Introduction to Markov Decision Processes

Martin L. Puterman and Timothy C. Y. Chan

July 15, 2024

Chapter 1

Simulation with Function Approximation

Everything we know is only some kind of approximation, because we know that we do not know all the laws yet. Therefore, things must be learned only to be unlearned again or, more likely, to be corrected.

Richard Feynman, American theoretical physicist, 1918-1988.

Modern applications of Markov decision processes require both function approximation (Chapter **??**) and simulation[1] (Chapter **??**). 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.

Methods in this chapter motivate, describe and apply methods for:

- value-function approximation,
- state-action value function approximation, and
- policy approximation.

Whereas value function and state-action value function approximation generalize methods in Chapters **??** and **??**, this chapter also introduces the concept of policy approximation. Policy approximation refers to approximating the action-choice probability distribution of a randomized policy.

The general approach is to represent a value function, a state-action value function, or an action-choice probability in terms of a parametric function of features and then to estimate the parameters on-line or in batch mode using output from simulations. Often parameters are referred to as *weights*.

Figure 1.1 summarizes the methods described in this chapter. Note that this chapter considers only discounted and episodic models and requires familiarity with gradient descent and least-squares regression.

[1]As noted earlier we use the expression *simulation* to refer to either computer-based simulators or real-time experimentation.

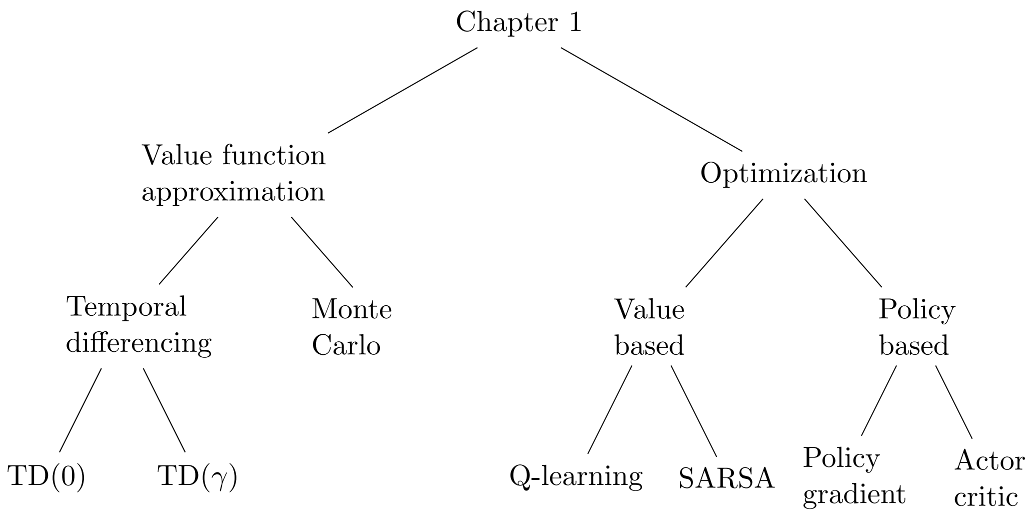


Figure 1.1: Schematic representation of methods described in Chapter 1

1.1 The challenge of large models

In contrast to the tabular models analyzed in Chapter **??**, models with large state (and action) spaces require methods to represent value functions and decision rules in forms suitable for computation and application. The following sections discuss methods for doing so.

1.1.1 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. We refer to these functions 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 grid-world type 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 choosing features to be indicator functions of all individual states or state-action pairs is equivalent to using a *tabular model*.

Combining features

Once features are specified, the issue of how to combine features to obtain estimates of quantities of interest arises. In general, value functions, state-action value functions or action choice probabilities can be approximated by:

1. linear functions of features
2. non-linear functions of features
3. neural networks[2]

Regarding notation, vectors of features evaluated at s or (s, a) are denoted by $\mathbf{b}(s)$ or $\mathbf{b}(s, a)$ and vectors of weights in both cases by $\boldsymbol{\beta}$. Components of vectors are written as $b_k(s)$ or $b_{k,j}(s, a)$.

1.1.2 Value function approximation

We begin with a discussion of approximating the value function approximation[3] of a fixed policy. For ease of exposition and to encompass many practical applications, we assume a linear value function approximation of the form:

$$v(s; \beta_0, \ldots, \beta_K) := \beta_0 b_0(s) + \beta_1 b_1(s) + \ldots + \beta_K b_K(s), \tag{1.1}$$

where $\{\beta_0, \ldots, \beta_k\}$ denote parameters and $\{b_0(s), \ldots, b_K(s)\}$ denote the value of the features evaluated at $s \in S$. We begin the indexing at 0 too conform to the convention in linear regression models in which $b_0(s_0) = 1$ for all $s \in S$ so that β_0 corresponds to a constant.

