Chapter 11

Simulation with Function Approximation

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 corrected[1].

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 simulation[2] (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.

[1] Feynman et al. [1963].

[2] As 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.

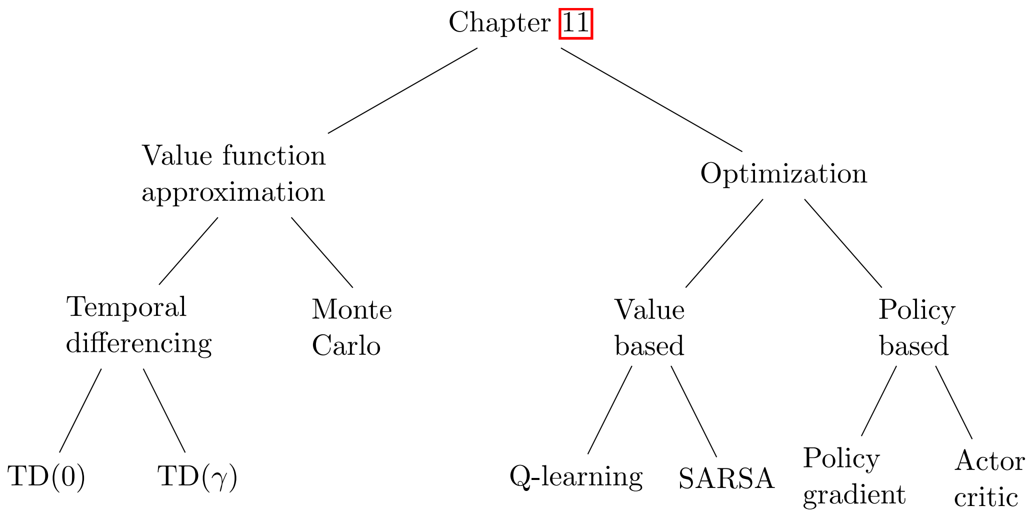


Figure 11.1: Schematic representation of methods described in Chapter 11

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 networks[3].

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 approximation[4] for a fixed policy, which was introduced in Chapter 9. For ease of exposition and to encompass many practical

[3]Of course, neural networks are nonlinear functions but they are distinguished here because they have a vast and specialized array of methods for parameter estimation.

[4]The 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

$$v(s; \beta_0, \dots, \beta_I) := \beta_0 b_0(s) + \beta_1 b_1(s) + \dots + \beta_I b_I(s), \tag{11.1}$$

where $\beta_0, \dots, \beta_I$ 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:

$$v(s; \boldsymbol{\beta}) = \boldsymbol{\beta}^{\mathsf{T}} \mathbf{b}(s), \tag{11.2}$$

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

$$b_{i,j}(s, a) = f_i(s) h_j(a), \tag{11.3}$$

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

$$q(s, a; \boldsymbol{\beta}) = \sum_{i=1}^{I} \sum_{j=1}^{J} \beta_{i,j} b_{i,j}(s, a) = \sum_{i=1}^{I} \sum_{j=1}^{J} \beta_{i,j} f_i(s) h_j(a), \tag{11.4}$$

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

$$h_j(a) = I_{\{a_j\}}(a) = \begin{cases} 1 & a = a_j \\ 0 & a \neq a_j. \end{cases} \tag{11.5}$$

In this case, $J = |A_s|$. Consequently,

$$b_{i,j}(s, a) = \begin{cases} f_i(s) & a = a_j \\ 0 & a \neq a_j \end{cases} \tag{11.6}$$

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

$$\begin{aligned} q(s, a; \boldsymbol{\beta}) &= \beta_{0,1} b_0(s, a_1) + \beta_{0,2} b_0(s, a_2) + \beta_{1,1} b_1(s, a_1) + \beta_{1,2} b_1(s, a_2) \\ &= \beta_{0,1} f_0(s) h_1(a) + \beta_{0,2} f_0(s) h_2(a) + \beta_{1,1} f_1(s) h_1(a) + \beta_{1,2} f_1(s) h_2(a) \\ &= \beta_{0,1} I_{\{a_1\}}(a) + \beta_{0,2} I_{\{a_2\}}(a) + \beta_{1,1} s I_{\{a_1\}}(a) + \beta_{1,2} s I_{\{a_2\}}(a). \end{aligned}$$

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

$$q(s, a; \boldsymbol{\beta}) = \boldsymbol{\beta}^{\mathsf{T}} \mathbf{b}(s, a), \tag{11.7}$$

[5]Recall 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 $\boldsymbol{\beta}$ 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 $\boldsymbol{\beta}$ can be written as

$$\boldsymbol{\beta} = \begin{bmatrix} \boldsymbol{\beta}_{a_1} \\ \vdots \\ \boldsymbol{\beta}_{a_J} \end{bmatrix}, \tag{11.8}$$

where the column vector $\boldsymbol{\beta}_{a_j}$ 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

$$\boldsymbol{\beta} = \begin{bmatrix} \beta_{0,1} \\ \beta_{1,1} \\ \beta_{0,2} \\ \beta_{1,2} \end{bmatrix} = \begin{bmatrix} \boldsymbol{\beta}_{a_1} \\ \boldsymbol{\beta}_{a_2} \end{bmatrix}, \quad \mathbf{b}(s,a_1) = \begin{bmatrix} 1 \\ s \\ 0 \\ 0 \end{bmatrix} \quad \text{and} \quad \mathbf{b}(s,a_2) = \begin{bmatrix} 0 \\ 0 \\ 1 \\ s \end{bmatrix}. \tag{11.9}$$

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

$$q(s,a;\boldsymbol{\beta}) = \beta_{0,0} + \beta_{1,0}s + \beta_{0,1}a + \beta_{1,1}as.$$

where $\boldsymbol{\beta} = (\beta_{0,0}, \beta_{1,0}, \beta_{0,1}, \beta_{1,1})$.

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 $q(\mathbf{s},\mathbf{a};\boldsymbol{\beta})$ 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

$$\hat{v}_d(s) = \begin{cases} \boldsymbol{\beta}^\mathsf{T}\mathbf{b}(s,d(s)) & \text{if } d \text{ is deterministic} \\ \sum_{a \in A_s} w_d(a|s)\boldsymbol{\beta}^\mathsf{T}\mathbf{b}(s,a) & \text{if } d \text{ is randomized.} \end{cases}$$

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 $\boldsymbol{\beta}$ corresponds to a specific decision rule. That is, instead of explicitly encoding the policy, one can use $\boldsymbol{\beta}$ 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 $\boldsymbol{\beta}_0$.
2. Specify a state $s \in S$.
3. Compute $q(s, a; \boldsymbol{\beta}_0) \leftarrow \boldsymbol{\beta}_0^\mathsf{T}\mathbf{b}(s, a)$ for all $a \in A_s$.
4. (a) Deterministic action selection: Set $a_s \in \arg\max_{a \in A_s} q(s, a; \boldsymbol{\beta}_0)$.
   (b) Randomized action selection Sample $a_s$ using $\epsilon$-greedy or softmax
       sampling based on $q(s, a; \boldsymbol{\beta}_0)$.
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 $\boldsymbol{\beta}_0$. 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 $\boldsymbol{\beta}$ corresponds to a specific policy. In the special case when the model is known and rewards are independent of the subsequent state, choosing $\boldsymbol{\beta}_0 = \mathbf{0}$ would result in setting $q(s, a; \boldsymbol{\beta}_0) = r(s, a)$ 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 $\boldsymbol{\beta}$. A convenient representation, referred to as a softmax[6] or logistic transformation, is given by

$$w(a|s;\boldsymbol{\beta}) = \frac{e^{\boldsymbol{\beta}^{\mathsf{T}}\mathbf{b}(s,a)}}{\sum_{a' \in A_s} e^{\boldsymbol{\beta}^{\mathsf{T}}\mathbf{b}(s,a')}}. \tag{11.10}$$

An alternative to using (11.10) is to represent $w(a|s;\boldsymbol{\beta})$ by a *neural network* where $\boldsymbol{\beta}$ 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.

[6] Previously, 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 $\boldsymbol{\beta}$.

```
    (b) Specify a subset of starting states $\bar{S} \subseteq S \setminus \Delta$.
    (c) For all $s \in \bar{S}$, create an empty list VALUES($s$).
2. **Generate values:** For all $\bar{s} \in \bar{S}$:
    (a) $k \leftarrow 1$.
    (b) While $k \le K$:
        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 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 $\hat{\boldsymbol{\beta}} = (\hat{\beta}_0, \ldots, \hat{\beta}_I)$ using least squares or weighted least squares.
4. **Terminate:** Return $\hat{\boldsymbol{\beta}}$.
```

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 $\hat{\boldsymbol{\beta}}$ 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 $(s, \text{mean}(\textsc{Values}(s)))$ 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 $S = \{1, \ldots, 500\}$, 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

$$v(s; \beta_0, \beta_1, \ldots, \beta_I) = \beta_0 + \beta_1 s + \beta_2 s^2 + \ldots + \beta_I s^I$$

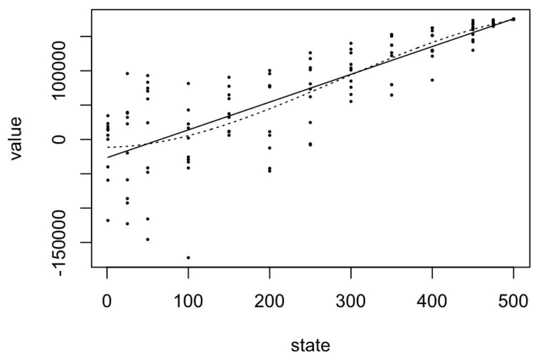
for small values of I and considers three policies:

π_1: Stop in all states, that is $\Delta = S$.

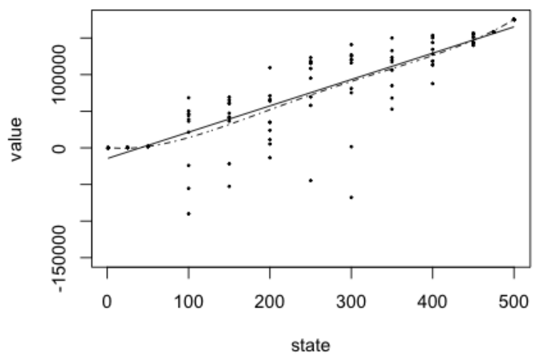
π_2: Stop only when in state 500, that is $\Delta = \{500\}$.

π_3: Stop in states $\{1, \ldots, 50\}$ and $\{475, \ldots, 500\}$, that is $\Delta = \{1, \ldots, 50\} \cup \{475, \ldots, 500\}$.

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 $\bar{S} = \{1, 25, 50, 100, 150, \ldots, 400, 450, 475, 500\}$ and $K = 10$ replications for each $s \in \bar{S}$[a].



(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

$$v^{\pi_3}(s, \hat{\beta}_0, \hat{\beta}_1, \ldots, \hat{\beta}_4) = 647.3 - 123.43s + 3.41s^2 - 0.009s^3 + 0.000009s^4.$$

It is left as an exercise to investigate other choices for $\bar{S}$ and parameter values.

