\beta})$.
 - (b)

$$d' \leftarrow \arg \text{c-max}_{d \in D^{\text{MD}}} \{ \mathbf{r}_d + \lambda \mathbf{P}_d \mathbf{v}' \}.$$
 - (c) Apply LSPE using d' to return $\boldsymbol{\beta}'$.
 - (d) $\Delta \leftarrow \| \boldsymbol{\beta}' - \boldsymbol{\beta} \|_2$.
 - (e) $\boldsymbol{\beta} \leftarrow \boldsymbol{\beta}'$.
3. **Terminate:** Return d' and $\boldsymbol{\beta}$.

Note that step 2(c) produces a parameter estimate corresponding to policy $(d')^\infty$. Its value is computed at the next iteration by $\mathbf{f}(\boldsymbol{\beta})$. Note also that both of these algorithms can be applied on subsets of S using the modification described in the LSPE section.

The following example illustrates the use of nonlinear LSPI in the queuing service rate control model.

Example 9.11. As in Example 9.7 the value function is approximated by

$$v(s) = \beta_0 + \beta_1 e^{\beta_2 s}$$

and $\lambda = 0.9$.

Applying the stopping criterion “stop if $d' = d$ ”, nonlinear LSPI terminates

in three iterations. Note that since the algorithm identified the same decision rule at two successive iterations, it also generated the same values of β at these iterations so a stopping rule based on $\|\beta - \beta'\|_2$ would also terminate after three iterations.

Figure 9.12a shows the optimal value function and the exponential approximation found with Algorithm 9.7. Figure 9.12b shows the stationary policy based on the optimal exponential approximation. Figure 9.12c shows bounds on the optimal value function based on the exponential approximation.

Note that the approximation does not fit well for low state values and the policy obtained using the approximation deviates from the optimal policy in states 4 to 13 and in state 31. (see Figure 9.10). Therefore one can conclude that the cubic polynomial approximation is preferable for this instance.

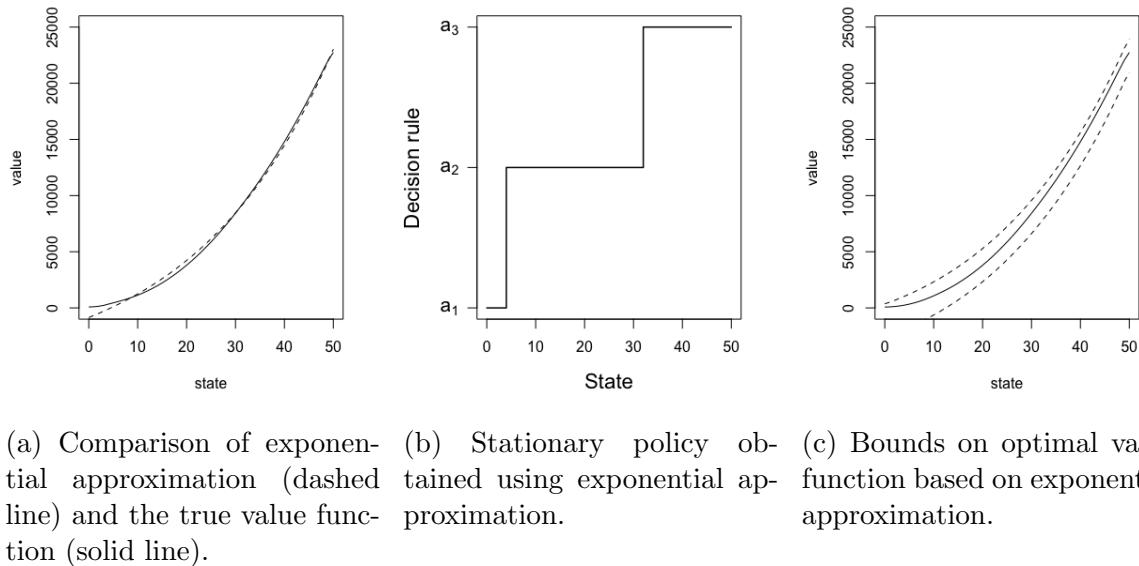


Figure 9.12: Value function, policy and bounds for Example 9.11.

9.5.3 Optimization using linear programming: Linear architectures

The linear programming approach in Section 9.3.2 was used to approximate the policy value function of a fixed policy. This section extends this approach to obtain approximations to the optimal value function. As noted above, this estimate differs from the least squares estimates obtained using LSVI, LSPI or LSMPI. And of course, this approach is only appropriate for approximations that are linear functions of the parameters.

Primal LP approximation

In a reward maximization problem, the primal LP approximation in component form can be written as follows:

Approximate primal linear program (APLP): component form

$$\text{minimize} \quad \sum_{i=0}^I \beta_i \sum_{s \in S} \alpha(s) b_i(s) \quad (9.37a)$$

$$\text{subject to} \quad \sum_{i=0}^I \beta_i b_i(s) - \lambda \left(\sum_{i=0}^I \beta_i \sum_{j \in S} p(j|s, a) b_i(j) \right) \geq r(s, a), \quad a \in A_s, s \in S. \quad (9.37b)$$

In a cost minimization problem this APLP becomes:

$$\text{maximize} \quad \sum_{i=0}^I \beta_i \sum_{s \in S} \alpha(s) b_i(s) \quad (9.38a)$$

$$\text{subject to} \quad \sum_{i=0}^I \beta_i b_i(s) - \lambda \left(\sum_{i=0}^I \beta_i \sum_{j \in S} p(j|s, a) b_i(j) \right) \leq r(s, a), \quad a \in A_s, s \in S. \quad (9.38b)$$

Observe that both models have $I+1$ variables and $\sum_{s \in S} |A_s|$ constraints. Note that the difference between formulation (9.37) and formulation (9.32) is that the constraints are for all $a \in A_s$ rather than for a fixed policy d . Hence, there are $\sum_{s \in S} |A_s|$ constraints instead of $|S|$ constraints.

Matrix version of the approximate linear program

Instead of implementing the model in component notation it is considerably easier to approach it from a matrix perspective. Recall that the (exact) primal model in Section 5.9 can be written as:

$$\begin{aligned} & \text{minimize} && \boldsymbol{\alpha}^\top \mathbf{v} \\ & \text{subject to} && \mathbf{A}\mathbf{v} \geq \mathbf{r}, \end{aligned} \quad (9.39)$$

where \mathbf{r} is a column vector with entries $r(s, a)$ listed in the same order as the constraints represented in \mathbf{A} and $\boldsymbol{\alpha}$ is an arbitrary positive $|S|$ -dimensional vector. Substituting the linear approximation $\mathbf{v} \approx \mathbf{B}\boldsymbol{\beta}$ into (9.39) yields:

Approximate primal linear program: vector form

$$\begin{aligned} & \text{minimize} && (\boldsymbol{\alpha}^T \mathbf{B}) \boldsymbol{\beta} \\ & \text{subject to} && (\mathbf{A} \mathbf{B}) \boldsymbol{\beta} \geq \mathbf{r}. \end{aligned} \tag{9.40}$$

This suggests the following approach for finding policies based on the APLP.

Algorithm 9.8. Using APLP to find an approximate optimal policy

1. Specify an approximation $\hat{\mathbf{v}} \approx \mathbf{B}\boldsymbol{\beta}$.
2. Formulate the exact primal LP (9.39).
3. Transform the model to (9.40).
4. Solve (9.40) to obtain $\hat{\boldsymbol{\beta}}$.
5. Set $\hat{\mathbf{v}} \leftarrow \mathbf{B}\hat{\boldsymbol{\beta}}$.
6. Choose

$$\hat{d} \in \arg \underset{d \in D^{\text{MD}}}{\text{c-max}} \{ \mathbf{r}_d + \lambda \mathbf{P}_d \hat{\mathbf{v}} \}.$$

7. Obtain bounds on \mathbf{v}_λ^* using (9.34).

The beauty of this approach is that \mathbf{B} can easily be modified to try different approximations. The bounds in the last step can be used to assess the quality of the approximation¹⁷. Note that $\hat{\mathbf{v}}$ only equals $\mathbf{v}_\lambda^{d^\infty}$ when $\hat{\mathbf{v}}$ is the optimal value (which it most likely will not be the case for an arbitrary \mathbf{B} and $\hat{\boldsymbol{\beta}}$).

Dual LP approximation

Note that the approximate **dual** LP has the following form:

$$\begin{aligned} & \text{maximize} && \mathbf{r}^T \mathbf{x} \\ & \text{subject to} && (\mathbf{A} \mathbf{B})^T \mathbf{x} = \mathbf{B}^T \boldsymbol{\alpha} \\ & && \mathbf{x} \geq \mathbf{0}. \end{aligned} \tag{9.41}$$

This dual linear program has $I + 1$ equality constraints and $\sum_{s \in S} |A_s|$ variables. Whether the primal or dual is more efficiently solved is largely a function of the problem instance. Modern solvers are capable of deciding the best approach to solving a given

¹⁷We believe the idea of using these bounds in the linear programming context is new. Moreover they can be obtained when finding an approximately optimal policy in the previous step.

LP formulation. Note that if the dual is solved, the primal variables are also available because they are the shadow prices of the dual constraints (and vice versa). This is important because the primal variables are the quantities needed to construct the approximation to the optimal value function. Assuming the dual has many more columns than variables, it may also be solved using column generation¹⁸ methods.

An example

The following example uses linear programming to find linear approximations to the optimal value function in the queuing service rate control model.

Example 9.12. Recall that in this model $A_s = \{a_1, a_2, a_3\}$ with $a_1 = 0.2$, $a_2 = 0.4$ and $a_3 = 0.6$, the probability of an arrival $b = 0.2$ and the cost $c(s, a_k) = s^2 + 5k^3$, which is the sum of the delay cost $f(s) = s^2$ and the serving cost per period $m(a_k) = 5k^3$. The state space is truncated at $N = 50$, $\lambda = 0.9$ and approximations of the form

$$v(s) = \beta_0 + \beta_1 s + \beta_2 s^2$$

are considered.

Since the objective is to minimize costs, formulation (9.38) is used. Because the above quadratic approximation has three parameters, the primal LP has 3 variables and 153 constraints. Note that the corresponding dual linear program, has 153 variables and 3 constraints (plus non-negativity constraints).

Choosing $\alpha(s) > 0$ so that $\sum_{s \in S} \alpha(s) = 1$ and letting $M_1 = \sum_{s \in S} \alpha(s)s$ and $M_2 = \sum_{s \in S} \alpha(s)s^2$ the APLP becomes (after considerable algebra):

$$\begin{aligned} \text{maximize} \quad & \beta_0 + M_1\beta_1 + M_2\beta_2 \\ \text{subject to} \quad & (1 - \lambda)\beta_0 - \lambda b\beta_1 - \lambda b\beta_2 \leq c(s, a_k), \quad s = 0, k = 1, 2, 3, \\ & (1 - \lambda)\beta_0 + (s + \lambda(a_k - b))\beta_1 + (s^2 - \lambda(b + a_k) - 2\lambda(b - a_k)s)\beta_2 \\ & \leq c(s, a_k), \quad s \in \{1, \dots, 49\}, k = 1, 2, 3, \\ & (1 - \lambda)\beta_0 + (s + \lambda a_k)\beta_1 + (s^2 - \lambda a_k + 2\lambda a_k s)\beta_2 \\ & \leq c(s, a_k), \quad s = 50, k = 1, 2, 3. \end{aligned}$$

Using the matrix formulation of the APLP (9.40) avoids the error-prone and inflexible manipulations used to derive the above constraints in component form. As noted previously the matrices of basis function values for linear and quadratic

¹⁸Desrosiers and Lübbcke [2005] provides a primer on column generation in LPs.

approximations are given by

$$\mathbf{B} = \begin{bmatrix} 1 & 0 \\ 1 & 1 \\ 1 & 2 \\ \cdot & \cdot \\ \cdot & \cdot \\ \cdot & \cdot \\ 1 & 50 \end{bmatrix} \quad \text{and} \quad \mathbf{B} = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 1 \\ 1 & 2 & 2^2 \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \\ 1 & 50 & 50^2 \end{bmatrix},$$

respectively. Hence the primal LP model becomes:

$$\begin{aligned} &\text{maximize} && (\boldsymbol{\alpha}^T \mathbf{B}) \boldsymbol{\beta} \\ &\text{subject to} && (\mathbf{AB}) \boldsymbol{\beta} \leq \mathbf{r}. \end{aligned}$$

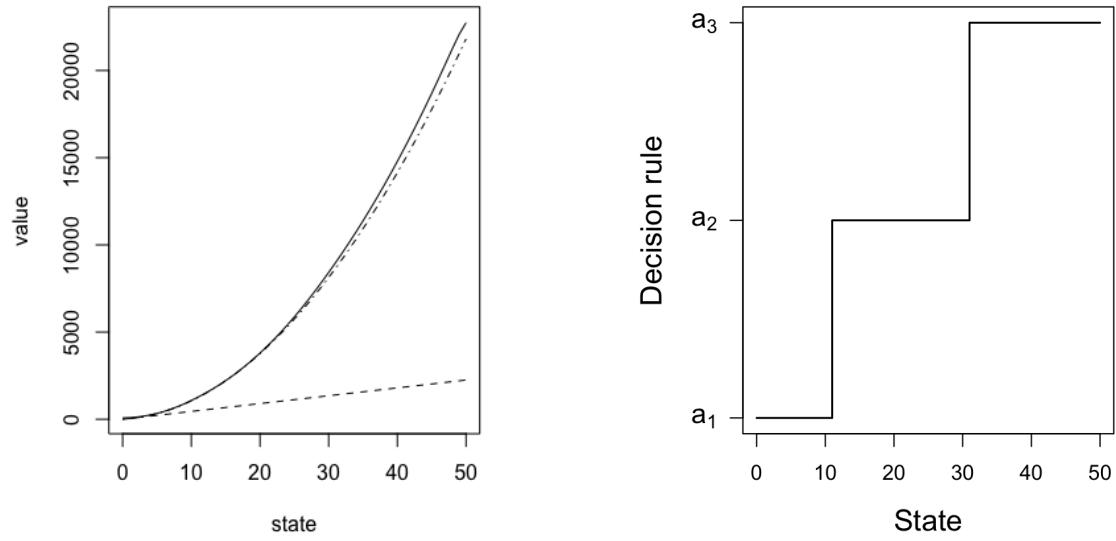
Solving it using the command *simplex* in R provides the estimates $\hat{\boldsymbol{\beta}} = (0, 45)$ for the linear approximation and $\hat{\boldsymbol{\beta}} = (0, 23.2, 8.26)$ for the quadratic approximation.

Figure 9.13a shows the resulting linear and quadratic approximations to the optimal value function. Observe that the linear approximation is inadequate and the quadratic approximation fits well. In both cases, the approximations lie below the true optimal value function. This is because the LP maximizes the greatest lower bound to the optimal value function that satisfies all of the constraints. Moreover the APLP objective function based on the quadratic approximation equals 7532 compared with the optimal objective function of 7538 obtained from the exact LP.

Figure 9.13b shows the stationary policy obtained using the above algorithm based on the quadratic approximation. It is monotone and only differs from the optimal policy in states 30 and 31 where it chooses actions a_2 instead of a_3 .

Note also that for this instance, a cubic approximation resulted in identical values to the quadratic approximation because the coefficient of the cubic term was zero. To further explore this issue, $\lambda = 0.999$ was used. In this case the quadratic and cubic approximations differed, but the greedy policy based on each was the same.

Based on the observations, the linear programming approach described in Algorithm 9.8 offers an attractive and flexible approach for obtaining approximations to the optimal value function.



(a) Linear and quadratic approximations to exact value function (solid line) obtained using APLP.

(b) Stationary policy obtained using quadratic LP approximation.

Figure 9.13: Results of using the approximate LP in the queuing service rate control model with $N = 50$.

9.6 Application: strategic scheduling

This section applies the above methods to the following simplified version of the advanced scheduling problem in Section 3.7. The lengthy discussion in the later part of this section illustrates the type of insights that can be gained by solving a Markov decision process and issues arising when using approximations.

In the simplified model, requests for appointments arrive randomly throughout the current day (today). At the end of the day, appointment requests are scheduled for service on some day in the future (tomorrow and subsequent days). Once scheduled, an appointment cannot be rescheduled or delayed. On account of these assumptions, appointment requests arriving today cannot be scheduled until tomorrow.

There are two appointment request types (Class 1 and Class 2) and two types of service (regular time and overtime¹⁹). Class 1 appointments are urgent and Class 2 appointments are less urgent. Class 1 appointments **must** be served the next day (tomorrow) through either regular time or overtime and Class 2 appointments can be scheduled for regular time tomorrow (if space is available after assigning the Class

¹⁹In the medical setting, overtime is often referred to as *surge capacity*.

1 requests), between 2 and N days into the future or through overtime tomorrow²⁰. Refer to the quantity N as the booking horizon; no appointments can be assigned to an appointment beyond that day. If a Class 1 appointment is scheduled for regular service tomorrow, **no** cost is incurred. If it is scheduled for overtime, the cost is C . If a Class 2 appointment request is scheduled for regular service on day n in the future, the cost is c_n , $n = 1, \dots, N$ ²¹ and the cost is C if it is served through overtime. It is reasonable to assume c_n is non-decreasing in n and that $C > c_N$ ²². Figure 9.14 illustrates the cost structure and decision problem symbolically.

The number of Class 1 and Class 2 appointment requests arriving each day are random and independent. For each day, the probability of j Class 1 requests is p_j and the probability of k Class 2 requests is q_k . The vectors $\mathbf{p} = (p_0, \dots, p_J)$ and $\mathbf{q} = (q_0, \dots, q_K)$ represent the probability distributions. Assume a capacity of M regular time appointments each day, no limit on overtime and at most J Class 1 arrivals and K Class 2 arrivals on any day. Moreover assume demand on successive days is independent.

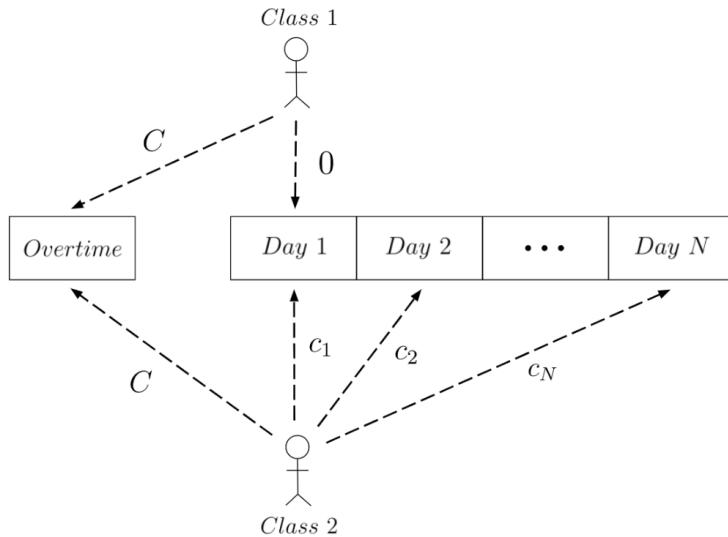


Figure 9.14: Symbolic representation of simplified advance appointment scheduling problem to be analyzed in this section. Note that Day 1 corresponds to “tomorrow”.

Model formulation

A cost minimization Markov decision process formulation follows.

²⁰If an appointment is to be scheduled for overtime, it is better to do so tomorrow instead of incurring delay costs.

²¹Note $n = 1$ corresponds to tomorrow. It seems reasonable to set $c_1 = 0$ but it is left arbitrary to simplify notation.

²²Otherwise it would be preferable to use overtime rather than schedule an appointment on or before day N .

Decision epochs: Decision epochs correspond to the specific point of time at the end of a day when the scheduler assigns requests to appointments. Although there is a finite booking horizon, the problem is ongoing so that an infinite horizon formulation is appropriate. Therefore

$$T = \{1, 2, \dots\}.$$

States: A state is a vector $(s_1, \dots, s_N, d_1, d_2)$ where s_n for $n = 1, \dots, N$ denotes the number of previously booked appointments on days n and d_i for $i = 1, 2$ denotes the number of Class i appointment requests to schedule. Therefore $|S| = (M+1)^N \times (J+1) \times (K+1)$. Note that an alternative state description would replace the number of booked appointments on day n by the number of available appointment slots on day n .

Actions: Actions are the number of Class 2 requests to assign to day n or to overtime. More formally let a_n denote the number of Class 2 requests assigned to day n , $n = 1, \dots, N$ and a_0 denote the number of Class 2 appointments assigned to overtime. Then the following rather complicated expression describes the feasible assignments given demands d_1 and d_2 :

$$\begin{aligned} A_{(s_1, \dots, s_N, d_1, d_2)} = & \{(a_0, \dots, a_N) \mid a_0 \geq 0; a_1 = (M - s_1 - d_1)^+; \\ & 0 \leq a_n \leq M - s_n, n = 2, \dots, N; \sum_{n=0}^N a_n = d_2\}. \end{aligned}$$

An interpretation of these constraints follows:

1. The condition $a_0 \geq 0$ means that there is no bound on the number of units served through overtime.
2. The condition $a_1 = (M - s_1 - d_1)^+$ means that if after scheduling Class 1 demand for day 1 (tomorrow) there is any capacity available, as much Class 2 demand as possible will be assigned to day 1. Since it is least expensive to schedule demand on day 1, any optimal policy will use this assignment. Moreover it is the only way in which Class 1 demand explicitly interacts with Class 2 demand.
3. Since $M - s_n$ represents the number of available appointment slots on day n , the condition $0 \leq a_n \leq M - s_n$ constrains the number of Class 2 appointments that can be assigned on day n .
4. The condition $\sum_{n=0}^N a_n = d_2$ means that all Class 2 demand has to be assigned to appointment slots (including overtime).

This is one of the few examples in the book where the action varies with the state. This complicates coding because it requires deriving and storing or generating action sets as needed. Note that there is no decision to make for Class 1 appointments because they are served in regular time if capacity is available and otherwise through overtime.

Costs: Costs are incurred on the basis of the number of appointments booked in overtime time, C , and the number of Class 2 appointments booked on day n , c_n . That is

$$c((s_1, \dots, s_N, d_1, d_2), (a_0, \dots, a_N)) = C(d_1 - (M - s_1))^+ + Ca_0 + \sum_{n=1}^N c_n a_n.$$

In this expression, the first term corresponds to overtime charges for Class 1 appointments where $M - s_1$ denotes the number of free spots on day 1. If d_1 exceeds this number, overtime costs are incurred. Recall that it is assumed Class 1 appointment requests incur no cost if assigned to regular time on day 1. Note that the model is formulated in terms of a cost $c(s, a)$ instead of a negative reward $-r(s, a)$.

Transition probabilities: Transition probabilities can be expressed as:

$$\begin{aligned} p((s'_1, \dots, s'_N, d'_1, d'_2) | (s_1, \dots, s_N, d_1, d_2), (a_0, \dots, a_N)) \\ = \begin{cases} p_{d'_1} q_{d'_2}, & s'_n = s_{n+1} + a_{n+1}, n = 1, \dots, N-1, s'_N = 0 \\ 0, & \text{otherwise.} \end{cases} \end{aligned}$$

Transition probabilities are simpler than they appear. When there is a transition from today to tomorrow, after scheduling appointment requests, tomorrow's state becomes today's state and the state on day N becomes 0 since no appointments were previously scheduled on that day. The only probabilistic term corresponds to the arrival of Class 1 and Class 2 appointment requests today.

The Bellman equation

The Bellman equation for the infinite horizon discounted cost minimization version is given by:

$$\begin{aligned} v((s_1, \dots, s_N, d_1, d_2)) = \min_{(a_0, \dots, a_N) \in A_{(s_1, \dots, s_N, d_1, d_2)}} & \left\{ C \left((d_1 - (M - s_1))^+ + a_0 \right) \right. \\ & \left. + \sum_{n=1}^N c_n a_n + \lambda \sum_{j=0}^J \sum_{k=0}^K p_j q_k v((s_2 + a_2, \dots, s_N + a_N, 0, j, k)) \right\} \end{aligned} \quad (9.42)$$

for all $(s_1, \dots, s_N, d_1, d_2) \in S$. Note that the argument of $v(\cdot)$ inside the “min” represents tomorrow's state after allocating appointment requests that arrived today. The probabilities p_j and q_k only impact the last two components of the state vector.

The challenge faced by the scheduler is to determine on which day to assign Class 2 demand so as to retain sufficient capacity for future random Class 1 demand and, at the same time, not waste capacity.

Scheduling in practice presents many challenges, most notably anticipating future demand when scheduling current demand. Schedulers often act myopically and schedule appointments at the earliest possible date instead of trying to reserve capacity to meet future demand. The greedy policies based on approximations to the optimal value function do the latter.

9.6.1 A numerical example

This section formulates a small numerical example chosen so that an ϵ -optimal solution can be obtained without resorting to approximations. Let $N = 3, M = 4, J = K = 4$ so that the model has $5^3 \times 5 \times 5 = 3125$ states. The number of actions in each state varies from 1 to 35²³ so that the number of state-action pairs is large²⁴. Clearly increasing any of these quantities to practical levels will result in a model that requires solution by approximation.

A comparison of ϵ -optimal policies and values found with value iteration to those found using LSVI with linear and quadratic approximations follows. Results are for an instance with $\lambda = 0.95, c_n = 2n, C = 20, \mathbf{p} = (0, 0.2, 0.2, 0.3, 0.3)$ and $\mathbf{q} = (0.3, 0, 0, 0.3, 0.4)$. The arrival probabilities were chosen so that the total expected daily demand of $5.2 = 2.7 + 2.5$ exceeds the capacity of 4, making strategic scheduling a necessity. Results and interpretations appear in the following sections. Because of the complexity of the action set, policy iteration and linear programming are left to the reader.

9.6.2 Value iteration without approximation

Value iteration, using the recursion implicit in iterating the Bellman equation (9.42), was the easiest to code²⁵ and as well converged quickly. For this instance, the iterates of value iteration achieved the stopping criterion $s\mathbf{p}(\mathbf{v}' - \mathbf{v}) < 0.0001$ in 11 iterations. Columns 2 and 3 of Table 9.1 gives actions and values obtained using value iteration for selected states.

To interpret these results, consider the case $s = (1, 1, 0, 2, 4)$. This means that there are 3 units of regular-time capacity available on days 1 and 2 and 4 units available on day 3. Since the Class 1 demand is 2, it can be scheduled on day 1 leaving 1 unit of capacity available for Class 2 demand. Thus 3 units of Class 2 demand remain to be scheduled. The greedy action based on the ϵ -optimal value function, assigns one unit to day 2 and two units to day 3. Thus at the next decision epoch, $(s'_1, s'_2, s'_3) = (2, 2, 0)$.

²³In state $(0, 0, 0, 0, 4)$ there are 35 actions and in any state with $d_2 = 0$ or no available capacity, there is only one action.

²⁴It is left as an exercise to determine the exact number of actions.

²⁵Generating the set of actions for each state and manipulating arrays was challenging to code. Complexities included a “feature” of R that converted one-dimensional matrices to vectors.

State (s_1, s_2, s_3, d_1, d_2)	ϵ -optimal action	ϵ -optimal value	Linear approx. action	Linear approx. value
(2,1,2,4,2)	(0,0,1,1)	63.39	(0,0,0,2)	66.06
(2,3,0,2,4)	(0,0,0,4)	46.64	(0,0,0,4)	48.11
(1,1,0,2,4)	(0,1,1,2)	28.98	(0,1,0,3)	31.45
(1,2,2,2,3)	(0,1,0,2)	30.58	(0,1,0,2)	33.09
(0,0,0,4,4)	(0,0,2,2)	30.98	(0,0,0,4)	33.04
(0,0,0,2,4)	(0,2,1,1)	19.22	(0,2,0,2)	19.82

Table 9.1: This table gives results for the value iteration solution (columns 2 and 3) and that found using LSVI with a linear value function approximation (columns 4 and 5) for the instance described in the text. The first component of the action, a_0 , corresponds to Class 2 demand assigned to overtime and the next three components of the action (a_1, a_2, a_3) represent the number of units of Class 2 demand scheduled on day $i = 1, 2, 3$.

In contrast, the greedy policy based on the linear approximation schedules the 3 units of Class 2 demand to day 3, perhaps causing challenges in the future.

9.6.3 LSVI with linear value function approximation

LSVI was used to obtain parameter estimates based on a linear architecture and basis functions that are linear in the state. That is,

$$\hat{v}((s_1, s_2, s_3, d_1, d_2)) = \beta_0 + \beta_1 s_1 + \beta_2 s_2 + \beta_3 s_3 + \beta_4 d_1 + \beta_5 d_2. \quad (9.43)$$

The advantages of using this approximation are:

1. It is easy to obtain parameter estimates using readily available linear regression packages.
2. The number of quantities to approximate and store is reduced from the 3125 components of \mathbf{v} (in its tabular representation) to the six components of $\boldsymbol{\beta}$.
3. The parameter estimates provide insight into the effect demand and capacity have on cost. In (9.43), $\beta_i, i = 1, 2, 3$ represents the *marginal cost* of one less unit of capacity on day i , and β_4 and β_5 represent the marginal costs of an extra unit of Class 1 and Class 2 demand.
4. Greedy actions corresponding to the approximate value function can be easily determined.
5. Parameter estimates for this approximation can also be found using linear programming.

LSVI²⁶ as stated in Algorithm 9.3 was implemented as follows. It was initialized

²⁶LSVI was used instead of LSPI because it was simpler to code and converged quickly.

with $\beta = \mathbf{0}$ and $\epsilon = 0.0001$. Steps 2(a) and 2(b) were implemented component-wise instead of in vector form. Step 2(a) used representation (9.43) instead of the matrix \mathbf{B} . The right hand side of step 2(b) becomes

$$\min_{(a_0, a_1, a_2, a_3) \in A_{(s_1, s_2, s_3, d_1, d_2)}} \left\{ c((s_1, s_2, s_3, d_1, d_2), (a_0, a_1, a_2, a_3)) + \lambda \sum_{j=0}^4 \sum_{k=0}^4 p_j q_k \hat{v}((s_2 + a_2, s_3 + a_3, 0, j, k)) \right\}. \quad (9.44)$$

After substituting the linear approximation, the summation above becomes

$$\begin{aligned} & \sum_{j=0}^4 \sum_{k=0}^4 p_j q_k \hat{v}((s_2 + a_2, s_3 + a_3, 0, j, k)) \\ &= \sum_{j=0}^4 \sum_{k=0}^4 p_j q_k (\beta_0 + \beta_1(s_2 + a_2) + \beta_2(s_3 + a_3) + \beta_3 0 + \beta_4 j + \beta_5 k) \\ &= \beta_0 + \beta_1(s_2 + a_2) + \beta_2(s_3 + a_3) + \beta_4 E(D_1) + \beta_5 E(D_2), \end{aligned} \quad (9.45)$$

where the random variables D_1 and D_2 represent the amount of Class 1 and Class 2 demand, respectively. Since after substitution, several of the expressions in (9.44) do not involve the action explicitly, the greedy action based on the linear approximation chooses

$$(\hat{a}_0, \hat{a}_1, \hat{a}_2, \hat{a}_3) \in \arg \min_{(a_0, a_1, a_2, a_3) \in A_{(s_1, s_2, s_3, d_1, d_2)}} \left\{ Ca_0 + c_1 a_1 + (c_2 + \lambda \beta_1) a_2 + (c_3 + \lambda \beta_2) a_3 \right\} \quad (9.46)$$

The example was solved using the representation (9.44) to allow easy generalization to other approximations for $\hat{v}(\cdot)$. LSVI converged in 13 iterations. Curiously, convergence of the quantity $\|\beta' - \beta\|_2$ was not monotone; at one iteration the norm of the difference $\beta' - \beta$ exceeded that at a previous iteration. This is consistent with the observation above that the LSVI operator need not be a contraction mapping.

The linear approximation to the optimal value function obtained by substituting the value of $\hat{\beta}$ is given by

$$\hat{v}((s_1, s_2, s_3, d_1, d_2)) = -43.98 + 12.59s_1 + 6.54s_2 + 4.28s_3 + 12.60d_1 + 8.98d_2. \quad (9.47)$$

Observe that all coefficients (except the constant) are positive meaning that the greater the occupancy or demand, the greater the expected discounted cost. More importantly, the expected discounted cost of an extra unit of occupancy decreases from day 1 to day 3 and an extra unit of Class 1 demand contributes more to the expected discounted cost than a unit of Class 2 demand because of the greater flexibility in scheduling Class 2 demand.

The structure of a greedy policy based on the linear approximation can be obtained as follows. Recall that the daily capacity is $M = 4$ units, overtime cost is $C = 20$ and the cost of scheduling a unit of Class 2 demand on day $n = 1, 2, 3$ is $2n$. Substituting the parameter estimates in (9.47) into (9.46) implies that

$$(\hat{a}_0, \hat{a}_1, \hat{a}_2, \hat{a}_3) \in \arg \min_{(a_0, a_1, a_2, a_3) \in A_{(s_1, s_2, s_3, d_1, d_2)}} \{20a_0 + 2a_1 + 10.21a_2 + 10.06a_3\}. \quad (9.48)$$

where $10.21 = 4 + 0.95 \cdot 6.54$ and $10.06 = 6 + 0.95 \cdot 4.28$. The action set can be expressed as

$$A_{(s_1, s_2, s_3, d_1, d_2)} = \left\{ (a_0, a_1, a_2, a_3) \mid \begin{array}{l} a_0 \geq 0; a_1 = (4 - s_1 - d_1)^+; \\ 0 \leq a_n \leq 4 - s_n, n = 2, 3; \sum_{n=0}^3 a_n = d_2 \end{array} \right\}. \quad (9.49)$$

Observe further that the arg min only involves the current state through its constraints.

As a result of (9.48) and (9.49) it follows that the greedy action based on the linear approximation has the following structure:

In each state in which $d_2 > 0$, assign as much Class 2 demand as possible to day 1, then assign as much as possible to day 3, then assign as much as possible to day 2. Only assign Class 2 demand to overtime when there is no other capacity available.

This policy can be summarized succinctly as: after using up capacity on day 1, fill capacity starting from the end of the booking horizon and working backwards to the current day. Note that this characterization avoids the need to compute the optimal action in each state although it is done so in Table 9.1 for comparison.

9.6.4 LSVI with quadratic value function approximation

LSVI was also applied using a value function approximation of the form:

$$\begin{aligned} \hat{v}((s_1, s_2, s_3, d_1, d_2)) &= \beta_0 + \beta_1 s_1 + \beta_2 s_2 + \beta_3 s_3 + \beta_4 d_1 + \beta_5 d_2 \\ &\quad + \beta_6 s_1^2 + \beta_7 s_2^2 + \beta_8 s_3^2 + \beta_9 d_1^2 + \beta_{10} d_2^2. \end{aligned}$$

A more general quadratic model would include 10 additional cross product terms of the form $s_i s_j$, $s_i d_j$ and $d_1 d_2$. Since the model is linear in the parameters, they can be easily estimated using linear regression. In contrast to the approximation in the previous section, the marginal costs vary with the state vector. For example the marginal cost

of an additional unit of Class 2 demand would be $\beta_5 + 2\beta_{10}d_2$. Again LSVI converged in 13 iterations and the norms of the differences of successive parameter estimates were not monotone.

The estimated value function is given by

$$\begin{aligned}\hat{v}((s_1, s_2, s_3, d_1, d_2)) = & -34.93 + 5.93s_1 + 3.55s_2 + 1.92s_3 + 5.93d_1 + 5.07d_2 \\ & + 1.68s_1^2 + 0.85s_2^2 + 0.70s_3^2 + 1.68d_1^2 + 1.04d_2^2.\end{aligned}\quad (9.50)$$

Using the same approach as in the case of the linear value function approximation, substituting $\hat{v}(s)$ into the right hand side of the Bellman equation, the analogous expression to (9.48), after some algebra, becomes

$$20a_0 + 2a_1 + (7.37a_2 + 0.81(s_2 + a_2)^2) + (7.82a_3 + 0.66(s_3 + a_3)^2). \quad (9.51)$$

Given a state, choosing the action to minimize this expression gives the greedy action. Since the action set is finite, this minimization can be done by enumeration. For example, in state $(2, 1, 1, 2, 1)$ there are three feasible actions as shown in Table 9.2. Evaluating the above expression for each action shows that the greedy action is $(0, 0, 1, 0)$; that is, assign the single unit of Class 2 demand to day 2. This contrasts with the action based on the linear approximation which would assign this unit of Class 2 demand to day 3.

Action	Value of (9.51)
$(1, 0, 0, 0)$	20.00
$(0, 0, 1, 0)^*$	11.21
$(0, 0, 0, 1)$	11.27

Table 9.2: Greedy action choice in state $(2, 1, 1, 2, 1)$ based on quadratic value function approximation. The greedy action is indicated with an asterisk.

9.6.5 Comparison of policies and values

The three left most columns of Table 9.1 above give the action chosen by the ϵ -optimal policy and its value in some selected high demand states. Observe that in these states, the policy does not assign Class 2 demand to the earliest day possible, instead it **reserves** capacity for future Class 1 demand. For example, in state $(2, 1, 2, 4, 2)$, all day 1 capacity is used by Class 1 demand and the two units of Class 2 demand are split between days 2 and 3. In state $(0, 0, 0, 4, 4)$, all Class 1 demand is served on day 1 and the 4 units of Class 2 demand are split between days 2 and 3. Moreover in these states Class 2 demand is never served with overtime.

Columns 4 and 5 of Table 9.1 show the greedy action obtained using the linear value function approximation and the value \hat{v} . Observe that in several of these states, the action based on the linear approximation differs from that using the ϵ -optimal policy.

In cases where it differed, it assigned the Class 2 demand to later appointments²⁷. For example in state $(0, 0, 0, 4, 4)$ the policy chosen using the linear approximation assigned **all** Class 2 demand to day 3 while the ϵ -optimal policy split it between days 2 and 3. Overall, the action based on the linear approximation differed from that chosen by the ϵ -optimal policy in 632 out of 3125 states.

Remarkably, the greedy actions chosen using the quadratic approximation were **identical** to those corresponding to the ϵ -optimal policy. Moreover the RMSE of the difference between the ϵ -optimal value and that found using a linear approximation was 1.99 and the RMSE of the difference between the ϵ -optimal value and that found using a quadratic approximation was 1.11. Hence the quadratic approximation fit was considerably better and might be appropriate in larger instances of the model.

9.6.6 Further insights

This section provides some additional insights that can be gained by investigating other aspects of the model and solution methods.

Using overtime strategically

To test whether overtime was ever used strategically, overtime costs were chosen to be class specific. Setting the overtime cost for Class 1 to 40 and for Class 2 to 10, made it more essential to reserve capacity for Class 1 demand. For example the ϵ -optimal policy in state $(2, 3, 0, 2, 4)$ chose action $(2, 0, 0, 2)$. This means that it assigns 2 units of Class 2 demand to overtime and 2 units as late as possible, that is to day 3. Thus the ϵ -optimal policy uses overtime strategically by reserving capacity for Class 1 demand.

Note that the greedy action corresponding to the linear value function approximation remained $(0, 0, 0, 0, 4)$ but the quadratic approximation again identified the optimal action. This provides further support for using a quadratic approximation.

State sampling

In the above analysis, LSVI approximations were based on estimating parameters using value function updates for all states. However, good results may be possible if LSVI is applied to only a subset of the states. More importantly, in larger models, obtaining approximations by updating the above recursion in all states would be computationally prohibitive.

The advantage of state sampling is that it reduces the number of times that the Bellman operator

$$L\hat{\mathbf{v}}(s) = \min_{a \in A_s} \left\{ c(s, a) + \lambda \sum_{j \in S} p(j|s, a) \hat{v}(j) \right\}$$

²⁷This observation is similar to the policy found by Patrick et al. [2008] using linear programming with a linear value function approximation.

needs to be evaluated at each iteration of LSVI.

The following offline approach was used to apply LSVI using a subset of S . A subset of states was selected by:

1. specifying the fraction of states to sample,
2. randomly sampling the specified number of states,
3. discarding duplicates²⁸, and
4. applying LSVI as described in Section 9.3.1 using the designated sample.

Thus after selecting a random subset of states, LSVI uses it for all iterations. The next two chapters consider online versions of these algorithms, which sample states sequentially and avoid computing the expectation $\sum_{j \in S} p(j|s, a) \hat{v}(j)$.

The following analysis is based on using the above quadratic approximation. It applies LSVI to a large number of samples, in which the fraction of states sampled was varied between 0.1 to 0.9. The quality of fit was compared on the basis of

$$\text{RMSE} = \left(\frac{1}{11} \sum_{i=0}^{10} (\hat{\beta}_i - \hat{\beta}_i^{\text{all}})^2 \right)^{0.5},$$

where $\hat{\beta}_i^{\text{all}}$ denotes the parameter estimates based on using all states.

Results of this experiment appear in Figure 9.15, which shows two things:

1. the average quality of the fits improve with sample size, and
2. there is considerable variability between samples.

We speculate that by instead choosing the state samples to be representative of the entire state space, much of this between sample variability can be reduced. Moreover, it appears that the accuracy based on sampled fractions greater than 0.6 are similar.

The quadratic value function approximation based on a random sample of half of the states is given by

$$\begin{aligned} \hat{v}((s_1, s_2, s_3, d_1, d_2)) = & -36.86 + 5.61s_1 + 3.75s_2 + 2.21s_3 + 7.21d_1 + 5.12d_2 \\ & + 1.73s_1^2 + 0.80s_2^2 + 0.70s_3^2 + 1.37d_1^2 + 1.03d_2^2. \end{aligned} \quad (9.52)$$

Comparing it to (9.50) obtained using **all** states shows that the estimates are reasonably close and the order of magnitude of the corresponding parameters is similar. Derivation of the corresponding greedy policy and comparison to that obtained using all states is left to the reader.

²⁸Since there is no variability in this approach other than state sampling, each replication of state s would yield the same value for $L\hat{v}(s)$. Thus, it was unnecessary to evaluate the same state more than once. In a more general simulation approach, this would not be the case.

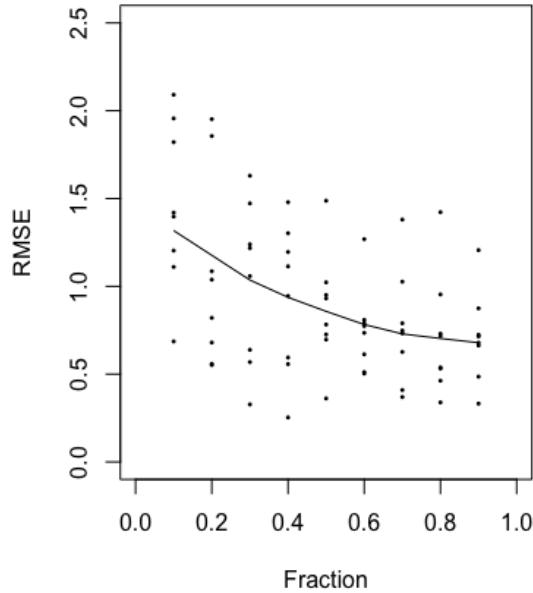


Figure 9.15: RMSE as a function of the fraction of states sampled over multiple state samples. The dots correspond to a specific sample and the line is a Lowess smoothing of the individual RMSEs.

Numerical example: Neural network approximation

Obtaining good results using a neural network approximation proved challenging when using the package “neuralnet” in R. When incorporated in LSVI, the algorithm either oscillated and did not satisfy a stopping criterion or the parameter estimation algorithm did not converge. Several alternatives were explored. Most notably, when the ϵ -optimal value function on the entire state space was approximated by several different neural network structures, the greedy policy was similar to that found with a linear value function approximation. In this application, we believe that neural networks **do not** provide an attractive approximation architecture but leave it to the reader to prove the supposition false.

Approximate linear programming

An analysis based on linear programming is not presented here. The main challenge was to construct constraint matrices. This model has been analyzed in considerable detail in [Patrick et al. 2008]. Therein, the linear program was solved using column generation on the dual, which contained few rows and many columns. The coefficients for the linear approximation were obtained from the corresponding primal solution.

This work identified the same greedy policy as described above for the LSVI solution of the linear approximation.

9.7 Application: Golf strategy

Often, in real-life situations, transition probabilities, reward functions and the policy used are not known. This section provides a case study of the estimation of a value function based on multiple realizations of finite sequences of states and rewards that terminate when a *goal state* is reached. This case study, drawn from the game of golf, illustrates this approach and provides a transition to the next two chapters.

9.7.1 Value functions and strokes gained in golf

This section describes dynamic programming based concepts developed by Broadie [2014] and now widely used in all levels of competitive golf.

Golf as a Markov decision process

Golf is a game where players strive to propel a 1.68-inch diameter golf ball into a 4.25-inch diameter cup (often referred to as a “hole”) in the fewest number of strokes. A typical golf course comprises 18 holes, each featuring a tee box, a putting green (where the cup is located), a fairway, and possibly some rough, sand traps, obstructed areas, out-of-bounds areas, and natural obstacles like woods, deserts, or water. The distance from the tee to the cup typically ranges from 100 to 600 yards.

Play begins from the tee, and subsequent shots are played from the point where the previous shot came to rest. A penalty stroke is added if a ball is lost or deemed unplayable. Playing **each hole** in a round of golf may be viewed as a Markov decision process as follows.

Decision epochs: Decision epochs correspond to the instance before a shot is hit:

$$T = \{1, 2, \dots\}.$$

Note that an infinite horizon formulation is appropriate because the number of shots required to complete a hole of golf is not fixed; it is determined by the quality of play. In theory it could take an arbitrarily large number of strokes to hit the ball into the hole from any location of the golf course. In reality this is not supported by data but often it might seem like the case to a frustrated golfer²⁹.

²⁹In men’s professional (PGA) golf, the highest recorded hole score to date is 16 in 2011 by Kevin Na at the Valero Open.

States: The state represents the distance from the hole $\delta \in \mathcal{D}$ and the terrain type $\tau \in \mathcal{T}$ (fairway, rough, etc.) where the ball lies immediately before a shot is taken. Note that the state $(0, \text{green})$ is the zero-reward absorbing state corresponding to the ball being “in the hole”. Therefore,

$$S = \{\mathcal{D} \times \mathcal{T}\}.$$

On most golf courses $\mathcal{D} = \{100, 101, \dots, 600\}$ and

$$\mathcal{T} = \{\text{tee, fairway, rough, sand trap, hazard, obstructed, green}\}.$$

Actions: Each action represents the type of shot a player tries to hit and its intended target. These vary by location, wind and course configuration.

$$\begin{aligned} A_{(\delta, \tau)} &= \{\text{Possible targets and shot types from } (\delta, \tau)\} \\ A_{(0, \text{green})} &= \{a_0\}, \end{aligned}$$

where a_0 indicates that the hole is completed.

Elements of $A_{(\delta, \tau)}$ can be thought of as capturing the choice of a golfer to aim the ball in a certain direction, to hit the ball at a certain speed and angle, and to attempt to give the shot a particular curvature. Note that at this level, shot types are not described explicitly because they vary greatly with the skill of the player and the terrain. For example, if the ball is behind a tree then the player may hit a recovery shot. If the ball is on the green, the player will putt. In the fairway, the player may pick a target 10 feet short of the hole and attempt to hit a low draw³⁰.

Transition probabilities: The transition probabilities $p((\delta', \tau') | (\delta, \tau), a)$ are determined from a probability distribution of shot outcomes given the target and shot type. One would expect that shots from closer to the hole will have less variable outcomes than shots from greater distances.

Since the hole terminates when $(0, \text{green})$ is reached, $p((0, \text{green}) | (0, \text{green}), a_0) = 1$.

The assumption that the transition probability depends only on the current state and action makes sense since given the location of the ball and its terrain, the outcome of the current shot should not depend on the outcome of previous shots³¹.

Rewards: It requires one stroke to hit a shot so that:

$$r((\delta, \tau), a, (\delta', \tau')) = -1 \quad \text{for all } (\delta, \tau) \in S$$

and

$$r((0, \text{green}), a_0, (0, \text{green})) = 0.$$

³⁰A shot that curves from the right to the left for a right-handed golfer.

³¹Unless of course the golfer is really frustrated after hitting a poor shot, since mood may affect execution of future shots.

Because the objective in golf is to minimize the number of shots on each hole, the reward represents the “cost” of a shot. Because this is formulated as an infinite horizon model, the state (0, green) is the zero-cost (reward-free) absorbing state.

Note that it is preferable to model this as a minimum cost problem, in which case $r((\delta, \tau), a, (\delta', \tau')) = -1$ is replaced by $c((\delta, \tau), a, (\delta', \tau')) = 1$.

Value functions in golf

Define the value of each location as the expected number of shots to “hole out”, that is get the ball into the cup, from that location. Both the distance and terrain affect this value, that is

$v(\delta, \tau)$ = the expected number of strokes to hole out from distance δ and terrain τ .

Of course $v(\delta, \tau) \geq 0$ and $v(0, \text{green}) = 0$. Moreover for a fixed τ , $v(\delta, \tau)$ should be non-decreasing in δ .

As an example, for an average male professional golfer³²,

$$v(100, \text{fairway}) = 2.80 \quad \text{and} \quad v(100, \text{rough}) = 3.02.$$

This means that the expected number of shots for an average male professional golfer to hole out from 100 yards in the fairway is 2.80 and from 100 yards in the rough is 3.02. Thus, on average a player adds 0.22 shots if a shot ends up in the rough as opposed to the fairway. This may not sound like much but over a tournament made up of four 18 hole rounds, missing the fairway several times can have a significant impact on the player’s final score.

The value function can be overlaid on a picture of a hole to show the expected number of shots from each location (see Figure 9.16). Such a figure can also guide player strategy as discussed below. Knowing this value from every distance and terrain is fundamental to developing a performance metric referred to as *strokes gained*, described below.

A case study

The following case study is based on Marty’s collaboration with the University of British Columbia (UBC) women’s golf team. The objective of this study was to establish value functions for female golf team members. Unlike in the case of male professional golfers, such specialized information was not available at the time of the study.

The following approach was used to estimate values of shots from the fairway for distances between 10 and 350 yards from the hole. Note that the data contained shots from all terrains, however for illustrative purposes only shots from the fairway are analyzed here. Therefore the component τ will be dropped from the state description

³²Broadie [2014] p. 85.



Figure 9.16: Graphical representation of golf value functions for male professional golfers based on Broadie [2014]. For example, on average it requires 2.8 shots to hole out from anywhere on the black arc at 100 yards from the green.

for the remainder of this section. Hence, the objective of this analysis is to estimate $v(\delta)$; the expected number of shots to hole out from distance δ on the fairway.

The complete data set, which contained the starting location (state) and ending location (state) for 10,100 shots executed during the 2016/2017 golf season, was entered by team members in a custom application. It was manipulated to create a data set consisting of a large number of pairs of distances and number shots to “hole out” from those distances. For example, the data point $(\delta, v) = (200, 4)$ indicates that for a particular player, on a particular hole in a particular round, it required 4 shots to “hole out” from 200 yards on the fairway.

A value function approximation

The value function was approximated by a linear combination of the basis functions $b_0(\delta) = 1, b_1(\delta) = \delta, b_2(\delta) = \delta^2$ and $b_3(\delta) = \delta^3$. That is, it was approximated by a cubic polynomial. In contrast to other examples in this chapter based on an MDP model,

an observed value was represented by

$$v(\delta) = \beta_0 + \beta_1\delta + \beta_2\delta^2 + \beta_3\delta^3 + \epsilon, \quad (9.53)$$

where ϵ represented an unobservable random disturbance.

The reason for developing an approximation using basis functions as opposed to separate estimates of $v(\delta)$ for each δ based on the average of shots from that distance was to account for data sparsity and to obtain a value function approximation that was non-decreasing in distance. For some distances, there were many records while for others there were none. Moreover, it made sense that the further from the hole, the greater the number of strokes it should take to hole out so that $v(\delta)$ should be non-decreasing in δ .

Using least squares regression resulted in the following fitted model:³³

$$\hat{v}(\delta) = 1.95 + 0.015\delta - 0.000040\delta^2 - 0.000000049\delta^3. \quad (9.54)$$

The data together with the fitted model are shown in Figure 9.17. Note that more complicated models or including random effects had little effect on estimated values.

Based on the fitted model, $\hat{v}(100) = 3.11$, which is 0.31 higher than for a male professional golfer from that location or equivalent to the value for a male professional from 185 yards.

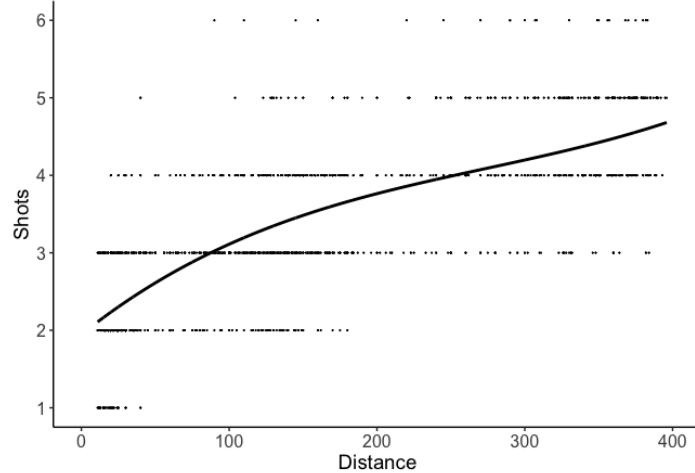


Figure 9.17: Data and fitted model for (9.54). The data values are discrete since they represented the number of shots required to “hole out” from each distance on the fairway. Note that the fitted function is monotone increasing.

³³An alternative is to fit a Poisson regression since the dependent variable represents a count. Thanks to Antoine Sauré for this suggestion.

Golf strategy based on value functions

In the context of the Markov decision process framework, the value functions can be used to determine strategy. Strategic choices in golf include anticipated shot distance and shot type. Outcomes are impacted by shot dispersion patterns represented by probability distributions. Shorter shots have less dispersion than longer shots.

As an example, consider the option of a shot from the fairway when the hole is 240 yards away and there is a water hazard in front of the green. Suppose the player considers two choices. Choice 1 is to “go for the green”, which means to be aggressive and try to reach green in one shot. Choice 2 is to “lay up”, which means to be conservative and hit a shorter shot that most likely avoids the water, but will require a subsequent shot to get on the green.

To simplify exposition, suppose that under Choice 1 if the player aims at the green and lands on it, the ball ends up 50 feet from the hole which occurs with probability p . The value from that point would be $v(50, \text{green}) = 2.14$ ³⁴. However, if the ball ends up in the water, which occurs with probability $1 - p$, the player incurs a one stroke penalty and must hit his next shot from 60 yards in the fairway where the value is $v(60, \text{fairway}) = 2.70$. Thus the expected value of the action “go for the green” is

$$v_1 = 2.14p + (2.70 + 1)(1 - p) = 3.70 - 1.56p.$$

Suppose the alternative action is to aim for a target 80 yards from the green where there is a 0.9 chance of being close to that target where $v(80, \text{fairway}) = 2.75$ and a 0.1 chance of being in the rough at the same distance where $v(80, \text{rough}) = 2.96$. The expected value of the action “lay up” is $v_2 = 2.77$. Since the goal is to minimize the number of strokes taken to hole out, the decision to go for the green is optimal if $v_1 \leq v_2$, otherwise the golfer should lay up. Therefore noting that $3.70 - 1.56p = 2.77$ implies $p = 0.596$, the optimal decision rule is

$$d(240, \text{fairway}) = \begin{cases} \text{go for green,} & p \geq 0.596 \\ \text{lay up,} & p < 0.596. \end{cases}$$

Hence if the player thinks there is greater than a 59.6% chance of hitting the green, then he³⁵ should go for the green; otherwise he should lay up. In reaching this decision, a player must take into account factors including the weather, the lie in the fairway and his stress level.

Using value functions: Strokes gained

Suppose a golfer scores a 3 on a par 4 hole. To which shot (or shots) can this outcome be attributed? Was it the result of an outstanding drive, a precise approach shot or

³⁴Distance on the green are measured in feet and distances from other terrain are measured in yards.

³⁵The calculations above use men’s professional values from p. 85 of Broadie [2014].

an excellent putt or was it some combination of these shots? This is the question that the concept of *strokes gained* seeks to answer.

Determining which action contributes to success in an episodic model is referred to in the machine learning literature to as the *the credit assignment problem*³⁶. This challenge arises in domains such as the games chess and Go, robotic navigation and natural language processing. Value functions and state-action value functions provide a way to approach this problem in general.

Solving the credit assignment problem is crucial for effective learning. If a learning system cannot assign credit (or blame) to actions, then it will not be able to learn which actions are beneficial and which are detrimental. Without being able to make this distinction, the learner will not be able to improve performance over time.

Returning to golf, *strokes gained* measures the difference between the expected value of a shot and its result. Let X_D be a random variable denoting the distance of the ball to the hole after the current shot and X_T be a random variable denoting the terrain of the ball after the current shot. Also, let

(δ', τ') = Actual location of the ball after the current shot

(δ, τ) = Actual location of the ball immediately prior to the current shot.

Then strokes gained, SG³⁷, is defined as

$$\text{SG}(\delta, \tau) = E_{(\delta, \tau)}[v(X_D, X_T)] - v(\delta', \tau'). \quad (9.55)$$

Recalling the MDP formulation of golf above, the policy evaluation equation

$$v(s) = r(s) + \sum_{j \in S} p(j|s)v(j)$$

expressed in terms of random variables and costs becomes

$$v(\delta, \tau) = 1 + E_{(\delta, \tau)}[v(X_D, X_T)]. \quad (9.56)$$

So rearranging terms and substituting (9.56) into (9.55) yields

$$\text{SG}(\delta, \tau) = v(\delta, \tau) - (v(\delta', \tau') + 1) \quad (9.57)$$

The first quantity on the right hand side of (9.57) denotes the expected number of shots to hole out prior the current shot. The second quantity is the expected number of shots to hole out from where the current shot ended up plus the “cost” of 1 stroke to reach this point.

³⁶A nice discussion of this issue appears in Bennett [2023].

³⁷see Broadie [2014]

To interpret this quantity, recall that the goal in golf is to *minimize* the expected number of strokes to hole out. If the player hits a better than average shot, $v(\delta', \tau') + 1$ will be less than $v(\delta, \tau)$ and strokes gained will be positive. If the shot outcome is worse than average, strokes gained will be negative. When a player hits an “average” shot, $v(\delta, \tau) = v(\delta', \tau') + 1$ so that strokes gained will be zero.

Example 9.13. To illustrate this concept consider the following scenario where values are based on the fitted curve (9.54). Suppose a women’s golf team member is at 275 yards in the fairway, then $v(275, \text{fairway}) = 4.09$. After hitting an average shot from this location, her strokes gained would be $4.09 - 1 = 3.09$. This is equivalent to a distance of 98 yards in the fairway, so an average shot to the fairway would be $275 - 98 = 177$ yards. Hitting the ball to 75 yards in the fairway results in a value of 2.88 and the corresponding strokes gained is $4.09 - 2.88 - 1 = 0.21$. If instead she hits the ball to 125 yards in the fairway, the value is 3.31 corresponding to a strokes gained of -0.22 . Note that hitting the ball into the rough or a sand trap at 125 yards would result in more negative strokes gained.

This calculation ignores the possibility that player may have made a strategic choice to lay up to specific distance so as to avoid a hazard.

From the credit assignment problem perspective, strokes gained averaged over several rounds identifies a player’s strengths and weaknesses and which areas need improvement. Hence, this information can be used to develop a practice plan. For example, if average strokes gained for putts in the range 8 to 12 feet is negative, then this is an area where the player needs to improve. This was the objective of the UBC golf team project.

9.8 Technical appendix: Regression and nonlinear optimization

To effectively apply and interpret value function approximation, familiarity with the basics of linear regression as described in this appendix is crucial.

9.8.1 Linear regression

Given observations of a dependent variable y_j and independent variables $x_{j,1}, \dots, x_{j,I}$ for $j = 1, \dots, J$, the objective of linear regression is to find a set of *regression coefficients*, *weights* or *parameters* β_i , $i = 0, \dots, I$ so that

$$f(x_{j,1}, \dots, x_{j,I}; \beta_0, \dots, \beta_I) := \beta_0 + \beta_1 x_{j,1} + \dots + \beta_I x_{j,I} \quad (9.58)$$

well approximates y_j for $j = 1, \dots, J$. This is referred to as a *linear* regression problem because $f(x_{j,1}, \dots, x_{j,I}; \beta_0, \dots, \beta_I)$ is a linear function of its parameters.

The standard approach³⁸ to estimation is to obtain parameters that minimize the sum of squared errors

$$g(\beta_0, \dots, \beta_I) := \sum_{j=1}^J (y_j - (\beta_0 + \beta_1 x_{j,1} + \dots + \beta_I x_{j,I}))^2 \quad (9.59)$$

$$= \sum_{j=1}^J g_j(\beta_0, \dots, \beta_I)^2 \quad (9.60)$$

where

$$g_j(\beta_0, \dots, \beta_I) := y_j - (\beta_0 + \beta_1 x_{j,1} + \dots + \beta_I x_{j,I})$$

for $j = 1, \dots, J$.

The function $g_j(\cdot)$ is introduced to provide more elegant expressions and simplify the derivation of Gauss-Newton iteration below. Note that this notation expresses dependence on observation $(y_j, x_{j,1}, \dots, x_{j,I})$ through the subscript j .

Parameters that minimize $g(\beta_0, \dots, \beta_I)$ are referred to as *least squares* or *ordinary least squares (OLS)* estimates. Using a matrix formulation leads to a closed form representation. Parameter estimates are often denoted by $\hat{\beta}_i, i = 0, \dots, I$.

Note that the statistical literature formulates the regression problem in the context of the statistical model

$$y_j = \beta_0 + \beta_1 x_{j,1} + \dots + \beta_I x_{j,I} + \epsilon_j, \quad (9.61)$$

where ϵ_j represents an *unobservable* random disturbance (or error) drawn from a distribution with mean 0 and constant variance σ^2 . Furthermore, it is usually assumed that $E[\epsilon_j \epsilon_k] = 0$ for $j \neq k$, that is, that the random disturbances are uncorrelated between observations. When in addition, the random disturbances are assumed to be normally distributed, the least squares estimates of $\beta_i, i = 0, \dots, I$ are also *maximum likelihood estimates* and consequently have many desirable theoretical properties that allow for statistical inference, such as the computation of confidence intervals and hypothesis testing. Usually, these statistical concepts are not taken into account in approximate dynamic programming.

When the errors are assumed to be normal, the above model is often expressed as

$$y_j \sim NID(\beta_0 + \beta_1 x_{j,1} + \dots + \beta_I x_{j,I}, \sigma^2), \quad (9.62)$$

where the expression $NID(\mu, \sigma^2)$ corresponds to a set of independent normally distributed random variables with mean μ and variance σ^2 . From this perspective, the mean of y_j varies from observation to observation and the variance is constant. The beauty of writing the model this way is that it allows generalization to other distributions such as Poisson or binomial, or different non-constant variance, or auto-correlated errors.

³⁸Other choices for $g(\cdot)$ can be used including sums of absolute values or other weighting functions that may downweight outlying observations.

9.8.2 Matrix formulation

The most elegant approach for analyzing regression models is through its matrix formulation. It is widely used in approximate dynamic programming and reinforcement learning, and leads to elegant formulae for predicted values and parameter estimates. To do so define the following vectors and matrices:

$$\mathbf{y} := \begin{bmatrix} y_1 \\ \vdots \\ y_J \end{bmatrix}, \quad \mathbf{X} := \begin{bmatrix} 1 & x_{1,1} & \dots & x_{1,I} \\ \vdots & \ddots & \dots & \vdots \\ \vdots & \ddots & \dots & \vdots \\ 1 & x_{J,1} & \dots & x_{J,I} \end{bmatrix}, \quad \boldsymbol{\beta} := \begin{bmatrix} \beta_0 \\ \vdots \\ \beta_I \end{bmatrix} \text{ and } \boldsymbol{\epsilon} := \begin{bmatrix} \epsilon_1 \\ \vdots \\ \epsilon_J \end{bmatrix}. \quad (9.63)$$

In (9.63), \mathbf{y} denotes the $J \times 1$ column vector of observations of the dependent variable, \mathbf{X} is the $J \times (I + 1)$ matrix of values of the independent variables, $\boldsymbol{\epsilon}$ denotes the $J \times 1$ column vector of random disturbances and $\boldsymbol{\beta}$ is the $(I + 1) \times 1$ column vector of parameters. It is also convenient to denote the j -th row of \mathbf{X} by the vector \mathbf{x}_j , that is

$$\mathbf{x}_j := (1, x_{j,1}, \dots, x_{j,I})$$

for $j = 1, \dots, J$. When written as a row vector, it will be expressed as \mathbf{x}_j^\top .

Using this matrix notation, the data generating model, which provides an elegant way of representing (9.61) for all j in a single equation, is given by

$$\mathbf{y} = \mathbf{X}\boldsymbol{\beta} + \boldsymbol{\epsilon}, \quad (9.64)$$

where $E[\boldsymbol{\epsilon}] = \mathbf{0}$ and $\boldsymbol{\Sigma} := \text{cov}[\boldsymbol{\epsilon}] = \sigma^2 \mathbf{I}$.

The squared error criterion can be expressed in matrix form as

$$g(\boldsymbol{\beta}) = \sum_{j=1}^J g_j(\boldsymbol{\beta})^2 = (\mathbf{y} - \mathbf{X}\boldsymbol{\beta})^\top (\mathbf{y} - \mathbf{X}\boldsymbol{\beta}) = \mathbf{y}^\top \mathbf{y} - 2\boldsymbol{\beta}^\top \mathbf{X}^\top \mathbf{y} + \boldsymbol{\beta}^\top \mathbf{X}^\top \mathbf{X}\boldsymbol{\beta}. \quad (9.65)$$

Note that $g(\boldsymbol{\beta})$ is often written as

$$g(\boldsymbol{\beta}) = \|\mathbf{y} - \mathbf{X}\boldsymbol{\beta}\|_2^2. \quad (9.66)$$

Computing the gradient of $g(\boldsymbol{\beta})$ and setting it equal to zero shows that $\hat{\boldsymbol{\beta}}$, the least squares estimator of $\boldsymbol{\beta}$, satisfies the “normal equation”

$$\mathbf{X}^\top \mathbf{X}\boldsymbol{\beta} = \mathbf{X}^\top \mathbf{y}. \quad (9.67)$$

When $\mathbf{X}^\top \mathbf{X}$ is invertible³⁹ the least square parameter estimates can be written in closed form as:

³⁹This is true when the columns of \mathbf{X} are linearly independent. Otherwise they are said to be *collinear*. In regression theory this phenomenon is referred to as *multi-collinearity*; several approaches, most notably ridge regression, have been developed to address this issue.

$$\hat{\boldsymbol{\beta}} = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{y} := \boldsymbol{\Gamma} \mathbf{y}. \quad (9.68)$$

Note that $\hat{\boldsymbol{\beta}}$ determined by either (9.67) or (9.68) can also be written as

$$\hat{\boldsymbol{\beta}} \in \arg \min_{\boldsymbol{\beta} \in \mathbb{R}^{I+1}} \|\mathbf{y} - \mathbf{X}\boldsymbol{\beta}\|_2. \quad (9.69)$$

The vector of predicted (fitted) values $\hat{\mathbf{y}}$ can be expressed as:

$$\hat{\mathbf{y}} = \mathbf{X}(\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{y} := \boldsymbol{\Pi} \mathbf{y}. \quad (9.70)$$

The matrices $\boldsymbol{\Gamma}$ and $\boldsymbol{\Pi}$ defined in (9.68) and (9.70) are fundamental to the discussion of iterative algorithms in this chapter. The matrix $\boldsymbol{\Pi}$ is sometimes referred to as the *hat matrix* because it maps the vector of observed values \mathbf{y} into the vector $\hat{\mathbf{y}}$ of fitted values; that is, it puts a “hat” on \mathbf{y} . It is also referred to as a *projection matrix* for reasons described below.

The following example illustrates this notation.

Example 9.14. Quadratic regression: Suppose the dependent variable y_i is represented by a quadratic function of the independent variable x_i written as

$$y_j = \beta_0 + \beta_1 x_j + \beta_2 x_j^2 + \epsilon_j$$

for $j = 1, \dots, J$. To form the matrix representation for this model, $\mathbf{y} = \mathbf{X}\boldsymbol{\beta} + \boldsymbol{\epsilon}$, set

$$\mathbf{X} = \begin{bmatrix} 1 & x_1 & x_1^2 \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \\ 1 & x_J & x_J^2 \end{bmatrix} \quad \text{and} \quad \boldsymbol{\beta} = \begin{bmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \end{bmatrix}.$$

There is no necessity that the x_j be distinct. Also, they can be pre-specified or randomly sampled. The subject *design of experiments* addresses how to specify these values so as to obtain the most precise estimates.

Least squares geometry

A beautiful geometric representation underlies least squares estimation and is fundamental to the LSVI and LSPI algorithms.

Since $\Pi^2 = \Pi$, Π is a projection matrix and it maps \mathbf{y} onto the space of vectors spanned by the columns of \mathbf{X} denoted $\text{col}(\mathbf{X})$ (see Figure 9.18). In other words $\Pi\mathbf{y}$ is the linear combination of the columns of \mathbf{X} that is closest in the Euclidean norm sense to \mathbf{y} .

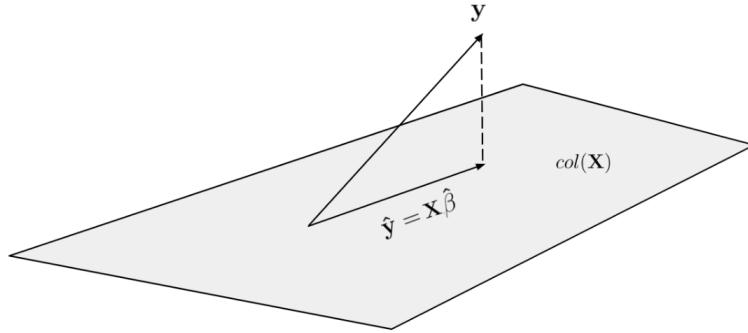


Figure 9.18: Illustration of the geometry underlying least squares regression. It shows the plane $\text{col}(\mathbf{X})$, the vector \mathbf{y} which lies outside the plane and its projection $\hat{\mathbf{y}} = \mathbf{X}\hat{\boldsymbol{\beta}}$ on $\text{col}(\mathbf{X})$. The dashed line corresponds to the vector of residuals $\mathbf{y} - \hat{\mathbf{y}}$.

Furthermore, rewriting the normal equations (9.67) as

$$(\mathbf{y} - \mathbf{X}\hat{\boldsymbol{\beta}})^T \mathbf{X} = \mathbf{0}$$

shows that at the least squares estimator $\hat{\boldsymbol{\beta}}$, the vector of residuals, $\mathbf{y} - \hat{\mathbf{y}} = \mathbf{y} - \mathbf{X}\hat{\boldsymbol{\beta}}$, is orthogonal to the space spanned by the columns of \mathbf{X} as shown in Figure 9.18. What this means is that $\mathbf{X}\hat{\boldsymbol{\beta}}$ is the closest point to \mathbf{y} in the subspace $\text{col}(\mathbf{X})$. Moreover the vector of residuals $\mathbf{y} - \mathbf{X}\boldsymbol{\beta}$ and the predicted values $\mathbf{y} = \mathbf{X}\boldsymbol{\beta}$ are uncorrelated. Hence the independent variables (represented by columns of \mathbf{X}) contain no additional explanatory power about the dependent variable.

Weighted least squares

In some situations, for example when the variance of y_i is a function of the independent variables, it might be preferable to minimize the weighted sum of squared errors

$$g_{\mathbf{w}}(\boldsymbol{\beta}) := \sum_{j=1}^J w_j g_j(\boldsymbol{\beta})^2, \quad (9.71)$$

where $\mathbf{w} = (w_1, \dots, w_J)$ is a vector of positive scalar weights. Defining the *squared weighted Euclidean norm* of $\mathbf{u} \in \mathbb{R}^J$ as

$$\|\mathbf{u}\|_{\mathbf{w}}^2 := \sum_{j=1}^J w_j u_j^2, \quad (9.72)$$

then

$$g_{\mathbf{w}}(\boldsymbol{\beta}) = \|\mathbf{y} - \mathbf{X}\boldsymbol{\beta}\|_{\mathbf{w}}^2.$$

The matrix form of this expression is

$$g_{\mathbf{w}}(\boldsymbol{\beta}) = (\mathbf{y} - \mathbf{X}\boldsymbol{\beta})^\top \mathbf{W} (\mathbf{y} - \mathbf{X}\boldsymbol{\beta}),$$

where $\mathbf{W} = \mathbf{I}\mathbf{w}$ is a diagonal matrix with entries w_1, \dots, w_J .

Corresponding to this optimality criterion, the *weighted least squares (WLS)* parameter estimates are given by

$$\boldsymbol{\beta}_{\text{WLS}} = (\mathbf{X}^\top \mathbf{W} \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{W} \mathbf{y}. \quad (9.73)$$

Note that when $\mathbf{w} = \mathbf{I}$, these reduce to the ordinary least squares parameter estimates.

There are several reasons one might want to use the weighted sum of squares as an optimality criterion:

1. Some regions of the state space might be more important than others, so it is desirable that estimates be more precise in such regions.
2. In statistical models it might be the case that $\text{cov}[\boldsymbol{\epsilon}] = \sigma^2 \mathbf{W}$. For example if for $j = 1, \dots, J$,

$$y_j = \beta_0 + \beta_1 x_j + \epsilon_j$$

and $\text{var}[\epsilon_j] = \sigma^2 x_j$, then it would be appropriate to use WLS with $w_j = 1/x_j$. This simple model applies widely, especially in economic data. It means observations become more variable the greater the value of the independent variable.

3. To adjust for unequal state sampling proportions when the data is either an average or a total of the observations at each state.
4. In the body of this chapter it is noted that the operator corresponding to LSPE is a contraction mapping with respect to the weighted *supremum* norm with weight function equal to the stationary distribution of the underlying Markov chain.

9.8.3 Nonlinear regression

In some cases it may be preferable to represent value functions by nonlinear models, which can be denoted in scalar notation as⁴⁰

$$y_j = f(x_{j,1}, \dots, x_{j,I}; \beta_0, \dots, \beta_I) + \epsilon_j \quad (9.74)$$

for $j = 1, \dots, J$ or in vector notation as

$$\mathbf{y} = \mathbf{f}(\mathbf{X}; \boldsymbol{\beta}) + \boldsymbol{\epsilon}, \quad (9.75)$$

where

$$\mathbf{f}(\mathbf{X}; \boldsymbol{\beta}) := \begin{bmatrix} f(x_{1,1}, \dots, x_{1,I}; \beta_0, \dots, \beta_I) \\ \vdots \\ f(x_{J,1}, \dots, x_{J,I}; \beta_0, \dots, \beta_I) \end{bmatrix} = \begin{bmatrix} f(\mathbf{x}_1; \boldsymbol{\beta}) \\ \vdots \\ f(\mathbf{x}_J; \boldsymbol{\beta}) \end{bmatrix}.$$

As in the case of linear regression, the least squares estimates of the parameters β_0, \dots, β_I minimize the squared error

$$g(\beta_0, \dots, \beta_I) = \sum_{j=1}^J (y_j - f(x_{j,1}, \dots, x_{j,I}; \beta_0, \dots, \beta_I))^2, \quad (9.76)$$

which can be expressed in matrix form as

$$g(\boldsymbol{\beta}) = \sum_{j=1}^J g_j(\boldsymbol{\beta})^2 = \mathbf{g}(\boldsymbol{\beta})^\top \mathbf{g}(\boldsymbol{\beta}), \quad (9.77)$$

where $g_j(\boldsymbol{\beta}) = y_j - f(\mathbf{x}_j; \boldsymbol{\beta})$ and $\mathbf{g}(\boldsymbol{\beta})$ denotes an $J \times 1$ vector with components $g_j(\boldsymbol{\beta})$. Finding $\boldsymbol{\beta}$ such that $g(\boldsymbol{\beta})$ is minimized can be done by applying nonlinear optimization methods to (9.77) such as those described in the next subsection.