In vector notation (which we prefer) we write this as:

$$\boxed{v(s; \boldsymbol{\beta}) = \mathbf{b}(s)^T \boldsymbol{\beta} \qquad (1.2)}$$

where $\mathbf{b}(s)$ denotes a (column) vector of pre-specified features evaluated at $s \in S$ and $\boldsymbol{\beta}$ denotes a weight vector. Since the quantity in (1.2) is a scalar, it equivalently can be written as $\boldsymbol{\beta}^T \mathbf{b}(s)$.

The beauty of such a representation is that in an expression such as (1.1) 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, $v(s; \boldsymbol{\beta}) = \beta_0 + \beta_1 s$ represents an approximation to the expected infinite horizon discounted cost, then β_1 represents the increase in expected cost of adding an additional 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 β will be denoted by $\hat{\boldsymbol{\beta}}$ and estimates of value functions by either $\hat{v}(s)$ or $v(s; \hat{\boldsymbol{\beta}})$.

[2]Of course, neural networks are non-linear functions but we distinguish them because they have a vast and specialized array of methods for parameter estimation.

[3]As noted earlier, the problem of estimating a value function for a specified policy is referred to as *value function prediction* in the computer science literature.

1.1.3 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

$$b_{k,j}(s,a) = f_k(s)h_j(a) \tag{1.3}$$

where $f_k(s)$ for $k = 0, 1, \ldots, K$ is a function defined on states and $j = 1, \ldots, J$ and $h_j(a)$ is a function defined on actions[4]. In the linear case this becomes

$$\boxed{q(s,a;\boldsymbol{\beta}) \approx \sum_{k=0}^{K}\sum_{j=1}^{J} \beta_{k,j} f_k(s) h_j(a) \qquad (1.4)}$$

where $\boldsymbol{\beta} = \{b_{0,1}, \ldots, \beta_{K,J}\}$. In greater generality, $q(s,a,\boldsymbol{\beta})$ may be a pre-specified non-linear function of features.

When $A_s = \{a_1, \ldots, a_J\}$ contains a small number of elements and doesn't 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 :

$$h_j(a) = I_{\{a_j\}}(a) = \begin{cases} 1 & a = a_j \\ 0 & a \neq a_j. \end{cases} \tag{1.5}$$

Consequently

$$b_{k,j}(s,a) = \begin{cases} f_k(s) & a = a_j \\ 0 & a \neq a_j \end{cases} \tag{1.6}$$

for $a \in A_s$ and $s \in S$. This is equivalent to representing $q(s,a)$ as a function of s that uses different weights for each $a \in A_s$.

To make this concrete consider a queuing-control model (such as in Section **??** with service rates a_1 and a_2 and suppose $f_0(s) = 1$ and $f_1(s) = s$ and $h_j(a) = I_{\{a_j\}}(a)$ as in (1.5). Then using (1.4), the approximation of $q(s,a)$ can be written as

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

which is equivalent to using the following possibly different linear functions for each action as follows

$$\begin{aligned} q(s,a_1;\boldsymbol{\beta}) &= \beta_{0,1} + \beta_{1,1}s \\ q(s,a_2,\boldsymbol{\beta}) &= \beta_{0,2} + \beta_{1,2}s. \end{aligned}$$

[4]Note that we start indexing $f_k(s)$ at 0 to allow for a constant function in the state approximation.

This means that using this approximation corresponds to approximating $q(s, a)$ by lines with different slopes and intercepts for each action.

We find it convenient to write (1.4) in vector form

$$\boxed{q(s, a; \boldsymbol{\beta}) = \boldsymbol{\beta}^T \mathbf{b}(s, a) \qquad (1.7)}$$

where $\boldsymbol{\beta}$ is a column vector of weights of length $(K+1)J$ and $\mathbf{b}(s, a)$ is a column vector of features evaluated at (s, a) of length $(K + 1)J$ where the ordering of weights and features in the vectors is consistent.