[a]The 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 $d \in D^{\mathrm{MR}}$.
   (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 $\textsc{Values}(s)$.
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 $\textsc{Values}(\bar{s})$.
       iv. $k \leftarrow k+1$.
3. Estimate parameters: Use data $\{(s,v) \,|\, s \in \bar{S},\, v \in \textsc{Values}(s)\}$ to estimate parameters $\hat{\boldsymbol{\beta}} = (\hat{\beta}_0, \ldots, \hat{\beta}_I)$ using least squares or weighted least squares.
4. Terminate: Return $\hat{\boldsymbol{\beta}}$.
```

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 $S = \{0, \ldots, 50\}$. 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 $s \in \bar{S} = \{0, 5, 10, \ldots, 45, 50\}$. Cumulative discounted rewards are truncated at $M = 600$.

Outcomes were compared on the basis of the RMSE of the fit:

$$\text{RMSE} = \left(\frac{1}{|S|} \sum_{s \in S} (v(s, \hat{\boldsymbol{\beta}}) - v_\lambda^{d^\infty}(s))^2 \right)^{\frac{1}{2}}. \tag{11.11}$$

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.

	d_1			d_2		
λ	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

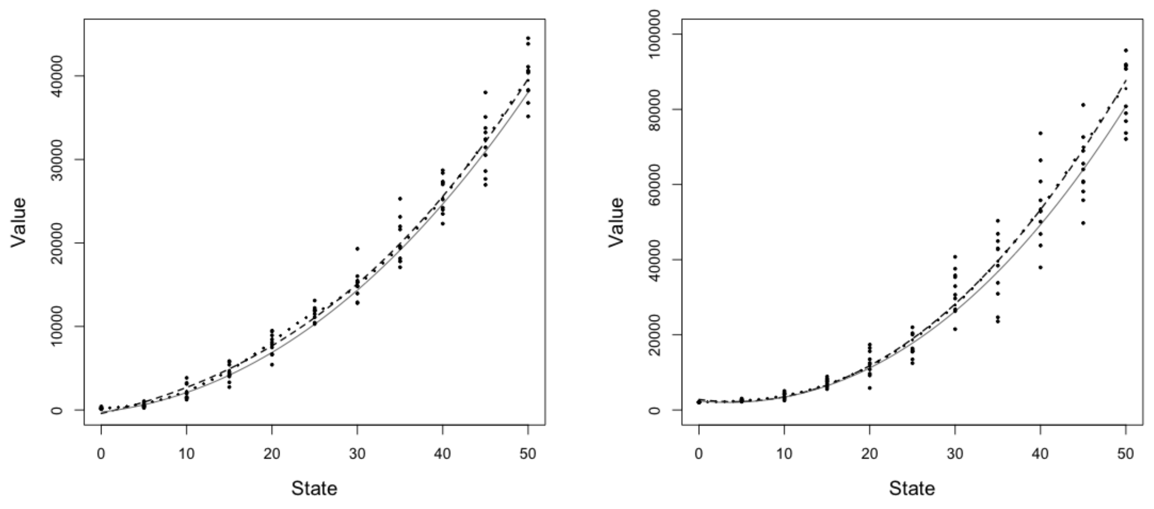
As an alternative to Monte Carlo estimation in which estimates are derived after simulating (or observing) many trajectories, this section describes an online approach that generalizes TD(0) by updating parameter estimates after each state transition[7].

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

[7]As noted above, when the features are indicators of states or states and actions, this is equivalent to the online methods in Chapter 10.



(a) Decision rule d_1 with $\lambda = 0.95$. (b) Decision rule d_2 with $\lambda = 0.98$.

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.

$\boldsymbol{\beta} \in \mathbb{R}^I$ so as to "solve" the policy evaluation (Bellman) equation

$$v(s;\boldsymbol{\beta}) = E^{d^\infty}\left[r(X,Y,X') + \lambda v(X';\boldsymbol{\beta})\middle|X = s\right]$$

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 $\boldsymbol{\beta}$ that minimizes the conditional expected squared error loss

$$E^{d^\infty}\left[(r(X,Y,X') + \lambda v(X';\boldsymbol{\beta}) - v(X;\boldsymbol{\beta}))^2\middle|X = s\right] \tag{11.12}$$

for all $s \in S$.

Since a different $\boldsymbol{\beta}$ may achieve the minimum above for each $s \in S$, an alternative is to choose $\boldsymbol{\beta}$ to minimize the *unconditional* squared error loss:

$$G_d(\boldsymbol{\beta}) := E^{d^\infty}\left[(r(X,Y,X') + \lambda v(X';\boldsymbol{\beta}) - v(X;\boldsymbol{\beta}))^2\right], \tag{11.13}$$

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

$$\boldsymbol{\beta}' = \boldsymbol{\beta} + \tau (u - v(s; \boldsymbol{\beta})) \nabla_{\boldsymbol{\beta}} v(s; \boldsymbol{\beta}), \tag{11.14}$$

where the factor 2 is absorbed into τ.

When $v(s; \boldsymbol{\beta})$ is a linear function of $\boldsymbol{\beta}$, $v(s; \boldsymbol{\beta}) = \boldsymbol{\beta}^{\mathsf{T}} \mathbf{b}(s)$ and $\nabla_{\boldsymbol{\beta}} v(s; \boldsymbol{\beta}) = \mathbf{b}(s)$, in which case the gradient descent recursion becomes

$$\boldsymbol{\beta}' = \boldsymbol{\beta} + \tau \big(u - \boldsymbol{\beta}^{\mathsf{T}} \mathbf{b}(s)\big) \mathbf{b}(s). \tag{11.15}$$

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:

$$\boxed{\boldsymbol{\beta}' = \boldsymbol{\beta} + \tau \, \delta \, \mathbf{b}(s). \qquad (11.16)}$$

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

$$g(\boldsymbol{\beta}) = \big(r + v(s'; \boldsymbol{\beta}) - v(s; \boldsymbol{\beta})\big)^2$$

using gradient descent, the gradient of $g(\boldsymbol{\beta})$ is given by

$$\nabla_{\boldsymbol{\beta}} g(\boldsymbol{\beta}) = 2\big(r + v(s'; \boldsymbol{\beta}) - v(s; \boldsymbol{\beta})\big)\big(\nabla_{\boldsymbol{\beta}} v(s'; \boldsymbol{\beta}) - \nabla_{\boldsymbol{\beta}} v(s; \boldsymbol{\beta})\big). \tag{11.17}$$

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

$$\nabla_{\boldsymbol{\beta}} g(\boldsymbol{\beta}) \approx -2\big(r + v(s'; \boldsymbol{\beta}) - v(s; \boldsymbol{\beta})\big)\nabla_{\boldsymbol{\beta}} v(s; \boldsymbol{\beta}). \tag{11.18}$$

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 diverge[8].

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.

[8]See 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 $d \in D^{\mathrm{MR}}$, $\boldsymbol{\beta}$ arbitrary, and $s \in S$.
      (b) Specify the number of iterations $N$ and set $n \leftarrow 1$.
      (c) Specify the learning rate sequence $\tau_n, n = 1, 2, \ldots$.
  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 $v(s; \boldsymbol{\beta}) \leftarrow \boldsymbol{\beta}^{\mathsf{T}}\mathbf{b}(s)$,
          $v(s'; \boldsymbol{\beta}) \leftarrow \boldsymbol{\beta}^{\mathsf{T}}\mathbf{b}(s')$ and
              $\delta \leftarrow r + \lambda v(s'; \boldsymbol{\beta}) - v(s; \boldsymbol{\beta}).$     (11.19)
      (d) Update $\boldsymbol{\beta}$:
              $\boldsymbol{\beta} \leftarrow \boldsymbol{\beta} + \tau_n \delta \mathbf{b}(s).$     (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 $\epsilon$ small and a state $s_0$. With probability $\epsilon$, $s \leftarrow s_0$
          and with probability $1 - \epsilon$, $s \leftarrow s'$.
      (f) $n \leftarrow n + 1$.
  3. Terminate: Return $\boldsymbol{\beta}$.
```

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 $q(s, a; \boldsymbol{\beta})$.

Actor-critic[9] samples a' from $w(\cdot|s; \boldsymbol{\beta}^C)$ where the estimate of the weight $\boldsymbol{\beta}$, denoted by $\boldsymbol{\beta}^C$, is updated using gradient ascent.

2. **Convergence:** When $v(s; \boldsymbol{\beta})$ is a linear function of $\boldsymbol{\beta}$ and the features are linearly independent, the iterates of $\boldsymbol{\beta}$ converge[10] with probability 1 provided the samples follow the trajectory of the Markov chain corresponding to d and $\tau_n, n = 1, 2, \ldots$, 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 $\boldsymbol{\beta}$. 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

$$\left(\frac{1}{I} \sum_{i=1}^{I} (\beta_i' - \beta_i)^2 \right)^{\frac{1}{2}} \tag{11.21}$$

or changes in fitted values:

$$\left(\frac{1}{|S|} \sum_{s \in S} (v(s; \boldsymbol{\beta}') - v(s; \boldsymbol{\beta}))^2 \right)^{\frac{1}{2}}. \tag{11.22}$$

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.

[9]See Section 11.6 below.

[10]See 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:

$$d_1(s) = \begin{cases} a_1 & 0 \leq s \leq 25 \\ a_3 & 26 \leq s \leq 50 \end{cases}$$
$$d_2(s) = \quad a_2 \quad 0 \leq s \leq 50$$
$$d_3(s) = \begin{cases} a_2 & 0 \leq s \leq 9 \\ a_3 & 10 \leq s \leq 29 \\ a_1 & 30 \leq s \leq 50 \end{cases}$$

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:

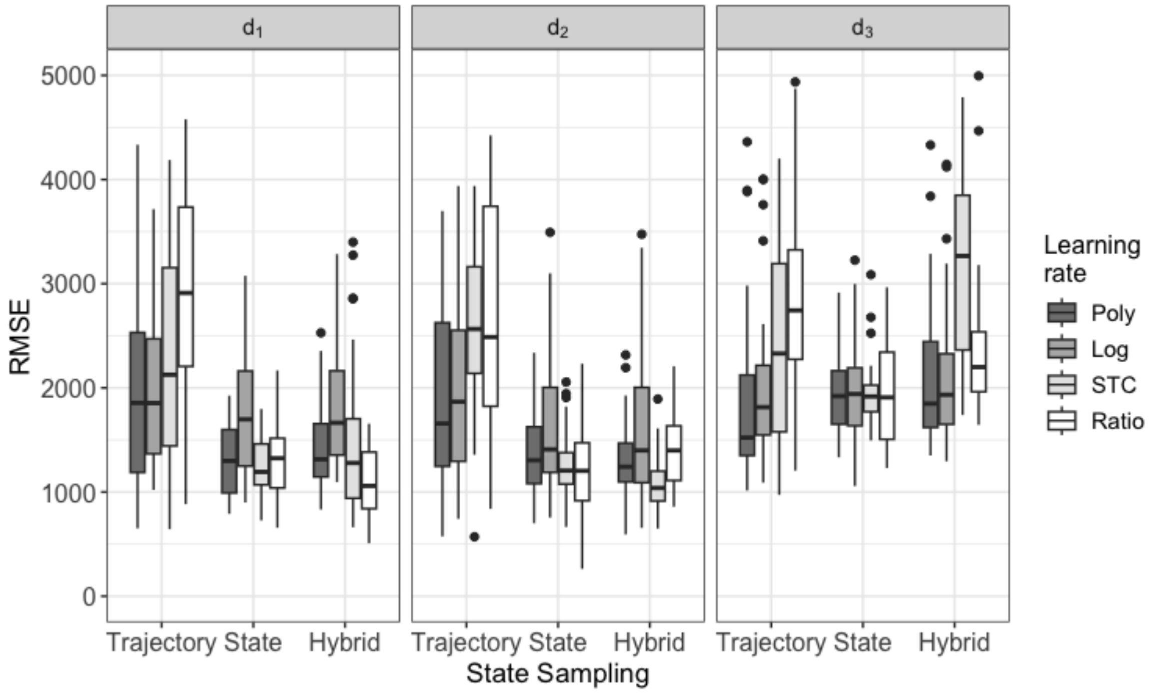


Figure 11.4: Comparison of the RMSEs of policy value function estimates for the queuing control model. To put results in perspective, Table 11.1 showed that a cubic polynomial approximation derived from Monte Carlo estimates had an average RMSE of 1,029.04 for d_1 and 1,098.57 for d_2.