9.8.4 Nonlinear optimization

Many approaches can be used to find least squares estimates of nonlinear models. The most widely used are variants of gradient descent and Gauss-Newton iteration, which are described here.

Gradient descent

Let $g(\boldsymbol{\beta})$ denote a real-valued function of the $(I + 1)$ -dimensional parameter vector $\boldsymbol{\beta} = (\beta_0, \dots, \beta_I)$. *Gradient descent* refers to algorithms based on recursions of the form

$$\boldsymbol{\beta}' \leftarrow \boldsymbol{\beta} - \tau \nabla_{\boldsymbol{\beta}} g(\boldsymbol{\beta}). \quad (9.78)$$

⁴⁰Recall that in general the number of independent variables does not need to equal I .

Written in terms of its iterates this recursion is expressed as:

$$\boldsymbol{\beta}^{n+1} = \boldsymbol{\beta}^n - \tau_n \nabla_{\boldsymbol{\beta}} g(\boldsymbol{\beta}^n), \quad (9.79)$$

where the gradient of g evaluated at $\boldsymbol{\beta}$ is defined by

$$\nabla_{\boldsymbol{\beta}} g(\boldsymbol{\beta}) := \begin{bmatrix} \frac{\partial g(\boldsymbol{\beta})}{\partial \beta_0} \\ \vdots \\ \cdot \\ \vdots \\ \frac{\partial g(\boldsymbol{\beta})}{\partial \beta_I} \end{bmatrix} \quad (9.80)$$

and τ_n is the *step size* or *learning rate* at iteration n .

The motivation for this method is that $-\nabla_{\boldsymbol{\beta}} g(\boldsymbol{\beta}^n)$ represents the direction of steepest descent of g at $\boldsymbol{\beta}^n$ so that decreasing $\boldsymbol{\beta}$ in this direction will decrease g the fastest. Unfortunately, this method often converges very slowly and is very sensitive to starting values.

In the context of nonlinear regression, the i -th component of the gradient of g evaluated at $\boldsymbol{\beta}$ is given by

$$\frac{\partial g(\boldsymbol{\beta})}{\partial \beta_i} = -2 \sum_{j=1}^J \left((y_j - f(x_{j,1}, \dots, x_{j,I}; \beta_0, \dots, \beta_I)) \frac{\partial f(x_{j,1}, \dots, x_{j,I}; \beta_0, \dots, \beta_I)}{\partial \beta_i} \right)$$

for $i = 0, \dots, I$. In matrix form this can be written as

$$\nabla_{\boldsymbol{\beta}} g(\boldsymbol{\beta}) = -2 \mathbf{J}(\boldsymbol{\beta})^T \mathbf{g}(\boldsymbol{\beta}), \quad (9.81)$$

where $\mathbf{J}(\boldsymbol{\beta})$ denotes the $J \times (I + 1)$ *Jacobian matrix* of $\mathbf{f}(\mathbf{X}; \boldsymbol{\beta})$ with components

$$\mathbf{J}(\boldsymbol{\beta})_{j,i} := \frac{\partial f(\mathbf{x}_j; \boldsymbol{\beta})}{\partial \beta_i}$$

for $j = 1, \dots, J$ and $i = 0, \dots, I$. Alternatively,

$$\mathbf{J}(\boldsymbol{\beta}) = \begin{bmatrix} \nabla_{\boldsymbol{\beta}} f(\mathbf{x}_1; \boldsymbol{\beta})^T \\ \vdots \\ \cdot \\ \vdots \\ \nabla_{\boldsymbol{\beta}} f(\mathbf{x}_J; \boldsymbol{\beta})^T \end{bmatrix}.$$

The following example illustrates the above elements used to specify gradient descent.

Example 9.15. The expressions below are used to fit the nonlinear^a model

$$y_j = f(x_j; \beta_0, \beta_1, \beta_2) + \epsilon_j = \beta_0 + \beta_1 e^{\beta_2 x_j} + \epsilon_j$$

$j = 1, \dots, J$ to data using gradient descent. In this model, $\boldsymbol{\beta} = (\beta_0, \beta_1, \beta_2)$. Let

$$g(\boldsymbol{\beta}) = \sum_{j=1}^J (y_j - f(x_j; \beta_0, \beta_1, \beta_2))^2 = \sum_{j=1}^J g_j(\boldsymbol{\beta})^2,$$

where $g_j(\boldsymbol{\beta}) = y_j - f(x_j; \beta_0, \beta_1, \beta_2)$.

For this model

$$-\frac{1}{2} \nabla_{\boldsymbol{\beta}} g(\boldsymbol{\beta}) = \begin{bmatrix} \sum_{j=1}^J (y_j - (\beta_0 + \beta_1 e^{\beta_2 x_j})) \\ \sum_{j=1}^J (y_j - (\beta_0 + \beta_1 e^{\beta_2 x_j})) e^{\beta_2 x_j} \\ \sum_{j=1}^J (y_j - (\beta_0 + \beta_1 e^{\beta_2 x_j})) \beta_1 x_j e^{\beta_2 x_j} \end{bmatrix} = \mathbf{J}(\boldsymbol{\beta})^T \mathbf{g}(\boldsymbol{\beta}),$$

where

$$\mathbf{J}(\boldsymbol{\beta}) = \begin{bmatrix} 1 & e^{\beta_2 x_1} & \beta_1 x_1 e^{\beta_2 x_1} \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \\ 1 & e^{\beta_2 x_J} & \beta_1 x_J e^{\beta_2 x_J} \end{bmatrix} \quad \text{and} \quad \mathbf{g}(\boldsymbol{\beta}) = \begin{bmatrix} y_1 - (\beta_0 + \beta_1 e^{\beta_2 x_1}) \\ \vdots \\ y_J - (\beta_0 + \beta_1 e^{\beta_2 x_J}) \end{bmatrix}.$$

^aIf the errors were multiplicative and $\beta_0 = 0$, this model would be linearizable by taking logarithms.

Newton's method

Because gradient descent may converge slowly, Newton's method and its variants provide attractive alternative approaches to solve nonlinear least squares problems. Recall that to find a zero of the scalar function $f(u) = 0$, Newton's method uses the iteration scheme

$$u^{n+1} = u^n - \frac{f(u^n)}{f'(u^n)}$$

For a J -dimensional vector-valued function (\mathbf{u}) with components $f_j(\mathbf{u})$ for $j = 1, \dots, J$ and $\mathbf{u} = (u_1, \dots, u_J)$ the iterates of Newton's method become

$$\mathbf{u}^{n+1} = \mathbf{u}^n - (\mathbf{J}(\mathbf{u}^n))^{-1} \mathbf{f}(\mathbf{u}^n),$$

where as before $\mathbf{J}(\mathbf{u}^n)$, which is assumed to be invertible, denotes the Jacobian matrix of $\mathbf{f}(\mathbf{u})$.

The nonlinear least squares regression problem can be solved by applying Newton's method to the first order optimality condition $\nabla_{\beta} g(\beta) = \mathbf{0}$. This results in the recursion

$$\beta^{n+1} = \beta^n - (\nabla_{\beta}^2 g(\beta^n))^{-1} \nabla_{\beta} g(\beta^n), \quad (9.82)$$

where the *Hessian matrix* $\nabla_{\beta}^2 g(\beta^n)$ has components

$$[\nabla_{\beta}^2 g(\beta^n)]_{i,k} = \frac{\partial^2 g(\beta)}{\partial \beta_i \partial \beta_k}$$

for $i = 0, \dots, I$ and $k = 0, \dots, I$.

Gauss-Newton iteration provides an alternative to Newton's method that avoids computing the second partial derivatives.

Gauss-Newton iteration

Gauss-Newton is based on approximating the real-valued function $g_j(\beta)$ as defined in (9.77) by the first-order Taylor series approximation

$$g_j(\beta) \approx g_j(\beta^n) + (\nabla_{\beta} g_j(\beta^n))^T (\beta - \beta^n). \quad (9.83)$$

Since $\mathbf{g}(\beta) = (g_1(\beta), \dots, g_J(\beta))$ and $g(\beta) = \mathbf{g}(\beta)^T \mathbf{g}(\beta)$

$$g(\beta) \approx (\mathbf{g}(\beta^n) + \mathbf{J}(\beta^n)(\beta - \beta^n))^T (\mathbf{g}(\beta^n) + \mathbf{J}(\beta^n)(\beta - \beta^n)), \quad (9.84)$$

where $\mathbf{J}(\beta^n)$ denotes the $J \times (I + 1)$ Jacobian matrix of $\mathbf{g}(\beta^n)$. Note that in (9.84) β^n is fixed and β is variable so that $\mathbf{g}(\beta^n)$ is a fixed $J \times 1$ vector and $\mathbf{J}(\beta^n)$ is a fixed $J \times (I + 1)$ matrix. Hence, analogous to linear regression, this expression is minimized by choosing

$$\beta - \beta^n = -(\mathbf{J}(\beta^n)^T \mathbf{J}(\beta^n))^{-1} \mathbf{J}(\beta^n)^T \mathbf{g}(\beta^n).$$

Rearranging terms yields the Gauss-Newton recursion:

$$\boxed{\beta^{n+1} = \beta^n - (\mathbf{J}(\beta^n)^T \mathbf{J}(\beta^n))^{-1} \mathbf{J}(\beta^n)^T \mathbf{g}(\beta^n).} \quad (9.85)$$

The following comments are worth noting:

1. The recursion in (9.85) involves matrix inversion. Instead of inverting this matrix, one can solve the system of equations

$$(\mathbf{J}(\beta^n)^T \mathbf{J}(\beta^n)) \Delta^{n+1} = -\mathbf{J}(\beta^n) \mathbf{g}(\beta^n).$$

and set $\beta^{n+1} = \beta^n + \Delta^{n+1}$.

2. Often it might be inconvenient to compute $\mathbf{J}(\beta^n)$ analytically. Instead it can be approximated numerically.

3. Convergence of Gauss-Newton is sensitive to the starting value β^0 . A good choice can enhance convergence.
4. Modifications are available when $\mathbf{J}(\beta^n)^\top \mathbf{J}(\beta^n)$ is close to singular.
5. Gauss-Newton can also be derived as a special case of Newton's method in which the Hessian is approximated by the Jacobian.

Bibliographic remarks

The concept of using approximations in Markov decision processes appears to originate in the clear and concise paper Schweitzer and Seidmann [1985]. This paper, motivated by work on operations research problems, introduced the idea of approximating value functions by low order multi-variable polynomials and showed how to incorporate these approximations in linear programming and policy iteration. Moreover it considers both discounted and average reward formulations and, like this chapter, did not consider the use of simulation in estimation. We do not believe the authors anticipated how significant and widely used this approach would become.

The texts by Bertsekas [2012], Bertsekas and Tsitsiklis [1996] and Powell [2007] provide comprehensive discussions of value function approximation.

The use of features to reduce state spaces appears to originate with Samuel [1959], who is also credited with coining the expression “Machine Learning” to represent the study of using computers to learn tasks without explicitly programming them to do so.

We base our discussion of least squares parameter estimates in policy iteration on Lagoudakis and Parr [2003] and the surveys by Bertsekas [2011] and Busoniu et al. [2012]. The bounds for LSVI and LSPI appear to be new and useful, although other bounds exist in the literature.

de Farias and Roy [2003] provides an in depth study of the approximate LP model. Moreover they analyze an instance of the queuing admission control model used for examples in this chapter and elsewhere.

Nielsen [2015] provides a very readable step-by-step introduction to neural networks. Private communications with several researchers corroborate the difficulties we observed when incorporating neural networks in LSVI and LSPI. In fact, most analyses in the literature are based on linear approximations.

Our discussion and analysis of the advanced scheduling model draws on work of Patrick et al. [2008] and Sauré et al. [2015]. Sauré et al. [2015] also analyzes the queuing service rate control model. The shots gained metric was developed by Broadie [2014]. A similar metric, points gained, was developed for American football by Chan et al. [2021].

Draper and Smith [1998] provide a good introduction to and overview of regression.

Exercises

1. Consider the queuing service rate control model analyzed throughout this chapter. Describe the basis functions you would use if you intended to approximate the value function by:
 - (a) Aggregating states into subgroups.
 - (b) A piecewise linear approximation that interpolates between a subgroup of states.
 - (c) A piecewise quadratic approximation that interpolates between a subgroup of states.
2. Consider a two-state model with a single policy $d(s)$, with $\lambda = 1/2$,
$$\mathbf{r}_d = \begin{bmatrix} 4 \\ 2 \end{bmatrix} \quad \text{and} \quad \mathbf{P}_d = \begin{bmatrix} 0.5 & 0.5 \\ 1 & 0 \end{bmatrix}. \quad (9.86)$$
 - (a) Find $\hat{\beta}_{\text{OLS}}$, $\hat{\beta}_{\text{LSVI}}$ and $\hat{\beta}_{\text{BR}}$ when the basis function is $b_0(s) = 1$.
 - (b) Represent these estimates graphically as in Figure 9.6. Include the coordinates for all quantities.
3. Consider the model in Example 9.3 using an approximation based on $b(s) = s$.
 - (a) Obtain parameter estimates based on LSPE and compare them to those in the text.
 - (b) Repeat your analysis using APLP.
 - (c) Repeat your analysis with $b(s) = s^{0.5}$.
4. Compute linear and quadratic approximations to the value function in Example 9.5 using Bellman residual minimization and compare them to those in the text.
5. Repeat the analysis in Example 9.5 for $S = 0, \dots, 100$ and a decision rule that always uses action a_2 .
 - (a) Compare the quality of quadratic and cubic approximations obtained using OLS, LSPE and Bellman residual minimization to the exact value function and its projection on $\text{col}(\mathbf{B})$.
6. Consider the queuing service rate control model with $S = \{0, \dots, 60\}$.
 - (a) Solve the model exactly using a method of your choice.
 - (b) **Extrapolation.** Find a quadratic value function approximation using LSVI or LSPI on $S' = \{0, \dots, N\}$ for $N = 20, 30, 40$ and use them to obtain greedy policies for all of S . How do these policies compare to the optimal policy?

- (c) Use the value function approximations obtained using the above subsets to obtain bounds on the value functions on all of S using (9.35).
 - (d) **State sampling.** Find a quadratic value function approximation by sampling a fraction of states and then applying LSVI or LSPI on this subset. Compare the resulting greedy policies and values functions to the optimal values and policies as in Figure 9.15. What can you conclude?
 - (e) **State aggregation.** Consider the effect of state aggregation on the accuracy of approximate value functions and corresponding greedy policies where basis functions are indicator functions of disjoint sequences of m consecutive states. Investigate the impact of m on the quality of the approximations.
7. **Oscillation of LSPI.**⁴¹ Let $S = \{s_1, s_2\}$; $A_{s_1} = \{a_{1,1}, a_{1,2}\}$, $A_{s_2} = \{a_{2,1}\}$; $r(s_1, a_{1,1}) = 0.1$, $r(s_1, a_{1,2}) = 0$, $r(s_2, a_{2,1}) = 0$; $p(s_1|s_1, a_{1,1}) = 0.8$, $p(s_2|s_1, a_{1,1}) = 0.2$, $p(s_2|s_1, a_{1,2}) = 1$, $p(s_1|s_2, a_{2,1}) = 1$ and $\lambda = 0.9$.
- (a) Find the optimal policy by enumeration.
 - (b) Consider the basis function represented by $b(s_1) = 1, b(s_2) = 2$ so that the linear approximation to the value function is of the form $\tilde{v} = \begin{bmatrix} \beta \\ 2\beta \end{bmatrix}$. For what values of β is it optimal to use action $a_{1,1}$, respectively $a_{1,2}$, in s_1 .
 - (c) Solve the problem using LSPI starting with $\beta = 1$. What behavior of the iterates do you observe?
8. Consider the replacement model in Section 5.10.2 with $M = 500, p = 0.1, \lambda = 0.95, K = 1200$ and a linear operating cost $2s$ in state s .
- (a) Find an optimal policy by a method of your choosing and describe its structure.
 - (b) Find linear and quadratic value function approximations using LSVI, LSMPI, LSPI and linear programming. Compare the quality of the approximations and the greedy policies identified by each.
9. Solve the instance of the advanced scheduling problem in Section 9.6 using policy iteration, linear programming and evaluate quadratic approximations obtained using LSPI and approximate linear programming.
10. For the advanced appointment scheduling model in Section 9.6, compare the greedy policy based on (9.52) obtained from a random sample of half the states to that in (9.50) based on applying LSVI to the entire state space.
11. (Mini project.) This open-ended problem asks you to carry out your own analysis of the advanced appointment scheduling model analyzed in Section 9.6. To do

⁴¹This example is adopted from Example 1 in Bertsekas [2011].

so entails coding the model, solving it optimally, analyzing it with LSVI using polynomial and neural network approximations, and comparing policies and the quality of approximations. You must specify λ, N, M and K , costs and demand distributions. Some questions to consider:

- (a) How large a problem can you solve exactly?
 - (b) What is the structure of the greedy policies?
 - (c) What is the impact of state sampling?
 - (d) How does the quality of the approximations vary with cost structure?
 - (e) How can you extend your analyses to three or more classes?
12. (Mini project.) Consider an inventory system that manages L products it obtains from a single source. Every time an order is placed, there is a fixed charge of K dollars and a per unit charge of c_l for product l . Assume orders placed arrive prior to the next business day. The cost for storing one unit of product l for one day is h_l .
- Customer orders for products arrive daily and are filled immediately from stock on hand. Assume demand for product l is Poisson distributed with mean μ_l . The selling price for product l is r_l and orders that can not be filled from stock on hand are “lost”, that is the customer seeks the product elsewhere.
- (a) Formulate this as a Markov decision problem where the objective is to maximize discounted expected revenue over an infinite horizon.
 - (b) What quantities would need to be truncated in order to solve this problem. How would you truncate them?
 - (c) Consider a model with $L = 5, K = 200, c_l = 50, r_l = 100, h_l = 2$ and $\mu_l = 5$. Find approximately optimal policies using LSVI, LSPI, LSMPI and linear programming. Choose suitable basis functions and architectures. Derive bounds to assess the quality of your approximations.
 - (d) Is there any obvious structure to the approximate policies? Investigate how they vary when the model parameters change and the products are not identical.
13. Show that the LSPE operator is a contraction in the weighted sup-norm with weighting function given by the stationary distribution of the underlying regular Markov chain.
14. Show that all equivalences in (9.65) hold.
15. Consider the model in Example 9.15.
- (a) Write out the Newton’s method and Gauss-Newton recursions for this model.

- (b) Fit this model to the data in Table 9.3 using gradient descent, Newton's Method and Gauss-Newton. Compare the effort to set the recursion up, the rate of convergence and the sensitivity to starting values.

Variable / Observation	1	2	3	4	5	6	7
y_j	8.37	15.39	4.93	8.92	33.74	7.14	62.95
x_j	2	5	1	3	8	2	10

Table 9.3: Data for Exercise 15. The data generating model was $y_j = 2 + 3e^{0.3x_j} + \epsilon_j$ where $\epsilon_j \sim NID(0, 1)$.

16. Formulate the game of Scrabble as an MDP. What features would you use when approximating a value function?
17. Formulate the game of Tetris as an MDP. What features would you use when approximating a value function?

Chapter 10

Simulation in Tabular Models

This material will be published by Cambridge University Press as “Markov Decision Processes and Reinforcement Learning” by Martin L. Puterman and Timothy C. Y. Chan. This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale, or use in derivative works. ©Martin L. Puterman and Timothy C. Y. Chan, 2025.

You've got to win a little, lose a little, yes, and always have the blues a little.¹

From the song *The Glory of Love*, by Billy Hill, 1899-1940.

This is the first of two chapters on reinforcement learning, focusing on how trial-and-error learning is implemented in simulated and real-world environments. As the song lyric suggests, learning is an ongoing interplay of winning and losing. Successes reinforce behavior while failures diminish it. This dynamic lies at the heart of reinforcement learning and underpins the temporal difference and Q-learning methods introduced in this chapter.

This chapter describes simulation-based methods for evaluating value and state-action value functions, a crucial step in identifying *effective* policies. The focus is on models that can be represented in *tabular* form, that is, when the sets of states and actions are sufficiently small so that value functions, state-action value functions and stationary policies can be represented in look-up tables indexed by either states or state-action pairs. Chapter 11 builds on this foundation by extending methods to complex environments that instead rely on approximations to represent value functions, state-action value functions, and policies.

A brief comment on terminology:

1. The term *methods* replaces algorithms² because the proposed approaches do not

¹Hill [1936].

²Note that in the text, they are still indexed as algorithms.

have stopping criteria that guarantee precise error bounds. The number of iterations or replicates is often chosen arbitrarily.

2. The expressions *good* or *effective* distinguish policies identified by the methods herein from optimal or ϵ -optimal policies generated by the algorithms in Chapters 5, 6 and 7.
3. The term *simulation* refers broadly to any mechanism, be it computer-based or a physical system operating in the real world, that as shown in Figure 10.1, generates successor states and rewards from current states and actions based on an underlying known or unknown MDP.



Figure 10.1: Symbolic representation of the data generation process underlying the methods in this chapter. The simulator, be it computer-based or a real-time process, generates the subsequent state-reward pair (s', r) from a current state-action pair (s, a) .

This chapter will consider three broad classes of methods:

1. Monte Carlo
2. Temporal differencing
3. Q-learning

Each has numerous variants that will be explored below. Policy gradient methods, which are compatible with simulation, will be discussed in the next chapter.

Model classification

The methods herein are frequently classified as either *model-based* or *model-free*. The expression “model-free” is somewhat a misnomer; it refers to settings where system dynamics are governed by a Markov decision process, but where the Markov decision process is not used to estimate value functions, state-action value functions or policies. Instead the system learns by interacting directly with the environment; this feedback guides policy choice. Model-free approaches may be used when the underlying model is very complex and challenging to formulate, when the state space and/or reward function is unknown, or when complex dynamics makes it easier to use a simulator.

More explicitly:

Model-based: Analysis is based on a Markov decision process model of the system. This means that S , $\{A_s | s \in S\}$, $r(s, a, j)$ and $p(j|s, a)$ are known and can be used to generate data by sampling s' from $p(\cdot|s, a)$ and setting $r' = r(s, a, s')$. Moreover, they can be used directly in computation. Examples include inventory management, queuing control, and revenue management models.

Model-free: There is an underlying model but it is not used for analysis because it may be complex or not known. *At a minimum the analyst knows the actions available at each decision epoch.* The set of states may or may not be known. State transitions and rewards are generated by a simulator or a real-world process as depicted in Figure 10.1. Methods must be based on value functions, state-action value functions or policies only. Applications include gaming, autonomous vehicle control, and robotics. This approach is often applied to episodic models in which there is a task to complete. Moreover, there may not be an explicit reward structure; the analyst must design one to achieve the objectives as efficiently as possible.

The distinction between model-based and model-free reinforcement learning is conceptually useful but often blurred in practice. Model-based methods rely on explicit knowledge or estimation of the underlying MDP — namely, the transition probabilities and reward function — and use this model to evaluate or optimize policies. Model-free methods, in contrast, learn value functions or policies directly from experience, without requiring knowledge of the MDP’s structure.

For example, in applications such as advance appointment scheduling (Section 3.7) a Markov decision process model can be explicitly formulated, but it may be preferable to adopt a model-free approach³.

This chapter describes and illustrates policy evaluation and optimization methods, focusing on episodic and infinite horizon discounted models, with a brief discussion of average reward models. Its intent is not to be comprehensive and theoretically rigorous, but to provide a high-level framework for delving into Chapter 11 and the voluminous and rapidly advancing reinforcement learning literature.

The key concept underlying the methods in this chapter and the next is that:

Simulation replaces computation of expectations in Bellman equations.

10.1 Preliminaries

This section discusses data generation and underlying trade-offs when choosing methods.

³The idea here is that instead of deriving a (complicated) analytical model for the MDP, it might be easier to simulate events, apply a policy, and observe the outcome.

10.1.1 Data generation

The two primary sources of data are computer simulation and real-world interaction. They can be used for policy evaluation or optimization. Each creates data sequences by transforming states and actions to successor states and rewards as discussed above:

Simulation: A computer-based simulator generates successor states and rewards using either underlying probabilistic mechanisms or MDP model components.

Real-world interaction: A system, process or experiment generates successor states and rewards.

Repeated application of this mechanism generates *trajectories*. A few examples may help clarify this distinction:

1. In a queuing service rate control model, sampling the number of arrivals and service completions in a period transforms a state and action to a successor state and reward. This is a classic example of an application of simulation. It is preferable to simulating directly from the MDP model, which would require first computing a convolution of the service and arrival distributions. Clearly, this latter approach would not be scalable to larger models.
2. A physical navigational robot interacts with its environment by choosing an action in a given state and (possibly) moving to a new location and generating a reward. This could be costly if some actions could damage the robot. Alternatively a simulation of robot motion could avoid such costs. However if the robot is complex, policies generated through simulation might not perform identically in real life.
3. In Chess, White's choice of a move results in a new board configuration prior to White's next move. Randomness arises through the implementation of Black's strategy. This can be implemented in actual play or more likely through a simulator. Self-play is often used to develop computer-based Chess players.
4. In the pre-board screening application described in the Section 8.6.5, one approach is to develop an MDP model and derive and then simulate system behavior using the derived transition probabilities. Clearly, this is an onerous task. Instead, one could develop a simulation of the system where arrival and service probability distributions combine with operational policies to update the system state and determine one-period costs. Real-world interaction would be impractical since it would impact passenger wait times.
5. In managing patient flow in a health care system, historical data on lengths of stay and transfer decisions could provide appropriate data for analyses.

Our computational experiments suggest that distinguishing computer-based simulations from real-world interaction was important to bring to the forefront for the following reason. In a simulation environment, the analyst may restart the system at any state at any time, while in real-world interaction, states usually follow system dynamics. As a case in point, consider the pre-board screening application referred to above. Clearly the system state cannot be altered in real time. It is determined by passenger arrivals and inspection completions. Computer simulation allows testing policies in a wide range of configurations that might not have been observed or that occur infrequently.

Data generation mechanisms

Starting at $s^1 \in S$, a simulator or process will generate a sequence

$$(s^1, a^1, r^1, s^2, a^2, r^2, \dots, s^N, r^N),$$

where N may be random or fixed. In the model-based environment, $r^n = r(s^n, a^n, s^{n+1})$, so it can be computed from a sequence of state-action pairs. In the model-free environment, the reward will be observed directly.

Data in reinforcement learning applications may be generated in a variety of ways, depending on the setting, objectives, and available resources. The following four categories capture common approaches, each with different implications for learning, evaluation, and exploration. Their significance will become clearer in the examples that follow.

Trajectory-based: Specify a stationary randomized or deterministic policy π .

Starting from an initial state $s^1 \in S$, generate a trajectory $(s^1, a^1, r^1, s^2, a^2, r^2, \dots, s^N, r^N)$ from a simulator or a real-world process. The initial state may be fixed or chosen randomly from a pre-specified distribution.

State-based: Specify a stationary randomized or deterministic decision rule $d(s)$. Sample $s \in S$ from a specified distribution. Apply $d(s)$ to generate⁴ a successor state s' and a reward r . In state-based data generation, s is not the result of a transition from the previous state; it is sampled from a pre-specified probability distribution. This is sometimes referred to as *random restart* data generation.

Action-based: Given $s \in S$, generate $a \in A_s$ by some mechanism that does not correspond⁵ to a pre-specified policy or decision rule. Then generate s' and r .

⁴When d is randomized, this action is generated from $w_d(a|s)$. When d is deterministic, $d(s)$ denotes an action.

⁵Logically, any way of generating actions from states may be viewed as a policy but it need not be explicitly stated as such or used in the system.

State-action pair-based: Sample $s \in S$ and $a \in A_s$ from specified distributions, and then generate r and s' . State-action based generation combines exploration in *both* the state and action spaces.

Comments regarding the use, benefits, and limitations of these data generation approaches follow:

1. Data may be generated either *sequentially* or by *sampling states or state-action pairs after each iteration*. Sequential data may be realized in real-time as the output of a real or simulated process. In contrast, in most applications, simulation must be used to generate random restart data because random restarts would not be able to be physically realized.
2. Trajectory-based and state-based approaches provide the basis for policy evaluation.
3. Trajectory-based methods generate data that behaves like the stationary distribution of a Markov chain generated by a policy.
4. State-based methods, in which states are generated either randomly or deterministically, enable investigation of system behavior in states that are infrequently (or never) visited under trajectory-based data generation.
5. Action-based methods are particularly useful in optimization because they facilitate exploration in the action space (and state space) enabling identification of potentially beneficial actions.
6. State-action pair-based methods combine the features of state-based and action-based methods. They combine exploration in the action space with the ability to generate states that are infrequently observed under policies generated by optimization methods.

Common random numbers

Simulation-based comparisons of policies or methods can be enhanced using *common random numbers*. The idea is to use the same random number stream to evaluate multiple policies or algorithmic parameter choices. The benefit is that by eliminating one source of variability, namely the random number sequence seed, the variability of comparisons is reduced⁶

⁶Suppose X and Y are two random variables. Then

$$\text{Var}(X - Y) = \text{Var}(X) + \text{Var}(Y) - 2\text{Cov}(X, Y).$$

When X and Y are positively correlated, that is when $\text{Cov}(X, Y) > 0$, the variance of the difference of X and Y is reduced. Common random numbers seek to achieve this by removing one source of variability, the random number sequence seed.

In the context of this chapter and the next, common random numbers potentially reduce the variance of the *differences* of estimates of value function and state-action value functions across policies or methods. In this chapter, they will be primarily used for comparing the performance of algorithms under different parameter choices or implementation methods.

Examples of applications of common random numbers include using the same stream of arrivals and service times in queuing control models or the same demand stream in inventory models such as the newsvendor model.

10.1.2 Trade-offs

This section describes some terminology and trade-offs that may be encountered when using the methods described in subsequent sections.

1. **Offline vs. online:** *Online* methods update value functions or state-action value functions after each transition. *Offline (batch)* methods use a complete realization of a trajectory for computation. In practice one may combine these two approaches by first using an offline implementation to obtain good starting values for subsequent application of an online method in real-world interaction.
2. **Exploration vs. exploitation:** In light of one of the most famous quotes commonly mis-attributed⁷ to Albert Einstein (1879-1955):

Insanity is doing the same thing over and over again and expecting different results,

it is advantageous to try new things occasionally. Thus, to find good policies, a method needs to deviate from greedy action selection so as to investigate the impact of other actions on system performance. Greedy action selection is referred to as *exploitation* and non-greedy action selection as *exploration*. Exploration methods include ϵ -greedy and softmax sampling, which are described below. Exploitation may be attractive when the decision maker wishes to accrue rewards during the learning phase or improve estimation of value functions and state-action value functions.

3. **On-policy vs. off-policy:** A method is said to be *on-policy* if it evaluates the data-generating policy, often referred to as the *learning* or *behavioral* policy. A method is said to be *off-policy* if it evaluates a policy that differs from the behavioral policy. This distinction is relevant when designing algorithms and when the reward received during learning is of concern to the investigator. Off-policy methods should be considered when reusing previously collected data or when safety or cost constraints limit exploration under the behavioral policy.

⁷The source of the quote appears to be the book *Sudden Death* by Rita Mae Brown, Bantam, 1983, p. 68, as noted in *The Ultimate Quotable Einstein*, by Alice Calaprice, Princeton University Press, 2011, p. 474.

4. **Variance vs. bias:** In statistical estimation one often seeks a minimum-variance unbiased estimator. When evaluating a method, the analyst may be willing to accept some bias in exchange for reduced variance. The *root mean squared error* for distinguished states or averaged over states captures this trade-off. (See Appendix 10.11.1 for a detailed discussion of this issue.)
5. **Synchronous vs. asynchronous:** A *synchronous* method simultaneously updates value functions in **all** states or state-action value functions in **all** state-action pairs in a similar way to value and policy iteration. An *asynchronous* method updates these quantities one state or state-action pair at a time in a similar way to Gauss-Seidel iteration. With the exception of $\text{TD}(\gamma)$, all the methods in this chapter are asynchronous. The reason for this distinction is that in asynchronous methods, the precision of estimates varies over states or state-action pairs.

Table 10.1 distinguishes the methods discussed in this chapter with respect to their input data, use, and updating approach.

Method	Data	Use	Updating
Monte Carlo	Trajectory-based	Policy Evaluation	Offline
Temporal differencing	Trajectory-based or state-based	Policy Evaluation	Online
Q-learning	Trajectory-based or state-action pair-based	Optimization	Online

Table 10.1: Methods, compatible data generation mechanisms, and purpose.

10.2 Methods overview: The learning newsvendor

This section provides an overview of the methods introduced in this chapter by demonstrating how they can be used to determine the optimal order quantity in the classic newsvendor model, which was introduced in Section 3.1.2. The newsvendor model occupies a central role in the operations research literature due to its analytical tractability and real-world relevance. It applies to a wide-range of products and services with uncertain demand and a limited selling season including bread and dairy products, daily (printed) newspapers, fashion items and time-sensitive services such as hotel rooms and airplane seats.

The reason for focusing on the newsvendor problem is that it provides a transparent yet rich framework, namely a single-state, one-period model, in which to highlight core concepts without the complicating features of multi-period dynamics and multiple states. Thus, it provides an ideal setting to introduce Monte Carlo methods, temporal differencing, and Q-learning.

As noted in Section 3.1.2, the newsvendor's objective is to choose an order quantity $a^* \in A_0$ that maximizes the expected revenue

$$E[r(a, Z)] = \sum_{z=0}^M r(a, z) f(z), \quad (10.1)$$

where A_0 denotes the set of possible order quantities and M denotes a finite upper bound on the demand⁸. Recall that the random variable Z denotes the demand, $f(z) := P[Z = z]$, and $r(a, z)$, the revenue if a units of the item are ordered and the demand is z , was shown to be equal to

$$r(a, z) := G \min(z, a) - L(a - z)^+. \quad (10.2)$$

In this expression G denotes the profit (selling price minus cost) obtained by selling a unit and L the loss (cost minus salvage value) associated with disposing of an unsold item.

Moreover, it was shown that if the demand distribution is known and continuous, the optimal order quantity is available in closed form as

$$a^* = F^{-1} \left(\frac{G}{G + L} \right), \quad (10.3)$$

where $F(z) := P[Z \leq z]$.

10.2.1 Analysis by simulation

This may be regarded as either an episodic model with all episodes of length one, or equivalently, as a one-period model as formulated in Chapter 3. The following rather formal notation will help relate it to later material in this chapter and as well to general Markov decision process notation. Let⁹ $q(a)$, $a \in A_0$, denote the expected reward when choosing order quantity (action) a in state 0, that is,

$$q(a) := E[r(a, Z)].$$

Since in this simple model decision rules correspond to actions, the expected total reward of using decision rule $d'(0) = a'$ is given by $v^{d'}(0) = q(a')$. Note the equivalence between the value function and state-action value function here because in general:

A value function equals a state-action value function evaluated at a specific decision rule.

Offline (batch) and online approaches for obtaining estimates $\hat{q}(a)$ of $q(a)$ are described below.

⁸In contrast to Chapter 3, the demand is truncated to facilitate demand simulation and analyses.

⁹In the standard state-action value function notation, this quantity would be written as $q(0, a)$. The expression $q(a)$ is used for simplicity, since there is only one state.

Monte Carlo estimation

Monte Carlo estimation is an offline (batch) approach. A policy corresponds to a single fixed order quantity \bar{a} . To evaluate it using a Monte Carlo approach, take N samples $\{z^1, \dots, z^N\}$ from $f(\cdot)$, set

$$r^n := r(\bar{a}, z^n) \quad (10.4)$$

and

$$\hat{q}(\bar{a}) = \frac{1}{N} \sum_{n=1}^N r^n. \quad (10.5)$$

To optimize using Monte Carlo, repeat the above simulation for each $a \in A_0$ and set

$$a^* \in \arg \max_{a \in A_0} \hat{q}(a). \quad (10.6)$$

Note that

1. the same demand sequence could be used for evaluating all actions,
2. the ordering of r^n does not affect the estimate, and
3. estimates of the standard deviation and other distributional properties of $q(a)$ can be easily obtained.

Temporal differencing

Temporal differencing provides an online approach for policy value function estimation¹⁰. Instead of waiting until all N values of r^n have been realized, estimates of $q(\bar{a})$ are updated as data arrives. It works as follows. To evaluate the policy \bar{a} , set $q^1(\bar{a})$ to an arbitrary value¹¹ and after realizing z^n , compute r^n using (10.4) and update the estimate of $q(a)$ according to

$$q^{n+1}(\bar{a}) = q^n(\bar{a}) + \tau_n(r^n - q^n(\bar{a})), \quad (10.7)$$

where $0 < \tau_n \leq 1$ for $n = 1, 2, \dots$ is a sequence of positive constants converging to 0. The expression (10.7) can be thought of as follows. The quantity $q^n(\bar{a})$ is the “prediction” of $q(\bar{a})$ before observing r^n . After observing r^n a “new prediction” is obtained by adding a *fraction* of the difference between the observation and the previous prediction to the previous prediction. Referring to τ_n as the *learning rate*, the above recursion can be expressed as

New prediction = old prediction + learning rate \times (observation – old prediction).

¹⁰This is referred to as a *prediction problem* in the reinforcement learning literature.

¹¹0 is a good choice.

Choosing a learning rate less than 1 avoids over-correction and smooths out noise. Requiring that it approaches 0 as n increases stabilizes the estimate. Moreover, choosing the learning rate to satisfy some technical conditions described in this chapter and Appendix 10.11.2 provides theoretical guarantees of convergence.

Some comments about temporal differencing follow:

1. This approach is referred to as TD(0) for reasons to be discussed below.
2. Choosing $\tau_n = 1/n$ is equivalent to online estimation of the sample mean¹².
3. Choosing the learning rate in practice requires experimentation.
4. The order of the r^n does impact the value of the estimate for non-constant learning rate sequences.
5. Distributional properties of $q^n(\bar{a})$ can be obtained by choosing multiple permutations or bootstrap¹³ samples from (r^1, \dots, r^n) .

Q-learning

Instead of computing an estimate of $q(a)$ for each action and then maximizing, one could instead incorporate the maximization directly in the recursion. This is the idea underlying the *Q-learning* recursion:

$$q^{n+1}(a) = q^n(a) + \tau_n \left(\max_{a \in A_0} r(a, z^n) - q^n(a) \right). \quad (10.8)$$

When A_0 consists of a single action, Q-learning is equivalent to temporal differencing.

Because of noise (sampling variability) alternatives to greedy action choice may be preferable. Recursions based on this approach can be expressed as

$$q^{n+1}(a) = q^n(a) + \tau_n(r(a', z^n) - q^n(a)), \quad (10.9)$$

where a' is chosen randomly in some way¹⁴.

One approach is to use a version of ϵ -greedy action selection that samples a' according to:

$$a' = \begin{cases} a^* \in \arg \max_{a \in A_0} r(a, z^n) & \text{with probability } 1 - \epsilon_n \\ a \in A_0 \setminus \{a^*\} & \text{with probability } \epsilon_n / (|A_0| - 1) \end{cases} \quad (10.10)$$

¹²See Example 10.14 in the chapter appendix

¹³Bootstrapping here refers to the statistical procedure that resamples from a data set with replacement to obtain properties of estimators.

¹⁴In a one-period model such as this, this algorithm is identical to an algorithm referred to as SARSA below.

for some sequence of $\epsilon_n, n = 1, 2, \dots$. Note that decreasing ϵ_n leads to less exploration later on in the iterative process. Moreover, setting $\epsilon_n = 1$ for all n corresponds to randomly sampling actions and $\epsilon_n = 0$ for all n to using greedy action selection.

An alternative is to use *softmax* sampling (action selection) in which action a' is chosen with probability

$$p(a') := \frac{e^{\eta_n q(a')}}{\sum_{a \in A_0} e^{\eta_n q(a)}}, \quad (10.11)$$

where the sequence $\eta_n, n = 1, 2, \dots$, affects the relative weighting of actions. Large values of η_n put more weight on actions with the largest values of $q(a')$, while small values of η_n make the weight on actions more uniform. Note that when all values of $q(a)$ are equal, softmax sampling corresponds to random sampling.

The ϵ -greedy method selects actions other than the greedy action with equal probability, while the softmax approach samples among actions with greater rewards. When there are a few actions in each state, either method would be appropriate. However, when there are many actions, the softmax sampling may be more appropriate because it concentrates exploration on fewer actions, namely those with greater q -values. This distinction is significant in the calculations below.

Q-learning: An “Earning while learning” perspective

Instead of simply describing a way to find good actions, recursions (10.8) and (10.9) may be viewed from the perspective of an agent that learns by trial-and-error and accumulates reward beginning as follows.

1. Choose $q^1(a)$ and set $R^0 = 0$.
2. Choose a^1 greedily, ϵ -greedily or by softmax sampling.
3. Implement a^1 and observe r^1 .
4. Set $R^1 = R^0 + r^1$.
5. Update $q^2(a^1)$ using (10.8) or (10.9).

The fourth step highlights the subtle distinction with the Q-learning approach described previously. Here, the perspective is that rewards earned are tracked while learning takes place.

By repeating this procedure¹⁵ for N steps, the newsvendor obtains total reward R^N , and uses $q^N(a)$ to choose order quantities. Variants of this approach¹⁶ can be compared on the basis of a running average of the cumulative reward and on how accurately they determine the known optimal order quantity.

¹⁵We choose not to state this formally here.

¹⁶Variants may correspond to different choices of tuning parameters and update methods.

10.2.2 A numerical study

This section illustrates and discusses computation using the above approaches.

Model

Demand is generated from either a discrete (rounded) and truncated normal distribution with mean 50 and standard deviation 15 or a gamma distribution¹⁷ with shape parameter 20 and scale parameter 4. The item cost is 20, the salvage value is 5, and price varies among 21, 30, and 100. The corresponding G values are 1, 10, and 80, and $L = 15$. The corresponding ratios of $G/(G+L)$ are 0.0625, 0.400, and 0.842, illustrating several distinct quantiles of the demand distribution. Set $A_0 = \{0, 1, \dots, 120\}$.

Monte Carlo estimation

Monte Carlo methods are compared using sample sizes of $N = 100$ and $N = 5,000$. A replicate consists of estimating $q(a)$ for each $a \in A_0$ for each price. The expected reward is estimated using (10.5) and the optimal order quantity is chosen according to (10.6). Using a common random number seed, the same realized demand was used in each replicate. There was no need to resample; estimates could be based on stored demand samples.

Table 10.2 gives the optimal order quantity (for a typical replicate) using the closed form representation (10.3) and that based on the arg max of the Monte Carlo estimates of $q(a)$. Of course, choosing $N = 5,000$ produces more accurate estimates of the optimal order quantity than using $N = 100$ but the difference was surprisingly small. An exception was when demand was normal and price was 21, when the optimal order quantity corresponded to $G/(G + L) = 0.0625$. This is because larger sample sizes are needed to estimate values in the tail of a distribution.

Distribution	Price	Optimal	Monte Carlo ($N = 100$)	Monte Carlo ($N = 5,000$)
Normal	21	27	36	26
	30	46	42	44
	100	65	69	64
Gamma	21	55	57	55
	30	74	81	78
	100	98	109	104

Table 10.2: Optimal order quantity and that based on Monte Carlo estimates of $q(a)$ in a typical replicate of the newsvendor model using $N = 100$ and $N = 5,000$.

Figure 10.2 shows $\hat{q}(a)$ based on a single replicate with gamma distributed demand, price equal to 30 and two values of N . Observe that the estimates with $N = 100$ are

¹⁷For this choice of parameters, the mean of the gamma distribution is 80 and its standard deviation is 17.89.

considerably noisier than those with $N = 5,000$. However, surprisingly, the estimated optimal order quantities vary little. Note that for both values of N the curves are quite similar for small values of a . Moreover, Figure 10.2b shows that $\hat{q}(a)$ is quite flat near the true optimum, suggesting that there may be considerable variability in choosing the action using Monte Carlo estimates.

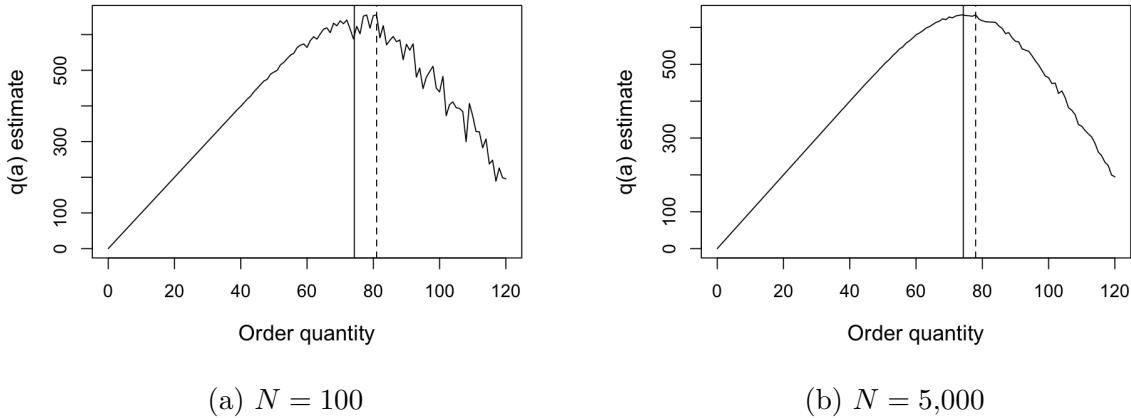


Figure 10.2: Estimates of $q(a)$ based on a single replicate with gamma distributed demand and price equal to 30 for two sample sizes. The solid vertical line represents the theoretical optimal order quantity and the dashed line that obtained using (10.6).

Temporal differencing

This section illustrates the use of temporal differencing by applying it to estimate $q(80)$ for $G = 80$, $L = 15$ and gamma distributed demand. It compares learning rates $\tau_n = 0.05, n^{-1}, 0.5n^{-0.75}$, and $10/(40 + n)$. Figure 10.3 shows the evolution of the iterates of (10.7) over a typical replicate with $N = 1,000$ iterations starting from 0 and the same demand sequence for each learning rate. Observe that the sample mean, which corresponds to $\tau_n = n^{-1}$, best approximates the true value, with the estimate using $\tau_n = 0.5n^{-0.75}$ quite close. Note also that the estimate corresponding to $\tau_n = 0.05$ oscillates. More formally, estimates can be compared using the RMSE as described in Appendix 10.11.1. The respective RMSE values corresponding to the above learning rates are 467.21, 163.87, 214.29 and 419.08, consistent with Figure 10.3. Such calculations over multiple configurations and with many replicates provide the basis for parameter studies in this chapter and the next.

Q-learning

Applying Q-learning using (10.9) was more challenging than the methods above. The ϵ -greedy action selection method was unreliable. Even when ϵ_n was slightly less than one,

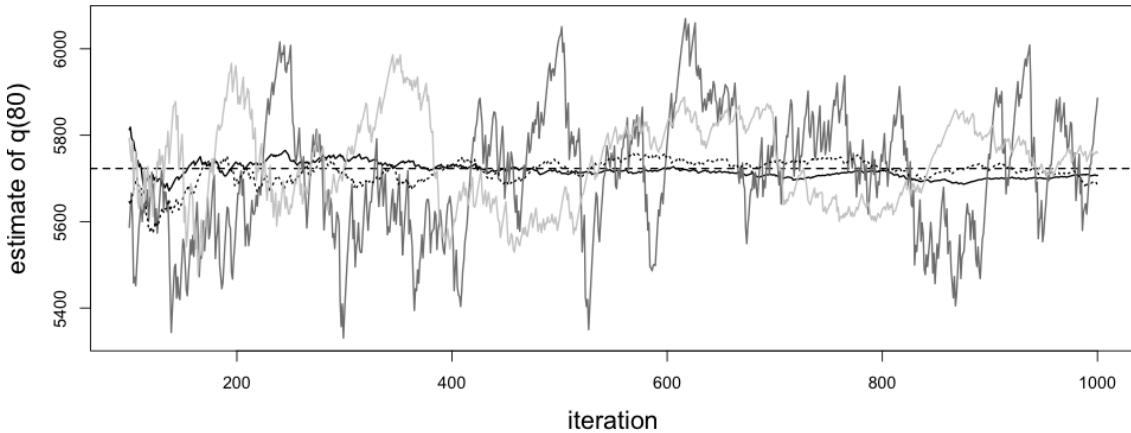


Figure 10.3: Comparison, based on a single replicate, for estimates of $q(80)$ for the four choices of τ_n described in the text. The horizontal dashed line represents the exact value $q(80) = 5,723$, and the other four lines represent different choices of τ_n . The dark grey line corresponds to $\tau_n = 0.05$, the solid line to $\tau_n = n^{-1}$, the dotted line to $\tau_n = 0.5/(n^{0.75})$ and the light grey line to $\tau_n = 10/(n + 40)$.

a single high-value action was sampled too frequently; all other actions were sampled equally and infrequently. Thus the recursion “got stuck” at a potentially non-optimal action.

Softmax action selection was better suited to this problem since there are 121 actions and $q(a)$ was quite flat near the optimum and dropped off quickly as shown in Figure 10.2b. Consequently, softmax sampled high-value actions more frequently, thus providing more accurate estimates of $q(a)$ close to the optimum.

After preliminary experimentation, three learning rates, $\tau_n = 5/(20 + n)$, $\tau_n = 0.5n^{-0.5}$ and $\tau_n = 0.05/(1 + (10^{-5})n^2)$, were chosen for further analyses. The third choice for τ_n is justified later in this chapter. Note that estimates based on $\tau_n = n^{-1}$ and $\tau_n = 0.5n^{-0.75}$ converged too quickly, producing very inaccurate estimates of $q(a)$ and the optimal order quantity.

This study used the scenarios in the model description above and generated 40 replicates for each using common random number seeds for each replicate. For each scenario, the recursions began with $q(a) = 0$, set $N = 50,000$ and the softmax parameter $\eta_n = 1/6,000$ for all n . The number of times an action was previously chosen (as opposed to the iteration number) was used as the index of the learning rate.

Table 10.3 summarizes results in terms of the mean and standard deviation of the optimal order quantity over 40 replicates for each configuration of distribution, price and learning rate. The learning rate $\tau_n = 0.5n^{-0.5}$ best estimated the optimal order quantity on average. However, the estimates were more variable when the price equaled 100. Note that when $N = 10,000$, $\tau_n = 0.5n^{-0.5}$ produced the most accurate estimates

but their variability was greater.

Distribution	Price	Optimal	$\tau_n = 5/(20 + n)$	$\tau_n = 0.05/(1 + (10^{-5})n^2)$	$\tau_n = 0.5n^{-0.5}$
Normal	21	27	29.79 (2.89)	30.72 (3.97)	28.59 (2.73)
	30	46	48.64 (3.09)	48.41 (4.00)	46.62 (2.92)
	100	65	67.69 (5.91)	68.74 (6.70)	67.18 (8.01)
Gamma	21	55	57.62 (3.05)	57.15 (3.66)	55.18 (2.60)
	30	74	76.41 (3.91)	76.64 (4.22)	74.64 (4.54)
	120	98	99.87 (6.32)	104.13 (6.71)	99.15 (8.65)

Table 10.3: Optimal order quantity in the newsvendor model and mean (standard deviation) of estimates based on 40 replicates of Q-learning with three different learning rates.

Q-learning: An “Earning while learning” perspective

The above analysis is from the perspective of solving an optimization problem; that is, Q-learning is an online method to optimize the order quantity. Alternatively, from a simultaneous earning and learning perspective, the newsvendor might seek to maximize revenue while learning what quantity to order.

To illustrate this approach, consider an instance with normal demand, price equal 30, $N = 100,000$ iterates, the three learning rates used above, and softmax action selection with $\eta_n = 600n^{0.2}$. The index for the learning rate was the number of times an action was previously used and the index for softmax sampling was the iteration number. The effect of this specification for η_n is to allow random action choice at the beginning and then make it more likely that sampling is done from actions with high values of the estimate of $q(a)$, which is shown in Figure 10.4. Observe that in this replicate, the estimate of $q(a)$ is smoothest around the optimal order quantity where most of the actions are chosen.

Figure 10.5 shows the running average revenue in a typical replicate¹⁸. Observe that all three learning rates result in learning as indicated by increasing average revenue per iterate. Using the learning rate $\tau_n = 5/(20 + n)$ resulted in the greatest revenue early on. Eventually, revenue streams from all three learning rates become approximately the same but that using $\tau_n = 5/(20+n)$ continued to have the greatest average revenue. Note that over 100,000 iterates even a small increase in average revenue contributes to a large increase in total revenue.

10.2.3 Conclusions

The purpose of this section was to provide an introduction to the simulation methods in this chapter. By choosing a practical and easy to understand **one-state** example, several implementation issues that arise in multi-state models were swept under the rug.

¹⁸In 40 random sequences, the average revenues always exhibited this pattern.

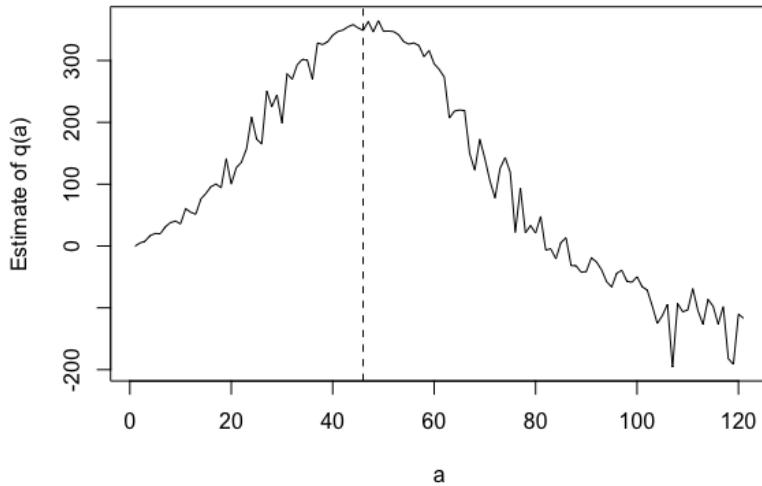


Figure 10.4: Estimate of $q(a)$ based on 100,000 iterates of Q-learning. The vertical dashed line corresponds to a^* found using (10.3).

However, this example was sufficiently rich to provide insight into issues arising when using these techniques. The remainder of this chapter generalizes these approaches to multi-state models.

Some noteworthy observations include:

1. Monte Carlo methods are easy to understand, however learning does not occur until all data has been collected. As the above calculations show, Monte Carlo methods involve costly experimentation for actions that may be far from optimal. However, because $q(a)$ is relatively flat near the maximum, large sample sizes near this value are useful. Note that Monte Carlo could be integrated with a search procedure to reduce experimentation.
2. Temporal differencing provides an online approach for policy value function estimation. This means that estimates are available after each observation. However, the quality of the approximation depends significantly on the learning rate. Note that $\tau_n = n^{-1}$ corresponds to an online version of Monte Carlo.
3. Q-learning worked best with softmax sampling and on-policy updates using (10.9). Again, outcomes were sensitive to the learning rates. After preliminary experimentation three promising learning rates were used in a more detailed study. Optimal order quantity estimates were quite similar with the exponential learning rate working slightly better than the others in determining an order quantity. From the perspective of learning, the ratio learning rate generated the greatest revenue. Differences in revenue over learning rates was greatest initially.

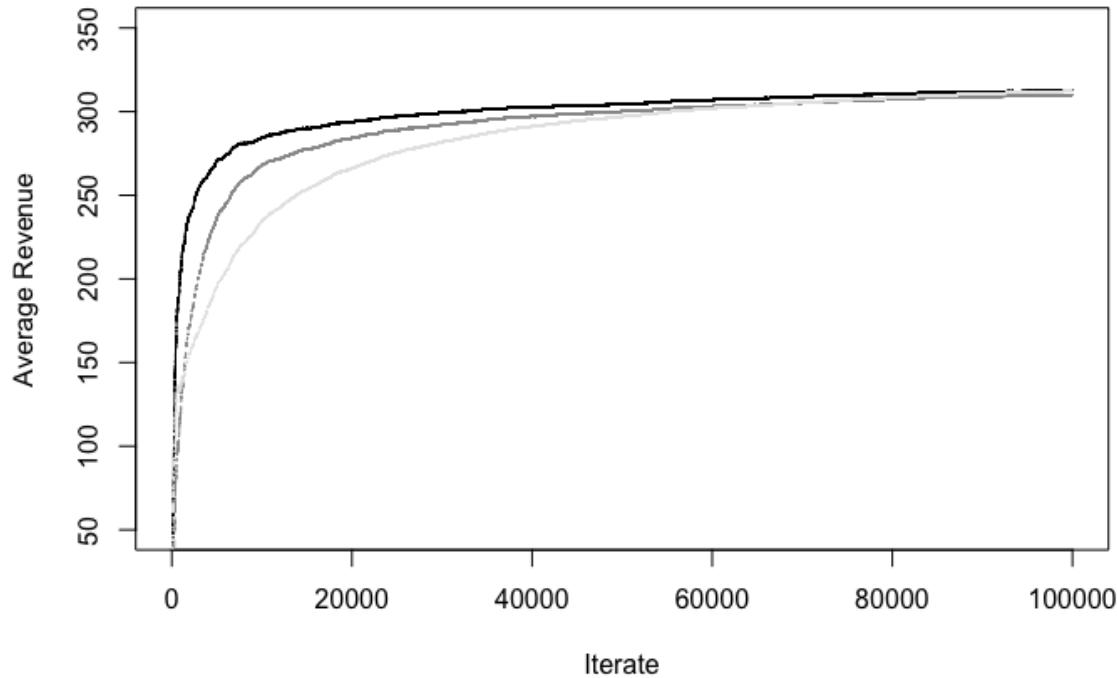


Figure 10.5: Cumulative average revenue vs. iterate using Q-learning. Data is from a single replicate of newsvendor model using softmax sampling and three learning rates. The black line corresponds to $\tau_n = 5/(20 + n)$, the dark grey line (between the other two) to $\tau_n = 0.05/(1 + (10^{-5})n^2)$ and the light grey to $\tau_n = 0.5n^{-0.5}$.

10.3 Policy evaluation: Episodic models

In an episodic model, the system accrues rewards until it reaches a reward-free absorbing state at some (possibly) random stopping time in the future. Usually this will be when a task is completed or a game ends. This section focuses on models in which stopping times are determined by the evolution of the system. For example, in a robot navigation model, an episode terminates when the robot achieves its goal¹⁹.

As noted in Section 2.3.2, a discounted model may be viewed as an episodic model in which the episode terminates at a geometrically distributed stopping time independent of the evolution of the system. Policy evaluation for discounted models will be discussed in Section 10.4 below.

Finite horizon models may also be regarded to be episodic models in which the stopping time is fixed and after which the system transitions to a zero-reward absorbing

¹⁹The episode could end when the robot fails, but computation in various applications suggests that it is more expedient to terminate an episode only when the goal is achieved.

state. This chapter will not discuss finite horizon models²⁰ but many of the techniques here can be generalized to apply to them.

In episodic models, the expected total reward of stationary policy $\pi = d^\infty$ is given by

$$v^\pi(s) = E^\pi \left[\sum_{n=1}^{N-1} r(X_n, Y_n, X_{n+1}) + g(X_N) \middle| X_1 = s \right], \quad (10.12)$$

where the random variable N represents the random time at which the process terminates, X_n denotes the state at decision epoch n , Y_n denotes the action chosen at decision epoch n , X_{n+1} denotes the state at decision epoch $n+1$ that results from choosing action Y_n in state X_n , and $g(X_N)$ denotes the reward when terminating in state X_N ²¹. The random variable $N = \min\{n \geq 1 \mid X_n \in \Delta\}$, sometimes referred to as the *effective horizon*, denotes the time the system enters a set of zero-reward absorbing states, Δ .

Chapter 6 established the optimality of deterministic policies²² for such models. However, allowing for randomized policies broadens the applicability of the methods herein. Chapter 6 shows that in several classes of models²³, $v^\pi(s)$ is well-defined for most policies, including any policy that could be optimal.

10.3.1 Monte Carlo methods

The most obvious approach for estimating $v^\pi(\bar{s})$, for a fixed state \bar{s} , is to sum up the rewards for each episode and average over episodes. This leads to the following *Monte Carlo* algorithms, which differ according to which quantities are retained from each episode. Each assumes that an episode terminates when the system reaches a state in the set Δ .

Starting-state Monte Carlo

This simple Monte Carlo algorithm is stated for illustrative purposes. It produces an estimate for a single state \bar{s} only. First-visit and every-visit variants are preferable and slightly more involved to state but easy to implement.

Algorithm 10.1. Starting-state Monte Carlo policy evaluation for an episodic model

²⁰Chapter 4 showed that in finite horizon models, optimal policies are usually non-stationary.

²¹One could alternatively set $g(\cdot) = 0$ and include the termination reward in the definition of $r(X_{N-1}, Y_{N-1}, X_N)$. This is how the model was written in Chapter 6 but the representation with a specified terminal reward is more natural here.

²²When T is stochastic, optimal policies will also be stationary; when T is fixed and constant, non-stationary policies are often optimal.

²³Transient models, stochastic shortest path models, positive models and negative models.

1. Initialize:

- Specify $d \in D^{\text{MR}}$ and the number of episodes K .
- Specify $\bar{s} \in S \setminus \Delta$ and create an empty list VALUES.
- $k \leftarrow 1$.

2. Iterate: While $k \leq K$:

- $s \leftarrow \bar{s}$ and $v \leftarrow 0$.
- Generate episode:** While $s \notin \Delta$:
 - Sample a from $w_d(\cdot|s)$.
 - Simulate (s', r) or sample s' from $p(\cdot|s, a)$ and set $r \leftarrow r(s, a, s')$.
 - Update the value:

$$v \leftarrow r + v. \quad (10.13)$$
 - $s \leftarrow s'$.
- Terminate episode ($s \in \Delta$):**
 - Append $v + g(s)$ to VALUES.
 - $k \leftarrow k + 1$.

3. Terminate: Return

$$\hat{v}^{d^\infty}(\bar{s}) = \text{mean}(\text{VALUES}).$$

Some comments about the algorithm follow:

- The above algorithm estimates $\hat{v}^{d^\infty}(\bar{s})$ by setting it equal to the average of observed total rewards for all episodes that start in state \bar{s} . In most cases, one would seek estimates for all $s \in S \setminus \Delta$.
- If $d \in D^{\text{MD}}$, step 2(b)i is replaced by “Set $a = d(s)$ ”.
- The algorithm statement includes a model-free variant in which state transitions and rewards are simulated or observed, and a model-based variant in which states are sampled and rewards computed using the triple (s, a, s') .
- Since the total rewards from different episodes are independent and identically distributed, $\hat{v}^{d^\infty}(\bar{s})$ converges almost surely to its mean, $v^{d^\infty}(\bar{s})$, by the *strong law of large numbers*, and is asymptotically normal by the *central limit theorem*.
- By storing estimates of $v(\bar{s})$ in the list VALUES instead of updating the mean sequentially, one can investigate distributional properties of the estimates such as the standard deviation and percentiles. Moreover, one can compute standard

errors (i.e., standard deviation divided by the square root of the number of observations) and confidence intervals for the estimates, and use the width to guide the choice of K .

6. The algorithm as stated above *updates values only for the starting state \bar{s}* of each episode. Obviously data would be used more efficiently if it also estimated $v(s)$ for *all* states visited during an episode. If a state repeats, the estimate for that state may be based on the total return accumulated from the first visit to that state (or from every visit to that state). If such an approach is used, $v(s)$ should be updated for *all* states previously visited in step 2(b). The first-visit algorithm is stated below.
7. Algorithm 10.1 is an offline algorithm since it requires all data to estimate $v^{d^\infty}(s)$ in step 3.

First-visit Monte Carlo

The following algorithm uses data more efficiently by storing cumulative rewards for each distinct state visited during an episode.

Algorithm 10.2. First-visit Monte Carlo policy evaluation for an episodic model

1. **Initialize:**
 - (a) Specify $d \in D^{\text{MR}}$ and the number of episodes K .
 - (b) For all $s \in S \setminus \Delta$ create an empty list $\text{VALUES}(s)$.
 - (c) $k \leftarrow 1$.
2. **Iterate:** While $k \leq K$:
 - (a) Create empty list VISITED .
 - (b) $v(s) \leftarrow 0$ for all $s \in S \setminus \Delta$.
 - (c) Specify $s \in S \setminus \Delta$.
 - (d) **Generate episode:** While $s \notin \Delta$:
 - i. If $s \notin \text{VISITED}$, append s to VISITED .
 - ii. Sample a from $w_d(\cdot|s)$.
 - iii. Simulate (s', r) or sample s' from $p(\cdot|s, a)$ and set $r \leftarrow r(s, a, s')$.
 - iv. For all $s'' \in \text{VISITED}$

$$v(s'') \leftarrow r + v(s''). \quad (10.14)$$
 - v. $s \leftarrow s'$.

(e) **Terminate episode ($s \in \Delta$):**

- For all $s'' \in \text{VISITED}$, append $v(s'') + g(s)$ to $\text{VALUES}(s'')$.
- $k \leftarrow k + 1$.

3. **Terminate:** For all $s \in S \setminus \Delta$, return

$$\hat{v}^{d^\infty}(s) = \text{mean}(\text{VALUES}(s)).$$

Some comments about this algorithm follow:

1. The list VISITED keeps track of the *distinct* states visited during an episode and 2(d)iv updates the value for each.
2. An alternative to this algorithm would be an every-visit version, which in each replicate starts a new total every time a state is visited. That is, if the sequence of states was $(s_1, s_2, s_5, s_2, \dots)$, such an algorithm would generate two estimates for s_2 , one beginning from the first visit and one beginning from the second visit. It is left to the reader to formally state and apply such an algorithm.

An example

This section illustrates the starting-state and first-visit Monte Carlo algorithms by evaluating a policy for the Gridworld model of Section 3.2. The calculations assume an episode ends when *either* the robot reaches the office (cell 1) or falls down the stairs (cell 7). Thus $S = \{1, \dots, 15\}$ and $\Delta = \{1, 7\}$. The cell configuration diagram is repeated in Figure 10.6 below.

Consider the Gridworld model represented in Figure 10.6 with $p = 0.8$, a cost of 200 for falling down the stairs, a reward of 50 for successfully delivering the coffee and a per unit move cost of 1. This example analyzes the stationary policy d^∞ in which the robot seeks to move to the lowest numbered neighboring cell except when in state 10 when it instead tries to move toward cell 11. It moves to the intended cell with probability p and to each other adjacent cell with equal probability. The formulation assumes that the robot cannot move diagonally. For example, from cell 14, the robot moves to cell 11 with probability p and to cells 13 or 15 with probability $(1 - p)/2$.

The example compares Algorithm 10.1 and Algorithm 10.2, using $K = 2,000$ replicates for each starting state. Figures 10.7a and 10.7b show the estimates of the values starting in each state using the two algorithms.

Observe that estimates based on first-visit Monte Carlo had narrower simultaneous 95% confidence intervals²⁴ than those based on starting-state Monte Carlo. For example, the standard errors when starting in cell 13 were 2.34 for starting-state and 2.07

²⁴Such confidence intervals are constructed so that in 95% of samples all intervals contain the true mean. A Bonferroni correction (0.025 divided by the number of estimates) is used to obtain the appropriate t -value.

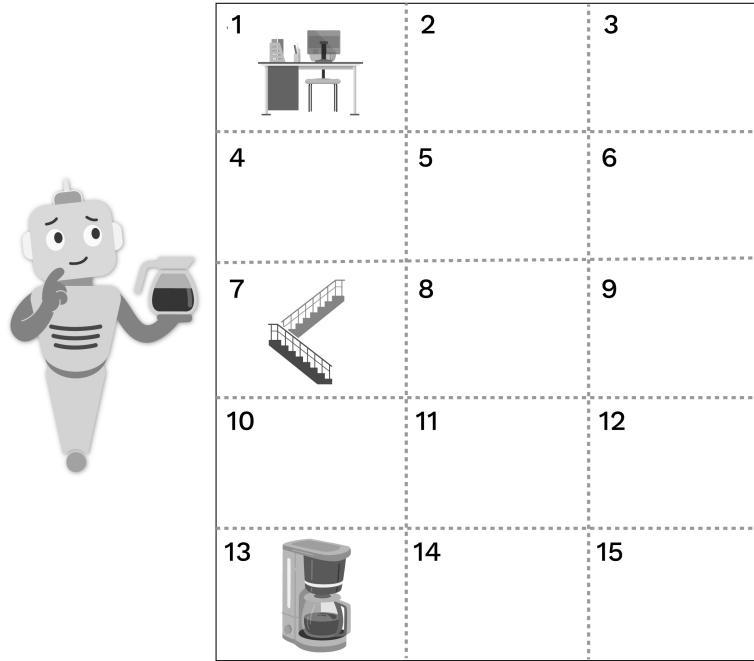


Figure 10.6: Schematic layout for Gridworld navigation example.

for first-visit. When starting in cell 2, the respective standard errors were 0.30 and 0.08. Thus, as expected, the first-visit Monte Carlo algorithm provides more accurate estimates with the same number of replicates especially for the most frequently visited states.

Figure 10.8 provides a histogram of values for cell 13 when using first-visit Monte Carlo. Observe that the distribution is extremely bi-modal corresponding to the possibilities of falling down the stairs or successfully delivering the coffee. Consequently, the mean does not represent this distribution well and is perhaps a poor choice of a value of a policy. Note that the 5th percentile is -204 , the 25th percentile is 38 , the median is 42 and the 75th percentile and 95th percentiles equal 44 . Thus, one might wish to consider the use of different summaries of the distribution (such as the median or 75th percentile) when analyzing such models.

10.3.2 Temporal differencing

Temporal differencing represents one of the most significant innovations in reinforcement learning. It refers to methods that update policy value function estimates based

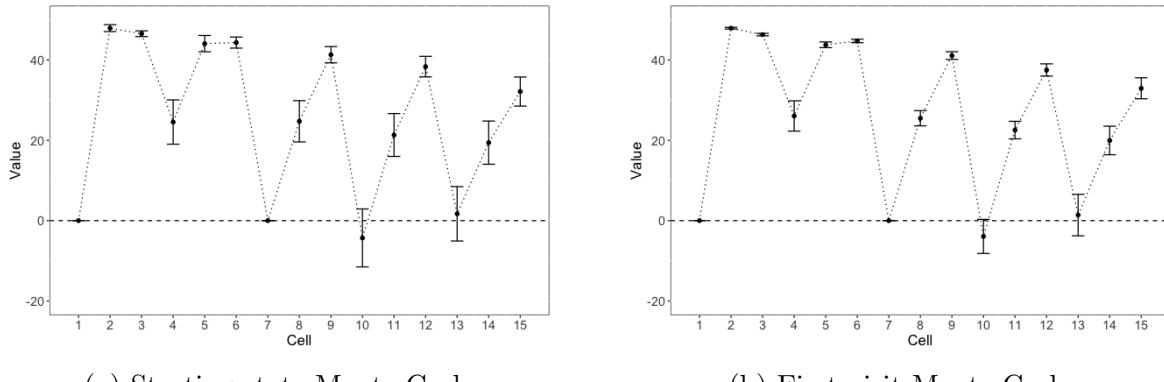


Figure 10.7: Value function estimates for the specified policy based on 2,000 replicates for each starting state and $p = 0.8$. Error bars in Figures 10.7a and 10.7b represent simultaneous (Bonferroni) 95% confidence intervals.

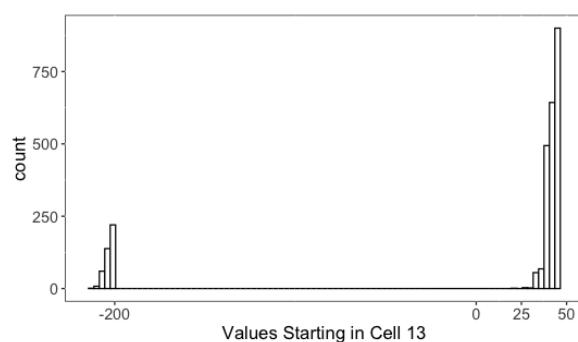


Figure 10.8: Histogram of estimates of $v(13)$ using Algorithm 10.2. Note the mean of these estimates is 0.64.

on a fraction of the difference between a current estimate and a one-step prediction equal to an observed reward and the value in a successor state. The difference is called the *temporal difference*. The term, temporal differencing, has its roots in behavioral neuroscience referring to situations in which actions are reinforced when predictions exceed expectations and degraded when they do not.

To motivate this approach, consider the operator

$$L_d \mathbf{v} := \mathbf{r}_d + \lambda \mathbf{P}_d \mathbf{v}$$

on the set of real-valued vectors on S when using a fixed decision rule $d \in D^{\text{MR}}$. When $0 \leq \lambda < 1$, it plays a fundamental role in discounted models (Chapter 5), and when $\lambda = 1$, it is fundamental in transient models and applies to proper policies in stochastic shortest path models (Chapter 6). In these models, value iteration, expressed as

$$\mathbf{v}' = L_d \mathbf{v} \tag{10.15}$$

converges to the unique fixed point of L_d , which was shown to equal the value function of d^∞ . Rewriting (10.15) as

$$\mathbf{v}' = \mathbf{v} + (L_d \mathbf{v} - \mathbf{v}) \quad (10.16)$$

provides a conceptually different approach for thinking of value iteration, namely:

$\text{new estimate} = \text{old estimate} + \text{correction}.$

A related numerical iterative scheme, referred to as *successive over-relaxation*, modifies value iteration by introducing a scaling factor τ satisfying $0 \leq \tau \leq 1$ that down weights the correction to obtain:

$$\mathbf{v}' = \mathbf{v} + \tau(L_d \mathbf{v} - \mathbf{v}). \quad (10.17)$$

It is not hard to show that the expression on the right hand side of (10.17) is a contraction mapping and that it also converges to the unique fixed point of L_d .

Suppose instead of computing $L_d \mathbf{v}$ directly, one instead observes a random sample \mathbf{u} from a distribution with expectation²⁵ $L_d \mathbf{v}$, then the obvious generalization of (10.17) becomes

$$\mathbf{v}' = \mathbf{v} + \tau \boldsymbol{\delta} \quad (10.18)$$

where

$$\boldsymbol{\delta} = \mathbf{u} - \mathbf{v}.$$

Temporal differencing applies such an approach on a state-by-state basis by choosing $s \in S$, using the decision rule $d(s)$ to obtain action $a \in A_s$, observing r' and s' , and setting:

$$\delta = r' + \lambda v(s') - v(s)$$

and

$$v'(s) = v(s) + \tau \delta.$$

In this expression, δ is the temporal difference. This approach uses a concept that is often referred to as *bootstrapping*, namely it uses current value estimates to update other value estimates.

Stochastic approximation using whole trajectories

As a different motivation for temporal differencing, consider updates obtained using stochastic approximation based on *complete* episodes. Appendix 10.11.2 of this chapter introduces the following recursion for estimating the mean μ of a random variable:

²⁵This means that $E[\mathbf{u}] = L_d \mathbf{v}$, for example when $\mathbf{u} = L_d \mathbf{v} + \boldsymbol{\epsilon}$ with $E[\boldsymbol{\epsilon}] = \mathbf{0}$.

$$\mu^{n+1} = \mu^n + \tau_n(y^n - \mu^n). \quad (10.19)$$

The appendix shows that this recursion is the result of applying stochastic gradient descent to obtain least squares estimates of a mean. The parameters $\tau_n, n = 1, 2, \dots$ are referred to as *learning rates* or *step-sizes*.

Convergence of μ^n to μ occurs with probability 1 if the following conditions²⁶ are satisfied.