In the above example

$$\boldsymbol{\beta} = \begin{bmatrix} \beta_{0,1} \\ \beta_{0,2} \\ \beta_{1,1} \\ \beta_{1,2} \end{bmatrix}, \quad \mathbf{b}(s, a_1) = \begin{bmatrix} 1 \\ 0 \\ s \\ 0 \end{bmatrix} \quad \text{and} \quad \mathbf{b}(s, a_2) = \begin{bmatrix} 0 \\ 1 \\ 0 \\ s \end{bmatrix}. \tag{1.8}$$

Another alternative, which is appropriate when the states and actions are vectors $\mathbf{s}$ and $\mathbf{a}$ respectively, is to approximate $q(\mathbf{s}, \mathbf{a}; \boldsymbol{\beta})$ by a neural network with inputs $\mathbf{s}$ and $\mathbf{a}$.

State-action functions and 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 (1.7) shows that we can approximation value functions by

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

State-action value function approximations and decision rules

When we analyzed tabular models, we were were able to specify decision rules **explicitly**. In the case of deterministic decision rules we could specify the action to choose in each state in the form of a look-up table and for a randomized policy we could specify 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 $\boldsymbol{\beta}$ assumes the role of a decision rule. That is instead of explicitly encoding the policy, one can use $\boldsymbol{\beta}$ to generate actions in practice or in an algorithm as follows:

Algorithm 1.1. Implicit action choice

```
1. Specify $\boldsymbol{\beta}$.
2. Select a state $s \in S$.
3. Compute $q(s, a; \boldsymbol{\beta}) = \boldsymbol{\beta}^T \mathbf{b}(s, a)$ for all $a \in A_s$.
4. (a) Deterministic action selection: Set $a_s = \arg\max_{a \in A_s} q(s, a; \boldsymbol{\beta})$.
   (b) Randomized action selection: Sample $a_s$ using $\epsilon$-greedy or using
       softmax action selection based on $q(s, a; \boldsymbol{\beta})$.
5. Return $a_s$.
```

To think of this another way, instead of carrying around a look-up table, our agent will have a (much lighter) weight function $\boldsymbol{\beta}$ in hand. When in state s, the agent can then apply Algorithm 1.1 and use the resulting action a_s.

From this perspective it would be difficult to specify $\boldsymbol{\beta}$ so as to represent a specific policy implicitly. However in the special case when the model is known and rewards are independent of the subsequent state, Choosing $\boldsymbol{\beta} = \mathbf{0}$ would result in setting $q(s, a; \boldsymbol{\beta}) = r(s, a)$ so that greedy action selection would generate a myopic policy.

1.1.4 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 $w(a|s) \geq 0$ 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[5] or logistic transformation, is given by

$$w(a|s; \boldsymbol{\beta}) = \frac{e^{\boldsymbol{\beta}^T \mathbf{b}(s,a)}}{\sum_{a' \in A_s} e^{\boldsymbol{\beta}^T \mathbf{b}(s,a')}} \tag{1.9}$$

where $\boldsymbol{\beta}_a$ is the parameter associated with choosing action a in state s.

An alternative is to represent $w(a|s; \boldsymbol{\beta})$ as a *neural network* where $\boldsymbol{\beta}$ is the vector of weights.

1.2 Value function approximation

This section focuses on simulation methods for approximating value function. It describes Monte Carlo and temporal differencing methods.

[5]Previously, the softmax function was used for exploration, here it is used to represent a policy directly.

1.2.1 Monte Carlo 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 value function using least squares. For ease of exposition we only provide a starting state version of the algorithm.

We take the view that the objective of this section is to show how one might approximate value functions in applications in which the value function approximation is generated without analyst input as would be the case when seeking an optimal policy. Therefore the set of starting states and features are not chosen to conform to a specific policy.

Monte Carlo estimation of value function approximations in an episodic model

Recall that S_Δ denotes the set of stopped states[6] and $g(s)$ denotes the value on termination in state s. The algorithm assumes a randomized stationary policy.

Algorithm 1.2. Starting state Monte Carlo value function approximation in an episodic model

```
1. Initialize:
    (a) Choose $d \in D^{MR}$ and the number of replicates $M$.
    (b) Specify a subset of starting states $\bar{S}$ of $S$.
    (c) Create empty list VALUES($s$) for each $s \in \bar{S}$.
2. Generate values:
    (a) For each $s \in \bar{S}$:
    (b) $m \leftarrow 1$,
    (c) While $m < M$,
        i. Start episode: $v(s) \leftarrow 0$.
        ii. Sample $a$ from $w_d(\cdot|s)$.
        iii. Generate $(s', r)$ or sample $s'$ from $p(\cdot|s,a)$ and set $r = r(s,a,s')$ .
        iv.
            $v(s) \leftarrow \begin{cases} r + v(s) & s' \notin S_\Delta \\ g(s') + v(s), & s' \in S_\Delta. \end{cases}$
```

[6]As noted earlier we can assume without loss of generality that S_Δ consists of a single state by adding a zero-reward transition from each $s \in S_\Delta$ into a single absorbing state Δ.

   v. **End episode:** If $s' \in S_\Delta$, append $v(s)$ to VALUES(s) and $m \leftarrow m+1$. Else, $s \leftarrow s'$ and return to 2c(ii).