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

	Replicate				
Method	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 $d \in D^{\mathrm{MR}}$, $\boldsymbol{\beta}$ 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 $\tau_n, n = 1, 2, \ldots$.
2. Iterate: While $n \le N$:
    (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 $v(s; \boldsymbol{\beta}) \leftarrow \boldsymbol{\beta}^{\mathsf{T}} \mathbf{b}(s)$,
        $v(s'; \boldsymbol{\beta}) \leftarrow \boldsymbol{\beta}^{\mathsf{T}} \mathbf{b}(s')$ and
            $\delta \leftarrow r + \lambda v(s'; \boldsymbol{\beta}) - v(s; \boldsymbol{\beta}).$    (11.23)
    (d) Update eligibility trace:
            $\mathbf{z} \leftarrow \gamma \lambda \mathbf{z} + \mathbf{b}(s).$    (11.24)
    (e) Update $\boldsymbol{\beta}$:
            $\boldsymbol{\beta} \leftarrow \boldsymbol{\beta} + \tau_n \delta \mathbf{z}.$    (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 $\epsilon$ small and a state $s_0$. With probability $\epsilon$, $s \leftarrow s_0$
        and with probability $1 - \epsilon$, $s \leftarrow s'$.
    (g) $n \leftarrow n + 1$.
3. Terminate: Return $\boldsymbol{\beta}$.
```

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 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 TD(γ) 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 TD(γ) with $\gamma > 0$ instead of 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 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 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 TD(γ) 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.

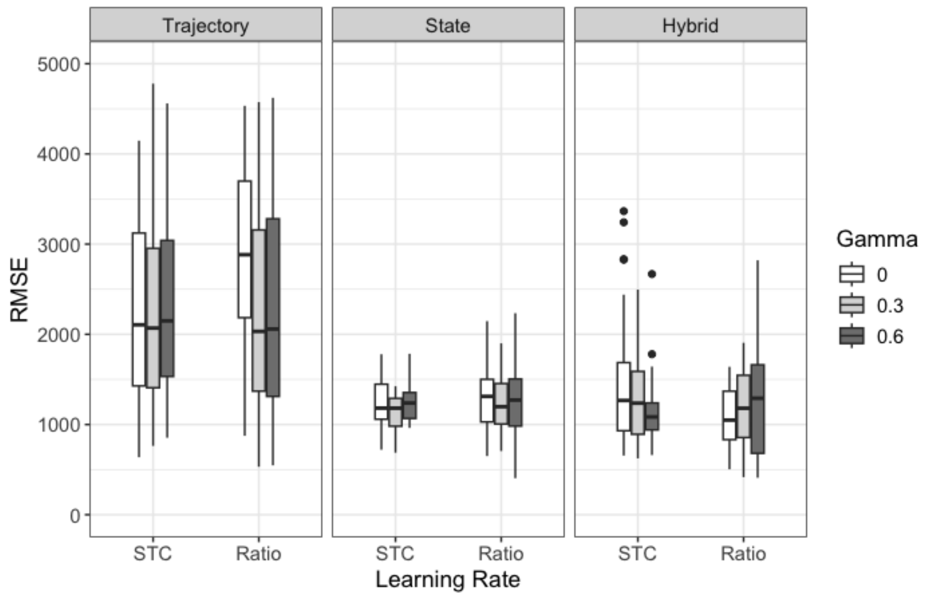


Figure 11.5: Boxplots of the RMSEs of the TD(γ) estimates based on 40 replicates with two learning rates and three state generation methods for the model in Example 11.3.

From this example, it appears that the effect of using TD(γ) 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

$$q(s,a) = \sum_{j \in S} p(j|s,a)\big(r(s,a,j) + \lambda \max_{a \in A_s} q(j,a)\big)$$

and in expectation form as

$$q(s,a) = E\left[r(X,Y,X') + \lambda \max_{a' \in A_{X'}} q(X',a') \,\middle|\, X = s, Y = a \right]. \tag{11.26}$$

Recall that its unique solution satisfies

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

or in expectation form as

$$q^*(s,a) = E\Big[r(X,Y,X') + \lambda v^*(X') \Big| X = s, Y = a \Big], \tag{11.27}$$

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 $q(s,a;\boldsymbol{\beta})$ in (11.26) to obtain

$$q(s,a;\boldsymbol{\beta}) = E\left[r(X,Y,X') + \lambda \max_{a' \in A_{X'}} q(X',a';\boldsymbol{\beta}) \,\middle|\, X = s, Y = a \right]. \tag{11.28}$$

As in the case of a tabular model, Q-learning seeks to find a $\boldsymbol{\beta}$ that minimizes the discrepancy between the two-sides of (11.28) in the squared error sense as

$$E\left[\left(r(X,Y,X') + \lambda \max_{a' \in A_X} q(X',a';\boldsymbol{\beta}) - q(X,Y;\boldsymbol{\beta}) \right)^2 \,\middle|\, X = s, Y = a \right], \tag{11.29}$$

where the expectation is with respect to the distribution $p(\cdot|s,a)$.

The semi-gradient[11] of this expression with respect to $\boldsymbol{\beta}$ (see (11.18)) is written as

$$-2E\left[\left(r(s,a,X') + \lambda \max_{a' \in A_{X'}} q(X',a';\boldsymbol{\beta}) - q(s,a;\boldsymbol{\beta}) \right) \nabla_{\boldsymbol{\beta}} q(s,a;\boldsymbol{\beta}) \right].$$

[11]The full gradient would include an additional term corresponding to the gradient of $\lambda \max_{a' \in A_{X'}} q(X',a';\boldsymbol{\beta})$.

When $q(s, a)$ is approximated by a linear function of features $\mathbf{b}(s, a)$, the above expression becomes

$$-2E\left[\left(r(s, a, X') + \lambda \max_{a' \in A_{X'}} \boldsymbol{\beta}^{\mathsf{T}}\mathbf{b}(X', a') - \boldsymbol{\beta}^{\mathsf{T}}\mathbf{b}(s, a)\right)\mathbf{b}(s, a)\right].$$

This expression leads to the following recursion for estimating $\boldsymbol{\beta}$:

$$\boldsymbol{\beta} \leftarrow \boldsymbol{\beta} + \tau\left(r(s, a, s') + \lambda \max_{a \in A_{s'}} \boldsymbol{\beta}^{\mathsf{T}}\mathbf{b}(s', a') - \boldsymbol{\beta}^{\mathsf{T}}\mathbf{b}(s, a)\right)\mathbf{b}(s, a), \tag{11.30}$$

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

$$\hat{a}_s \in \arg\max_{a \in A_s} q(s, a; \hat{\boldsymbol{\beta}}) \tag{11.31}$$

or randomly using softmax sampling based on $q(s, a; \hat{\boldsymbol{\beta}})$. Moreover, the corresponding optimal value function approximation can be obtained from

$$\hat{v}(s) = \max_{a \in A_s} q(s, a; \hat{\boldsymbol{\beta}}). \tag{11.32}$$

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 $\tau_n, n = 1, 2, \ldots$.
   (c) Specify the number of iterations N.
   (d) Specify a sequence $\epsilon_n, n = 1, 2, \ldots$ (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 $\boldsymbol{\beta}$ 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 $a \in A_{s'}$

$$q(s', a; \boldsymbol{\beta}) \leftarrow \boldsymbol{\beta}^{\mathsf{T}} \mathbf{b}(s', a). \tag{11.33}$$

(d) Set

$$\delta \leftarrow r + \lambda \max_{a' \in A_{s'}} q(s', a'; \boldsymbol{\beta}) - q(s, a; \boldsymbol{\beta}). \tag{11.34}$$

(e) **Update $\boldsymbol{\beta}$:**

$$\boldsymbol{\beta} \leftarrow \boldsymbol{\beta} + \tau_n \mathbf{b}(s, a)\delta. \tag{11.35}$$

(f) $s \leftarrow s'$.

(g) $n \leftarrow n + 1$.

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

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 $\boldsymbol{\beta}$ changes, the elusive policy may change. The policy iteration algorithms below avoid this by fixing $\boldsymbol{\beta}$ 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 function[12] 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 distribution[13] 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 $S = \{0, \ldots, 50\}$. 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,

[12] This could be computed by solving policy evaluation equations or approximated by simulation.

[13] Recall that these can be found by solving $\mathbf{w}^\mathsf{T} = \mathbf{w}^\mathsf{T}\mathbf{P}$ subject to $\mathbf{w}^\mathsf{T}\mathbf{e} = 1$ 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 $n = 1, 2, \ldots$, 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.

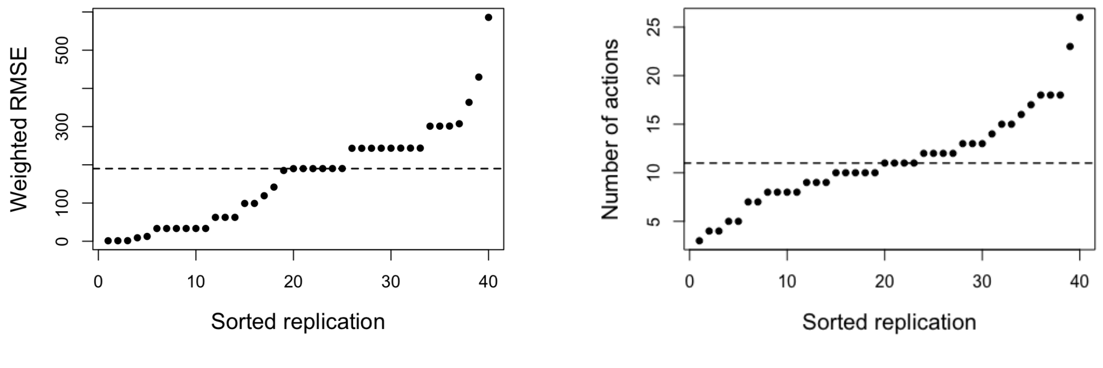
Figure 11.6a shows the sorted wRMSE values in ascending order. It shows the wRMSE was less than 300 in all but three replicates.

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.



(a) Sorted wRMSEs. (b) Sorted non-optimal actions.

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: $S = \{(m, n) \mid m = 1, \ldots, M, n = 1, \ldots, N\}$

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:

$$r\big((m,n), a, (m',n')\big) = \begin{cases} -c & (m',n') \neq (m^*,n^*),\ a \in A \\ R - c & (m,n) \neq (m^*,n^*),\ (m',n') = (m^*,n^*),\ a \in A \\ 0 & (m,n) = (m^*,n^*), a = \delta \end{cases}$$

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 enumerate[14], 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., $i = 1, \ldots, MN$). In code, it might be more convenient to index $f_i(m',n')$ by the row and column index.

[14]For 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:**

$$f_i(m',n') = \begin{cases} I_{\{m\}}(m',n') & \text{for } m = 1,\ldots,M \text{ and } i = 1,\ldots,M, \\ I_{\{n\}}(m',n') & \text{for } n = 1,\ldots,N \text{ and } i = M+1,\ldots,M+N. \end{cases}$$

   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, $q(s,a;\boldsymbol{\beta})$ 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

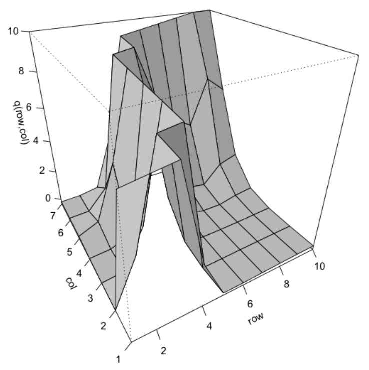
Calculations for the tabular model apply Algorithm 11.6 to two instances with $(m^*,n^*) = (M,N)$, $R = 10$ and $c = 0.1$. In this application the robot seeks to learn a path to cell (M,N) from the starting cell $(1,1)$.

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.

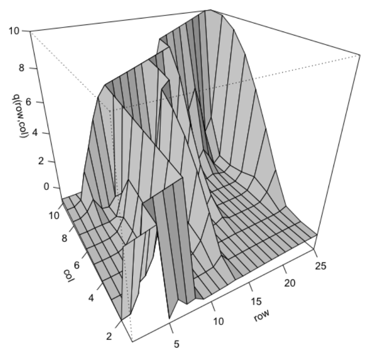
For small values of M and N, it was convenient to use the tabular representation of Q-learning to gain some insight into the form of the q-functions. For all replications of the experiment, this approach found an optimal path through the grid.

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.

[15] If one sought a good policy for every starting state, episodes could be restarted at a random non-target cell.



(a) 10×7 grid



(b) 25×10 grid

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. $\boldsymbol{\beta}$ 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 ($\tau_n = 50/(1{,}000 + n)$),

[16]The 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.

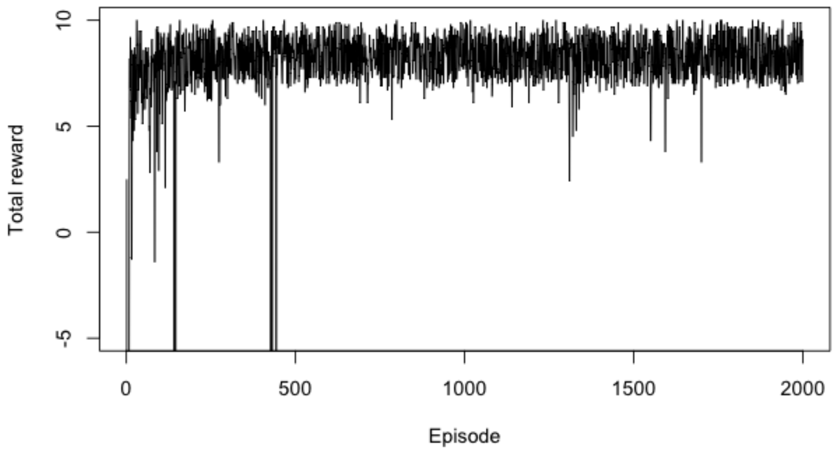


Figure 11.8: Reward per episode in a typical realization of Q-learning in a 10×25 grid. The ongoing variability of the total reward per episode is a result of the random starting state for each epsiode.

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 $\boldsymbol{\beta}$.
2. **Improve:** Find an "improved" policy using $q(s, a; \boldsymbol{\beta})$.
3. **Evaluate:** Find $\boldsymbol{\beta}'$ 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 $\boldsymbol{\beta}_0$, using $q(s, a; \boldsymbol{\beta}_0)$ 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 $q(s, a; \boldsymbol{\beta}_0)$ 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 $\boldsymbol{\beta}_0$.
   (b) Specify the learning rate sequence $\tau_n, n = 1, 2, \ldots$, and a sequence $\epsilon_n, n = 1, 2, \ldots$ (or $\eta_n, n = 1, 2, \ldots$, 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 $\boldsymbol{\beta}_0$: $m \leftarrow 1$ and $\boldsymbol{\beta} \leftarrow \boldsymbol{\beta}_0$.
    (b) Generate $a \in A_s$ randomly using Algorithm 11.1 applied to $\boldsymbol{\beta}_0$ 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
                $q(s, a; \boldsymbol{\beta}_0) \leftarrow \boldsymbol{\beta}_0^{\mathsf{T}} \mathbf{b}(s, a).$   (11.36)
        iii. Generate $a' \in A_{s'}$ randomly using Algorithm 11.1 applied to $\boldsymbol{\beta}_0$ in state $s'$.
        iv. Set
                $q(s', a'; \boldsymbol{\beta}_0) \leftarrow \boldsymbol{\beta}_0^{\mathsf{T}} \mathbf{b}(s', a').$   (11.37)
        v. Set
                $\delta \leftarrow r + \lambda q(s', a'; \boldsymbol{\beta}_0) - q(s, a; \boldsymbol{\beta}_0)$   (11.38)
        vi. Update $\boldsymbol{\beta}$:
                $\boldsymbol{\beta} \leftarrow \boldsymbol{\beta} + \tau_k \mathbf{b}(s, a)\delta.$   (11.39)
        vii. $s \leftarrow s'$, $a \leftarrow a'$.
        viii. $k \leftarrow k + 1$ and $m \leftarrow m + 1$.
    (d) Weight updating:
        i. $\boldsymbol{\beta}_0 \leftarrow \boldsymbol{\beta}$.
        ii. $n \leftarrow n + 1$.
3. Terminate: Return $\boldsymbol{\beta}_0$.
```

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 $\boldsymbol{\beta}_0$ with evaluation in step 2(b)vi. While the action-generating policy being evaluated remains the same, the estimate of its corresponding $\boldsymbol{\beta}$ is revised.