Definition 10.1. The sequence $\tau_n, n = 1, 2, \dots$, satisfies the *Robbins-Monro step-size conditions* if

$$\tau_n \geq 0, \quad \sum_{n=1}^{\infty} \tau_n = \infty \quad \text{and} \quad \sum_{n=1}^{\infty} \tau_n^2 < \infty. \quad (10.20)$$

Let (h^1, h^2, \dots) denote independent samples from a distribution $H(\cdot)$ with expectation equal to the right hand side of (10.12). Then, applying (10.19) provides recursive estimates of $v^{d^\infty}(s)$ given by

$$v^{n+1}(s) = v^n(s) + \tau_n(h^n - v^n(s)). \quad (10.21)$$

One way to produce these samples is by generating complete episodes and setting

$$h^n = \sum_{l=1}^{N^n} r_l^n$$

where r_l^n is the l -th reward in the n -th episode and N^n is the length of the n -th episode. The astute reader will recognize that this is equivalent to generating a sequence of starting-state Monte Carlo estimates and updating the value function estimate using (10.19) after each episode completes²⁷.

The downsides of this approach are that:

1. It delays updating the estimate until the completion of an episode.
2. It wastes considerable within-episode information.
3. A separate estimate must be obtained for each $s \in S$.

²⁶These are frequently referred to simply as the *Robbins-Monro conditions* or the *usual conditions*.

²⁷Sutton and Barto [2018] refers to such an approach as Monte Carlo- α and h^n as the *target* of the update.

An online alternative

To address these concerns, instead of basing updates on an entire reward sequence, temporal differencing bases updates on the (one-step) evaluation equation

$$v(s) = E^d \left[r(X, Y, X') + v(X') \mid X = s \right] = E^d [r(s, Y, X') + v(X')] \quad (10.22)$$

derived in (6.23). In this expression, the expectation is with respect to the distribution of the action a chosen by decision rule d in state s (when it is randomized) and the resulting transition probabilities $p(\cdot|s, a)$.

An implementable recursion is based on replacing the sampled trajectories h^n in (10.21) by the results of a single simulation step. In this step, a is deterministically chosen as $a = d(s)$ or sampled from the randomized distribution $w_d(\cdot|s)$. The successor state s' and reward r are jointly simulated or s' is sampled from $p(\cdot|s, a)$ and the reward is computed by $r = r(s, a, s')$. This enables direct calculation of $r + v^n(s')$. Thus, the following recursion

$$v^{n+1}(s) = v^n(s) + \tau_n (r + v^n(s') - v^n(s)) \quad (10.23)$$

provides an online update of an estimate of $v(s)$ without waiting for an episode to complete.

Often this recursion appears as

$$v^{n+1}(s) = v^n(s) + \tau_n \delta^n \quad (10.24)$$

where

$$\delta^n := r + v^n(s') - v^n(s).$$

As noted above, the quantity δ^n is referred to as a temporal difference.

When applying this recursion, the usual approach is to start in some state, generate a transition using the policy, update value estimates using (10.23) and terminate when reaching a reward-free absorbing state. However, to obtain accurate estimates in each state, one would need to repeat this over multiple episodes by restarting this process in a state chosen either deterministically or randomly.

The following subtleties arise when using this recursion:

1. **Approximation:** This recursion uses *bootstrapping*²⁸. That is, it uses an approximation of the value function $v^n(\cdot)$ at each step instead of a known quantity.

²⁸This use of the expression bootstrapping here is consistent with the reinforcement learning literature. In the statistical literature, bootstrapping refers to methods based on resampling from a given set of data.

2. **Multiplicity of successor states:** The recursion requires estimates of $v^n(s')$ for all states reachable from s in one step under d .
3. **Asynchronicity:** The quality (variance) of the approximations will vary across states. Values of states frequently visited under d will be estimated accurately while those infrequently visited will be less accurate.

As a consequence of these remarks, a rigorous theoretical analysis of this algorithm requires methods above and beyond those used to establish convergence of stochastic approximation. In spite of this, many proofs of the following result are available²⁹. This issue is explored in Chapter 11 in a more transparent setting where value functions are parameterized by a vector β and a distinction is made between gradients and semi-gradients.

Theorem 10.1. Let \mathbf{v}^{d^∞} denote the unique solution to (10.22). Then for every \mathbf{v}^0 , if every state is visited infinitely often and τ_n satisfies (10.20), \mathbf{v}^n converges to \mathbf{v}^{d^∞} with probability 1.

Note that the expression “with probability 1” can be interpreted as “on almost every sample path”, where sample paths consist of sequences of states, actions and rewards determined by the decision rule, the transition probabilities, and the distribution used to start a new episode. To ensure that every state is visited infinitely often, after reaching a terminal state, the system must be restarted using a distribution that ensures all states are reached with positive probability.

Example 10.1. The example depicted in Figure 10.9 illustrates the above challenges of applying (10.23). In this example of an episodic model, there is a single stationary policy that uses action a_1 in s_1 , a_2 in s_2 and a_3 in s_3 . Actions choose arcs with the indicated probabilities and generate the indicated rewards. Note s_3 is a zero-reward absorbing state. When s_3 is reached, an episode ends.

Suppose the system starts in s_1 , then in 999 out of 1,000 trajectories will immediately jump to s_2 , remain there for on average 500 transitions and then jump to either s_1 or s_3 with equal probability. Therefore the value of s_2 will be accurately estimated but the value of s_1 need not be.

State-based sampling provides an attractive alternative to requiring a large number of episodes to accurately estimate $v(s_1)$. Of course, in a “real-world” system such restarts may not be possible.

²⁹The proofs of this result are beyond the level of this book; references appear in the Bibliographic Remarks section.

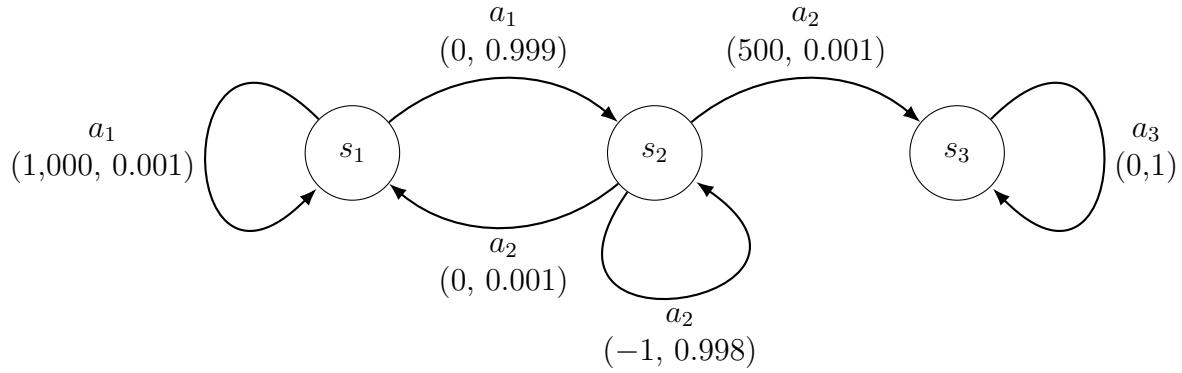


Figure 10.9: Graphical representation of model in Example 10.1. Numbers in parentheses represent $(r(s, a, s'), p(j|s, a))$.

Temporal differencing in algorithmic form

This section describes an algorithm to implement (10.23) in an episodic model. The algorithm evaluates a randomized stationary policy d^∞ , and combines both model-free and model-based implementations.

Algorithm 10.3. Temporal differencing for policy evaluation for an episodic model

1. **Initialize:**
 - (a) Specify $d \in D^{\text{MR}}$.
 - (b) Specify the number of episodes K .
 - (c) Specify the learning rate sequence $\tau_n, n = 1, 2, \dots$
 - (d) $v(s) \leftarrow 0$ for all $s \in S$.
 - (e) $\text{count}(s) \leftarrow 0$ for all $s \in S$.
 - (f) $k \leftarrow 1$.
2. **Iterate:** While $k \leq K$:
 - (a) Specify $s \in S \setminus \Delta$.
 - (b) **Generate episode:** While $s \notin \Delta$:
 - i. $\text{count}(s) \leftarrow \text{count}(s) + 1$.
 - ii. Sample a from $w_d(\cdot|s)$.
 - iii. Simulate (s', r) or sample s' from $p(\cdot|s, a)$ and set $r \leftarrow r(s, a, s')$.

iv. Evaluate temporal difference

$$\delta \leftarrow r + v(s') - v(s). \quad (10.25)$$

v. Set

$$v(s) \leftarrow v(s) + \tau_{\text{count}(s)} \delta. \quad (10.26)$$

vi. $s \leftarrow s'$.

(c) $k \leftarrow k + 1$.

3. **Terminate:** Return $v(s)$ for all $s \in S$.

Some comments about Algorithm 10.3 follow:

1. Temporal differencing is an online algorithm with asynchronous updates.
2. It provides estimates of the policy value function for every state visited during at least one episode when episodes are replicated.
3. In contrast to Monte Carlo estimation in which value function updates occur at the end of an episode, this algorithm updates the value function estimate within each episode after each transition. At the end of an episode, a new initial state is generated in step 2(a).
4. The initial state may be generated by any, possibly random, mechanism that is appropriate for a problem. For example, it is often advantageous to start far from absorbing states so that many states may be visited (and the values corresponding to those states updated) during each episode. However, a random start over all states may provide accurate estimates for states visited infrequently under decision rule d . This will be particularly relevant when seeking an optimal policy.
5. Monte Carlo methods might be used to obtain good starting values for $v(s)$ before applying temporal differencing.
6. The algorithm uses the number of visits to a state as the index for the learning rate parameter τ_n to account for asynchronicity in updates.
7. Choosing the learning rate τ_n is somewhat of an art. A systematic approach (sometimes referred to as a *parameter study*) that combines RMSE with graphical summaries for guidance is described in the chapter appendix 10.11.1 and illustrated in examples below. The next section expands on this issue.

8. After applying (10.26) in state s , it is possible to update values in states previously visited during an episode. This is the idea underlying TD(γ) algorithms, which are described below.
9. In step 2(b)iii, the next state s' (and the reward r) may be directly obtained using a simulator in a model-free environment or generated from transition probabilities and rewards in a model-based environment. Note that when d is deterministic, step 2(b)ii becomes “Set $a = d(s)$ ”.

Learning rate choice

The following observation³⁰ is consistent with our experience choosing the learning rates (step-sizes): “*Step-size rules are important. As you experiment with ADP³¹, it is possible to find problems where provably convergent algorithms simply do not work, and the only reason is a poor choice of step-sizes.*”

Ideally, learning rates should produce fast learning early and slow learning later. Theorem 10.1 provides conditions (10.20) that do this and as well guarantee convergence of temporal differencing. However, ensuring that learning rates satisfy these conditions is not sufficient to obtain accurate estimates in practice.

Learning rate sequences that satisfy the Robbins-Monro conditions include:

Polynomial:

$$\tau_n = \alpha n^{-\beta}$$

where $0.5 < \beta \leq 1$. Note that $1/n$ is a special case corresponding to $\alpha = \beta = 1$.

Ratio:

$$\tau_n = \frac{\alpha}{\beta + n}$$

for $\alpha > 0$ and $\beta \geq 0$. Note that this sequence starts at $\alpha/(\beta + 1)$. Moreover when $\alpha = 1$ and $\beta = 0$ this reduces to $1/n$.

Search-then-converge (STC)³²:

$$\tau_n = \alpha \left(\frac{1 + \frac{\beta n}{\alpha \gamma}}{1 + \frac{\beta n}{\alpha \gamma} + \frac{n^2}{\gamma}} \right)$$

When n is much smaller than γ , τ_n is nearly constant while for large n it behaves as β/n . Notice that choosing $\beta = 0$ makes τ_n decay more quickly.

Log:

$$\tau_n = \log(n+1)/n$$

A multiplicative constant can be added to increase flexibility.

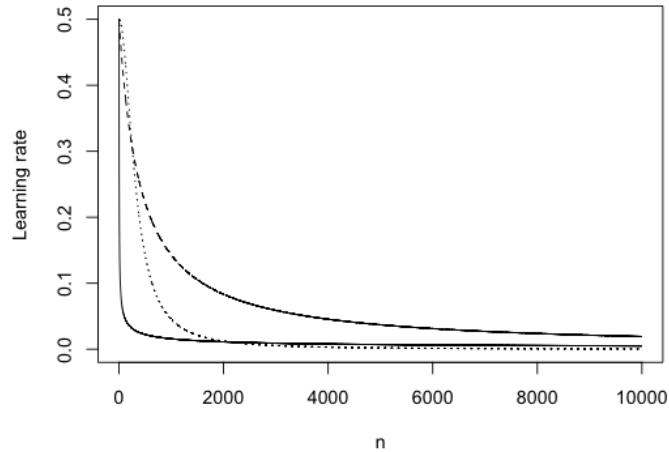


Figure 10.10: Graphical representation of three learning rate sequences. The solid line represents the polynomial sequence with $\alpha = \beta = 0.5$, the dashed line indicates the ratio sequence with $\alpha = 200$ and $\beta = 400$ and the dotted line corresponds to the STC sequence with $\alpha = 0.5$, $\beta = 0.0001$ and $\gamma = 100,000$.

Figure 10.10 shows how the learning rate decays with n for three specific τ_n . Observe that for the indicated parameter values, the polynomial sequence decays very quickly, the ratio sequence decays more slowly and the STC sequence lies somewhere in between.

Note that the quantity n in the above specifications may represent either:

1. the iteration number, or
2. the number of times a state³³ has been visited.

The iteration number benefits from simplicity. However, in algorithms using trajectory-based sampling, in which some states are visited infrequently, using the number of times a state has been visited seems more reasonable. It ensures that infrequently visited states are updated at a larger learning rate than if the iteration number had been used. Of course this involves extra storage but when a tabular model specification is appropriate, this should not be problematic. When states, or state-action pairs, are sampled randomly, either specification can be used.

³⁰Powell, p. 184.

³¹In this context, ADP refers to simulation-based methods.

³²This approach was described in Darken et al. [1992].

³³Or state-action pair when estimating a state-action value function.

Temporal differencing in the Gridworld model

This section applies Algorithm 10.3 to the Gridworld model analyzed above using Monte Carlo methods in Section 10.3.1. It estimates the value function for the policy specified in that example. It compares four learning rates over 40 replicates of 10,000 episodes each with common random number seeds for each replicate. Each episode starts in cell 13 (the coffee room) and terminates when the robot reaches either cell 1 or 7.

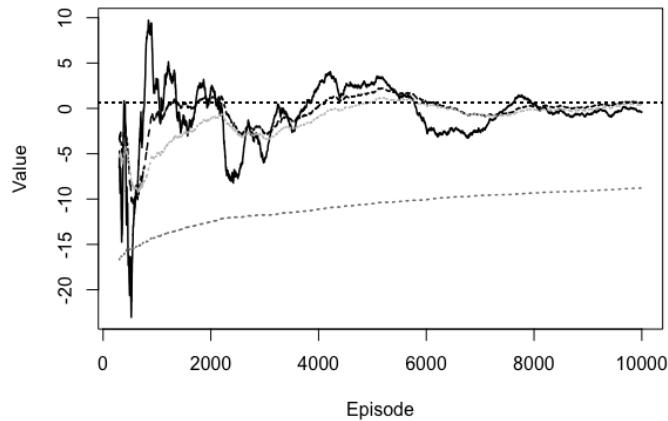


Figure 10.11: Estimates of $v^{d^\infty}(13)$ in the Gridworld model using Algorithm 10.3 based on a single replicate of 10,000 episodes. The figure shows the sequence of iterates for $\tau_n = 20/(50 + n)$ (solid black), $\tau_n = \log(n + 1)/n$ (dashed black), $\tau_n = 0.8n^{-0.75}$ (light grey) and $\tau_n = 0.8n^{-0.95}$ (dashed grey). The horizontal dotted line corresponds to the theoretical (true) value of this quantity 0.638.

Figure 10.11 displays a sequence of temporal difference estimates of $v^{d^\infty}(13)$, the expected total reward achieved by the robot starting in the coffee room. It shows iterates for the four choices of τ_n described in the figure caption. These were chosen based on some preliminary parameter studies.

Observe that $\tau_n = \log(n + 1)/n$ and $\tau_n = 0.8n^{-0.75}$ produced the best estimates in this replicate. They converged quickly to the true value and oscillations damped out. Estimates using $\tau_n = 20/(50 + n)$ oscillated around the true value but were more variable than the other two choices. That for $\tau_n = 0.8n^{-0.95}$ stabilized quickly and had low variance but was extremely inaccurate.

Figure 10.12 illustrates the variability in RMSE across 40 replicates where the

RMSE is computed over non-terminal states using

$$\text{RMSE} = \left(\frac{1}{13} \sum_{s \in S \setminus \Delta} (\hat{v}(s) - v(s))^2 \right)^{\frac{1}{2}}. \quad (10.27)$$

In this expression, $\hat{v}(s)$ is based on the final episode for that state, while the true value is determined exactly using the methods from Chapter 6. This measure accounts for estimation error in all non-terminal states.

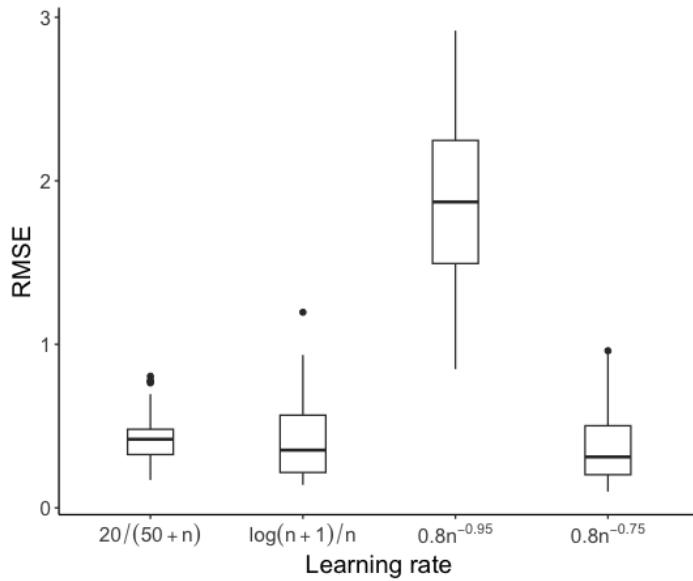


Figure 10.12: Boxplots of RMSE showing effect of learning rate when applying temporal differencing in Gridworld model.

It shows that the ratio estimate was least variable but slightly less accurate than those based on the logarithmic learning rate and the polynomial learning rate $0.8n^{-0.75}$. Clearly $\tau_n = 0.8n^{-0.95}$ was the worst choice. Note that RMSE weights all states equally. Alternative weightings are possible.

10.3.3 $\text{TD}(\gamma)$

The temporal differencing method above updates values one state at a time. Consequently, it ignores the effect of the current temporal difference on previously visited state values. This section describes an approach that overcomes this. It may be viewed as either an *offline or online method*.

As motivation, suppose in the Gridworld model (Figure 10.6) that the robot starts in cell 4 and visits cells 5, 6, 3, and 2 before terminating in cell 1. Recall that each step costs 1 and termination in cell 1 generates a reward of 50.

Consider updates starting in cell 6. After a transition to cell 3, the temporal difference update would be

$$v(6) \leftarrow v(6) + \tau(-1 + v(3) - v(6)).$$

But the value function could just as well be updated using a two-step update based on transitions from 6 to 3 and then 3 to 2 according to

$$v(6) \leftarrow v(6) + \tau(-1 - 1 + v(2) - v(6)),$$

or even a three-step update

$$v(6) \leftarrow v(6) + \tau(-1 - 1 - 1 + 50 + v(1) - v(6)).$$

Instead of selecting just one, it might be expedient to combine these three updates in some way. TD(γ)³⁴ provides a structured way to do so.

Two-step evaluation equations

The following formalizes the above argument in a model-based environment. The temporal difference algorithm above updates an estimate of the value of stationary policy d^∞ in state s according to

$$v(s) \leftarrow v(s) + \tau(r(s, a, s') + v(s') - v(s)),$$

where a is sampled from $w_d(\cdot|s)$ and the subsequent state s' is sampled from $p(\cdot|s, a)$. As noted above, this may be viewed as applying gradient descent to minimize the squared difference between the two sides of

$$v(s) = E^{d^\infty} [r(X_n, Y_n, X_{n+1}) + v(X_{n+1}) | X_n = s]. \quad (10.28)$$

Applying a similar recursion for $v(X_{n+1})$ and substituting it into (10.28) leads to the two-step evaluation equation beginning at decision epoch n :

$$v(s) = E^{d^\infty} [r(X_n, Y_n, X_{n+1}) + r(X_{n+1}, Y_{n+1}, X_{n+2}) + v(X_{n+2}) | X_n = s]. \quad (10.29)$$

This suggests an alternative two-step update for $v(s)$ given by

$$\begin{aligned} v(s) &\leftarrow v(s) + \tau(r(s, a, s') + r(s', a', s'') + v(s'') - v(s)) \\ &= v(s) + \tau(r(s, a, s') + v(s') - v(s) + r(s', a', s'') + v(s'') - v(s')) \\ &= v(s) + \tau(\delta + \delta'), \end{aligned} \quad (10.30)$$

³⁴Note that most of the literature refers to this approach as TD(λ) but since this book uses λ to denote the discount factor, it instead uses γ to represent the algorithmic parameter.

where a' denotes the realization of the action at decision epoch $n + 1$ and s'' is a realization of the state at decision epoch $n + 2$,

$$\delta = r(s, a, s') + v(s') - v(s) \quad \text{and} \quad \delta' = r(s', a', s'') + v(s'') - v(s').$$

This implies that the update of $v(s)$ is delayed until two decision epochs later.

However, this update can be viewed as two one-step updates as follows. After the first transition, update

$$v(s) \leftarrow v(s) + \tau_1 \delta$$

and leave the values the same for all other states. Then, in the next iteration, update

$$v(s) \leftarrow v(s) + \tau_2 \delta' \quad \text{and} \quad v(s') \leftarrow v(s') + \tau_2 \delta'$$

and leave values for other states unchanged. Note that in contrast to the combined two-step update in (10.30), different learning rates can be used in each iteration.

Online TD(γ) modifies the second update for state s by reducing its impact by a factor of γ , $0 \leq \gamma \leq 1$, so that the second update becomes

$$v(s) \leftarrow v(s) + \tau_2 \gamma \delta' \quad \text{and} \quad v(s') \leftarrow v(s') + \tau_2 \delta'.$$

Multi-step and weighted evaluation equations

Repeating the above argument leads to the m -step evaluation equation (beginning at decision epoch n):

$$v(s) = E^{d^\infty} \left[\sum_{k=0}^{m-1} r(X_{n+k}, Y_{n+k}, X_{n+k+1}) + v(X_{n+m}) \middle| X_n = s \right]. \quad (10.31)$$

Define

$$R_n^m := \sum_{k=0}^{m-1} r(X_{n+k}, Y_{n+k}, X_{n+k+1}) + v(X_{n+m}) \quad (10.32)$$

for $m \geq 1$. The random variable R_n^m represents the total reward over decision epochs $n, n + 1, \dots, n + m - 1$ plus a terminal reward $v(X_{n+m})$. These cumulative rewards may be viewed as the result of *rolling out* a policy over multiple periods starting at decision epoch n . Observe that (10.31) can be rewritten compactly as

$$v(s) = E[R_n^m | X_n = s].$$

Instead of arbitrarily combining these rollouts or using a pre-specified number of them, an option is to use the following exponential weighted average:

$$v(s) = E \left[\sum_{m=1}^{\infty} (1 - \gamma) \gamma^{m-1} R_n^m \middle| X_n = s \right]. \quad (10.33)$$

From (10.32), R_n^m is itself a summation, so interchanging the order of summation as carried out in Appendix 10.11.4 leads to the following equivalent representation for $v(s)$:

$$v(s) = v(s) + \sum_{m=0}^{\infty} \gamma^m D_{n+m}, \quad (10.34)$$

where

$$D_{n+m} := E[r(X_{n+m}, Y_{n+m}, X_{n+m+1}) + v(X_{n+m+1}) - v(X_{n+m}) | X_n = s].$$

Offline TD(γ)

Equation (10.34) provides the basis for deriving an offline representation for TD(γ). As shown in Appendix 10.11.4, the offline TD(γ) representation becomes

$$v(s^n) \leftarrow v(s^n) + \tau \sum_{m=n}^{\infty} \gamma^{m-n} \delta_m, \quad (10.35)$$

where

$$\delta_k := r(s^k, a^k, s^{k+1}) + v(s^{k+1}) - v(s^k).$$

Under the assumption of a transient model or a proper policy in a stochastic shortest path model, the rewards based on a trajectory (s^1, a^1, s^2, \dots) equal zero after some finite number of terms. Therefore, even though (10.35) has an infinite number of terms, they will equal zero after some trajectory-specific index N so that the above sum can be evaluated.

This can be applied offline in a similar way to a sequential implementation of first-visit or every-visit Monte Carlo over many trajectories. To do so, start with initial values of $v(s)$ for all $s \in S \setminus \Delta$. After each trajectory is completed, update the values for all states visited during that trajectory using (10.35) with τ decreasing over trajectories.

Observe that when $\gamma = 0$, (10.35) becomes

$$v(s^n) \leftarrow v(s^n) + \tau \delta_n,$$

which is exactly the temporal differencing update encountered earlier. Thus:

Temporal differencing is equivalent to TD(0).

When $\gamma = 1$, (10.35) becomes

$$v(s^n) \leftarrow v(s^n) + \tau \sum_{m=n}^{\infty} \delta_m = (1 - \tau)v(s^n) + \tau \sum_{m=n}^{\infty} r(s^m, a^m, s^{m+1}).$$

Since the quantity in the sum is a Monte Carlo estimate of the total reward starting in state s^n , TD(1) is equivalent to a stochastic approximation method that updates the value function estimate for each state using the Monte Carlo update as in (10.21). Thus:

A variant of Monte Carlo estimation is equivalent to TD(1).

From a theoretical perspective³⁵, offline (batch) TD(γ) converges with probability one to $v^{d^\infty}(s)$ for all $s \in S \setminus \Delta$ provided there are no inaccessible states under the initial distribution, $v^{d^\infty}(s)$ is finite for all $s \in S \setminus \Delta$, and the Robbins-Monro step-size conditions hold.

Online TD(γ)

To obtain an online version of TD(γ) requires a change in perspective. In each episode, it uses the most recently observed temporal difference to update values in *all previously* visited states in that episode. This is in contrast to the temporal differencing method described above which updates values for the most recently visited state. The weightings are chosen to be consistent with the geometric weightings in offline TD(γ).

Suppose that, under a specified stationary policy, the states visited during an episode of length N are denoted by (s^1, s^2, \dots, s^N) . Then after observing s^n and computing δ_n , online TD(γ) updates the policy value function in states (s^1, s^2, \dots, s^n) according to:

$$\begin{aligned} v(s^n) &\leftarrow v(s^n) + \tau_n \delta_n \\ v(s^{n-1}) &\leftarrow v(s^{n-1}) + \tau_n \gamma \delta_n \\ &\vdots \\ v(s^{n-k}) &\leftarrow v(s^{n-k}) + \tau_n \gamma^k \delta_n \\ &\vdots \\ v(s^1) &\leftarrow v(s^1) + \tau_n \gamma^{n-1} \delta_n. \end{aligned} \tag{10.36}$$

³⁵See Dayan and Sejnowski [1994] which establishes this result for models with linear value function approximations, tabular models being a special case.

Observe that the update in state s^n is the same as in the temporal differencing algorithm and that the temporal difference δ_n is used to update all previously visited states.

It is important to note that when the current state s^n has been visited more than once in a trajectory, the above scheme updates its value multiple times. An alternative approach would be to only update the value *once* using the temporal differencing update $v(s^n) \leftarrow v(s^n) + \tau_n \delta_n$.

Eligibility traces

The *eligibility trace* is an innovation that assigns the appropriate weighting of the current temporal difference when updating values for previously visited states. Two variants have been proposed, the *accumulating eligibility trace* and the *replacement eligibility trace*, differing with respect to how values in repeat observations are updated. Often they will be referred to as simply an accumulating trace or a replacement trace.

More formally, an eligibility trace is a sequence of non-negative real-valued functions defined on a sequence of states (s^1, s^2, \dots) in $S \setminus \Delta$. The accumulating trace, which corresponds to updating values corresponding to all previous visits to the current state, is defined recursively by $e_1(s) = I_{\{s^1\}}(s)$ and

$$e_n(s) := \gamma e_{n-1}(s) + I_{\{s^n\}}(s). \quad (10.37)$$

Observe that for $n = 1$,

$$e_1(s) = \begin{cases} 1 & \text{for } s = s^1 \\ 0 & \text{for } s \neq s^1 \end{cases}$$

and $n > 1$,

$$e_n(s) = \begin{cases} \gamma e_{n-1}(s) + 1 & \text{for } s = s^n \\ \gamma e_{n-1}(s) & \text{for } s \neq s^n. \end{cases} \quad (10.38)$$

Using the eligibility trace as defined above, the sequence of updates (10.36) for $s \in S \setminus \Delta$ can be written in condensed form as:

$$v(s) \leftarrow v(s) + \tau_n \delta_n e_n(s). \quad (10.39)$$

The replacement trace modifies (10.38) for $n > 1$ to be

$$e_n(s) = \begin{cases} 1 & \text{for } s = s^n \\ \gamma e_{n-1}(s) & \text{for } s \neq s^n. \end{cases} \quad (10.40)$$

The effect of this modification becomes clear by using it in (10.39). It is easy to see that the consequence of using the replacement trace is that the update of $v(s^n)$ at iteration n is $v(s^n) \leftarrow v(s^n) + \tau_n \delta_n$, whereas when using the accumulating trace the expression $\tau_n \delta_n$ would be multiplied by a polynomial in γ .

An example

The following example³⁶ compares online and offline TD(γ) and also illustrates the effect of trace types on online updates. Consider a two-state model with a single policy. The state space consists of a state s_0 and a zero-reward absorbing state Δ . There is a single policy that generates a reward of r in s_0 at each decision epoch and the system remains in s_0 with probability p or jumps to Δ with probability $1 - p$.

Since there is a single non-absorbing state, all temporal differences prior to absorption in Δ simplify to

$$\delta_k = r + v(s_0) - v(s_0) = r.$$

It is easy to see that $v(s_0) = r/(1 - p)$, but that is not the point of this example. Suppose the system starts in s_0 and is absorbed in Δ after two transitions so that the trajectory is (s_0, s_0, Δ) . The following calculations compare online and offline TD(γ).

Initialize calculations with $v(s_0) = c$. Then the offline update corresponding to this trajectory is

$$v^{\text{offline}}(s_0) \leftarrow c + \tau(1 + \gamma)r.$$

The online updates using the accumulating trace and assuming $\tau_1 = \tau_2 = \tau$ are $e_1(s_0) = 1$ and

$$v_1^{\text{online}}(s_0) \leftarrow c + \tau r$$

corresponding to the first transition and since $e_2(s_0) = 1 + \gamma$,

$$v_2^{\text{online}}(s_0) \leftarrow c + \tau r + \tau(1 + \gamma)r = c + 2\tau r + \tau\gamma r$$

corresponding to the second transition.

Using the replacement trace, $e_1(s_0) = e_2(s_0) = 1$ so that the first update is the same as using the accumulation trace, but the second update becomes

$$v_2^{\text{online}'}(s_0) \leftarrow c + 2\tau r.$$

Observe that:

1. The initial condition $v(s_0) = c$ persists in all updates. Therefore, it must be set to $c = 0$ to ensure convergence to the appropriate quantity.
2. The online and offline updates differ by a factor that is linear in τ .
3. The two online updates differ by a factor of $\tau\gamma$.
4. In trajectories of length n , the online update with replacement trace will be $c + n\tau r$. Since the expected total reward is $E[N]r$, the online estimate has exactly the same form as the quantity being estimated.

Consequently, if τ is small, estimates based on longer trajectories will be close.

³⁶This is motivated by Example 5.7 in Bertsekas and Tsitsiklis [1996]. Their example has two non-absorbing states so it also shows the effect of the learning rate on transitions.

An online TD(γ) algorithm

The following algorithm implements online TD(γ). It incorporates the eligibility trace in the algorithmic flow. The precise update depends on whether an accumulating or replacement trace is used.

Algorithm 10.4. TD(γ) for policy evaluation for an episodic model

1. **Initialize:**

- (a) Specify $d \in D^{\text{MR}}$.
- (b) Set $v(s) \leftarrow 0$, $e(s) \leftarrow 0$ and $\text{count}(s) \leftarrow 0$ for all $s \in S$.
- (c) Specify $0 \leq \gamma \leq 1$ and learning rate sequence $\tau_n, n = 1, 2, \dots$
- (d) Specify the number of episodes K and set $k \leftarrow 1$.

2. **Iterate:** While $k \leq K$:

- (a) Specify $s \in S \setminus \Delta$.
- (b) **Generate episode:** While $s \notin \Delta$:
 - i. $e(s) \leftarrow e(s) + 1$ (accumulating trace) or $e(s) \leftarrow 1$ (replacement trace).
 - ii. $\text{count}(s) \leftarrow \text{count}(s) + 1$.
 - iii. Sample a from $w_d(\cdot|s)$.
 - iv. Simulate (s', r) or sample s' from $p(\cdot|s, a)$ and set $r \leftarrow r(s, a, s')$.
 - v. Evaluate temporal difference

$$\delta \leftarrow r + v(s') - v(s). \quad (10.41)$$

vi. For all $s \in S$,

$$v(s) \leftarrow v(s) + \tau_{\text{count}(s)} \delta e(s) \quad (10.42)$$

and

$$e(s) \leftarrow \gamma e(s). \quad (10.43)$$

vii. $s \leftarrow s'$.

- (c) $k \leftarrow k + 1$.

3. **Terminate:** Return $v(s)$ for all $s \in S$.

Some observations regarding this algorithm follow:

1. It is an online algorithm. It differs from Algorithm 10.3 in that at each state it updates values for **all** previously visited states. The magnitude of the update

decreases exponentially at rate γ .

2. A variant of the algorithm would reset eligibility traces to 0 at the end of each episode.
3. Small values of γ result in updating values for only a few previously visited states while values near 1 update this value far into the past.
4. The algorithm updates the eligibility trace online in steps 2(b)i and 2(b)vi. These ensure that the eligibility trace values agree with (10.38) and (10.40).
5. Convergence results apply to the algorithm implemented over infinitely many episodes. If every state is visited infinitely often over repeated episodes and the Robbins-Monro conditions are met, the online $\text{TD}(\gamma)$ estimates converge³⁷ with probability one to the true policy value function.

An example

Before applying this algorithm in an example, examine how the eligibility trace updating works. Suppose in the Gridworld example the robot follows the trajectory $4 \rightarrow 5 \rightarrow 2 \rightarrow 5 \rightarrow 4 \rightarrow 1$. The above algorithm updates the eligibility trace at step 2(b)i prior to applying (10.42) and again in step 2(b)vi through (10.43).

Non-zero values of $e(s)$ (when using the accumulating trace) prior to applying (10.43) are given below:

1. $e(4) = 1$.
2. $e(5) = 1, e(4) = \gamma$.
3. $e(2) = 1, e(4) = \gamma^2, e(5) = \gamma$.
4. $e(5) = 1 + \gamma, e(2) = \gamma, e(4) = \gamma^3$.
5. $e(4) = 1 + \gamma^3, e(5) = \gamma(1 + \gamma), e(2) = \gamma^2$.

Note that in step 5, $e(5)$ combines two terms to account for the two visits to cell 5.

If instead, the replacement trace had been used, the last two steps would be different. In step 4, $e(5) = 1$, and in step 5, $e(4) = 1$ and $e(5) = \gamma$. Note that if γ were near 1, the accumulating trace can exceed 1 in value.

The following example illustrates the application of $\text{TD}(\gamma)$ to evaluate a policy in the Gridworld model.

³⁷See Bertsekas and Tsitsiklis (1996) and Jaakkola et al. (1994) for details.

Example 10.2. Policy evaluation using $\text{TD}(\gamma)$ in an episodic model

This example applies $\text{TD}(\gamma)$ to the Gridworld model. Each episode starts in cell 13 and terminates when the robot reaches either cell 1 or cell 7. $\text{TD}(\gamma)$ is used to again evaluate the policy that aims to move the robot to the lowest number adjacent cell, except in cell 10 when it attempts to move it to cell 11. It explores the effect of γ , trace type and learning rate on the accuracy of estimates.

Forty replicates of $K = 5,000$ episodes were compared for each combination of $\gamma \in \{0, 0.2, 0.4, 0.6, 0.8, 0.95\}$, accumulating and replacement eligibility trace and $\tau_n = 0.8n^{-0.75}$, $\tau_n = 20/(50 + n)$, $\tau_n = \log(n + 1)/n$ with the same seed in each replicate for all combinations. Results were assessed on the basis of the RMSE of the final estimates as defined in (10.27).

When $p = 0.8$, the most accurate estimates were obtained using $\tau_n = 0.8n^{-0.75}$, $0 \leq \gamma \leq 0.6$ with the trace type having little effect when γ was small (Figure 10.13). Observe also that for large values of γ , using the replacement trace resulted in more accurate estimates.

Moreover with this learning rate, the $\text{TD}(\gamma)$ estimates were similar for $\gamma = 0, 0.2, 0.4$ with minimal differences between their accuracy. Consequently, there was minimal benefit using $\text{TD}(\gamma)$ instead of temporal differencing ($\text{TD}(0)$).

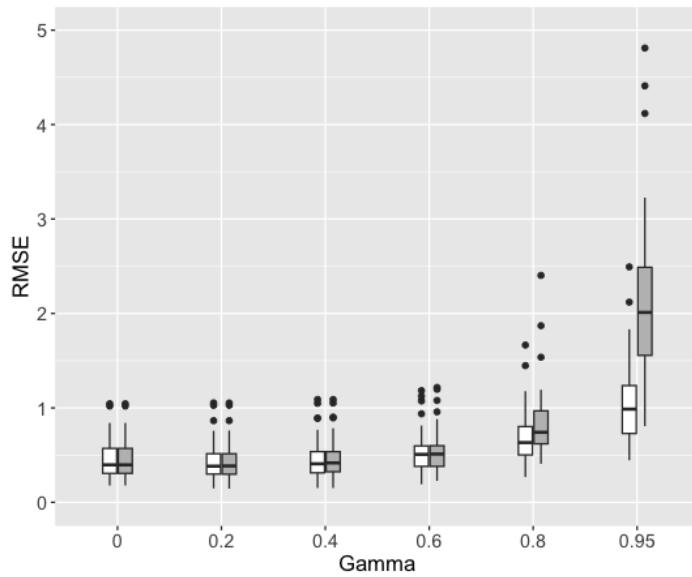


Figure 10.13: Boxplot of RMSE for Gridworld model showing the effect of γ and trace type (white represents replacement trace and grey represents accumulating trace) for learning rate $\tau_n = 0.8n^{-0.75}$.

10.4 Policy evaluation: Infinite horizon discounted models

This section discusses Monte Carlo and TD(γ) approaches for estimating the value of a policy in a discounted model.

10.4.1 Monte Carlo methods

Monte Carlo methods in an infinite horizon discounted model are based on either truncating the reward sequence or viewing the discount rate as the parameter of a geometrically distributed stopping time. Only starting-state Monte Carlo is applicable because the truncation index or geometric stopping time are based on the first decision epoch. For example, suppose an episode is truncated (or stopped) after 50 decisions, the system starts in state s^1 and visits states s^2, s^3, \dots, s^{50} . Then an episode starting in state s^{25} would be based on accumulating only 25 rewards, while that starting in state s^1 would be based on accumulating 50 rewards. Therefore, an estimate based on s^{25} would not estimate the value function to the same precision as one based on s^1 . This differs from the discussion above of episodic models where stopping occurs when a state in Δ is reached.

More formally, a policy value function in a discounted model may be written as

$$v_\lambda^{d^\infty}(s) = E^{d^\infty} \left[\sum_{n=1}^{\infty} \lambda^{n-1} r(X_n, Y_n, X_{n+1}) \mid X_1 = s \right].$$

Because of the infinite sum, Monte Carlo methods cannot be applied directly to evaluate $v_\lambda^{d^\infty}(s)$. Instead, they can be based on approximating $v_\lambda^{d^\infty}(s)$ by either a *truncated sum of discounted rewards*

$$v_\lambda^{d^\infty}(s) \approx E^{d^\infty} \left[\sum_{n=1}^M \lambda^{n-1} r(X_n, Y_n, X_{n+1}) \mid X_1 = s \right] \quad (10.44)$$

for some finite and sufficiently large M or the *undiscounted random sum* representation

$$v_\lambda^{d^\infty}(s) = E_T^{d^\infty} \left[\sum_{n=1}^T r(X_n, Y_n, X_{n+1}) \mid X_1 = s \right], \quad (10.45)$$

where T is an independent geometric random variable with parameter $1-\lambda$. In the latter case, the expectation is with respect to the probability distribution of (X_1, Y_1, X_2, \dots) generated by the specified policy and T . In (10.45), the discount factor appears implicitly in the expectation of T . Note that equation 10.45 is exact while 10.44 is an approximation.

Truncated reward sequences

Given a realization of states and actions, (s^1, a^1, s^2, \dots) , at issue is when to truncate $\sum_{n=1}^{\infty} \lambda^{n-1} r(s^n, a^n, s^{n+1})$. This can be done *a priori* by choosing M so that

$$\lambda^{M-1} \max_{s \in S, a \in A_s, s' \in S} |r(s, a, s')| \leq \epsilon \quad (10.46)$$

for some pre-specified tolerance ϵ . Alternatively, M can be chosen arbitrarily or by stopping when the change in values of successive estimates is small.

Algorithm 10.5. Monte Carlo policy evaluation for a discounted model using truncation

1. **Initialize:**
 - (a) Specify $d \in D^{\text{MR}}$.
 - (b) Specify the number of replicates K and set $k \leftarrow 1$.
 - (c) Specify the truncation level M .
 - (d) Specify $\bar{s} \in S$.
 - (e) Create an empty list VALUES.
2. **Replicate:** While $k \leq K$:
 - (a) $s \leftarrow \bar{s}$, $m \leftarrow 1$ and $u \leftarrow 0$.
 - (b) **Generate a replicate:** While $m \leq M$:
 - i. Sample a from $w_d(\cdot|s)$.
 - ii. Simulate (s', r) or sample s' from $p(\cdot|s, a)$ and set $r \leftarrow r(s, a, s')$.
 - iii. $u \leftarrow u + \lambda^{m-1} r$.
 - iv. $s \leftarrow s'$.
 - v. $m \leftarrow m + 1$.
 - (c) **Update:**
 - i. Append u to VALUES.
 - ii. $k \leftarrow k + 1$.
3. **Terminate:** Return

$$\hat{v}_\lambda^{d^\infty}(\bar{s}) = \text{mean}(\text{VALUES}).$$

Some comments about the algorithm follow:

1. This algorithm uses the expression *replicate* to refer to a single realization of a

truncated reward series in contrast to the use of the expression *episode* in episodic Monte Carlo.

2. As stated, the algorithm updates *online* within each replicate to obtain an estimate u to append to the list of values generated so far. Alternatively, this algorithm can be implemented *offline* by generating a sequence (r^1, \dots, r^M) and setting $u = \sum_{m=1}^M \lambda^{m-1} r^m$.
3. As stated, the algorithm generates a list VALUES. The advantage of doing so is that it can be used to investigate distributional properties of estimators.
4. After evaluating u based on a complete replicate, the estimate of $v_\lambda^{d^\infty}(s)$ can be updated using the stochastic approximation recursion:

$$v(\bar{s}) \leftarrow v(\bar{s}) + \tau_n(u - v(\bar{s})). \quad (10.47)$$

Of course, choosing $\tau_n = n^{-1}$ gives an estimate of the mean.

Geometric stopping times

Using representation (10.45) for $v_\lambda^{d^\infty}(s)$ is equivalent to terminating each episode at the first “success” in a sequence of Bernoulli trials with success probability $1-\lambda$.

To implement this approach modify Algorithm 10.5 as follows:

1. At the start of each replicate, generate a random stopping time from a geometric distribution³⁸.
2. Set $\lambda = 1$ in 2(b)iii so that the update is $u \leftarrow u + r$.

An example

The following example compares variants of Monte Carlo estimation using the two-state model of Section 2.5. Of course it would not be prudent to use simulation in such a simple model but it provides simple illustrations of the issues arising when estimating policy value functions for a discounted model using simulation.

Example 10.3. Comparison of truncation and random geometric stopping time estimates

This example analyzes the two-state model using Monte Carlo estimation with truncation and geometric stopping times. It evaluates the deterministic stationary policy d^∞ with $d(s_1) = a_{1,1}$ and $d(s_2) = a_{2,2}$. Setting $\lambda = 0.9$ and solving

³⁸Note that some programming languages (such as R) represent the geometric distribution as the number of failures up to the first success. If this is the case, it is necessary to add 1 to the simulated value to be consistent with (10.45).

$(\mathbf{I} - \lambda \mathbf{P}_d) \mathbf{v} = \mathbf{r}_d$ establishes that $v_\lambda^{d^\infty}(s_1) = 30.147$ and $v_\lambda^{d^\infty}(s_2) = 27.941$.

The example compares the estimate of $v_\lambda^{d^\infty}(s)$ based on fixed truncation at $M = 30, 50, 70$ and random geometric truncation where the estimates are updated between replicates using (10.47) with learning rates $\tau_n = \log(1 + n)/n, 20/(50 + n), n^{-1}, 0.8n^{-0.75}$. As noted earlier, $\tau_n = n^{-1}$ provides a running estimate of the mean. For each combination of method and learning rate, 100 instances of 1,000 replicates each were generated with common random number seeds for each instance. The quality of the estimates was assessed on the basis of the RMSE over the two states for the 100 instances.

Figure 10.14 provides boxplots of the RMSEs by method and learning rate. Observe that:

1. For each method, the estimate based on the mean ($\tau_n = n^{-1}$) had the smallest median RMSE and was the least variable. This is to be expected since the mean over instances is the minimum variance unbiased estimator of the mean.
2. Estimates using the ratio learning rate were the least accurate and most variable.
3. The quality of truncation estimates improved with increasing M with the average accuracy of those for $M = 50$ and $M = 70$ similar. The mean RMSE for $M = 50$ and $M = 70$ differed by 0.005 and the standard deviation when $M = 50$ was 0.44 and that for $M = 70$ was 0.48.
4. For all learning rates, the estimates based on geometric stopping times were most variable. This could be partially explained by noting that the mean number of episodes evaluated using geometric stopping times is $(1-\lambda)^{-1} = 10$.

The above observations indicate that using the Monte Carlo estimate with $M = 50$ or $M = 70$ gave the most precise estimates. Nothing was to be gained by using the hybrid TD-Monte Carlo approach based on (10.47). The reason for the imprecision of the geometric estimates was the presence of an additional source of variability resulting from simulating the reward sequence length.

10.4.2 Online $\text{TD}(\gamma)$ in an infinite horizon discounted model

Because reward sequences for discounted models are infinitely long, offline $\text{TD}(\gamma)$ faces the same challenges that impacted Monte Carlo estimation. Consequently, $\text{TD}(\gamma)$ is typically implemented *online* using either trajectory-based or state-based data. While both data types are appropriate in simulations, only trajectory-based data is viable for real-time applications.

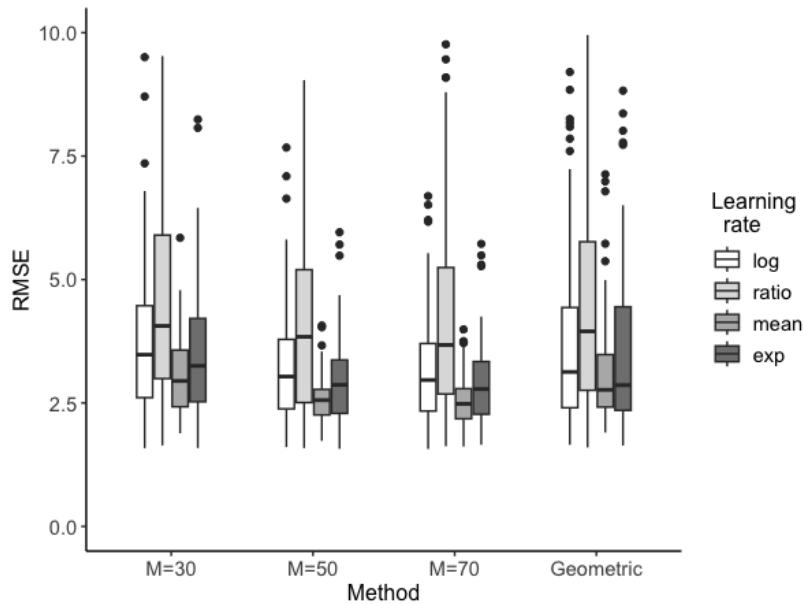


Figure 10.14: Boxplots of RMSEs for Example 10.3 by method and learning rate.

The TD(γ) algorithm below assumes that data is trajectory-based and the accumulating trace is used. A modification for a state-based approach is described below.

Algorithm 10.6. TD(γ) for policy evaluation for a discounted model

1. **Initialize:**

- (a) Specify $d \in D^{\text{MR}}$.
- (b) Set $v(s) \leftarrow 0$, $e(s) \leftarrow 0$ and $\text{count}(s) \leftarrow 0$ for all $s \in S$.
- (c) Specify $0 \leq \gamma \leq 1$ and learning rate sequence $\tau_n, n = 1, 2, \dots$
- (d) Specify the number of iterations N , set $n \leftarrow 1$ and specify $s \in S$.

2. **Iterate:** While $n \leq N$:

- (a) $e(s) \leftarrow e(s) + 1$.
- (b) $\text{count}(s) \leftarrow \text{count}(s) + 1$.
- (c) Sample a from $w_d(\cdot|s)$.
- (d) Simulate (s', r) or sample s' from $p(\cdot|s, a)$ and set $r \leftarrow r(s, a, s')$.
- (e) Evaluate temporal difference

$$\delta \leftarrow r(s, d(s), s') + \lambda v(s') - v(s). \quad (10.48)$$

(f) For all $s \in S$,

$$v(s) \leftarrow v(s) + \tau_{\text{count}(s)} \delta e(s) \quad (10.49)$$

and

$$e(s) \leftarrow \lambda \gamma e(s). \quad (10.50)$$

(g) $s \leftarrow s'$ and $n \leftarrow n + 1$.

3. **Terminate:** Return $v(s)$ for all $s \in S$.

Comments on online TD(γ) for discounted models follow:

1. In contrast to the undiscounted episodic model, the discount factor λ appears explicitly in (10.48) and (10.50).
2. The above algorithm is stated for an accumulating eligibility trace. Because in (10.50) the eligibility trace is damped by the factor $\lambda\gamma$, the impact on previous terms is less than in the episodic model. Thus, distinguishing between the accumulating trace and the replacement trace should have less effect on estimates than in an episodic model. To use a replacement trace, replace step 2(a) by $e(s) \leftarrow 1$. The example below investigates the impact of trace type on estimate quality.
3. The Markov chain generated in step 2(d) may visit some states infrequently leading to high variability in policy value function estimates. Instead, when data is simulated, a state-based approach may be used. If that is the case, the first part of step 2(g) would be replaced by “Sample s from a discrete distribution on S .” The distribution could be uniform or put more weight on critical states. This approach is sometimes referred to as *random restart*.
4. The algorithm may be sensitive to initial values. Choosing a good initial value such as

$$v_0(s) = (1 - \lambda)^{-1} r_d(s) \quad (10.51)$$

where $r_d(s) = E[r(X_0, d(X_0), X_1) | X_0 = s]$ for each $s \in S$ may enhance convergence. This value of $v_0(s)$ corresponds to starting the system in state s and receiving an identical reward of $r_d(s)$ at each decision epoch over the infinite horizon.

5. Online TD(γ) converges if all states are visited infinitely often and the usual conditions on τ_n apply.
6. Temporal differencing or TD(0) is a special case corresponding to $\gamma = 0$. In this case, the discount rate enters only through the temporal difference (10.48) since (10.50) sets $e(s) = 0$ for all $s \in S$.

An example

The following example applies online $\text{TD}(\gamma)$ to the two-state model in Section 2.5. It estimates the expected infinite horizon discounted reward, $v_\lambda^{d^\infty}(s)$, for the stationary deterministic policy d^∞ with $d(s_1) = a_{1,2}$ and $d(s_2) = a_{2,2}$. Setting $\lambda = 0.9$ and solving $(\mathbf{I} - \lambda \mathbf{P}_d)\mathbf{v} = \mathbf{r}_d$ establishes that $v_\lambda^{d^\infty}(s_1) = 30.147$ and $v_\lambda^{d^\infty}(s_2) = 27.941$. Estimation methods were compared using the RMSE of the final observation averaged over both states.

Experiments consisting of 40 replicates of length $N = 5,000$ with common random number seeds. They compared the following four factors:

1. Trace: accumulating vs. replacement;
2. Learning rate³⁹: $\tau_n = 20/(50 + n)$ vs. $\tau_n = \log(n + 1)/n$;
3. $\text{TD}(\gamma)$: $\gamma = 0, 0.2, 0.4, 0.6, 0.8$, and 1;
4. Simulation method: Trajectory-based vs. state-based (random restart using a uniform distribution).

Output was analyzed graphically and using analysis of variance⁴⁰ (ANOVA). Results include:

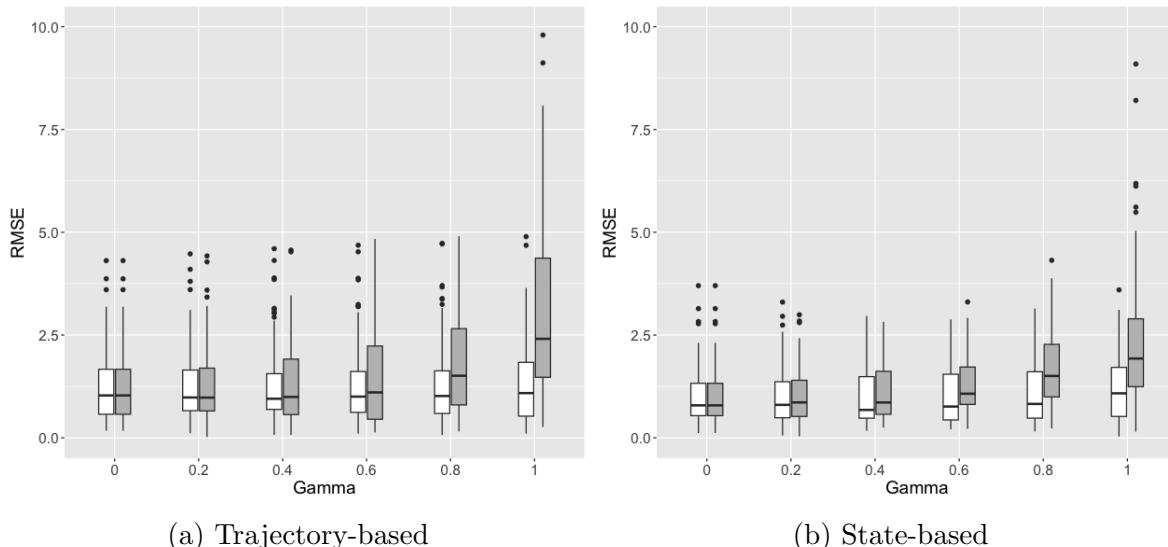


Figure 10.15: Graphical summary of an experiment applying online $\text{TD}(\gamma)$ to a discounted version of the two-state example. It shows variability in RMSE by γ and trace type (accumulating - grey; replacement - white) over 40 replicates of length 5,000.

³⁹These values were chosen based on preliminary experimentation.

⁴⁰*Analysis of variance* is a statistical methodology for investigating the individual and simultaneous effect of discrete factors on the mean value of a response variable.

1. The most accurate estimate of $v_\lambda^{d^\infty}(s)$ over all combinations of factors was the combination of logarithmic learning rate, replacement trace, random restart and $\gamma = 0.4$. For this configuration, the average RMSE was 0.96.
2. ANOVA established that the effect of learning rate on RMSE was statistically insignificant but that simulation method, trace type and γ were highly significant.
3. ANOVA also showed that the interaction between γ and trace type was significant, meaning that the effect of trace type differed depending on γ as shown in Figure 10.15. As expected, γ affected results for the accumulating trace and not the replacement trace. This observation was to be expected because in this two-state example, using the replacement trace, especially in the case of trajectory-based data, frequently reset $e(s)$ to 1 so that the effect of γ was minimal. In contrast, when using the accumulating trace, γ impacted the values of $e(s)$ and hence the estimates of $v_\lambda^{d^\infty}(s)$.
4. For trajectory-based data, Figure 10.15a shows that $\gamma = 0$ resulted in the minimal median and least variable RMSE (1.25) over all configurations. Of course, trace type does not impact TD(0) estimates.
5. For state-based data, the results were more complicated. Figure 10.15b shows that the (median) RMSE was increasing in γ for the accumulating trace. For the replacement trace, the median RMSE was minimal at $\gamma = 0.4$ but the differences with respect to the values associated with other γ values were small.
6. Estimates for trajectory-based data were more variable than those for the random restart data. This is to be expected since, when following the Markov chain corresponding to long trajectory state generation, state s_1 is visited twice as frequently as state s_2 .

The above results suggest that in this small example, the replacement trace is preferable to the accumulating trace, and that TD(0) provides the best estimates in trajectory-based data. If state-based simulation is used, there is a small benefit to choosing $\gamma > 0$. It is left to the reader to compare this approach to results using Monte Carlo estimation.

Of course, the same conclusions need not hold in general. However, the data generation and analysis of this example suggest an approach to follow in other applications.

10.5 Policy evaluation: Infinite horizon average reward models

This section provides a brief overview of issues arising when applying simulation in the context of infinite horizon average reward models. As a prelude to policy optimization, computing only the average reward (or gain) of a policy does not suffice; an estimate of the bias is also required to apply the Bellman equation.

Most research in simulation-based dynamic programming has focused on infinite horizon episodic and discounted models. This is because the average reward criterion is less sensitive to short-term actions; it depends more on what happens in the distant future. *Therefore, while online learning is fundamental in episodic and discounted models, it is less relevant when using the average reward criterion.* This means that to evaluate policies and find optimal policies for average reward models:

1. Large data sets are required for accurate estimation of the average reward and bias.
2. Offline analyses are practical in average reward settings but online methods exist and still can be used.

This section will describe Monte Carlo and TD(0) approaches for estimating the gain and bias of a stationary policy. Generalization to TD(γ) is left to the reader.

Technical preliminaries

The focus of this section will be on policies with constant gain, therefore, it implicitly assumes a recurrent or unichain model⁴¹. Recall from Chapter 7 that for a stationary policy d^∞ , its average reward (or gain) is defined by

$$g^{d^\infty}(s) = \lim_{N \rightarrow \infty} \frac{1}{N} E^{d^\infty} \left[\sum_{n=1}^N r(X_n, Y_n, X_{n+1}) \middle| X_1 = s \right] \quad (10.52)$$

and its bias by

$$h^{d^\infty}(s) = E^{d^\infty} \left[\sum_{n=1}^{\infty} (r(X_n, Y_n, X_{n+1}) - g^{d^\infty}(X_n)) \middle| X_1 = s \right]. \quad (10.53)$$

Moreover, Chapter 7 establishes that in unichain models, the gain and bias of a stationary policy d^∞ satisfy the system of equations

$$h(s) = \sum_{j \in S} p(j|s, d(s)) (r(s, d(s), j) + h(j)) - g, \quad \forall s \in S. \quad (10.54)$$

⁴¹Or a regular or unichain policy in a multi-chain model.

Since there are $|S|$ equations in $|S| + 1$ unknowns, an extra condition is required to guarantee a unique solution. Solving (10.54) subject to $h(s_0) = 0$ for some s_0 provides the *relative values*, denoted $h_{\text{rel}}^{d^\infty}(s)$, which suffice for optimization using the Bellman equation.

When setting $h(s_0) = 0$, the equation corresponding to s_0 becomes

$$g = \sum_{j \in S} p(j|s_0, d(s_0)) (r(s_0, d(s_0), j) + h(j)). \quad (10.55)$$

This equation will prove useful when motivating TD(0) below.

10.5.1 Monte Carlo methods

Monte Carlo methods are based on the representations for the gain and bias in (10.52) and (10.53). For the same reasons as in the discounted case, only starting-state Monte Carlo provides estimates with the same degree of accuracy for all states.

Algorithm 10.7. Monte Carlo policy evaluation for an average reward model

1. **Initialize:**
 - (a) Specify $d \in D^{\text{MR}}$.
 - (b) Specify the number of replicates K_g and K_h , and set $k_g \leftarrow 1$ and $k_h \leftarrow 1$.
 - (c) Specify the truncation levels M_g and M_h , and set $m_g \leftarrow 1$ and $m_h \leftarrow 1$.
 - (d) Specify $\bar{s} \in S$.
 - (e) Create empty lists VALUES_g and VALUES_h .
2. **Estimate the gain:** While $k_g \leq K_g$:
 - (a) $m_g \leftarrow 1$ and $u \leftarrow 0$.
 - (b) Sample $s \in S$ from a random distribution.
 - (c) **Generate a replicate:** While $m_g \leq M_g$:
 - i. Sample a from $w_d(\cdot|s)$.
 - ii. Simulate (s', r) or sample s' from $p(\cdot|s, a)$ and set $r \leftarrow r(s, a, s')$.
 - iii. $u \leftarrow u + r$.
 - iv. $s \leftarrow s'$.
 - v. $m_g \leftarrow m_g + 1$.
 - (d) **Update:**

- i. Append u/M_g to VALUES_g .
- ii. $k_g \leftarrow k_g + 1$.
3. Set $g = \text{mean}(\text{VALUES}_g)$.
4. **Estimate the bias:** While $k_h \leq K_h$:
 - (a) $m_h \leftarrow 1$ and $u \leftarrow 0$.
 - (b) $s \leftarrow \bar{s}$.
 - (c) **Generate a replicate:** While $m_h \leq M_h$:
 - i. Sample a from $w_d(\cdot|s)$.
 - ii. Simulate (s', r) or sample s' from $p(\cdot|s, a)$ and set $r \leftarrow r(s, a, s')$.
 - iii. $u \leftarrow u + (r - g)$.
 - iv. $s \leftarrow s'$.
 - v. $m_h \leftarrow m_h + 1$.
 - (d) **Update:**
 - i. Append u to VALUES_h .
 - ii. $k_h \leftarrow k_h + 1$.
5. Set $h(\bar{s}) = \text{mean}(\text{VALUES}_h)$.
6. **Terminate:** Return g and $h(\bar{s})$.

Some comments follow:

1. This rather lengthy algorithmic statement provides estimates of g and $h(\bar{s})$ using truncation with possibly different truncation levels and number of replicates for each quantity estimated.
2. Replicates to estimate g can start from the same state or be randomly chosen.
3. It returns estimates of the bias for a single state. To obtain estimates of the bias for all states requires repeating step 4 for all $\bar{s} \in S$.
4. Note that an estimate of the gain is required before estimating the bias.
5. Of course, Monte Carlo is an offline algorithm. It applies to both model-free and model-based settings.

Example 10.4. Average reward Monte Carlo policy evaluation

This example evaluates the gain and bias in the two-state model for the station-

ary policy d^∞ with $d(s_1) = a_{1,2}$ and $d(s_2) = a_{2,2}$. Solving the evaluation equations (10.54) with $h(s_1) = 0$ gives $g^{d^\infty} = 2.857$ and $h_{\text{rel}}^{d^\infty}(s_2) = -2.143$. Alternatively, solving (10.54) subject to $\mathbf{P}_d^* \mathbf{h} = \mathbf{0}$ or equivalently using the matrix representation in Theorem 7.3, establishes that $h^{d^\infty}(s_1) = 1.531$ and $h^{d^\infty}(s_2) = -0.612$.

The gain, bias and relative values were estimated using Algorithm 10.7. Preliminary calculations showed that estimates, especially of the relative value, were extremely variable. Consequently a large number of replicates was required. The following values were used to obtain the estimates in Figure 10.4: $K_g = 850$, $M_g = 850$, $K_h = 10,000$ and $M_h = 500$. Results described below were based on 50 instances using the above configuration with different random number seeds for each instance.

Estimates of the gain had mean 2.850 and standard deviation 0.038 over all instances. Estimates of the bias $h^{d^\infty}(s)$ were extremely variable and unreliable, however estimates of relative values based on $h^{d^\infty}(s_2) - h^{d^\infty}(s_1)$ were more accurate with mean -2.130 and standard deviation 0.204 over all instances. Figure 10.4 displays the variability of the estimates of the gain and relative value across instances.

Because the estimate of g^{d^∞} in step 3 of the algorithm impacts the estimate of the bias in step 4(c)iii, one might hypothesize that the estimates of the gain and relative values were correlated. Surprisingly, this was not the case.

One can conclude that Monte Carlo methods require very large samples to estimate relative values. So, they may not be appropriate for algorithmic optimization.

10.5.2 Temporal differencing (TD(0))

This section provides a temporal differencing method to evaluate the average reward or gain, g^{d^∞} , and relative values $h_{\text{rel}}^{d^\infty}(s)$ of a stationary policy d^∞ . For simplicity, they are expressed as g and $h(s)$.

To motivate temporal differencing, express (10.54) in expectation form as

$$h(s) = E^{d^\infty} \left[r(X, Y, X') + h(X') - g \mid X = s \right] \quad (10.56)$$

and (10.55) as

$$g = E^{d^\infty} \left[r(X, Y, X') + h(X') \mid X = s_0 \right]. \quad (10.57)$$

Applying stochastic approximation to these two expressions and replacing g and $h(s)$ by the most recent estimates leads to the following algorithm. Note that it applies to trajectory-based and state-based data.

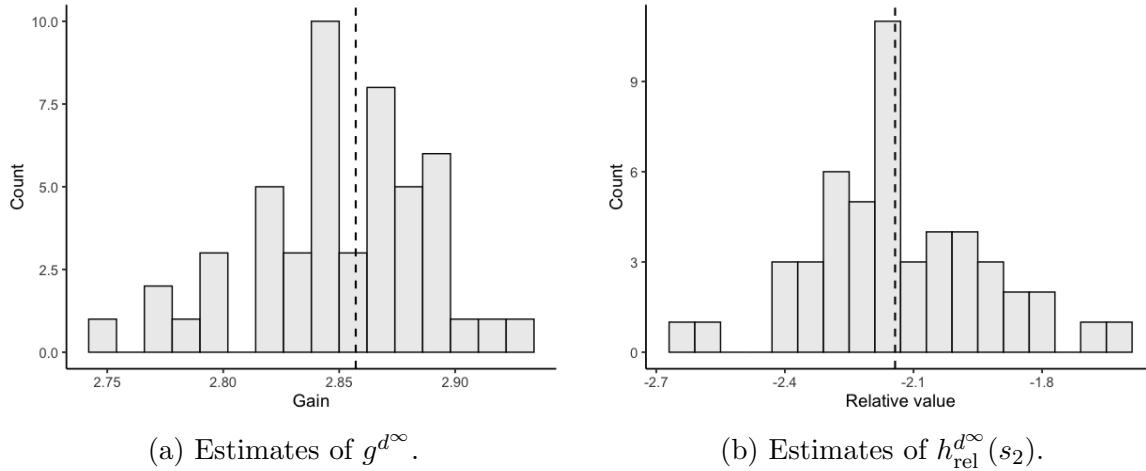


Figure 10.16: Histograms showing variability of Monte Carlo estimates of gain and relative value over 50 instances in Example 10.4. Vertical dashed lines give the true value of the quantities being estimated. Note that the axis scale differs for the two estimates.

Algorithm 10.8. Temporal differencing for policy evaluation for an average reward model

1. **Initialize:**

- (a) Specify $d \in D^{\text{MR}}$ and a distinguished state s_0 .
- (b) Specify sequences τ_n and β_n , $n = 1, 2, \dots$
- (c) Set $g \leftarrow 0$, $h(s) \leftarrow 0$ and $\text{count}(s) \leftarrow 0$ for all $s \in S$.
- (d) Specify $s \in S$.
- (e) Specify the number of iterations N and set $n \leftarrow 1$.

2. **Iterate:** While $n \leq N$:

- (a) Sample a from $w_d(\cdot|s)$.
- (b) Simulate (s', r) or sample s' from $p(\cdot|s, a)$ and set $r \leftarrow r(s, a, s')$.
- (c) $\text{count}(s) \leftarrow \text{count}(s) + 1$.
- (d) If $s \neq s_0$,

$$h(s) \leftarrow h(s) + \tau_{\text{count}(s)} (r(s, d(s), s') - g + h(s') - h(s)) \quad (10.58)$$

and if $s = s_0$,

$$g \leftarrow g + \beta_{\text{count}(s_0)} (r(s_0, d(s_0), s') + h(s') - g). \quad (10.59)$$

- (e) $n \leftarrow n + 1$.
- (f) $s \leftarrow s'$ (trajectory-based) or sample s from a specified distribution on S (state-based).
3. **Terminate:** Return g , and $h(s)$ for all $s \in S$.

Some comments follow:

1. The above algorithm uses the relative evaluation equation with $h(s_0) = 0$ in step 2(d). It is based on (10.57). Consequently, the algorithm generates relative values.
2. A variant of the algorithm⁴² replaces equation (10.59) with

$$g \leftarrow g + \beta_n(r(s, d(s), s') - g) \quad (10.60)$$

at **every** iteration. Thus, the index for β_n is the iteration number. Note that $\beta_n = n^{-1}$ is equivalent to recursive estimation of the mean of g based on the observed rewards. This is equivalent to online Monte Carlo estimation of the average reward.

3. The algorithm allows for different learning rates in (10.58) and (10.59).
4. Note that in both variants of the algorithm, updates are asynchronous; when using the algorithm with (10.59), g is only updated whenever s_0 is visited and $h(s)$ is only updated when visiting state $s \neq s_0$. When using the algorithm with (10.60), g is updated at every iteration.
5. A TD(γ) variant of this algorithm using update equation (10.60) converges⁴³ with probability one under the assumption that the Markov chain corresponding to d^∞ is irreducible and aperiodic, and the learning rates satisfy the Robbins-Monro conditions and are linearly related.

The following example investigates the two variants of the algorithm with different learning rates using the frequently analyzed two-state model.

Example 10.5. Temporal differencing in an average reward model.

This example applies the above algorithm and its variant to analyze the policy d^∞ where $d(s_1) = a_{1,2}$ and $d(s_2) = a_{2,2}$. The gain and relative values are given in Example 10.4.

It compares the quality of estimates based on Algorithm 10.8 and its variant

⁴²This variant has been proposed by Tsitsiklis and Roy [1999].

⁴³Tsitsiklis and Roy [1999] established this result as well for models with function approximation.

that replaces (10.59) with (10.60) using 50 replicates of $N = 25,000$ iterates. After some preliminary calculations, two values for each learning rate were chosen for further investigation: $\tau_n^1 = n^{-1} \log(n + 1)$, $\tau_n^2 = 150/(300 + n)$ and $\beta_n^1 = n^{-1} \log(n + 1)$, $\beta_n^2 = n^{-1}$. Thus, for each random number seed, eight instances were compared: each instance corresponded to a different pair of learning rate and update equation for g . Only the trajectory-based version of the algorithm was investigated.

Estimates are compared on the basis of the difference between the final estimates g and $h(s_2)$ and the true values g^{d^∞} and $h_{\text{rel}}^{d^\infty}(s_2)$. Results are displayed as boxplots in Figure 10.17. The figure shows that when estimating the gain, estimates based on (10.60) were more accurate for all learning rate choices. Clearly, using $\beta_n^2 = n^{-1}$ provided the least variable estimates of the gain.

However, the results for estimating the relative value differed. Using the learning rate τ_n^2 provided less variable estimates than using τ_n^1 . Moreover estimates based on the two specifications for the gain differed slightly when using τ_n^2 . Note also that the combination of learning rates (τ_n^2, β_n^1) produced the most accurate estimates of the bias.

Overall the estimates of the gain were more accurate than those of the relative value. This is important because the relative value update (10.58) uses the estimate of the gain explicitly. However, accurate estimates of the relative value are required when seeking optimal policies so that precision of the relative values should guide algorithmic and learning rate choice.

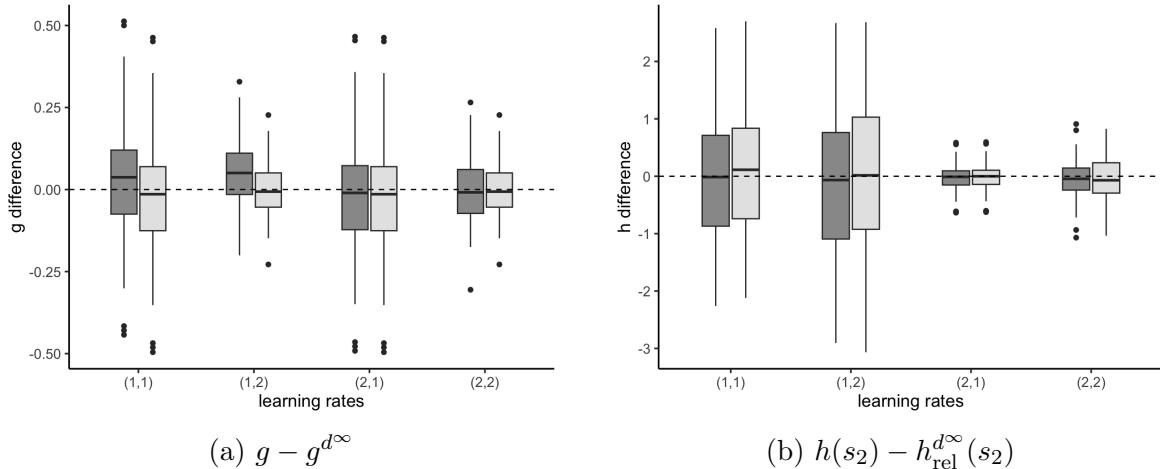


Figure 10.17: Boxplots comparing accuracy of estimates of the gain and relative values for the eight instances in Example 10.5. Learning rates τ_n^i and β_n^j are represented as (i, j) . Dark fill corresponds to g update equation (10.59) and light fill to (10.60). Note that the vertical scale differs in the two plots.

10.6 Policy optimization

This section describes simulation-based methods for finding effective policies. The methods described here either optimize step-by-step, in a similar fashion to value iteration, or combine one of Monte Carlo or temporal differencing with some form of policy improvement. Most are based on state-action value functions, as opposed to value functions. One such approach, Q-learning, represents one of the most significant advances in the development of reinforcement learning methods.

10.6.1 Two perspectives

As noted earlier, simulation-based optimization can be viewed from two perspectives:

1. That of a decision maker or agent interacting with its environment in order to *learn* a good policy through experimentation. Often, the agent may also seek “earning while learning”, that is to maximize accumulated rewards during the learning phase.
2. That of an external controller (or analyst) using simulation or real-world experimentation as a tool to learn good policies to be used in subsequent applications.

Of course these are not mutually exclusive; policies identified during learning may be subsequently implemented in the future⁴⁴.

10.6.2 Model selection

To assess the quality of estimates of the value of a fixed policy, standard *goodness-of-fit* metrics (Section 10.11.1) can be used. When a model is available, these estimates can be compared to the exact policy value function obtained by solving appropriate evaluation equations. Alternatively, comparisons can be based on results of real-time implementations or simulation output.

Since the optimization methods in this chapter are based on state-action value functions, similar RMSE and graphical approaches can be used to assess how well a method estimates the optimal state-action value function. However these should *not* be the primary measure of the quality of a method. Such approaches are limited by two considerations:

⁴⁴In the reinforcement learning literature this is referred to as *planning*.