3. **Estimate parameters:** Use data $\{(s,v) : s \in \bar{S}, v \in \text{VALUES}(s)\}$ to obtain (weighted) least squares estimates of parameters $\hat{\boldsymbol{\beta}} = (\hat{\beta}_0, \ldots, \hat{\beta}_K)$.

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 $|\bar{S}|M$ data points of the form $(s, v(s))$. When $v(s)$ is assumed to be linear in parameters, least squares parameter 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 **??** it was easy and more efficient to modify the algorithm to store estimates for all visited states.
4. When the variability of the Monte Carlo estimates vary with state, weighted least squares provides an alternative approach to parameter estimation.

An example

To illustrate these concepts, we apply the above algorithm to compute the value function in a model of optimal stopping in a random walk.

Example 1.1. Monte Carlo value function estimation on a random walk with stopping.

Consider the random walk model described in Example **??** but with $S = \{1, \ldots, 500\}$, continuation cost $c = 8$, stopping reward $g(s) = .3s + .7s^2$ and $p = .51$ where p denotes the probability of a transition from s to $s+1$ except in state N where it denotes the probability of remaining in state N.

The decision maker trades off the cost of continuing versus the cost 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 linear polynomial approximations of the form

$$v(s; \beta_0, \beta_1, \ldots, \beta_K) = \beta_0 + \beta_1 s + \ldots + \beta_K s^K$$

for small values of K and considers three policies:

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

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

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

Clearly under π_1, no approximation is necessary and $v^{\pi_1}(s) = g(s)$. To apply Algorithm 1.2 to π_2 and π_3, set $\bar{S} = \{1, 25, 50, 100, 150, \ldots, 400, 450, 475, 500\}$ and $M = 10$ replications for each $s \in \bar{S}$[a]

Figure 1.2a) shows linear (solid line) and cubic ($K = 3$) approximations to the Monte Carlo values for policy π_2. The linear approximation was

$$v^{\pi_2}(s; \hat{\beta}_0, \hat{\beta}_1) = -26549.1 + 404.7s.$$

and the cubic approximation was

$$v^{\pi_2}(s; \hat{\beta}_0, \hat{\beta}_1, \hat{\beta}_2, \hat{\beta}_3) = -11459.94 + 17.90s + 1.72s^2 - .0021s^3.$$

The cubic fit was slightly more accurate with a smaller standard error. The fourth order fits were almost identical to that of the cubic model.

Figure 1.2b shows linear and fourth-order approximations for policy π_3. Note that a fourth-order polynomial was necessary to well represent 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

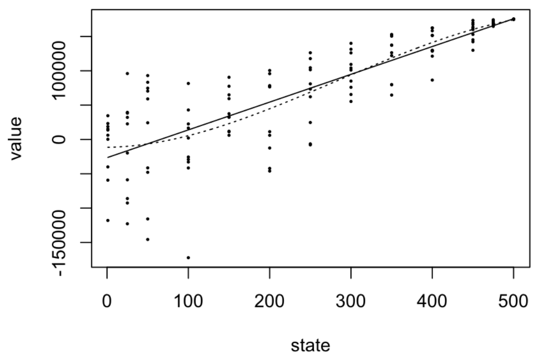
$$v^{\pi_3}(s, \hat{\beta}_0, \hat{\beta}_1, \ldots, \hat{\beta}_4) = 647.3 - 123.43s + 3.41s^2 - 0.009s^3 + 0.000009s^4.$$

We leave it as an exercise to develop suitable function approximations for other parameter values.

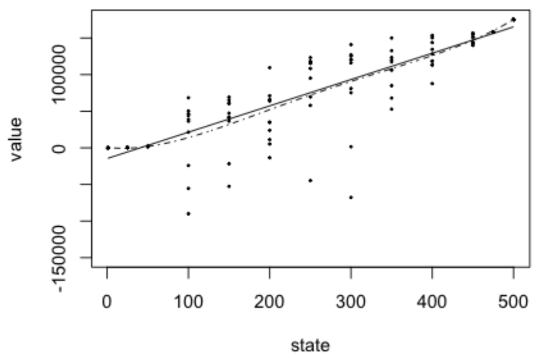
[a] The example chooses extra values close to the endpoints to achieve greater accuracy. Note also that replications in the stopping regions were unnecessary since $v(s) = g(s)$ therein.