On the other hand, the algorithm may be viewed as a simulation-based version of modified policy iteration[17] (Algorithm 5.8) in which improvement and evaluation are

[17]Note 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 $q(s, a; \boldsymbol{\beta})$ as the basis for action selection, steps 2(b) and 2(c)iii choose actions using ϵ-greedy or softmax sampling based on $q(s, a; \boldsymbol{\beta}_0)$ 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 $\boldsymbol{\beta}_0$.

3. Steps 2(c)iii-2(c)v can be replaced by the two steps:

   (a) Set
   $$q(s', a; \boldsymbol{\beta}_0) \leftarrow \boldsymbol{\beta}_0^\mathsf{T} \mathbf{b}(s', a) \tag{11.40}$$
   for all $a \in A_{s'}$.

   (b) Set
   $$\delta \leftarrow r + \lambda \max_{a' \in A_{s'}} q(s', a'; \boldsymbol{\beta}_0) - q(s, a; \boldsymbol{\beta}_0). \tag{11.41}$$

   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 $\boldsymbol{\beta}_0$ 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 $\boldsymbol{\beta}_0$, 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 deviation[18], so that

$$q(a; \boldsymbol{\beta}) = \boldsymbol{\beta}^{\mathsf{T}}\mathbf{b}(a).$$

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 $r(a, z^1), \dots, r(a, z^N)$. Then use least squares to approximate $\hat{q}(a)$ by a linear function of its features to obtain $\hat{\boldsymbol{\beta}}_{\text{MC}}$. 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 $\tau_n, n = 1, 2, \dots$.

[18]This 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 $\epsilon$-greedy (or softmax) sampling with respect to
            $q(a; \boldsymbol{\beta}) = \boldsymbol{\beta}^{\mathsf{T}} \mathbf{b}(a).$
    (b) Generate demand $z^n$.
    (c) Set
            $\delta \leftarrow r(a', z^n) - \boldsymbol{\beta}^{\mathsf{T}} \mathbf{b}(a').$    (11.42)
    (d) **Update $\boldsymbol{\beta}$:**
            $\boldsymbol{\beta} \leftarrow \boldsymbol{\beta} + \tau_n \delta \mathbf{b}(a').$    (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 $\epsilon$-greedy (or softmax) sampling with respect to
                $q(a; \boldsymbol{\beta}_0) = \boldsymbol{\beta}_0^{\mathsf{T}} \mathbf{b}(a).$
        ii. Generate demand $z^m$.
        iii. Set
                $\delta \leftarrow r(a', z^m) - \boldsymbol{\beta}^{\mathsf{T}} \mathbf{b}(a')$    (11.44)
```

```
            iv. **Update $\boldsymbol{\beta}$:**
                $$\boldsymbol{\beta} \leftarrow \boldsymbol{\beta} + \tau_k \delta \mathbf{b}(a').$$
            v. $k \leftarrow k+1$, $m \leftarrow m+1$, and $a \leftarrow a'$.
        (c) $\boldsymbol{\beta}_0 \leftarrow \boldsymbol{\beta}$ and $n \leftarrow n+1$.
    3. **Terminate:** Return $\hat{\boldsymbol{\beta}}_{\text{QPI}} = \boldsymbol{\beta}_0$.
```

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 $\boldsymbol{\beta}$ at each iteration, which is updated in (11.43). In Q-policy iteration, a fixed $\boldsymbol{\beta}_0$ is used for action generation in step 2(b)i for M evaluation steps and a different $\boldsymbol{\beta}$ (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 $\boldsymbol{\beta}_0$.

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 $A_0 = \{0, 1, \ldots, 120\}$.

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 $\boldsymbol{\beta} = \mathbf{0}$, 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

$$a^* \in \arg\max_{a \in A_0} \hat{\boldsymbol{\beta}}^{\mathsf{T}} \mathbf{b}(a)$$

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.

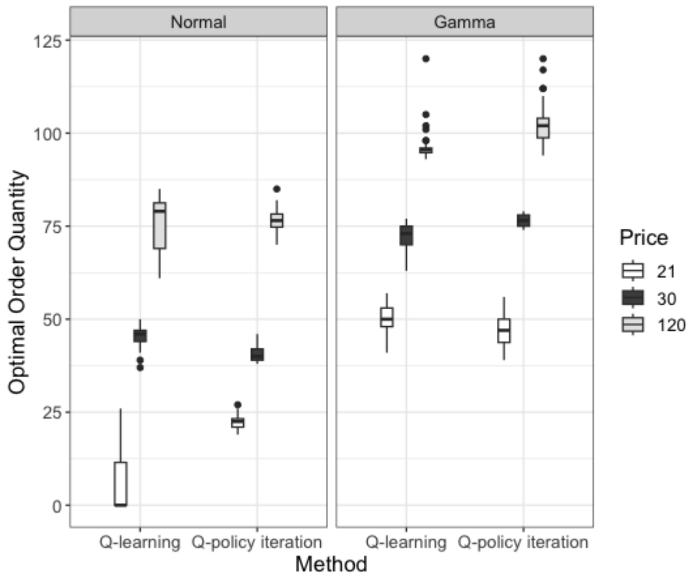


Figure 11.9: Boxplots comparing the actions chosen by Q-learning and Q-policy iteration for the newsvendor model over 40 replicates broken down by demand distribution and price.

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
Q-policy iteration	Non-optimal actions	5	9	10	11	20
	RMSE	3.5	12.9	19.9	25.3	139.0
$(M = 100, N = 2{,}500)$	wRMSE	33.5	105.3	157.9	208.2	585.9
Q-policy iteration	Non-optimal actions	3	6	7.5	10	21
	RMSE	1.7	12.2	16.8	22.5	169.9
$(M = 50, N = 5{,}000)$	wRMSE	12.6	141.7	150.6	212.7	587.0
	Non-optimal actions	5	9.8	11.5	13	40
SARSA	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	Non-optimal actions	3	6	7	9	18
variant	RMSE	1.7	8.3	12.6	22.1	126.7
$(M = 50,\ N = 5{,}000)$	wRMSE	12.6	98.9	141.7	229.9	585.9
	Non-optimal actions	6	8.8	12	13	17
Q-learning	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 replicates[19] 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.

[19] Results 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 function[20] by stochastic approximation in which gradient estimates are obtained by sampling from a distribution with a specified expectation.

[20]Of 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 vectors[21] $\boldsymbol{\beta}$. 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:

$$w(a|\boldsymbol{\beta}) = \frac{e^{\boldsymbol{\beta}^\mathsf{T}\mathbf{b}(a)}}{\sum_{a'\in A_s} e^{\boldsymbol{\beta}^\mathsf{T}\mathbf{b}(a')}} \tag{11.45}$$

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),

$$w(a|s;\boldsymbol{\beta}) = \frac{e^{\boldsymbol{\beta}^\mathsf{T}\mathbf{b}(s,a)}}{\sum_{a'\in A_s} e^{\boldsymbol{\beta}^\mathsf{T}\mathbf{b}(s,a')}}. \tag{11.46}$$

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

$$v^* := \max_{\boldsymbol{\beta}} v(\boldsymbol{\beta}) = \max_{\boldsymbol{\beta}} \sum_{a\in A_0} w(a|\boldsymbol{\beta})r(a) = \max_{\boldsymbol{\beta}} \sum_{a\in A_0} \frac{e^{\boldsymbol{\beta}^\mathsf{T}\mathbf{b}(a)}}{\sum_{a'\in A_s} e^{\boldsymbol{\beta}^\mathsf{T}\mathbf{b}(a')}} r(a). \tag{11.47}$$

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:

$$\nabla_{\boldsymbol{\beta}} v(\boldsymbol{\beta}) = \mathbf{0}. \tag{11.48}$$

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 $\{0,1,\ldots,M\}$.) 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 $f(\cdot)$.

[21]Many authors represent randomized decision rules by $\pi_{\boldsymbol{\phi}}(a|s)$ where $\boldsymbol{\phi}$ denotes a vector of parameters.

The gradient of $v(\boldsymbol{\beta})$ 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 $v(\boldsymbol{\beta})$. 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 $v(\boldsymbol{\beta})$ can be written as:

$$\begin{aligned}\nabla_{\boldsymbol{\beta}} v(\boldsymbol{\beta}) &= \sum_{a \in A_0} \nabla_{\boldsymbol{\beta}} w(a|\boldsymbol{\beta}) r(a) \\ &= \sum_{a \in A_0} w(a|\boldsymbol{\beta}) \nabla_{\boldsymbol{\beta}} \ln(w(a|\boldsymbol{\beta})) r(a) &(11.49) \\ &= E_{\boldsymbol{\beta}} \left[\nabla_{\boldsymbol{\beta}} \ln(w(Y|\boldsymbol{\beta})) r(Y) \right]. &(11.50)\end{aligned}$$

The expression in (11.49) follows by applying the calculus identity

$$\frac{dg(x)}{dx} = g(x) \frac{d \ln(g(x))}{dx} \tag{11.51}$$

to each component of $\boldsymbol{\beta}$.

The benefits of establishing this equivalence are that:

1. When using the softmax to represent $w(a|\boldsymbol{\beta})$, the gradient of $\ln(w(a|\boldsymbol{\beta}))$ has a nice closed-form representation.

2. As a result of the representation (11.50), the gradient of $v(\boldsymbol{\beta})$ can be estimated by sampling
$$\nabla_{\boldsymbol{\beta}} \ln(w(Y|\boldsymbol{\beta})) r(Y)$$
from $w(\cdot|\boldsymbol{\beta})$.

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 analyze[22].

It is left as an exercise to show that when $w(a|\boldsymbol{\beta})$ is defined by (11.45),

$$\nabla_{\boldsymbol{\beta}} \ln(w(a|\boldsymbol{\beta})) = \mathbf{b}(a) - \sum_{a' \in A_0} w(a'|\boldsymbol{\beta}) \mathbf{b}(a'). \tag{11.52}$$

Combining (11.49) with (11.52) leads to the following closed-form expression for the gradient of $v(\boldsymbol{\beta})$ in the newsvendor model:

[22]Note that in statistical maximum likelihood estimation one bases results on maximizing log-likelihoods instead of likelihoods.

$$\nabla_{\boldsymbol{\beta}} v(\boldsymbol{\beta}) = \sum_{a \in A_0} w(a|\boldsymbol{\beta}) \Big(\mathbf{b}(a) - \sum_{a' \in A_0} w(a'|\boldsymbol{\beta}) \mathbf{b}(a') \Big) r(a). \tag{11.53}$$

Online estimation of the gradient in the newsvendor model

This section applies stochastic gradient ascent[23] to solve $\nabla_{\boldsymbol{\beta}} v(\boldsymbol{\beta}) = \mathbf{0}$ for the newsvendor model. This method involves three steps:

1. Sample action a from $w(\cdot|\boldsymbol{\beta})$,
2. sample the demand from $f(\cdot)$ to estimate $r(a) = E[r(a, Z)]$, and
3. estimate the gradient by $\nabla_{\boldsymbol{\beta}} \ln(w(a|\boldsymbol{\beta}))r(a)$.

Thus, the gradient ascent recursion becomes

$$\boldsymbol{\beta}' \leftarrow \boldsymbol{\beta} + \tau \nabla_{\boldsymbol{\beta}} \ln(w(a|\boldsymbol{\beta}))r(a), \tag{11.54}$$

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 $\boldsymbol{\beta}$ and a learning rate sequence $\tau_n, n = 1, 2, \ldots$.
   (b) Specify the number of iterations N and set $n \leftarrow 1$.
2. **Iterate:** While $n \leq N$:
   (a) Sample a from $w(\cdot|\boldsymbol{\beta})$.
   (b) Generate demand z^n and set $r \leftarrow r(a, z^n)$.
   (c) **Estimate the gradient:**
$$\nabla_{\boldsymbol{\beta}} v(\boldsymbol{\beta}) = \Big(\mathbf{b}(a) - \sum_{a' \in A_0} w(a'|\boldsymbol{\beta}) \mathbf{b}(a') \Big) r. \tag{11.55}$$
   (d) **Update $\boldsymbol{\beta}$:**
$$\boldsymbol{\beta} \leftarrow \boldsymbol{\beta} + \tau_n \nabla_{\boldsymbol{\beta}} v(\boldsymbol{\beta}). \tag{11.56}$$

[23] "Ascent" is used since this is a maximization problem.