1. The objective of optimization is to identify high-quality policies. State-action value functions are a construct to do so.
2. The better the method, the more quickly it will identify effective greedy policies. As a result, many state-action pairs will be visited infrequently resulting in inaccurate estimates of state-action value functions for some state-action pairs.

The following *policy-based* considerations should guide method selection:

1. **Policy agreement:** In experiments with known optimal policies, the agreement between these and ones identified by a method can be assessed graphically or through measures of policy closeness.
2. **Long-run policy quality:** The long-run performance of the policies can be compared on the basis of the specified optimality criterion. In a model-based environment this can be based on comparing the exact value of the identified policy computed by solving policy evaluation equations to the optimal value function obtained through algorithms such as policy iteration. In a model-free environment, these can be assessed against simulations using good heuristics.
3. **Policy variability:** Methods that produce similar policies across multiple replications are preferable.
4. **Accumulated reward:** In learning environments, the agent may seek to maximize the accumulated reward during the learning phase of an experiment. Methods that achieve higher cumulative rewards during training are preferable.
5. **Data efficiency:** Methods that require less data to obtain high quality policies are preferable.
6. **Robustness:** Methods that identify high quality policies across different applications and parameter settings are preferable.

Definitive statements require appropriate metrics and carefully designed experiments spanning diverse environments⁴⁵.

10.6.3 Preliminaries

Given an estimate of the value function $v(\cdot)$, iterative optimization algorithms in Chapters 5–7 are based on evaluating expressions of the form

$$\max_{a \in A_s} \left(\sum_{j \in S} p(j|s, a)(r(s, a, j) + \lambda v(j)) \right),$$

⁴⁵Patterson et al. [2024] provides an in-depth discussion of issues to consider when evaluating a reinforcement learning method.

where $0 < \lambda \leq 1$ and choosing actions *greedily*, that is, those in

$$\arg \max_{a \in A_s} \left(\sum_{j \in S} p(j|s, a)(r(s, a, j) + \lambda v(j)) \right).$$

Actions chosen in this way are referred to as *greedy*⁴⁶ actions. Instead of focusing on value functions, the methods in this section focus on state-action value functions

$$q(s, a) := \sum_{j \in S} p(j|s, a)(r(s, a, j) + \lambda v(j))$$

with $\lambda = 1$ in episodic and average reward⁴⁷ models and $0 \leq \lambda < 1$ in discounted models. In particular, they seek to find the optimal state-action value function

$$q^*(s, a) := \sum_{j \in S} p(j|s, a)(r(s, a, j) + \lambda v^*(j)), \quad (10.61)$$

where $v^*(\cdot)$ is the optimal value function for the corresponding MDP. If this quantity were available, optimal policies can be found by selecting actions greedily using $q^*(s, a)$.

The elegance of these methods is that they regard $q(s, a)$ as the primitive. Implementation does not require knowledge of the underlying model. All that is needed are:

1. methods to generate actions in states,
2. methods to generate rewards and transitions to new states, and
3. methods to recursively update state-action value functions.

Recall that $q^*(s, a)$ solves the Bellman equation

$$q(s, a) = \sum_{j \in S} p(j|s, a)(r(s, a, j) + \lambda \max_{a' \in A_j} q(j, a')) \quad (10.62)$$

with $\lambda = 1$ in episodic models and $0 \leq \lambda < 1$ in discounted models. The solution is unique in transient and stochastic shortest path episodic models and discounted models (Theorems 6.6 and 5.7).

⁴⁶They are also referred to as **v-greedy** actions below.

⁴⁷In average reward models, the bias or relative value functions replace $v(s)$.

10.6.4 Selecting actions: Exploration vs. exploitation

To find optimal policies using simulation or real-time experimentation, **all** state-action pairs must be evaluated sufficiently often to obtain accurate estimates of state-action value functions $q(s, a)$. Doing this is referred to as *exploration* to distinguish it from *exploitation*, which only implements greedy actions.

When seeking a good policy, it is desirable to:

1. begin by exploring broadly and frequently,
2. then narrow exploration by searching among actions with large $q(s, a)$ values, and
3. finally, implement greedy actions frequently.

Two approaches to achieve these objectives are ϵ -greedy sampling and softmax sampling. Both use randomized decision rules to sample actions.

ϵ -greedy sampling

This approach selects⁴⁸ an action a' in state $s \in S$ according to

$$a' = \begin{cases} a^* \in \arg \max_{a \in A_s} q(s, a) & \text{with probability } 1 - \epsilon \\ a \in A_s \setminus \{a^*\} & \text{with probability } \epsilon/(|A_s| - 1). \end{cases} \quad (10.63)$$

Alternatively,

$$a' = \begin{cases} a^* \in \arg \max_{a \in A_s} q(s, a) & \text{with probability } 1 - \epsilon + \epsilon/|A_s| \\ a \in A_s \setminus \{a^*\} & \text{with probability } \epsilon/|A_s|. \end{cases}$$

When A_s contains only a few elements, such as in the examples below, the former specification enables more exploration. To see this, suppose A_s contains three elements and $\epsilon = 0.3$. The first specification chooses the greedy action with probability 0.7 and each other action with probability 0.15, while the second specification chooses the greedy action with probability 0.8 and each other action with probability 0.1. In models with large action sets, such a distinction is irrelevant.

From another perspective, ϵ -greedy action selection corresponds to implementing the *randomized* decision rule $d(s)$ defined by

$$w_d(a|s) := \begin{cases} 1 - \epsilon & a \in \arg \max_{a' \in A_s} q(s, a') \\ \epsilon/(|A_s| - 1) & a \neq \arg \max_{a' \in A_s} q(s, a'). \end{cases} \quad (10.64)$$

⁴⁸This specification needs some modification if the arg max is not unique. For example if there are two elements attaining the arg max select each with probability $(1 - \epsilon)/2$ when using (10.63).

Softmax (Boltzmann) sampling

Another approach to exploration is to choose action a in state s with probability determined by the *softmax* function:

$$\frac{e^{\eta q(s,a)}}{\sum_{a' \in A_s} e^{\eta q(s,a')}}. \quad (10.65)$$

The quantity η is a parameter that affects action choice probabilities and also helps avoid numerical instability⁴⁹. Sometimes $1/\eta$ is referred to as the *temperature parameter*. When η is small, the differences in $q(s,a)$ have less effect on action choice probabilities than when η is large. For small η , softmax sampling selects actions with nearly equal probabilities. Conversely, when η is large, the softmax function makes it more likely to select actions with large $q(s,a)$ values.

As above, softmax sampling may be viewed as implementing a *randomized* decision rule $d(s)$ defined by

$$w_d(a|s) := \frac{e^{\eta q(s,a)}}{\sum_{a' \in A_s} e^{\eta q(s,a')}}. \quad (10.66)$$

Note that viewing softmax sampling from the perspective of a randomized decision rule is fundamental to the policy gradient and actor-critic methods in Chapter 11.

Discussion

Both methods depend on parameters that trade-off exploration and exploitation. Clearly these parameters should be varied through the learning (or computational) process to impose early exploration and late exploitation. This means that ϵ should decrease and η increase with respect to either the iteration number or the count of the number of visits to a state. Therefore algorithms are usually specified in terms of sequences of ϵ_n or η_n , $n = 1, 2, \dots$

The main difference between these methods is that for each $s \in S$, softmax sampling assigns probabilities to actions based on the relative values of $q(s,a)$, ensuring that all actions have a non-zero probability of being selected but favoring those with the greatest values of $q(s,a)$. In contrast, ϵ -greedy sampling selects greedy actions with probability $1 - \epsilon$ and does not distinguish between non-greedy actions; all have a small and equal probability of being selected regardless of their $q(s,a)$ values.

The use of either of these methods with appropriate parameter choices ensures that all state-action pairs are visited infinitely often, which is usually sufficient for theoretical convergence.

⁴⁹We have frequently encountered divergence of parameter estimates in our calculations.

10.7 Q-learning and SARSA

How would one go about using state-action value functions to learn good policies? An obvious iterative approach would be to start with a guess for $q(s, a)$, select a state s , use softmax or ϵ -greedy sampling to select an action a , implement it and observe a reward r and a transition to a subsequent state s' .

Q-learning and SARSA follow this general approach, but differ in how they use this information to update $q(s, a)$. Each may use exploration to generate an action a' in state s' , but Q-learning updates $q(s, a)$ according to

$$q(s, a) \leftarrow q(s, a) + \tau \left(r + \lambda \max_{a' \in A_{s'}} q(s', a') - q(s, a) \right) \quad (10.67)$$

while SARSA updates $q(s, a)$ according to

$$q(s, a) \leftarrow q(s, a) + \tau(r + \lambda q(s', a') - q(s, a)), \quad (10.68)$$

where a' denotes an action selected by exploration in the subsequent state. The acronym SARSA stands for “state-action-reward-state-action”, corresponding to the sequence of objects (s, a, r, s', a') involved in the update in (10.68)⁵⁰.

Another way of thinking of this is through the concept of a *learning* or *behavioral* policy.

Definition 10.2. A *learning policy* or *behavioral policy* is the policy used to generate rewards in a simulation or real-world implementation.

Suppose both Q-learning and SARSA use the same behavioral policy to generate action a' . SARSA corresponds to an on-policy update of $q(s, a)$ in agreement with the behavioral policy, while the off-policy update using Q-learning may differ⁵¹ as shown in Figure 10.18.

More formally, Q-learning updates estimates of state-action value functions using a temporal difference relationship of the form

$$q(s, a) \leftarrow q(s, a) + \tau \left(r(s, a, s') + \lambda \max_{a' \in A_s} q(s', a') - q(s, a) \right) \quad (10.69)$$

where $0 \leq \tau \leq 1$. This recursion is based on using a variant⁵² of gradient descent to

⁵⁰Note that when the reward depends on the subsequent state, the unappealing acronym SASRA would be more appropriate.

⁵¹They will agree when $a' \in \arg \max_{a \in A_{s'}} q(s', a)$.

⁵²This variant, based on a construct referred to as a *semi-gradient*, is discussed in Section 11.2.2 in the next chapter.

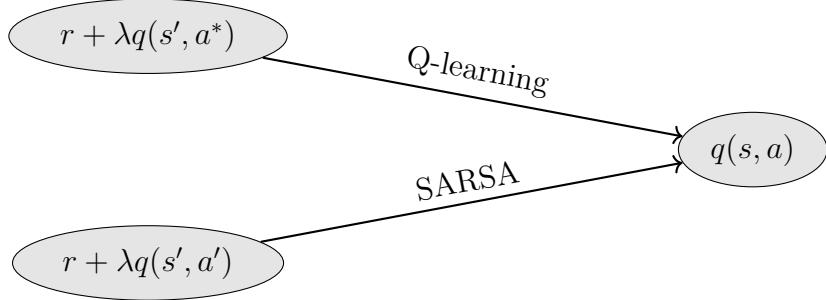


Figure 10.18: Comparison of quantity used to update $q(s, a)$ using SARSA and Q-learning. These updates differ when the action chosen by the greedy policy a^* differs from the action chosen by the learning policy a' .

minimize the expected squared error between the two sides of the expression

$$q(s, a) = E \left[r(X, Y, X') + \lambda \max_{a' \in A_{X'}} q(X', a') \mid X = s, Y = a \right], \quad (10.70)$$

which is the Bellman equation (10.62) written in expectation form. Note that the expectation is with respect to the transition probability $p(j|s, a)$ that specifies the distribution of X' . Note that in episodic and average reward models, $\lambda = 1$. In the latter case, (10.69) would also contain a $-g$ term to correspond to the appropriate recursion for the gain and bias.

SARSA

SARSA updates estimates of state-action value functions according to:

$$q(s, a) \leftarrow q(s, a) + \tau(r(s, a, s') + \lambda q(s', a') - q(s, a)), \quad (10.71)$$

where a' in state s' is chosen based on the randomized policy that was used to choose a in state s . The justification for SARSA is more subtle than Q-learning because after generating s' from $p(\cdot|s, a)$, it requires a distribution for choosing a' at the subsequent decision epoch.

Given a randomized decision rule d , SARSA is based on replacing (10.70) by

$$q(s, a) = E^d \left[r(X, Y, X') + \lambda q(X', Y') \mid X = s, Y = a \right], \quad (10.72)$$

where X' has distribution $p(j|s, a)$ and Y' has conditional distribution $w_d(\cdot|X')$ corresponding to the exploration method.

Thus, in essence, SARSA is applying a variant of gradient descent to estimate the state-action value function corresponding to the behavioral policy d^∞ . This would

be appropriate if the objective was to estimate $q^{d^\infty}(s, a)$, but the objective is to find the **optimal** state-action value function. Therefore, the decision d must evolve over time so that it approaches the decision rule corresponding to an optimal policy. This is typically achieved using ϵ -greedy or softmax action selection with parameters that ensure that eventually only greedy actions are selected. This requirement is described in more detail following the description of SARSA in algorithmic form.

10.7.1 Episodic models

Q-learning and SARSA algorithms for episodic models are presented, discussed and compared below. Recall that in an episodic model, an episode terminates when a state in Δ is reached.

Q-learning

The following online algorithm can be implemented in a model-free or model-based environment.

Algorithm 10.9. Q-learning for an episodic model

1. Initialize:

- (a) Set $q(s, a) \leftarrow 0$ and $\text{count}(s, a) \leftarrow 0$ for each $a \in A_s$ and $s \in S$.
- (b) Specify the learning rate $\tau_n, n = 1, 2, \dots$
- (c) Specify $\bar{s} \in S$ and $\bar{a} \in A_{\bar{s}}$.
- (d) Specify the number of episodes K .
- (e) Specify a sequence $\epsilon_n, n = 1, 2, \dots$ (or η_n for softmax sampling).
- (f) $k \leftarrow 1$.

2. Iterate: While $k \leq K$:

- (a) $s \leftarrow \bar{s}$ and $a \leftarrow \bar{a}$.
- (b) **Generate episode:** While $s \notin \Delta$:
 - i. $\text{count}(s, a) \leftarrow \text{count}(s, a) + 1$.
 - ii. Simulate (s', r) or sample s' from $p(\cdot|s, a)$ and set $r \leftarrow r(s, a, s')$.
 - iii. **Update** $q(s, a)$:

$$q(s, a) \leftarrow q(s, a) + \tau_{\text{count}(s, a)} \left(r + \max_{a' \in A_{s'}} q(s', a') - q(s, a) \right). \quad (10.73)$$

- iv. Choose $a \in A_{s'}$ using ϵ -greedy (or softmax) sampling.
 - v. $s \leftarrow s'$.
 - (c) $k \leftarrow k + 1$.
3. **Terminate:** Return $q(s, a)$ for all $a \in A_s$ and $s \in S$.

Some comments follow:

1. Note that in (10.73), r depends only on the *current* action a while $\max_{a' \in A_{s'}} q(s', a')$ accounts for action choice at the *next* decision epoch. The consequence of this is that updates of $q(s, a)$ use the value associated with a greedy action although the action generated by the learning policy in step 2(b)iv to generate rewards and transition probabilities at the next decision epoch need not be the greedy action. Because the learning policy may differ from the policy used to compute the temporal difference in (10.73), Q-learning is said to be an *off-policy algorithm*.
2. The index for ϵ or η is best set to $\sum_{a \in A_s} \text{count}(s, a)$, corresponding to the number of times the state was previously visited.
3. The nature of the application determines whether it is most prudent to start each episode in the same state (as the algorithm is currently written) or in a random state.
4. Using a similar approach to $\text{TD}(\gamma)$, past state-action values can be updated at each iteration.
5. Under the assumption that each state-action pair is visited infinitely often and the learning rate satisfies the Robbins-Monro conditions, the state-action value function estimates converge with probability one to the optimal state-action value function and hence can be used to identify an optimal policy in the limit.

SARSA

SARSA provides the following *on-policy* variant to Q-learning.

Algorithm 10.10. SARSA for an episodic model

1. **Initialize:** Same as in Algorithm 10.9.
2. **Iterate:** While $k \leq K$:
 - (a) $s \leftarrow \bar{s}$ and $a \leftarrow \bar{a}$.
 - (b) **Generate episode:** While $s \notin \Delta$:
 - i. $\text{count}(s, a) \leftarrow \text{count}(s, a) + 1$.

- ii. Simulate (s', r) or sample s' from $p(\cdot|s, a)$ and set $r \leftarrow r(s, a, s')$.
- iii. Choose $a' \in A_{s'}$ using ϵ -greedy (or softmax) sampling.
- iv. **Update** $q(s, a)$:
$$q(s, a) \leftarrow q(s, a) + \tau_{\text{count}(s, a)} (r + q(s', a') - q(s, a)). \quad (10.74)$$
- v. $s \leftarrow s', a \leftarrow a'$.
- (c) $k \leftarrow k + 1$.
- 3. **Terminate:** Return $q(s, a)$ for all $a \in A_s$ and $s \in S$.

Note that both SARSA and Q-learning use ϵ -greedy or softmax action selection but SARSA uses the action to *both* update the state-action value function and implement at the subsequent iteration. On the other hand, Q-learning uses ϵ -greedy or softmax action selection *only* to select the next action. Thus, the updates of $q(s, a)$ using SARSA correspond to the “implemented” action or the behavioral policy, while the updates using Q-learning correspond to the greedy action, which may differ from the implemented action (see Figure 10.18).

Without further conditions, SARSA *may converge to a sub-optimal policy*. For example, when the exploration rate does not decrease to zero, the state-action value functions may correspond to a non-optimal randomized policy. In order for SARSA to converge to the optimal state-action value function and identify an optimal policy requires, the learning policy must

1. sample each state-action pair infinitely often, and
2. select greedy actions in the limit with probability one.

Such learning policies are referred to as *greedy in the limit with infinite exploration*⁵³. Thus, if the learning policy is greedy in the limit with infinite exploration, SARSA will converge to the optimal state-action value function under the Robbins-Monro step-size conditions.

The on-policy versus off-policy dichotomy

Why should one care about this distinction? Analysis of a **deterministic** version of the Gridworld example in Figure 10.6 provides some insight.

Example 10.6. Distinguishing on-policy and off-policy updates

Suppose in a deterministic model the coffee-delivering robot occupies cell 11

⁵³Singh et al. [2000] introduces this concept and provides technical conditions that ensure convergence of SARSA.

(see Figure 10.6), plans to move upward (denoted action 1) to cell 8 and the learning policy based on ϵ -greedy or softmax sampling in cell 8 selects the action that moves it to the left (denoted action 2). As a result, the robot falls down the stairs and incurs a penalty of 200. If this happens, the SARSA and Q-learning updates of $q(11, 1)$ differ.

The SARSA update is

$$q(11, 1) \leftarrow q(11, 1) + \tau(-1 + q(8, 2) - q(11, 1))$$

while the Q-learning update is

$$q(11, 1) \leftarrow q(11, 1) + \tau\left(-1 + \max_{a' \in A_8} q(8, a') - q(11, 1)\right).$$

Choosing $q(s, a)$ equal to the optimal value (in the deterministic problem) sets $q(11, 1) = 46$, $q(8, 2) = -201$ and $q(8, 1) = 47$. Thus, the SARSA update becomes

$$q(11, 1) \leftarrow 46 + \tau(-1 - 201 - 46) = 46 - 248\tau$$

while the Q-learning update becomes

$$q(11, 1) \leftarrow 46 + \tau(-1 + 47 - 46) = 46.$$

Thus in future episodes, under SARSA, it would be less likely for the robot to choose the action ‘move up’ in cell 11 and instead move to the right to cell 12, while under Q-learning the robot would move to cell 8 risking the likelihood of falling down the stairs.

This example shows that because of the presence of action sampling, the SARSA guided robot will take more conservative actions than the Q-learning guided robot. However, Q-learning preserves the optimal state-action value function while SARSA does not.

What this example shows is that when using exploration with either ϵ -greedy or softmax sampling:

Use Q-learning if your objective is to find an optimal policy for future use, while SARSA might be preferable when reward acquisition during the learning phase is of concern.

Computational examples

This section compares Q-learning and SARSA using two versions of the Gridworld model.

Example 10.7. Model-free deterministic Gridworld

In this version of the Gridworld model, the robot can select any of the actions “up”, “right”, and “left” in any cell. Action choice results in a deterministic transition in the intended direction when possible. Otherwise the robot remains in the current cell. For example, if the robot is in cell 13 and tries to move left, it will remain in cell 13. Each time the robot chooses an action it incurs a cost of 1, even if it cannot move.

This example applies Q-learning and SARSA with $q(s, a)$ initialized at 0 and with initial state designated as cell 13. In contrast to earlier Gridworld examples, an episode ends when the robot reaches cell 1. Thus, an episode may consist of instances when the robot falls down the stairs (cell 7) and incurs a penalty. In such instances, the robot returns to cell 13, refills the coffee cup and then attempts to deliver the coffee.

Exploration introduces randomness. The results below apply softmax sampling with $\eta_n = 0.04$ and updates $q(s, a)$ using a learning rate of $\tau_n = 0.8n^{-0.75}$ where n denotes the number of visits to pair (s, a) . These parameter choices were based on considerable experimentation. Both Q-learning and SARSA used $K = 500$ episodes and used the same random number seed for each episode.

Using Q-learning, the estimated state-action value function obtained from a typical replicate of 500 episodes appears in Table 10.4. The entries labeled “n/a” correspond to the termination cells 1 and 7, where no action choice is available. Starting from cell 13, the greedy policy’s path, represented by the bold entries in the table, is $13 \rightarrow 10 \rightarrow 11 \rightarrow 8 \rightarrow 5 \rightarrow 2 \rightarrow 1$. Note that $q(13, \text{“up”}) = 42.89$, while the optimal deterministic policy total reward starting in 13 and following the shortest path generates a reward of 44. The discrepancy is a result of exploration and the small number of episodes. When K is increased to 5,000, $q(13, \text{“up”}) = 44$ and there also were several optimal policies.

SARSA with softmax sampling and $K = 500$ identified the same optimal policy but the $q(s, a)$ values were considerably smaller than those obtained using Q-learning. Moreover “optimal” action choice differed in cell 15. In cell 15 SARSA chose the more conservative action “up” and Q-learning chose “left”.

Figure 10.19 shows the sequence of running-average rewards acquired through the 500 episodes using Q-learning and SARSA. Observe that in this instance the running-average reward during learning using SARSA exceeded that of Q-learning. The mean (standard deviation) of the total reward per episode for Q-learning was 31.36 (14.94) and for SARSA was 34.58 (16.92). This pattern persisted across all experiments.

As a result of the above computational results, the robot can abandon exploration, encode the optimal policy and obtain the maximum reward.

The results in Example 10.7 are quite remarkable. *With no intervention, and no knowledge of the world, the robot identified the optimal policy and accurately estimated*

state	up	right	left
1	n/a	n/a	n/a
2	48.00	46.99	49.00
3	47.00	47.00	48.00
4	49.00	46.87	47.96
5	47.99	45.81	47.97
6	46.98	45.86	46.89
7	n/a	n/a	n/a
8	46.94	44.41	-160.8
9	45.81	42.25	42.86
10	-160.8	44.52	43.14
11	45.78	42.25	42.86
12	44.22	41.49	44.20
13	42.89	42.33	40.52
14	44.17	37.08	39.95
15	39.40	27.90	41.41

Table 10.4: Table of estimates of $q^*(s, a)$ using Algorithm [10.9] in a single experiment consisting of 500 episodes using Q-learning.

the optimal state-action value function.

The following example considers the Gridworld model with noisy transitions.

Example 10.8. Comparison of Q-learning and SARSA in Gridworld

Using the Gridworld model with $p = 0.8$, this example investigates the effect of learning rate and algorithm (Q-learning versus SARSA) on:

1. The extent of agreement between the greedy policy based on the estimated state-action value function and the optimal policy,
2. the difference between the value function of the greedy policy and that of the optimal policy, and
3. the RMSE of the estimated state-action value function.

Since the model is available it can be solved to find the optimal policy and the optimal value function. If it were not, Monte Carlo policy evaluation could be used.

Preliminary calculations investigating exploration method, learning rate and number of episodes suggested the following choices:

1. **Exploration:** ϵ -greedy sampling with

$$\epsilon_n = \frac{200}{500 + n}, \quad (10.75)$$

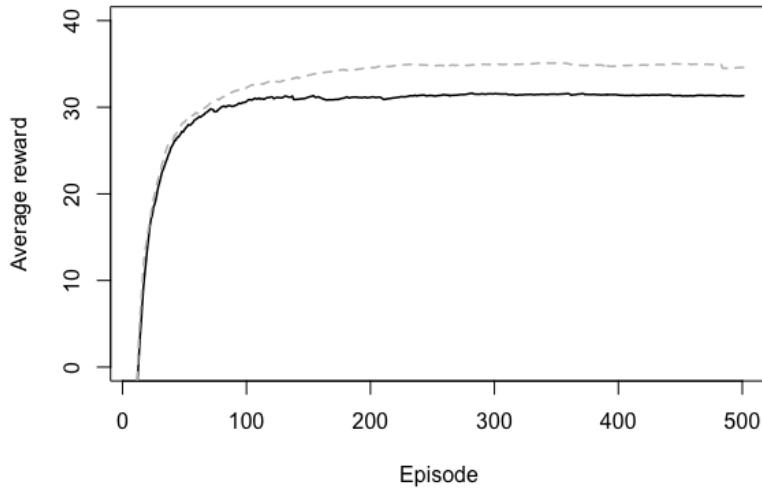


Figure 10.19: Running-average reward using SARSA (dashed line) and Q-learning (solid line) based on a single experiment with 500 episodes.

where n denotes the episode number, produced the most accurate estimates of $q(s, a)$.

2. **Learning rates:**

$$\text{Exp1: } \tau_n = 0.4n^{-0.5}$$

$$\text{Exp2: } \tau_n = 0.8^{-0.75}$$

$$\text{STC: } \tau_n = 0.1/(1 + 10^{-5}n^2)$$

$$\text{Log: } \tau_n = n^{-1} \log(n + 1)$$

where the learning rate index n represented the number of times a state-action pair was previously visited.

3. **Replicates:** 40 starting in cell 13 using common random number seeds for each combination of factors in each replicate.

4. **Number of episodes:** 5,000

Table 10.5 and Figure 10.20 summarize the results. Observe that:

1. **Policy optimality:** The greedy policy derived from the final estimate of $q(s, a)$ agreed with the optimal policy in at most 21 of the 40 replications (SARSA with learning rate Exp2). Averaged over the four learning rates,

the policy identified by SARSA was more likely to be optimal than that identified by Q-learning.

2. **Features of non-optimal policies:** When greedy policies were non-optimal, they differed from the optimal policy in at most 2 states (cells 4 and cell 8) for all but one instance^a. These states were visited infrequently under the optimal policy, which steered the robot towards the right wall before reaching the stairs. This could be remedied by additional sampling of infrequently sampled state-action pairs.
3. **Impact of non-optimal actions:** Table 10.6 shows the effect of incorrect action choice on policy value functions found using the exact methods in Chapter 6. Observe that having one or two non-optimal actions had little effect on the estimate for replications starting in cell 13.
4. **State-action value function estimation:** For all learning rates, Q-learning estimates of $q(s, a)$ had smaller median and mean RMSEs^b than those obtained with SARSA. The effect of learning rate on the accuracy of the estimate of the optimal state-action value function was not consistent for the two methods.
5. **Value vs. policy** The learning rates that most accurately estimated the optimal state-action value function differed from those that were more likely to identify the optimal policy.

The take away from this example is that, in most cases, greedy policies obtained from estimated state-action value functions identified policies with values that differed little from those of the optimal policy. On the other hand, accuracy of the estimates of state-action value functions varied considerably. Increased sampling of infrequently visited state-action pairs may improve the quality of estimates and perhaps policies.

^aIn this instance, the greedy policy differed from the optimal policy in cells 4, 8 and 14. Cells 4 and 8 were sampled infrequently and in cell 14 there was only a small difference in estimated state-action values.

^bThe RMSE compared the estimated state-action value function to the optimal state-action value function obtained using exact methods.

10.7.2 Infinite horizon discounted models

This section describes and examines the use of Q-learning and SARSA in infinite horizon discounted models assuming trajectory-based data.

Non-optimal actions	Q-learning				SARSA			
	Exp1	Exp2	STC	Log	Exp1	Exp2	STC	Log
0	14	7	14	20	20	21	17	16
1	20	26	20	17	16	15	18	21
2	6	6	6	3	4	4	5	3
3	0	1	0	0	0	0	0	0

Table 10.5: Comparison of update method and learning rate on the basis of the number of non-optimal actions determined by the greedy policy.

Non-optimal action(s)	$v^*(13) - v(13)$	Median difference
4	0.12	0.12
8	0.35	0.30
4,8	0.44	0.41
4,8,14	2.22	0.54

Table 10.6: Effect of non-optimal actions on the deviation of the optimal value function from that of policies with indicated non-optimal actions. The last column indicates the median difference over all states. As a point of reference $v^*(13) = 29.22$ and the median of $v^*(s)$ over all s was 35.29.

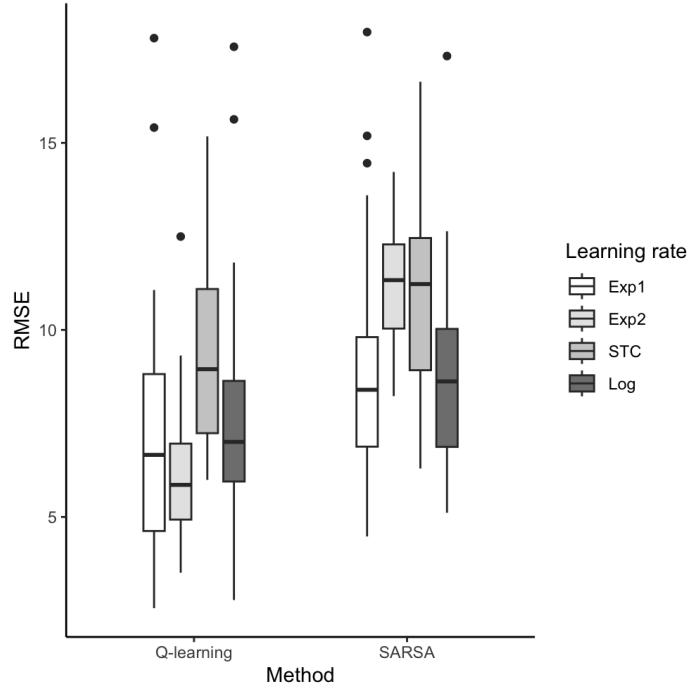


Figure 10.20: Boxplots of RMSE by learning rate and method for Example 10.8.

Algorithm 10.11. Q-learning for a discounted model**1. Initialize:**

- (a) Set $q(s, a) \leftarrow 0$ and $\text{count}(s, a) \leftarrow 0$ for each $a \in A_s$ and $s \in S$.
- (b) Specify the learning rate $\tau_n, n = 1, 2, \dots$
- (c) Specify the number of iterations N .
- (d) Specify a sequence $\epsilon_n, n = 1, 2, \dots$ (or η_n for softmax sampling).
- (e) Specify $s \in S$ and $a \in A_s$.
- (f) $n \leftarrow 1$.

2. Iterate: While $n \leq N$:

- (a) $\text{count}(s, a) \leftarrow \text{count}(s, a) + 1$.
- (b) Simulate (s', r) or sample s' from $p(\cdot|s, a)$ and set $r \leftarrow r(s, a, s')$.
- (c) **Update** $q(s, a)$:

$$q(s, a) \leftarrow q(s, a) + \tau_{\text{count}(s, a)} \left(r + \lambda \max_{a' \in A_s} q(s', a') - q(s, a) \right). \quad (10.76)$$

- (d) Choose $a \in A_{s'}$ using ϵ -greedy (or softmax) sampling.
- (e) $s \leftarrow s'$.
- (f) $n \leftarrow n + 1$.

3. Terminate: Return $q(s, a)$ for all $a \in A_s$ and $s \in S$.

Some comments follow:

1. The above algorithm is trajectory-based. It can be modified to be state-action pair-based by randomly sampling states and actions prior to generating s' and r in step 2(b) and removing step 2(e). Such a modification can be implemented in a simulator and may apply in some real systems.
2. As noted in the episodic case, SARSA replaces steps 2(c)-2(e) with:
 - (c) Sample $a' \in A_{s'}$
 - (d) Update $q(s, a)$ by

$$q(s, a) \leftarrow q(s, a) + \tau_{\text{count}(s, a)} (r + \lambda q(s', a') - q(s, a)). \quad (10.77)$$

- (e) $s \leftarrow s', a \leftarrow a'$.

The difference is that it uses action a' to *both* update $q(s, a)$ and to generate a state and reward at the next iteration.

3. The algorithm returns a state-action value function that can be used to identify a greedy decision rule $d^*(s)$ and policy $(d^*)^\infty$. The value of this policy can be approximated by $q(s, d^*(s))$, evaluated exactly by solving the policy evaluation equations if a model is available, or by simulation or real-time implementation otherwise.
4. If every state-action pair is visited infinitely often and the learning rates satisfy the Robbins-Monro conditions, then the iterates of Q-learning converge with probability one to the optimal state-action value function.

Example of Q-learning and SARSA in a discounted model

The following example illustrates the challenges of selecting parameters for Q-learning and SARSA by investigating the impact of exploration and learning rate on policy choice and state-action value function estimation.

It applies Algorithm [10.11] and its SARSA variant to find estimates of $q^*(s, a)$ in a discounted ($\lambda = 0.9$) version of the two-state model from Section 2.5. Although this example has only four stationary policies, it provides insights that can be applied in more complex settings such as the example in Section [10.9].

Results below are based on setting $N = 30,000$ and using trajectory-based data. They compare learning rates $\tau_n = 0.5n^{-0.6}$ (Exp), $\tau_n = 150/(300 + n)$ (Ratio), $\tau_n = 0.1(1 + n^2/10^5)^{-1}$ (STC) and $\tau_n = \log(n + 1)/n$ (Log), and exploration approaches ϵ -greedy with step function decay $\epsilon_n = 0.01 + 0.34I_{[1, 5,000]}(n) + 0.09I_{[0, 10,000]}(n)$ (Step), ϵ -greedy with ratio decay $\epsilon_n = 150/(500+n)$ (Ratio) and with $\eta_n = 0.01n^{0.5}$ (Softmax).

For the learning rate, n equals the number of times a state-action pair was selected. For action choice, n was set to number of times a state was visited. This was done to account for the asynchronous evaluation of state-action pairs. Comparison to setting n equal to the iteration number is left as an exercise.

The simulation study compared all 24 combinations of learning rate, exploration method and recursion (SARSA vs. Q-learning) using 40 replicates with common random number seed for each. Results are summarized in terms of:

1. the fraction of instances in which the greedy policy and the optimal policy were identical, and
2. the RMSE of the estimated matrix of state-action values compared to the exact optimal state-action value function represented by the matrix \mathbf{Q}^* .

Solving for $v_\lambda^{d^\infty}(s)$ and applying (10.61) yields

$$\mathbf{Q}^* = \begin{bmatrix} 29.74 & \mathbf{30.15} \\ 20.15 & \mathbf{27.94} \end{bmatrix},$$

where the entries in bold represent the maximum of the row. Thus the optimal policy $(d^*)^\infty$, corresponds to $d^*(s_1) = a_{1,2}$ and $d^*(s_2) = a_{2,2}$, with value $\mathbf{v}_\lambda^* = (30.15, 27.94)$. Recall that the next best policy with $d(s_1) = a_{1,1}$ and $d(s_2) = a_{2,2}$ had value $(27.18, 25.62)$.

		Q-learning			SARSA		
		Step	Ratio	Softmax	Step	Ratio	Softmax
Learning Rate	Exp	67	74	56	79	18	85
	Ratio	62	74	67	28	51	97
	STC	92	90	62	0	46	46
	Log	67	90	72	79	38	92

Table 10.7: Percentage of replicates in which the greedy policy agreed with the optimal policy.

Table 10.7 shows the percentage of replicates in which the optimal policy was identified by greedy action choice using the estimated state-action value function. Observe that:

1. There was considerable variability in accuracy.
2. **Main effects:** Averaged over all other factors, Q-learning chose the optimal policy more frequently than SARSA (73% vs. 55%), softmax exploration chose the optimal policy most frequently (72%) among exploration methods, and the Log learning rate chose the optimal policy most frequently (73%) among learning rates.
3. **Q-learning:** STC with step function and ratio exploration chose the optimal policy most frequently. Softmax exploration was the least accurate.
4. **SARSA:** Softmax exploration was most accurate, especially in the case of Ratio and Log learning rates.

Figure 10.21 provides boxplots of the RMSE as it varied across 40 replicates for each combination of learning rate, exploration method and recursion.

They show that:

1. Q-learning estimates were considerably more accurate than SARSA estimates.
2. Q-learning estimates were less sensitive to exploration method than SARSA estimates. This is to be expected since, as noted above, SARSA evaluates the learning policy implied by the exploration method.
3. For Q-learning, the Log learning rate produced the most accurate estimates. In this case, estimates based on exploration with the step function or ratio estimate produced the most accurate and least variable estimates, respectively.

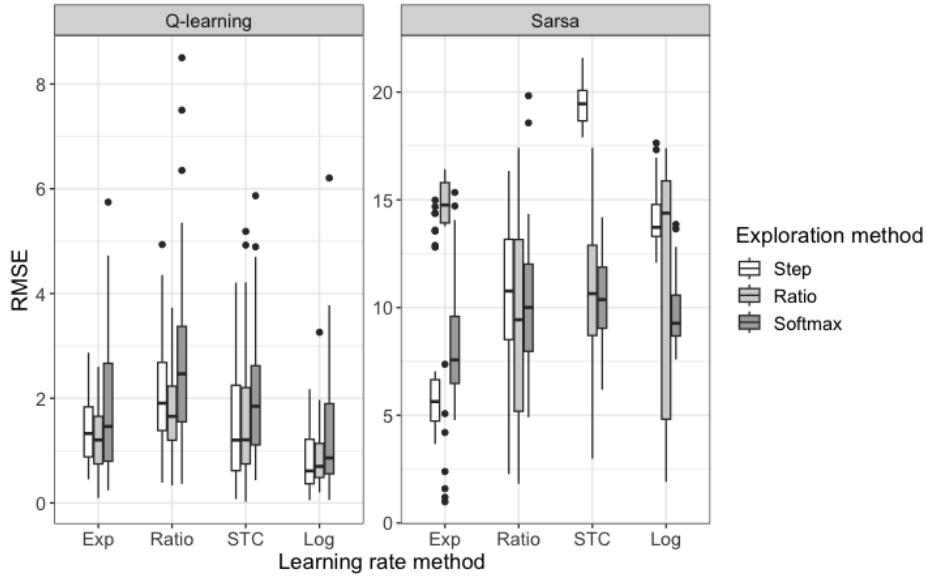


Figure 10.21: Boxplot of RMSE based on 40 simulated replicates. Note that the vertical scales on the left and right plots differ.

4. For SARSA, the exponential decay learning rate with step function exploration produced the most accurate estimates. The estimates using a ratio learning rate were highly variable.

These results show that the most accurate estimates in terms of RMSE were not always in agreement with the methods that chose the best policy. Since SARSA methods were less accurate in estimating the q -function, softmax may be able to account for this by using the relative values of the q -function as its basis for action choice.

The following state-action value matrices, generated using a specific random number seed, offer deeper insight into the effects of parameters and method on estimates. The matrix labels use subscripts to denote the method (Q-learning or SARSA), learning rate, and exploration technique. The bold entry within each row denotes the greedy action in that state.

$$\begin{aligned} \mathbf{Q}_{Q,STC,Step} &= \begin{bmatrix} 30.36 & \mathbf{30.51} \\ 21.00 & \mathbf{28.51} \end{bmatrix} & \mathbf{Q}_{Q,Log,Rat} &= \begin{bmatrix} 29.92 & \mathbf{30.64} \\ 21.00 & \mathbf{28.42} \end{bmatrix} \\ \mathbf{Q}_{Q,Exp,Smax} &= \begin{bmatrix} \mathbf{30.93} & 30.79 \\ 20.75 & \mathbf{29.86} \end{bmatrix} & \mathbf{Q}_{S,Rat,Smax} &= \begin{bmatrix} 15.00 & \mathbf{30.56} \\ 13.58 & \mathbf{29.17} \end{bmatrix} \\ \mathbf{Q}_{S,Exp,Step} &= \begin{bmatrix} 20.25 & \mathbf{26.90} \\ 16.68 & \mathbf{26.20} \end{bmatrix} & \mathbf{Q}_{S,Log,Rat} &= \begin{bmatrix} \mathbf{16.15} & 11.90 \\ 7.45 & \mathbf{13.95} \end{bmatrix}. \end{aligned}$$

Observe that for Q-learning, the instances $Q, STC, Step$ and Q, Log, Rat accurately estimate the optimal state-action value functions and identify the optimal policy, while the instance $Q, Exp, Smax$ quite accurately estimated \mathbf{Q}^* but the greedy action was

not optimal. In the case of SARSA, no instance provided an accurate estimate of \mathbf{Q}^* . The instances for $S, Rat, Smax$ and $S, Exp, Step$ identified the optimal policy while S, Log, Rat did not. Note also that when the greedy policy was not optimal, the discrepancy occurred in state s_1 where the two values in \mathbf{Q}^* are quite close.

The above analysis provides a framework for use in other applications but cautions against generalizing specific recommendations regarding selection of learning rates and exploration methods. As in the episodic example, results are sensitive to learning rates and exploration method.

10.7.3 Infinite horizon average reward models

This section describes and illustrates two Q-learning approaches for finding effective policies for infinite horizon models with the average reward criterion. Reviewing Chapter 7 provides relevant background.

Following (7.63), the optimal state-action value function $q^*(s, a)$ for a recurrent or unichain average reward model is given by:

$$q^*(s, a) = \sum_{j \in S} p(j|s, a)(r(s, a, j) - g^* + h^*(j)) \quad (10.78)$$

and the optimal state-action value function and the optimal gain is a solution of the Bellman equation:

$$q(s, a) = \sum_{j \in S} p(j|s, a) \left(r(s, a, j) - g + \max_{a' \in A_j} q(j, a') \right). \quad (10.79)$$

Written as an expected value, (10.79) is equivalent to:

$$q(s, a) = E^{d^*} \left[r(X, Y, X') - g + \max_{a' \in A_{X'}} q(X', a') \mid X = s, Y = a \right].$$

Applying gradient descent⁵⁴ to minimize the squared difference between $q(s, a)$ and its realized values suggests the following model-based recursion for estimating $q(s, a)$ using simulation or process data:

$$q(s, a) \leftarrow q(s, a) + \tau \left(r(s, a, s') - g + \max_{a' \in A_{s'}} q(s', a') - q(s, a) \right) \quad (10.80)$$

where s' is sampled from $p(\cdot|s, a)$. The Q-learning algorithm below implements this recursion by combining it with a recursion that updates the estimate of g at *every* iteration. Note that the model-free version replaces $r(s, a, s')$ by an observed reward r .

⁵⁴With the caveat that the semi-gradient replaces the gradient.

Q-learning

The following Q-learning algorithm generalizes the variant of Algorithm 10.8 that uses (10.60) to update the gain.

Algorithm 10.12. Q-learning for an average reward model

1. Initialize:

- (a) Set $q(s, a) \leftarrow 0$, $g \leftarrow 0$ and $\text{count}(s, a) \leftarrow 0$ each $a \in A_s$ and $s \in S$.
- (b) Specify learning rates $\tau_n, n = 1, 2, \dots$ and $\beta_n, n = 1, 2, \dots$
- (c) Specify the number of iterations N .
- (d) Specify a sequence $\epsilon_n, n = 1, 2, \dots$ (or η_n for softmax sampling).
- (e) Specify $s \in S$ and $a \in A_s$.
- (f) $n \leftarrow 1$.

2. Iterate: While $n \leq N$:

- (a) $\text{count}(s, a) \leftarrow \text{count}(s, a) + 1$.
- (b) Simulate (s', r) or sample s' from $p(\cdot|s, a)$ and set $r \leftarrow r(s, a, s')$.
- (c) **Update** $q(s, a)$:

$$q(s, a) \leftarrow q(s, a) + \tau_{\text{count}(s, a)} \left(r - g + \max_{a' \in A_{s'}} q(s', a') - q(s, a) \right). \quad (10.81)$$

- (d) **Update** g :

$$g \leftarrow g + \beta_n(r - g). \quad (10.82)$$

- (e) Generate $a \in A_{s'}$ using ϵ -greedy (or softmax) sampling.
- (f) $s \leftarrow s'$.
- (g) $n \leftarrow n + 1$.

3. Terminate: Return $q(s, a)$ for all $a \in A_s$ and $s \in S$.

Some comments follow:

1. Note that since g is updated at every iteration, its learning rate β_n in (10.82) is indexed by the iteration index n . On the other hand, the learning rate for $q(s, a)$ in (10.81) may be indexed by n or the number of times that state-action pair (s, a) has been previously evaluated.

2. Perhaps better estimates could be obtained by updating g using (10.82) before updating $q(s, a)$ using (10.81).
3. Observe that there are four hyperparameters to specify: two learning rates, the exploration rate and the number of iterations. Experimentation with these parameters in the context of policy evaluation through temporal differencing (Algorithm 10.8) might guide choice.
4. As noted earlier in this chapter, the above implementation is off-policy, that is the action chosen for execution at the next iteration need not correspond to the value $\max_{a' \in A_s} q(s', a')$ used in the update (10.81). A SARSA (on-policy update) replaces this expression by

$$q(s, a) \leftarrow q(s, a) + \tau_{\text{count}(s, a)} (r - g + q(s', a') - q(s, a)).$$

where a' denotes an ϵ -greedy action choice in state s' . Note that in numerical experiments, SARSA did not perform well.

5. Algorithm 10.12 converges⁵⁵ under the conditions that every state-action pair is visited infinitely often, the Markov chain of every stationary policy is regular and aperiodic, and the Robbins-Monro conditions on the learning rates hold.
6. If the algorithm is implemented using simulation, state-action based sampling at the start of step 2 might enhance empirical performance through more accurate estimates of $q(s, a)$ for infrequently visited states.

Relative Q-learning

Numerical studies suggest that the iterates in the above algorithm frequently diverge. The following alternative approach⁵⁶ generalizes relative value iteration (Algorithm 7.2). It considers updates of the form

$$q(s, a) \leftarrow q(s, a) + \tau \left(r - f(q) + \max_{a' \in A_{s'}} q(s', a') - q(s, a) \right), \quad (10.83)$$

where possible choices for $f(q)$ include:

1. $f(q) = q(\bar{s}, \bar{a})$ for some specified pair (\bar{s}, \bar{a}) ,
2. $f(q) = \max_{a \in A_{\bar{s}}} q(\bar{s}, a)$ for some specified state \bar{s} , and
3. $f(q)$ equal to the average of $q(s, a)$ over all state-action pairs.

⁵⁵See Tsitsiklis and Roy [1999].

⁵⁶This algorithm and variants was proposed and analyzed by Abounadi et al. [2001].

Many implementations of relative value iteration use the specification $f(q) = q(\bar{s}, \bar{a})$, which was used to analyze (numerically) relative value iteration in Chapter 7. However, its performance is sensitive to the designation of (\bar{s}, \bar{a}) . If visited infrequently, estimates will be inaccurate resulting in inaccurate estimates of $q(s, a)$. If possible, frequent restarts in (\bar{s}, \bar{a}) may result in improved performance.

Algorithm 10.13. Relative Q-learning for an average reward model
1. Initialize:

- Specify a real-valued function $f(q)$.
- Set $q(s, a) \leftarrow 0$ and $\text{count}(s, a) \leftarrow 0$ each $a \in A_s$ and $s \in S$.
- Specify the learning rate $\tau_n, n = 1, 2, \dots$
- Specify the number of iterations N .
- Specify a sequence $\epsilon_n, n = 1, 2, \dots$ (or η_n for softmax sampling).
- Specify $s \in S$ and $a \in A_s$.
- $n \leftarrow 1$.

2. Iterate: While $n \leq N$:

- $\text{count}(s, a) \leftarrow \text{count}(s, a) + 1$.
- Simulate (s', r) or sample s' from $p(\cdot|s, a)$ and set $r \leftarrow r(s, a, s')$.
- Update $q(s, a)$:**

$$q(s, a) \leftarrow q(s, a) + \tau_{\text{count}(s, a)} \left(r - f(q) + \max_{a' \in A_{s'}} q(s', a') - q(s, a) \right). \quad (10.84)$$

- Generate $a \in A_{s'}$ using ϵ -greedy (or softmax) sampling.
- $n \leftarrow n + 1$ and $s \leftarrow s'$.

3. Terminate: Return $q(s, a)$ for all $a \in A_s$ and $s \in S$.

Observe that Relative Q-learning and Q-learning differ in how they estimate g . Relative Q-learning computes the estimate of g , $f(q)$, in step 2(c) using the previously evaluated $q(s, a)$, while Q-learning directly updates g at every iteration using (10.82).

From a theoretical perspective, in a unchain model:

- The iterates of $f(q)$ converge to g^* ,
- The iterates of $q(s, a)$ converge to $q^*(s, a)$ normalized so that $f(q^*) = g$,
- The greedy policy based on $q^*(s, a)$ is optimal.

These results hold under the Robbins-Monro step-size conditions and any of the above specifications for $f(q)$.

The two-state example

Applying Q-learning and Relative Q-learning to the two-state model of Section 2.5 highlights some inherent challenges when using these methods. Calculations in Section 7.7.2 establish that the stationary policy $(d^*)^\infty$ that chooses $d^*(s_1) = a_{1,2}$ and $d^*(s_2) = a_{2,2}$ is optimal, that $g^* = 2.86$, and the relative value obtained setting $h^*(s_1) = 0$ is $h_{\text{rel}}^*(s_2) = h^*(s_2) - h^*(s_1) = -2.14$.

An immediate issue is how to compare methods and model specifications. Comparing resulting policies on the basis of their gain and agreement with the optimal policy is straightforward. Comparisons based on the state-action value functions is problematic because $q^*(s, a)$ is unique up to a constant that is independent of (s, a) . Instead results can be compared on the basis of relative values⁵⁷, under a specification such as $h^*(s_1) = 0$.

To determine an easy to use representation for relative values, recall that $q^*(s, a)$ is defined in Chapter 7 by

$$q^*(s, a) = E^{(d^*)^\infty} \left[\sum_{n=1}^{\infty} (r(X_n, Y_n) - g^*) \middle| X_1 = s, Y_1 = a \right].$$

Therefore $h^*(s) = q^*(s, d^*(s))$ for all $s \in S$ where $(d^*)^\infty$ is an optimal stationary policy. Since

$$d^*(s) \in \arg \max_{a \in A_s} q^*(s, a),$$

$h^*(s) = \max_{a \in A_s} q^*(s, a)$. To ensure $h^*(s_1) = 0$, subtract $h^*(s_1)$ from $h^*(s)$ for all $s \in S$ so that

$$h_{\text{rel}}^*(s) := h^*(s) - h^*(s_1)$$

for all $s \in S$.

Putting this altogether, if in this example $q(s, a)$ is obtained by applying Q-learning or Relative Q-learning, an estimate of $h_{\text{rel}}^*(s_2)$ is given by

$$h_{\text{rel}}(s_2) = q(s_2, d^*(s_2)) - q(s_1, d^*(s_1)) = \max_{a \in A_{s_2}} q(s, a) - \max_{a \in A_{s_1}} q(s_1, a) \quad (10.85)$$

Therefore, $h_{\text{rel}}(s_2)$ can be compared to $h^*(s_2) = -2.14$. The beauty of using this representation for $h_{\text{rel}}(s)$ is that it extends easily to larger applications and avoids evaluating the greedy policy explicitly. It also could be the basis for a formal termination criterion for Q-learning or Relative Q-learning.

Estimates using Q-learning and Relative Q-learning set $\tau_n = n^{-1} \log(n + 1)$, $\beta_n = n^{-1}$ (for Q-learning), and used ϵ_n -greedy exploration with $\epsilon_n = 150/(300 + n)$. The

⁵⁷This quantity is more stable than $q^*(s, a)$ since some state-action pairs may be visited infrequently and not accurately estimated.

index for τ_n was the number of visits to each state-action pair and the index for ϵ_n was the number of previous visits to that state⁵⁸. Relative Q-learning used seven choices for $f(q)$: $f(q) = q(\bar{s}, \bar{a})$ where (\bar{s}, \bar{a}) varied over all (four) state-action pairs (denoted RQL1-RQL4), $f(q) = \max_{a \in A_s} q(\bar{s}, a)$ for some \bar{s} (denoted RQL5-RQL6) and $f(q) = \text{mean } q(\cdot, \cdot)$ (denoted RQL7). Comparisons were based on 40 independent replicates. Each replicate evaluated Q-learning and Relative Q-learning across all 7 choices of $f(q)$, with $N = 20,000, 35,000$, and $50,000$. Common random number seeds were used for each of 24 instances evaluated within a replicate.

Table 10.8 gives the number of replications in which there were the designated number of non-optimal actions, classified by algorithm version and number of iterations. Observe that:

1. Q-learning most accurately identified the optimal policy. The impact of run length was minimal.
2. Relative value results varied with the form of $f(q)$. Accuracy for all improved as the number of iterations increased.
3. The differences between RQL1-RQL4 show that the method is sensitive to choice of (\bar{s}, \bar{a}) .
4. RQL5, which used $f(q) = \max_{a \in A_{s_1}} q(s_1, a)$, more accurately chose the optimal policy than RQL6, which used $\bar{s} = s_2$.

Iterations	Non-optimal actions	QL	RQL1	RQL2	RQL3	RQL4	RQL5	RQL6	RQL7
20,000	0	38	33	23	30	27	34	28	32
	1	0	5	10	7	4	3	7	6
	2	2	2	7	3	9	3	5	2
35,000	0	39	34	28	34	26	36	29	34
	1	0	4	10	5	6	2	8	5
	2	1	2	2	1	8	2	3	1
50,000	0	39	34	28	37	26	37	31	36
	1	0	6	10	2	8	1	6	3
	2	1	0	2	1	6	2	3	1

Table 10.8: Distribution of the number of non-optimal actions in 40 replicates broken down by method and run-length for average reward version of the two-state model.

Figure 10.22 shows the accuracy and variability of estimates of g^* and $h^*(s_2)$, classified by run length and optimization method. Observe that:

1. Q-learning produced the most accurate and least variable estimates of g^* . Its variability decreased as N increased.

⁵⁸For state s , $n = \sum_{a \in A_s} \text{count}(s, a)$.

2. RQL7 and RQL5 provided the best estimates of g^* among the Relative Q-learning methods. These methods used the most iterations to estimate $f(q)$. Overall, Relative Q-learning methods were more sensitive to the number of iterations than Q-learning.
3. Q-learning estimates of $h^*(s_2)$ were as at least as accurate as those of any other methods. Precision increased and variability decreased with the number of iterations.
4. The accuracy and variability of Relative Q-learning estimators were similar across all choices of $f(q)$ and most were comparable to that of Q-learning. Accuracy increased and variability decreased with increasing number of iterations.

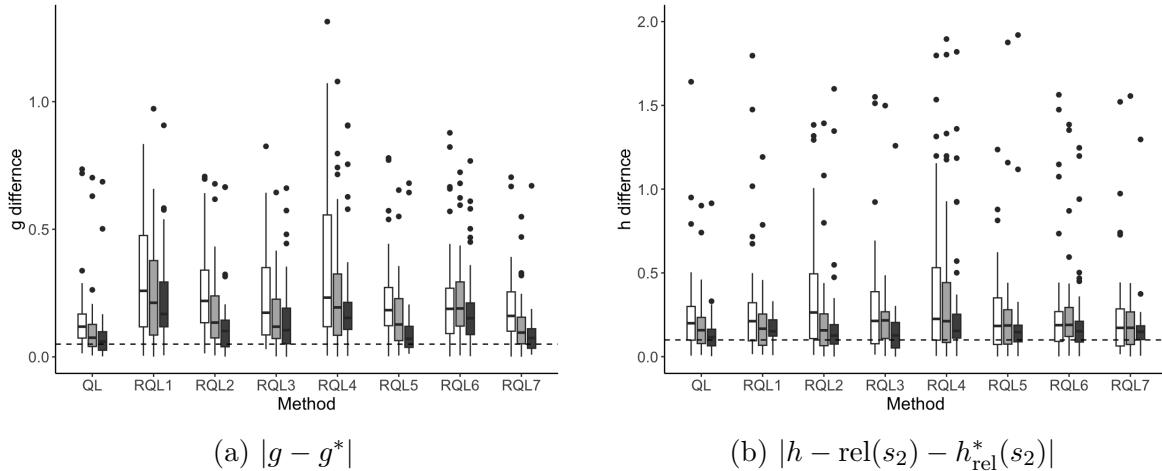


Figure 10.22: Difference between estimates and true values classified by estimation and number of iterations (white fill - 25,000; gray fill - 35,000; dark gray fill - 50,000). Dashed lines are included to facilitate comparisons and not meant as targets for the estimators. Note that the scale of the two figures differ.

As a consequence of these observations, Q-learning, as implemented in Algorithm 10.12, most frequently and efficiently⁵⁹ identified the optimal policy. Moreover it provided the most accurate and least variable estimates of g^* and $h^*(s_2)$.

One reason for the accuracy of estimating the gain was that estimates were updated at *every* iteration (using a running mean). Accuracy and variability of Relative Q-learning methods were sensitive to choice of $f(q)$ and were surpassed by Q-learning.

Of course, these conclusions apply only to this example and the specified learning rate and action sampling parameters. To establish broader validity requires similar comparisons in other environments and a review of relevant recommendations in the literature.

⁵⁹The accuracy of the policies identified with $N = 35,000$ and $N = 50,000$ were the same.

10.8 Policy iteration type algorithms*

There is little general agreement on the structure of policy iteration algorithms in simulation-based environments. The methods proposed below provide plausible implementations.

This section focuses on discounted models; development of simulation-based policy iteration algorithms for episodic and average reward models is left to the reader. Q-learning and SARSA may be viewed as adaptations of value iteration algorithms from Section 5.6 to a simulation environment. Each iterate involves some form of maximization with Q-learning using the “max” to update $q(s, a)$ and SARSA using ϵ -greedy or softmax sampling to select actions at each update.

As was seen in Chapter 5, policy iteration and modified policy iteration involved less frequent maximizations and faster convergence, so it may be fruitful to investigate simulation-based versions of policy iteration (Section 5.7) and modified policy iteration (Section 5.8). In general, the structure of such algorithms can be represented by Figure 10.23. In earlier chapters, such an algorithm was implemented using the known Markov decision process model to compute $q^{d^\infty}(s, a)$ exactly using (5.64). This section explores the use of simulation methods to estimate this intermediate quantity. As a consequence of the variability introduced through simulation, the stopping rule “stop when $d' = d$ ” may require modification.

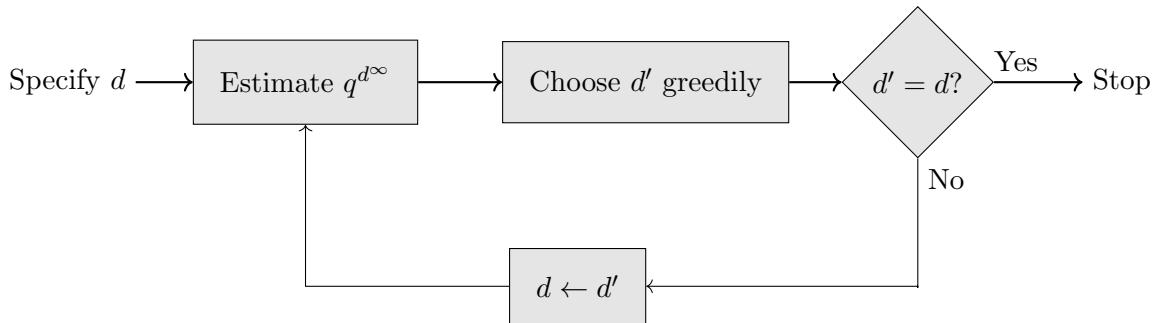


Figure 10.23: Structure of a policy iteration algorithm.

Simulation-based policy iteration methods can implement the estimation step in different ways:

1. Estimate $v_\lambda^{d^\infty}(s)$ and compute $q_\lambda^{d^\infty}(s, a)$ using the model.
2. Sample to estimate $r(s, a, j)$ and $p(j|s, a)$, then use these quantities to estimate $v_\lambda^{d^\infty}(s)$ and $q_\lambda^{d^\infty}(s, a)$.
3. Estimate $v_\lambda^{d^\infty}(s)$ and then simulate to estimate $q_\lambda^{d^\infty}(s, a)$.
4. Estimate $q_\lambda^{d^\infty}(s, a)$ directly.

A discussion of the first and last methods follow. Analysis of the other two approaches is left to the reader.

10.8.1 Hybrid policy improvement

First, consider a value function-based algorithm in which the evaluation step uses simulation to estimate the policy value function and then implements “exact” improvement in the same way as deterministic policy iteration in Chapter 5. While this may be impractical to implement when either S or A_s are large, and contrasts with the philosophy of this chapter, it will provide a useful baseline for algorithms in which improvement also involves simulation.

Algorithm 10.14. Hybrid policy improvement for a discounted model

1. **Initialize:** Specify $d \in D^{\text{MD}}$, stopping tolerance $\nu \in \mathbb{Z}_+$, and set $\Gamma = S$.
2. **Iterate:** While $|\Gamma| > \nu$:
 - (a) **Evaluation:** Evaluate d using TD(0), TD(γ) or Monte Carlo to obtain $\hat{v}(s)$ for $s \in S$.
 - (b) **Construct $\hat{q}(s, a)$:** For all $s \in S$ and $a \in A_s$, set

$$\hat{q}(s, a) \leftarrow p(j|s, a)(r(s, a, j) + \lambda\hat{v}(j)). \quad (10.86)$$
 - (c) **Exact improvement:** For all $s \in S$,

$$d'(s) \in \arg \max_{a \in A_s} \hat{q}(s, a), \quad (10.87)$$
 setting $d'(s) = d(s)$ if possible.
 - (d) $\Gamma \leftarrow \{s \in S \mid d'(s) \neq d(s)\}$.
 - (e) $d \leftarrow d'$.
3. **Terminate:** Return d .

Note that the termination criterion is based on the number of states in which actions change in successive iterations of step 2. While the condition $\nu = 0$ works for exact methods, empirical results suggest that it is too stringent for practical applications.

This algorithm does not apply to model-free settings in which $p(j|s, a)$ and $r(s, a, j)$ are not known. However, it can be adapted to a model-free setting using a two-stage approach in which $\hat{p}(j|s, a)$ and $\hat{r}(s, a, j)$ are learned from data. To facilitate this, the algorithm should start with a decision rule that randomizes action choice in all states. After obtaining these estimates, Algorithm 10.14 can be applied using

$$\hat{q}(s, a) := \hat{p}(j|s, a)(\hat{r}(s, a, j) + \lambda\hat{v}(j)). \quad (10.88)$$

Furthermore these estimates can be improved while evaluating subsequent decision rules.

Example 10.9. Hybrid policy improvement in the two-state model.

This example applies Algorithm 10.14 to the two-state model from Section 2.5 with $\lambda = 0.9$ and stopping tolerance $\nu = 0$. Results from the example in Section 10.7.2 guide parameter choice. It uses TD(0) for evaluation, sets $N = 5,000$ and compares two learning rates: $\tau_n = \log(n+1)/n$ and $\tau_n = 20/(50+n)$, where n represents the number of visits to the state. It also compares implementing TD(0) using state-based uniform random sampling with trajectory-based sampling. The algorithm is initialized with the sub-optimal policy $d = (a_{1,1}, a_{2,1})$.

The state-action value functions corresponding to this policy are

$$\begin{aligned}\hat{q}(s_1, a_{1,1}) &= 3 + 0.9(0.8\hat{v}(s_1) + 0.2\hat{v}(s_2)) \\ \hat{q}(s_1, a_{1,2}) &= 5 + 0.9\hat{v}(s_2) \\ \hat{q}(s_2, a_{2,1}) &= -5 + 0.9\hat{v}(s_2) \\ \hat{q}(s_1, a_{2,2}) &= 2 + 0.9(0.4\hat{v}(s_1) + 0.6\hat{v}(s_2))\end{aligned}$$

where $\hat{v}(s)$ is an estimate of $v_\lambda^{d^\infty}(s)$ for the policy being evaluated.

Experiments consisted of 50 replicates using common random number seeds to compare learning rates and methods for generating the next state. Comparisons were based on the number of iterations to converge, the policy identified and the RMSE between the estimated $\hat{q}(s, a)$ and the known value of $q_\lambda^*(s, a)$.

In *all* cases the algorithm terminated with the optimal policy in a few iterations. For $N = 5,000$, the termination condition was achieved in most cases in five or fewer iterations. When N was increased to 20,000, trajectory-based sampling ensured convergence within three iterations for both learning rate choices. However, with state-based sampling, a few replications required up to five iterations.

Table 10.9 shows that TD(0) implemented with the log learning rate and trajectory-based state sampling most accurately estimated $q_\lambda^*(s, a)$ for both values of N . Furthermore, increasing N from 5,000 to 20,000 significantly improved the quality of estimates and resulted in faster algorithmic convergence.

It is left to the reader to apply variants of policy iteration that:

1. Estimate $r(s, a, j)$ and $p(j|s, a)$ assuming a model-free environment and then using them to estimate $q_\lambda^{d^\infty}(s, a)$, and
2. Estimate $\sum_{j \in S} p(j|s, a)(r(s, a, j) + \lambda\hat{v}(j))$ using sampling after estimating $\hat{v}(\cdot)$.

Learning rate	State sampling	RMSE	
		$N = 5,000$	$N = 20,000$
$\log(n+1)/n$	State-based	1.88 (1.37)	0.92 (0.73)
	Trajectory-based	1.59 (1.43)	0.66 (0.55)
$20/(50+n)$	State-based	2.36 (1.54)	1.00 (0.75)
	Trajectory-based	2.22 (1.31)	0.77 (0.79)

Table 10.9: Mean (standard deviation) of RMSE errors of q -function estimates using hybrid policy iteration classified by learning rates and state generation method based on 50 replicates.

10.8.2 Hybrid policy iteration bounds

While theoretically justified in exact computation, the stopping rule in Algorithm 10.14 lacks a theoretical basis. However, one would expect that if values were accurately estimated, the condition to stop when $d' = d$ would eventually hold. Value function bounds, previously discussed in Chapter 5, provide a more rigorous approach for stopping, as well as bounding the difference between the value of a policy and the optimal value function.

Theorem 5.12 shows that for any function $v(s)$ on S ,

$$\begin{aligned} v(s) + (1 - \lambda)^{-1} \min_{s' \in S} \{L\mathbf{v}(s') - v(s')\} &\leq v_\lambda^{d^\infty}(s) \\ &\leq v_\lambda^*(s) \leq v(s) + (1 - \lambda)^{-1} \max_{s' \in S} \{L\mathbf{v}(s') - v(s')\} \end{aligned}$$

where

$$L\mathbf{v}(s) := \max_{a \in A_s} \left\{ \sum_{j \in S} p(j|s, a) (r(s, a, j) + \lambda v(j)) \right\} = \max_{a \in A_s} q(s, a)$$

and

$$d(s) \in \arg \max_{a \in A_s} q(s, a).$$

Applying the above expressions to the iterates of hybrid policy iteration results in the following readily applicable bounds⁶⁰.

⁶⁰We do not believe these have previously been used with simulated data.

Theorem 10.2. Suppose $\hat{v}(s)$, $\hat{q}(s, a)$ and d' are determined in step 2 of Algorithm 10.14. Then

$$\begin{aligned} \hat{v}(s) + (1 - \lambda)^{-1} \min_{s' \in S} \left\{ \max_{a' \in A_{s'}} \hat{q}(s', a') - \hat{v}(s') \right\} &\leq v_\lambda^{(d')^\infty}(s) \\ &\leq v_\lambda^*(s) \leq \hat{v}(s) + (1 - \lambda)^{-1} \max_{s' \in S} \left\{ \max_{a' \in A_{s'}} \hat{q}(s', a') - \hat{v}(s') \right\} \end{aligned} \quad (10.89)$$

It is important to emphasize that for (10.89) to be valid $\hat{q}(s, a)$ must be defined by (10.86). While one might hypothesize that this inequality remains valid when $\hat{q}(s, a)$ is estimated directly (as in Q-learning), our numerical experiments indicated that this supposition is false. Obtaining efficient stopping rules for algorithms based on state-action value functions remains an open question. One approach in a model-based environment would be to intermittently set $\hat{v}(s) = \max_{a \in A_s} \hat{q}(s, a)$ and subsequently use (10.86) to obtain $\hat{q}(s, a)$ that is compatible with the bounds.

Since all of the quantities in (10.89) are available after step 2(b) of hybrid policy iteration, the above bounds provide the basis for the following stopping criterion for Algorithm 10.14.

Corollary 10.1. Let

$$\delta := (1 - \lambda)^{-1} \left(\max_{s' \in S} \left\{ \max_{a' \in A_{s'}} \hat{q}(s', a') - \hat{v}(s') \right\} - \min_{s' \in S} \left\{ \max_{a' \in A_{s'}} \hat{q}(s', a') - \hat{v}(s') \right\} \right). \quad (10.90)$$

Then if $\delta < \epsilon$,

$$v_\lambda^*(s) - \hat{v}(s) < \epsilon \quad \text{for all } s \in S.$$

Moreover (10.89) provides upper and lower bounds on $v_\lambda^*(s)$.

Example 10.10. This example illustrates the application of the stopping criterion in Corollary 10.1 and the bounds on the optimal value function in Theorem 10.2 when hybrid policy iteration is applied to the two-state model. Guided by results in Example 10.9, it generates 50 replicates of hybrid policy iteration with the policy value function estimated by TD(0) with trajectory-based state sampling, a log learning rate and $N = 20,000$.

The stopping criterion was implemented with $\epsilon = 2$ meaning that the algorithm terminates when $\delta < 2$. This tolerance was chosen since it was known for this model with $\lambda = 0.9$ that the difference^a between the value of the optimal policy and that of its closest stationary policy exceeded 2. Smaller values of δ were problematic because of variability in estimates of $\hat{v}(s)$.

Table 10.10 shows the distribution of iterations required to satisfy $\delta < 2$ across replicates. Observe that the algorithm terminated in four or fewer iterations in 86% of the replicates.

Upper and lower bounds along with the value of δ for a typical replicate appear in Table 10.11. In this replicate, hybrid policy iteration would terminate after four iterations. Note that δ is not monotone as a result of variability in TD(0) estimates of $\hat{v}(s)$.

The empirical validity of the bound is supported by the observation that $\mathbf{v}_\lambda^* = (30.15, 27.94)$ lies within *all* computed upper and lower bounds. Moreover, the value of the second-best stationary policy, $(27.19, 25.63)$, lies outside of the bounds for all iterations except the first.

^aOf course this tolerance would not be known a priori in practice.

Iterations	2	3	4	5	6	7
Count	10	15	18	2	4	1

Table 10.10: Distribution of the number of iterations for hybrid policy iteration to satisfy $\delta < 2$ in 50 replicates.

Iteration	Lower bound	Upper bound	δ
1	$\begin{bmatrix} -59.42 \\ 62 \\ -109.09 \end{bmatrix}$	$\begin{bmatrix} 248.51 \\ 198.83 \end{bmatrix}$	307.92
2	$\begin{bmatrix} 27.71 \\ 26.03 \end{bmatrix}$	$\begin{bmatrix} 34.91 \\ 33.23 \end{bmatrix}$	7.20
3	$\begin{bmatrix} 28.54 \\ 26.69 \end{bmatrix}$	$\begin{bmatrix} 33.29 \\ 31.43 \end{bmatrix}$	4.74
4	$\begin{bmatrix} 30.10 \\ 27.90 \end{bmatrix}$	$\begin{bmatrix} 30.25 \\ 28.05 \end{bmatrix}$	0.15
5	$\begin{bmatrix} 29.11 \\ 26.79 \end{bmatrix}$	$\begin{bmatrix} 30.67 \\ 28.36 \end{bmatrix}$	1.57

Table 10.11: Upper and lower bounds for $v_\lambda^*(s)$, and the bound, δ , obtained from a typical replicate in Example 10.10.

10.8.3 Online policy improvement

The following policy iteration algorithm is suitable for both model-free and model-based implementations and is designed for online implementation. Its beauty is that it estimates the optimal state-action function directly without first estimating a policy value function, making a model unnecessary. Simulation-based evaluation of $q(s, a)$ is analogous to modified policy iteration.

The improvement step of a policy improvement algorithm requires an estimate of

$$q_\lambda^{d^\infty}(s, a) := \sum_{j \in S} p(j|s, a)(r(s, a, j) + \lambda v_\lambda^{d^\infty}(j)). \quad (10.91)$$

Therefore, it is not sufficient to just evaluate d through simulation since that will only provide an estimate of $q_\lambda^{d^\infty}(s, d(s)) = v_\lambda^{d^\infty}(s)$.

To overcome the absence of estimates of $q(s, a)$ for $a \neq d(s)$, the proposed algorithm estimates $q(s, a)$ for all states and actions by replacing a deterministic policy d^∞ by a randomized policy that is “close” to d^∞ . To do this, the algorithm evaluates the ϵ -tweaked policy d_ϵ that chooses actions in state s according to

$$w_{d_\epsilon}(a|s) = \begin{cases} 1 - \epsilon & a = d(s) \\ \epsilon/(|S| - 1) & a \neq d(s). \end{cases} \quad (10.92)$$

It is important to emphasize that this is different than an ϵ -greedy policy because the policy it is based on need not be optimal. This policy can be implemented in a simulation by sampling action $d(s)$ with probability $1 - \epsilon$ and choosing a specific other action with probability $\epsilon/(|S| - 1)$. It may be prudent to decrease ϵ as calculations ensue.

Algorithm 10.15. Online simulated policy improvement for a discounted model

1. **Initialize:**
 - (a) Specify $d \in D^{\text{MD}}$, stopping tolerance $\nu \in \mathbb{Z}_+$, and set $\Gamma = S$.
 - (b) Specify the number of evaluation iterations N .
 - (c) Specify a sequence $\epsilon_n, n = 1, 2, \dots$ for ϵ -tweaked action sampling and learning rate sequence $\tau_n, n = 1, 2, \dots$
2. **Iterate:** While $|\Gamma| > \nu$:
 - (a) Specify $s \in S$ and set $n \leftarrow 1$.
 - (b) $q(s, a) \leftarrow 0$ and $\text{count}(s, a) \leftarrow 0$ for all $a \in A_s$ and $s \in S$.
 - (c) **Evaluate** $q^{d^\infty}(s, a)$: While $n \leq N$:
 - i. $i \leftarrow \sum_{a \in A_s} \text{count}(s, a)$ and $\epsilon \leftarrow \epsilon_i$.

- ii. Sample a from $w_{d_\epsilon}(\cdot|s)$.
- iii. Simulate (s', r) or sample s' from $p(\cdot|s, a)$ and set $r \leftarrow r(s, a, s')$.
- iv. **Update** $q(s, a)$:

$$q(s, a) \leftarrow q(s, a) + \tau_{\text{count}(s, a)}(r + \lambda q(s', d(s')) - q(s, a)). \quad (10.93)$$

- v. $\text{count}(s, a) \leftarrow \text{count}(s, a) + 1$ and $n \leftarrow n + 1$.
- vi. $s \leftarrow s'$.

(d) **Greedy improvement:** For all $s \in S$,

$$d'(s) \in \arg \max_{a \in A_s} q(s, a), \quad (10.94)$$

setting $d'(s) = d(s)$ if possible.

- (e) $\Gamma = \{s \in S \mid d'(s) \neq d(s)\}$.
- (f) $d \leftarrow d'$.

3. **Terminate:** Return d .