Discounted models using truncation and Monte Carlo estimation

Our approach to approximating the value function in a discounted model is similar to that in an episodic model however we can either truncate trajectories after a fixed number of iterations or at a geometric stopping time. The truncation version runs each trajectory for N decision epochs. The geometric stopping time version is identical to that of an episodic model in which at each decision epoch the system can enter an



(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.

absorbing state with probability $(1 - \lambda)$. We formally state a truncation version of the algorithm here and leave algorithmic description and application of the geometric stopping version as an exercise. The algorithm assumes a randomized stationary policy.

Algorithm 1.3. Starting state Monte Carlo policy approximation in a discounted model with truncation.

```
1. Initialize:
    (a) Choose $d \in D^{MR}$.
    (b) Specify the number of replicates $M$ and the truncation level $N$.
    (c) Specify a subset of starting states $\bar{S}$.
    (d) Create empty list VALUES($s$) for all $s \in \bar{S}$.
2. Generate values:
    (a) For each $s \in \bar{S}$:
    (b) $m \leftarrow 1$.
    (c) While $m \leq M$,
        i. Start episode: $v(s) \leftarrow 0$ and $n \leftarrow 0$.
        ii. While $n < N$:
            A. Sample $a$ from $w_d(\cdot|s)$.
            B. Simulate to obtain $(s', r)$ or sample $s'$ from $p(\cdot|s, a)$ and set
               $r = r(s, a, s')$.
```

```
                C. $v(s) \leftarrow \lambda^n r + v(s)$.
                D. $s \leftarrow s'$.
                E. $n \leftarrow n + 1$.
            iii. Add $v(s)$ to VALUES($s$).
        (d) $m \leftarrow m + 1$.
3. Estimate parameters: Use data $\{(s, v) : s \in \bar{S}, v \in \text{VALUES}(s)\}$ to
   obtain (weighted) least squares estimates of parameters $\hat{\boldsymbol{\beta}} = (\hat{\beta}_0, \ldots, \hat{\beta}_K)$.
```

Note that an every-state version of this algorithm would be problematic because the truncation lengths would vary from state to state.

An example

Example 1.2. Monte Carlo estimates of value function in queuing service rate control model.

This example applies Algorithm 1.3 to the queuing control model on $S = \{0, \ldots, 50\}$. It explores the quality of cubic polynomial approximations to 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

$$d_2(s) = a_2 \quad \text{for } 0 \leq s \leq 50,$$

with $\lambda = 0.9, 0.95, 0.98$ over 40 replicates with common random number seeds. In each replicate, episodes are truncated at $N = 600$, $\bar{S} = \{0, 5, 10, \ldots, 45, 50\}$ and $M = 10$.

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

$$\text{RMSE} = \left(\frac{1}{|S|} \sum_{s \in S} (v(s, \hat{\boldsymbol{\beta}}) - v_\lambda^{d^\infty}(s))^2 \right)^{\frac{1}{2}}. \tag{1.10}$$

Note that $v_\lambda^{d^\infty}(s)$ can be easily computed exactly using policy evaluation methods in Chapter **??**. which can be computed because the values of these policies can be evaluated exactly.

Table 1.1 summarizes results. The column "True" establishes a baseline by providing the RMSE of a cubic polynomial fit to the exact value function. Observe that:

1. The accuracy of a cubic fit to the true model decreases with λ as does the accuracy of each of the Monte Carlo estimates.

2. The cubic model fits d_2 better than d_1. This is expected because of the step change in d_1 at $s = 26$ (see Figure 1.3a).

3. Least square fits provide more accurate estimates of the value function than weighted least square estimates using the estimated variance of values at each $s \in \bar{S}$.

4. Results (not shown) indicated that accuracy 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 1.3 compares, for a single replicate, the true value function, its cubic approximation and a cubic polynomial fitted to Monte Carlo estimates. Figure 1.3a shows that for decision rule d_1 and $\lambda = 0.95$, the cubic polynomial deviates from the true value function and the cubic fit based on the Monte Carlo simulation lies below the true value function. Figure 1.3b shows that for decision rule d_2 and $\lambda = 0.98$, the cubic polynomial approximation well approximates the true value function, the cubic fit based on the Monte Carlo simulation deviates from the true value function at high levels.

We investigate the impact of these discrepancies on optimization below.