```
      (e) $n \leftarrow n + 1$.
  3. **Terminate:** Return $\boldsymbol{\beta}$.
```

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 $w(\cdot|\boldsymbol{\beta})$.
3. The policy gradient algorithm converges to a global maximum under mild conditions on $r(a)$.
4. The algorithm terminates with an estimate of $\boldsymbol{\beta}$ and the corresponding stationary randomized policy $w(a|\boldsymbol{\beta})$.
5. Alternatively to stopping after a specified number of iterations, the algorithm can terminate when the change in $\boldsymbol{\beta}$ 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 in to the workings of gradient ascent as the following calculations show.

For concreteness, assume that $a \in \{0, 1, 2, 3\}$ and $\boldsymbol{\beta} = (0, 0, 0, 0)$, so that defining $w(a|\boldsymbol{\beta})$ by (11.45) gives $w(a|\boldsymbol{\beta}) = 0.25$ for $a \in \{0, 1, 2, 3\}$. Suppose sampling generates action $a = 1$ and $r(a) = 17$. Then applying (11.55) shows that

$$\nabla_{\boldsymbol{\beta}} v(\boldsymbol{\beta}) = 17 \left(\begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} - \begin{bmatrix} 0.25 \\ 0.25 \\ 0.25 \\ 0.25 \end{bmatrix} \right) = 17 \begin{bmatrix} -0.25 \\ 0.75 \\ -0.25 \\ -0.25 \end{bmatrix}.$$

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 $\boldsymbol{\beta}$ corresponding to $a = 1$ will increase and all other components will decrease. Hence the policy $w(a|\boldsymbol{\beta}')$, where $\boldsymbol{\beta}'$ corresponds to the updated parameter vector, will select $a = 1$ with greater probability and all other actions with smaller probability than $w(a|\boldsymbol{\beta})$. 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 $0, \ldots, a_{\max}$.

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 $w(a|\boldsymbol{\beta})$ evaluated at the estimated $\boldsymbol{\beta}$ 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 $\arg\max_{a \in A_0} w(a|\boldsymbol{\beta})$, 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 $w(a|\hat{\boldsymbol{\beta}})$ over a where $\hat{\boldsymbol{\beta}}$ 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

$$v^\pi = \sum_{s\in S} \rho(s) v^\pi(s). \tag{11.57}$$

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 $d^\infty \in D^{\mathrm{MR}}$ 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 $\boldsymbol{\beta}$ and let $w(a|s;\boldsymbol{\beta})$ denote the corresponding parameterized action-choice probability distribution. Let $d_{\boldsymbol{\beta}}$ represent the decision rule that chooses actions according $w(a|s;\boldsymbol{\beta})$. That is $w_{d_{\boldsymbol{\beta}}}(a|s) := w(a|s;\boldsymbol{\beta})$.

Hence, the (scalar) objective becomes that of finding a weight vector $\boldsymbol{\beta}$ that maximizes

$$v(\boldsymbol{\beta}) := \sum_{s\in S} \rho(s) v^{(d_{\boldsymbol{\beta}})^\infty}(s) \tag{11.58}$$

or equivalently, a

$$\hat{\boldsymbol{\beta}} \in \arg\max_{\boldsymbol{\beta}} v(\boldsymbol{\beta}). \tag{11.59}$$

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 $v(\boldsymbol{\beta})$ in terms of transition probabilities, action choice probabilities and rewards. It shows that

$$v(\boldsymbol{\beta}) = \sum_{s^1\in S}\sum_{a^1\in A_{s^1}}\sum_{s^2\in S}\rho(s^1)w(a^1|s^1;\boldsymbol{\beta})p(s^2|s^1,a^1)\Bigg(r(s^1,a^1,s^2) + \sum_{a^2\in A_{s^2}}\sum_{s^3\in S}w(a^2|s^2;\boldsymbol{\beta})p(s^3|s^2,a^2)\big(r(s^2,a^2,s^3)+\dots\big)\Bigg) \tag{11.60}$$

where eventually the rewards are zero after reaching an absorbing state in Δ.

Now consider the expected reward in the first period only

$$v_1(\boldsymbol{\beta}) := \sum_{s^1\in S}\sum_{a^1\in A_{s^1}}\sum_{s^2\in S}\rho(s^1)w(a^1|s^1;\boldsymbol{\beta})p(s^2|s^1,a^1)r(s^1,a^1,s^2).$$

In this case,

$$\nabla_{\boldsymbol{\beta}}v_1(\boldsymbol{\beta}) = \sum_{s^1\in S}\sum_{a^1\in A_{s^1}}\sum_{s^2\in S}\nabla_{\boldsymbol{\beta}}\big(\rho(s^1)w(a^1|s^1;\boldsymbol{\beta})p(s^2|s^1,a^1)r(s^1,a^1,s^2)\big). \tag{11.61}$$

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

$$\nabla_{\boldsymbol{\beta}}v_1(\boldsymbol{\beta}) = \sum_{s^1\in S}\sum_{a^1\in A_{s^1}}\sum_{s^2\in S}\rho(s^1)w(a^1|s^1;\boldsymbol{\beta})p(s^2|s^1,a^1)r(s^1,a^1,s^2) \times \nabla_{\boldsymbol{\beta}}\ln\big(\rho(s^1)w(a^1|s^1;\boldsymbol{\beta})p(s^2|s^1,a^1)r(s^1,a^1,s^2)\big). \tag{11.62}$$

Since $\rho(s)$, $p(s^2|s^1,a^1)$ and $r(s^1,a^1,s^2)$ do not involve $\boldsymbol{\beta}$,

$$\begin{aligned}\nabla_{\boldsymbol{\beta}}\ln\big(\rho(s^1)w(a^1|s^1;\boldsymbol{\beta})p(s^2|s^1,a^1)r(s^1,a^1,s^2)\big) &= \nabla_{\boldsymbol{\beta}}\ln\big(\rho(s)\big)+\nabla_{\boldsymbol{\beta}}\ln\big(w(a^1|s^1;\boldsymbol{\beta})\big)+\nabla_{\boldsymbol{\beta}}\ln\big(p(s^2|s^1,a^1)\big)+\nabla_{\boldsymbol{\beta}}\ln\big(r(s^1,a^1,s^2)\big)\\ &= \nabla_{\boldsymbol{\beta}}\ln\big(w(a^1|s^1;\boldsymbol{\beta})\big).\end{aligned}$$

Therefore, it follows from (11.62) that

$$\boxed{\nabla_{\boldsymbol{\beta}}v_1(\boldsymbol{\beta}) = \sum_{s^1\in S}\sum_{a^1\in A_{s^1}}\sum_{s^2\in S}\rho(s^1)w(a^1|s^1;\boldsymbol{\beta})p(s^2|s^1,a^1)r(s^1,a^1,s^2)\nabla_{\boldsymbol{\beta}}\ln\big(w(a^1|s^1;\boldsymbol{\beta})\big).} \tag{11.63}$$

The significance of this expression is that it shows that $\nabla_{\boldsymbol{\beta}} v_1(\boldsymbol{\beta})$ can be evaluated by averaging $\nabla_{\boldsymbol{\beta}} \ln\big(w(a^1|s^1;\boldsymbol{\beta})\big)r(s^1,a^1,s^2)$ 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 $w(a|s;\boldsymbol{\beta})$ is differentiable with respect to each component of $\boldsymbol{\beta}$. Then

$$\nabla_{\boldsymbol{\beta}} v(\boldsymbol{\beta}) = E^{(d_{\boldsymbol{\beta}})^\infty}\left[\sum_{j=1}^{N_\Delta} \nabla_{\boldsymbol{\beta}} \ln\big(w(Y_j|X_j;\boldsymbol{\beta})\big)\left(\sum_{i=j}^{N_\Delta} r(X_i,Y_i,X_{i+1})\right)\right]. \qquad (11.64)$$

Observe that the multiplier of the gradient $\nabla_{\boldsymbol{\beta}} \ln\big(w(Y_j|X_j;\boldsymbol{\beta})\big)$ 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 $\nabla_{\boldsymbol{\beta}} \ln(w(Y_j|X_j;\boldsymbol{\beta}))$ 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 $\boldsymbol{\beta}$ for decision rule $d_{\boldsymbol{\beta}}$.

11.5.4 A simple policy gradient algorithm

The following algorithm, frequently referred to as *the policy gradient algorithm*, describes an early[24] 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 $\boldsymbol{\beta}$, a learning rate sequence $\tau_n, n = 1,2,\ldots$, 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 $k \le K$:

[24]This 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 $w(a|s;\boldsymbol{\beta}^A)$ to
        generate actions. Save the sequence of states, actions and rewards
            $(s^1, a^1, s^2, r^1, \ldots, s^N, a^N, s^{N+1}, r^N),$
        where $N$ denotes the termination epoch of the episode.
    (b) $n \leftarrow 1$.
    (c) While $n \leq N$:
        i. Update weights sequentially:
            $\boldsymbol{\beta} \leftarrow \boldsymbol{\beta} + \tau_k \nabla_{\boldsymbol{\beta}} \ln\big(w(a^n|s^n;\boldsymbol{\beta})\big)\left(\sum_{i=n}^{N} r^i - B(s^n)\right).$   (11.65)
        ii. $n \leftarrow n+1$.
    (d) $k \leftarrow k+1$.
3. Terminate: Return $\boldsymbol{\beta}$.
```

This algorithm may be viewed as a simulated policy improvement algorithm. Starting with a policy defined by $\boldsymbol{\beta}$, 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 update[25].

To implement this batch policy gradient step, replace the sequential update 2(c) by

$$\boldsymbol{\beta} \leftarrow \boldsymbol{\beta} + \tau_k \sum_{n=1}^{N} \nabla_{\boldsymbol{\beta}} \ln\big(w(a^n|s^n;\boldsymbol{\beta})\big)\left(\sum_{i=n}^{N} r^i - B(s^n)\right). \tag{11.66}$$

Some further comments follow:

1. After generating an entire episode using $w(a|s;\boldsymbol{\beta})$, step 2(c) updates $\boldsymbol{\beta}$ by stepping backwards through the tail of cumulative rewards (adjusted by the baseline).

2. 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

[25] See 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 $\boldsymbol{\beta}$ 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 $w(a|s;\boldsymbol{\beta})$ is the softmax function. Then
$$\frac{\partial}{\partial \beta_{s,a}} \ln\big(w(a^n|s^n;\boldsymbol{\beta})\big) = \begin{cases} 1 - w(a|s;\boldsymbol{\beta}) & \text{for } s = s^n,\ a = a^n \\ -w(a|s;\boldsymbol{\beta}) & \text{for } s = s^n,\ a \neq a^n \\ 0 & \text{for } s \neq s^n. \end{cases}$$

   Consequently, the only positive component of $\nabla_{\boldsymbol{\beta}} \ln\big(w(a|s^n;\boldsymbol{\beta})\big)$ 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 $\nabla_{\boldsymbol{\beta}} \ln\big(w(a^n|s^n;\boldsymbol{\beta})\big)$ can be evaluated numerically, in closed form when $w(a|s;\boldsymbol{\beta})$ is a softmax function of features, or by back-propagation when $w(a|s;\boldsymbol{\beta})$ 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 available[26].

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.

[26]For 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 shortest[27] 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 $\boldsymbol{\beta}$ generates an episode with two continuation actions followed by stopping at the third step. For concreteness, set the observed sequence to $5 \to 6 \to 7 \to \Delta$ 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 $w(a|s;\boldsymbol{\beta})$ with $\eta = 1$ becomes

$$w(C|s;\boldsymbol{\beta}) = \frac{e^{\beta_{0,0}+\beta_{1,0}s}}{e^{\beta_{0,0}+\beta_{1,0}s} + e^{\beta_{0,1}+\beta_{1,1}s}}$$

and $w(Q|s;\boldsymbol{\beta}) = 1 - w(C|s;\boldsymbol{\beta})$. Some simple calculus establishes that the gradients are given by

$$\nabla_{\boldsymbol{\beta}} \ln\big(w(C|s;\boldsymbol{\beta})\big) = \begin{bmatrix} w(Q|s;\boldsymbol{\beta}) \\ -w(Q|s;\boldsymbol{\beta}) \\ sw(Q|s;\boldsymbol{\beta}) \\ -sw(Q|s;\boldsymbol{\beta}) \end{bmatrix} = w(Q|s;\boldsymbol{\beta}) \begin{bmatrix} 1 \\ -1 \\ s \\ -s \end{bmatrix}$$

[27]Since rewards are maximized, this is actually a *longest* path model.

and

$$\nabla_{\boldsymbol{\beta}} \ln\big(w(Q|s;\boldsymbol{\beta})\big) = w(C|s;\boldsymbol{\beta}) \begin{bmatrix} -1 \\ 1 \\ -s \\ s \end{bmatrix}$$

where $\boldsymbol{\beta} = (\beta_{0,0}, \beta_{0,1}, \beta_{1,0}, \beta_{1,1})$. Note that in examples such as this it is more convenient to represent the components of $\boldsymbol{\beta}$ 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

$$\boldsymbol{\beta} \leftarrow \boldsymbol{\beta} + \tau_n \left(\sum_{i=n}^{N} r^i\right) w(Q|s;\boldsymbol{\beta}) \begin{bmatrix} 1 \\ -1 \\ s \\ -s \end{bmatrix}. \tag{11.67}$$

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 $\boldsymbol{\beta}$. Standardizing rewards can reduce this variability.