Some comments about applying this algorithm follow:

1. The algorithm might be initiated with a randomized policy that chooses actions in each state with equal probability to obtain a good preliminary estimate of $q(s, a)$, and ϵ could be set to 1 for the first pass through the evaluation step.
2. Choosing $\epsilon_1 = 1 - (1/|S|)$ starts the evaluation with a decision rule that randomizes over all actions. In general, ϵ_n should decrease with n but not too quickly. The algorithm uses the number of times a state has been visited (as opposed to the iteration number) to ensure that there is sufficient action sampling in each state.
3. Step 2(a) can be implemented with a fixed or random s .
4. The algorithm stops when d' and d differ in at most a small number, ν , of states. Setting $\nu = 0$ recovers the stopping rule “stop when a decision rule repeats”. However, this rule may be too stringent when q -functions are estimated directly. A simpler option is to specify the number of times to execute step 2.
5. There are two options for the update of $q(s, a)$ in (10.93). As stated, it is an off-policy algorithm that corresponds to Q-learning where the maximum is over the single policy d . Alternatively an on-policy update, corresponding to SARSA, requires sampling the action a' in state s' according to the ϵ -tweaked decision rule d_ϵ and replacing $d(s')$ in (10.93) by a' . This would require adding $a \leftarrow a'$ at the end of the Evaluate loop.

One would suspect that the off-policy variant would be preferable since the objective is to obtain an estimate of $q_\lambda^{d^\infty}(s, a)$ and not $q_\lambda^{d_\epsilon^\infty}(s, a)$.

6. Step 2(c)vi is specified for trajectory-based sampling. State-based sampling replaces $s \leftarrow s'$ by “Sample $s \in S$ ”. Alternatively, state-action pair sampling would also replace Step 2(c)ii by “sample $a \in A_s$ ”.
7. The algorithm is stated with TD(0) updates. Alternatively, TD(γ) updates could be used.

Example 10.11. Online simulated policy iteration in the two-state model

This example applies Algorithm 10.15 to the two-state model with $\tau_n = \log(n + 1)/n$, $\epsilon_n = 1,500/(3,000 + n)$, $N = 50,000$ and $\nu = 0$ corresponding to stopping when two successive policies are identical. Forty replicates with common random number seeds were used to investigate the combined effect of using:

1. on-policy or off-policy updates in (10.93), and
2. trajectory-based or state-based sampling in step 2(c)vi.

In all cases, the algorithm terminated with an optimal policy. The on-policy variant terminated in three iterations regardless of how the next state was generated while the off-policy variant converged in three iterations in 73 out of 80 cases and in five iterations in 7 out of 80 cases.

Table 10.12 shows the accuracy of the q -function estimates based on the RMSE between the estimated and true values. As expected, the off-policy methods, which are related to Q-learning, were considerably more accurate than the on-policy methods, which are related to SARSA.

The off-policy trajectory-based version produced the most accurate estimates suggesting that an online implementation produces good results. But, the algorithm cycled with less judicious choices of τ_n and ϵ_n so care is needed when specifying parameters.

Update	State sampling	RMSE (mean \pm SD)
On-Policy	State-based	17.48 ± 7.70
	Trajectory-based	20.69 ± 5.35
Off-Policy	State-based	0.88 ± 1.21
	Trajectory-based	0.31 ± 0.35

Table 10.12: Summary of the accuracy of q -function estimates using variants of online simulated policy iteration for the two-state model based on 40 replicates with common random number seeds.

10.9 Queuing service rate control revisited

The infinite horizon version of the queuing service rate control model introduced in Section 3.4.1 provides a good platform to investigate some challenges that arise when implementing the above algorithms in a simple but more realistic model. Particular features that make this problem attractive for analysis include:

1. the ordered and interpretable state space,
2. the known structure of optimal value functions, and
3. the proven monotonicity of the optimal policy.

This section focuses on the discounted version but a short discussion of the average reward version is also included. The objective of this analysis is to find an optimal policy for future use so it is not necessary to use an online implementation or evaluate performance while learning.

Even though implementation involves only 51 states with three actions in each, the problem is sufficiently complex to raise a range of implementation issues that must be considered when dealing with modern large-scale applications. Experiments with a wide range of options are reported and provide a framework that may be applied broadly.

The analysis truncates the length of the queue at 50 and uses the parameters: arrival rate $b = 0.2$, service rates $a_1 = 0.2$, $a_2 = 0.4$ and $a_3 = 0.6$, holding cost $f(s) = s^2$, service rate cost $m(a_k) = 5k^3$ for $k = 1, 2, 3$, and discount rate $\lambda = 0.9$. Because the objective is to minimize the expected discounted cost, it is formulated as a *minimization* problem with positive costs.

Using value iteration and policy iteration (Chapter 5) when $\lambda = 0.9$, the optimal policy $(d^*)^\infty$ was found to be

$$d^*(s) = \begin{cases} a_1 = 0.2 & s \in \{0, \dots, 10\} \\ a_2 = 0.4 & s \in \{11, \dots, 29\} \\ a_3 = 0.6 & s \in \{30, \dots, 50\} \end{cases} \quad (10.95)$$

and the optimal value function was monotone increasing and had monotone increasing differences (was convex) on $\{0, 1, \dots, 48\}$ ⁶¹. Under the average reward criterion, the optimal policy uses actions $a_1 = 0.2$ in states $\{0, 1, 2\}$, $a_2 = 0.4$ in states $\{3, \dots, 8\}$ and $a_3 = 0.6$ in states $\{9, \dots, 50\}$.

10.9.1 Monte Carlo policy evaluation

This section explores the use of truncation and geometric stopping time Monte Carlo estimates of the discounted value function for the policy d^∞ with $d(s) = a_2$ for all

⁶¹As a result of truncation, the values for $s \in \{49, 50\}$ deviated from this pattern.

$s \in S$. Recall that estimates of $v_\lambda^{d^\infty}(s)$ in each state are obtained by averaging over replicates in that state.

Some observations and comments follow:

1. Only starting-state Monte Carlo applies. This is because the truncation horizon or geometric stopping time are defined in terms of the length of the horizon starting from the first transition. Hence, first-visit estimates, which would start later, would correspond to shorter horizons.
2. In this model, the estimated standard deviation $\hat{\sigma}(s)$ of $\hat{v}(s)$ (Figure 10.24a) varied over the states. Thus, sample sizes (number of Monte Carlo replicates) needed to obtain the same precision for all states must vary with the state. Note that the “artificial” boundary at $s = 50$ resulted in the non-monotone behavior near the upper boundary.
3. By the Central Limit Theorem, the Monte Carlo estimate for state s is asymptotically normal with standard deviation approximately equal to $\hat{\sigma}(s)/n^{-0.5}$. This relationship can be used to choose the sample size to obtain a pre-specified level of confidence.
4. In light of the previous comment, the truncation estimator used $K = \max\{100, 4s^2\}$ replicates in state s resulting in a total of 172,080 replicates.
5. Differences between the estimated and true value function are shown in Figure 10.24b for the truncation estimate. Note the RMSE of these estimates (over states) was 12.3. These estimates were more precise than TD(0) estimates described below (Figure 10.26a). However, the total number of replicates was considerably larger.
6. Geometric stopping time estimates were considerably more variable than the truncation estimates (standard deviation in state 50 was 200 times greater than that of the truncation estimator) and much less accurate overall (RMSE = 103.7). Note that the expected number of terms in the geometric sums was 11 as opposed to 125 used in truncation.

10.9.2 TD(γ) policy evaluation

Evaluation using TD(γ) presents some challenges. Under the policy that uses action a_1 in every state, the system is a symmetric reflecting random walk on $\{0, \dots, 50\}$ so that in steady state all states are visited with equal probability. Therefore applying TD(γ) for evaluation of this policy with trajectory-based sampling is straightforward since all states will be visited approximately the same number of times in a simulation. However, evaluating other reasonable policies becomes problematic because the system

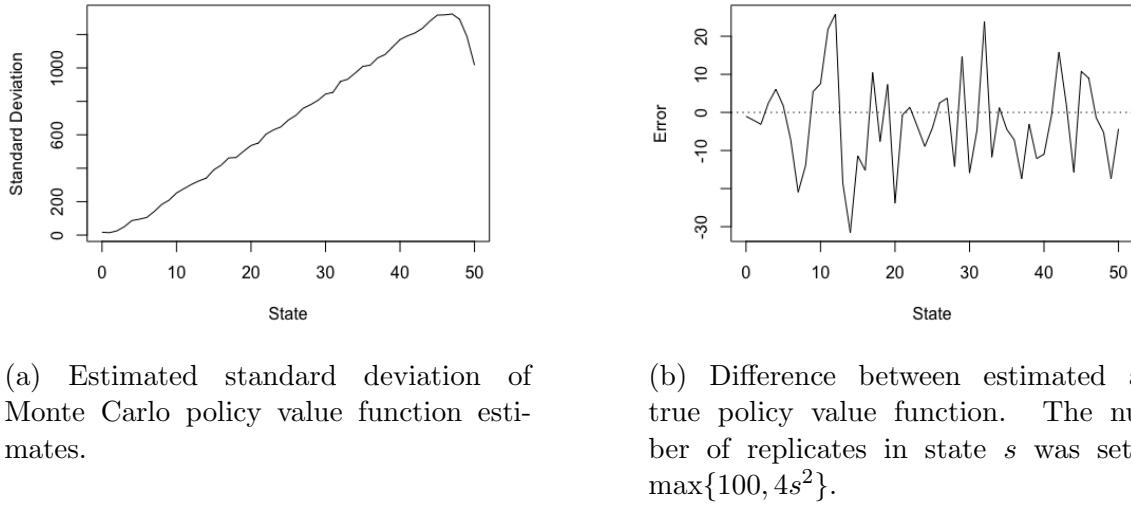


Figure 10.24: Some results for Monte Carlo estimation based on reward sequence truncation at $M = 125$.

primarily visits the low-numbered states⁶². Hence, estimates of value functions or state-action value functions using trajectory-based sampling will be highly variable for the infrequently visited, high-occupancy states. Therefore, it seems plausible that state-based sampling after each transition would improve the accuracy of $\text{TD}(\gamma)$ methods.

The following small study investigated the effect of learning rate, parameter of $\text{TD}(\gamma)$, trace type and policy on the accuracy of policy value function estimates. The study consisted of 40 replicates of 100,000 iterations for each of five learning rates (exponential, ratio $(150/(300 + n)$ and $15/(30 + n)$), STC, and Log), four values of γ , two trace types (replacement and accumulation) and two stationary policies (one that used action a_2 in every state and another that repeated the sequence (a_1, a_2, a_3) beginning in state 0 and continuing through state 50). The discount rate was set equal to $\lambda = 0.9$, states were generated randomly at each iteration and common random number seeds were used in each replicate. The index of the learning rate was the number of previous visits to a state.

Since the true policy value function could be found using exact methods for policy evaluation, replicates were summarized in terms of the RMSE of the policy value function estimates over all states. Figure 10.25 displays the main results graphically. Observe that:

1. Trace type and policy had insignificant effect on results and were not included in the figure.

⁶²It is left as an exercise to compute the stationary distribution under policies that use action a_2 or a_3 in all states.

2. The Log and ratio $(150/(300+n))$ learning rate generated very high RMSEs and were not included in the figure.
3. The ratio learning rate $\tau_n = 15/(30 + n)$ generated estimates with the smallest RMSEs. For this learning rate, $\gamma = 0$, corresponding to TD(0), gave the smallest RMSE (41.2). For the STC learning rate, $\gamma = 0.2$ gave the smallest RMSE.
4. Combined TD(0) and ratio estimates were most accurate in low-numbered states on an absolute level (Figure 10.26).

Figure 10.26 shows that the TD(0) policy value function estimates deviated from the true policy value function by at most 112. On a relative basis this error is at most 2.3%.

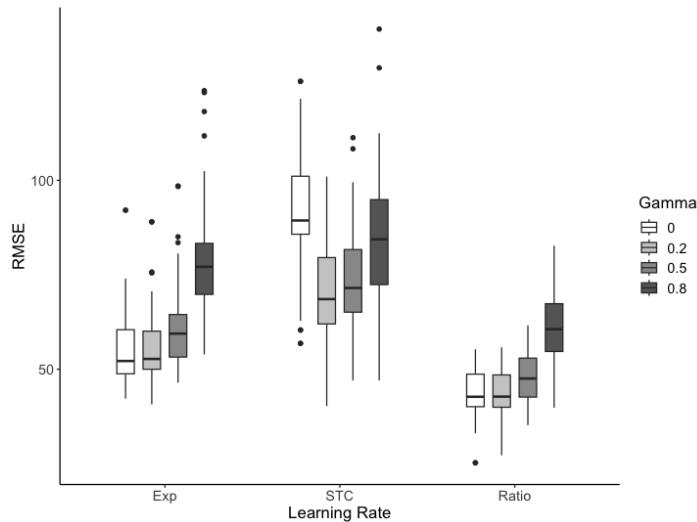
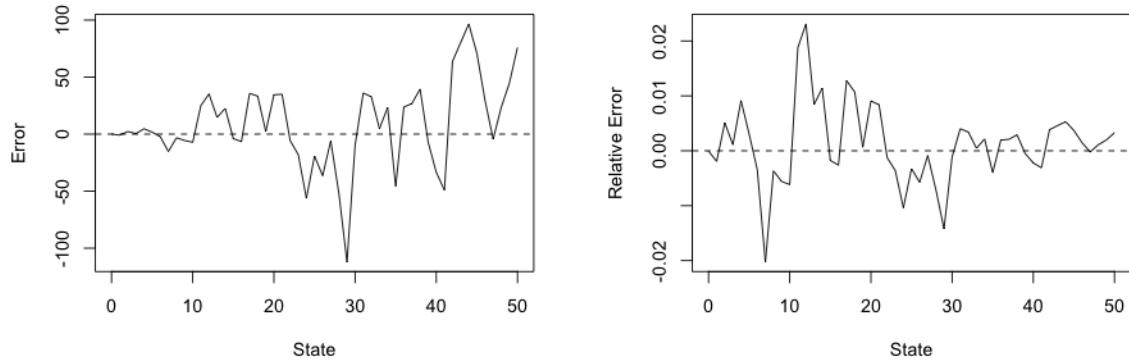


Figure 10.25: Graphical summary of the dependence of RMSE on learning rate and γ when using $\text{TD}(\gamma)$ for policy value function approximation for d^∞ with $d(s) = a_2$ for all $s \in S$. Each boxplot represents a summary over 40 replicates. Note the effect of trace and policy were insignificant and are not shown.

Comparison of $\text{TD}(\gamma)$ and Monte Carlo estimates

Table 10.13 compares two Monte Carlo estimates with the best $\text{TD}(\gamma)$ estimate on the basis of RMSE and the number of transitions evaluated to obtain a policy value function estimate for all states. Note that the RMSE for $\text{TD}(0)$ was the average over 40 replicates of length 100,000 but in practice one would only generate one such estimate. On the other hand the large number of replicates required to obtain Monte Carlo estimates was due to the need to obtain multiple replicates for each starting state. Thus it appears that the $\text{TD}(0)$ method is the computationally most efficient (at the expense of parameter tuning).



(a) Difference between estimated and true policy value function.
(b) Relative difference between estimated and true policy value function.

Figure 10.26: Error and relative error using TD(0) with learning rate $\tau_n = 15/(30+n)$ for estimating the policy value function of d^∞ where $d(s) = a_2$ for all $s \in S$ based on single randomly selected replicate.

Method	RMSE	Total number of transitions to estimate the policy value function
Monte Carlo - Truncation	12.3	$182,384 \times 125$
Monte Carlo - Geometric	103.7	$182,384 \times 11$
TD(0) with ratio learning	41.2	100,000

Table 10.13: Summary of value function calculations. Note 182,384 represents the total number of sequences generated when obtaining estimates.

10.9.3 Q-learning

This section explores the use of the Q-learning algorithm (Algorithm 10.11) and its SARSA variant. Since the optimal value function and policy can be computed using methods in Chapter 5, the quality of the estimates can be assessed by comparing the policy value function of the greedy policy⁶³ to the optimal value function on the basis of RMSE over all states. In addition, the greedy policy can be compared to the optimal policy on the basis of the percentage of states on which they agree.

State-based versus trajectory-based sampling

The first step in the analysis was a comparison of trajectory-based sampling and uniform state-based sampling in individual replicates. Results for a typical replicate are displayed in Figure 10.27. Figures 10.27a and 10.27c show the greedy policy (points)

⁶³Found by solving $(\mathbf{I} - \lambda \mathbf{P}_d)\mathbf{v} = \mathbf{r}_d$ where d^∞ denotes the greedy policy.

and the optimal policy (dashed line). Figures 10.27b and 10.27d show the difference in values (on different scales) of the greedy policy and the optimal policy.

Observe that the greedy policy derived using state-based sampling closely matched the optimal policy and achieved a value very close to it. On the other hand, the greedy policy derived using trajectory-based sampling had a similar structure and value to the optimal policy only in states 0 – 17.

The reason for such poor performance in high-occupancy states using trajectory-based sampling is that in 450,000 replicates, states 18 – 50 were visited infrequently: for example, states 30 – 50 were visited at most three times. Hence, in these states, the $q(s, a)$ estimates were extremely inaccurate. Theoretically, although the underlying Markov chain is regular, the limiting probabilities for high-occupancy states are extremely small making them rarely observed under trajectory-based sampling.

In practice, this limitation may not impact overall system performance since high-occupancy state are seldom encountered. However, this issue can be addressed by:

1. Comparing values using RMSE weighted by the limiting probabilities of the optimal policy⁶⁴.
2. Using state-based (or state-action based sampling) to ensure all states or state-action pairs are sampled sufficiently.

This issue is not limited to this queuing model. Similar limitations arise when using Q-learning with trajectory-based sampling in other process-type applications. Thus, where the objective is to find an optimal policy applicable in **all** states, state-based or state-action based sampling is preferred.

State-based sampling

The following results were based on an experiment that used 40 replicates of length 450,000 with random state-based sampling at each iteration. It compared five learning rate specifications (used for policy evaluation above), four choices (0.1, $150/(300 + n)$, $15/(30 + n)$, $0.7n^{-0.5}$) for ϵ_n and two estimation methods (Q-learning and SARSA).

Q-learning and SARSA produced equivalent results. The most accurate learning rate τ_n and ϵ_n -greedy specifications among those considered were

$$\tau_n = \frac{0.1}{1 + n^2/10^5} \quad \text{and} \quad \epsilon_n = \frac{150}{300 + n}. \quad (10.96)$$

For the learning rate, n represented the number of visits to a state-action pair and for ϵ_n , n represented the number of visits to a state. This combination generated the

⁶⁴Of course these probabilities would not be known in practice. Instead, one could assign weights based on observed empirical frequencies or the importance of states. To use a sports analogy, training should focus on skills for situations most likely to be encountered in competition and limited for unlikely scenarios.

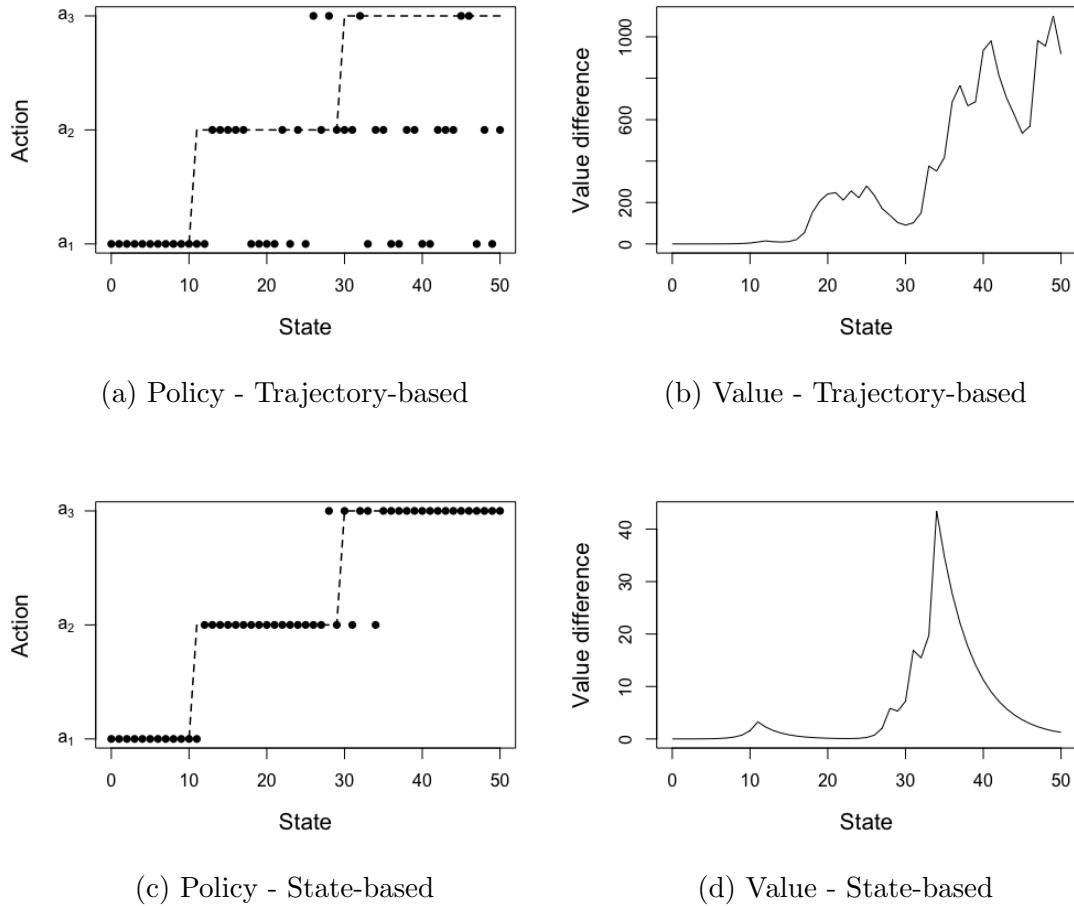


Figure 10.27: Comparison of policies and values using Q-learning with trajectory-based and state-based sampling. Note the difference in the scales in Figures 10.27b and 10.27d.

most accurate estimates of the state-action value function and the highest percentage of states in which the greedy policy was optimal over 40 replicates.

Results in Figures 10.27c and 10.27d were typical for this specification. Using Q-learning, the state-action value function estimates were extremely accurate. The mean RMSE of the state-action value function (across replicates) was 19.8 with standard deviation of 11.1, which is small relative to the range of the $q(s, a)$ values spanning 70 to 22,600. Greedy policies deviated from the optimal policy in only a few states. On average, 90.6% of actions generated by the greedy policy agreed with the optimal policy, with a standard deviation of 3.7%.

10.9.4 Policy iteration type algorithms

This section describes the use of hybrid policy iteration (Algorithm 10.14) and online policy iteration (Algorithm 10.15). Recall hybrid policy iteration estimates a policy value function using simulation and then uses the model to compute the q -function, while online policy iteration estimates q -functions directly. Implementation choices⁶⁵ include:

Policy evaluation method: TD(0), TD(γ), or Monte Carlo with truncated discounted rewards or geometric stopping times;

Algorithmic parameters: learning rates, ϵ -greedy exploration settings, and number of iterations;

Sampling method: state-based or trajectory-based updates;

Improvement step: exact vs. estimated q -functions;

Policy value function estimates: smoothed vs. unsmoothed value functions;

q -function estimates: smoothed vs. unsmoothed state-action value functions;

Smoothing technique: Parametric model used for smoothing;

Stopping criterion: Extent of agreement of policies at successive iterations.

Configurations were compared on the basis of whether the algorithms terminated or cycled, how similar the identified policy was to the optimal policy, and how well the estimated value function approximated the optimal value function.

To obtain convergence in online policy iteration, it was necessary to smooth $q(s, a)$ as a function of s prior to greedy improvement. Moreover, smoothing could potentially enhance convergence of hybrid policy iteration. A convenient choice for this purpose was a *quadratic B-spline*⁶⁶. Chapter 11 provides a systematic analysis for combining simulation and function approximation.

Hybrid policy iteration: Implementation

Algorithm 10.14 combines policy value function estimation with greedy improvement. Given a policy value function estimate $\hat{v}(s)$, it uses the model to evaluate (10.86) according to

$$q(s, a) = \sum_{j \in S} p(j|s, a)(r(s, a, j) + \hat{v}(j)) \quad (10.97)$$

⁶⁵Not all apply to each method.

⁶⁶B-splines are linear combinations of polynomials that flexibly approximate nonlinear functions. In R, they are specified with the function `bs()` and fit with function `lm()`. To avoid pre-specifying knot locations, our implementations here used the specification `degree=2` and `df=2`.

$$= \begin{cases} 5a^3 + (1 - q)\hat{v}(0) + q\hat{v}(1) & s = 0 \\ s^2 + 5a^3 + a\hat{v}(s-1) + (1 - a - q)\hat{v}(s) + q\hat{v}(s+1) & s \in \{1, 2, \dots, 49\} \\ 50^2 + 5a^3 + a\hat{v}(49) + (1 - a)\hat{v}(50) & s = 50. \end{cases}$$

Note that $r(s, a, j)$ is independent of j .

Policy evaluation used TD(0) and starting-state Monte Carlo with truncation and geometric stopping times, and both smoothed and unsmoothed q -functions. For each of these six combinations, 40 replicates with common seeds across each configuration within a replicate were evaluated.

Details of the implementation follow:

1. TD(0) parameters were specified as $\tau_n = 15/(30 + n)$ and $N = 100,000$. States were resampled after each transition.
2. Each replicate of Monte Carlo evaluated all starting states. Both truncated and geometric stopping time versions used $\max(100, 3s^2)$ iterations in state s . Truncation was set at $M = 100^{67}$.
3. The stopping criterion was “stop when $|\Gamma| \leq 2$ ” where Γ denotes⁶⁸ the set of states on which the policy at two successive iterates differ. More restrictive conditions led to cycling.

Hybrid policy iteration: Results

Results of the comparisons are summarized in Table 10.14. Without smoothing, hybrid policy iteration with TD(0) and Monte Carlo with truncation achieved the stopping criteria in approximately six improvement steps, with identical ranges of 2 to 18 steps. The algorithm did not converge when Monte Carlo estimates were based on geometric stopping times.

Average RMSE⁶⁹ for both approaches were comparable but values of greedy policies based on Monte Carlo replicates were less variable. Hybrid policy iteration with TD(0) identified the optimal policy in 4 of 40 cases while the Monte Carlo variant with truncation identified the optimal policy in 5 of 40 cases. However, when q -function smoothing was applied, both methods reduced the number of improvement steps to two and almost always identified the optimal policy.

Since both TD(0) and Monte Carlo with truncation accurately estimated the policy value function, hybrid policy iteration using these estimates achieved the stopping criterion and produced accurate estimates without smoothing. Given the greater computational effort to implement Monte Carlo with truncation, TD(0) is the preferred evaluation choice if hybrid policy iteration is to be used without smoothing. When

⁶⁷These values were modified from Section 10.9.1 to speed up execution. Nonetheless, Monte Carlo with truncation was extremely time consuming (several hours on a Macbook Pro M3).

⁶⁸See step 2(d) in the algorithm.

⁶⁹Between the greedy policy and the optimal policy.

Evaluation Method	Smoothing q -function	Average Iterations	Range Iterations	Average RMSE	Range RMSE
TD(0)	No	5.58	(2, 18)	15.48	(0, 71.07)
	Yes	2	2	0	0
Monte Carlo (truncation)	No	5.85	(2, 18)	17.0	(0, 39.8)
	Yes	2	2	0	0
Monte Carlo (geometric stopping)	No	Did not converge	-	-	-
	Yes	2	2	0.31	(0, 1.52)

Table 10.14: Comparison of results of hybrid policy improvement methods based on 40 replications with common random number seeds.

smoothing is applied, all evaluation methods, including Monte Carlo with geometric stopping, perform similarly.

Online policy iteration: Implementation

Algorithm 10.15 was applied using $N = 450,000$, $\nu = 2$, learning rates $\tau_n = 0.1/(1 + n^2/10^5)$ and exploration parameters $\epsilon_n = 1,500/(3,000 + n)$. State transitions were modeled using random state-based sampling and actions were sampled using a ϵ_n -tweaked decision rule d_{ϵ_n} .

Both off-policy and on-policy updates for the state-action value function were examined. The on-policy update used

$$q(s, a) = q(s, a) + \tau_n(r(s, a, a') + \lambda q(s, a') - q(s, a)), \quad (10.98)$$

where a' is chosen using the ϵ_n -tweaked decision rule d_{ϵ_n} . The off-policy update uses (10.93).

Additionally, the effect of q -function smoothing was investigated. A total of 40 replicates were evaluated using common random numbers in each.

Online policy iteration: Results

Table 10.15 summarizes the results. Results for unsmoothed q -functions are omitted because decision rules generated using both on-policy and off-policy updates failed to achieve the proposed stopping criterion, even after 25 evaluation-improvement cycles. In fact, between 20 and 30 actions changed at each cycle.

Results in the above table suggests that using d for updating the q -function provides more reliable performance. This makes sense because $q(s, a)$ reflects the value of using action a in state s for one period and then following the policy d^∞ , not the exploratory policy based on d_{ϵ_n} .

Thus, when applying online policy iteration, smoothing appears necessary to obtain convergence. Even in small problems, this suggests the potential value of function approximation.

Decision rule used in (10.93)	Smoothing q -function	Average Iterations	Range Iterations	Average RMSE	Range RMSE
d	Yes	3.95	(2, 10)	15.02	(0, 68.3)
d_{ϵ_n}	Yes	4.33	(2, 10)	12.81	(0, 58.4)

Table 10.15: Summary of results based on 40 replicates of Algorithm 10.15. In two replicates, the variant using d_{ϵ_n} in the update terminated after two iterations with sub-optimal policies ($\text{RMSE} = 1523$). These two outliers are excluded from the reported RMSE values for d_{ϵ_n} .

10.9.5 Computational summary: Discounted model

Analysis of the discounted queuing control model suggests:

1. Convergence and accuracy of algorithms was highly sensitive to learning rate and, when appropriate, to the exploration parameter ϵ_n . The specification of n (state-action pair visits for learning rate or state visits for exploration rate) impacted convergence.
2. TD(0) provided reliable estimates of the policy value function. TD(γ) estimates were less accurate.
3. Starting-state Monte Carlo provided accurate policy value function estimates but required more replicates than TD(0) to obtain comparable precision. Methods based on truncated discounted rewards were more accurate than those using geometric stopping times.
4. Optimization methods required state-based sampling to obtain near optimal policies. Trajectory-based methods sampled some parts of the state space too infrequently.
5. Using state-based sampling, Q-learning produced policies that differed from the optimal policies in a few states. Although smoothing the q -function was not necessary to obtain convergence, it likely would have generated monotone policies.
6. Policy iteration algorithms behaved best with a stopping criterion of the form “Stop when the decision rule at successive iterates differs in at most two states.” Tighter stopping rules often led to cycling.
7. Hybrid policy improvement with state-based sampling terminated without q -function smoothing. However, q -function smoothing resulted in faster convergence to the optimal policy.
8. Online policy iteration required q -function smoothing for convergence. The off-policy variant as stated in Algorithm 10.15 converged more reliably.

These findings suggest that when a model is available, hybrid policy iteration with TD(0) or truncated Monte Carlo policy value function estimation were most reliable and precise. However, the Monte Carlo estimates required considerably more computation. In the absence of a model, Q-learning was easiest to apply and provided good results without smoothing the q -function. Simulated policy iteration required q -function smoothing to ensure convergence. In contrast to the two-state example above, methods were most reliable when states were randomly sampled at each transition as opposed to trajectory-based sampling.

Smoothing the q -function enhanced convergence and produced monotone policies. Smoothing involved two-steps at each iteration, estimating the q -function for all state-action pairs and then approximating it by a smooth function prior to greedy action selection. This was possible in a small model, the next chapter investigates incorporating function approximation directly in the optimization algorithm enabling generalization to larger models.

10.9.6 Average reward queuing control

This section investigates the application of Algorithms 10.12 and 10.13 to an average reward version of the queuing service rate control model. Preliminary analyses suggest that the only effective approach was Relative Q-learning with state-based sampling.

Q-learning (Algorithm 10.12) consistently identified suboptimal policies that deviated significantly from the optimal policy under any state sampling regime. The key issue was the inaccuracy of estimates of g , which resulted in inaccurate estimates of state-action value functions. Relative Q-learning with trajectory-based sampling also failed to identify good policies, primarily because of infrequent visits to high-occupancy states.

Implementation and results

A discussion of the application of Relative Q-learning with state-based sampling follows. The implementation specified $f(q) = q(0, a_1)$ meaning this quantity was used to estimate the gain in (10.83). Computations compared configurations with learning rates $100/(200 + n)$ and $0.2/(1 + 10^{-6}n^2)$ paired with either ϵ -greedy exploration with $\epsilon_n = 100/(200 + n)$ or softmax exploration⁷⁰ with $\eta = 1/2,000$.

Each configuration was evaluated over 40 replicates with common random number seeds, simulating $N = 300,000$ total iterates (roughly 6,000 per state) per replicate. Methods were assessed on the accuracy of estimates of the optimal average reward g^* and the number of states in which the greedy policy differed from the optimal policy.

Table 10.16 summarizes the results. Clearly ϵ -greedy exploration was preferable to softmax exploration, the latter being highly sensitive to parameter choice. There was

⁷⁰Results were sensitive to specification of η ; choosing it too small resulted in limited search across actions.

a small benefit to using the STC learning rate as opposed to the ratio learning rate. As the table shows, its estimate of the optimal gain was more accurate.

Learning Rate	Exploration	Non-optimal Actions	Average Reward Difference
Ratio	ϵ -greedy	2.18 (1.31)	2.41 (1.70)
	Softmax	11.40 (3.33)	4.71 (2.73)
STC	ϵ -greedy	2.18 (1.20)	0.95 (0.84)
	Softmax	4.85 (2.19)	2.96 (1.49)

Table 10.16: Accuracy of Relative Q-learning estimates as a function of learning rate and exploration method. Entries are means (standard deviations) over 40 replicates.

Figure 10.28 compares the greedy policy to the optimal policy for a typical selected replicate. Observe that the resulting policy closely approximates the optimal policy, differing from it in only three states. In this replicate, the estimate of $g^* = 19.42$ was 16.58.

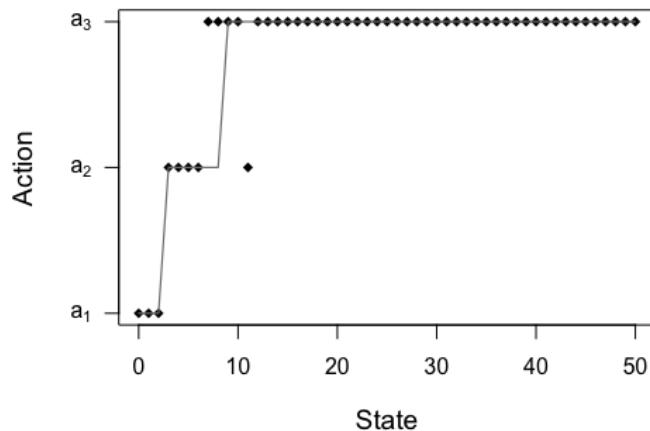


Figure 10.28: Plot of greedy decision rule (diamonds) and average optimal decision rule (line) for a randomly selected replicate using STC learning rate and ϵ -greedy exploration.

It was encouraging to find that Relative Q-learning, without any additional smoothing, identified policies that are close to optimal. The implementation used state-based sampling; actions were chosen by exploration. It is possible that state-action based sampling would result in even more accurate results.

10.10 Conclusion

The foundational methods in this chapter provide a “toolbox” of potential approaches (Monte Carlo, temporal differencing, TD(γ) for policy evaluation; Q-learning, SARSA and policy iteration type algorithms for optimization) to apply in simulated or process-based environments suitable for tabular representations. However, applying them requires considerable trial-and error tuning of learning rates, exploration parameters, sampling methods and run lengths to produce useful results. Moreover, appropriate specifications for a given application are not obvious *a priori*.

10.11 Technical appendix

10.11.1 Choosing good estimators

Policy evaluation methods in this chapter are concerned with estimates of an unknown quantity based on a sequence of observations or replicates of an experiment. Estimates may be computed offline using the whole data set, for example a sample mean, or online, updated sequentially using stochastic approximation algorithms described in Appendix 10.11.2. Standard statistical concepts such as bias, variance and mean squared error are useful for assessing the quality of an estimator.

Many examples in this chapter require choosing among estimates such as those represented by the different lines in Figure 10.29. They may be generated by different algorithms or from a single algorithm under different parameter specifications. In the figure, the light gray estimate is the least variable but fails to converge to the true value given by the horizontal black line, in other words, it is biased. The dark gray and black estimates both oscillate around the true value but the black is less variable and therefore preferred. This illustrates that choosing an estimator involves trading-off bias and variability as described below.

First, some caveats are in order.

1. The estimates in Figure 10.29 are based on a single replicate of a simulation using a common random number seed. Results may differ with different random sequences.
2. In practice, a single estimation method and a pre-specified number of iterates will be specified.
3. Results are sensitive to run lengths. If sampling stopped after approximately 4,200 steps, one might conclude that the estimate corresponding to the dark grey line gave the best approximation of the unknown quantity.

Visualizing results across different run lengths, as in this figure, facilitates selection of estimators that produce good estimates over a range of run lengths.

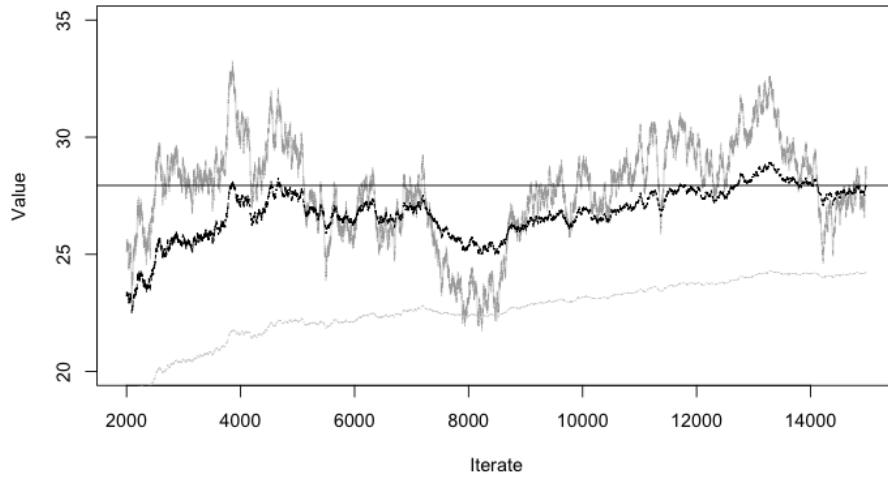


Figure 10.29: Three sequences of estimates of a quantity based on a single replicate of an online algorithm with run length of 15,000. The horizontal line represents the true value which is unknown in practice but known in the illustrative experiments used in many of the examples in Chapters 10 and 11.

RMSE for point estimation

Consider an experiment with K replicates that generates estimates $y^k, k = 1, 2, \dots, K$ of a fixed scalar C . For example, y^k may be the estimate of a policy value function in a specified state obtained from a single Monte Carlo replicate.

The quality of the estimate is often assessed using the *mean squared error (MSE)* defined by

$$\text{MSE} := \frac{1}{K} \sum_{k=1}^K (y^k - C)^2.$$

Since the MSE is expressed in squared units, the *root mean squared error (RMSE)*, defined by

$$\text{RMSE} = \sqrt{\text{MSE}},$$

is preferred because it is on the same scale as the estimates and hence provides a more interpretable measure of accuracy.

The MSE is related to two other statistical measures, the *sample bias* given by

$$\text{Bias} := |\bar{y} - C|$$

and the *sample variance* given by

$$\text{Variance} := \frac{1}{K-1} \sum_{k=1}^K (y^k - \bar{y})^2,$$

where \bar{y} denotes the sample mean. Note that the variance and MSE measure deviations from different quantities, the sample mean and true value, respectively. When K replaces $K-1$ in the variance, this quantity is referred to as the *population variance*.

These quantities are related through the expression:

$$\text{MSE} = \left(\frac{K-1}{K} \right) \text{Variance} + \text{Bias}^2. \quad (10.99)$$

To see this, note that

$$\begin{aligned} \sum_{k=1}^K (y^k - C)^2 &= \sum_{k=1}^K (y^k - \bar{y} + \bar{y} - C)^2 \\ &= \sum_{k=1}^K (y^k - \bar{y})^2 + 2 \sum_{k=1}^K (y^k - \bar{y})(\bar{y} - C) + \sum_{k=1}^K (\bar{Y} - C)^2 \\ &= \sum_{k=1}^K (y^k - \bar{Y})^2 + K(\bar{Y} - C)^2, \end{aligned}$$

since the cross product term sums to zero, the result follows by dividing both sides by K .

Hence the mean squared error contains both a variance and a bias component. If the population variance replaces the sample variance in (10.99) the fraction disappears so that this relationship becomes:

$$\text{MSE} = \text{Population Variance} + \text{Bias}^2. \quad (10.100)$$

Many conclusions in examples will be based on the RMSE, so the breakdown in either (10.99) or (10.100) is intended primarily for interpretability of this concept. It is referred to as the *bias-variance tradeoff*.

Table 10.17 provides summary measures⁷¹ for the estimators represented in Figure 10.29. Observe that the estimator represented by the light grey line has small

⁷¹This example does not fall exactly within the above framework, since in it, y^k represents the k -th value in a single replicate of length 15,000 for a single estimator. To interpret results, assume that statistical measures are computed after obtaining all 15,000 values so \bar{y} is the overall mean value for that estimator.

variance and high bias, the estimator represented in dark grey has high variance and low bias, and that in black has both small variance and small bias. Moreover, the RMSE of the estimator in black is the smallest. On this basis one would prefer the estimator corresponding to the black line, consistent with the conclusions based on inspecting the figure directly.

Estimator	RMSE	MSE	Population Variance	Bias ²
Black	1.59	2.54	1.17	1.37
Dark-grey	2.09	4.37	4.33	0.04
Light-grey	5.47	29.93	1.58	28.35

Table 10.17: Statistical summaries of estimators in Figure 10.29.

RMSE for value function estimation

The RMSE is frequently used in a different context in this chapter, namely for assessing the accuracy of different approaches for estimating policy or optimal value functions. Suppose $\hat{v}(s)$ is an estimator of a known function $v(s)$ defined over a finite state space S . The RMSE of the estimate is defined by:

$$\text{RMSE}(\hat{\mathbf{v}}) := \sqrt{\frac{1}{|S|} \sum_{s \in S} (\hat{v}(s) - v(s))^2}. \quad (10.101)$$

Studies of simulation-based algorithms, such as in Section 10.9, often generate K replicates $\hat{v}^1(s), \dots, \hat{v}^K(s)$ by varying the random number seed. In such settings, algorithmic performance is often assessed using summary statistics such as the mean, standard deviation and range of the sequence⁷² of RMSE values $\text{RMSE}(\hat{\mathbf{v}}^1), \dots, \text{RMSE}(\hat{\mathbf{v}}^K)$ displayed in tables or graphically using boxplots.

Of course when the primary objective is to find effective policies, as discussed in Section 10.6.2, precise estimation of value or state-action value functions may be of secondary importance. Algorithmic comparisons should be based on how close policies and their theoretical or simulated policy value functions are to optimal.

Weighted root mean squared error

Often it makes sense to replace RMSE by a *weighted RMSE*, especially when:

1. evaluating value functions which differ by orders of magnitude over S , or
2. in trajectory-based simulations where subsets of states are visited infrequently so that values in these states are not accurately estimated.

⁷²Recall \mathbf{v} represents a vector of values over states.

Moreover, a related concept, the weighted supremum norm is sometimes used to derive theoretical properties of estimators, especially when S is not finite.

The weighted RMSE of an estimator $\hat{\mathbf{v}}$ of a vector \mathbf{v} is defined by

$$\text{wRMSE}(\hat{\mathbf{v}}) := \sqrt{\sum_{s \in S} w(s)(\hat{v}(s) - v(s))^2}, \quad (10.102)$$

where $w(s) \geq 0$ for $s \in S$. Note setting $w(s) = 1/|S|$ yields the unweighted RMSE.

Suppose $w(s)$ is chosen to be the *stationary distribution* of the Markov chain corresponding to a policy being evaluated or that identified by an algorithm when optimizing. Such a measure better represents the accuracy (or potential loss of optimality) that will be observed in practice when implementing the policy. Of course, this distribution will not be known in optimization problems. However, it can be calculated in experimental settings in which the optimal policy is known.

Alternative choices for weights include the reciprocal of the variance of $\hat{v}(s)$ over multiple replicates of an experiment or an exponential or polynomial growth function.

10.11.2 Stochastic approximation

A new type of approximation process called “stochastic approximation” will play an important role in multistage decision processes⁷³.

Richard P. Bellman, Mathematician and father of dynamic programming,
1920-1984.

This prescient remark by Bellman underscores our decision to include this lengthy appendix on stochastic approximation. As anticipated by Bellman, stochastic approximation underlies the temporal differencing, Q-learning and policy gradient methods described in this chapter and in Chapter 11.

Stochastic approximation originates with the seminal paper [Robbins and Monroe 1951]⁷⁴, which developed an online algorithm for finding the root of a function based on noisy observations of the function value at selected points. It has become especially important for analyzing simulation data because it avoids storing large amounts of data.

The Robbins-Monro procedure

Stochastic approximation is an *online* method for finding an x^* that solves

$$E[Y|X = x] = 0.$$

⁷³Bellman [1963].

⁷⁴As an aside, Herbert Robbins was the PhD advisor of Cy Derman, who was the PhD advisor of Arthur F. Veinott Jr., who was the PhD advisor of author MLP.

It uses samples $y^n, n = 1, 2, \dots$ from a distribution with conditional mean $f(x) := E[Y|X = x]$ as the basis of an estimator. For example, the sequence y^n may be generated by $y^n = f(x^n) + \epsilon_n$ where ϵ_n denotes a random error term with mean 0 and the sequence $x^n, n = 1, 2, \dots$ is chosen in some way.

A Monte Carlo approach would generate multiple samples at different values of x and use them to approximate $f(x)$ and estimate where it equals 0. [Robbins and Monro \[1951\]](#) proposed something different. Their procedure generates a sequence $x^n, n = 1, 2, \dots$ that represents both estimates of x^* and points at which to apply a recursion of the form:

The Robbins-Monro recursion:

$$x^{n+1} = x^n - \tau_n y^n, \quad (10.103)$$

where y^n denotes an observation from a distribution with conditional mean $E[Y|X = x^n]$ and τ_n , referred to as the *step-size* or *learning rate*, represents a sequence of non-negative constants converging to 0.

The intuition underlying (10.103) (see Figure 10.30) is that when y^n is greater than 0, x^{n+1} will be *corrected* downward from x^n and conversely when y^n is less than 0, x^{n+1} will be corrected upward from x^n . Moreover, when y^n is close to zero the correction will be smaller as will be the case for large n when τ_n is smaller.

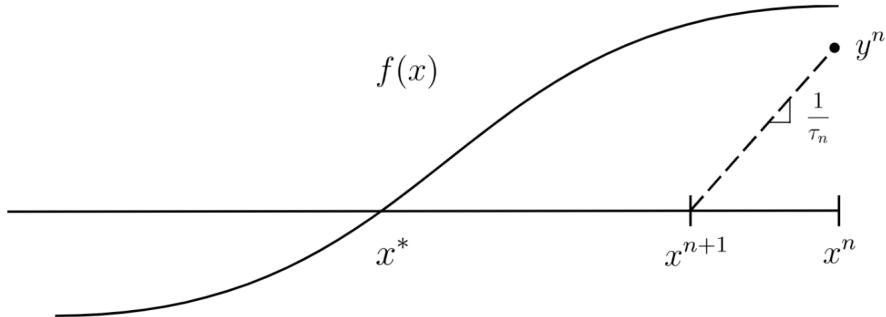


Figure 10.30: Geometric representation of Robbins-Monro recursion for solving $f(x) = 0$. Since $y^n/(x^n - x^{n+1}) = 1/\tau_n$, $x^{n+1} = x^n - \tau_n y^n$.

This recursion may be viewed as a stochastic generalization of Newton's method. When $f(x^n)$ is observable (and differentiable), Newton's method finds its zero by iter-

ating

$$x^{n+1} = x^n - (f'(x^n))^{-1} f(x^n). \quad (10.104)$$

Equation (10.103) replaces the reciprocal of the derivative by the constant τ_n and $f(x^n)$ by an observation sampled from a distribution with mean $f(x^n)$.

The following theorem (stated without proof) describes the limiting behavior of the sequence of iterates generated by (10.103).

Theorem 10.3. Suppose $f(x) = E [Y|X = x]$,

1. $f(x)$ is monotone,
2. $E [(Y - f(X))^2|X = x] < M < \infty$ for some constant M , and
3. $|f(x)| \leq d|x| + c$ for finite positive constants c and d ,
4. the sequence $\tau_n, n = 1, 2, \dots$ satisfies

$$\tau_n \geq 0, \quad \sum_{n=1}^{\infty} \tau_n = \infty \quad \text{and} \quad \sum_{n=1}^{\infty} \tau_n^2 < \infty. \quad (10.105)$$

Then for any x^1 , the sequence

$$x^{n+1} = x^n - \tau_n y^n$$

converges almost surely to x^* where $f(x^*) = 0$.

Some comments about this theorem follow:

1. In most of the applications of this result in this book, x^n will represent an estimate of a parameter (weight) or vector of parameters and be denoted by β^n .
2. The theorem provides a stronger version of the result in Robbins and Monroe [1951] under slightly different conditions on $f(x)$. It is due to Blum [1954]. It establishes almost sure convergence⁷⁵ as opposed to L^2 convergence. The key conditions on the step-size in (10.105) remain the same.
3. Proofs of this theorem and its variants are beyond the scope of this book. They use subtle arguments and advanced probabilistic concepts. See the Bibliographic Remarks section for references.
4. Much of the literature refers to (10.105) as “the usual conditions on τ_n ”.

⁷⁵Also called convergence with probability one.

5. The conditions on τ_n in Theorem 10.3 are satisfied when $\tau_n = k/n^b$ for $b \in (.5, 1]$, $\tau_n = c/(d+n)$ where c and d are positive constants and $\tau_n = \log(n+1)/n$. Note the choice $\tau_n = 1/n$ satisfies these conditions.
6. Note that in addition to specifying the correction term to the current estimate of the solution of $f(x) = 0$, the Robbins-Monro recursion specifies at which value of x^n to generate a new observation.
7. This theorem applies to a univariate method; a vector version is also valid. It will be discussed in the context of gradient methods below.
8. Equation (10.103) can be rewritten as

$$x^{n+1} = (1 - \tau_n)x^n + \tau_n(x^n - y^n)$$

This representation writes x^{n+1} as a weighted average of the previous estimate x^n and its difference with the new observation y^n . This is referred to as *exponential smoothing* in the forecasting literature.

Example 10.12 below shows that in agreement with theory, when $\sum_{n=1}^{\infty} \tau_n < \infty$, the step-sizes converge to 0 too quickly leading to convergence to an incorrect value. Moreover, when $\sum_{n=1}^{\infty} \tau_n^2 = \infty$, variability is not damped out so that iterates oscillate rapidly. When the conditions on the sums of τ_n and τ_n^2 are satisfied, the iterates converge.

Example 10.12. Impact of τ_n on stochastic approximation estimates.

This example shows how the conditions in (10.105) impact convergence of the stochastic approximation iterates. Consider the problem of finding the fifth root of 2. Put into the stochastic approximation framework, it can be expressed as solving

$$f(x) := x^{1/5} - 2 = 0$$

based on realizations of $f(x) + \epsilon$ where ϵ is normally distributed with mean 0 and standard deviation 1.

The example compares estimates based on four choices for τ_n corresponding to cases when (10.105) hold and are violated in two different ways. Each estimate is based on 50,000 iterates, common random number sequences for the four choices of τ_n and $x_0 = 20$. Of course $f(32) = 0$ so that $x^* = 32$.

Clearly conditions 1 and 3 of Theorem 10.3 hold and since $\text{var}(\epsilon) = 1$, condition 2 holds. Choosing $\tau_n = 0.9n^{-0.5}$ and $\tau_n = 100/(400+n)$ correspond to the cases in which both summation conditions in (10.105) are satisfied, choosing $\tau_n = n^{-1.000001}$ corresponds to $\sum_{n=1}^{\infty} \tau_n < \infty$ and $\sum_{n=1}^{\infty} \tau_n^2 < \infty$ and choosing $\tau_n = n^{-0.25}$ corresponds to $\sum_{n=1}^{\infty} \tau_n = \infty$ and $\sum_{n=1}^{\infty} \tau_n^2 = \infty$. Note that $\sum_{n=1}^{\infty} \tau_n < \infty$ and $\sum_{n=1}^{\infty} \tau_n^2 = \infty$ is impossible since for $\tau_n > 0$ small, $\tau_n^2 < \tau_n$. (As an exercise,

verify that these sequences satisfy the indicated conditions.)

Figure 10.31 displays the sequence x^n corresponding to each of the four step-size sequences. The figure legend identifies the sequences. The figure shows that:

1. sequences corresponding to $\tau_n = 0.9n^{-0.5}$ and $\tau_n = 100/(400 + n)$ converge to x^* with that based on the ratio more stable,
2. the sequence corresponding to $\tau_n = n^{-1.0000001}$, converges but to the wrong value, and
3. the sequence corresponding to $\tau_n = n^{-0.25}$ oscillates around the target value $x^* = 32$.

Consequently, different violations of the assumptions on τ_n have different impacts on the sequence of iterates x^n . Moreover, the ratio step-size parameter choice provides the most stable estimates.

The following example illustrates the use of stochastic approximation in a non-standard application.

Example 10.13. Using stochastic approximation to find a percentile

Since the p -th, $0 \leq p \leq 100$, percentile of the distribution of a random variable Y can be represented as the solution x^* of $E[I_{\{Y \leq x\}}] = p/100$, (10.103) can be used to find this percentile. To do so, specify x , sample y from its distribution and compute $I_{\{y \leq x\}}$.

For example, consider the problem of finding the 75th percentile of a standard Cauchy distribution.^a In this case, (10.103) becomes

$$x^{n+1} = x^n - \tau_n (I_{\{y^n \leq x^n\}} - 0.75).$$

Note that the quantity $I_{\{y^n \leq x^n\}} - 0.75$ only takes on the values 0.25 and -0.75 .

Stochastic approximation is applied to estimate the 10th, 20th, 50th, 75th and 85th percentile of the Cauchy distribution with $\tau_n = 20/(200 + n)$, $\tau_n = n^{-0.75}$ and $\tau_n = n^{-1}$. A total of 40 replicates of 10,000 iterates were generated using a common random number seed for each replicate and an initial value $x^1 = 0$.

Estimates were compared on the basis of within-replicate RMSE values (after deleting first 200 estimates) and visual inspection of plots. Table 10.18 summarizes the results. Observe that estimates using $\tau_n = n^{-1}$ (the sample average) produced the least accurate estimates for all percentiles so clearly it is advantageous to explore using stochastic approximation with learning rates other than $\tau_n = n^{-1}$. The reason that the running mean performed so poorly is that τ_n

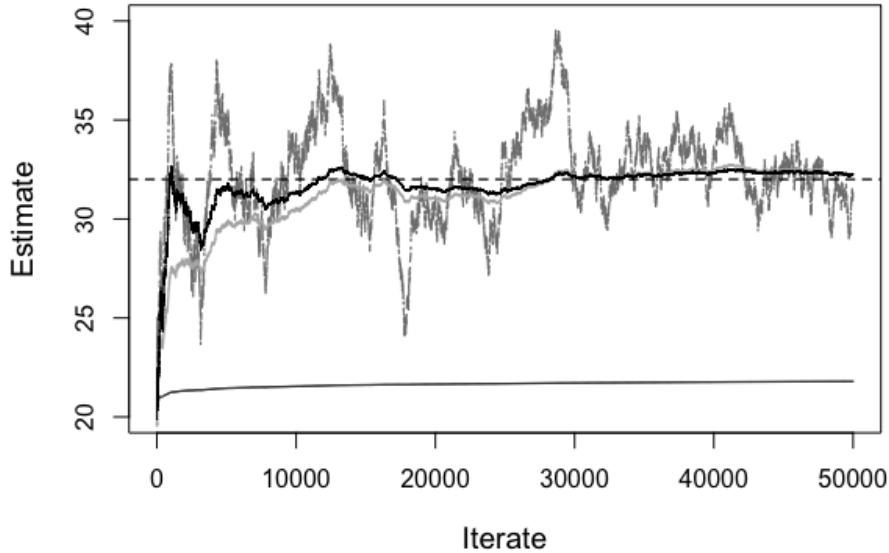


Figure 10.31: Graphical representation of sequence of estimates of x^* in Example 10.12. The horizontal dashed line gives the solution $x^* = 32$, the dark grey line, the iterates when $\tau_n = n^{-0.25}$, the line near the bottom of the figure corresponds to $\tau_n = n^{-1.0000001}$ and the grey and black lines that stabilize around 32 correspond to $\tau_n = 0.9n^{-0.5}$ and $\tau_n = 100/(400 + n)$, respectively.

decreased too quickly to allow learning.

The quality of estimates declined the further the percentile was from the mean 0. When accounting for both the mean and standard deviation, the ratio learning rate produced the most accurate estimates, especially in the tails of the distribution. Figure 10.32 shows a typical replicate comparing the three learning rates, substantiating the conclusions above.

^aRecall that the Cauchy distribution has infinite variance so large values occur frequently. A standard Cauchy distribution has location parameter 0 and scale parameter 1.

10.11.3 Stochastic approximation and optimization

Stochastic approximation plays a key role in optimization where the objective is to minimize an expected loss function. It serves as a foundational concept for the online learning algorithms in this chapter and the next as follows.

Consider the problem of finding the value that minimizes a real-valued function of

Percentile	$\tau_n = \frac{20}{200+n}$	$\tau_n = n^{-0.75}$	$\tau_n = \frac{1}{n}$
10	3.43 (5.06)	5.02 (6.60)	17.40 (28.91)
20	0.64 (0.36)	0.57 (0.68)	5.58 (7.81)
50	0.29 (0.08)	0.13 (0.07)	0.72 (0.77)
75	0.45 (0.23)	0.38 (0.39)	5.43 (9.30)
85	1.08 (0.97)	3.82 (8.34)	12.42 (22.16)

Table 10.18: Mean (standard deviation) of within-replicate RMSE over 40 replicates of stochastic approximation estimates of percentiles of a Cauchy distribution using three learning rates.

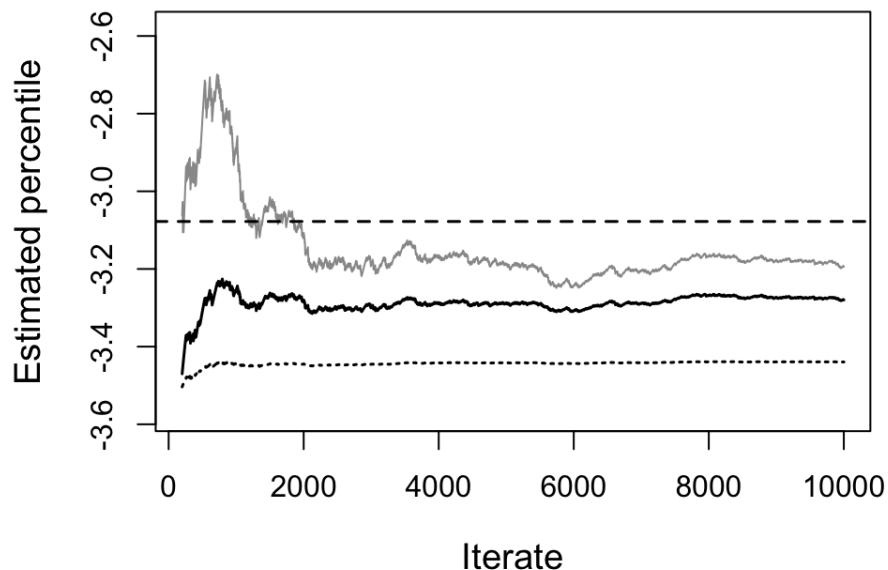


Figure 10.32: Plot of a single replicate of estimates for three learning rates of the 10th percentile of a Cauchy distribution in Example 10.13. The grey line corresponds to $\tau_n = 20/(200 + n)$, the black line to $\tau_n = n^{-0.75}$ and the dotted line to $\tau_n = n^{-1}$. The horizontal line represents the true value of the 10th percentile.

a vector of variables β , $G(\beta)$. That is

$$\beta^* \in \arg \min_{\beta} G(\beta). \quad (10.106)$$

Gradient descent methods achieve this by solving the equation

$$\nabla_{\beta} G(\beta) = 0, \quad (10.107)$$

where $\nabla_{\beta} G(\beta)$ denotes the gradient of $G(\beta)$. Implicit is the assumption that $G(\beta)$ is differentiable with respect to each component of β . Under mild conditions, this minimum is unique but it's possible that there may be multiple minima.

Gradient descent is an iterative procedure that chooses β , observes $G(\beta)$, evaluates its gradient at β and updates β according to the recursion

$$\beta' \leftarrow \beta - \tau \nabla_{\beta} G(\beta), \quad (10.108)$$

where τ is the learning rate or step-size. This method should be well known to readers of this book.

Instead of observing $G(\beta)$ and computing its gradient, many of the methods in Chapters 10 and 11 are based on estimating the gradient using **sampled** data. That is, instead of computing the gradient directly, observations are generated from a distribution with **expectation** equal to $\nabla_{\beta} G(\beta)$. Then stochastic approximation is applied using the sample gradient $\widehat{\nabla_{\beta} G(\beta)}$ to update β according to

$$\beta' \leftarrow \beta - \tau \widehat{\nabla_{\beta} G(\beta)} \quad (10.109)$$

This recursion is often referred to as *stochastic gradient descent (SGD)* but other definitions of SGD appear in the literature. This book will refer to recursion (10.109) as SGD. Note that when tracking iteration number (as in Appendix of Chapter 9) this expression can be written as

$$\beta^{n+1} = \beta^n - \tau_n \widehat{\nabla_{\beta} G(\beta^n)}.$$

Doing so allows specifying the explicit dependence of τ on n .

The following section illustrates this approach in a very important application.

Least squares regression

Least squares regression assumes a model for data of the form

$$y_j = f(\mathbf{x}_j; \beta) + \epsilon_j \quad (10.110)$$

for $j = 1, \dots, J$, where \mathbf{x}_j denotes the j -th value of a $(I + 1)$ -dimensional vector of features⁷⁶, β denotes a $(I + 1)$ -dimensional vector of parameters, y_j denotes the j -th observation of the dependent variable and ϵ_j is an unobserved random disturbance

⁷⁶Features are referred to as covariates or independent variables in statistical models.

with mean 0 and constant variance. The function $f(\mathbf{x}_j; \boldsymbol{\beta})$ denotes a **known** real-valued function that can be either linear or nonlinear in the parameters.

Least squares regression seeks parameter values $\boldsymbol{\beta}^*$ that minimize the *expected squared error loss function*

$$G(\boldsymbol{\beta}) := E[g(\boldsymbol{\beta})] = E[(Y - f(\mathbf{x}; \boldsymbol{\beta}))^2], \quad (10.111)$$

where the expectation is with respect to the joint distribution of column vector \mathbf{x} and scalar Y and the random variable

$$g(\boldsymbol{\beta}) := (Y - f(\mathbf{x}; \boldsymbol{\beta}))^2.$$

This representation assumes that the random vector \mathbf{x} is sampled from some distribution and Y is sampled from the conditional distribution of Y given \mathbf{x} which has mean $f(\mathbf{x}; \boldsymbol{\beta})$.

A standard approach to obtaining parameter estimates is to solve the equation

$$\nabla_{\boldsymbol{\beta}} G(\boldsymbol{\beta}) = \mathbf{0}.$$

However, doing this might be complicated because of the expectation inside the gradient. Instead, under mild regularity conditions,

$$\nabla_{\boldsymbol{\beta}} G(\boldsymbol{\beta}) = E[\nabla_{\boldsymbol{\beta}} g(\boldsymbol{\beta})] \quad (10.112)$$

so that

$$\nabla_{\boldsymbol{\beta}} G(\boldsymbol{\beta}) = E[-2(Y - f(\mathbf{x}; \boldsymbol{\beta}))\nabla_{\boldsymbol{\beta}} f(\mathbf{x}; \boldsymbol{\beta})], \quad (10.113)$$

which in the linear case, when $f(\mathbf{x}; \boldsymbol{\beta}) = \mathbf{x}^T \boldsymbol{\beta}$, simplifies to

$$E[\nabla_{\boldsymbol{\beta}} g(\boldsymbol{\beta})] = E[-2(Y - \mathbf{x}^T \boldsymbol{\beta})\mathbf{x}]. \quad (10.114)$$

Note that in this expression the expected gradient is a column vector with the number of components equal to the dimension of $\boldsymbol{\beta}$.

The reason for this formality is to set things up to apply stochastic approximation. Instead of solving $\nabla_{\boldsymbol{\beta}} G(\boldsymbol{\beta}) = \mathbf{0}$ directly to minimize the expected loss in (10.111), samples are obtained from a *distribution* with mean $E[\nabla_{\boldsymbol{\beta}} g(\boldsymbol{\beta})]$ and the Robbins-Monro recursion is applied as follows.

Let $(\mathbf{x}^1, y^1), \dots, (\mathbf{x}^N, y^N)$ denote a sequence of samples from the joint distribution of \mathbf{x} and Y ⁷⁷. Then this sequence is used to estimate the gradient according to:

$$[\nabla_{\boldsymbol{\beta}} g(\boldsymbol{\beta})]^n := -2(y^n - f(\mathbf{x}^n; \boldsymbol{\beta}))\nabla_{\boldsymbol{\beta}} f(\mathbf{x}^n; \boldsymbol{\beta}) \quad (10.115)$$

where the superscript refers to the n -th sample of the gradient. When $f(\mathbf{x}; \boldsymbol{\beta})$ is a linear function of the parameters this expression becomes

$$[\nabla_{\boldsymbol{\beta}} g(\boldsymbol{\beta})]^n = -2(y^n - (\mathbf{x}^n)^T \boldsymbol{\beta})\mathbf{x}^n. \quad (10.116)$$

⁷⁷In the context of regression, subscripts indexed by j indicate data points, whereas superscripts indexed by n indicate a sequence of iterates.

In the linear case, the Robbins-Monro recursion becomes

$$\boldsymbol{\beta}' \leftarrow \boldsymbol{\beta} + \tau_n (y^n - (\mathbf{x}^n)^\top \boldsymbol{\beta}) \mathbf{x}^n, \quad (10.117)$$

where the multiplier 2 in the gradient expression has been absorbed into the learning rate and the negative sign in the gradient leads to a plus sign before τ_n .

Figure 10.33 summarizes the approach used to apply stochastic approximation when optimizing an expected value.

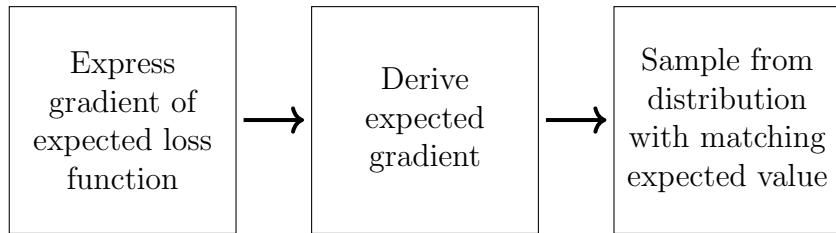


Figure 10.33: General approach for applying Robbins-Monro stochastic approximation to minimize expected loss.

A key theoretical result is that if the hypotheses of Theorem 10.3 hold for the gradient of the loss function, the sequence of iterates generated by stochastic gradient descent converges (in expectation) to a zero of the expected gradient.

Examples

Two applications of this approach follow.

Example 10.14. Using stochastic approximation to find a mean.

This example applies (10.117) to obtain an online recursion for estimating the mean μ of a random variable Y . In the additive error case, this can be viewed as a least squares regression problem in which $\boldsymbol{\beta}$ is a scalar equal to μ and \mathbf{x} is a scalar equal to 1. This means that observations are generated according to

$$y^n = \mu + \epsilon^n$$

and the Robbins-Munro recursion becomes

$$\mu^{n+1} = \mu^n + \tau_n (y^n - \mu^n). \quad (10.118)$$

This recursion is the basis for the standard online formula for the sample mean

$$\bar{y}^n$$

$$\bar{y}^n = \frac{1}{n}((n-1)\bar{y}^{n-1} + y^n). \quad (10.119)$$

To see this, set $\tau_n = 1/n$ for $n = 1, 2, \dots$, choose $\mu^1 = 0$ and apply (10.118) to obtain $\mu^2 = y^1$,

$$\mu^3 = \mu^2 + \frac{1}{2}(y^2 - \mu^2) = \frac{1}{2}(y^1 + y^2) = \bar{y}^2$$

and in general $\mu^{n+1} = \sum_{i=1}^n y^i/n = \bar{y}^n$. Then using (10.118)

$$\bar{y}^n = \mu^{n+1} = \mu^n + \frac{1}{n}(y^n - \mu^n) = \frac{n-1}{n}\mu^n - \frac{1}{n}y^n = \frac{1}{n}((n-1)\bar{y}^{n-1} + y^n),$$

which is the desired result. Observe that it requires storing only the current estimate of the mean and the number of observations. This approach has been referred to often in this chapter.

To illustrate this approach numerically, (10.118) is applied to estimate the mean of an exponential distribution. For illustrative purposes, learning rates $\tau_n = 20/(200+n)$, $n^{-0.75}$, and n^{-1} are compared. Simulations use a common random number seed and 5,000 iterates starting with $y^1 = 1$.

Figure (10.34) shows that the estimate using $\tau_n = n^{-1}$ (corresponding to the sample mean) is least variable and best estimates the true mean. Moreover its RMSE was 1.62 compared to 34.14 for the ratio learning rate and 10.10 when the learning rate was $n^{-0.75}$. Note that this conclusion appears to be valid even when observations are auto-correlated.

The following example illustrates the vector version of (10.117).

Example 10.15. Online regression

This example shows how to obtain estimates of regression parameters using (10.117). Consider a quadratic regression model of the form

$$y = \beta_0 + \beta_1 x + \beta_2 x^2 + \epsilon.$$

To write it in vector form, define

$$\boldsymbol{\beta} = \begin{bmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \end{bmatrix} \text{ and } \mathbf{x} = \begin{bmatrix} 1 \\ x \\ x^2 \end{bmatrix}$$

so that (10.117) becomes

$$\begin{bmatrix} \beta'_0 \\ \beta'_1 \\ \beta'_2 \end{bmatrix} \leftarrow \begin{bmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \end{bmatrix} + \tau_n(y^n - \boldsymbol{\beta}^\top \mathbf{x}^n) \begin{bmatrix} 1 \\ x^n \\ (x^n)^2 \end{bmatrix}. \quad (10.120)$$

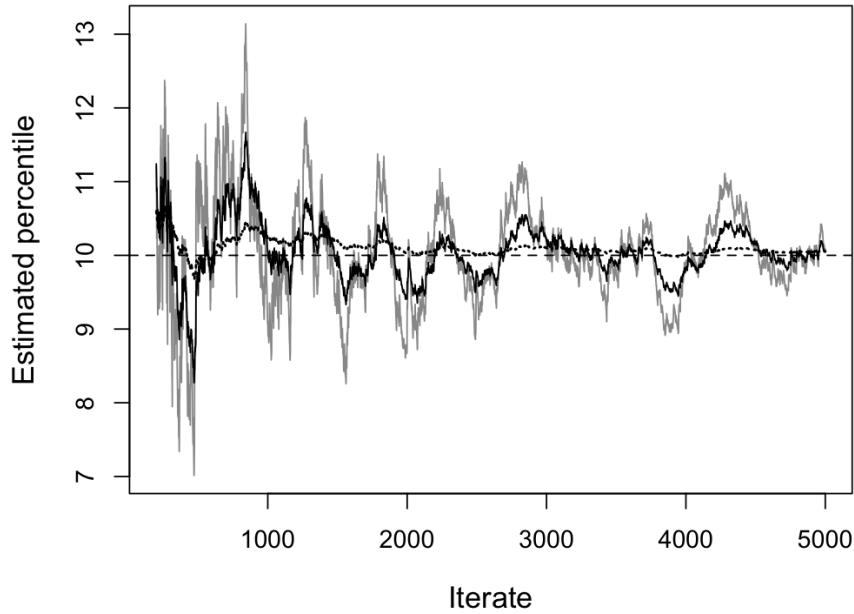


Figure 10.34: Results from a single replicate showing stochastic approximation estimates using (10.118) based on observations generated by an exponential distribution with mean 10. The gray line corresponds to $\tau_n = 20/(200 + n)$, the solid black line to $\tau_n = n^{-0.75}$ and the dashed black line to $\tau_n = n^{-1}$.

Observe that each component of the vector \mathbf{x}^n is multiplied by the same scalar quantity $\tau_n(y^n - \boldsymbol{\beta}^\top \mathbf{x}^n)$. Note also that is common practice to scale \mathbf{x}^n (and sometimes y^n) to have mean zero and standard deviation one. This can be done by keeping online estimates of the mean and standard deviations of \mathbf{x}^n and y^n .

As a numerical example suppose

$$y = -2 + 4x + 2x^2 + \epsilon$$

with ϵ sampled from a uniform distribution on $[-2, 2]$. Applying (10.120) with $\tau_n = 10/(80+n)$, each component of $\boldsymbol{\beta}$ initialized to zero and x randomly sampled from a normal distribution with mean 0 and standard deviation 1, the estimates from a typical replicate appear in Figure 10.35. They show convergence to true values. Note that when x was more variable, the estimates often diverged.

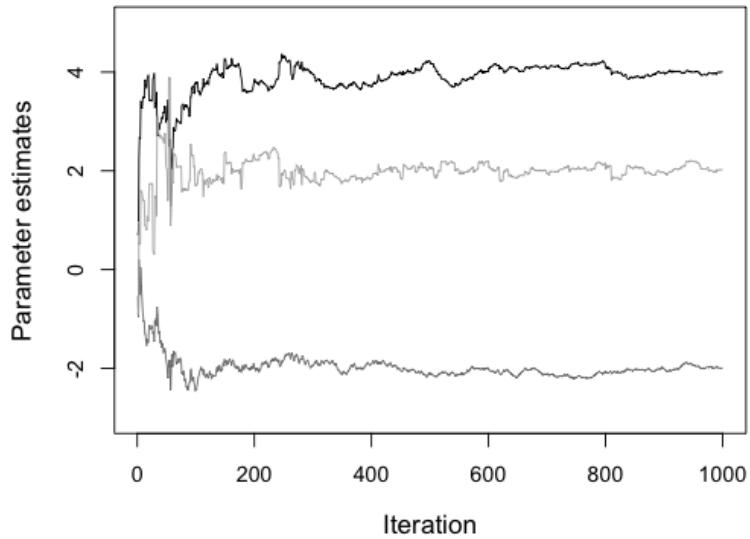


Figure 10.35: Parameter estimates from a single replicate in Example 10.15. The dark grey line corresponds to $\beta_0 = -2$, the black line to $\beta_1 = 4$ and the light grey line to $\beta_2 = 2$.

10.11.4 Offline TD(γ): Technical details

This technical section derives a representation for TD(γ) that is fundamental to the use of eligibility traces. The motivation is that value function estimation may be enhanced by using a weighted average of rollouts of different lengths.

The following lemma is fundamental to deriving the forward looking expression (10.34). Its proof is straightforward and involves interchanging the order of summation.