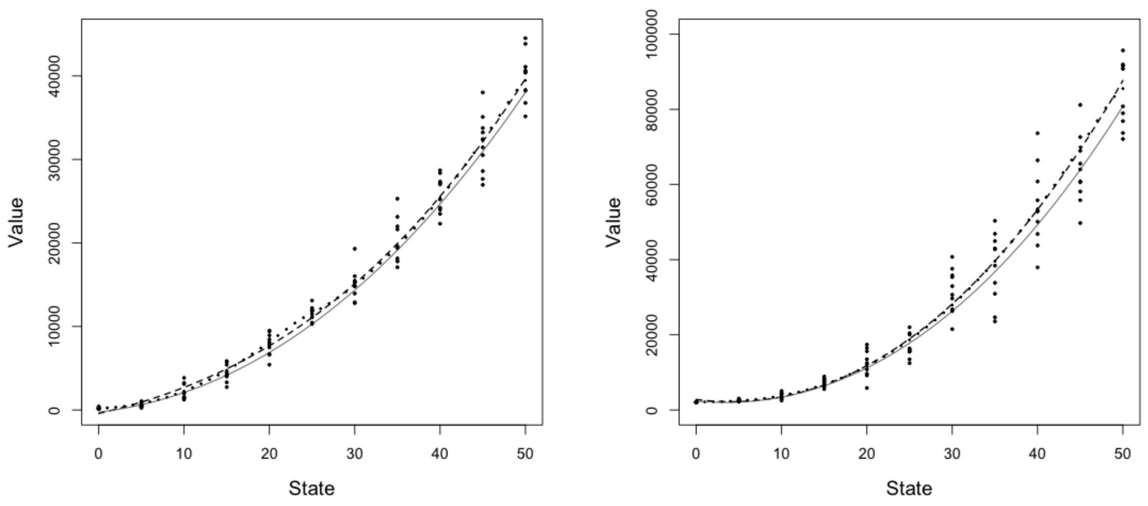
	d_1			d_2		
λ	True	MC-LS	MC-WLS	True	MC-LS	MC-WLS
0.90	67.99	551.14	569.17	59.15	567.90	579.10
0.95	344.73	1029.94	1135.56	153.05	1098.57	1151.50
0.98	1261.33	2161.07	2801.03	432.18	2480.18	2770.44

Table 1.1: 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 value (True) as a function of discount rate and decision rule.

1.2.2 TD(0) with value function approximation

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

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



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

Figure 1.3: Plots of estimated and true value functions based on starting state Monte Carlo based on a single replicate. Points indicate Monte Carlo values, the dotted line equals the true value function, the dashed line indicates the fitted cubic polynomial approximation to the true value function and the grey solid line denotes the least square estimate based on the Monte Carlo values. Note that the y-axis scales for the two figures differ.

Motivation

This section applies stochastic approximation methods in Chapter **??** Appendix B to obtain a recursive method for estimating a vector of weights in an approximate value function.

Let d denote a Markovian randomized decision rule, d^∞ the corresponding stationary policy and s an arbitrary state in S. The objective is find a vector of weights $\boldsymbol{\beta} \in \Re^K$ so as to "solve" the Bellman equation

$$v(s;\boldsymbol{\beta}) \approx E\big[r(X,Y,X') + \lambda v(X';\boldsymbol{\beta})\big|X = s\big]$$

for all $s \in S$, where the expectation is with respect to the conditional distribution of the action Y and next 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 expected squared error loss

$$E^{d^\infty}\Big[(r(X,Y,X') + \lambda v(X';\boldsymbol{\beta}) - v(X;\boldsymbol{\beta}))^2\Big|X = s\Big] \tag{1.11}$$

for all $s \in S$ where the expectation is respect to the conditional distribution of action Y and the next state X' under the probability distribution corresponding to decision rule d in state s. Since a different $\boldsymbol{\beta}$ may achieve the minimum above for each $s \in S$, an alternative is the objective

$$G_d(\boldsymbol{\beta}) := E^{d^\infty}\left[(r(X,Y,X') + \lambda v(X';\boldsymbol{\beta}) - v(X;\boldsymbol{\beta}))^2\right] \tag{1.12}$$

where the expectation is now with respect to the state X as well. Possible choice for a distribution for X are a uniform distribution over states or the stationary distribution of the Markov chain corresponding to d^∞.

If one could evaluate the above expectation directly then the gradient descent recursion

$$\boldsymbol{\beta}' = \boldsymbol{\beta} - \tau \nabla_{\boldsymbol{\beta}} G_d(\boldsymbol{\beta})$$

would provide an estimate of $\boldsymbol{\beta}$.