Suppose that $\boldsymbol{\beta} = (0,0,0,0)$ and $\tau_n = 0.1$, then the iterates (numbered in the order they are computed) become

$$\boldsymbol{\beta}^1 = 0.1 \cdot 49 \cdot 0.5 \begin{bmatrix} 1 \\ -1 \\ 7 \\ -7 \end{bmatrix} = \begin{bmatrix} 2.45 \\ -2.45 \\ 17.15 \\ -17.15 \end{bmatrix}, \ \boldsymbol{\beta}^2 = \begin{bmatrix} 2.25 \\ -2.25 \\ 11.05 \\ -11.05 \end{bmatrix} \text{ and } \boldsymbol{\beta}^3 = \begin{bmatrix} 2.25 \\ -2.25 \\ 11.05 \\ -11.05 \end{bmatrix}.$$

Note that corresponding to the final iterate $\boldsymbol{\beta}^3$, the action choice probabilities in state 5 are $w(Q|5;\boldsymbol{\beta}^3) = 1$ and $w(C|5;\boldsymbol{\beta}^3) = 0$ and in state 0, $w(Q|0;\boldsymbol{\beta}^3) = 0.99$ and $w(C|0;\boldsymbol{\beta}^3) = 0.01$.

Alternatively, setting $B = 47$, which equals the mean of the σ^n, results in the sequence

$$\boldsymbol{\beta}^1 = \begin{bmatrix} -0.1 \\ 0.1 \\ -0.7 \\ 0.7 \end{bmatrix}, \ \boldsymbol{\beta}^2 = \begin{bmatrix} -0.1 \\ 0.1 \\ -0.7 \\ 0.7 \end{bmatrix} \text{ and } \boldsymbol{\beta}^3 = \begin{bmatrix} -0.3 \\ -0.3 \\ -1.7 \\ 1.7 \end{bmatrix}.$$

Although the $\boldsymbol{\beta}$ estimates have changed and have become smaller in magnitude, the corresponding-action selection probabilities in state 5 remain $w(Q|5;\boldsymbol{\beta}^3) = 1$ and $w(C|5;\boldsymbol{\beta}^3) = 0$, and in state 0, $w(Q|0;\boldsymbol{\beta}^3) = 0.65$ and $w(C|0;\boldsymbol{\beta}^3) = 0.35$. Thus, estimates of $\boldsymbol{\beta}$ 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 $v(\boldsymbol{\beta})$ 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 $S = \{-50, \ldots, 50\}$ 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

$$w(a'|s'; \boldsymbol{\beta}) = \frac{e^{\eta \beta_{s',a'}}}{\sum_{s=-50}^{50} (e^{\eta \beta_{s,C}} + e^{\eta \beta_{s,Q}})}, \tag{11.68}$$

where $\beta_{s',a'}$ is the coefficient corresponding to $b(s', a')$.

For this model, the gradient is available in closed form as:

$$\frac{\partial}{\partial \beta_{s,a}} \ln w(a'|s'; \boldsymbol{\beta}) = \begin{cases} \eta(1 - w(a|s'; \boldsymbol{\beta})) & (s, a) = (s', a') \\ -\eta w(a|s'; \boldsymbol{\beta}) & (s, a) = (s', a) \text{ and } a \neq a' \\ 0 & \text{otherwise.} \end{cases} \tag{11.69}$$

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:

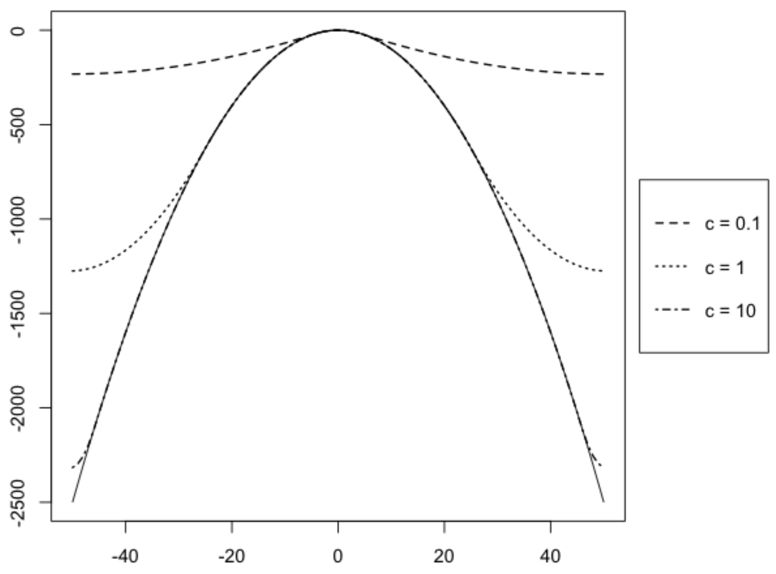


Figure 11.10: Optimal value functions (dashed lines) for three choices of c for the problem of optimally stopping in a random walk with $p = 0.5$ The solid line shows the reward of stopping in each state. The optimal policy is to stop when the optimal value function and the reward from stopping agree. Observe that the stopping region increases with respect to the continuation cost.

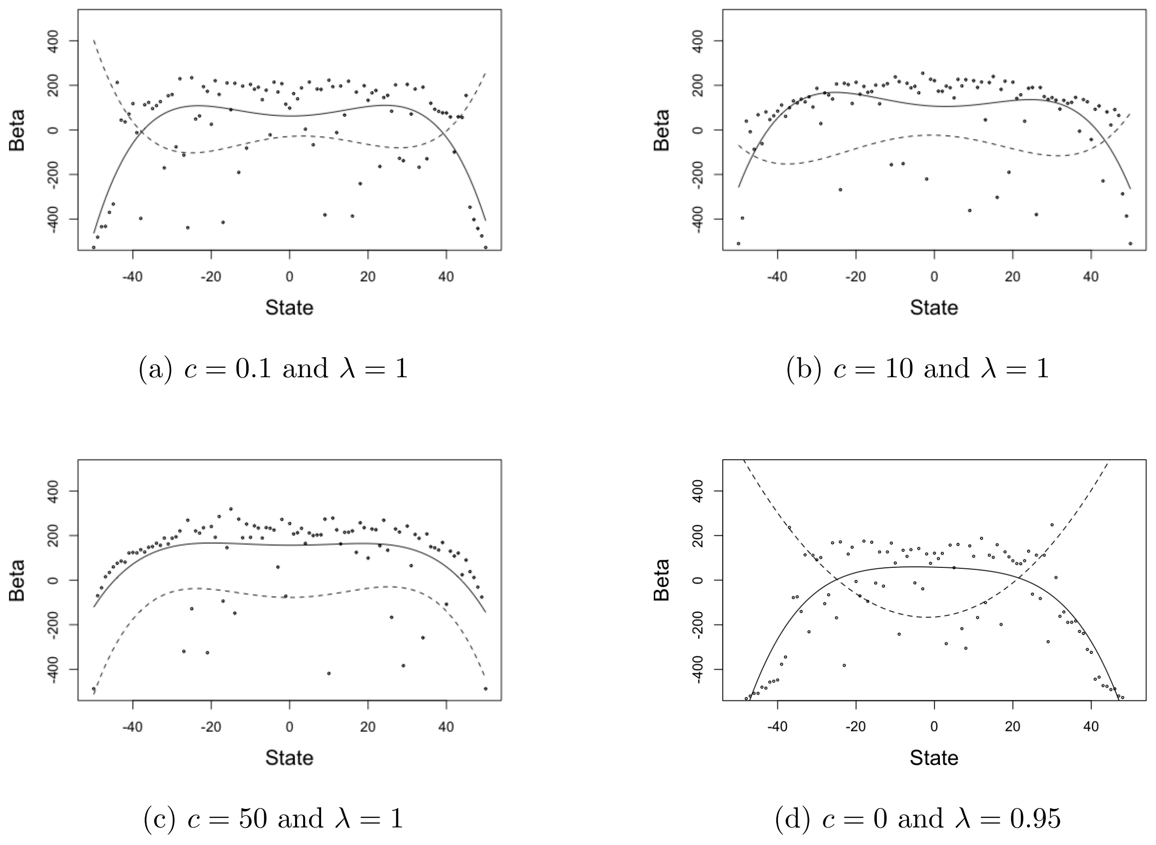
- Uniform initial distribution on S;
- $c \in \{0.1, 10, 50\}$;
- $p = 0.5$;
- $\boldsymbol{\beta}$ initialized to $\mathbf{0}$;
- $K = 500{,}000$;
- episodes were truncated after 100 steps;
- softmax exponent $\eta = 1/100$;
- learning rate, $\tau_k = 1{,}000/(10{,}000 + k)$, where k represented the 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 $\boldsymbol{\beta}$ 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 $\boldsymbol{\beta}$ 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.



(a) $c = 0.1$ and $\lambda = 1$

(b) $c = 10$ and $\lambda = 1$

(c) $c = 50$ and $\lambda = 1$

(d) $c = 0$ and $\lambda = 0.95$

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 stop[28]. 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.

[28]Alternatively, 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

$$w(a|s;\boldsymbol{\beta}) = \frac{e^{\eta\left(\sum_{i=0}^{4}\beta_{i,a}\tilde{s}^i\right)}}{\sum_{k=\{C,Q\}} e^{\eta\left(\sum_{i=0}^{4}\beta_{i,k}\tilde{s}^i\right)}}$$

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

$$\boldsymbol{\beta} = \begin{bmatrix} \beta_{0,C} & \beta_{0,Q} \\ \beta_{1,C} & \beta_{1,Q} \\ \beta_{2,C} & \beta_{2,Q} \\ \beta_{3,C} & \beta_{3,Q} \\ \beta_{4,C} & \beta_{4,Q} \end{bmatrix} := \begin{bmatrix} \boldsymbol{\beta}_C & \boldsymbol{\beta}_Q \end{bmatrix}$$

Again, the gradient of $\ln w(a|s;\boldsymbol{\beta})$ is available in closed form with components

$$\frac{\partial}{\partial\beta_{i,j}} \ln w(C|s;\boldsymbol{\beta}) = \begin{cases} \eta\big(1 - w(C|s;\boldsymbol{\beta})\big)\tilde{s}^i & j = C \\ -\eta w(Q|s;\boldsymbol{\beta})\tilde{s}^i & j = Q \end{cases} \tag{11.70}$$

and

$$\frac{\partial}{\partial\beta_{i,j}} \ln w(Q|s;\boldsymbol{\beta}) = \begin{cases} -\eta w(C|s;\boldsymbol{\beta})\tilde{s}^i & j = C \\ \eta\big(1 - w(Q|s;\boldsymbol{\beta})\big)\tilde{s}^i & j = Q. \end{cases}$$

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.

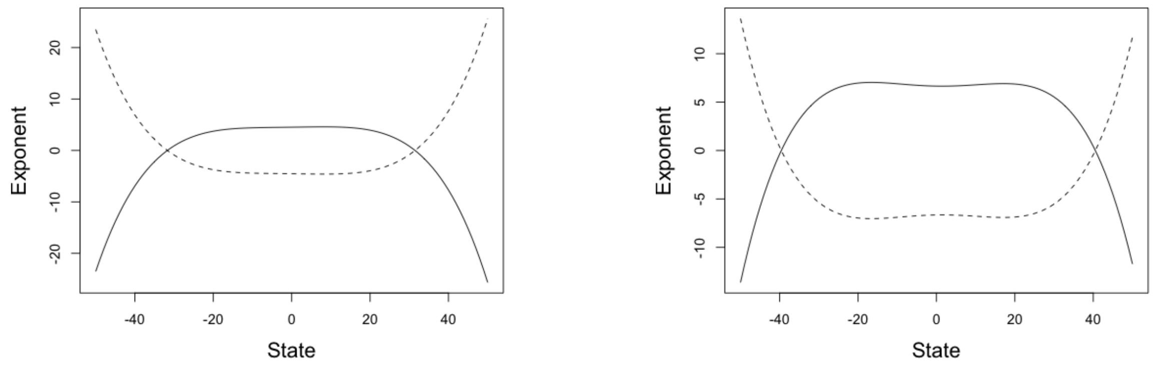
Figure 11.12 shows the fitted exponents as a function of the state using a fourth degree polynomial with $\eta = 1/100$ and $B(s) = -0.3(50)^2$. Observe that the stopping regions differ from the optimal in the case $c = 0.1$ and $\lambda = 1.0$, and also from those obtained using a polynomial fit applied 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.



(a) $c = 0.1$ and $\lambda = 1$ (b) $c = 0$ and $\lambda = 0.95$

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:

$$\boldsymbol{\beta} \leftarrow \boldsymbol{\beta} + \tau_k \nabla_{\boldsymbol{\beta}} \ln \left(w(a^n|s^n; \boldsymbol{\beta}) \right) \left(\sum_{i=n}^{N} \lambda^{i-n} r^i - B(s^n) \right) \tag{11.71}$$

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

$$w(a_{1,2}|s_1;\boldsymbol{\beta}) = 0.988 \quad \text{and} \quad w(a_{2,2}|s_2;\boldsymbol{\beta}) = 0.999.$$

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 literature[29], 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 state[30] s^n and choosing action a^n.

[29]For instance Gronauer et al. [2021]

[30]Recall superscripts correspond to decision epochs.