Lemma 10.1. Let $0 \leq \gamma \leq 1$ and r^k and v^k for $k = 0, 1, \dots$ denote scalars for which the summations in the following expressions are convergent. Then

$$(1 - \gamma) \sum_{m=1}^{\infty} \gamma^{m-1} \left(\sum_{k=0}^{m-1} r^k + v^m \right) = v^0 + \sum_{m=0}^{\infty} \gamma^m (r^m + v^{m+1} - v^m). \quad (10.121)$$

Proof. Reverse the order of summation in the first term on the left-hand side of (10.121) to obtain

$$(1 - \gamma) \sum_{m=1}^{\infty} \left(\gamma^{m-1} \sum_{k=0}^{m-1} r^k \right) = (1 - \gamma) \sum_{m=0}^{\infty} r^m \sum_{k=m}^{\infty} \gamma^k = \sum_{m=0}^{\infty} \gamma^m r^m,$$

where the last equality follows by using the standard formula for the sum of a geometric series.

Next consider the second term. Simple algebra shows

$$\begin{aligned}(1 - \gamma) \sum_{m=1}^{\infty} \gamma^{m-1} v^m &= \sum_{m=1}^{\infty} (\gamma^{m-1} - \gamma^m) v^m = v^1 + \sum_{m=1}^{\infty} \gamma^m (v^{m+1} - v^m) \\ &= v^0 + \sum_{m=0}^{\infty} \gamma^m (v^{m+1} - v^m),\end{aligned}$$

where the last equality comes from adding and subtracting v^0 . Combining these expressions and rearranging terms yields (10.121). \square

To apply this result to expression (10.33) expand it as

$$v(s^n) = (1 - \gamma) E^{d^\infty} \left[\sum_{m=1}^{\infty} \gamma^{m-1} \left(\sum_{k=0}^{m-1} r(X_{n+k}, Y_{n+k}, X_{n+k+1}) + v(X_{n+m}) \right) \middle| X_n = s^n \right],$$

set

$$r^k = E^{d^\infty} [r(X_{n+k}, Y_{n+k}, X_{n+k+1}) \mid X_n = s^n]$$

and

$$v^m = E^{d^\infty} [v(X_{n+m}) \mid X_n = s^n].$$

Applying the above lemma shows that:

$$v(s^n) = v(s^n) + E^{d^\infty} \left[\sum_{m=0}^{\infty} \gamma^m (r(X_{n+m}, Y_{n+m}, X_{n+m+1}) \right. \quad (10.122)$$

$$\left. + v(X_{n+m+1}) - v(X_{n+m}) \right) \mid X_n = s^n \Big]. \quad (10.123)$$

Offline TD(γ) applies stochastic approximation to (10.122) as follows. Given an observed trajectory $(s^n, a^n, s^{n+1}, \dots)$, stochastic approximation results in the recursion

$$\begin{aligned}v(s^n) &\leftarrow v(s^n) + \tau \sum_{m=0}^{\infty} \gamma^m (r(s^{n+m}, a^{n+m}, s^{n+m+1}) + v(s^{n+m+1}) - v(s^{n+m})) \\ &= v(s^n) + \tau \sum_{m=0}^{\infty} \gamma^m \delta_{n+m}\end{aligned}$$

where

$$\delta_k := r(s^k, a^k, s^{k+1}) + v(s^{k+1}) - v(s^k).$$

Then changing variables for the summation index yields the closed form representation for an offline implementation:

$$v(s^n) \leftarrow v(s^n) + \tau \sum_{m=n}^{\infty} \gamma^{m-n} \delta_m \quad (10.124)$$

in which τ denotes a learning rate.

This representation provides the basis for the theoretical analysis of TD(γ) in Bertsekas and Tsitsiklis [1996], generalization to function approximation and provides motivation for an online implementation. To make the argument above rigorous in an episodic model requires that the decision rule be transient⁷⁸, that is, it generates a process that terminates in a finite time with probability one.

A similar representation is also valid in discounted models with discount factor λ in which case (10.124) becomes

$$v(s^n) \leftarrow v(s^n) + \tau \sum_{m=n}^{\infty} (\lambda\gamma)^{m-n} \delta_m. \quad (10.125)$$

Derivation of this expression is left as an exercise.

⁷⁸Or proper in a stochastic shortest path problem.

Bibliographic Remarks

This chapter aimed to provide a user-friendly and practical introduction to simulation-based methods applicable to both model-free and model-based environments, with a focus on tabular models. It is not intended to be comprehensive or fully up to date, as this area of research and application is rapidly advancing.

Gosavi [2015] provides an accessible and informative overview of simulation methods for Markov decision processes, well-suited to readers of this book. His insights and comments significantly improved both the results and exposition in this chapter, particularly regarding the use of importance ratios, stochastic trace correction (STC), log-based learning rates, and choices for ϵ_n in ϵ -greedy exploration. In addition, our discussions with him regarding Relative Q-learning in average reward models enabled us to solve that version of the queuing control problem.

Noteworthy is the book Sutton and Barto [2018], which presents a reinforcement learning perspective on this material in a clear and engaging style. It served as an important source for this chapter and the next. For more rigorous and in-depth approaches grounded in Markov decision process and control theory, see Bertsekas and Tsitsiklis [1996], Bertsekas [2012], and Meyn [2022], which were our primary references for the underlying theory. In addition, there are a wealth of teaching notes and lectures available online to support further learning.

The computational results in this chapter highlight the importance of distinguishing among data generation approaches. Xiao et al. [2022] addresses this issue from a complexity-theoretic perspective, which motivates the discussion in Section 10.1.1.

Temporal-difference (TD) learning, which originated in the seminal work of Sutton (see references in Sutton and Barto [2018]), is based on applying stochastic approximation to policy evaluation, which is referred to as *prediction* in that context. Our discussion of learning rates draws on Powell [2007] and other sources.

The analysis of $\text{TD}(\gamma)$ follows Sutton and Barto [2018], Tsitsiklis and Roy [1997], and especially Bertsekas and Tsitsiklis [1996], pp. 195–197. $\text{TD}(\gamma)$ originated in foundational papers by Sutton cited in these works.

Solving Markov decision processes in a model-free online environment traces back to Watkins' Ph.D. thesis and the seminal paper Watkins and Dayan [1992] on Q-learning. That work includes a convergence proof for Q-learning under a discounted model when values are stored in a lookup table and all state-action pairs are sampled infinitely often. Singh et al. [2000] provides an in-depth analysis of single-step SARSA.

Average-reward temporal-difference learning and Q-learning are based on Tsitsiklis and Roy [1999], Mahadevan [1996], and pp. 548–551 in Bertsekas [2012]. Our treatment of relative value iteration follows Abounadi et al. [2001].

We believe the material on simulated policy iteration is original. It was partially inspired by Buşoniu et al. [2010], which describes a more general approach that incorporates function approximation. To our knowledge, the introduction of bounds for hybrid policy iteration is new.

The description of stochastic approximation in the appendix draws on the seminal paper by Robbins and Monro [1951] and the follow-up by Blum [1954]. More modern treatments appear in Bertsekas [2012], Powell [2007], and Meyn [2022]. The example using stochastic approximation to estimate percentiles is due to Ribes et al. [2019]. Sugiyama [1994] provides an excellent overview of stochastic approximation with several detailed examples. That work also drew our attention to Bellman’s prescient remark quoted at the start of Section 10.11.2.

The next chapter extends these methods to models with function approximation and contains many additional references.

Exercises

1. The exercise compares data generation mechanisms in Section 10.1.1 in the context of the queuing service rate control model analyzed in Section 10.9.
 - (a) Compare and contrast trajectory-based sampling and state-based sampling by developing code snippets for evaluating a fixed randomized stationary policy.
 - (b) Compare and contrast all four sampling methods by developing code snippets for finding an optimal policy using Q-learning.
2. Repeat the Q-learning analysis of the newsvendor model for other prices, demand distributions, learning rates and choices of η_n .
3. Extend Algorithm 10.2 to the every-visit variant and apply it to evaluate the policy in the Gridworld model in Section 10.3.1. Compare its computational effort to starting-state and first-visit Monte Carlo.
4. Evaluate the policy in Example 10.1 using Monte Carlo methods and TD(0). Note the challenges encountered in obtaining accurate estimates.
5. Show that the operator defined by the right hand side of (10.17) is a contraction mapping and that the recursion converges to the same fixed point for every choice of τ .
6. Extend the study of Q-learning and SARSA in Gridworld (Example 10.8) by considering other exploration methods, varying the number of episodes in each replicate and model parameters, especially p .
7. Repeat the analysis in the example in Section 10.7.2 in which the index of the learning rate and exploration method is the number of iterations as opposed to the count of state-action pair or state, visits respectively. How do these changes impact results?
8. This example illustrates the difference between SARSA and Q-learning. Consider the snippet of a deterministic Markov decision process depicted in Figure 10.36. In state s there are two actions, a and b and in state u there are two actions \bar{a}

and \bar{b} . Choosing action a in state s yields a reward of 1 and a transition to state u . All other rewards are immaterial.

Suppose that $q(s, a) = 0$, $q(u, \bar{a}) = 5$, $q(u, \bar{b}) = 3$ and exploration resulted in choosing action \bar{b} in state u . Show that the Q-learning and SARSA updates of $q(s, a)$ differ.

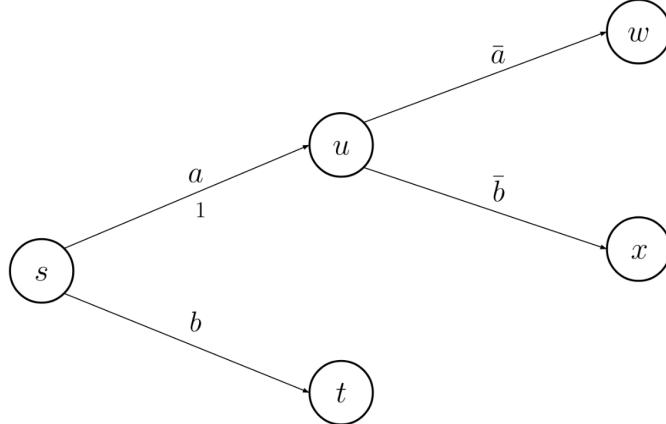


Figure 10.36: Graphical representation of model in Exercise 8

9. Consider the deterministic version of Gridworld as in Example 10.7 except assume that the robot can move “up”, “down”, “right” and “left”. Apply Q-learning and SARSA to find good policies using softmax and ϵ -greedy exploration. Compare these methods on the basis of the mean and standard deviation of the total reward per replicate.
10. Develop simulation-based policy iteration algorithms for an episodic model and apply them to the Gridworld model. Summarize your conclusions verbally and graphically.
11. Repeat the analysis in Example 10.8 with a small fraction of episodes starting in cell 4 in which the state-action frequencies were inaccurately estimated when beginning all episodes in cell 13. How do your results differ from those in the example? Also explore the use of softmax exploration instead of ϵ -greedy exploration.
12. **Finite Horizon Q-learning:** Develop a Q-learning algorithm for a **finite horizon** model. Apply it to solve the revenue management problem analyzed in Section 4.3.2.
13. **Cliff walking**⁷⁹. In this problem on a 4×12 grid, graphically represented in Figure 10.37, a robot must traverse the grid by moving from the origin “O” in

⁷⁹This colorful example was proposed by Sutton and Barto [2018]. It has a similar structure to our Gridworld example of a coffee delivering robot.

cell (1,1) in the lower left hand corner of the grid to the destination “D” in cell (1,12) in the lower right hand corner of the grid without falling off the cliff in cells (1,2) to (1,11). The robot incurs a cost of 1 unit for each step it takes and a penalty of 100 if it falls off the cliff. If it falls off the cliff it returns to the origin and starts again.

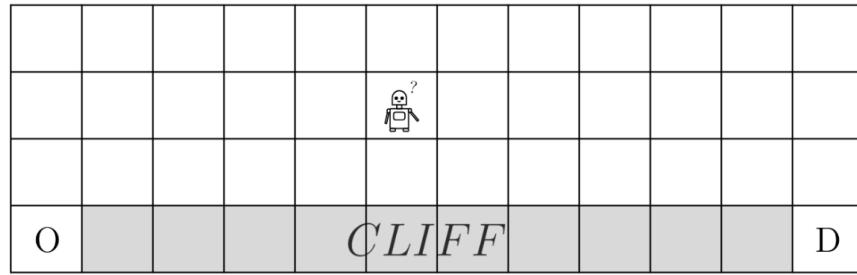


Figure 10.37: The cliff walking problem

The robot moves deterministically and can choose to move “up”, “down”, “left” or “right” in every cell. If it tries to move in a direction which is not possible, it incurs a cost of one unit and remains in that cell.

The goal is to learn an optimal path through this grid. Assume the robot does not know the layout and must learn it by ϵ -greedy exploration.

- (a) Formulate this problem as an episodic model in which an episode starts with the robot at the origin and concludes when it reaches its destination.
 - (b) Find (and state) an optimal policy using Q-learning and SARSA with a learning rate $\tau = 0.1$, ϵ -greedy exploration parameter $\epsilon = 0.1$ and a discount rate of 1.
 - (c) Accumulate the total reward per episode for Q-learning and SARSA for the first 500 episodes of each. Compare them graphically.
 - (d) Investigate the impact of τ and ϵ on algorithmic performance.
14. Show that using $TD(\gamma)$ for policy evaluation is equivalent to temporal differencing when $\gamma = 0$ and Monte Carlo estimation when $\gamma = 1$. Note in the latter case some care has to go into choosing the trace, specifying τ_n , and distinguishing the various forms of Monte Carlo estimation.
15. Analyze the model in Example 10.9 using the following two variants of hybrid policy iteration:

- (a) Estimate $r(s, a, j)$ and $p(j|s, a)$ assuming a model-free environment and then use them to estimate $q^{d^\infty}(s, a)$ as in the example.
- (b) Estimate $\sum_{j \in S} p(j|s, a)(r(s, a, j) + \lambda \hat{v}(j))$ using Monte Carlo simulation after estimating $\hat{v}(\cdot)$.

Apply the bounds in (10.89) and stopping criterion implicit in (10.90) and summarize your observations.

16. Show either theoretically or using simulation that in the queuing control model, the steady state probabilities when the service rate and arrival rates are identical (for example, each equal to 0.2) is a symmetric random walk on the state space and that the limiting probabilities are equal for each state. Find the limiting probabilities when the service rate exceeds the arrival rate for example when action $a_2 = 0.4$ or 0.6.
17. In the context of the queuing control model in Section 10.9, investigate the convergence of a variant of Algorithm 10.14 that smooths the policy value function prior to exact derivation using (10.86). Compare the effects of smoothing both the policy value function and the q -function to smoothing only the policy value function.
18. **A large queuing service rate control model.** Use Q-learning and the policy iteration algorithms of Section 10.8 to analyze the larger version of the discounted queuing service rate control model previously solved with (deterministic) policy iteration in Section 5.11.4. In it the queue capacity is 5,000, $A_s = \{0.2, 0.3, \dots, 0.7\}$, $b = 0.2$, $m(a_k) = 2k^3$, $f(s) = s^2$ and the discount rate equals 0.4. A low discount rate was chosen to obtain an interesting optimal policy.

Use Relative Q-learning to solve an average reward version of this problem instance.

19. **Inventory control with hidden Markov demand.** Consider the following inventory model. At the end of each day, the inventory manager can order j units from a supplier that arrive the next morning at a cost of c per item plus a fixed ordering cost of k per order. When the demand d on any day exceeds the number of units on hand, the excess demand is unfilled and lost. If demand is less than or equal the number of units received, the system receives a revenue r for each unit sold. Unsold inventory is carried over to the next day at a cost of h per unit.

Suppose that daily demand follows a binomial distribution with parameters D and p and moreover p takes on the values p_L and p_H , which vary from day to day according to a Markov chain with transition probability matrix

$$P = \begin{bmatrix} 0.2 & 0.8 \\ 0.6 & 0.4 \end{bmatrix},$$

where row 1 corresponds to p_L . Suppose that $c = 10, K = 5, r = 20, h = 3, p_H = 0.5, p_L = 0.2$ and $D = 20$.

- (a) Develop a simulation model of this system and use it to evaluate the policy that orders 5 units every day.
 - (b) Find a good policy for the infinite horizon discounted model with $\lambda = 0.95$.
20. Consider the model in Exercise 1 of Chapter 2. In performing the following calculations investigate the effect of learning rate sequence $\tau_n, n = 1, 2, \dots$ and sample size on the accuracy of policy evaluation. Compare results on the basis of RMSE and visual inspection of the policy value functions.
- In addition, when optimizing, investigate the impact of exploration method. Summarize results in terms of the similarity of the policies to the optimum, the true values of the policies, and appropriate RMSE measures.
- (a) Use temporal differencing, $\text{TD}(\gamma)$, and Monte Carlo with truncated discounted rewards and geometric sampling to estimate the infinite horizon discounted reward with $\lambda = 0.9$ for the following policies:
 - i. The deterministic stationary policy that uses $d(s_i) = a_{i,1}$ in state s_i for $i = 1, 2, 3$.
 - ii. The randomized stationary policy that uses action $a_{i,1}$ with probability $e^{-0.5i}$ and action $a_{i,2}$ with probability $1 - e^{-0.5i}$ in state s_i for $i = 1, 2, 3$.
 - (b) Evaluate the average reward and bias of these policies using Monte Carlo methods and temporal differencing.
 - (c) Develop and apply a $\text{TD}(\gamma)$ algorithm for an average reward model.
 - (d) Use Q-learning, SARSA, hybrid policy iteration and online policy iteration to find an optimal policy for a discounted version of the model. Consider both $\lambda = 0.9$ and $\lambda = 0.999$. Comment on the effect of λ on the execution of the algorithm.
 - (e) Use Q-learning and Relative Q-learning to find an optimal policy for the average reward version of this model.
21. Consider the admission control queuing model model of Section 3.4.2 with reward $R = 20$, holding cost $f(j) = 1.05j$, service probability $w = 0.45$, and arrival probability $b = 0.5$. Truncate the state space at $M = 100$.
- (a) Write down recursions for the state-action value function assuming:
 - i. The holding cost is incurred before an arrival or service completion.
 - ii. The holding cost is incurred after an arrival or service completion.
 - (b) Find an optimal policy for a discounted version of this model with $\lambda = 0.95$ and investigate its sensitivity to the problem data.

- (c) Use Q-learning, hybrid policy iteration and online policy iteration to find a good policy for this model using simulation.
 - (d) Investigate the impact of truncation on the optimal policy and its gain by varying the truncation level M .
 - (e) Use Q-learning and Relative Q-learning to find good policies for an average reward variant of this model.
22. Develop simulation-based policy iteration algorithms for an average reward model and use it to solve the queuing service rate control model. Compare results to those obtained using Relative Q-learning in this chapter.
23. Develop a Q-learning algorithm for a finite horizon model and apply it to the revenue management problem in Section 3.3. Note that this requires defining state-action value functions that vary with decision epoch.
Based on these observations, what can you conclude about the challenges of using trajectory-based sampling for value function estimation and optimization?
24. Use the recursion (10.103) to find a root of the function $f(x) = 1/(1+e^{-0.5x}) - 0.5$ based on observing $f(x) + \epsilon$ where ϵ is normally distributed with mean 0 and standard deviation 0.1. Investigate the variability and accuracy of estimates as you vary the learning rate τ_n and the number of observations.
25. Derive (10.125) by modifying the argument at the start of Section 10.11.4 to include a discount factor.
26. **Queuing with a budget constraint.** Formulate and analyze the queuing service rate control model in which an episode ends when a finite budget has been expended and the objective is to maximize *throughput*, that is the number of customers served. Assuming the same parameter values as in Section 10.9, investigate the impact of the budget constraint level on your results.

Chapter 11

Simulation with Function Approximation

This material will be published by Cambridge University Press as “Markov Decision Processes and Reinforcement Learning” by Martin L. Puterman and Timothy C. Y. Chan. This pre-publication version is free to view and download for personal use only. Not for re-distribution, re-sale, or use in derivative works. ©Martin L. Puterman and Timothy C. Y. Chan, 2025.

In fact, everything we know is only some kind of approximation, because we know that we do not know all the laws as yet. Therefore, things must be learned only to be unlearned again or, more likely, to be corrected¹.

Richard P. Feynman, American theoretical physicist, 1918-1988.

As Feynman so eloquently puts it, every model is an approximation that must be revised as new information becomes available. This is exactly the principle underlying this chapter that combines function approximation with simulation.

Modern applications of Markov decision processes require both function approximation (Chapter 9) and simulation² (Chapter 10). Such methods are usually referred to as *reinforcement learning* (RL), but as noted at the beginning of Part III, RL refers to any setting in which the decision maker (or agent) learns by trial and error.

This chapter motivates, describes, and applies methods for:

- value function approximation,
- state-action value function approximation, and
- policy approximation.

¹Feynman et al. [1963].

²As noted earlier the expression *simulation* refers to either computer-based simulation or real-time experimentation.

Whereas value function and state-action value function approximation generalize the methods introduced in Chapters 9 and 10, this chapter describes approaches that approximate the action-choice probability distribution of a randomized policy directly, that is, without using value functions as intermediaries.

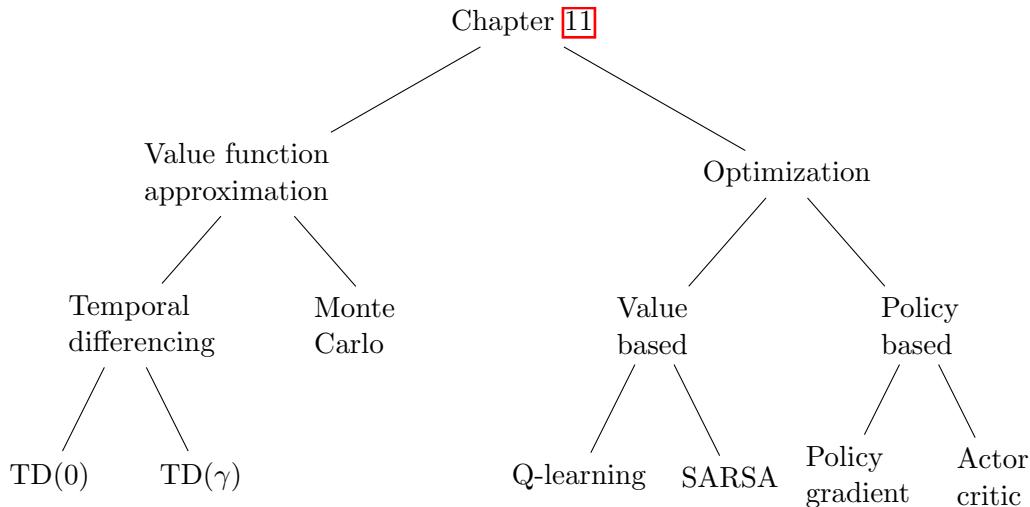


Figure 11.1: Schematic representation of methods described in Chapter 11

The methods in this chapter represent value functions, state-action value functions, and randomized policies by parametric functions of features and then estimate the parameters online or offline using simulated data. Parameters are often referred to as *weights*.

Figure 11.1 summarizes the methods described in this chapter. Note that this chapter considers only discounted and episodic models. Generalization to average reward models is left to the reader. Technical prerequisites include familiarity with gradient descent and the matrix formulation of least squares regression (see Chapter 9 Appendix).

11.1 The challenge of large models

In contrast to the tabular models analyzed in Chapter 10, models with large state (and action) spaces require methods that represent value functions and decision rules in forms suitable for computation and application. The following sections discuss methods for doing so.

11.1.1 Features

The underlying approach is to summarize the set of states or state-action pairs by lower-dimensional sets of real-valued functions defined on the set of states or state-action pairs. Such functions are referred to as *features* or *basis functions*.

Specifying features

Choosing suitable features presents challenges. In physical models, such as navigating a drone through three-dimensional space, features may represent the position, velocity and angular orientation. In discrete models, with numerical-valued states and actions such as controlling a multi-dimensional queuing system, features may be (scaled) powers and products of powers of the number of entities in each queue.

In Gridworld models, specifying features may be more challenging. For example, in a robot guidance problem on an $M \times N$ grid, a possible choice of features is the row index, the column index and some higher order and cross-product terms of these quantities. Another possibility is to aggregate cells by choosing features to be indicator functions of overlapping or disjoint subsets of cells.

Note that:

Specifying features to be indicator functions of individual states or state-action pairs is equivalent to using a *tabular model*.

Consequently, the results and methods in Chapter 10 are special cases of those in this chapter.

Combining features

Once features are specified, the issue of how to combine them to obtain estimates of quantities of interest arises. In general, value functions, state-action value functions and action-choice probabilities can be approximated by:

1. linear functions of features
2. nonlinear functions of features
3. neural networks³

Regarding notation, vectors of features evaluated at s or (s, a) are denoted by $\mathbf{b}(s)$ or $\mathbf{b}(s, a)$, respectively. Components of these vectors are written as $b_i(s)$ or $b_{i,j}(s, a)$. In both cases, vectors of weights are denoted by $\boldsymbol{\beta}$, with components β_i and $\beta_{i,j}$, respectively.

11.1.2 Policy value function approximation

This section briefly reviews value function approximation⁴ for a fixed policy, which was introduced in Chapter 9. For ease of exposition and to encompass many practical

³Of course, neural networks are nonlinear functions but they are distinguished here because they have a vast and specialized array of methods for parameter estimation.

⁴The problem of estimating a value function for a specified policy is referred to as *prediction* in the computer science literature.

applications, assume a linear value function approximation of the form

$$v(s; \beta_0, \dots, \beta_I) := \beta_0 b_0(s) + \beta_1 b_1(s) + \dots + \beta_I b_I(s), \quad (11.1)$$

where β_0, \dots, β_I denote weights (parameters) and $b_0(s), \dots, b_I(s)$ denote the value of the features evaluated at $s \in S$. Note that indexing begins at 0 to conform with the convention in linear regression models in which $b_0(s) = 1$ for all $s \in S$ so that the first term in the sum corresponds to a constant β_0 . The above approximation is linear in the weights, however, features may be nonlinear functions (for example powers) of the state.

In vector notation (which is preferable) (11.1) can be written succinctly as:

$$v(s; \boldsymbol{\beta}) = \boldsymbol{\beta}^T \mathbf{b}(s), \quad (11.2)$$

where $\mathbf{b}(s)$ denotes a (column) vector of pre-specified features evaluated at $s \in S$ and $\boldsymbol{\beta}$ denotes a weight vector.

The beauty of such a representation is that in (11.1) the coefficients can be interpreted as *marginal* rewards or costs. For example, if s denotes the number of entities in a single-server queuing system and $v(s; \boldsymbol{\beta}) = \beta_0 + \beta_1 s$ represents an approximation to the expected infinite horizon discounted cost, then β_1 represents the increase in expected discounted cost that results from adding one customer to the system.

In greater generality, $v(s; \boldsymbol{\beta})$ may be a nonlinear function such as a neural network. As a convention, estimates of $\boldsymbol{\beta}$ will be denoted by $\hat{\boldsymbol{\beta}}$ and estimates of value functions by either $\hat{v}(s)$ or $v(s; \hat{\boldsymbol{\beta}})$.

11.1.3 State-action value function approximation

Approximations of state-action value functions require features expressed in terms of states and actions. Determination of suitable functional forms for the components of the vector $\mathbf{b}(s, a)$ can present some challenges.

One can set

$$b_{i,j}(s, a) = f_i(s)h_j(a), \quad (11.3)$$

where $f_i(s)$ for $i = 1, \dots, I$ is a function defined on states and $h_j(a)$ for $j = 1, \dots, J$ is a function defined on actions. In the linear case, the approximation can be represented by

$$q(s, a; \boldsymbol{\beta}) = \sum_{i=1}^I \sum_{j=1}^J \beta_{i,j} b_{i,j}(s, a) = \sum_{i=1}^I \sum_{j=1}^J \beta_{i,j} f_i(s)h_j(a), \quad (11.4)$$

where $\boldsymbol{\beta}$ is the “long” vector $(\beta_{1,1}, \dots, \beta_{I,J})$. In greater generality, $q(s, a; \boldsymbol{\beta})$ may be a pre-specified nonlinear function of features.

When A_s contains a small number of elements and does not vary with the state, such as in the queuing service rate control model, one may choose $h_j(a)$ to be the indicator function of the action a_j , that is

$$h_j(a) = I_{\{a_j\}}(a) = \begin{cases} 1 & a = a_j \\ 0 & a \neq a_j. \end{cases} \quad (11.5)$$

In this case, $J = |A_s|$. Consequently,

$$b_{i,j}(s, a) = \begin{cases} f_i(s) & a = a_j \\ 0 & a \neq a_j \end{cases} \quad (11.6)$$

for $a \in A_s$ and $s \in S$. This is equivalent to representing $q(s, a)$ as a different linear combination of the functions $f_i(s)$ for each $a \in A_s$.

To make this concrete, consider a queuing control model (such as in Section 10.9) with service rates a_1 and a_2 , and suppose $f_0(s) = 1^5$, $f_1(s) = s$ and $h_j(a) = I_{\{a_j\}}(a)$ as in (11.5). Then using the product-form representation in (11.4), the approximation of $q(s, a)$ can be written as

$$\begin{aligned} q(s, a; \boldsymbol{\beta}) &= \beta_{0,1}b_0(s, a_1) + \beta_{0,2}b_0(s, a_2) + \beta_{1,1}b_1(s, a_1) + \beta_{1,2}b_1(s, a_2) \\ &= \beta_{0,1}f_0(s)h_1(a) + \beta_{0,2}f_0(s)h_2(a) + \beta_{1,1}f_1(s)h_1(a) + \beta_{1,2}f_1(s)h_2(a) \\ &= \beta_{0,1}I_{\{a_1\}}(a) + \beta_{0,2}I_{\{a_2\}}(a) + \beta_{1,1}sI_{\{a_1\}}(a) + \beta_{1,2}sI_{\{a_2\}}(a). \end{aligned}$$

This is equivalent to using the following, possibly different, linear functions for each action:

$$\begin{aligned} q(s, a_1; \boldsymbol{\beta}) &= \beta_{0,1} + \beta_{1,1}s \\ q(s, a_2; \boldsymbol{\beta}) &= \beta_{0,2} + \beta_{1,2}s. \end{aligned}$$

Hence, this representation corresponds to approximating $q(s, a)$ by lines with different slopes and intercepts for each action.

Vector representation

It is convenient to write (11.4) in vector form as

$$q(s, a; \boldsymbol{\beta}) = \boldsymbol{\beta}^\top \mathbf{b}(s, a), \quad (11.7)$$

⁵Recall in Chapter 9 that the index started at 0 in the linear case so as to include an explicit constant term in the function approximation. This example and several others in this chapter adopt this convention.

where β is a column vector of weights of length IJ and $\mathbf{b}(s, a)$ is a column vector of features evaluated at (s, a) of length IJ . Of course, the weights and features must be ordered consistently. For example, if $|A_s| = J$ for each s , then β can be written as

$$\beta = \begin{bmatrix} \beta_{a_1} \\ \vdots \\ \beta_{a_J} \end{bmatrix}, \quad (11.8)$$

where the column vector β_{a_j} contains the coefficients corresponding to action a_j for each state s . Following this approach, the approximation in the above queuing example can be represented using

$$\beta = \begin{bmatrix} \beta_{0,1} \\ \beta_{1,1} \\ \beta_{0,2} \\ \beta_{1,2} \end{bmatrix} = \begin{bmatrix} \beta_{a_1} \\ \beta_{a_2} \end{bmatrix}, \quad \mathbf{b}(s, a_1) = \begin{bmatrix} 1 \\ s \\ 0 \\ 0 \end{bmatrix} \quad \text{and} \quad \mathbf{b}(s, a_2) = \begin{bmatrix} 0 \\ 0 \\ 1 \\ s \end{bmatrix}. \quad (11.9)$$

Another possible approximation for $q(s, a)$ is to write it as a linear combination of the product of state space features and action space features of the form $f(s)h(a)$. For example, suppose $f(s) = 1 + s$ and $g(a) = 1 + a$. Then

$$f(s)h(a) = 1 + s + a + as.$$

In this case

$$q(s, a; \beta) = \beta_{0,0} + \beta_{1,0}s + \beta_{0,1}a + \beta_{1,1}as.$$

where $\beta = (\beta_{0,0}, \beta_{1,0}, \beta_{0,1}, \beta_{1,1})$.

Without the interaction term, $\beta_{1,1}as$, greedy action choice based on this approximation to $q(s, a)$ would depend only on the value of $\beta_{0,1}$, independent of s . When it is present, it would depend on the state through $\beta_{0,1} + \beta_{1,1}s$.

When the states and actions are vectors \mathbf{s} and \mathbf{a} respectively, it may be preferable to approximate $q(\mathbf{s}, \mathbf{a}; \beta)$ by a neural network with inputs \mathbf{s} and \mathbf{a} . As pointed out above, if all basis functions are indicator functions of state-action pairs, the approximation is equivalent to the tabular representation studied in Chapter 10.

Recovering value functions

Recall that given a state-action value function, the value function of the stationary policy d^∞ can be represented by:

$$v_d(s) = \begin{cases} q(s, d(s)) & \text{if } d \text{ is deterministic} \\ \sum_{a \in A_s} w_d(a|s)q(s, a) & \text{if } d \text{ is randomized.} \end{cases}$$

Combining this observation with (11.7) results in using approximations

$$\hat{v}_d(s) = \begin{cases} \beta^\top \mathbf{b}(s, d(s)) & \text{if } d \text{ is deterministic} \\ \sum_{a \in A_s} w_d(a|s)\beta^\top \mathbf{b}(s, a) & \text{if } d \text{ is randomized.} \end{cases}$$

Decision rules

When analyzing tabular models, it was possible to state decision rules **explicitly**. Deterministic decision rules could be specified in a look-up table and randomized decision rules could be specified as a distribution over the set of actions. However in models with large state spaces this will not be possible, or even necessary.

When using function approximation, the quantity β corresponds to a specific decision rule. That is, instead of explicitly encoding the policy, one can use β to generate actions as follows.

Note that algorithms in this chapter assume that basis functions or features are pre-specified and fixed.

Algorithm 11.1. Implicit action choice

1. Specify β_0 .
2. Specify a state $s \in S$.
3. Compute $q(s, a; \beta_0) \leftarrow \beta_0^\top \mathbf{b}(s, a)$ for all $a \in A_s$.
4. (a) **Deterministic action selection:** Set $a_s \in \arg \max_{a \in A_s} q(s, a; \beta_0)$.
 (b) **Randomized action selection** Sample a_s using ϵ -greedy or softmax sampling based on $q(s, a; \beta_0)$.
5. Return a_s .

To think of this another way, instead of carrying around a look-up table, our agent will store a (much lighter) vector β_0 . When in state s , the agent can then apply Algorithm 11.1 to choose an action a_s to apply. In applications, one would prefer a greedy action; randomized action selection is appropriate for exploration or using randomized policies generated by policy approximation methods.

From this perspective, it would be difficult to determine which β corresponds to a specific policy. In the special case when the model is known and rewards are independent of the subsequent state, choosing $\beta_0 = \mathbf{0}$ would result in setting $q(s, a; \beta_0) = r(s, a)$ so that greedy action selection would generate a myopic policy.

11.1.4 Policy approximation

A third approach focuses on parameterizing the policy directly. It represents $w(a|s) = P(Y_n = a|X_n = s)$ as a parametric function of features defined in terms of states and actions. Of course the functional form must ensure that $w(a|s) \geq 0$ for all $a \in A_s$ and $\sum_{a \in A_s} w(a|s) = 1$. Features are represented by the vector $\mathbf{b}(s, a)$ and weights

by the vector β . A convenient representation, referred to as a softmax⁶ or logistic transformation, is given by

$$w(a|s; \beta) = \frac{e^{\beta^\top b(s,a)}}{\sum_{a' \in A_s} e^{\beta^\top b(s,a')}}. \quad (11.10)$$

An alternative to using (11.10) is to represent $w(a|s; \beta)$ by a *neural network* where β is a vector of weights.

11.2 Policy value function approximation

This section focuses on simulation methods for approximating a policy value function. It describes Monte Carlo and temporal differencing methods.

11.2.1 Monte Carlo policy value function approximation

The idea is quite simple. Simulate (or observe) the process under a fixed policy for a set of starting states, store the observed values and then estimate the policy value function using least squares. For ease of exposition, this section only provides a starting-state version of the algorithm; the first-visit variant is a simple modification while the every-visit version takes some care to describe.

The following discussion takes the perspective that policy value functions are to be approximated without direct analyst input — such is the case when they are intermediaries in a method to find optimal policies. Consequently, the set of starting states and features are not selected to align with any particular policy.

Monte Carlo estimation of policy value function approximations in an episodic model

Recall that Δ denotes the set of stopped states and $g(s)$ denotes the value on termination in state s . The algorithm assumes a randomized stationary policy and pre-specified features.

Algorithm 11.2. Starting-state Monte Carlo policy value function approximation for an episodic model

1. Initialize:

- (a) Specify $d \in D^{\text{MR}}$ and the number of episodes K .

⁶Previously, the softmax function was used for exploration. Here it is used to represent a policy directly. Note the constant η previously used to parameterize the function is accounted for by the scale of β .

- (b) Specify a subset of starting states $\bar{S} \subseteq S \setminus \Delta$.
 - (c) For all $s \in \bar{S}$, create an empty list $\text{VALUES}(s)$.
2. **Generate values:** For all $\bar{s} \in \bar{S}$:
- (a) $k \leftarrow 1$.
 - (b) While $k \leq K$:
 - i. $s \leftarrow \bar{s}$ and $v \leftarrow 0$.
 - ii. **Generate episode:** While $s \notin \Delta$:
 - A. Sample a from $w_d(\cdot|s)$.
 - B. Simulate (s', r) or sample s' from $p(\cdot|s, a)$ and set $r \leftarrow r(s, a, s')$.
 - C. Update the value:

$$v \leftarrow r + v.$$
 - D. $s \leftarrow s'$.
 - iii. **Terminate episode** ($s \in \Delta$):
 - A. Append $v + g(s)$ to $\text{VALUES}(\bar{s})$.
 - B. $k \leftarrow k + 1$.
 - 3. **Estimate parameters:** Use data $\{(s, v) | s \in \bar{S}, v \in \text{VALUES}(s)\}$ to obtain estimates of $\hat{\beta} = (\hat{\beta}_0, \dots, \hat{\beta}_I)$ using least squares or weighted least squares.
 - 4. **Terminate:** Return $\hat{\beta}$.

Some comments follow:

1. Instead of specifying a set of states at which to evaluate $v(s)$, states can be chosen randomly. Specifying states judiciously may result in more accurate parameter estimates.
2. The above algorithm generates $|\bar{S}|K$ data points of the form $(s, v(s))$. When $v(s)$ is linear in the weights or parameters, least squares estimates are available in closed form. When a nonlinear approximation is used, iterative estimation methods are required. Since these methods will be coded, both approaches can easily be included in optimization algorithms.
3. The above algorithm records values for the starting state only. As noted in Chapter 10 it is easy and more efficient to modify the algorithm to store estimates for all visited states.
4. When the variability of the Monte Carlo estimates depends on the state, weighted least squares provides a more appropriate approach for parameter estimation.

5. The key difference between this algorithm and Algorithm 10.1 (starting-state Monte Carlo without value function approximation) is the addition of the concluding step, which estimates the parameters $\hat{\beta}$ from pairs of states-value function estimates. Note also that estimation requires values associated with multiple states (those in \bar{S}), whereas Algorithm 10.1 provides an estimate of the value for a single state.
6. Note that an alternative is to take the mean of all values for each s , and then regress on the $(s, \text{mean}(\text{VALUES}(s)))$ pairs, where there is now a single value for each s . However, this approach would lose information on the variability of different v estimates for the same s .

An example

The following model of optimal stopping in a random walk illustrates the application of the above algorithm.

Example 11.1. Monte Carlo policy value function estimation for a random walk with stopping.

Consider the random walk model described in Example 6.19 but with $S = \{1, \dots, 500\}$, continuation cost $c = 8$, stopping reward $g(s) = 0.3s + 0.7s^2$ and $p = 0.51$, where p denotes the probability of a transition from s to $s + 1$ except in state 500 where it denotes the probability of remaining in state 500.

The decision maker trades off the cost of continuing versus the benefit of reaching the high reward states. There are two regions where the decision maker might consider stopping: in very low states when the cost of reaching the high reward states might exceed the benefit of waiting, or in very high states.

This example investigates the use of a linear function of polynomials of the form

$$v(s; \beta_0, \beta_1, \dots, \beta_I) = \beta_0 + \beta_1 s + \beta_2 s^2 + \dots + \beta_I s^I$$

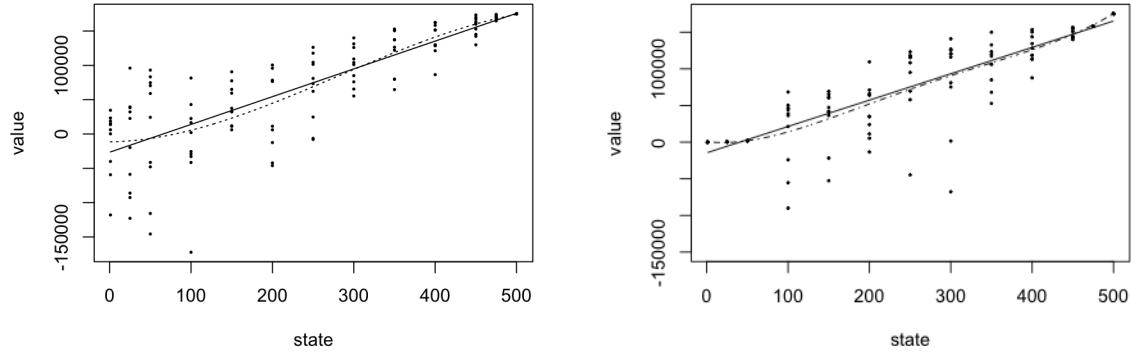
for small values of I and considers three policies:

π_1 : Stop in all states, that is $\Delta = S$.

π_2 : Stop only when in state 500, that is $\Delta = \{500\}$.

π_3 : Stop in states $\{1, \dots, 50\}$ and $\{475, \dots, 500\}$, that is $\Delta = \{1, \dots, 50\} \cup \{475, \dots, 500\}$.

Under π_1 , no approximation is necessary and $v^{\pi_1}(s) = g(s)$. To apply Algorithm 11.2 to π_2 and π_3 , choose $\bar{S} = \{1, 25, 50, 100, 150, \dots, 400, 450, 475, 500\}$ and $K = 10$ replications for each $s \in \bar{S}$.



(a) Linear (solid line) and cubic (dashed line) value function approximations for policy π_2 . (b) Linear (solid line) and fourth-order (dashed line) value function approximations for policy π_3 .

Figure 11.2: Monte Carlo value function estimation for two policies in Example 11.1.

Figure 11.2a) shows linear ($I = 1$, solid line) and cubic ($I = 3$, dashed line) approximations to the Monte Carlo values for policy π_2 . The linear approximation was

$$v^{\pi_2}(s; \hat{\beta}_0, \hat{\beta}_1) = -26,549.1 + 404.7s.$$

and the cubic approximation was

$$v^{\pi_2}(s; \hat{\beta}_0, \hat{\beta}_1, \hat{\beta}_2, \hat{\beta}_3) = -11,459.94 + 17.90s + 1.72s^2 - 0.0021s^3.$$

The cubic fit was slightly more accurate with a smaller standard error. The fourth order fit was almost identical to that of the cubic model.

Figure 11.2b) shows linear and fourth-order approximations for policy π_3 . Note that a fourth-order polynomial was necessary to represent well the values in the stopped region. The estimated approximations were:

$$v^{\pi_3}(s; \hat{\beta}_0, \hat{\beta}_1) = -14594.7 + 359.6s$$

and

$$v^{\pi_3}(s; \hat{\beta}_0, \hat{\beta}_1, \dots, \hat{\beta}_4) = 647.3 - 123.43s + 3.41s^2 - 0.009s^3 + 0.000009s^4.$$

It is left as an exercise to investigate other choices for \bar{S} and parameter values.

^aThe example chooses extra values close to the endpoints to achieve greater accuracy. Note also that replications in the stopping regions were unnecessary since $v(s) = g(s)$ therein.

Discounted models using truncation and Monte Carlo estimation

The approach to approximating the policy value function for a discounted model is similar to that for an episodic model, however, it is necessary to either truncate trajectories after a fixed number of iterations or use geometric stopping times. The truncation version accumulates the discounted reward for some pre-specified number, M , of decision epochs. The geometric stopping time version is identical to that for an episodic model in which at each decision epoch the system can enter an absorbing state with probability $1 - \lambda$. A formal statement of the truncation version of the algorithm is provided here; the geometric stopping time version is left to the reader. The algorithm assumes a randomized stationary policy.

Algorithm 11.3. Starting-state Monte Carlo policy value function approximation for a discounted model with truncation

1. **Initialize:**
 - (a) Specify $d \in D^{\text{MR}}$.
 - (b) Specify the number of replicates K and the truncation level M .
 - (c) Specify a subset of starting states $\bar{S} \subseteq S$.
 - (d) For all $s \in \bar{S}$, create an empty list $\text{VALUES}(s)$.
2. **Generate values:** For all $\bar{s} \in \bar{S}$:
 - (a) $k \leftarrow 1$.
 - (b) While $k \leq K$:
 - i. $s \leftarrow \bar{s}$, $v \leftarrow 0$ and $m \leftarrow 1$.
 - ii. **Generate replicate:** While $m \leq M$:
 - A. Sample a from $w_d(\cdot|s)$.
 - B. Simulate (s', r) or sample s' from $p(\cdot|s, a)$ and set $r \leftarrow r(s, a, s')$.
 - C. $v \leftarrow \lambda^{m-1}r + v$.
 - D. $s \leftarrow s'$.
 - E. $m \leftarrow m + 1$.
 - iii. Append v to $\text{VALUES}(\bar{s})$.
 - iv. $k \leftarrow k + 1$.
 3. **Estimate parameters:** Use data $\{(s, v) | s \in \bar{S}, v \in \text{VALUES}(s)\}$ to estimate parameters $\hat{\beta} = (\hat{\beta}_0, \dots, \hat{\beta}_I)$ using least squares or weighted least squares.
 4. **Terminate:** Return $\hat{\beta}$.

As previously discussed, a first-visit and an every-state version of this algorithm would not be applicable because truncation lengths would vary from state to state.

Example 11.2. Monte Carlo policy value function estimation for the queuing service rate control model.

This example applies Algorithm 11.3 to the queuing control model with $S = \{0, \dots, 50\}$. It explores the quality of cubic polynomial approximations to the policy value function for the deterministic stationary policies derived from decision rules

$$d_1(s) = \begin{cases} a_1 & \text{for } s \leq 25 \\ a_3 & \text{for } s > 25 \end{cases}$$

and

$$d_2(s) = a_2 \quad \text{for } 0 \leq s \leq 50,$$

using 40 replicates of the algorithm for $\lambda = 0.9, 0.95, 0.98$ and common random numbers in each replicate. Each replicate consists of $K = 10$ episodes for each $s \in \bar{S} = \{0, 5, 10, \dots, 45, 50\}$. Cumulative discounted rewards are truncated at $M = 600$.

Outcomes were compared on the basis of the RMSE of the fit:

$$\text{RMSE} = \left(\frac{1}{|S|} \sum_{s \in S} (v(s, \hat{\beta}) - v_\lambda^{d^\infty}(s))^2 \right)^{\frac{1}{2}}. \quad (11.11)$$

Note that $v_\lambda^{d^\infty}(s)$ can easily be determined exactly using the policy evaluation methods in Chapter 5.

Table 11.1 summarizes results. The column “True” establishes a baseline by providing the RMSE of a cubic polynomial fit to the exact policy value function. Observe that:

1. The accuracy of a cubic fit to the true policy value function decreases with λ as does the accuracy of each of the Monte Carlo estimates.
2. The cubic model fits the policy value function of $(d_2)^\infty$ better than that of $(d_1)^\infty$. This is expected because of the step change in d_1 at $s = 26$ (see Figure 11.3a).
3. Least squares estimates of policy value functions are more accurate than weighted least squares estimates based on the estimated variance of values at each $s \in \bar{S}$.
4. Results (not shown) indicated that the accuracy of the estimates varied considerably with the truncation level N . When $N = 300$ the RMSE was more than three times greater than that shown when $\lambda = 0.98$.

Figure 11.3 compares, for a single replicate, the true policy value function, its cubic polynomial fit and a cubic polynomial approximation based on Monte Carlo estimates. Figure 11.3a shows that for decision rule d_1 and $\lambda = 0.95$, the cubic polynomial fit deviates from the true policy value function, and the cubic approximation based on Monte Carlo estimation lies below the true policy value function. Figure 11.3b shows that for decision rule d_2 and $\lambda = 0.98$, the cubic polynomial fit accurately approximates the true policy value function, but the cubic approximation based on Monte Carlo estimation deviates from the true policy value function at high occupancy states.

The impact of these discrepancies on optimization is investigated in subsequent sections of this chapter.

λ	d_1			d_2		
	True	MC-LS	MC-WLS	True	MC-LS	MC-WLS
0.90	67.99	551.14	569.17	59.15	567.90	579.10
0.95	344.73	1,029.94	1,135.56	153.05	1,098.57	1,151.50
0.98	1,261.33	2,161.07	2,801.03	432.18	2,480.18	2,770.44

Table 11.1: Mean of RMSE over 40 replicates for least squares (MC-LS) and weighted least squares (MC-WLS) fit to Monte Carlo estimates and the RMSE for a cubic regression fit to the exact policy value (True) as a function of discount rate and decision rule.

11.2.2 Temporal differencing with policy value function approximation

As an alternative to Monte Carlo estimation in which estimates are derived after simulating (or observing) many trajectories, this section describes an online approach that generalizes TD(0) by updating parameter estimates after each state transition⁷.

Motivation

This section applies stochastic gradient descent type methods from the Appendix of Chapter 10 to obtain a recursive method for estimating a vector of weights in an approximate policy value function. As detailed below, this approach modifies gradient descent to simplify computation and improve numerical stability.

Let d denote a Markovian randomized decision rule, d^∞ the corresponding stationary policy, and s an arbitrary state in S . The objective is to find a vector of weights

⁷As noted above, when the features are indicators of states or states and actions, this is equivalent to the online methods in Chapter 10.

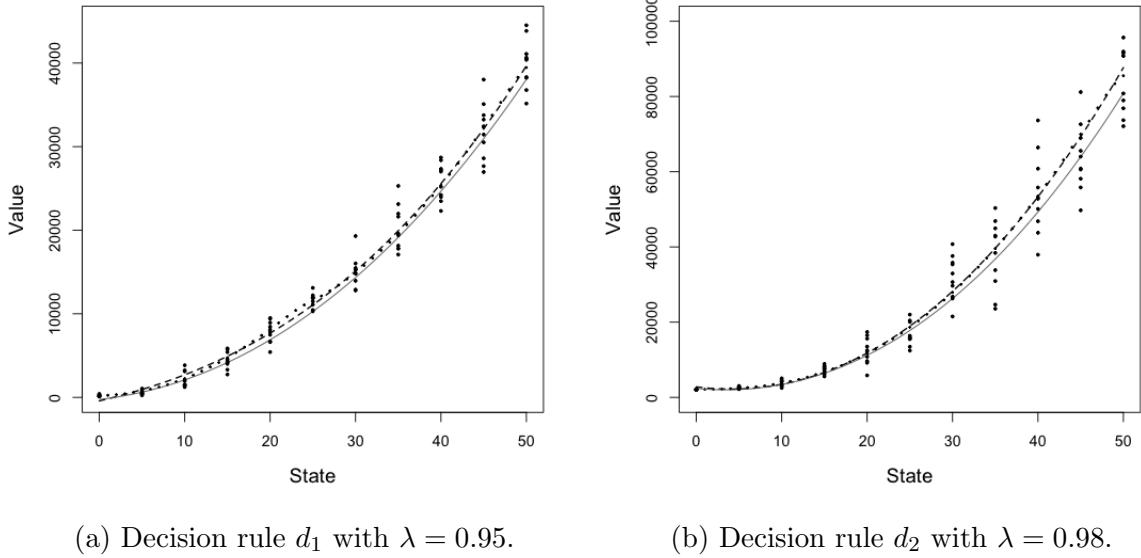


Figure 11.3: Plots of estimated and true policy value functions based on a single replicate of starting-state Monte Carlo estimation consisting of $K = 10$ episodes for each $s \in \bar{S}$. Points indicate episode value estimates, the dotted line equals the true policy value function, the dashed line indicates the fitted cubic polynomial approximation to the true policy value function and the gray solid line denotes the least squares estimate based on Monte Carlo estimation. Note that the y-axis scales for the two figures differ.

$\beta \in \mathbb{R}^I$ so as to “solve” the policy evaluation (Bellman) equation

$$v(s; \beta) = E^{d^\infty} [r(X, Y, X') + \lambda v(X'; \beta) | X = s]$$

defined for all $s \in S$. In this equation the expectation is with respect to the conditional distribution of the action Y and successor state X' under decision rule d given state $s \in S$. To make this more concrete, the goal is to find a β that minimizes the conditional expected squared error loss

$$E^{d^\infty} [(r(X, Y, X') + \lambda v(X'; \beta) - v(X; \beta))^2 | X = s] \quad (11.12)$$

for all $s \in S$.

Since a different β may achieve the minimum above for each $s \in S$, an alternative is to choose β to minimize the *unconditional* squared error loss:

$$G_d(\beta) := E^{d^\infty} [(r(X, Y, X') + \lambda v(X'; \beta) - v(X; \beta))^2], \quad (11.13)$$

where the expectation is now with respect to state X as well. Possible choices for a distribution for X include a uniform distribution over states or the stationary distribution of the Markov chain corresponding to d^∞ .

If one could evaluate the above expectation directly, the gradient descent recursion

$$\boldsymbol{\beta}' = \boldsymbol{\beta} - \tau \nabla_{\boldsymbol{\beta}} G_d(\boldsymbol{\beta})$$

would provide an estimate of $\boldsymbol{\beta}$.

An alternative approach suitable for use on trajectory-based data can be obtained as follows. Given a value of $\boldsymbol{\beta}$, suppose in state s one observes a pair (r, s') , evaluates $u := r + \lambda v(s'; \boldsymbol{\beta})$ and seeks a vector $\boldsymbol{\beta}$ to minimize

$$g(\boldsymbol{\beta}) := (u - v(s; \boldsymbol{\beta}))^2,$$

regarding the scalar u as a fixed value independent of $\boldsymbol{\beta}$ (which of course it is not).

In this case, temporal differencing applies gradient descent to $g(\boldsymbol{\beta})$ by taking the gradient of $g(\boldsymbol{\beta})$,

$$\nabla_{\boldsymbol{\beta}} g(\boldsymbol{\beta}) = -2(u - v(s; \boldsymbol{\beta})) \nabla_{\boldsymbol{\beta}} v(s; \boldsymbol{\beta})$$

and using the recursion

$$\boldsymbol{\beta}' = \boldsymbol{\beta} + \tau(u - v(s; \boldsymbol{\beta})) \nabla_{\boldsymbol{\beta}} v(s; \boldsymbol{\beta}), \quad (11.14)$$

where the factor 2 is absorbed into τ .

When $v(s; \boldsymbol{\beta})$ is a linear function of $\boldsymbol{\beta}$, $v(s; \boldsymbol{\beta}) = \boldsymbol{\beta}^T \mathbf{b}(s)$ and $\nabla_{\boldsymbol{\beta}} v(s; \boldsymbol{\beta}) = \mathbf{b}(s)$, in which case the gradient descent recursion becomes

$$\boldsymbol{\beta}' = \boldsymbol{\beta} + \tau(u - \boldsymbol{\beta}^T \mathbf{b}(s)) \mathbf{b}(s). \quad (11.15)$$

Another way to think of (11.15) is to treat u as a “target” and at each iteration $\boldsymbol{\beta}$ is modified so as to move $v(s; \boldsymbol{\beta})$ closer to its target. The reinforcement learning literature refers to this approach as *bootstrapping* because it uses an approximation to the policy value function in the target.

The scalar quantity $\delta := r + \lambda v(s'; \boldsymbol{\beta}) - v(s; \boldsymbol{\beta})$ is often referred to as a *temporal difference* or *Bellman error* and the following recursion is referred to as temporal differencing or TD(0) with function approximation. Using this additional notation, the recursion can be represented by:

$$\boxed{\boldsymbol{\beta}' = \boldsymbol{\beta} + \tau \delta \mathbf{b}(s).} \quad (11.16)$$

Note that in this expression τ and δ are scalars, while $\boldsymbol{\beta}$ and $\mathbf{b}(s)$ are column vectors of the same dimension.

Gradients and semi-gradients

The following discussion describes how gradient descent is usually employed in reinforcement learning recursions.

Given a simulated update, $r + v(s'; \boldsymbol{\beta})$, to minimize the squared-error loss

$$g(\boldsymbol{\beta}) = (r + v(s'; \boldsymbol{\beta}) - v(s; \boldsymbol{\beta}))^2$$

using gradient descent, the gradient of $g(\boldsymbol{\beta})$ is given by

$$\nabla_{\boldsymbol{\beta}} g(\boldsymbol{\beta}) = 2(r + v(s'; \boldsymbol{\beta}) - v(s; \boldsymbol{\beta})) (\nabla_{\boldsymbol{\beta}} v(s'; \boldsymbol{\beta}) - \nabla_{\boldsymbol{\beta}} v(s; \boldsymbol{\beta})). \quad (11.17)$$

In this expression the gradient includes **two** terms that correspond to the gradient of $v(\cdot; \boldsymbol{\beta})$ evaluated at two states, s' and at s . Temporal differencing is based on *ignoring the first expression* $\nabla_{\boldsymbol{\beta}} v(s'; \boldsymbol{\beta})$, that is, it approximates the gradient of $g(\boldsymbol{\beta})$ by

$$\nabla_{\boldsymbol{\beta}} g(\boldsymbol{\beta}) \approx -2(r + v(s'; \boldsymbol{\beta}) - v(s; \boldsymbol{\beta})) \nabla_{\boldsymbol{\beta}} v(s; \boldsymbol{\beta}). \quad (11.18)$$

The expression in (11.18) is referred to in the reinforcement learning literature as a *semi-gradient*. Using the semi-gradient in TD(0) is equivalent to treating the sampled update (target) $r + v(s'; \boldsymbol{\beta})$ as fixed and independent of $\boldsymbol{\beta}$.

When features are chosen to be indicator functions of states and $v(s; \boldsymbol{\beta})$ is a linear function of the features, the components of $\boldsymbol{\beta}$ correspond to the value function. Hence the semi-gradient also underlies tabular TD(0), Q-learning, and SARSA first described in Chapter 10.

What is the effect of using the semi-gradient instead of the gradient?

1. Using the semi-gradient is simpler because it avoids evaluating the gradient at *both* s and s' .
2. Despite being an approximation, the semi-gradient is often used in practice due to its *computational efficiency and empirical success*, especially in large-scale or deep reinforcement learning problems.
3. In trajectory-based on-policy applications, that is when the trajectory is generated by the policy being evaluated, both methods typically give similar results.
4. In off-policy applications using function approximation, that is when the trajectory is generated by a different policy than the one being evaluated, semi-gradient methods may diverge⁸.
5. Semi-gradient methods do not correspond to the gradient of any objective function in general. This limits their theoretical justification and makes convergence analysis more difficult.
6. Semi-gradients underlie the recursions in both SARSA and Q-learning.

Exercise 17 asks you to investigate some of these issues in an example.

⁸See Baird [1995].

An algorithm

The following algorithm implements TD(0) in a discounted model. It evaluates a Markovian random decision rule d and allows for flexibility in selecting the sequence of states. It assumes features are fixed and pre-specified.

Algorithm 11.4. TD(0) with linear policy value function approximation for a discounted model

1. **Initialize:**

- (a) Specify $d \in D^{\text{MR}}$, β arbitrary, and $s \in S$.
- (b) Specify the number of iterations N and set $n \leftarrow 1$.
- (c) Specify the learning rate sequence $\tau_n, n = 1, 2, \dots$

2. **Iterate:** While $n \leq N$:

- (a) Sample a from $w_d(\cdot|s)$.
- (b) Simulate (s', r) or sample $s' \in S$ from $p(s'|s, a)$ and set $r \leftarrow r(s, a', s')$.
- (c) **Compute the temporal difference:** Set $v(s; \beta) \leftarrow \beta^T \mathbf{b}(s)$, $v(s'; \beta) \leftarrow \beta^T \mathbf{b}(s')$ and

$$\delta \leftarrow r + \lambda v(s'; \beta) - v(s; \beta). \quad (11.19)$$

(d) **Update β :**

$$\beta \leftarrow \beta + \tau_n \delta \mathbf{b}(s). \quad (11.20)$$

(e) **Next state generation:**

- (Trajectory-based) $s \leftarrow s'$, or
- (State-based) Generate s from a uniform distribution on S , or
- (Hybrid) Specify ϵ small and a state s_0 . With probability ϵ , $s \leftarrow s_0$ and with probability $1 - \epsilon$, $s \leftarrow s'$.

- (f) $n \leftarrow n + 1$.

3. **Terminate:** Return β .

Some comments on implementing this algorithm follow:

1. **Specifying a decision rule to evaluate:** The algorithm is expressed in a form that assumes $d(s)$ can be explicitly specified for all $s \in S$. Unfortunately when S is large, this may not be possible. Optimization algorithms, described below, get around this requirement in different ways.

Q-learning chooses a' using ϵ -greedy or softmax sampling based on an estimated state-action value function $q(s, a; \beta)$.

Actor-critic⁹ samples a' from $w(\cdot | s; \beta^C)$ where the estimate of the weight β , denoted by β^C , is updated using gradient ascent.

2. **Convergence:** When $v(s; \beta)$ is a linear function of β and the features are linearly independent, the iterates of β converge¹⁰ with probability 1 provided the samples follow the trajectory of the Markov chain corresponding to d and $\tau_n, n = 1, 2, \dots$, satisfies the Robbins-Monro conditions. Of course, this also means that the corresponding policy value functions converge.
3. **Initialization:** As in most nonlinear optimization problems, convergence is sensitive to the initial specification of β . A limited Monte Carlo analysis based on one or more replicates for a selected subset of states followed by least squares parameter estimation may provide reliable starting values.
4. **Stopping Rules:** They can be based on either changes in parameter values

$$\left(\frac{1}{I} \sum_{i=1}^I (\beta'_i - \beta_i)^2 \right)^{\frac{1}{2}} \quad (11.21)$$

or changes in fitted values:

$$\left(\frac{1}{|S|} \sum_{s \in S} (v(s; \beta') - v(s; \beta))^2 \right)^{\frac{1}{2}}. \quad (11.22)$$

or a weighted variant thereof. Use of these criteria have limitations: (11.21) may be dominated by the impact of a few large coefficients, while (11.22) may be computationally prohibitive due to the magnitude of S . Note that (11.22) may be more appropriate because it is consistent with the least squares objective function used to derive TD(0).

5. **State updating:** Step 2(e) provides three approaches for generating “subsequent” states for evaluation: following the trajectory, randomly restarting, or a combined approach. When this algorithm is used in a simulation environment such as in the queuing service rate control example immediately below, the random restart and hybrid methods will provide better coverage of S , especially if s_0 is chosen judiciously. This is because under a specified decision rule, the process may occupy only a small portion of the state space. In real-world implementations, random restarts may be difficult to implement so that the long-trajectory approach may be necessary. The hybrid approach provides a restart method that might be easier to implement.

⁹See Section 11.6 below.

¹⁰See Tsitsiklis and Roy [1997].

6. **Episodic models:** Since episodic models terminate at a state in Δ after a finite (but random) number of iterations, it is necessary to provide a mechanism to generate enough data to accurately estimate parameters. One possibility is to jump to a random state after termination. Also, in an episodic model, λ may be set equal to 1.

Policy value function approximation in the queuing service rate control model

This example applies Algorithm 11.4 to the model in Example 11.2 with $\lambda = 0.95$. It uses a cubic polynomial approximation to the policy value function of three deterministic stationary policies derived from the decision rules:

$$d_1(s) = \begin{cases} a_1 & 0 \leq s \leq 25 \\ a_3 & 26 \leq s \leq 50 \end{cases}$$

$$d_2(s) = \begin{cases} a_2 & 0 \leq s \leq 50 \end{cases}$$

$$d_3(s) = \begin{cases} a_2 & 0 \leq s \leq 9 \\ a_3 & 10 \leq s \leq 29 \\ a_1 & 30 \leq s \leq 50 \end{cases}$$

Step 2(e) used trajectory-based, state-based and a hybrid variant that with probability 0.008 causes the process to jump to $s_0 = 50$ and with probability 0.992 followed the existing trajectory. Note that under d_3 choosing $s_0 = 0$ would be more appropriate.

Preliminary analysis suggested the following four learning rates for further evaluation: $\tau_n = 0.1n^{-0.5}$, $0.1 \log(n+1)/n$, $0.1/(1+10^{-5}n^2)$, $150/(750+n)$ referred to respectively as polynomial, logarithmic, STC and ratio. Experiments compared all 36 combinations of decision rule, state updating method and learning rate over 40 replicates of length $N = 20,000$ using common random number seeds for all instances in a replicate.

The “convergence” of estimates was highly sensitive to starting values and feature scaling so that the following steps were taken to obtain initial weights:

1. Weights were initialized with one replicate of starting-state Monte Carlo implemented on a subset $\bar{S} = \{5, 15, 25, 35, 45\}$ of S ;
2. States were scaled so as to have mean zero and standard deviation one, and
3. Least squares estimation was used to obtain preliminary estimates of the parameters of a cubic polynomial approximation.

Figure 11.4 compares the effect of the factors on the quality of the fit for each configuration. Observations include:

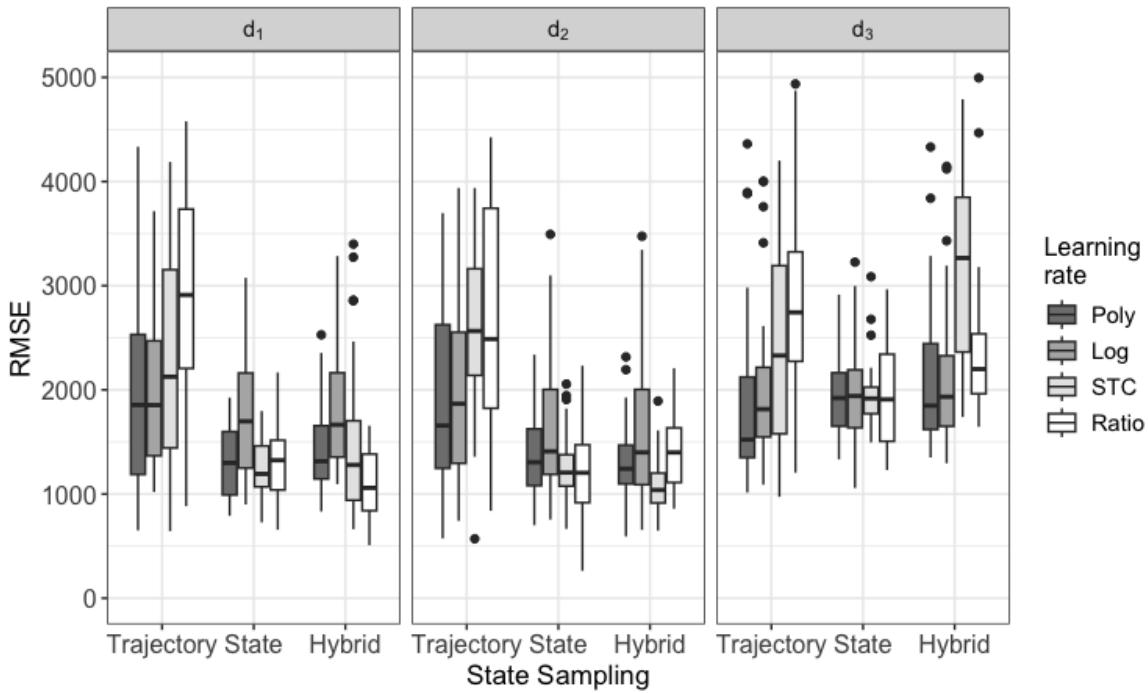


Figure 11.4: Comparison of the RMSEs of policy value function estimates for the queuing control model. To put results in perspective, Table 11.1 showed that a cubic polynomial approximation derived from Monte Carlo estimates had an average RMSE of 1,029.04 for d_1 and 1,098.57 for d_2 .

1. With the exception of trajectory-based sampling, value function estimates for d_1 and d_2 had smaller RMSEs than for d_3 . As d_3 is unlikely to arise in optimization, the following observations focus on d_1 and d_2 .
2. Trajectory-based sampling resulted in the least accurate and most variable policy value function estimates for d_1 and d_2 .
3. For d_1 and d_2 , state-based sampling using a uniform distribution and hybrid state sampling produced similar results. For d_3 , state-based sampling was less sensitive to learning rate than trajectory-based or hybrid sampling.
4. With state-based sampling, the effect of learning rate was similar for d_1 and d_2 . Estimates based on STC were least variable and most accurate.
5. For hybrid sampling, the quality of estimates varied with decision rule. For d_1 , the ratio learning rate resulted in the most accurate estimates and for d_2 , the STC estimate was the most accurate and least variable.

Table 11.2 provides the RMSE of the policy value function estimate obtained from five randomly selected replicates. Each replicate used decision rule d_1 and the STC

Method	Replicate				
	1	2	3	4	5
MC - initialization only	1,209	1,259	794	973	2,111
MC + TD(0) - hybrid	967	1,209	873	882	1620
MC + TD(0) - state-based	1,270	1,070	1,441	926	1,131

Table 11.2: RMSE of fitted value functions for 5 replicates for decision rule d_1 . It compares the RMSE of estimates based on Monte Carlo initialization only and Monte Carlo initialization combined with TD(0) using state-based and hybrid state updating.

learning rate, and compared value function estimates obtained by the Monte Carlo initialization only to those based on Monte Carlo initialization followed by TD(0) estimates with state-based and hybrid state generation. As a baseline, the RMSE from fitting a cubic regression function to the true value function is 344.73 so it would be unlikely that any approximation would generate a better fit. Moreover, Monte Carlo estimates obtained previously (Table 11.1) had RMSE 1,029.04 for d_1 and 1,098.57 for d_2 averaged over 40 replicates.

From Table 11.2 observe that:

1. For replicate 3, both TD(0) methods gave a worse fit than the Monte Carlo initialization that was based on one realization at five states. For all other replicates at least one of the TD(0) methods improved the fit from the initial value.
2. There was no clear pattern as to whether hybrid or random state-based updating gave more accurate estimates.
3. Neither method gave estimates with RMSE close to that of the regression fit to the true policy value function.

In conclusion, for this example, estimates based on Algorithm 11.4 were highly variable and not significantly better than Monte Carlo estimates, particularly when using trajectory-based state updating. There was some evidence suggesting that under hybrid state-sampling, a well-chosen learning rate may yield more accurate estimates than when using random state-based sampling. The next section considers a TD(γ) variant.

11.2.3 TD(γ) with linear policy value function approximation

This section extends TD(γ) to models with policy value function approximation. It presents an algorithm, provides some comments and includes an illustrative example. The focus is restricted to discounted models with linear policy value function approximations. This chapter uses the vector \mathbf{z} to represent the eligibility trace to avoid confusion with the vector \mathbf{e} of all ones.

Algorithm 11.5. TD(γ) for linear policy value function approximation for a discounted model

1. **Initialize:**

- (a) Specify $d \in D^{\text{MR}}$, β arbitrary, and $s \in S$.
- (b) Specify the number of iterations N and set $n \leftarrow 1$.
- (c) Set the eligibility trace vector $\mathbf{z} \leftarrow \mathbf{0}$ and specify $\gamma \in [0, 1]$.
- (d) Specify the learning rate sequence $\tau_n, n = 1, 2, \dots$

2. **Iterate:** While $n \leq N$:

- (a) Sample a from $w_d(\cdot|s)$.
- (b) Simulate (s', r) or sample $s' \in S$ from $p(\cdot|s, a)$ and set $r \leftarrow r(s, a, s')$.
- (c) **Compute the temporal difference:** Set $v(s; \beta) \leftarrow \beta^\top \mathbf{b}(s)$, $v(s'; \beta) \leftarrow \beta^\top \mathbf{b}(s')$ and

$$\delta \leftarrow r + \gamma v(s'; \beta) - v(s; \beta). \quad (11.23)$$

(d) **Update eligibility trace:**

$$\mathbf{z} \leftarrow \gamma \lambda \mathbf{z} + \mathbf{b}(s). \quad (11.24)$$

(e) **Update β :**

$$\beta \leftarrow \beta + \tau_n \delta \mathbf{z}. \quad (11.25)$$

(f) **Next state generation:**

(Trajectory-based) $s \leftarrow s'$, or

(State-based) Generate s from a uniform distribution on S , or

(Hybrid) Specify ϵ small and a state s_0 . With probability ϵ , $s \leftarrow s_0$ and with probability $1 - \epsilon$, $s \leftarrow s'$.

(g) $n \leftarrow n + 1$.

3. **Terminate:** Return β .

Some comments follow:

1. The eligibility trace is a weighted linear combination of past feature vectors. The smaller the value of γ , the more quickly the effect of past features dies out. Note that when using nonlinear policy value function approximations, the gradient of the value function (with respect to its parameters) replaces the feature vector in (11.24).

2. When $\gamma = 0$, the algorithm reduces to $\text{TD}(0)$.
3. It is left as an exercise to show that this algorithm reduces to Algorithm 10.6 when features are indicator functions of individual states.

Example 11.3. This example investigates the application of $\text{TD}(\gamma)$ to the queuing model. Its primary purpose is to examine if and when there is an increase in accuracy or a reduction in variability that results from using $\text{TD}(\gamma)$ with $\gamma > 0$ instead of $\text{TD}(0)$.

Previous analysis suggested the best estimates occurred with STC and ratio learning rates. Accordingly, this example compares the impact on estimation of using these two learning rates, $\gamma \in \{0, 0.3, 0.6\}$ and three state-updating regimes. Each of 40 replicates sets $N = 20,000$ and uses decision rule d_1 and common random number seeds. Estimates are compared on the basis of the RMSE of the fitted values. Figure 11.5 summarizes results graphically.

Observe that the effect of γ varied with learning rate and state updating method:

1. The effect of γ was inconsistent.
 - (a) For trajectory-based state updating with a ratio learning rate, the estimates when $\gamma > 0$ were more accurate than when $\gamma = 0$. However, this effect was not observed for the STC learning rate.
 - (b) For state-based updating, the effect of γ was similar for both learning rates. Estimates based on $\gamma = 0.3$ were more accurate and less variable than those using $\text{TD}(0)$ but the effect was small.
 - (c) For hybrid updating, the effect of γ varied with the learning rate. For the STC learning rate, $\gamma = 0.6$ produced the most accurate and least variable estimates. However, when using the ratio learning rate, the $\text{TD}(0)$ estimates were most accurate and least variable.
2. As before, trajectory-based sampling produced the least accurate and most variable estimates and state-based uniform random sampling produced the most accurate and least variable estimates. Estimates using hybrid state-updating behaved similarly to state-based updating although they were more variable.
3. Overall, the STC learning rate appeared to be preferable to the ratio learning rate. With this learning rate, there was a small benefit to using $\gamma > 0$ for state-based and hybrid state updating methods. For trajectory-based sampling, the effect of γ was minimal.

Overall, it appears there are small benefits of using $\text{TD}(\gamma)$ with $\gamma > 0$, but the effect varies with the configuration. It is unlikely that the analyst would be able to anticipate a priori when that might be. The effect of the state updating method is far greater.

Analysis of variance methods would allow a formal analysis of the above observations. It seems likely that only some of the results observed above would be statistically significant.