We now describe an alternative approach suitable for use on simulated data. This scheme assumes that given a specified value of $\boldsymbol{\beta}'$ 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 $\boldsymbol{\beta}$. To do so, TD(0) 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

$$\boldsymbol{\beta}' = \boldsymbol{\beta} - \tau \nabla_{\beta} v(s;\boldsymbol{\beta})\big(u - v(s;\boldsymbol{\beta})\big) \tag{1.13}$$

where the factor 2 is absorbed into τ.

When $v(s,\boldsymbol{\beta})$ is a linear function of $\boldsymbol{\beta}$, $v(s;\boldsymbol{\beta}) = \boldsymbol{\beta}^T \mathbf{b}(s)$, t $\nabla_{\boldsymbol{\beta}} v(s;\boldsymbol{\beta}) = \mathbf{b}(s)$, so that the gradient descent recursion becomes:

$$\boldsymbol{\beta}' = \boldsymbol{\beta} - \tau \mathbf{b}(s)\big(u - \boldsymbol{\beta}^T \mathbf{b}(s)\big). \tag{1.14}$$

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 TD(0):

$$\boxed{\boldsymbol{\beta}' = \boldsymbol{\beta} - \tau\, \delta\, \mathbf{b}(s). \qquad (1.15)}$$

Note that in this expression τ and δ are scalars and $\boldsymbol{\beta}$ and $\mathbf{b}(s)$ are column vectors.

TD(0) with value function approximation in algorithmic form; linear approximation

We now provide an algorithm for implementing TD(0) in a discounted model. It evaluates a Markovian random decision rule d and allows for flexibility in selecting the sequence of states.

Algorithm 1.4. TD(0) with linear value function approximation.

1. **Initialize:**
   - (a) Specify the number of iterations N and the learning rate sequence $\{\tau_n : n = 1, \ldots\}$.
   - (b) Specify a decision rule d to be evaluated, an initial vector $\boldsymbol{\beta}$ and a state $s \in S$.
   - (c) $n \leftarrow 1$.
2. **Iterate:** For $n < N$:
   - (a) Sample a' from $w_d(a|s)$.
   - (b) Generate (s', r) or sample $s' \in S$ from $p(s'|s, a)$ and set $r = r(s, a', s')$.
   - (c) **Compute the temporal difference:** Set $v(s; \boldsymbol{\beta}) = \mathbf{b}(s)^T \boldsymbol{\beta}$, $v(s'; \boldsymbol{\beta}) = \mathbf{b}(s')^T \boldsymbol{\beta}$ and
   $$\delta = r + \lambda v(s'; \boldsymbol{\beta}) - v(s; \boldsymbol{\beta}). \tag{1.16}$$
   - (d) **Update $\boldsymbol{\beta}$:**
   $$\boldsymbol{\beta}' \leftarrow \boldsymbol{\beta} - \tau_n \delta \mathbf{b}(s) \tag{1.17}$$
   - (e) $\boldsymbol{\beta} \leftarrow \boldsymbol{\beta}'$ and $n \leftarrow n + 1$.
   - (f) **Next state generation:**
     (Long-trajectory) $s \leftarrow s'$;
     (Random) Generate s from a uniform distribution on S.
     (Hybrid) Specify δ small and a state s_0. With probability δ, $s \leftarrow s_0$ and with probability $1 - \delta$, $s \leftarrow s'$.
3. Return $\hat{\boldsymbol{\beta}}$.

Some comments on implementing this algorithm follow:

1. **Specifying d:** 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**, which is based on state-action value functions $q(s, a)$, chooses a' (deterministically) in step 2(a) by
   $$a' \in \arg\max_{a \in A_s} q(s, a; \boldsymbol{\beta}).$$

- **Actor-critic** samples a' from $w(\cdot|s;\boldsymbol{\beta}^C)$ where the estimate of the weight $\boldsymbol{\beta}$ 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[8] with probability 1 provided the samples follow the trajectory of the Markov chain corresponding to δ and $(\tau_n : n = 1, 2, \ldots)$ satisfies the Robbins-Monro conditions. Of course this also means that the corresponding value functions converge. Generalization to non-linear approximations is more problematic.

3. **Initialization:** As in most nonlinear optimization problems, convergence is sensitive to initial specification of $\boldsymbol{\beta}$. We believe a limited Monte Carlo analysis based on one or more replicates at a selected subset of states followed by least squares parameter estimate provides reliable staring values.