In other words, the sum represents a realization of the state-action value function $q^{(d_{\boldsymbol{\beta}})^\infty}(s^n, a^n)$ under the randomized stationary policy $(d_{\boldsymbol{\beta}})^\infty$ resulting from using the weights $\boldsymbol{\beta}$. Realizing this, it has been shown[31] that $v^{(d_{\boldsymbol{\beta}})^\infty}(s^n)$, the expected value of the policy $(d_{\boldsymbol{\beta}})^\infty$ 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 $q(s, a; \boldsymbol{\beta}) := q^{(d_{\boldsymbol{\beta}})^\infty}(s, a)$ and $v(s; \boldsymbol{\beta}) := v^{(d_{\boldsymbol{\beta}})^\infty}(s)$. Then

- If $q(s, a; \boldsymbol{\beta}) - v(s; \boldsymbol{\beta}) > 0$, it is desirable to increase the probability of selecting action a in state s.
- If $q(s, a; \boldsymbol{\beta}) - v(s; \boldsymbol{\beta}) < 0$, 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*[a] is defined as

$$A(s, a; \boldsymbol{\beta}) := q(s, a; \boldsymbol{\beta}) - v(s; \boldsymbol{\beta}) \tag{11.73}$$

for $s \in S$, $a \in A_s$ and real-valued vectors $\boldsymbol{\beta}$ of appropriate dimension.

[a]Hopefully, 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 $q(s, a; \boldsymbol{\beta})$ and $v(s; \boldsymbol{\beta})$ 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

$$\nabla_{\boldsymbol{\beta}} v(\boldsymbol{\beta}) = E^{(d_{\boldsymbol{\beta}})^\infty}\left[\sum_{j=1}^{N_\Delta} \nabla_{\boldsymbol{\beta}} \ln\left(w(Y_j|X_j; \boldsymbol{\beta})\right) A(X_i, Y_i; \boldsymbol{\beta})\right]. \tag{11.74}$$

[31]See Sutton et al. [1999] or Grondman et al. [2012].

This representation takes into account that

$$E^{(d_{\boldsymbol{\beta}})^\infty}\left[\sum_{j=1}^{N_\Delta} \nabla_{\boldsymbol{\beta}} \ln\big(w(Y_j|X_j;\boldsymbol{\beta})\big)v(s;\boldsymbol{\beta})\right] = 0.$$

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 $\boldsymbol{\beta}^A$ denote the corresponding vector of weights. Similarly, let $\mathbf{b}_C(s)$ denote the vector of critic (policy value function) features and $\boldsymbol{\beta}^C$ 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.

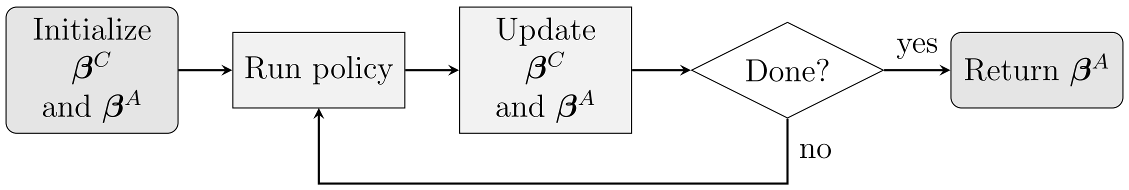

Initialize $\boldsymbol{\beta}^C$ and $\boldsymbol{\beta}^A$
Run policy
Update $\boldsymbol{\beta}^C$ and $\boldsymbol{\beta}^A$
Done?
yes
Return $\boldsymbol{\beta}^A$
no

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 $\boldsymbol{\beta}^A$ and then both $\boldsymbol{\beta}^A$ and $\boldsymbol{\beta}^C$ are updated. Variants of the algorithm interchange the order of updating the weights as well as the method. The critic weights $\boldsymbol{\beta}^C$ can be updated using TD(0), TD(γ) or iterative least squares and the actor weights $\boldsymbol{\beta}^A$ 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

$$v(s;\boldsymbol{\beta}^C) = (\boldsymbol{\beta}^C)^\mathsf{T}\mathbf{b}_C(s).$$

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 $\tau_n^C, n = 1, 2, \ldots$, and $\tau_n^A, n = 1, 2, \ldots$.
   (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:**
   $$A(s,a;\boldsymbol{\beta}^C) \leftarrow r + \lambda v(s';\boldsymbol{\beta}^C) - v(s;\boldsymbol{\beta}^C). \tag{11.75}$$
   (d) **Update critic:**
   $$\boldsymbol{\beta}^C \leftarrow \boldsymbol{\beta}^C + \tau_n^C A(s,a;\boldsymbol{\beta}^C)\mathbf{b}_C(s). \tag{11.76}$$
   (e) **Update actor:**
   $$\boldsymbol{\beta}^A \leftarrow \boldsymbol{\beta}^A + \tau_n^A \nabla_{\boldsymbol{\beta}^A} \ln\big(w(a|s;\boldsymbol{\beta}^A)\big) A(s,a;\boldsymbol{\beta}^C). \tag{11.77}$$
   (f) $n \leftarrow n+1$ and $s \leftarrow s'$.
3. **Terminate:** Return $\boldsymbol{\beta}^A$.

Some comments about this algorithm follow:

1. As stated, the critic update uses TD(0). To see this, express (11.76) as:

$$\boldsymbol{\beta}^C \leftarrow \boldsymbol{\beta}^C + \tau_n^C\big(r + \lambda v(s'; \boldsymbol{\beta}^C) - v(s; \boldsymbol{\beta}^C)\big)\mathbf{b}_C(s).$$

   Alternatively, a TD(γ) update may be used in which case the critic update step will be more complex.

2. The advantage function (11.75) depends on the action a chosen in step 2(a).

3. 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.

4. 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).

5. 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.

6. 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 conditions[32] ensure convergence with probability one under linear function approximation:

1. **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

$$\lim_{n\to\infty} \frac{\tau_n^A}{\tau_n^C} = 0. \tag{11.78}$$

2. **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.

3. **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.

[32]See 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 $s \in S = \{s_1, s_2\}$

$$v(s_i; \boldsymbol{\beta}^C) = \beta^C_{s_i} \quad \text{for} \quad i = 1, 2,$$

where $\boldsymbol{\beta}^C = (\beta^C_{s_1}, \beta^C_{s_2})$.

To implement the above algorithm, sample a from $w(a|s; \boldsymbol{\beta}^A)$, observe (s', r) and compute the advantage function:

$$A(s, a; \boldsymbol{\beta}^C) = r + \lambda v(s'; \boldsymbol{\beta}^C) - v(s; \boldsymbol{\beta}^C) = r + \lambda \beta^C_{s'} - \beta^C_s.$$

Using this notation, the critic update equation (11.76) becomes

$$\beta^C_s = v(s; \boldsymbol{\beta}^C) \leftarrow v(s; \boldsymbol{\beta}^C) + \tau^C_n \big(r + \lambda v(s'; \boldsymbol{\beta}^C) - v(s; \boldsymbol{\beta}^C)\big) = \beta^C_s + \tau^C_n (r + \lambda \beta^C_{s'} - \beta^C_s).$$

The actor update equation (11.77) is given by

$$\boldsymbol{\beta}^A \leftarrow \boldsymbol{\beta}^A + \tau^A_n \nabla_{\boldsymbol{\beta}^A} \ln \big(w(a|s; \boldsymbol{\beta}^A)\big) A(s, a; \boldsymbol{\beta}^C).$$

This means that updates of the actor weights are given by

$$\beta^A_{s',a'} \leftarrow \begin{cases} \beta^A_{s,a} + \tau^A_n \big(1 - w(a|s; \boldsymbol{\beta}^A)\big) A(s, a; \boldsymbol{\beta}^C) & \text{if } s' = s, a' = a \\ \beta^A_{s,a} + \tau^A_n \big(- w(a|s; \boldsymbol{\beta}^A)\big) A(s, a; \boldsymbol{\beta}^C) & \text{if } s' = s, a' \neq a \\ \beta^A_{s,a} & \text{if } s' \neq s. \end{cases}$$

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:

$$w(a_{1,2}|s_1; \boldsymbol{\beta}^A) = 1.00 \quad \text{and} \quad w(a_{2,2}|s_2; \boldsymbol{\beta}^A) = 1.00,$$

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.

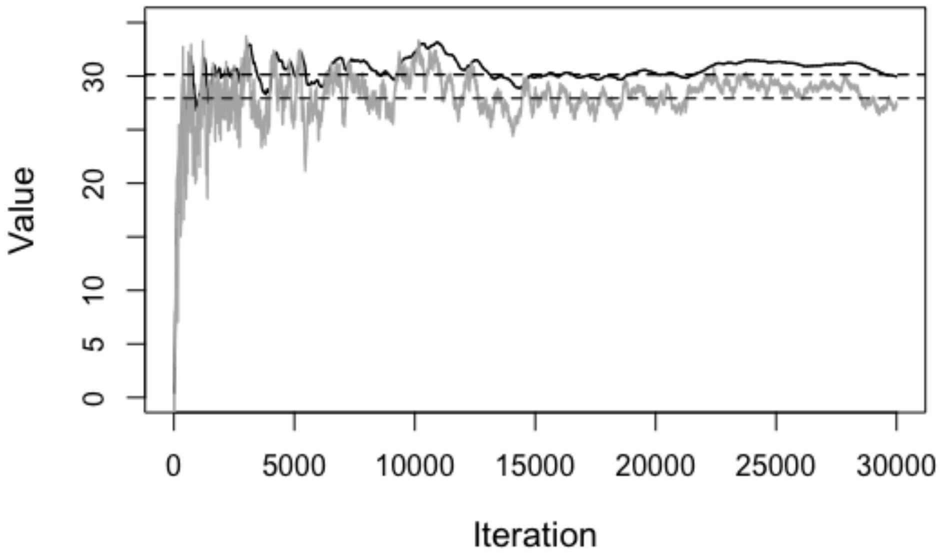


Figure 11.14: Optimal value function estimates in two-state example using Algorithm 11.12 obtained for a typical replicate. Black lines correspond to $v^*(s_1)$ and gray lines to $v^*(s_2)$; solid lines correspond to estimates and dashed lines to the exact value.