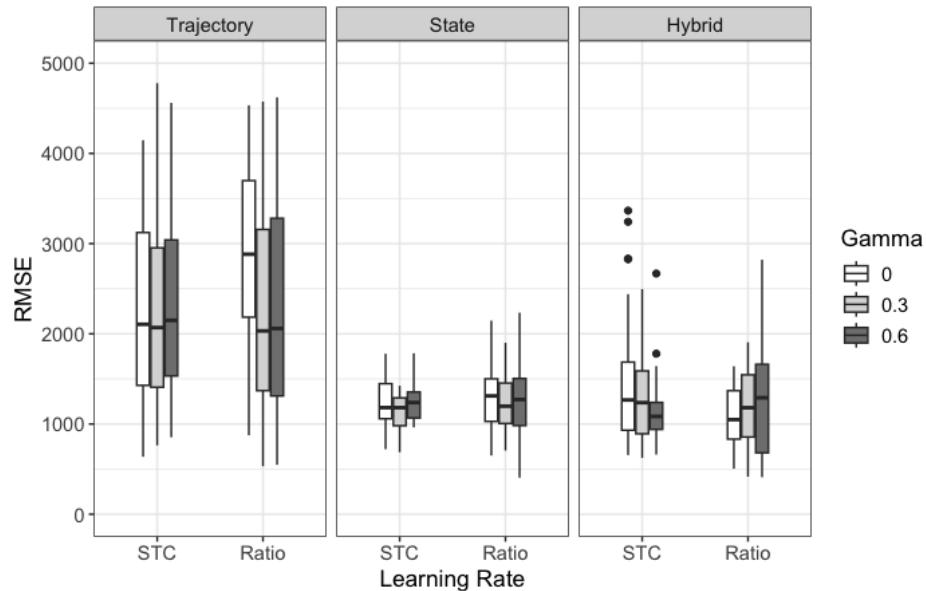


Figure 11.5: Boxplots of the RMSEs of the $\text{TD}(\gamma)$ estimates based on 40 replicates with two learning rates and three state generation methods for the model in Example 11.3.

From this example, it appears that the effect of using $\text{TD}(\gamma)$ with $\gamma > 0$ is small. Moreover, applying it could be problematic when the dimension of the eligibility trace, \mathbf{z} , is high, which is the case when using large neural network value function representations.

11.3 Optimization based on value function approximation: Q-learning

This section extends tabular Q-learning to settings in which state-action value functions are approximated by functions of features. An algorithm is provided for discounted models. It can be easily generalized to episodic models by adding a mechanism to

choose the starting state for subsequent episodes. Examples illustrate both the discounted and episodic cases.

11.3.1 Motivation

Q-learning in the discounted case is based on the Bellman equation for optimal state-action value functions. It was shown in Chapter 5 that the Bellman equation can be expressed as

$$q(s, a) = \sum_{j \in S} p(j|s, a)(r(s, a, j) + \lambda \max_{a \in A_s} q(j, a))$$

and in expectation form as

$$q(s, a) = E \left[r(X, Y, X') + \lambda \max_{a' \in A_{X'}} q(X', a') \mid X = s, Y = a \right]. \quad (11.26)$$

Recall that its unique solution satisfies

$$q^*(s, a) = \sum_{j \in S} p(j|s, a)(r(s, a, j) + \lambda v^*(j))$$

or in expectation form as

$$q^*(s, a) = E \left[r(X, Y, X') + \lambda v^*(X') \mid X = s, Y = a \right], \quad (11.27)$$

where $v^*(\cdot)$ denotes the optimal value function. Note that the expectation representation is interpretable directly in a model-free environment.

State-action value function approximation replaces $q(s, a)$ by $q(s, a; \beta)$ in (11.26) to obtain

$$q(s, a; \beta) = E \left[r(X, Y, X') + \lambda \max_{a' \in A_{X'}} q(X', a'; \beta) \mid X = s, Y = a \right]. \quad (11.28)$$

As in the case of a tabular model, Q-learning seeks to find a β that minimizes the discrepancy between the two-sides of (11.28) in the squared error sense as

$$E \left[\left(r(X, Y, X') + \lambda \max_{a' \in A_X} q(X', a'; \beta) - q(X, Y; \beta) \right)^2 \mid X = s, Y = a \right], \quad (11.29)$$

where the expectation is with respect to the distribution $p(\cdot|s, a)$.

The semi-gradient¹¹ of this expression with respect to β (see (11.18)) is written as

$$-2E \left[\left(r(s, a, X') + \lambda \max_{a' \in A_{X'}} q(X', a'; \beta) - q(s, a; \beta) \right) \nabla_\beta q(s, a; \beta) \right].$$

¹¹The full gradient would include an additional term corresponding to the gradient of $\lambda \max_{a' \in A_{X'}} q(X', a'; \beta)$.

When $q(s, a)$ is approximated by a linear function of features $\mathbf{b}(s, a)$, the above expression becomes

$$-2E \left[\left(r(s, a, X') + \lambda \max_{a' \in A_{X'}} \boldsymbol{\beta}^\top \mathbf{b}(X', a') - \boldsymbol{\beta}^\top \mathbf{b}(s, a) \right) \mathbf{b}(s, a) \right].$$

This expression leads to the following recursion for estimating $\boldsymbol{\beta}$:

$$\boldsymbol{\beta} \leftarrow \boldsymbol{\beta} + \tau \left(r(s, a, s') + \lambda \max_{a \in A_{s'}} \boldsymbol{\beta}^\top \mathbf{b}(s', a) - \boldsymbol{\beta}^\top \mathbf{b}(s, a) \right) \mathbf{b}(s, a), \quad (11.30)$$

where s' is the result of state sampling. Note that in the above expression $\boldsymbol{\beta}$ and $\mathbf{b}(s, a)$ are column vectors and the expression in parentheses is a scalar.

11.3.2 Discounted Q-learning with function approximation

The following Q-learning algorithm seeks to find the coefficients of a linear approximation to an optimal state-action value function $q^*(s, a)$ for a fixed set of features. The output of the algorithm is a vector $\hat{\boldsymbol{\beta}}$ of weights that can be used in subsequent applications to determine action choice in state s according to

$$\hat{a}_s \in \arg \max_{a \in A_s} q(s, a; \hat{\boldsymbol{\beta}}) \quad (11.31)$$

or randomly using softmax sampling based on $q(s, a; \hat{\boldsymbol{\beta}})$. Moreover, the corresponding optimal value function approximation can be obtained from

$$\hat{v}(s) = \max_{a \in A_s} q(s, a; \hat{\boldsymbol{\beta}}). \quad (11.32)$$

Algorithm 11.6. Q-learning with linear state-action value function approximation for a discounted model

1. **Initialize:**

- (a) Specify $\boldsymbol{\beta}$.
- (b) Specify the learning rate $\tau_n, n = 1, 2, \dots$
- (c) Specify the number of iterations N .
- (d) Specify a sequence $\epsilon_n, n = 1, 2, \dots$ (or η_n for softmax sampling).
- (e) Specify $s \in S$.
- (f) $n \leftarrow 1$.

2. **Iterate:** While $n \leq N$:

(a) Generate $a \in A_s$ randomly using Algorithm 11.1 applied to β in state s .

(b) Simulate (s', r) or sample $s' \in S$ from $p(\cdot|s, a)$ and set $r \leftarrow r(s, a, s')$.

(c) For all $a \in A_{s'}$

$$q(s', a; \beta) \leftarrow \beta^T \mathbf{b}(s', a). \quad (11.33)$$

(d) Set

$$\delta \leftarrow r + \lambda \max_{a' \in A_{s'}} q(s', a'; \beta) - q(s, a; \beta). \quad (11.34)$$

(e) **Update β :**

$$\beta \leftarrow \beta + \tau_n \mathbf{b}(s, a) \delta. \quad (11.35)$$

(f) $s \leftarrow s'$.

(g) $n \leftarrow n + 1$.

3. **Terminate:** Return β .

Some comments about this algorithm follow:

1. The algorithm requires ϵ -greedy or softmax action selection to ensure exploration. If only greedy actions were chosen in step 2(a), the algorithm would most likely end up evaluating a sub-optimal policy.
2. When computing the learning rate, the index is chosen to be the iteration number to avoid storing the number of visits to each state-action pair. In tabular models, using the frequency of visits to each state-action pair may be preferable since, unlike the more general parametric function approximation cases considered in this chapter, the values for state-action pairs are only updated when that state-action pair is observed. When using function approximation, weights are updated at every iteration so they should stabilize faster. Consequently, smaller learning rates and ϵ -greedy parameters can be used.
3. As stated, the algorithm is trajectory-based, that is, it selects sequences of states according to the underlying transition probabilities. Note that in step 2(f), $s \leftarrow s'$ can be replaced by either state-based or hybrid updating.
4. The above algorithm never explicitly computes a decision rule. Action selection in step 2(a) implicitly corresponds to a policy but as β changes, the elusive policy may change. The policy iteration algorithms below avoid this by fixing β for several iterations.
5. Note this is an off-policy algorithm since the update in (11.35) may be based on a different action than that followed by the trajectory. A SARSA variant, as in Chapter 10, would provide an on-policy alternative.

6. Unlike its tabular counterpart, which corresponds to using state-action indicators as basis functions in the above algorithm, there are no general convergence guarantees for Q-learning with function approximation.
7. The algorithm can be easily modified to allow for nonlinear state-action value function approximations. This would require replacing 2(c) to allow for a nonlinear functional form and replacing $\mathbf{b}(s, a)$ in 2(e) by the gradient of the nonlinear function.

Assessing policies in computational experiments

This brief digression offers insights into assessing the quality of policies generated by an optimization algorithm. Suppose, as in the example below, the optimal value $v^*(s)$ is known. Let d^∞ be a policy generated by some approximation method and let its true value function¹² be given by $v'(s) := v^{d^\infty}(s)$. Methods for comparing these values were discussed in Section 10.11.1 and are expanded on here.

A key issue is that under d^∞ , the system most likely visits only a small region of the state space, say S' . For example, in the queuing service rate control model, the system will spend most of its time in low-occupancy states. Thus, the degree of sub-optimality of the approximate policy should be assessed based on a weighted combination of differences of values in states in S' . This is exactly what the weighted RMSE metric, denoted by wRMSE, defined in (10.102) achieves when weights $w(s)$ are chosen to be the stationary distribution¹³ of the implemented policy.

In contrast, using an unweighted RMSE can lead to over-weighting discrepancies in infrequently visited states thereby providing a misleading belief on the loss of optimality faced by the system under d^∞ .

An example

The example in this section applies Q-learning with value function approximation to the infinite horizon discounted queuing service rate control model.

Example 11.4. Consider the queuing service rate control model on $S = \{0, \dots, 50\}$. Basis functions are of the form (11.3) where the terms in $f_i(s)$ equal the scaled powers of a cubic polynomial and the terms in $h_j(a)$ represent indicator functions of each possible action. This is equivalent to representing $q(s, a)$ by a cubic polynomial in which the weights vary with the action.

As seen in the example in Section 11.2.2, good starting values are required to ensure convergence. By using the above specification for the approximation,

¹²This could be computed by solving policy evaluation equations or approximated by simulation.

¹³Recall that these can be found by solving $\mathbf{w}^\top = \mathbf{w}^\top \mathbf{P}$ subject to $\mathbf{w}^\top \mathbf{e} = 1$ or approximated by raising \mathbf{P} to a large power.

preliminary weight estimates for the cubic polynomial approximation can be obtained using Monte Carlo estimation for each action separately or alternative randomizing over all actions with equal probability. Note that these estimates are distinct from the optimal state-action value functions sought by the algorithm because after choosing a state and action, they follow the policy value function of the policy used to obtain starting values, not the optimal value function.

The algorithm required considerable tuning to obtain convergence to reasonable policies. Results are described for $N = 150,000$, learning rates

$$\tau_n = \frac{5,000}{50,000 + n},$$

ϵ -greedy parameter

$$\epsilon_n = \frac{100}{400 + n}$$

for $n = 1, 2, \dots$, and trajectory-based and random state-based state updating. The indices of the learning rate and ϵ -greedy parameter were the iteration number.

For reasons discussed above, results are assessed on the basis of the wRMSE of the greedy policy and the discrepancy between the greedy policy and the known optimal policy. The RMSE is reported for illustrative purposes.

Table 11.3 compares summary measures for the two state updating methods. Policies based on random state-based updating were closer to optimal on all three dimensions and also less variable. To put these quantities in perspective note that the exact optimal value function $v^*(s)$ satisfies $215 \leq v^*(s) \leq 39,365$ and $\sum_{s \in S} w(s)v^*(s) = 502$ for the stationary distribution $w(s)$ based on the optimal policy. For almost any reasonable policy, the system occupies states 0-10 most of the time so inaccuracy in high-occupancy and more-costly states is less important.

Figure 11.6a shows the sorted wRMSE values in ascending order. It shows the wRMSE was less than 300 in all but three replicates.

Figure 11.6b displays the number of states at which the greedy policy differed from the optimal policy under state-based sampling. For both state sampling methods greedy policies differed from the optimal policy in *all* replicates. Under state-based sampling, in 35 out of 40 replicates the greedy-policy had the same structure as the optimal policy; however, the states at which actions changed differed.

Consequently, from the perspective of near optimality, state-based sampling produced satisfactory results. However, this approach failed to identify the optimal policy in all cases. In contrast, Q-learning using a tabular representation was shown in Section 10.9 to be considerably more accurate. It is left to the reader to explore other approximations.

State updating	RMSE	wRMSE	Non-optimal actions
State-based	28.85 (37.47)	168.41 (133.03)	11.38 (4.93)
Trajectory-based	138.84 (80.36)	332.98 (567.03)	25.70 (10.33)

Table 11.3: Comparison of the mean (standard deviation) of the summary measures obtained from Example 11.4 broken down by state-updating method.

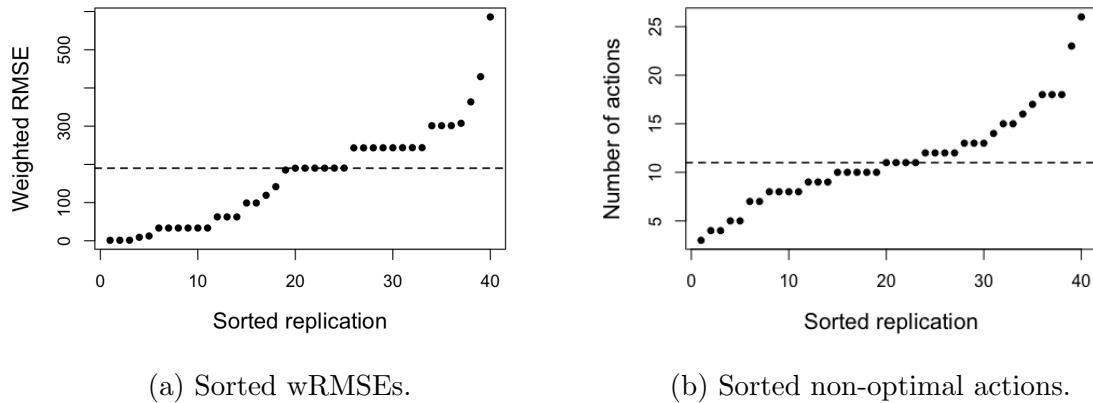


Figure 11.6: Plots of the sorted wRMSE values and the number of non-optimal actions for 40 replicates in Example 11.4 using state-based updating. In each plot the horizontal dashed line denotes the corresponding median.

11.3.3 Q-learning in an episodic model

This example applies Q-learning to an episodic model. It considers a shortest path problem on a rectangular grid with M rows and N columns. In each cell, the robot can *try* to move up, down, right or left. If a move is impeded by a boundary, the robot remains in its current location. The goal is to reach a pre-specified cell (m^*, n^*) . When that cell is reached the robot receives a reward of R . The cost of each attempted step is c (corresponding to a reward of $-c$). The objective is to maximize the expected total reward.

Markov decision process formulation

An MDP formulation follows:

States: $S = \{(m, n) \mid m = 1, \dots, M, n = 1, \dots, N\}$

Actions: For all $s = (m, n) \in S$, $A_s = A$ where

$$A = \begin{cases} \{\text{up, down, left, right}\} & (m, n) \neq (m^*, n^*) \\ \{\delta\} & (m, n) = (m^*, n^*) \end{cases}$$

Rewards:

$$r((m, n), a, (m', n')) = \begin{cases} -c & (m', n') \neq (m^*, n^*), a \in A \\ R - c & (m, n) \neq (m^*, n^*), (m', n') = (m^*, n^*), a \in A \\ 0 & (m, n) = (m^*, n^*), a = \delta \end{cases}$$

Transition probabilities:

$$p((m', n')|(m, n), a) = \begin{cases} 1 & \text{if move from } (m, n) \text{ to } (m', n') \text{ is possible under action } a \\ 0 & \text{otherwise.} \end{cases}$$

Note that the transition probabilities are awkward to write down explicitly because there are many cases to enumerate¹⁴, but easy to code in a simulation.

This model can be analyzed as an undiscounted total return model in which the action choice probabilities introduce randomness in transitions. There are many improper policies, each having reward $-\infty$, however, there exist (many) deterministic optimal policies that can be easily found by inspection.

Feature choice

This example compares three implementations of Q-learning differing by the choice of the form of the approximate state-action value function. All are based on representation (11.4) in which $h_j(a)$ is an indicator function of the action. That is

$$h_1(a) := I_{\{\text{up}\}}(a), \quad h_2(a) := I_{\{\text{down}\}}(a), \quad h_3(a) := I_{\{\text{left}\}}(a), \quad h_4(a) := I_{\{\text{right}\}}(a)$$

for $a \in A$. They differ in the form of the features $f_i(m, n)$ as follows:

1. Indicator functions of each cell (tabular model):

$$f_i(m', n') = I_{\{(m, n)\}}(m', n')$$

for each $(m, n) \in S$. For each action there are $M \times N$ features (i.e., $i = 1, \dots, MN$). In code, it might be more convenient to index $f_i(m', n')$ by the row and column index.

¹⁴For example, if the agent is at the left boundary, that is in state $(m, 1)$, and tries to move left, it remains in state $(m, 1)$ but if it tries, for example, to move right, a transition to $(m, 2)$ occurs.

2. Indicator functions of each row and column:

$$f_i(m', n') = \begin{cases} I_{\{m\}}(m', n') & \text{for } m = 1, \dots, M \text{ and } i = 1, \dots, M, \\ I_{\{n\}}(m', n') & \text{for } n = 1, \dots, N \text{ and } i = M + 1, \dots, M + N. \end{cases}$$

Note that in this case there are $M + N$ features for each action.

3. A parametric function of the row and column number:

Motivated by the calculations in the tabular case that are described below, it was hypothesized that linear features with an interaction term would well approximate the true state-action value function when combined with indicators functions of actions.

$$f_0(m, n) := 1, \quad f_1(m, n) := m, \quad f_2(m, n) := n, \quad f_3(m, n) := mn.$$

This means that for each action, $q(s, a; \beta)$ is approximated by a linear function of 1, m , n , and mn with coefficients varying across actions.

It is possible to also add higher order terms or even some judiciously chosen indicator functions. Note that when M and/or N are large, it may be necessary to scale the row and column index.

Results: Tabular model

Calculations for the tabular model apply Algorithm 11.6 to two instances with $(m^*, n^*) = (M, N)$, $R = 10$ and $c = 0.1$. In this application the robot seeks to learn a path to cell (M, N) from the starting cell $(1, 1)$.

It uses ϵ -greedy action selection with $\epsilon_n = 10/(100 + n)$ and a learning rate of $\tau_n = 50/(1,000 + n)$ where n denotes the episode number. After completion of an episode, the next episode starts from cell $(1, 1)$ ¹⁵. The experiments used a discount rate of $\lambda = 1$, 10,000 episodes and set all $q((m, n), a) = 0$ for initialization.

For small values of M and N , it was convenient to use the tabular representation of Q-learning to gain some insight into the form of the q -functions. For all replications of the experiment, this approach found an optimal path through the grid.

Figure 11.7 shows the optimal q -functions from a single replicate for $(M, N) = (10, 7)$ and $(M, N) = (25, 10)$. It shows that in both cases, the algorithm identified a policy such that the robot's path was close to the main diagonal of the grid. This suggested that the parametric model above with a cross-product term might identify optimal policies.

¹⁵If one sought a good policy for every starting state, episodes could be restarted at a random non-target cell.

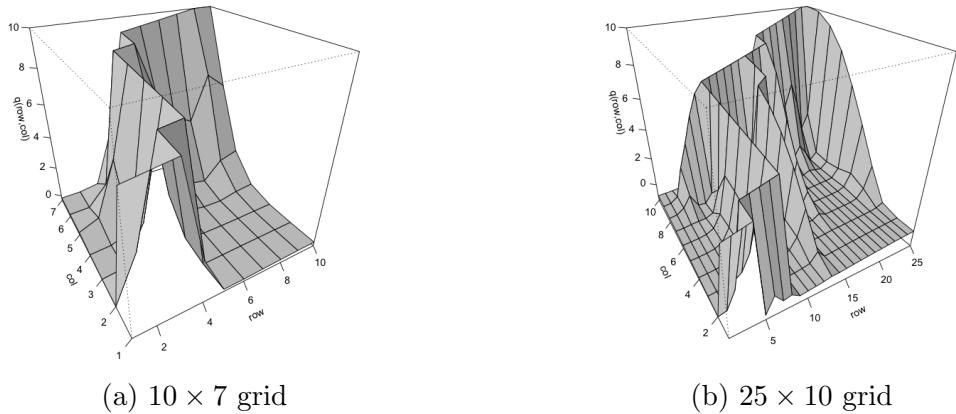


Figure 11.7: Optimal q -values for the Gridworld shortest path model obtained using Q-learning with a tabular representation. Horizontal axes are row and column and vertical axis represents the optimal state-action value function.

Results: Indicator functions of rows and columns

The above calculations were repeated using indicator functions of grid row and grid column using the same choice for ϵ_n , τ_n and the two grid sizes. To reliably find an optimal policy required discounting. Setting $\lambda = 0.95$ gave reliable results.

Unlike the policies generated using the tabular representation, which tended to move up along the main diagonal, the greedy policies chose actions that moved the robot along the boundaries (row 1 and column N or column 1 and row M) of the grid. Moreover, the algorithm found an optimal policy from all starting states even when the target was an interior point of the grid.

Results: Parametric representation

Developing a convergent implementation required considerable experimentation. Issues that arose were divergence of q -values or termination with non-optimal greedy policies. The challenge was a consequence of credit assignment. Because rewards are only received when reaching the goal state, it required many iterations for the impact of the reward in the goal state to impact q -values in distant states. In light of these observations, successful implementations^[16] used:

1. β initialized to be $\mathbf{0}$,
 2. high initial exploration rates ($\epsilon_n = 20/(20 + n)$),
 3. an upper bound (1,000) on the number of steps in an episode,
 4. small learning rates ($\tau_n = 50/(1,000 + n)$),

¹⁶The literature also suggests using experience replay to enhance convergence.

5. a discount factor of 0.9, and
6. random starting states.

With these specifications, greedy policies based on Q-learning always achieved the objective and were optimal when starting in state $(1, 1)$. Moreover, they were similar to those found using the tabular representation, that is, they tended to step up through the interior of the grid. They exhibited fast learning as shown in Figure 11.8.

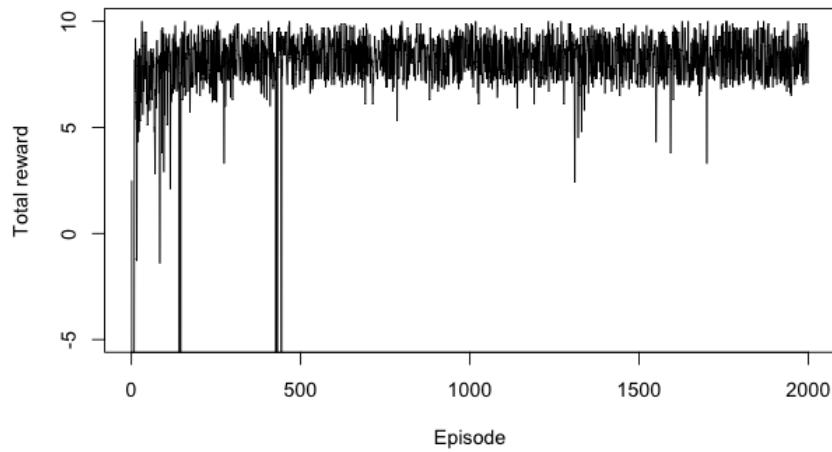


Figure 11.8: Reward per episode in a typical realization of Q-learning in a 10×25 grid. The ongoing variability of the total reward per episode is a result of the random starting state for each episode.

The beauty of using this parameterization was that it was not required to specify or even know the dimension of the grid before implementing Q-learning.

Summary

The above approximations used MN , $M + N$, and four features, respectively, for each action. While the first two approximations, which were based on indicator functions, reliably found optimal policies, those based on a low-dimensional parametric function required considerable experimentation to identify optimal policies. It is left to the reader to explore parametric approximations based on other functional forms or neural networks.

11.4 Q-policy iteration

Policy iteration algorithms require methods for both evaluating and improving stationary policies. This presents challenges for large models in which policies are represented

implicitly through vectors of weights. This section proposes, discusses and applies a weight-based policy iteration algorithm.

11.4.1 Motivation

Ideally, a weight-based policy iteration algorithm should include the following steps:

1. **Initialize:** Select a weight β .
2. **Improve:** Find an “improved” policy using $q(s, a; \beta)$.
3. **Evaluate:** Find β' corresponding to the improved policy.
4. **Iterate:** Repeat.

Implementing such an approach presents challenges, especially when the state space is large and simulated data is used.

If the state space was small, the improve step could be applied in all states using Algorithm 11.1, the weights corresponding to the improved policy could be evaluated by simulation, and the process repeated. Of course, the evaluation step will only generate state-action pairs corresponding to the improved policy so there may not be sufficient coverage of the state-action space to accurately update the weights. An exploration-based alternative would evaluate an ϵ -greedy or softmax-based policy instead.

However, it is not necessary to explicitly generate the policy and then evaluate it. These steps can be combined by fixing the weight at β_0 , using $q(s, a; \beta_0)$ to generate actions, and computing a revised estimate of the weight of this policy using temporal differencing. This is the logic underlying the following *Q-policy iteration* algorithm.

11.4.2 An algorithm

A Q-policy iteration algorithm follows. It is stated for discounted models; generalization to episodic models is direct and left to the reader. The reason why it may perform better than Q-learning is that actions are chosen on the basis of a q -function that approximates the value of the policy implicitly defined by $q(s, a; \beta_0)$ rather than a more unstable estimate that changes every iteration.

Algorithm 11.7. Q-policy iteration with linear state-action value function approximation for a discounted model

1. **Initialize:**
 - (a) Specify β_0 .
 - (b) Specify the learning rate sequence $\tau_n, n = 1, 2, \dots$, and a sequence $\epsilon_n, n = 1, 2, \dots$ (or $\eta_n, n = 1, 2, \dots$, for softmax sampling).

(c) Specify the number of evaluation iterations M and evaluation loops N .

(d) Specify $s \in S$, $k \leftarrow 1$, and $n \leftarrow 1$.

2. **Iterate:** While $n \leq N$:

(a) **Policy evaluation for fixed β_0 :** $m \leftarrow 1$ and $\beta \leftarrow \beta_0$.

(b) Generate $a \in A_s$ randomly using Algorithm 11.1 applied to β_0 in state s .

(c) While $m \leq M$:

i. Simulate (s', r) or sample $s' \in S$ from $p(\cdot|s, a)$ and set $r \leftarrow r(s, a, s')$.

ii. Set

$$q(s, a; \beta_0) \leftarrow \beta_0^T \mathbf{b}(s, a). \quad (11.36)$$

iii. Generate $a' \in A_{s'}$ randomly using Algorithm 11.1 applied to β_0 in state s' .

iv. Set

$$q(s', a'; \beta_0) \leftarrow \beta_0^T \mathbf{b}(s', a'). \quad (11.37)$$

v. Set

$$\delta \leftarrow r + \lambda q(s', a'; \beta_0) - q(s, a; \beta_0) \quad (11.38)$$

vi. **Update β :**

$$\beta \leftarrow \beta + \tau_k \mathbf{b}(s, a) \delta. \quad (11.39)$$

vii. $s \leftarrow s'$, $a \leftarrow a'$.

viii. $k \leftarrow k + 1$ and $m \leftarrow m + 1$.

(d) **Weight updating:**

i. $\beta_0 \leftarrow \beta$.

ii. $n \leftarrow n + 1$.

3. **Terminate:** Return β_0 .

On the surface this does not look like a policy iteration algorithm since there is no clear improvement step. The algorithm combines improvement in steps 2(b) and 2(c)iii using fixed weights β_0 with evaluation in step 2(b)vi. While the action-generating policy being evaluated remains the same, the estimate of its corresponding β is revised.

On the other hand, the algorithm may be viewed as a simulation-based version of modified policy iteration¹⁷ (Algorithm 5.8) in which improvement and evaluation are

¹⁷Note that some authors, especially Bertsekas [2012], refer to modified policy iteration as *optimistic* policy iteration.

intertwined.

Some further comments follow:

1. In contrast to Q-learning which uses $q(s, a; \beta)$ as the basis for action selection, steps 2(b) and 2(c)iii choose actions using ϵ -greedy or softmax sampling based on $q(s, a; \beta_0)$ for M iterations. This is done to enhance exploration.
2. When $M = 1$, this algorithm is equivalent to a SARSA variant of Q-learning. When $M > 1$, the evaluation step may be viewed as a TD(0) approximation to the state-action value function of the ϵ -greedy or softmax-based policy corresponding to β_0 .
3. Steps 2(c)iii-2(c)v can be replaced by the two steps:

(a) Set

$$q(s', a; \beta_0) \leftarrow \beta_0^\top \mathbf{b}(s', a) \quad (11.40)$$

for all $a \in A_{s'}$.

(b) Set

$$\delta \leftarrow r + \lambda \max_{a' \in A_{s'}} q(s', a'; \beta_0) - q(s, a; \beta_0). \quad (11.41)$$

This alternative replaces updating the value of a specific action by the maximum value based on the estimate of the state-action value function using β_0 for M iterations. When $M = 1$, this updating approach is equivalent to Q-learning and for $M > 1$, it provides an alternative to the on-policy SARSA-like variant in the algorithmic statement.

4. No explicit stopping criterion is specified. One might terminate the algorithm when successive weight vectors differ by a small amount.
5. The algorithm is stated assuming trajectory-based state updating. Of course step 2(c)vii can be modified to update s using state-based or state-action pair-based sampling.
6. The index k is included to decrease the learning rate and exploration rates at *every* iteration. Each may be varied in different ways. For example in step 2(c)vii, k may be replaced by m and in implicit action choice in steps 2(b) and 2(c)iii the ϵ -greedy or softmax parameters can vary with n .
7. Similarly to Q-learning, there is no guarantee of convergence of this algorithm.
8. If one wished to start the algorithm with a decision rule d rather than a vector of weights β_0 , then step 2(b) would need to evaluate an ϵ -tweaked variant of this decision rule.

11.4.3 Newsvendor model

This section applies the following methods to the newsvendor inventory model formulated in Section 3.1.2:

1. Monte Carlo estimation
2. Q-learning
3. Q-policy iteration

The reason for considering this simple model is that by removing the effect of state transitions, subtle differences between Q-learning and Q-policy iteration become obvious.

Since this one-period model always starts in state $s = 0$, the state-action value function $q(a, 0)$ simplifies to the expected revenue from choosing action a , $q(a)$, which was defined in Section 3.1.2. Approximations of $q(a)$ use features corresponding to a cubic polynomial expressed in terms of scaled actions:

$$b_0(a) = 1, b_1(a) = \tilde{a}, b_2(a) = \tilde{a}^2 \text{ and } b_3(a) = \tilde{a}^3,$$

where \tilde{a} represents a scaled by subtracting the mean and dividing by the standard deviation¹⁸, so that

$$q(a; \boldsymbol{\beta}) = \boldsymbol{\beta}^\top \mathbf{b}(a).$$

Monte Carlo estimation

To implement Monte Carlo estimation, for each action a in a subset A_0 of all actions, sample the demand N times to obtain z^0, \dots, z^N and set $\hat{q}(a)$ equal to the mean of $r(a, z^0), \dots, r(a, z^N)$. Then use least squares to approximate $\hat{q}(a)$ by a linear function of its features to obtain $\hat{\boldsymbol{\beta}}_{MC}$. Note that the same demand sequence can be used for each chosen a .

Q-learning

The following algorithm applies Q-learning to the newsvendor model.

Algorithm 11.8. Q-learning for the newsvendor model

1. **Initialize:**
 - (a) Specify $\boldsymbol{\beta}$, $a \in A$, the number of iterations N , and set $n \leftarrow 1$.
 - (b) Specify sequences $\epsilon_n, n = 1, 2, \dots$ and $\tau_n, n = 1, 2, \dots$

¹⁸This means $\tilde{a} = (a - \text{mean}(a))/\text{SD}(a)$ where the mean and standard deviation are over $\{0, 1, \dots, a_{\max}\}$.

2. **Iterate:** While $n \leq N$:

- (a) Sample a' using ϵ -greedy (or softmax) sampling with respect to

$$q(a; \boldsymbol{\beta}) = \boldsymbol{\beta}^\top \mathbf{b}(a).$$

- (b) Generate demand z^n .

- (c) Set

$$\delta \leftarrow r(a', z^n) - \boldsymbol{\beta}^\top \mathbf{b}(a'). \quad (11.42)$$

- (d) **Update $\boldsymbol{\beta}$:**

$$\boldsymbol{\beta} \leftarrow \boldsymbol{\beta} + \tau_n \delta \mathbf{b}(a'). \quad (11.43)$$

- (e) $n \leftarrow n + 1$ and $a \leftarrow a'$.

3. **Terminate:** Return $\hat{\boldsymbol{\beta}}_{\text{QL}} = \boldsymbol{\beta}$.

Note that the calculation of δ in (11.42) sets $\max_{a' \in A_s} q(a', s') = 0$ because the episode ends after realizing the reward corresponding to demand z^n .

Q-policy iteration

The following algorithm applies Q-policy iteration to the newsvendor model.

Algorithm 11.9. Q-policy iteration for the newsvendor model.

1. **Initialize:**

- (a) Specify $\boldsymbol{\beta}_0$, the number of evaluation iterations M , the number of evaluation loops N , $k \leftarrow 1$ and $n \leftarrow 1$.
- (b) Specify sequences $\epsilon_n, n = 1, 2, \dots$ and $\tau_n, n = 1, 2, \dots$

2. **Iterate:** While $n \leq N$:

- (a) $m \leftarrow 1$ and $\boldsymbol{\beta} \leftarrow \boldsymbol{\beta}_0$.

- (b) While $m \leq M$:

- i. Sample a' using ϵ -greedy (or softmax) sampling with respect to

$$q(a; \boldsymbol{\beta}_0) = \boldsymbol{\beta}_0^\top \mathbf{b}(a).$$

- ii. Generate demand z^m .

- iii. Set

$$\delta \leftarrow r(a', z^m) - \boldsymbol{\beta}^\top \mathbf{b}(a') \quad (11.44)$$

iv. **Update β :**

$$\beta \leftarrow \beta + \tau_k \delta \mathbf{b}(a').$$

v. $k \leftarrow k + 1$, $m \leftarrow m + 1$, and $a \leftarrow a'$.

(c) $\beta_0 \leftarrow \beta$ and $n \leftarrow n + 1$.

3. **Terminate:** Return $\hat{\beta}_{\text{QPI}} = \beta_0$.

While these algorithms appear similar, there are subtle but important differences.

1. In each of these algorithms, the “max” in the Bellman update disappears because the Bellman equation reduces to $\max_{a \in A_0} E[r(a, Z)]$ so that the continuation cost is zero.
2. In Q-learning, action selection in step 2(a) is with respect to a different β at each iteration, which is updated in (11.43). In Q-policy iteration, a fixed β_0 is used for action generation in step 2(b)i for M evaluation steps and a different β (corresponding to the policy that is being evaluated) is updated in step 2(b)iv. This is because the inner loop seeks to evaluate a fixed policy based on the ϵ -greedy or softmax based decision rule corresponding to β_0 .
3. When $M = 1$, Q-learning and Q-policy iteration are equivalent.

A numerical example

This example considers the newsvendor problem with two discrete demand distributions:

- a rounded normal distribution, truncated at 0, with mean 50 and standard deviation 15, and
- a rounded gamma distribution with shape parameter 20 and scale parameter 4.

It sets the item cost to 20, the salvage value to 5 and the price to be each of 21, 30, and 120. The corresponding ratios of $G/(G + L)$ that were used to obtain the optimal order quantity, equaled 0.0625, 0.400, and 0.870, representing a range of quantiles of the demand distributions. The possible order quantities were $A_0 = \{0, 1, \dots, 120\}$.

To obtain a baseline, 10,000 Monte Carlo replicates were generated for each $a \in A_0$ and the expected reward corresponding to each action was set equal to the average reward over the samples for that action. Table 11.4 gives the optimal order quantity using the closed form representation (3.3) denoted “Optimal”, the maximum of the Monte Carlo estimates denoted “Monte Carlo”, and the maximum obtained when fitting the Monte Carlo estimates with a cubic polynomial, denoted “MC-Cubic”, and a cubic spline with knots at 40 and 80, denoted “MC-Spline”.

Distribution	Price	Optimal	Monte Carlo	MC-Cubic	MC-Spline
Normal	21	27	26	23	29
	30	46	45	41	44
	120	67	65	76	69
Gamma	21	55	53	47	54
	30	74	76	76	75
	120	100	99	101	99

Table 11.4: Order quantities for the newsvendor model based on Monte Carlo estimates.

Observe that the order quantity that maximizes the Monte Carlo estimates accurately approximates the optimal order quantity. Observe also that the order quantity obtained by maximizing the spline approximation better approximates the Monte Carlo maximum than that derived from the cubic approximation. This is to be expected because the spline is a more flexible function with more parameters than a cubic polynomial.

Q-learning and Q-policy iteration were applied to these instances using a cubic approximation based on scaled values of a . Both algorithms were initiated with $\beta = \mathbf{0}$, used the STC learning rate $\tau_n = 0.05(1 + 10^{-5}n^2)^{-1}$ and ϵ -greedy exploration with parameter $\epsilon_n = 4,000/(20,000 + n)$. Calculations used $N = 20,000$ for Q-learning and $N = 10$ and $M = 2,000$ for Q-policy iteration, so that the total number of iterates in each case was the same. For each combination of demand distribution and price, 40 replicates of each algorithm were applied with common random number seeds, or equivalently, the same sequence of demands.

The boxplots in Figure 11.9 shows the variability of the estimates of the optimal order quantity chosen according to

$$a^* \in \arg \max_{a \in A_0} \hat{\beta}^T \mathbf{b}(a)$$

over the 40 replicates.

Comparison with Table 11.4 shows that Q-policy iteration better approximated the order quantity obtained using MC-Cubic than the Q-learning estimates in all cases. Moreover, with normal demand:

1. When price equals 21, the Q-learning estimate often resulted in a non-increasing approximating polynomial with $a^* = 0$.
2. The Q-policy iteration estimates were considerably less variable than those using Q-learning.

These observations suggest that Q-policy iteration offers a promising alternative to Q-learning.

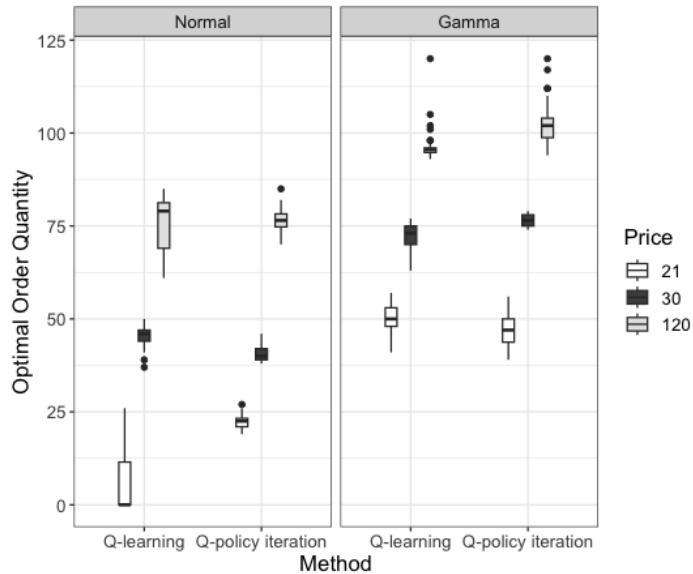


Figure 11.9: Boxplots comparing the actions chosen by Q-learning and Q-policy iteration for the newsvendor model over 40 replicates broken down by demand distribution and price.

11.4.4 Queuing service rate control model

The example in this section applies Algorithm 11.7 to the queuing control model and compares results to those obtained applying Q-learning directly. Again it assumes $q(s, a)$ is approximated by distinct cubic functions of the state for each action.

To obtain reasonable policy estimates required considerable experimentation with starting values, the total number of iterations, the learning rate and the ϵ -greedy parameters. Table 11.5 provides summaries of three performance metrics over 40 replicates of several variants of Q-policy iteration with

$$\tau_n = 5,000 / (50,000 + n)$$

and

$$\epsilon_n = 100 / (400 + n),$$

where the index for the parameters was the cumulative iteration number. State updating was based on random state-based sampling and performance measures were derived from a total of 250,000 iterations per replicate apportioned between M and N .

The resulting policies were assessed on the basis of the number of non-optimal actions, the RMSE between the true value of the greedy policy based on the final q -function estimate and the optimal value, and the wRMSE weighted by the stationary distribution of the greedy policy.

The results in Table 11.5 suggest:

Method	Quantity	min	25th %ile	median	75th %ile	max
(M = 100, N = 2,500)	Non-optimal actions	5	9	10	11	20
	RMSE	3.5	12.9	19.9	25.3	139.0
	wRMSE	33.5	105.3	157.9	208.2	585.9
(M = 50, N = 5,000)	Non-optimal actions	3	6	7.5	10	21
	RMSE	1.7	12.2	16.8	22.5	169.9
	wRMSE	12.6	141.7	150.6	212.7	587.0
SARSA	Non-optimal actions	5	9.8	11.5	13	40
	RMSE	3.0	19.2	26.4	30.8	334.4
	wRMSE	1.4	141.8	194.3	585.9	651.9
Q-policy iteration variant (M = 50, N = 5,000)	Non-optimal actions	3	6	7	9	18
	RMSE	1.7	8.3	12.6	22.1	126.7
	wRMSE	12.6	98.9	141.7	229.9	585.9
Q-learning	Non-optimal actions	6	8.8	12	13	17
	RMSE	3.9	13.3	21.0	23.8	95.7
	wRMSE	9.9	62.65	141.8	190.3	440.3

Table 11.5: Summary statistics based on 40 replicates of Q-policy iteration, Q-learning and SARSA applied to the queuing service rate control model. The Q-policy iteration variant uses (11.40) and (11.41) for parameter updating. Fractional values for percentiles of non-optimal actions result from interpolation.

1. No instance identified the optimal policy. Q-policy iteration and its variant with $M = 50$ and $N = 5,000$ generated policies that were closest to optimum. However, in a few replicates¹⁹ policies were non-monotone or only chose two actions.
2. Both instances of Q-policy iteration as stated in Algorithm 11.7 were preferable to SARSA with respect to all performance metrics. The combination $M = 50, N = 5,000$ gave slightly better results than using $M = 100, N = 2,000$.
3. Policies selected by the Q-policy iteration variant were closer than Q-learning to the optimum policy on the basis of number of non-optimal actions chosen. However, this did not translate to consistently improved performance with respect to RMSE and wRMSE.
4. The Q-policy iteration variant was preferable to choosing the implementation in Algorithm 11.7.

The main takeaways from this example are that Q-policy iteration based on the variant using (11.40) and (11.41) was preferable to directly applying Algorithm 11.7 and that there is weak evidence that the Q-policy iteration variant was preferable to Q-learning but the difference in performance was small.

¹⁹Results not shown.

11.4.5 Concluding remarks on examples

These numerical studies show that for the queuing control model, both Q-learning and Q-policy iteration produced similar results. However, Q-policy iteration was slightly better at identifying a policy in agreement with the optimal policy. In the newsvendor model, Q-policy iteration was better than Q-learning in identifying a policy that agreed with that found using a cubic approximation to Monte Carlo estimates.

Overall, both Q-learning and Q-policy iteration provide the analyst with tools for analyzing models that use function approximation. Their relative performance will depend on many factors including the choice of parameter settings and the specific problem being solved.

11.5 Policy space methods

This lengthy section describes a conceptually different approach to function approximation based on parameterizing the probability that a randomized stationary policy selects an action in a given state. Instead of directly improving a value function or state-action value function, it seeks to find a choice of parameter values that improves the value of a policy directly.

Its advantage is that unlike state-action value function focused methods (such as Q-learning), which may jump between deterministic policies, it allows for gradual changes in action-selection probabilities leading to greater stability in values. This is especially attractive when controlling physical systems in real-time.

Two concepts underlie this approach:

1. *Expressing the probability that a stationary randomized policy d^∞ chooses action a in state s as a parametric function of both the state and action.* This is somewhat similar to the approach used to parameterize a state-action value function in which both s and a are inputs and $q(s, a)$ is the output. In contrast, policy space methods are based on functions of s and a , however, the state s is an input and the probability of choosing action a , $w_d(a|s)$, is the output.
2. *Updating parameter values using (stochastic) gradient ascent.* This approach seeks to iteratively find a zero of the gradient of a policy value function²⁰ by stochastic approximation in which gradient estimates are obtained by sampling from a distribution with a specified expectation.

²⁰Of course, the value function varies over states. To obtain a single value, a weighted average over states is used. Surprisingly, this choice of weights will not affect the policy updating.

11.5.1 Motivation: A policy gradient algorithm for a one-state one-period model

Consider first a one-period, single-state multi-action model such as the newsvendor problem. Since this model does not include state transitions, it reduces to a classical univariate stochastic optimization model. The same notation for approximation is used as when approximating state-action value functions although as noted above, these are two conceptually different approaches. That is, features are represented by vectors $\mathbf{b}(s, a)$ and parameters by compatible vectors²¹ $\boldsymbol{\beta}$. In the single-state model, the state, say 0, is fixed so that the vector of features is represented by $\mathbf{b}(a) := \mathbf{b}(0, a)$.

Frequently, policies are expressed in terms of the softmax or logistic function. For single-state models:

$$w(a|\boldsymbol{\beta}) = \frac{e^{\boldsymbol{\beta}^\top \mathbf{b}(a)}}{\sum_{a' \in A_s} e^{\boldsymbol{\beta}^\top \mathbf{b}(a')}} \quad (11.45)$$

where $\boldsymbol{\beta}$ is a column vector of parameters and $\mathbf{b}(a)$ is a column vector of features associated with action a . In multi-state models (Markov decision processes),

$$w(a|s; \boldsymbol{\beta}) = \frac{e^{\boldsymbol{\beta}^\top \mathbf{b}(s, a)}}{\sum_{a' \in A_s} e^{\boldsymbol{\beta}^\top \mathbf{b}(s, a')}}. \quad (11.46)$$

Note that the temperature parameter of the softmax function is absorbed into the weight vector $\boldsymbol{\beta}$. However in some settings, such as the optimal stopping model below, it might be convenient to retain it to stabilize estimates.

The goal when analyzing this single-state model is to find a $\mathbf{b}(a)$ that attains the maximum in

$$v^* := \max_{\boldsymbol{\beta}} v(\boldsymbol{\beta}) = \max_{\boldsymbol{\beta}} \sum_{a \in A_0} w(a|\boldsymbol{\beta})r(a) = \max_{\boldsymbol{\beta}} \sum_{a \in A_0} \frac{e^{\boldsymbol{\beta}^\top \mathbf{b}(a)}}{\sum_{a' \in A_s} e^{\boldsymbol{\beta}^\top \mathbf{b}(a')}} r(a). \quad (11.47)$$

In this expression, $r(a)$ denotes the (expected) reward for choosing action a , $w(a|\boldsymbol{\beta}) := w(a|0; \boldsymbol{\beta})$ and

$$v(\boldsymbol{\beta}) := \sum_{a \in A_0} w(a|\boldsymbol{\beta})r(a) = E_{\boldsymbol{\beta}}[r(Y)],$$

where Y is the random action selected by $w(\cdot|\boldsymbol{\beta})$.

Policy gradient methods seek to maximize this expression by solving:

$$\nabla_{\boldsymbol{\beta}} v(\boldsymbol{\beta}) = \mathbf{0}. \quad (11.48)$$

In the newsvendor model, $r(a) = E[r(a, \delta)] = \sum_{z=0}^M r(a, z)f(z)$, where $f(z) = P[Z = z]$. (Recall that Z is the random demand with support $\{0, 1, \dots, M\}$.) In this simple model, $v(\boldsymbol{\beta})$ can be computed analytically, numerically or through simulation. To simulate this model requires sampling from both $w(\cdot|\boldsymbol{\beta})$ and the demand distribution $f(\cdot)$.

²¹Many authors represent randomized decision rules by $\pi_{\boldsymbol{\phi}}(a|s)$ where $\boldsymbol{\phi}$ denotes a vector of parameters.

The gradient of $v(\beta)$ in the newsvendor model

The machinery to apply stochastic gradient descent is developed next. To do so requires deriving a representation for the gradient of $v(\beta)$. This derivation is based on a simple calculus identity that is the key step in generalizing this result to multi-state models.

The gradient of $v(\beta)$ can be written as:

$$\nabla_\beta v(\beta) = \sum_{a \in A_0} \nabla_\beta w(a|\beta) r(a) \quad (11.49)$$

$$= \sum_{a \in A_0} w(a|\beta) \nabla_\beta \ln(w(a|\beta)) r(a) \quad (11.50)$$

$$= E_\beta [\nabla_\beta \ln(w(Y|\beta)) r(Y)]. \quad (11.50)$$

The expression in (11.49) follows by applying the calculus identity

$$\frac{dg(x)}{dx} = g(x) \frac{d \ln(g(x))}{dx} \quad (11.51)$$

to each component of β .

The benefits of establishing this equivalence are that:

1. When using the softmax to represent $w(a|\beta)$, the gradient of $\ln(w(a|\beta))$ has a nice closed-form representation.
2. As a result of the representation (11.50), the gradient of $v(\beta)$ can be estimated by sampling

$$\nabla_\beta \ln(w(Y|\beta)) r(Y)$$

from $w(\cdot|\beta)$.

3. In multi-period models, expectations are based on products of probabilities, in which case logarithms transform the product to a sum making it easier to analyze²².

It is left as an exercise to show that when $w(a|\beta)$ is defined by (11.45),

$$\nabla_\beta \ln(w(a|\beta)) = \mathbf{b}(a) - \sum_{a' \in A_0} w(a'|\beta) \mathbf{b}(a'). \quad (11.52)$$

Combining (11.49) with (11.52) leads to the following closed-form expression for the gradient of $v(\beta)$ in the newsvendor model:

²²Note that in statistical maximum likelihood estimation one bases results on maximizing log-likelihoods instead of likelihoods.

$$\nabla_{\beta} v(\beta) = \sum_{a \in A_0} w(a|\beta) \left(\mathbf{b}(a) - \sum_{a' \in A_0} w(a'|\beta) \mathbf{b}(a') \right) r(a). \quad (11.53)$$

Online estimation of the gradient in the newsvendor model

This section applies stochastic gradient ascent²³ to solve $\nabla_{\beta} v(\beta) = \mathbf{0}$ for the newsvendor model. This method involves three steps:

1. Sample action a from $w(\cdot|\beta)$,
2. sample the demand from $f(\cdot)$ to estimate $r(a) = E[r(a, Z)]$, and
3. estimate the gradient by $\nabla_{\beta} \ln(w(a|\beta))r(a)$.

Thus, the gradient ascent recursion becomes

$$\beta' \leftarrow \beta + \tau \nabla_{\beta} \ln(w(a|\beta))r(a), \quad (11.54)$$

where the gradient is computed using (11.53) (or by numerical differentiation). This observation leads to the following policy gradient algorithm for the newsvendor model (assuming the simulation of both actions and rewards).

Algorithm 11.10. Policy gradient for the newsvendor model

1. **Initialize:**
 - (a) Specify β and a learning rate sequence $\tau_n, n = 1, 2, \dots$
 - (b) Specify the number of iterations N and set $n \leftarrow 1$.
2. **Iterate:** While $n \leq N$:
 - (a) Sample a from $w(\cdot|\beta)$.
 - (b) Generate demand z^n and set $r \leftarrow r(a, z^n)$.
 - (c) **Estimate the gradient:**

$$\nabla_{\beta} v(\beta) = \left(\mathbf{b}(a) - \sum_{a' \in A_0} w(a'|\beta) \mathbf{b}(a') \right) r. \quad (11.55)$$

- (d) **Update β :**

$$\beta \leftarrow \beta + \tau_n \nabla_{\beta} v(\beta). \quad (11.56)$$

²³“Ascent” is used since this is a maximization problem.

(e) $n \leftarrow n + 1$.

3. **Terminate:** Return β .

Some comments follow:

1. As stated, the algorithm samples both the action and the reward given the action. In some examples the reward may be a deterministic function of the action so it can be computed directly once the action is known.
2. In a model-free environment, instead of sampling the reward, it can be observed. However, one would still sample the action from $w(\cdot|\beta)$.
3. The policy gradient algorithm converges to a global maximum under mild conditions on $r(a)$.
4. The algorithm terminates with an estimate of β and the corresponding stationary randomized policy $w(a|\beta)$.
5. Alternatively to stopping after a specified number of iterations, the algorithm can terminate when the change in β achieves a pre-specified tolerance.

How the algorithm works

To illustrate the workings of this algorithm take as features indicator variables of actions as follows:

$$b_i(a) = \begin{cases} 1 & a = a_i \\ 0 & a \neq a_i \end{cases}$$

for $i = \{0, 1, \dots, a_{\max}\}$. In this case, there are $a_{\max} + 1$ features. Note this choice of features is equivalent to using a tabular representation. The advantage of this choice of features is that it provides very clear insight into the workings of gradient ascent as the following calculations show.

For concreteness, assume that $a \in \{0, 1, 2, 3\}$ and $\beta = (0, 0, 0, 0)$, so that defining $w(a|\beta)$ by (11.45) gives $w(a|\beta) = 0.25$ for $a \in \{0, 1, 2, 3\}$. Suppose sampling generates action $a = 1$ and $r(a) = 17$. Then applying (11.55) shows that

$$\nabla_\beta v(\beta) = 17 \left(\begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} - \begin{bmatrix} 0.25 \\ 0.25 \\ 0.25 \\ 0.25 \end{bmatrix} \right) = 17 \begin{bmatrix} -0.25 \\ 0.75 \\ -0.25 \\ -0.25 \end{bmatrix}.$$

Observe that the component of the gradient corresponding to $a = 1$ is positive and all other components are negative. Thus, as a result of applying gradient ascent in step 2(d) of Algorithm 11.10, the component of β corresponding to $a = 1$ will increase and

all other components will decrease. Hence the policy $w(a|\beta')$, where β' corresponds to the updated parameter vector, will select $a = 1$ with greater probability and all other actions with smaller probability than $w(a|\beta)$. If on the other hand $r(a) < 0$, the reverse would occur, namely the probability of selecting $a = 1$ would decrease and all other probabilities would increase.

Example 11.5. Policy gradient applied to the newsvendor model.

This example sets $a_{\max} = 25$, the unit cost to 20 and the salvage value to 5. The possible prices are 21, 30, and 120. Three demand distributions are considered:

1. A truncated and rounded normal with mean 10 and standard deviation 3,
2. a binomial distribution with $n = 10$ and $p = 0.6$, and
3. a discrete uniform distribution on $0, \dots, a_{\max}$.

Algorithm 11.10 is applied with $N = 10,000$ and $\tau_n = 0.0001$ for each of 40 replicates across all nine combinations of price and demand distribution.

In all replicates, the distribution $w(a|\beta)$ evaluated at the estimated β was unimodal in a , with a single action having probability extremely close to 1 and all others near 0, corresponding to a deterministic policy.

Table 11.6 summarizes results. The column “PG: Mean order quantity” reports the average of $\arg \max_{a \in A_0} w(a|\beta)$, while the “PG: Standard deviation” column gives its standard deviation across the 40 replicates. In all instances, the mean order quantity closely approximates the “Optimal order quantity” calculated using (3.3). The greatest variability was observed when the price equaled 120, corresponding to the 87th percentile of the demand distribution.

11.5.2 A policy gradient algorithm for a Markov decision process

Intuition and results in the previous section provide the basis of a policy gradient algorithm for an episodic Markov decision process. Recall that the goal in an episodic model is to maximize the expected total reward prior to reaching a set of zero-reward absorbing states Δ . In such models one seeks to find a policy π that maximizes

$$v^\pi(s) = E^\pi \left[\sum_{n=1}^{N_\Delta} r(X_n, Y_n, X_{n+1}) \middle| X_1 = s \right]$$

over the set of all history-dependent randomized policies, where N_Δ denotes the random time the system enters Δ . Recall (see Section 2.2.3) that the expectation is with respect

Demand distribution	Price	Optimal order quantity	PG: Mean order quantity	PG: Standard deviation
Normal	21	5.40	5.55	0.64
	30	9.23	9.50	0.72
	120	13.37	14.43	2.05
Binomial	21	4	3.73	0.45
	30	6	5.58	0.50
	120	8	7.93	1.16
Uniform	21	1.63	1.15	0.36
	30	10.40	10.75	1.69
	120	22.61	21.93	2.10

Table 11.6: Mean and standard deviation over 40 replicates of the order quantity obtained by maximizing $w(a|\hat{\beta})$ over a where $\hat{\beta}$ is obtained using Algorithm 11.10. The optimal order quantity is based on the closed form expression (3.3) using continuous representations of the normal and uniform distributions and “PG” stands for policy gradient.

to the probability distribution of the sequence of states and actions generated by the stochastic process corresponding to π . The absorption time (effective horizon) N_Δ is implicitly determined by the trajectories. This probability distribution combines both Markov decision process transition probabilities and action choice probabilities corresponding to randomized decision rules.

To compute a gradient requires a single objective function as opposed to one for each $s \in S$. This is easily accomplished by adding an initial state distribution $\rho(s)$ so that the value of policy π becomes scalar-valued and represented by

$$v^\pi = \sum_{s \in S} \rho(s)v^\pi(s). \quad (11.57)$$

As a result of Theorem 6.3, under mild conditions there is a stationary deterministic policy that maximizes (11.57). But instead of focusing on deterministic policies, the methods herein operate within the larger family of randomized stationary policies $d^\infty \in D^{\text{MR}}$ which choose actions a in state $s \in S \setminus \Delta$ according to distribution $w_d(a|s)$.

As above, let $\mathbf{b}(s, a)$ denote a state and action-dependent feature vector with corresponding β and let $w(a|s; \beta)$ denote the corresponding parameterized action-choice probability distribution. Let d_β represent the decision rule that chooses actions according $w(a|s; \beta)$. That is $w_{d_\beta}(a|s) := w(a|s; \beta)$.

Hence, the (scalar) objective becomes that of finding a weight vector β that maximizes

$$v(\beta) := \sum_{s \in S} \rho(s)v^{(d_\beta)^\infty}(s) \quad (11.58)$$

or equivalently, a

$$\hat{\beta} \in \arg \max_{\beta} v(\beta). \quad (11.59)$$

11.5.3 The gradient of the policy value function for an undiscounted infinite horizon Markov decision process

Chapter 4 provides details regarding the evaluation of an expression similar to $v(\beta)$ in terms of transition probabilities, action choice probabilities and rewards. It shows that

$$\begin{aligned} v(\beta) &= \sum_{s^1 \in S} \sum_{a^1 \in A_{s^1}} \sum_{s^2 \in S} \rho(s^1) w(a^1|s^1; \beta) p(s^2|s^1, a^1) \left(r(s^1, a^1, s^2) \right. \\ &\quad \left. + \sum_{a^2 \in A_{s^2}} \sum_{s^3 \in S} w(a^2|s^2; \beta) p(s^3|s^2, a^2) (r(s^2, a^2, s^3) + \dots) \right) \end{aligned} \quad (11.60)$$

where eventually the rewards are zero after reaching an absorbing state in Δ .

Now consider the expected reward in the first period only

$$v_1(\beta) := \sum_{s^1 \in S} \sum_{a^1 \in A_{s^1}} \sum_{s^2 \in S} \rho(s^1) w(a^1|s^1; \beta) p(s^2|s^1, a^1) r(s^1, a^1, s^2).$$

In this case,

$$\nabla_\beta v_1(\beta) = \sum_{s^1 \in S} \sum_{a^1 \in A_{s^1}} \sum_{s^2 \in S} \nabla_\beta (\rho(s^1) w(a^1|s^1; \beta) p(s^2|s^1, a^1) r(s^1, a^1, s^2)). \quad (11.61)$$

Observe that this gradient involves both the initial state distribution and the transition probabilities and is not amenable (especially in later periods) to direct computation or simulation. Instead using (11.51) yields

$$\begin{aligned} \nabla_\beta v_1(\beta) &= \sum_{s^1 \in S} \sum_{a^1 \in A_{s^1}} \sum_{s^2 \in S} \rho(s^1) w(a^1|s^1; \beta) p(s^2|s^1, a^1) r(s^1, a^1, s^2) \\ &\quad \times \nabla_\beta \ln (\rho(s^1) w(a^1|s^1; \beta) p(s^2|s^1, a^1) r(s^1, a^1, s^2)). \end{aligned} \quad (11.62)$$

Since $\rho(s)$, $p(s^2|s^1, a^1)$ and $r(s^1, a^1, s^2)$ do not involve β ,

$$\begin{aligned} \nabla_\beta \ln (\rho(s^1) w(a^1|s^1; \beta) p(s^2|s^1, a^1) r(s^1, a^1, s^2)) \\ &= \nabla_\beta \ln (\rho(s^1)) + \nabla_\beta \ln (w(a^1|s^1; \beta)) + \nabla_\beta \ln (p(s^2|s^1, a^1)) + \nabla_\beta \ln (r(s^1, a^1, s^2)) \\ &= \nabla_\beta \ln (w(a^1|s^1; \beta)). \end{aligned}$$

Therefore, it follows from (11.62) that

$$\nabla_\beta v_1(\beta) = \sum_{s^1 \in S} \sum_{a^1 \in A_{s^1}} \sum_{s^2 \in S} \rho(s^1) w(a^1|s^1; \beta) p(s^2|s^1, a^1) r(s^1, a^1, s^2) \nabla_\beta \ln (w(a^1|s^1; \beta)). \quad (11.63)$$

The significance of this expression is that it shows that $\nabla_{\beta}v_1(\beta)$ can be evaluated by averaging $\nabla_{\beta} \ln(w(a^1|s^1; \beta))r(s^1, a^1, s^2)$ over replicates of (s^1, a^1, s^2) generated by the distribution of (X_1, Y_1, X_2) . Note that a model is not necessary to evaluate this expression; realizations of the reward can replace $r(s^1, a^1, s^2)$.

The following result, which is proved in the appendix to this chapter, generalizes the above argument to complete trajectories. It provides the basis for estimating the gradient of the policy value function using a sampled trajectory.

Theorem 11.1. Suppose N_{Δ} is finite with probability 1 and that $w(a|s; \beta)$ is differentiable with respect to each component of β . Then

$$\nabla_{\beta}v(\beta) = E^{(d_{\beta})^{\infty}} \left[\sum_{j=1}^{N_{\Delta}} \nabla_{\beta} \ln(w(Y_j|X_j; \beta)) \left(\sum_{i=j}^{N_{\Delta}} r(X_i, Y_i, X_{i+1}) \right) \right]. \quad (11.64)$$

Observe that the multiplier of the gradient $\nabla_{\beta} \ln(w(Y_j|X_j; \beta))$ only involves rewards after decision epoch j . Moreover, subtracting a function $B(\cdot)$ that is independent of the action from the cumulative reward, referred to as a *baseline*, does not affect the gradient but often stabilizes calculations.

The quantity $\nabla_{\beta} \ln(w(Y_j|X_j; \beta))$ is often referred to as the *score function* and is a fundamental quantity in maximum likelihood estimation. In fact, if the summation of the rewards was replaced by a constant equal to one, the above expression would correspond to the gradient used when applying gradient ascent to obtain the maximum likelihood estimator of β for decision rule d_{β} .

11.5.4 A simple policy gradient algorithm

The following algorithm, frequently referred to as *the policy gradient algorithm*, describes an early²⁴ and straightforward implementation of gradient ascent over the space of parameterized randomized policies.

Algorithm 11.11. Policy gradient for an episodic model

1. **Initialize:**
 - (a) Specify β , a learning rate sequence $\tau_n, n = 1, 2, \dots$, and a fixed baseline $B(s)$ for all $s \in S$.
 - (b) Specify the number of episodes K and set $k \leftarrow 1$.
2. **Iterate:** While $k \leq K$:

²⁴This is sometimes referred to as *REINFORCE* in the reinforcement learning literature. It represents one of the first implementations of this approach.

- (a) **Generate an episode:** Generate an episode using $w(a|s; \beta^A)$ to generate actions. Save the sequence of states, actions and rewards

$$(s^1, a^1, s^2, r^1, \dots, s^N, a^N, s^{N+1}, r^N),$$

where N denotes the termination epoch of the episode.

- (b) $n \leftarrow 1$.
(c) While $n \leq N$:

- i. **Update weights sequentially:**

$$\beta \leftarrow \beta + \tau_k \nabla_{\beta} \ln(w(a^n|s^n; \beta)) \left(\sum_{i=n}^N r^i - B(s^n) \right). \quad (11.65)$$

- ii. $n \leftarrow n + 1$.

- (d) $k \leftarrow k + 1$.

3. **Terminate:** Return β .

This algorithm may be viewed as a simulated policy improvement algorithm. Starting with a policy defined by β , it evaluates it by generating a single episode using this policy. Then, it improves the policy by adjusting the weights in the direction of the gradient of this policy in step 2(c).

The update can also be implemented in a single step by accumulating the terms of the gradient as in (11.64) prior to an update or even averaging the gradient over several trajectories. This variant is often referred to as a *batch policy gradient* algorithm and is the foundation for many effective algorithms. The highly successful *proximal policy optimization algorithm (PPO)* is a batch implementation of this algorithm that restricts how much policies can change at each policy update²⁵.

To implement this batch policy gradient step, replace the sequential update 2(c) by

$$\beta \leftarrow \beta + \tau_k \sum_{n=1}^N \nabla_{\beta} \ln(w(a^n|s^n; \beta)) \left(\sum_{i=n}^N r^i - B(s^n) \right). \quad (11.66)$$

Some further comments follow:

- After generating an entire episode using $w(a|s; \beta)$, step 2(c) updates β by stepping backwards through the tail of cumulative rewards (adjusted by the baseline).
- The above algorithm is on-policy. That is, it improves the same policy that generates the sequence of states, actions and rewards. An off-policy variant

²⁵See Schulman et al. [2017].

would improve a policy using episodes generated by a different policy. *Importance sampling* methods adjust iterates appropriately.

3. Many implementations of this algorithm appear without a baseline, which is equivalent to setting $B(s) = 0$. Including $B(s)$ does not invalidate the gradient derivation in Theorem 11.1 since the baseline does not change the expectation in (11.64). Moreover choosing the baseline appropriately enhances convergence by reducing the variance of the estimates. Actor-critic methods below are based on choosing an appropriate baseline.
4. Expression (11.65) provides insight into the qualitative behavior of the policy gradient algorithm. When $\sum_{i=n}^N r^i - B(s^n)$ is positive, components of β that make action a^n more likely in state s^n will be increased.

To illustrate this phenomenon, consider an implementation in which features are indicator functions of state-action pairs and $w(a|s; \beta)$ is the softmax function. Then

$$\frac{\partial}{\partial \beta_{s,a}} \ln(w(a^n|s^n; \beta)) = \begin{cases} 1 - w(a|s; \beta) & \text{for } s = s^n, a = a^n \\ -w(a|s; \beta) & \text{for } s = s^n, a \neq a^n \\ 0 & \text{for } s \neq s^n. \end{cases}$$

Consequently, the only positive component of $\nabla_\beta \ln(w(a|s^n; \beta))$ corresponds to $a = a^n$. Hence, when $\sum_{i=n}^N r^i - B(s^n)$ is positive, applying (11.65) increases β_{s^n, a^n} and decreases $\beta_{s^n, a}$ for $a \neq a^n$. Moreover, for $s \neq s^n$, $\beta_{s,a}$ remains unchanged. Thus, the probability of choosing action a^n in state s^n increases for future episodes.

5. The expression $\nabla_\beta \ln(w(a^n|s^n; \beta))$ can be evaluated numerically, in closed form when $w(a|s; \beta)$ is a softmax function of features, or by back-propagation when $w(a|s; \beta)$ is a neural net.
6. The index for the learning rate can vary according to the replicate (as stated), or the total number of passes through (11.65).
7. Step 2(c) describes a direct application of gradient ascent. Enhanced versions are available²⁶.
8. The main cost in implementing this algorithm is generating trajectories in step 2(a). In complicated episodic models, N can be very large. *Experience replay* described below strategically stores and replaces past trajectories to reduce this cost. It is especially relevant in batch implementations.

²⁶For example, the widely cited smoothed gradient descent algorithm Adam [Kingma and Ba, 2017].

11.5.5 An example

Consider the problem of optimal stopping in a reflecting random walk (Example 6.19). Let $S = \{-M, -(M-1), \dots, M-1, M\}$ and $A_s = \{C, Q\}$ with C corresponding to continuing and Q corresponding to stopping (Q for quit). Stopping in state s yields a reward of $g(s)$, while continuing costs c and results in a transition to state s' according to

$$p(s'|s, a_1) = \begin{cases} p & \text{if } s' = s + 1, s < N \text{ or } s' = s = M \\ 1 - p & \text{if } s' = s - 1, s > 1 \text{ or } s' = s = -M \\ 0 & \text{otherwise} \end{cases}$$

for $0 < p < 1$. Let Δ denote the stopped state.

This is a stochastic shortest²⁷ path model with the reward of the improper policy that continues in every state equal to $-\infty$. Under all other policies the system terminates in finite expected time so that it can be analyzed as an episodic model.

Illustrative calculations

Before providing computational results, the update in step 2(c) of Algorithm 11.11 is illustrated using a single hypothetical episode. Suppose a policy based on β generates an episode with two continuation actions followed by stopping at the third step. For concreteness, set the observed sequence to $5 \rightarrow 6 \rightarrow 7 \rightarrow \Delta$ and suppose $c = -2$ and $g(s) = s^2$. Then the trajectory of the episode would be

$$(s^1, a^1, r^1, s^2, a^2, r^2, s^3, a^3, r^3, s^4) = (5, C, -2, 6, C, -2, 7, Q, 49, \Delta).$$

The following steps illustrate the calculation in (11.65) with $B = 0$. Ignoring “artificial” transitions once Δ is reached, $\sigma^n := \sum_{i=n}^N r^i - B$ takes on the values

$$\sigma^3 = 49, \quad \sigma^2 = 47, \quad \sigma^1 = 45.$$

Now assume the features are linear in the state for each action so that

$$b(s) = \beta_{0,0} I_{\{C\}}(a) + \beta_{0,1} I_{\{Q\}}(a) + \beta_{1,0} s I_{\{C\}}(a) + \beta_{1,1} s I_{\{Q\}}(a)$$

and the softmax function $w(a|s; \beta)$ with $\eta = 1$ becomes

$$w(C|s; \beta) = \frac{e^{\beta_{0,0} + \beta_{1,0}s}}{e^{\beta_{0,0} + \beta_{1,0}s} + e^{\beta_{0,1} + \beta_{1,1}s}}$$

and $w(Q|s; \beta) = 1 - w(C|s; \beta)$. Some simple calculus establishes that the gradients are given by

$$\nabla_\beta \ln(w(C|s; \beta)) = \begin{bmatrix} w(Q|s; \beta) \\ -w(Q|s; \beta) \\ sw(Q|s; \beta) \\ -sw(Q|s; \beta) \end{bmatrix} = w(Q|s; \beta) \begin{bmatrix} 1 \\ -1 \\ s \\ -s \end{bmatrix}$$

²⁷Since rewards are maximized, this is actually a *longest* path model.

and

$$\nabla_{\beta} \ln(w(Q|s; \beta)) = w(C|s; \beta) \begin{bmatrix} -1 \\ 1 \\ -s \\ s \end{bmatrix}$$

where $\beta = (\beta_{0,0}, \beta_{0,1}, \beta_{1,0}, \beta_{1,1})$. Note that in examples such as this it is more convenient to represent the components of β in matrix form with columns corresponding to actions so that gradient can be represented in terms of a Jacobian matrix.

Thus, when $a = C$, (11.65) becomes

$$\beta \leftarrow \beta + \tau_n \left(\sum_{i=n}^N r^i \right) w(Q|s; \beta) \begin{bmatrix} 1 \\ -1 \\ s \\ -s \end{bmatrix}. \quad (11.67)$$

Observe that the only stochastic element in this expression is the term $\sum_{i=n}^N r^i$ so that the variability of this quantity affects the stability of the estimates of β . Standardizing rewards can reduce this variability.

Suppose that $\beta = (0, 0, 0, 0)$ and $\tau_n = 0.1$, then the iterates (numbered in the order they are computed) become

$$\beta^1 = 0.1 \cdot 49 \cdot 0.5 \begin{bmatrix} 1 \\ -1 \\ 7 \\ -7 \end{bmatrix} = \begin{bmatrix} 2.45 \\ -2.45 \\ 17.15 \\ -17.15 \end{bmatrix}, \quad \beta^2 = \begin{bmatrix} 2.25 \\ -2.25 \\ 11.05 \\ -11.05 \end{bmatrix} \text{ and } \beta^3 = \begin{bmatrix} 2.25 \\ -2.25 \\ 11.05 \\ -11.05 \end{bmatrix}.$$

Note that corresponding to the final iterate β^3 , the action choice probabilities in state 5 are $w(Q|5; \beta^3) = 1$ and $w(C|5; \beta^3) = 0$ and in state 0, $w(Q|0; \beta^3) = 0.99$ and $w(C|0; \beta^3) = 0.01$.

Alternatively, setting $B = 47$, which equals the mean of the σ^n , results in the sequence

$$\beta^1 = \begin{bmatrix} -0.1 \\ 0.1 \\ -0.7 \\ 0.7 \end{bmatrix}, \quad \beta^2 = \begin{bmatrix} -0.1 \\ 0.1 \\ -0.7 \\ 0.7 \end{bmatrix} \text{ and } \beta^3 = \begin{bmatrix} -0.3 \\ -0.3 \\ -1.7 \\ 1.7 \end{bmatrix}.$$

Although the β estimates have changed and have become smaller in magnitude, the corresponding-action selection probabilities in state 5 remain $w(Q|5; \beta^3) = 1$ and $w(C|5; \beta^3) = 0$, and in state 0, $w(Q|0; \beta^3) = 0.65$ and $w(C|0; \beta^3) = 0.35$. Thus, estimates of β and the corresponding probabilities are sensitive to the choice of baseline.

As an alternative, consider implementing the policy gradient update in a single (batch) step. To do this compute the gradient of $v(\beta)$ using (11.66) so that with the

baseline $B = 47$ and $\tau_k = 0.1$

$$\boldsymbol{\beta} \leftarrow \boldsymbol{\beta} + \tau_k \nabla_{\boldsymbol{\beta}} v(\boldsymbol{\beta}) = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} + 0.1 \begin{bmatrix} -1.1 \\ 1.1 \\ -6.1 \\ 6.1 \end{bmatrix} = \begin{bmatrix} -0.11 \\ 0.11 \\ -0.61 \\ 0.61 \end{bmatrix}.$$

With the baseline $B = 0$, the updated $\boldsymbol{\beta}$ would be

$$\begin{bmatrix} 0.83 \\ -0.83 \\ 4.56 \\ -4.56 \end{bmatrix}.$$

Thus in both cases, the batch update generates smaller increments in $\boldsymbol{\beta}$ and might be more stable.

A numerical experiment

This section describes a more detailed numerical study that implements Algorithm 11.11 in the context of an optimal stopping problem on $S = \{-50, \dots, 50\}$ and $g(s) = -s^2$. The objective is to maximize the expected total (undiscounted) reward averaged over the initial state distribution. With this choice for $g(s)$ the goal is stop as close to zero as possible, taking into account the expected cost to reach it.