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 $\boldsymbol{\beta}^A$.
    (b) Specify learning rate sequence $\tau_n^A, n = 1, 2, \ldots$.
    (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 $w(a|s; \boldsymbol{\beta}^A)$ to
        generate actions. Save the sequence of states, actions and rewards
            $(s^1, a^1, s^2, r^1, \ldots, s^N, a^N, s^{N+1}, r^N),$
        where $N$ denotes the termination epoch of the episode.
    (b) Update critic: Choose
            $\boldsymbol{\beta}^C \in \arg\min_{\boldsymbol{\beta}} \sum_{n=1}^{N} \left( \sum_{i=n}^{N} r^i - v(s^n; \boldsymbol{\beta}) \right)^2.$   (11.79)
    (c) $n \leftarrow 1$.
    (d) Update actor: While $n \leq N$:
        i. Evaluate advantage:
            $A(s^n, a^n; \boldsymbol{\beta}^C) \leftarrow \sum_{i=n}^{N} r^i - v(s^n; \boldsymbol{\beta}^C).$   (11.80)
        ii. Update actor weights:
            $\boldsymbol{\beta}^A \leftarrow \boldsymbol{\beta}^A + \tau_n^A \nabla_{\boldsymbol{\beta}^A} \ln\big(w(a^n|s^n; \boldsymbol{\beta}^A)\big) A(s^n, a^n; \boldsymbol{\beta}^C)$   (11.81)
        iii. $n \leftarrow n + 1$.
    (e) $k \leftarrow k + 1$.
3. Terminate: Return $\boldsymbol{\beta}^A$.
```

Some comments on the algorithm follow.

1. Critic weights $\boldsymbol{\beta}^C$ can also be updated after updating actor weights $\boldsymbol{\beta}^A$. 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
$$h(\boldsymbol{\beta}) := \sum_{n=1}^{N} \left(\sum_{i=n}^{N} r^i - v(s^n; \boldsymbol{\beta}) \right)^2$$
by gradient descent. To do so, an algorithm would compute
$$\nabla_{\boldsymbol{\beta}} h(\boldsymbol{\beta}) = \nabla_{\boldsymbol{\beta}} \left(\sum_{n=1}^{N} \left(\sum_{i=n}^{N} r^i - v(s^n; \boldsymbol{\beta}) \right)^2 \right),$$
which in the linear case equals
$$\nabla_{\boldsymbol{\beta}} h(\boldsymbol{\beta}) = -2 \sum_{n=1}^{N} \left(\sum_{k=n}^{N} r^k - v(s^n; \boldsymbol{\beta}) \right) \mathbf{b}^C(s^n),$$
where $\mathbf{b}^C(s)$ denotes the critic features evaluated at s. Absorbing the constant into the learning rate gives the recursion for $\boldsymbol{\beta}^C$:
$$\boldsymbol{\beta}^C \leftarrow \boldsymbol{\beta}^C - \tau_n^C \nabla_{\boldsymbol{\beta}} h(\boldsymbol{\beta}). \tag{11.82}$$
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:
$$A(s^n, a^n; \boldsymbol{\beta}^C) := r^n + v(s^{n+1}; \boldsymbol{\beta}^C) - v(s^n; \boldsymbol{\beta}^C) \tag{11.83}$$

4. Instead of using policy value functions, one could evaluate the advantage in terms of the state-action value function $q(s, a; \boldsymbol{\beta}^C)$ as follows
$$A(s^n, a^n; \boldsymbol{\beta}^C) := q(s^{n+1}, a^{n+1}; \boldsymbol{\beta}^C) - q(s^n, a^n; \boldsymbol{\beta}^C), \tag{11.84}$$
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

$$v\big((m,n);\boldsymbol{\beta}^C\big) = \beta_0^C + \beta_{1,0}^C m + \beta_{0,1}^C n + \beta_{1,1}^C mn + \beta_{2,0}^C m^2 + \beta_{0,2}^C n^2$$

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 $\boldsymbol{\beta}^A = \{\beta_{(m,n,a)}^A \,|\, (m,n) \in S \text{ and } a \in A_{(m,n)}\}$.

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.

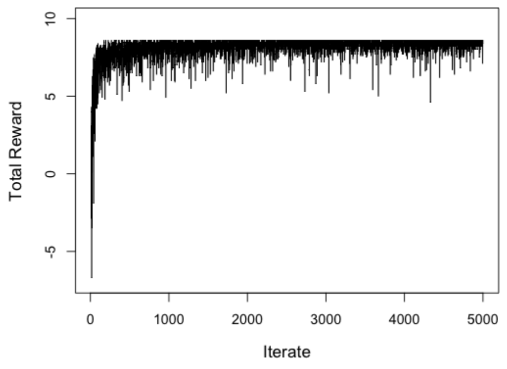
Note that when starting in cell $(1, 1)$, the actor-critic algorithm often finds a sequence of actions[33] with probabilities close to 1 that correspond to a shortest path through the grid. However, since most episodes will follow this path, the optimal probabilities off the path are not well estimated. 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.

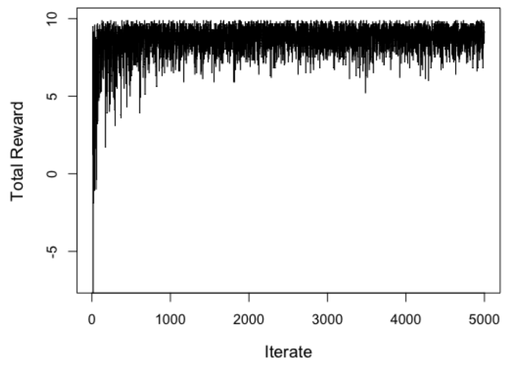
Delivering coffee

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

[33]For 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".



(a) All episodes start in cell $(1, 1)$.



(b) Episodes start in a random non-terminal cell.

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 ($B =$ 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 $w(a|s; \boldsymbol{\beta}^A)$.

Algorithmic details follow. The policy value function is approximated by

$$v\big((m,n); \boldsymbol{\beta}^C\big) = \boldsymbol{\beta}^C_{0,0} + \beta^C_{1,0} m + \beta^C_{0,1} n,$$

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

[34]For 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 $\boldsymbol{\beta}^A$ 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

$$v\big((m,n);\hat{\boldsymbol{\beta}}^C\big) = 4.164 + 0.127m + 0.055n.$$

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

$$v\big((m,n);\hat{\boldsymbol{\beta}}^C\big) = 3.30 + 0.16m + 0.003n.$$

In comparison to the critic estimate above, moving down one row[35] 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

[35] So that coefficients are comparable, note that moving down in this formulation is equivalent to moving up in the previous formulation.

→	↓			
	↓			
	→	→	↓	
×			↓	
×			↓	
×			→	↓
×				↓
×			↓	←
			↓	
*	←	←	←	

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

$$E\left[\Big(\nabla_{\boldsymbol{\beta}} \ln w(a|s;\boldsymbol{\beta})\Big)\Big(\nabla_{\boldsymbol{\beta}} \ln w(a|s;\boldsymbol{\beta})\Big)^{\mathsf{T}}\right].$$

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:

$$r^n + \ldots + r^{n+m} + v(s^{n+m+1};\boldsymbol{\beta}^C) - v(s^n;\boldsymbol{\beta}^C).$$

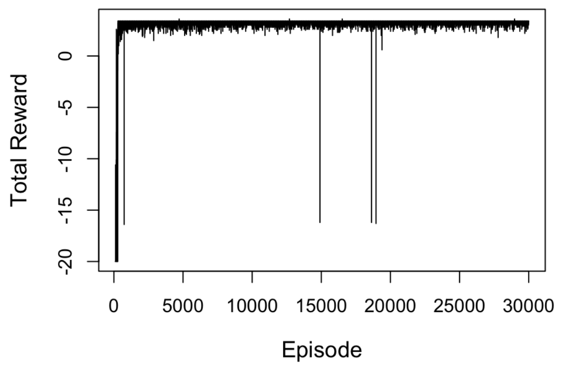


Figure 11.17: Total reward by iterate for a single replicate of the coffee delivering robot problem on a 10×5 grid obtained using Algorithm 11.13.

Improved gradient descent: Adaptive moment estimation[36] 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 $w(a|s; \boldsymbol{\beta}^A)$ and $v(s; \boldsymbol{\beta}^C)$, 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

[36]See 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,

$$E_q[f(X)] := \sum_{x \in X} f(x)q(x) = \sum_{x \in X} f(x)\frac{q(x)}{p(x)}p(x). \tag{11.85}$$

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 $x^1, \ldots, x^N$ 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

$$\frac{1}{5}\big(2 \cdot 1.8 + 10 \cdot 0.2 + 2 \cdot 1.8 + 2 \cdot 1.8 + 10 \cdot 0.2\big) = 2.9$$

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.

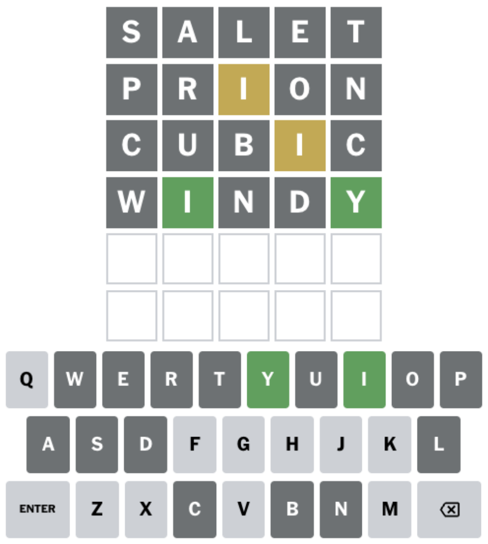


Figure 11.18: Typical Wordle output.

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 $w(a|s;\boldsymbol{\beta})$ is differentiable with respect to each component of $\boldsymbol{\beta}$. Then*

$$\nabla_{\boldsymbol{\beta}} v(\boldsymbol{\beta}) = E^{(d_{\boldsymbol{\beta}})^{\infty}}\left[\sum_{j=1}^{N} \nabla_{\boldsymbol{\beta}} \ln(w(Y_j|X_j;\boldsymbol{\beta}))\left(\sum_{i=j}^{N} r(X_i,Y_i,X_{i+1})\right)\right]. \qquad (11.86)$$

The proof is an easy consequence of the following two results.

Lemma 11.1. Let a_i for $i = 1, \ldots, N$ and b_j for $j = 1, \ldots, N$ be real numbers. Then for N finite

$$\sum_{i=1}^{N} \left(\sum_{j=1}^{i} b_j \right) a_i = \sum_{j=1}^{N} b_j \left(\sum_{i=j}^{N} a_i \right). \tag{11.87}$$

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 $d_{\boldsymbol{\beta}}$ denote a randomized decision rule that chooses actions in each decision epoch according to $w(a|s; \boldsymbol{\beta})$ and let $v_n(\boldsymbol{\beta})$ denote the expected total reward of $(d_{\boldsymbol{\beta}})^\infty$ averaged over initial state distribution $\rho(\cdot)$.

Then if $w(a|s; \boldsymbol{\beta})$ is differentiable with respect to each component of $\boldsymbol{\beta}$:

$$\nabla_{\boldsymbol{\beta}} v_n(\boldsymbol{\beta}) = E^{(d_{\boldsymbol{\beta}})^\infty} \left[\left(\sum_{j=1}^{n} \nabla_{\boldsymbol{\beta}} \ln(w(Y_j|X_j; \boldsymbol{\beta})) \right) r(X_n, Y_n, X_{n+1}) \right]. \tag{11.88}$$

[a]Recall that $r_i(s, a, j)$ denotes the reward in period i in a non-stationary Markov decision process.

Proof. From the definition of $v_n(\boldsymbol{\beta})$,

$$v_n(\boldsymbol{\beta}) := \sum_{h \in H_n} P_{\boldsymbol{\beta}}(h) r(s_n, a_n, s_{n+1}) = E^{(d_{\boldsymbol{\beta}})^\infty} \left[r(X_n, Y_n, X_{n+1}) \right], \tag{11.89}$$

where H_n denotes the set of all histories of the form $h = (s_1, a_1, \ldots, s_n, a_n, s_{n+1})$ and $P_{\boldsymbol{\beta}}(h)$ denotes the probability of history h where actions are chosen in each decision epoch according to $w(a|s; \boldsymbol{\beta})$. By the Markov property,

$$P_{\boldsymbol{\beta}}(h) = \rho(s_1) w(a_1|s_1; \boldsymbol{\beta}) p(s_2|s_1, a_1) \cdots w(a_n|s_n; \boldsymbol{\beta}) p(s_{n+1}|s_n, a_n). \tag{11.90}$$

Since $w(a|s; \boldsymbol{\beta})$ is differentiable with respect to each component of $\boldsymbol{\beta}$, using (11.51) establishes that

$$\nabla_{\boldsymbol{\beta}} P_{\boldsymbol{\beta}}(h) = P_{\boldsymbol{\beta}}(h) \nabla_{\boldsymbol{\beta}} \ln(P_{\boldsymbol{\beta}}(h)). \tag{11.91}$$

Therefore,

$$\nabla_{\boldsymbol{\beta}} v_n(\boldsymbol{\beta}) = \sum_{h \in H_n} P_{\boldsymbol{\beta}}(h) \nabla_{\boldsymbol{\beta}} \ln(P_{\boldsymbol{\beta}}(h)) r(s_n, a_n, s_{n+1}). \tag{11.92}$$

From (11.90)

$$\ln(P_{\boldsymbol{\beta}}(h)) = \ln(\rho(s_1)) + \ln(w(a_1|s_1;\boldsymbol{\beta})) + \ln(p(s_2|s_1,a_1)) \\ + \cdots + \ln(w(a_n|s_n;\boldsymbol{\beta})) + \ln(p(s_{n+1}|s_n,a_n)).$$

Since $\rho(s)$ and $p(j|s,a)$ do not depend on $\boldsymbol{\beta}$,

$$\nabla_{\boldsymbol{\beta}} \ln(P_{\boldsymbol{\beta}}(h)) = \sum_{j=1}^{n} \nabla_{\boldsymbol{\beta}} \ln(w(a_j|s_j;\boldsymbol{\beta})). \tag{11.93}$$

Hence, combining (11.92) and (11.93) gives

$$\nabla_{\boldsymbol{\beta}} v_n(\boldsymbol{\beta}) = \sum_{h \in H_n} P_{\boldsymbol{\beta}}(h) \left(\sum_{j=1}^{n} \nabla_{\boldsymbol{\beta}} \ln(w(a_j|s_j;\boldsymbol{\beta})) \right) r(s_n, a_n, s_{n+1}). \tag{11.94}$$

Rewriting this in expectation form gives (11.88). □

Proof of Theorem 11.2. It is easy to see that

$$\nabla_{\boldsymbol{\beta}} v(\boldsymbol{\beta}) = \sum_{n=1}^{N} \nabla_{\boldsymbol{\beta}} v_n(\boldsymbol{\beta}),$$

so that substituting the representation in (11.88) into the above result and applying Lemma 11.1, with $b_j = \ln w(Y_j|X_j;\boldsymbol{\beta})$ and $a_i = r(X_i, Y_i, X_{i+1})$, gives (11.86). □

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 REINFORCE[37] 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

[37]REINFORCE is an acronym for the expression "REward Increment = Non-negative Factor × Offset Reinforcement × 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 $G_1 := \{1, 2, 3, 4\}, G_2 := \{5, 6, 7, 8\}, \ldots, G_{15} = \{57, 58, 59, 60\}$. 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 $k = 1, \ldots, 15$ 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

$$w(a_1|\beta) = \frac{e^\beta}{e^\beta + 1}, \qquad w(a_2|\beta) = \frac{1}{e^\beta + 1}.$$

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

$$E^{(d_{\boldsymbol{\beta}})^\infty}\left[\sum_{j=1}^{N} \nabla_{\boldsymbol{\beta}} \ln\left(w(Y_j|X_j; \boldsymbol{\beta})\right) B(X_j)\right] = 0. \tag{11.95}$$

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:

    - $s_1 \to s_2$ with reward $r = 0$
    - $s_2 \to s_3$ 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 $s \to s'$ with reward r, define the squared temporal difference error:
    $$g(\boldsymbol{\beta}) = \left(r + \lambda v(s'; \boldsymbol{\beta}) - v(s; \boldsymbol{\beta})\right)^2.$$
    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 $s_1 \to s_2$:
    - $v(s_1; \boldsymbol{\beta})$, $v(s_2; \boldsymbol{\beta})$, 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 $s_2 \to s_3$ 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.