Figure 11.10 displays the *optimal* value functions and stopping region for three choices of the continuation cost. The optimal value function was found to a high degree of accuracy using value iteration with $v^0(s) = -50^2$. This starting value is a lower bound on the optimal value function, guaranteeing monotone convergence of value iteration (see Chapter 6).

To apply the policy gradient algorithm above, first assume a tabular representation so that features have the form

$$b(s, a) = I_{\{s', a'\}}(s, a), \text{ for } s' \in S \text{ and } a' \in \{C, Q\}$$

so that

$$w(a'|s'; \boldsymbol{\beta}) = \frac{e^{\eta \beta_{s', a'}}}{\sum_{s=-50}^{50} (e^{\eta \beta_{s, C}} + e^{\eta \beta_{s, Q}})}, \quad (11.68)$$

where $\beta_{s', a'}$ is the coefficient corresponding to $b(s', a')$.

For this model, the gradient is available in closed form as:

$$\frac{\partial}{\partial \beta_{s,a}} \ln w(a'|s'; \boldsymbol{\beta}) = \begin{cases} \eta(1 - w(a|s'; \boldsymbol{\beta})) & (s, a) = (s', a') \\ -\eta w(a|s'; \boldsymbol{\beta}) & (s, a) = (s', a) \text{ and } a \neq a' \\ 0 & \text{otherwise.} \end{cases} \quad (11.69)$$

Thus, the only positive term in the gradient corresponds to the state-action pair that is being evaluated. Moreover, the factor η can be absorbed into the learning rate.

Algorithm 11.11 was implemented with the following specifications:

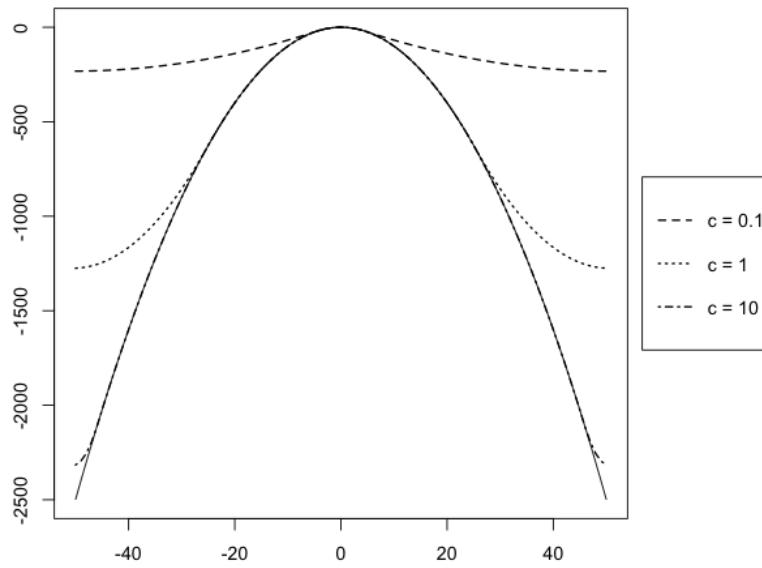


Figure 11.10: Optimal value functions (dashed lines) for three choices of c for the problem of optimally stopping in a random walk with $p = 0.5$. The solid line shows the reward of stopping in each state. The optimal policy is to stop when the optimal value function and the reward from stopping agree. Observe that the stopping region increases with respect to the continuation cost.

- Uniform initial distribution on S ;
- $c \in \{0.1, 10, 50\}$;
- $p = 0.5$;
- β initialized to $\mathbf{0}$;
- $K = 500,000$;
- episodes were truncated after 100 steps;
- softmax exponent $\eta = 1/100$;
- learning rate, $\tau_k = 1,000/(10,000 + k)$, where k represented the episode number, and
- baseline $B(s) = -50^2$ for all s .

The baseline was chosen so as to avoid overflow in probability estimates.

Figure 11.11 below shows estimates of β for the three choices of c for a single (typical) replicate and as well for a discounted model with $c = 0$ and the discount rate

$\lambda = 0.95$. The results obtained using a batch implementation of Algorithm 11.11 were similar.

To better understand the trends within the components of β for each action, fourth-order polynomials were subsequently fit to the individual, often noisy, component values (represented by dots). The resulting lines, solid for Q and dashed for C , smooth out this variability, making the underlying patterns more readily interpretable.

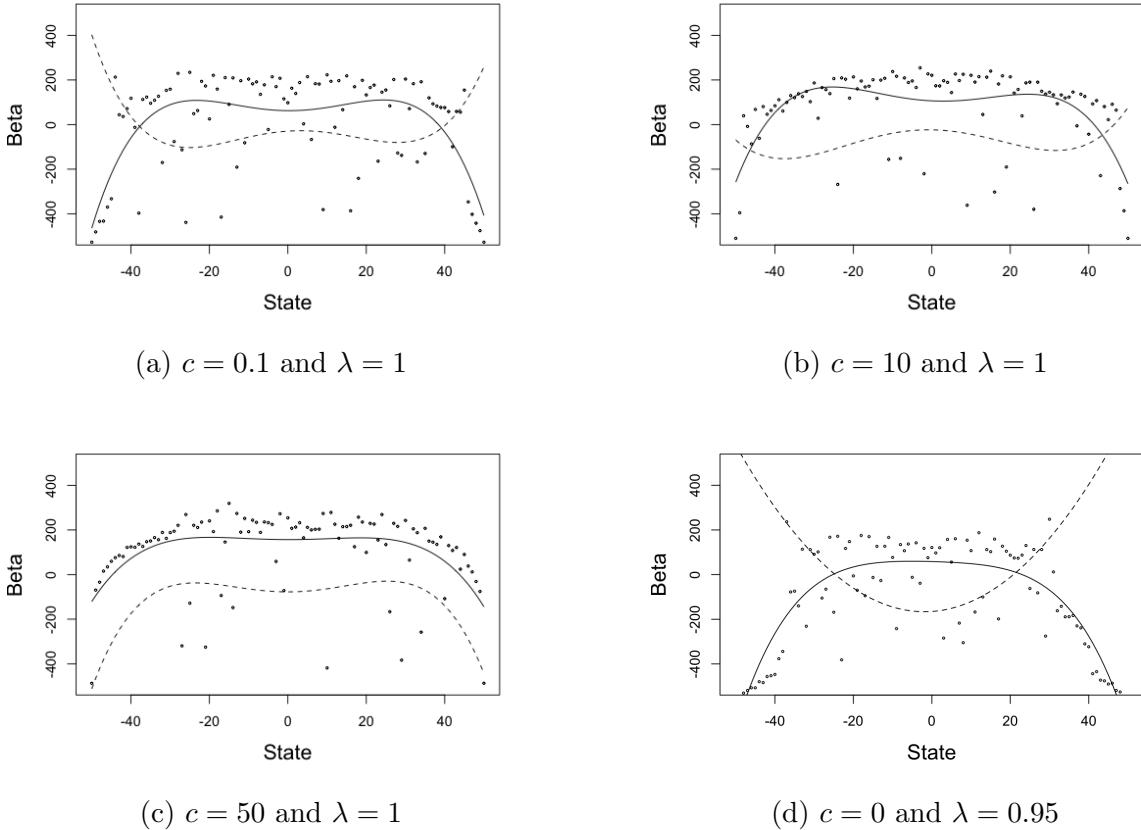


Figure 11.11: Parameter estimates obtained applying Algorithm 11.11 to optimal stopping on a random walk. The dots correspond to the estimates of $\beta_{s,Q}$, the solid line to the fit to these points using a fourth order polynomial and the dashed line indicates the fitted values to $\beta_{s,C}$ using a fourth order polynomial.

Note that when the solid line lies above the dashed line, it is more likely for the policy to stop²⁸. Thus, these figures show that the stopping region is centered on $s = 0$ and increases with respect to c . When c is large, it is optimal to stop in all states, and when $c = 0.1$, and in the discounted model, the stopping region is narrower. Note that when $c = 0.1$, the policy found using the Algorithm 11.11 differs considerably from the optimal policy represented in Figure 11.10.

²⁸Alternatively, the estimates can be plotted on the probability scale.

To conclude the analysis, consider an implementation that fits a fourth-order polynomial using centered and standardized states \tilde{s} as features for each action. That is

$$w(a|s; \boldsymbol{\beta}) = \frac{e^{\eta(\sum_{i=0}^4 \beta_{i,a} \tilde{s}^i)}}{\sum_{k=\{C,Q\}} e^{\eta(\sum_{i=0}^4 \beta_{i,k} \tilde{s}^i)}}$$

for $a \in \{C, Q\}$. This model has 10 parameters as opposed to the tabular model above which has 202 parameters. It is convenient to represent $\boldsymbol{\beta}$ in matrix form as

$$\boldsymbol{\beta} = \begin{bmatrix} \beta_{0,C} & \beta_{0,Q} \\ \beta_{1,C} & \beta_{1,Q} \\ \beta_{2,C} & \beta_{2,Q} \\ \beta_{3,C} & \beta_{3,Q} \\ \beta_{4,C} & \beta_{4,Q} \end{bmatrix} := [\boldsymbol{\beta}_C \quad \boldsymbol{\beta}_Q]$$

Again, the gradient of $\ln w(a|s; \boldsymbol{\beta})$ is available in closed form with components

$$\frac{\partial}{\partial \beta_{i,j}} \ln w(C|s; \boldsymbol{\beta}) = \begin{cases} \eta(1 - w(C|s; \boldsymbol{\beta})) \tilde{s}^i & j = C \\ -\eta w(Q|s; \boldsymbol{\beta}) \tilde{s}^i & j = Q \end{cases} \quad (11.70)$$

and

$$\frac{\partial}{\partial \beta_{i,j}} \ln w(Q|s; \boldsymbol{\beta}) = \begin{cases} -\eta w(C|s; \boldsymbol{\beta}) \tilde{s}^i & j = C \\ \eta(1 - w(Q|s; \boldsymbol{\beta})) \tilde{s}^i & j = Q. \end{cases}$$

Note that $w(Q|s; \boldsymbol{\beta}) = 1 - w(C|s; \boldsymbol{\beta})$, so that the above expressions can be simplified further. They are left in the above form in order to easily generalize to cases with more than two actions.

Figure 11.12 shows the fitted exponents as a function of the state using a fourth degree polynomial with $\eta = 1/100$ and $B(s) = -0.3(50)^2$. Observe that the stopping regions differ from the optimal in the case $c = 0.1$ and $\lambda = 1.0$, and also from those obtained using a polynomial fit applied to results from the tabular model. Note also that with these baseline and parameter choices, the estimates when $c = 10$ required a larger value of η in the softmax function to converge.

11.5.6 Policy gradient in a discounted model

As noted frequently, an infinite horizon discounted model may be analyzed through simulation as either:

1. a random horizon expected total reward model where the horizon length is sampled from a geometric distribution with parameter λ , or
2. a finite horizon model in which the total discounted reward is truncated at a fixed decision epoch.

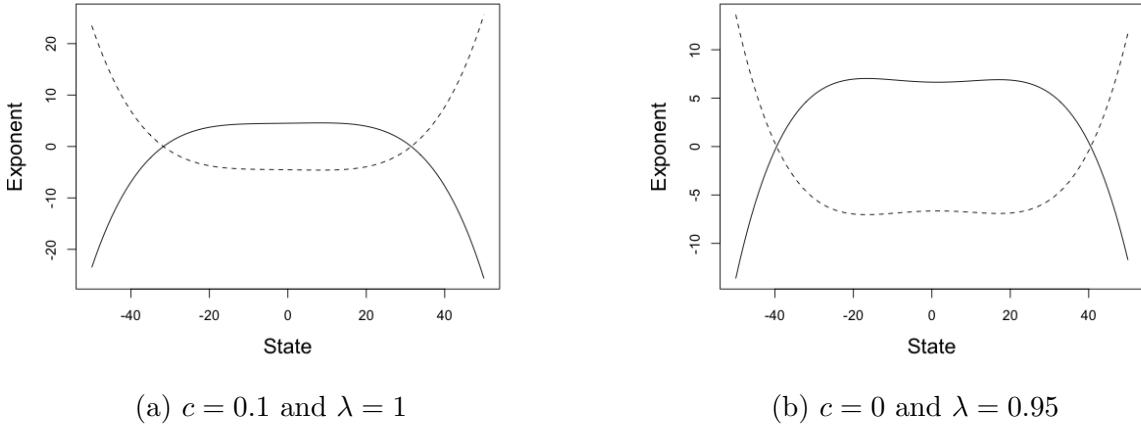


Figure 11.12: Exponent values of the softmax function as a function of the state, obtained by applying Algorithm 11.11 to the problem of optimal stopping on a random walk. The values are derived from estimates of the coefficients of a fourth-order polynomial for each action in a single replicate. The solid line corresponds to stopping and the dashed line to continuing.

Algorithm 11.11 applies directly to the first representation where the stopping time is either sampled before each episode or realized by sampling from a Bernoulli distribution at each decision epoch within an episode. To apply truncation, (11.65) must be modified to include the discount factor as follows:

$$\boldsymbol{\beta} \leftarrow \boldsymbol{\beta} + \tau_k \nabla_{\boldsymbol{\beta}} \ln(w(a^n | s^n; \boldsymbol{\beta})) \left(\sum_{i=n}^N \lambda^{i-n} r^i - B(s^n) \right) \quad (11.71)$$

Otherwise the algorithm is applied directly.

A numerical example

This example illustrates the truncation approach applied to the frequently analyzed two-state model. Experiments used softmax action selection probabilities and tabular representations for the policy so that $\boldsymbol{\beta} = (\beta_{1,1}, \beta_{1,2}, \beta_{2,1}, \beta_{2,2})$, where the first subscript corresponds to the state and the second subscript corresponds to the action. The implementation used a discount rate $\lambda = 0.9$, $K = 1,000$ episodes, truncation at $N = 100$, learning rate $\tau_k = 50/(1,000 + n)$ where k denotes the episode number, and weights initialized at $\boldsymbol{\beta} = \mathbf{0}$. Experiments explored the impact of the softmax exponent and the baseline over varying random number seeds.

By choosing the parameter of the softmax function $\eta = 1/10$, the impact of baseline was negligible and the algorithm terminated with action selection probabilities close to 1 for the optimal policy across a wide range of random number seeds. For example,

for a specific random number seed

$$w(a_{1,2}|s_1; \beta) = 0.988 \quad \text{and} \quad w(a_{2,2}|s_2; \beta) = 0.999.$$

Using the softmax parameter $\eta = 1$ resulted in frequent convergence to a non-optimal policy.

11.5.7 Policy gradient: Concluding remarks

In the discrete domain numerical examples in this section, the policy gradient algorithm was extremely sensitive to algorithmic specification including: initialization, baseline specification, state and reward scaling, softmax scaling, and learning rate.

Configurations that identified an optimal policy for one random number seed often converged to a non-optimal policy for another. These results are consistent with observations in the literature²⁹, which noted that:

- Experiments are difficult to reproduce because numerical results are sensitive to hyper-parameters and can vary over random number seeds.
- A major share of claimed performance increments is achieved less by innovative algorithmic properties but more through clever implementation.
- Results obtained using algorithms based on neural networks can be achieved by simpler models based on linear function approximations.

Therefore, we encourage the reader to experiment with algorithmic features when attempting to use the above policy gradient methods.

11.6 Actor-critic algorithms: Combining policy and policy value function approximation

The previous section argued that adding a baseline to the policy gradient algorithm reduces variance and improves convergence. Therefore, the expression

$$\delta(s^n) := \sum_{i=n}^N r^i - B(s^n) \tag{11.72}$$

in (11.65) in the episodic policy gradient algorithm needs to be looked at more closely. In this expression, N represents a realization of the episode length and $\sum_{i=n}^N r^i$ a realization of the total reward from decision epoch n onward starting in state³⁰ s^n and choosing action a^n .

²⁹For instance Gronauer et al. [2021]

³⁰Recall superscripts correspond to decision epochs.

In other words, the sum represents a realization of the state-action value function $q^{(d_{\beta})^\infty}(s^n, a^n)$ under the randomized stationary policy $(d_{\beta})^\infty$ resulting from using the weights β . Realizing this, it has been shown³¹ that $v^{(d_{\beta})^\infty}(s^n)$, the expected value of the policy $(d_{\beta})^\infty$ starting in state s^n , minimizes the variance of the gradient estimation for the specified policy. In other words, the optimal choice of the baseline is an appropriately specified policy value function.

To simplify notation, write $q(s, a; \beta) := q^{(d_{\beta})^\infty}(s, a)$ and $v(s; \beta) := v^{(d_{\beta})^\infty}(s)$. Then

- If $q(s, a; \beta) - v(s; \beta) > 0$, it is desirable to increase the probability of selecting action a in state s .
- If $q(s, a; \beta) - v(s; \beta) < 0$, it is desirable to decrease the probability of choosing action a in state s .

Thus, with this choice of baseline, *on average* the policy gradient algorithm will generate a new weight vector that corresponds to a policy with an increased value. The expression “on average” accounts for stochastic variation in the underlying simulation or process.

Advantage functions

Definition 11.1. The *advantage function*³² is defined as

$$A(s, a; \beta) := q(s, a; \beta) - v(s; \beta) \quad (11.73)$$

for $s \in S$, $a \in A_s$ and real-valued vectors β of appropriate dimension.

³²Hopefully, the use of A to represent the advantage function will not cause confusion. Recall that A_s denotes the set of actions in state s .

As argued above, the sign and magnitude of the advantage function impact the change in parameter values and consequently the probability of selecting the designated action. Since $q(s, a; \beta)$ and $v(s; \beta)$ are not known, the challenge is to incorporate their difference, namely the advantage function, into algorithms. Expressing algorithms in terms of the advantage function can be beneficial, as it may eliminate the need to compute its components separately.

Note that the gradient of the policy value function can be represented by

$$\nabla_{\beta} v(\beta) = E^{(d_{\beta})^\infty} \left[\sum_{j=1}^{N_{\Delta}} \nabla_{\beta} \ln(w(Y_j | X_j; \beta)) A(X_j, Y_j; \beta) \right]. \quad (11.74)$$

³¹See Sutton et al. [1999] or Grondman et al. [2012].

This representation takes into account that

$$E^{(d_{\beta})\infty} \left[\sum_{j=1}^{N_{\Delta}} \nabla_{\beta} \ln (w(Y_j|X_j; \beta)) v(s; \beta) \right] = 0.$$

Actor-critic algorithm overview

Algorithms that use policy value function estimates as baselines in a policy gradient algorithm are known as *actor-critic algorithms*. The *actor* corresponds to the policy and the *critic* to the value of that policy. Feedback from the critic enables the actor to improve the policy. Policy gradient algorithms may be referred to as actor-only algorithms and Q-learning algorithms as critic-only algorithms. The benefit of the combined actor-critic structure is that while Q-learning with function approximation may diverge, policy gradient algorithms are typically more stable and can converge to a local optimum under reasonable assumptions.

Actor-critic algorithms are based on approximating both the policy and advantage function (or its components) by weighted combinations of features. Let $\mathbf{b}_A(s, a)$ denote the vector of actor (randomized policy) features and β^A denote the corresponding vector of weights. Similarly, let $\mathbf{b}_C(s)$ denote the vector of critic (policy value function) features and β^C denote the corresponding vector of critic weights. Note that the form of features depend on whether value functions or state-action value functions are being used.

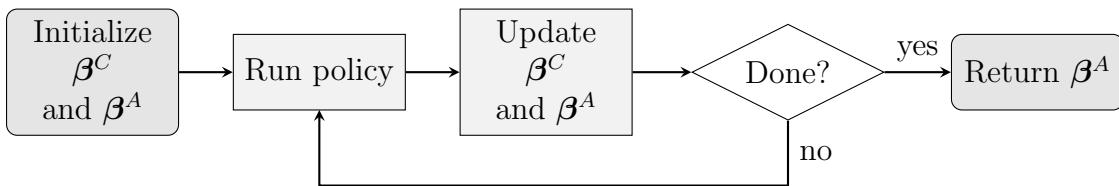


Figure 11.13: Steps in an actor-critic algorithm.

Figure 11.13 provides a high level schematic of an actor-critic algorithm. The algorithm can be implemented online or in batch mode. The former is suitable for continuing tasks modelled by an infinite horizon discounted (or average reward) model, while the latter is designed for episodic tasks (but can be used for discounted models by truncating rewards or including a geometric stopping time).

In an online implementation, “Run policy” generates a single transition after which both weight vectors are updated. In batch mode, “Run policy” generates a trajectory of states, actions and rewards using the current policy weights β^A and then both β^A and β^C are updated. Variants of the algorithm interchange the order of updating the weights as well as the method. The critic weights β^C can be updated using TD(0), TD(γ) or iterative least squares and the actor weights β^A can be updated using policy gradient with the baseline set equal to an estimate of the critic.

Note that an actor-critic algorithm avoids the need to explicitly evaluate an ϵ -tweaked policy in Q-policy iteration.

11.6.1 An online actor-critic algorithm

The online actor-critic algorithm is the easiest to describe. It applies to discounted infinite horizon applications. It is stated assuming a linear policy value function approximation

$$v(s; \boldsymbol{\beta}^C) = (\boldsymbol{\beta}^C)^\top \mathbf{b}_C(s).$$

Algorithm 11.12. Online actor-critic for a discounted model

1. **Initialize:**

- (a) Specify $\boldsymbol{\beta}^C$ and $\boldsymbol{\beta}^A$.
- (b) Specify learning rate sequences $\tau_n^C, n = 1, 2, \dots$, and $\tau_n^A, n = 1, 2, \dots$.
- (c) Specify the number of iterations N and $n \leftarrow 1$.
- (d) Specify $s \in S$.

2. **Iterate:** While $n \leq N$:

- (a) Sample a from $w(\cdot|s; \boldsymbol{\beta}^A)$.
- (b) Simulate (s', r) or sample $s' \in S$ from $p(\cdot|s, a)$ and set $r \leftarrow r(s, a, s')$.
- (c) **Evaluate advantage:**

$$A(s, a; \boldsymbol{\beta}^C) \leftarrow r + \lambda v(s'; \boldsymbol{\beta}^C) - v(s; \boldsymbol{\beta}^C). \quad (11.75)$$

(d) **Update critic:**

$$\boldsymbol{\beta}^C \leftarrow \boldsymbol{\beta}^C + \tau_n^C A(s, a; \boldsymbol{\beta}^C) \mathbf{b}_C(s). \quad (11.76)$$

(e) **Update actor:**

$$\boldsymbol{\beta}^A \leftarrow \boldsymbol{\beta}^A + \tau_n^A \nabla_{\boldsymbol{\beta}^A} \ln(w(a|s; \boldsymbol{\beta}^A)) A(s, a; \boldsymbol{\beta}^C). \quad (11.77)$$

- (f) $n \leftarrow n + 1$ and $s \leftarrow s'$.

3. **Terminate:** Return $\boldsymbol{\beta}^A$.

Some comments about this algorithm follow:

- As stated, the critic update uses $\text{TD}(0)$. To see this, express (11.76) as:

$$\boldsymbol{\beta}^C \leftarrow \boldsymbol{\beta}^C + \tau_n^C (r + \lambda v(s'; \boldsymbol{\beta}^C) - v(s; \boldsymbol{\beta}^C)) \mathbf{b}_C(s).$$

Alternatively, a $\text{TD}(\gamma)$ update may be used in which case the critic update step will be more complex.

- The advantage function (11.75) depends on the action a chosen in step 2(a).
- Note that the same temporal difference (advantage) is used to update the actor and critic through (11.76) and (11.77). This means that the updated critic parameter is not used until the next iterate to update the actor. Using the updated critic in the same iteration has been explored in the literature and found to be less stable.
- If a nonlinear policy value function approximation is used, $\nabla_{\boldsymbol{\beta}^C} v(s; \boldsymbol{\beta}^C)$ replaces the expression $\mathbf{b}_C(s)$ in (11.76).
- The advantage estimates $q(s, a)$ with its sampled version $r + \lambda v(s'; \boldsymbol{\beta}^C)$. Alternatively, one can approximate $q(s, a; \boldsymbol{\beta}^C)$ and estimate its weights iteratively.
- Many variants in the literature consider updates of the critic based on sampling states and rewards from parallel implementations.

Convergence of actor-critic

The following conditions³² ensure convergence with probability one under linear function approximation:

- Slower evaluation of the actor:** To ensure that the critic reliably evaluates the actor, its learning rate should be faster than that of the actor. This occurs, for example, if

$$\lim_{n \rightarrow \infty} \frac{\tau_n^A}{\tau_n^C} = 0. \quad (11.78)$$

- Compatible features:** The critic's approximation of the advantage function must lie in the space spanned by the policy's score function. That is, the critic must approximate the advantage by:

$$\hat{A}(s, a) = \mathbf{u}^\top \nabla_{\boldsymbol{\beta}^A} \ln w(a|s; \boldsymbol{\beta}^A)$$

for some \mathbf{u} . This ensures unbiased estimation of the policy gradient.

- The usual conditions:** Both learning rates must satisfy the Robbins-Monro conditions, features and rewards must be bounded, and all states visited infinitely often under the Markov chain of all policies.

³²See Konda and Tsitsiklis [2000] and Sutton et al. [1999].

In practical applications, such as when the advantage is approximated by the temporal difference (11.75) or when using nonlinear function approximations such as neural networks, the compatibility assumption and others may be violated. Despite the lack of formal convergence guarantees in these settings, actor–critic algorithms often perform well empirically.

An example

This example describes in some detail the application of the online actor-critic algorithm to the frequently analyzed two-state example.

Example 11.6. This example applies Algorithm 11.12 to a discounted ($\lambda = 0.9$) version of the frequently analyzed two-state example. It uses a tabular representation with state-action indicators as features for the actor and state indicators as features for the critic. That is, the features for the critic are represented by the two-dimensional vector $\mathbf{b}_C(s)$ with components $b_C(s_i) = I_{s_i}(s)$ for $i = 1, 2$ and the features for the actor are represented by the four-dimensional vector $\mathbf{b}_A(s, j)$ with components $b_A(s_i, a_{i,j}) = I_{(s_i, a_{i,j})}(a, s)$ for $i = 1, 2$ and $j = 1, 2$.

This means that for $s \in S = \{s_1, s_2\}$

$$v(s_i; \boldsymbol{\beta}^C) = \beta_{s_i}^C \quad \text{for } i = 1, 2,$$

where $\boldsymbol{\beta}^C = (\beta_{s_1}^C, \beta_{s_2}^C)$.

To implement the above algorithm, sample a from $w(a|s; \boldsymbol{\beta}^A)$, observe (s', r) and compute the advantage function:

$$A(s, a; \boldsymbol{\beta}^C) = r + \lambda v(s'; \boldsymbol{\beta}^C) - v(s; \boldsymbol{\beta}^C) = r + \lambda \beta_{s'}^C - \beta_s^C.$$

Using this notation, the critic update equation (11.76) becomes

$$\beta_s^C = v(s; \boldsymbol{\beta}^C) \leftarrow v(s; \boldsymbol{\beta}^C) + \tau_n^C (r + \lambda v(s'; \boldsymbol{\beta}^C) - v(s; \boldsymbol{\beta}^C)) = \beta_s^C + \tau_n^C (r + \lambda \beta_{s'}^C - \beta_s^C).$$

The actor update equation (11.77) is given by

$$\boldsymbol{\beta}^A \leftarrow \boldsymbol{\beta}^A + \tau_n^A \nabla_{\boldsymbol{\beta}^A} \ln(w(a|s; \boldsymbol{\beta}^A)) A(s, a; \boldsymbol{\beta}^C).$$

This means that updates of the actor weights are given by

$$\beta_{s', a'}^A \leftarrow \begin{cases} \beta_{s, a}^A + \tau_n^A (1 - w(a|s; \boldsymbol{\beta}^A)) A(s, a; \boldsymbol{\beta}^C) & \text{if } s' = s, a' = a \\ \beta_{s, a}^A + \tau_n^A (-w(a|s; \boldsymbol{\beta}^A)) A(s, a; \boldsymbol{\beta}^C) & \text{if } s' = s, a' \neq a \\ \beta_{s, a}^A & \text{if } s' \neq s. \end{cases}$$

Note that at each iteration, the critic is updated only in the observed state s , while the actor weights are updated for all actions in state s .

The algorithm converged to the optimal value function and policy for a wide range of learning rates and numbers of iterations. Figure 11.14 shows the sequences of values for a typical replicate with $N = 30,000$ iterations and $\tau_n^A = 100/(1,000 + n^{0.6})$ and $\tau_n^C = 100/(1,000 + n)$ satisfying the time scale condition (11.78). In this replicate the estimated policy selected actions with probabilities:

$$w(a_{1,2}|s_1; \beta^A) = 1.00 \quad \text{and} \quad w(a_{2,2}|s_2; \beta^A) = 1.00,$$

in agreement with the deterministic optimal policy. The critic estimate of the optimal value function was (29.97, 27.26) in close agreement with optimal values (see Figure 11.14).

It was also observed that the algorithm converged to the optimal policy for random number seeds in which the policy gradient algorithm (Algorithm 11.11) converged to a sub-optimal policy. Thus, by replacing an arbitrary baseline with a critic that tracked the “current policy”, the algorithm avoided convergence to a sub-optimal policy.

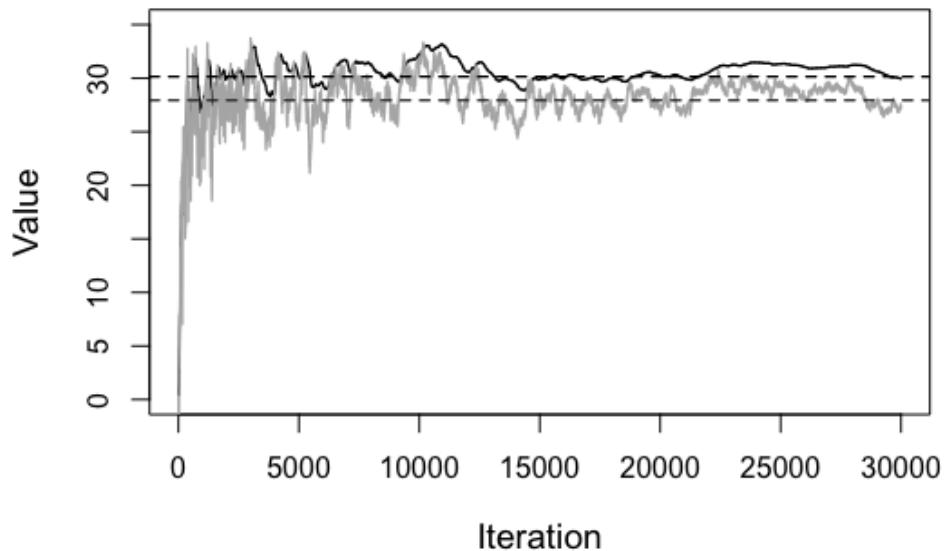


Figure 11.14: Optimal value function estimates in two-state example using Algorithm 11.12 obtained for a typical replicate. Black lines correspond to $v^*(s_1)$ and gray lines to $v^*(s_2)$; solid lines correspond to estimates and dashed lines to the exact value.

11.6.2 A batch actor-critic algorithm

The following algorithm describes an actor-critic implementation suitable for undiscounted episodic models. It generalizes the policy gradient algorithm above (Algorithm 11.11) by using the critic to update the baseline.

The algorithm as stated updates the actor and critic after every episode; the critic update (policy value function estimate) uses the whole episode, while the actor is updated sequentially after updating the critic. An alternative would be to also use a batch update of the actor represented by (11.66).

Algorithm 11.13. Batch actor-critic for an episodic model

1. **Initialize:**

- (a) Specify β^A .
- (b) Specify learning rate sequence $\tau_n^A, n = 1, 2, \dots$
- (c) Specify the number of episodes K and set $k \leftarrow 1$.

2. **Iterate:** While $k \leq K$:

- (a) **Generate an episode:** Generate an episode using $w(a|s; \beta^A)$ to generate actions. Save the sequence of states, actions and rewards

$$(s^1, a^1, s^2, r^1, \dots, s^N, a^N, s^{N+1}, r^N),$$

where N denotes the termination epoch of the episode.

- (b) **Update critic:** Choose

$$\beta^C \in \arg \min_{\beta} \sum_{n=1}^N \left(\sum_{i=n}^N r^i - v(s^n; \beta) \right)^2. \quad (11.79)$$

- (c) $n \leftarrow 1$.

- (d) **Update actor:** While $n \leq N$:

i. **Evaluate advantage:**

$$A(s^n, a^n; \beta^C) \leftarrow \sum_{i=n}^N r^i - v(s^n; \beta^C). \quad (11.80)$$

ii. **Update actor weights:**

$$\beta^A \leftarrow \beta^A + \tau_n^A \nabla_{\beta^A} \ln(w(a^n|s^n; \beta^A)) A(s^n, a^n; \beta^C) \quad (11.81)$$

- iii. $n \leftarrow n + 1$.

- (e) $k \leftarrow k + 1$.

3. **Terminate:** Return β^A .

Some comments on the algorithm follow.

1. Critic weights β^C can also be updated after updating actor weights β^A . Our experience, as illustrated by the example below, is that the algorithm converged more reliably when the critic was updated before updating the actor.
2. The above implementation does not explicitly describe how to estimate the critic weights. Alternatives include applying linear or nonlinear regression or alternatively to minimize

$$h(\beta) := \sum_{n=1}^N \left(\sum_{i=n}^N r^i - v(s^n; \beta) \right)^2$$

by gradient descent. To do so, an algorithm would compute

$$\nabla_\beta h(\beta) = \nabla_\beta \left(\sum_{n=1}^N \left(\sum_{i=n}^N r^i - v(s^n; \beta) \right)^2 \right),$$

which in the linear case equals

$$\nabla_\beta h(\beta) = -2 \sum_{n=1}^N \left(\sum_{k=n}^N r^k - v(s^n; \beta) \right) \mathbf{b}^C(s^n),$$

where $\mathbf{b}^C(s)$ denotes the critic features evaluated at s . Absorbing the constant into the learning rate gives the recursion for β^C :

$$\beta^C \leftarrow \beta^C - \tau_n^C \nabla_\beta h(\beta). \quad (11.82)$$

Note that this approach requires specifying an additional learning rate and monitoring the stability of gradient estimates.

3. The above algorithm uses the same advantage function as that used by the policy gradient algorithm. An alternative is to use the single-step (bootstrapped) advantage function:

$$A(s^n, a^n; \beta^C) := r^n + v(s^{n+1}; \beta^C) - v(s^n; \beta^C) \quad (11.83)$$

4. Instead of using policy value functions, one could evaluate the advantage in terms of the state-action value function $q(s, a; \beta^C)$ as follows

$$A(s^n, a^n; \beta^C) := q(s^{n+1}, a^{n+1}; \beta^C) - q(s^n, a^n; \beta^C), \quad (11.84)$$

or even update it directly.

An example

This example applies Algorithm 11.13 to an instance of the shortest path model in Section 11.3.3 with $M = 10$, $N = 7$, $(m^*, n^*) = (10, 7)$, $R = 10$ and $c = 0.1$. In this application the robot seeks to learn a path to cell $(10, 7)$ from each starting cell. The implementation represents the critic by the model

$$v((m, n); \boldsymbol{\beta}^C) = \beta_0^C + \beta_{1,0}^C m + \beta_{0,1}^C n + \beta_{1,1}^C mn + \beta_{2,0}^C m^2 + \beta_{0,2}^C n^2$$

so that $\boldsymbol{\beta}^C = (\beta_0^C, \beta_{1,0}^C, \beta_{0,1}^C, \beta_{1,1}^C, \beta_{2,0}^C, \beta_{0,2}^C)$. Critic weights, $\boldsymbol{\beta}^C$, are computed in step 2(b) using ordinary least squares. The actor is represented in tabular form with $\boldsymbol{\beta}^A = \{\beta_{(m,n,a)}^A \mid (m, n) \in S \text{ and } a \in A_{(m,n)}\}$.

The algorithm was initiated with components of $\boldsymbol{\beta}^A$ sampled from a standard normal distribution so as to avoid too many long episodes. Episodes were truncated at 200 iterations and the algorithm was run for 50,000 episodes, each starting from a random cell that differed from the target. The learning rate was $\tau_n^A = 40/(400 + n)$.

Figure 11.15 shows how the total reward per episode varies within a single replicate starting in cell $(1, 1)$ and at random. Observe that the reward increases and stabilizes with more variability in total rewards when starting in a random cell than when starting in cell $(1, 1)$. The advantage of the random start is that it identifies good policies for infrequently visited states.

Note that when starting in cell $(1, 1)$, the actor-critic algorithm often finds a sequence of actions³³ with probabilities close to 1 that correspond to a shortest path through the grid. However, since most episodes will follow this path, the optimal probabilities off the path are not well estimated. Moreover, when the critic was evaluated *after* the actor, the algorithm converged to sub-optimal policies that cycled between states and never reached the target cell.

The policy gradient algorithm (Algorithm 11.11) was also applied to this example. The implementation was the same as above but instead of using the critic, it set the baseline (B in Table 11.7) equal to 0, -8 or the (running) mean of the previous total rewards accumulated from the starting state. Observe from Table 11.7 that the quality of estimates varied significantly with the baseline, with the running mean giving the best results. In fact, the policy gradient algorithm with baseline equal to the running mean of total rewards gave a higher total reward than the actor-critic algorithm. However, there were many instances where the policy gradient diverged and the actor-critic algorithm converged to a good policy.

Delivering coffee

This section applies the actor-critic algorithm to find an optimal policy for the coffee delivering robot problem from Section 3.2. Rewards and costs are divided by 10 to

³³For example, when starting in the upper left corner of the grid, the sequence: “down”, “down”, “down”, “down”, “down”, “right”, “down”, “down”, “down”, “right”, “right”, “down”, “right”, “right”, “right”, “right”.

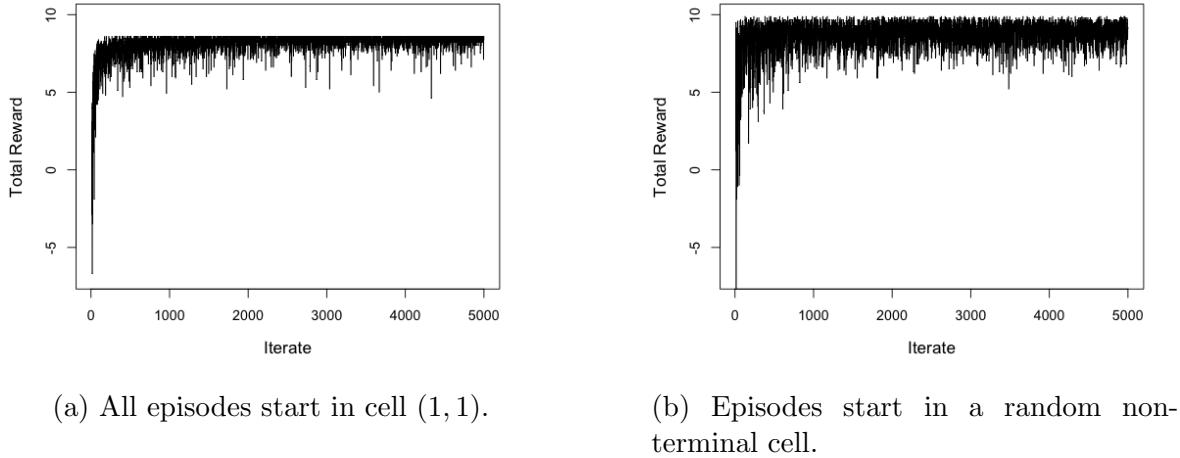


Figure 11.15: A typical replicate of the total reward per episode in a model with a 10×7 grid with the target cell in lower right corner.

Method	Total Reward mean	Total Reward SD
PG ($B = 0$)	6.31	6.14
PG ($B = -8$)	5.80	8.06
PG ($B = \text{mean}$)	8.52	0.47
AC	8.24	0.78

Table 11.7: Mean and standard deviation of total reward over all episodes for a single replicate in which all four methods converged. The results (using common random number seeds) were obtained using the policy gradient algorithm with baseline B and the actor-critic algorithm as described above.

improve numerical stability so that the per step cost is 0.1, the reward for delivering the coffee is 5 and the cost for falling down the stairs is 10. An episode ends when the robot successfully delivers the coffee; if it falls down the stairs, it returns to cell 13 and starts over. This example assumes that the robot's moves are deterministic in the sense that if it plans to go right and such a move is possible, it moves right. However, if there is a wall in the way, the robot stays in its current cell and incurs a cost associated with a single step. Randomness is introduced through $w(a|s; \beta^A)$.

Algorithmic details follow. The policy value function is approximated by

$$v((m, n); \beta^C) = \beta_{0,0}^C + \beta_{1,0}^C m + \beta_{0,1}^C n,$$

where m denotes the row (indexed from the bottom) and n denotes the column (indexed from the left)³⁴. Its parameters are estimated in step 2(b) of Algorithm 11.13 using

³⁴For example, cell 13 corresponds to row 1 and column 1.

linear regression. The actor is again represented by indicators of state-action pairs and estimated (by gradient ascent) using a learning rate of $\tau_n^A = 40/(400 + n)$ where n is the episode number. The model is run for 20,000 episodes with initial values for β^A generated randomly from a standard normal distribution.

This implementation identified an optimal policy for roughly half of the random number seeds. When it did not, it identified a policy that cycled without delivering the coffee. Note that for a few earlier episodes, the robot incurred a cost of 10 when it fell down the stairs but eventually it learned to avoid such incidents. For this model, the optimal value function estimate was

$$v((m, n); \hat{\beta}^C) = 4.164 + 0.127m + 0.055n.$$

With this value function, the effect of moving one row closer to the destination increases the reward by more than twice as much as moving one column to the right.

A larger grid

The batch actor-critic algorithm was also applied to a variant of the above problem on a larger 10×5 grid in which cell $(1, 1)$ corresponds to the upper left corner and $(10, 5)$ to the lower right corner. An obstacle (stairs) occupies cells $(4, 1)$ to $(8, 1)$ and the target cell is $(10, 1)$.

For an experiment with 30,000 episodes the critic was estimated to be

$$v((m, n); \hat{\beta}^C) = 3.30 + 0.16m + 0.003n.$$

In comparison to the critic estimate above, moving down one row³⁵ was considerably more valuable than moving right. In most cells, especially along the policy identified by Algorithm 11.13 in Figure 11.16, one direction choice had probability close to one. This path was slightly longer than optimal but kept the robot at a safe distance from the obstacle and completed the task. Note that for other random number seeds, the policy sometimes moved the robot closer to the obstacle and consequently fell down the stairs more often.

The sequence of total rewards per episode is shown in Figure 11.17. Observe that in several episodes the robot fell down the stairs more than once, with this event occurring less frequently in later episodes. Moreover, estimating the critic weights by gradient descent failed to identify a policy that reliably reached the destination.

11.6.3 Actor-critic methods: Concluding remarks and enhancements

Actor-critic algorithms combine the key features of the methods in Chapters 10 and 11. Namely, they combine the policy evaluation features of temporal differencing with the

³⁵So that coefficients are comparable, note that moving down in this formulation is equivalent to moving up in the previous formulation.

\rightarrow	\downarrow			
	\downarrow			
	\rightarrow	\rightarrow	\downarrow	
\times			\downarrow	
\times			\downarrow	
\times			\rightarrow	\downarrow
\times				\downarrow
\times			\downarrow	\leftarrow
			\downarrow	
*	\leftarrow	\leftarrow	\leftarrow	

Figure 11.16: Grid including policy identified by actor-critic algorithm for variant of the coffee delivering problem.

policy optimizing features of policy gradient methods. They provide a suite of elegant methods for solving problems using simulation.

Using a well-tuned critic reduces the variability observed in policy gradient methods and, by modifying the objective function, large swings in estimates of actor weights can be avoided. On the other hand, these algorithms contain many hyperparameters to tune and may limit exploration by using the current estimate of the actor for action selection.

Policy gradient and actor-critic algorithms have been researched extensively in the literature. The following variations have been suggested to enhance performance:

Normalization: Centering and scaling states and rewards, especially when they are highly variable) can enhance convergence.

Parallelization: Running many episodes in parallel (or on several identical robots) can provide better critic estimates.

Natural gradients: The gradients described above are sensitive to both the parameterization and the geometry (curvature) of the underlying surface. The natural gradient accounts for this in the actor by multiplying the gradient by the inverse of the Fisher information matrix given by

$$E \left[\left(\nabla_{\beta} \ln w(a|s; \beta) \right) \left(\nabla_{\beta} \ln w(a|s; \beta) \right)^T \right].$$

Generalized advantage estimation: Generalized advantage estimation provides a TD(γ)-like approach for improved estimation of the advantage in actor-critic algorithms. It is based on taking weighted sums of expressions of the form:

$$r^n + \dots + r^{n+m} + v(s^{n+m+1}; \beta^C) - v(s^n; \beta^C).$$

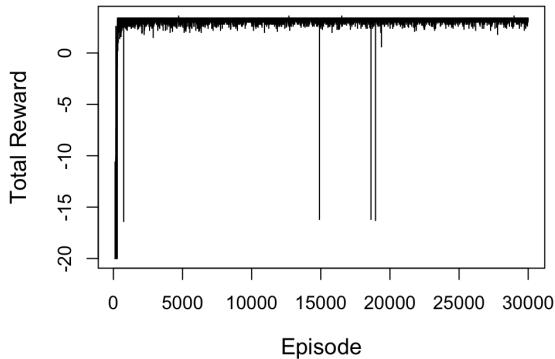


Figure 11.17: Total reward by iterate for a single replicate of the coffee delivering robot problem on a 10×5 grid obtained using Algorithm 11.13.

Improved gradient descent: Adaptive moment estimation³⁶ provides enhanced estimates of gradients based on moving averages of the mean and variance of the gradient. It is a widely used tool in gradient descent methods.

Policy and value function networks: Instead of using linear parameterizations for $w(a|s; \beta^A)$ and $v(s; \beta^C)$, one can replace them with neural networks using common or different features.

Thus, the analyst is faced with numerous options to improve the performance of policy gradient and actor-critic methods. However, achieving reliable results requires substantial experimentation and tuning. This part of the book has attempted to provide a foundational understanding of these methods and through several examples some guidance on how to apply them effectively.

11.7 Further topics

As the intent of this part of the book is to provide an introduction to reinforcement learning methods, it has been necessary to omit many topics. The following provides a brief overview of additional considerations.

11.7.1 Simultaneous learning and control

This book has frequently emphasized the distinction between model-based and model-free methods. Methods referred to as *model-based reinforcement learning* learn transition probabilities and rewards at the same time as they update a policy or value

³⁶See Kingma and Ba [2017] for details regarding the algorithm Adam.

function or both. Sometimes the learned models are referred to as *world models*. In them, the transition probabilities may be estimated by parametric functions including neural networks.

These approaches have been applied extensively in robotics (such as the Brio labyrinth described in the introduction to Part III of this book), game playing (Minecraft, chess and Go) and autonomous driving, where accurate modeling of the environment and the ability to plan based on simulated experiences is critical for success.

Benefits of this approach are:

Data augmentation: Training data can be augmented by data simulated from the derived model. The benefit of using simulation in this setting is that regions of the state-action space that may not be observed in real-life can be evaluated through simulation. For example, in the Brio labyrinth, one can simulate episodes starting at later points in the maze that would be reached infrequently during the learning phase.

Deriving state-action value functions from value functions: Given the challenge of specifying a functional form for state-action value functions, having a model enables the analyst to derive q -functions from value functions according to

$$q(s, a) = \hat{r}(s, a) + \sum_{j \in S} \hat{p}(j|s, a)v(j),$$

where $\hat{r}(s, a)$ and $\hat{p}(j|s, a)$ represent the estimated rewards and transition functions. Of course, computing this summation may be problematic in large models. However, in most real examples (with Gridworld as a prototype), $\hat{p}(\cdot|s, a)$ will be positive for only a few states. Alternatively, this expectation can be estimated by simulation.

Many options are available for how to combine data from the estimated model with data from real experiences. Algorithms that implement this approach include Dyna (Sutton and Barto [2018]) and DreamerV3 (?). The latter algorithm was used to find solutions to the Brio Labyrinth.

11.7.2 Experience replay

Experience replay is an offline method developed to improve the efficiency and stability of learning algorithms. It applies to Q-learning, policy gradient and actor-critic algorithms. The key concepts of experience replay include:

Replay buffer: A replay buffer (or memory) stores observed data encountered by the agent during training in the form (state, action, reward, next state, done?). This buffer can hold a fixed number of experiences; less relevant older experiences are discarded as new ones are added.

Random sampling: Instead of using the most recent experience to update weights, experiences are randomly sampled from the replay buffer. This breaks the temporal correlation between consecutive experiences that arise when using long trajectories, leading to more stable and efficient learning.

Batch updating: The agent updates its policy or state-action value function using a batch of experiences sampled from the replay buffer, rather than a single experience. In the actor-critic environment this allows for more robust gradient estimates and reduces the variance of updates.

The main benefits of using experience replay are:

Efficient use of data: By reusing past experiences, the agent can learn more from the same amount of data. This is important when data is expensive to collect and/or episodes are long.

Breaking correlations: Experiences encountered in a sequential manner are often highly correlated, which can lead to poor training performance. Experience replay mitigates this issue by shuffling the data.

Implementation issues include the size of the buffer, the refresh rate and the sampling strategy.

11.7.3 Deep learning

Deep learning refers to the use of high-dimensional neural networks to represent value functions and policies. We have chosen *not* to explicitly apply them in the book because using them presents a unique set of challenges. Neural networks provide alternatives to the function approximators described in this chapter. However, the simple examples analyzed herein do not warrant such powerful techniques.

Most modern applications of reinforcement learning use large-scale neural networks to approximate value functions, state-action value functions and policies.

11.7.4 Importance sampling

Importance sampling is a statistical technique used to estimate properties (expectations) of a specific (target) distribution while sampling from a different distribution. It is particularly useful when direct sampling from the target distribution is difficult, computationally expensive or does not provide wide coverage of the sample space. Importance sampling is used extensively in Bayesian statistics, risk analysis where costly rare events occur infrequently, and Monte Carlo methods, as it uses data more efficiently and can reduce variance of the estimates when properly applied.

In reinforcement learning, importance sampling is commonly used for off-policy learning, that is when the agent learns about a *target policy* from data generated by a *behavioral policy*. This is especially relevant for off-policy updates in policy gradient

methods but can also be applied to policy improvement like variants of Q-learning using experience replay or function approximation.

General use

Importance sampling works as follows. Suppose one seeks to estimate the expected value of a real-valued function $f(\cdot)$ with respect to a target distribution $q(\cdot)$ but wants to do so using samples obtained from a different distribution $p(\cdot)$. Assuming both distributions are discrete with the same domain X ,

$$E_q[f(X)] := \sum_{x \in X} f(x)q(x) = \sum_{x \in X} f(x) \frac{q(x)}{p(x)} p(x). \quad (11.85)$$

For this to make sense, it requires $p(x) > 0$ for all $x \in X$. As a result of (11.85),

$$E_q[f(X)] = E_p \left[f(X) \frac{q(X)}{p(X)} \right],$$

where the subscript on the expectation denotes the distribution used to compute the expectation. The quantities

$$\omega(x) := \frac{q(x)}{p(x)}$$

are referred to as *importance weights* or *importance ratios*.

Empirically, if x^1, \dots, x^N represent samples from $p(\cdot)$, an estimate of $E_q[f(X)]$ is given by

$$\widehat{E_q[f(X)]} = \frac{1}{N} \sum_{n=1}^N f(x^n) \frac{q(x^n)}{p(x^n)} = \frac{1}{N} \sum_{n=1}^N f(x^n) \omega(x^n).$$

Importance sampling in reinforcement learning

In reinforcement learning, importance sampling is used to evaluate or improve a target policy when using data obtained from implementing or simulating a behavioral policy. In particular, suppose actions are sampled from policy δ and one wishes to compute the expected reward in period 1 under policy γ starting in state s . Then

$$E^\gamma[r(X_1, Y_1, X_2)|X_1 = s] = E^\delta \left[r(X_1, Y_1, X_2) \frac{p(X_2|X_1, Y_1)w_\gamma(Y_1|X_1)}{p(X_2|X_1, Y_1)w_\delta(Y_1|X_1)} \middle| X_1 = s \right].$$

In this expression, the quantity

$$\omega(Y_1|s) := \frac{p(X_2|s, Y_1)w_\gamma(Y_1|s)}{p(X_2|s, Y_1)w_\delta(Y_1|s)} = \frac{w_\gamma(Y_1|s)}{w_\delta(Y_1|s)}$$

is the importance weight. It is noteworthy that the importance weight does not depend on the underlying transition probabilities that govern both policies.

Example 11.7. Consider a model with a single state and two actions $\{a_1, a_2\}$. Suppose the cost of choosing action a_1 is 10 and the cost of choosing action a_2 is 2. The objective is to estimate the expected cost under decision rule γ , which chooses action a_1 with probability 0.1. In a short realization, a_1 will be infrequently observed using γ .

Importance sampling can be used to estimate the cost under γ by sampling from a different policy, δ , which more frequently samples the high cost, low probability (important) action a_1 . In this setting, γ is the target policy and δ is the behavioral policy.

For concreteness, assume δ selects actions with equal probability. Suppose in a sample of size 5 using δ one observes the sequence $(a_2, a_1, a_2, a_2, a_1)$ and corresponding rewards $(2, 10, 2, 2, 10)$. Define the importance weights $\omega(a_1) = 0.1/0.5 = 0.2$ and $\omega(a_2) = 0.9/0.5 = 1.8$. Then the estimated cost under decision rule γ is

$$\frac{1}{5}(2 \cdot 1.8 + 10 \cdot 0.2 + 2 \cdot 1.8 + 2 \cdot 1.8 + 10 \cdot 0.2) = 2.9$$

Note that this well approximates the true expected cost 2.8. One would expect that estimates based on samples from γ would have high variance, hence requiring larger samples to estimate its mean accurately.

Suppose now that one wished to evaluate $E^{\gamma^\infty}[r(X_n, Y_n, X_{n+1})|X_1 = s]$ for arbitrary n . Then by a similar argument to that above

$$E^{\gamma^\infty}[r(X_n, Y_n, X_{n+1})|X_1 = s] = E^{\delta^\infty} \left[r(X_n, Y_n, X_{n+1}) \frac{w_\gamma(Y_1|X_1)}{w_\delta(Y_1|X_1)} \cdots \frac{w_\gamma(Y_n|X_n)}{w_\delta(Y_n|X_n)} \middle| X_1 = s \right].$$

Hence, by summing the expressions in the above equation, the value of policy γ^∞ can be calculated using trajectories generated by δ^∞ .

Such a calculation is fundamental in policy evaluation, especially when generating trajectories is costly. Using trajectories from one policy enables estimating expected costs for other policies. This is especially applicable when using policy gradient methods but also can be used in the Q-learning context.

See Sutton and Barto [2018] for examples of calculations based on this approach. Recent research, for example, Thomas and Brunskill [2016], has extended importance sampling methods in many ways.

11.7.5 Monte Carlo tree search

Monte Carlo tree search (MCTS) is a simulation-based method for evaluating the likelihood of an outcome (usually *win* or *lose*) when a specific action is chosen in a given state. It does so by rolling out and evaluating trajectories through simulation. MCTS applies to episodic models such as games against an opponent (where the outcome may

be *win* or *lose*) and bandit models. MCTS underlies Google’s AlphaGo and AlphaZero that obtained outstanding successes in Chess and Go.

The MCTS algorithm repeatedly executes the following steps:

1. **Select state:** Specify state to be evaluated.
2. **Select action:** Select an action on the basis of a current estimate of the probability of outcomes (payoffs or winning) for each action (or an upper confidence bound on this probability in multi-armed bandit models). Exploration can be introduced at this stage by sampling infrequently visited actions.
3. **Generate next state:** The action is implemented (in real life or a simulation) to obtain the next state. This may correspond to the opponent’s move after choosing your move.
4. **Generate a sample path:** Play or simulate a game until an outcome occurs or a state in which the value is accurately estimated is encountered. This is sometimes referred to as a *rollout*.
5. **Update path values:** Step back through the sample path and update the number of times a state has been visited and the number of times the game was won or task successfully completed when that state was reached. This is sometimes referred to as *back-propagation*, but its meaning differs from that used in neural networks.

The result of applying the above algorithm is a state-action value function that gives the probability of winning or the expected reward associated with choosing an action in a state. Use of this function can guide future strategies.

When applied to models with large state and action spaces such as Chess and Go, approximations are required. Often, neural networks are used to represent the values and opponent’s action choice probabilities.

Wordle

As an example of MCTS, consider the popular online game Wordle, which is available on the New York Times website.

The goal in Wordle is to guess an unknown target five-letter word in a minimum number of attempts by using feedback on the quality of the guess. Feedback is provided to the player by highlighting letters in the current guess as shown in Figure 11.18.

- Triangle, if the letter is in the same position as in the target word,
- Circle, if the letter is in the word but in the wrong position, and
- Nothing, if the letter is not in the word.



Figure 11.18: Typical Wordle output.

In Figure 11.18, the first guess “SALET” had no correct letters, the second guess “PRION” contained one correct letter “I” but in a different position than in the target word, and the fourth guess “WINDY” had two letters “I” and “Y” that were in the same position as in the target word. The keyboard below the guesses indicates the previously guessed letters with applicable highlighting (circle, triangle or no highlight) in accordance with their positions in the target word.

Note that the target word in this instance was “JIFFY”. Data provided by Wordle after completing the game showed that there were only three remaining words after guessing “WINDY”, namely “DIZZY”, “DITZY” and “JIFFY”. Moreover, the average number of steps to guess the target word over all players was 4.9. (Note this is a hard word to guess as the usual average is below 4.)

In the online version of the game, the hidden target word is chosen from a dictionary of 2,316 words and only 6 guesses are allowed. However, in modeling it, the game can proceed until a success is achieved.

This game can be analyzed using MCTS as follows. The state is rather complex; it encodes:

- Which letters have been guessed,
- For the guessed letters that are in the correct position in the target word, the exact positions they occupy, and
- For the guessed letters that are in the target word but not in the correct position, the positions previously guessed for those letters.

An action represents the word to guess, chosen from the set of not previously guessed words. Selection may be based on the distribution of the number of steps to find the

target word from each guess or the probability of eventually guessing the target word if a word is chosen.

After a guess, MCTS will generate sample paths of states and guesses starting from the current guess and ending when the target word is guessed. The value assigned to this state will be the number of guesses starting from the current state.

Clearly, this is non-trivial to implement and requires a more precise description of states and actions. Alternatively, Wordle can be solved using Q-learning.

11.7.6 Partially observable models

Reinforcement learning methods, especially policy gradient and actor-critic, apply directly to POMDPs. To use them, the observation, rather than the state, provides the input to the policy and/or a value function. That is, instead of using $w(a|s)$ to represent the policy and $v(s)$ to represent a policy value, one uses $w(a|o)$ and $v(o)$, where o denotes the observation. The consequence of doing so is that the action chosen by a policy is a function of an observation, rather than of the belief distribution over the unobservable states as analyzed in Chapter 8.

The benefit of this approach is that using observations simplifies the policy representation and learning process, as there is no need to update the belief state after each observation. However, policies based solely on observations might be less effective in environments where the history of observations is crucial for decision-making, as they lack explicit memory of past states and actions.

It would be informative to carry out a comprehensive study comparing these two approaches.

11.8 Technical appendix: Derivation of policy gradient representation

This appendix provides a proof of what is often referred to as the “policy gradient theorem”. This fundamental equivalence underlies policy gradient and actor-critic methods. The proof below is given for finite horizon models. Discussion of extensions to infinite horizon models follows.

Theorem 11.2. Let N be finite and suppose $w(a|s; \beta)$ is differentiable with respect to each component of β . Then

$$\nabla_{\beta} v(\beta) = E^{(d_{\beta})^{\infty}} \left[\sum_{j=1}^N \nabla_{\beta} \ln(w(Y_j|X_j; \beta)) \left(\sum_{i=j}^N r(X_i, Y_i, X_{i+1}) \right) \right]. \quad (11.86)$$

The proof is an easy consequence of the following two results.

Lemma 11.1. Let a_i for $i = 1, \dots, N$ and b_j for $j = 1, \dots, N$ be real numbers. Then for N finite

$$\sum_{i=1}^N \left(\sum_{j=1}^i b_j \right) a_i = \sum_{j=1}^N b_j \left(\sum_{i=j}^N a_i \right). \quad (11.87)$$

This lemma and its proof for $N = \infty$ is Theorem 8.3 in [Rudin 1964]. The following provides a representation for the expected value of a Markov decision process that receives a reward $r(s, a, j)$ only at the end of period n .

Lemma 11.2. Consider an n -period Markov decision process in which the reward function satisfies^a

$$r_i(s, a, j) = \begin{cases} 0 & i < n \\ r(s, a, j) & i = n. \end{cases}$$

Let d_β denote a randomized decision rule that chooses actions in each decision epoch according to $w(a|s; \beta)$ and let $v_n(\beta)$ denote the expected total reward of $(d_\beta)^\infty$ averaged over initial state distribution $\rho(\cdot)$.

Then if $w(a|s; \beta)$ is differentiable with respect to each component of β :

$$\nabla_\beta v_n(\beta) = E^{(d_\beta)^\infty} \left[\left(\sum_{j=1}^n \nabla_\beta \ln(w(Y_j|X_j; \beta)) \right) r(X_n, Y_n, X_{n+1}) \right]. \quad (11.88)$$

^aRecall that $r_i(s, a, j)$ denotes the reward in period i in a non-stationary Markov decision process.

Proof. From the definition of $v_n(\beta)$,

$$v_n(\beta) := \sum_{h \in H_n} P_\beta(h) r(s_n, a_n, s_{n+1}) = E^{(d_\beta)^\infty} [r(X_n, Y_n, X_{n+1})], \quad (11.89)$$

where H_n denotes the set of all histories of the form $h = (s_1, a_1, \dots, s_n, a_n, s_{n+1})$ and $P_\beta(h)$ denotes the probability of history h where actions are chosen in each decision epoch according to $w(a|s; \beta)$. By the Markov property,

$$P_\beta(h) = \rho(s_1) w(a_1|s_1; \beta) p(s_2|s_1, a_1) \cdots w(a_n|s_n; \beta) p(s_{n+1}|s_n, a_n). \quad (11.90)$$

Since $w(a|s; \beta)$ is differentiable with respect to each component of β , using (11.51) establishes that

$$\nabla_\beta P_\beta(h) = P_\beta(h) \nabla_\beta \ln(P_\beta(h)). \quad (11.91)$$

Therefore,

$$\nabla_\beta v_n(\beta) = \sum_{h \in H_n} P_\beta(h) \nabla_\beta \ln(P_\beta(h)) r(s_n, a_n, s_{n+1}). \quad (11.92)$$

From (11.90)

$$\begin{aligned}\ln(P_{\beta}(h)) &= \ln(\rho(s_1)) + \ln(w(a_1|s_1; \beta)) + \ln(p(s_2|s_1, a_1)) \\ &\quad + \cdots + \ln(w(a_n|s_n; \beta)) + \ln(p(s_{n+1}|s_n, a_n)).\end{aligned}$$

Since $\rho(s)$ and $p(j|s, a)$ do not depend on β ,

$$\nabla_{\beta} \ln(P_{\beta}(h)) = \sum_{j=1}^n \nabla_{\beta} \ln(w(a_j|s_j; \beta)). \quad (11.93)$$

Hence, combining (11.92) and (11.93) gives

$$\nabla_{\beta} v_n(\beta) = \sum_{h \in H_n} P_{\beta}(h) \left(\sum_{j=1}^n \nabla_{\beta} \ln(w(a_j|s_j; \beta)) \right) r(s_n, a_n, s_{n+1}). \quad (11.94)$$

Rewriting this in expectation form gives (11.88). \square

Proof of Theorem 11.2. It is easy to see that

$$\nabla_{\beta} v(\beta) = \sum_{n=1}^N \nabla_{\beta} v_n(\beta),$$

so that substituting the representation in (11.88) into the above result and applying Lemma 11.1 with $b_j = \ln w(Y_j|X_j; \beta)$ and $a_i = r(X_i, Y_i, X_{i+1})$, gives (11.86). \square

Extensions*

The proof above establishes the policy gradient theorem for finite N . The same proof applies for a random stopping time N that satisfies $P(N < \infty) = 1$; the result follows by applying the same proof to each sample path. Regarding an infinite horizon model as a model with random geometric stopping times allows extension to the infinite horizon case.

To directly analyze infinite horizon discounted models requires additional assumptions. Specifically Lemma 11.1 must apply with $N = \infty$. For this to hold, the series in Lemma 11.1 must converge absolutely. Moreover such an analysis requires the validity of interchanging the order of the gradient, expectation and infinite summation. The conditions $\lambda < 1$ and bounded rewards are sufficient for both requirements.

Bibliographic remarks

This chapter has merely touched the surface of the vast literature on reinforcement learning with function approximation, including value-based, policy-based, and actor-critic methods.

Function approximation has played a central role in scaling reinforcement learning to large or continuous state-action spaces. The seminal work of Sutton [1988] introduced temporal-difference learning with linear approximators. Tsitsiklis and Roy [1997] provided a rigorous theoretical foundation for TD methods, establishing convergence results that inspired much subsequent research. The instability of off-policy learning, notably highlighted in Baird [1995], remains a core challenge. Stable alternatives such as least squares TD and policy iteration methods were proposed by Lagoudakis and Parr [2003] for linear approximations.

Policy-based methods trace back to the REINFORCE³⁷ algorithm of Williams [1992], which appears here as Algorithm 11.11. Sutton et al. [1999] later proved its convergence. These methods enable direct optimization in continuous action spaces and have evolved into robust frameworks. Notable extensions include Trust Region Policy Optimization (TRPO) Schulman et al. [2015] and Proximal Policy Optimization (PPO) Schulman et al. [2017], which improved sample efficiency and training stability.

The actor-critic framework was foreshadowed in the adaptive elements of Widrow et al. [1973], and formally developed in Barto et al. [1983], which shows how a system with “*two neuron-like elements can solve a difficult learning control problem*”. A survey of policy gradient methods is provided by Lehmann [2024].

The use of nonlinear function approximators, especially neural networks, gained momentum with the success of deep reinforcement learning. The Deep Q-Network (DQN) introduced by Mnih et al. [2015] represented a landmark, combining Q-learning with convolutional networks and stabilizing techniques such as experience replay and target networks.

Monte Carlo Tree Search (MCTS), used in AlphaGo and related systems, is closely related to the adaptive multistage sampling algorithm of Chang et al. [2005]. An accessible overview is provided in Fu [2018]. The game Wordle, used in this text to illustrate MCTS, was created by Wardle [2021].

Several books offer accessible and insightful introductions to the field, including Sutton and Barto [2018] and Kochenderfer et al. [2022] from a computer science perspective. Comprehensive theoretical treatments are found in Bertsekas and Tsitsiklis [1996], Szepesvari [2010], and Bertsekas [2019]. Historical perspectives on the evolution of reinforcement learning appear in Sutton and Barto [2018], Bertsekas and Tsitsiklis [1996], and Gosavi [2015]. Moreover, excellent lecture notes are available online; we especially recommend those by Levine [2024] on policy gradients and actor-critic methods, and by Katselis [2019] on Q-learning.

Exercises

You are encouraged to replicate the computational results in this chapter and as well to try out all methods herein on problems of personal or research interest. Chapter 3

³⁷REINFORCE is an acronym for the expression “REward Increment = Non-negative Factor \times Offset Reinforcement \times Characteristic Eligibility”.

provides a wide range of problems to analyze. In particular, the advanced appointment scheduling problem in Section 9.6 offers an excellent vehicle for testing these methods. Analyzing it with neural networks offers considerable research potential.

1. Specify first-visit and every-visit Monte Carlo approximation algorithms for estimating the value of a policy in an episodic model. Apply these algorithms to the shortest path Gridworld model in Section 11.3.3.
2. **Optimal stopping in a random walk (Example 11.1).**
 - (a) Repeat the analysis in the example for other choices of \bar{S} and K .
 - (b) Obtain a linear value approximation (i.e., polynomial or piecewise-polynomial) for π_2 and π_3 when $p = 0.45, 0.48, 0.5, 0.52, 0.55$. Compare your analysis to that in the text.
 - (c) Find suitable policy value function approximations when a linear fit is inadequate.
3. (a) Describe a Monte Carlo policy value function approximation algorithm for a discounted MDP based on geometric stopping. Recall that in this case, you accumulate the *undiscounted* total reward up to a geometric stopping time. This is in contrast to the version based on truncation, which accumulated the *discounted* reward.

 (b) Use this algorithm to approximate the policy value function in the two instances described in Example 11.2.
4. Fit a cubic spline with two knots to the queuing control model in and use Algorithm 11.4 to estimate its parameters. Why may it be problematic to use splines in Q-learning?
5. Provide explicit expressions for features when $f_i(s)$ and $h_j(a)$ are polynomials in s and a , respectively.
6. **Tabular models and function approximation.** Show that when features are indicators of state:
 - (a) Algorithm 11.5 reduces to the tabular TD(γ) Algorithm 10.6,
 - (b) Algorithm 11.6 reduces to the tabular Q-learning Algorithm 10.11,
 - (c) Describe a tabular policy gradient algorithm.
7. **Queuing service rate control experiments.** Consider the discounted queuing service rate control model in Example 11.4. Use wRMSE and policy structure to guide interpretation of results.

- (a) Conduct an in-depth study of the model using Q-learning that varies model parameters, discount rate, learning and exploration parameters and data generation methods.
- (b) Repeat your analysis using policy gradient and actor-critic algorithms.
8. **Gridworld experiments.** Consider the episodic shortest path model in Section 11.3.3.
- (a) Verify the conclusions in Section 11.3.3 by developing and implementing your own codes.
 - (b) Repeat the calculations using a neural network and other parametric state-action value function approximations.
 - (c) Implement a SARSA variant of Q-learning Algorithm 11.6.
 - (d) Solve the model using the Q-policy iteration Algorithm 11.7.
9. **State aggregation in optimal stopping.** Apply policy gradient and actor-critic to the optimal stopping model as follows. Let $N = 60$ and compare approximations that aggregate consecutive states into groups of $M = 1, 2$ or 4 . For example when $M = 4$ the state space is partitioned into K groups of states of the form $G_1 := \{1, 2, 3, 4\}, G_2 := \{5, 6, 7, 8\}, \dots, G_{15} = \{57, 58, 59, 60\}$. In this case features can be written as
- $$b_{k,j}(s, a) = \begin{cases} 1 & s \in G_k, a = a_j \\ 0 & \text{otherwise} \end{cases}$$
- for $k = 1, \dots, 15$ and $j = 1, 2$. These features can be represented as 30-component vectors with the first 15 components corresponding to a_1 and the next 15 components corresponding to a_2 .
10. Derive the expression in (11.52).
11. Consider the shortest path through a grid analyzed in Section 11.6.2.
- (a) Replicate the analysis in the text using function approximations for both the actor and critic. See Section 11.3.3 for guidance on specification of basis functions.
 - (b) Repeat the analysis assuming the cost per step $c(m, n)$ in cell (m, n) depends on the row m and column n . For example, suppose $c(m, n) = -0.1 - 0.1n$. How do you think modifying the costs in this way affects policy choice. (Note we found actor-critic and policy gradient frequently diverged or converged to sub-optimal policies for this modification.)

12. **Comparing methods for evaluating the policy gradient.** Consider a two-state model with $S = \{1, \Delta\}$ where Δ is a zero-reward absorbing state and the system starts in state 1 so that $\rho(1) = 1$. There are two actions to choose from in state 1, a_1 and a_2 . For $i = 1, 2$, under action a_i , $p(1|1, a_i) = p_i$, $r(1, a_i, 1) = r_i$, and $r(1, a_i, \Delta) = 0$. Suppose the parameter vector is a scalar β and

$$w(a_1|\beta) = \frac{e^\beta}{e^\beta + 1}, \quad w(a_2|\beta) = \frac{1}{e^\beta + 1}.$$

Compute $v_1(\beta)$ as a function of β . Then evaluate the gradient of $v_1(\beta)$ in the following two ways and compare results:

- (a) Using the representation in (11.61).
- (b) Using the policy gradient theorem result expressed as (11.63).

Each case requires enumerating trajectories and computing the probability of each as a function of β .

13. **Expectation of a baseline in the Policy Gradient Theorem.** Prove that if the baseline $B(s)$ is independent of the action a , the policy gradient satisfies

$$E^{(d_\beta)^\infty} \left[\sum_{j=1}^N \nabla_\beta \ln(w(Y_j|X_j; \beta)) B(X_j) \right] = 0. \quad (11.95)$$

Consequently, subtracting a baseline in the policy gradient representation does not change its expectation but can reduce its variance.

14. **Partially observable models and reinforcement learning:** Consider the partially observable model in Example 3.

- (a) Apply a policy gradient algorithm to a version of this problem in which policies are formulated as functions of *the observation* rather than belief states. Note that this is an episodic model that terminates when choosing a door.
- (b) Develop and apply a policy gradient algorithm in which the policies are functions of belief states as in Chapter 8. Use it to find a good policy. This requires developing an approach for updating belief states and representing a continuous state variable.
- (c) Compare the form, the information used, and the quality of policies obtained using both approaches. Which would you prefer?

15. **Importance sampling.** Perform the following experiment for the setting in Example 11.7. Simulate 100 replicates of samples of length five using decision rule γ and decision rule δ . Compute the mean and standard deviation of the sample means from γ and from the estimates of γ obtained by using importance sampling based on realizations of δ . What conclusions can be drawn?

16. **Monte Carlo tree search.** Apply MCTS to the Gridworld model in Figure 11.16.
17. **Comparing gradient and semi-gradient TD updates.** Consider a policy in an episodic Markov decision process with two non-terminal states, s_1 and s_2 , and one terminal state, s_3 . Under this policy, transitions are deterministic and the only reward is obtained upon entering the terminal state. That is:

- $s_1 \rightarrow s_2$ with reward $r = 0$
- $s_2 \rightarrow s_3$ with reward $r = 1$
- s_3 is terminal

The value function is approximated using a linear function approximator:

$$v(s; \boldsymbol{\beta}) = \beta_1 x_1(s) + \beta_2 x_2(s)$$

with features defined as:

$$b(s_1) = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \quad b(s_2) = \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \quad b(s_3) = \begin{bmatrix} 0 \\ 0 \end{bmatrix}.$$

For this example, do the following:

- (a) For a single transition $s \rightarrow s'$ with reward r , define the squared temporal difference error:

$$g(\boldsymbol{\beta}) = (r + \lambda v(s'; \boldsymbol{\beta}) - v(s; \boldsymbol{\beta}))^2.$$

Derive the full gradient $\nabla_{\boldsymbol{\beta}} g(\boldsymbol{\beta})$ and the semi-gradient approximation used in TD learning.

- (b) For $\boldsymbol{\beta} = (0, 0)$, compute the following for the transition $s_1 \rightarrow s_2$:

- $v(s_1; \boldsymbol{\beta})$, $v(s_2; \boldsymbol{\beta})$, the TD target, and TD error
- The full gradient and semi-gradient update directions
- The updated $\boldsymbol{\beta}$ after one gradient step with learning rate $\alpha = 0.1$

- (c) Repeat part (b) for the transition $s_2 \rightarrow s_3$ with reward $r = 1$.

- (d) Compare the results of full gradient and semi-gradient updates. Which update more effectively reduces the TD error? How does the difference between $b(s)$ and $b(s')$ affect this?

- (e) Modify the example so that transitions are random rather than deterministic and repeat your analysis.

- (e) Implement both the full gradient and semi-gradient TD updates in a programming language of your choice. Simulate several transitions and observe the learning behavior. Plot the squared TD error over time for both methods.