4. **Stopping Rules:** We omit a precise stopping rule in the algorithm statement above. One could be based on either changes in parameter values

$$\left[\frac{1}{K+1}\sum_{k=0}^{K}(\beta_k' - \beta_k)^2\right]^{\frac{1}{2}} \tag{1.18}$$

or in changes in fitted values:

$$\left[\frac{1}{|S|}\sum_{s\in S}(v(s,\boldsymbol{\beta}') - v(s,\boldsymbol{\beta}))^2\right]^{\frac{1}{2}}. \tag{1.19}$$

or a weighted variant thereof. Use of these criteria have limitations: (1.18) may be dominated by the impact of a few large coefficients while (1.19) may be computationally prohibitive due to the magnitude of S. Note that (1.19) may be the more appropriate because it is consistent with least squares objective function used to derive TD(0).

5. **State updating:** Step 2(f) 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 Example 1.3 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.

[8]See Tsitsiklis and van Roy [1997].

6. **Episodic models:** Since episodic models terminate at a state in S_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.

Value function approximation in the queuing service rate control model

Because we intend to generalize this algorithm to state-action value function, we carried out a detailed study of the impact of learning rate and restart method for 3 decision rules.

Example 1.3. This example applies Algorithm 1.4 to the model in Example 1.2 with $\lambda = 0.95$. It uses a cubic polynomial approximation to the value function of three deterministic stationary policies based on the decision rules:

$$d_1(s) = \begin{cases} a_1 & 0 \le s \le 25 \\ a_3 & 25 < s \le 50 \end{cases}$$

$$d_2(s) = \quad a_2 \quad 0 \le s \le 50$$

$$d_3(s) = \begin{cases} a_2 & 0 \le s \le 9 \\ a_3 & 10 \le s \le 29 \\ a_1 & 30 \le s \le 50 \end{cases}$$

Step 2(f) used long-trajectory, random-restart options and a hybrid variant that with probability 0.008 jumped 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.

As a result of initial experimentation four learning rates are compared: $\tau_n = 0.1n^{-0.5}, \frac{0.1log(n+1)}{n}, \frac{0.1}{1+10^{-5}n^2}, \frac{150}{750+n}$ referred to respectively as polynomial, logarithmic, STC and ratio. Experiments compared all 36 combinations of these configurations over 40 replicates of simulations of length K =20,000 using common random number seeds for all instances in a replicate.

Preliminary calculations showed that that "convergence" of estimates was highly sensitive to starting values and feature scaling. To obtain reliable initial weights, implementations:

Were initialized with one replicate of starting-state Monte Carlo on a subset $\bar{S}$ of states;

Scaled the states by subtracting the mean and dividing by the standard deviation, and

Used least squares estimation (linear regression) to obtain preliminary estimates of the parameters of a cubic polynomial.

Figure 1.4 compares the effect of the the factors on the quality of the fit for each configuration. From it one can conclude that:

1. Fits for d_1 and d_2 were more accurate than for d_3.

2. The random restart method gave the best result over the three decision rules, however the hybrid method gave similar results to random restart for d_1 and d_2. The reason it didn't work well for d_3 is that under this policy the system remained in high occupancy states. If the hybrid method was modified to jump to a low occupancy state, we suspect its performance would be equivalent to the random restart method.

3. For d_1 and d_2, long trajectory methods gave the least accurate and most variable estimates in comparison to the other restart methods. This was because under these policies, the system remained in low occupancy states in steady state. Consequently, by allowing a small probability of restarting at state 50, accuracy and variability improved considerably.

4. For random restart, the effect of learning rate was small. With random restart, the STC and exponential learning rates gave the best results. Surprisingly the exponential learning rate worked well across all configurations, especially when using long trajectories.

We now describe 5 randomly selected replicates in more detail. Each used decision rule d_1, STC learning rate and compared random and hybrid state generation. Table 1.2 summarizes the RMSE of the fit. As a baseline the RMSE from fitting a cubic regression function to the exact value function is 344.73 so we would not expect any approximation algorithm to generate a better fit. To obtain a better fit one needs to change the features, For example spline functions provide an attractive alternative. The command *bs()* in R [2021] provides several options.

From Table 1.2 observe that:

1. In replicate 3 both TD(0) methods gave a worse fit than the Monte Carlo initialization that was based on one realization at 5 states. In 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 updating gave more accurate estimates.

3. Neither method gave estimates with RMSE close to that of the regression fit to the true value function.

We conclude from this example that using Algorithm 1.4 does not provide very accurate estimates of the value function. The following section considers a TD(γ) variant that might provide better estimates.

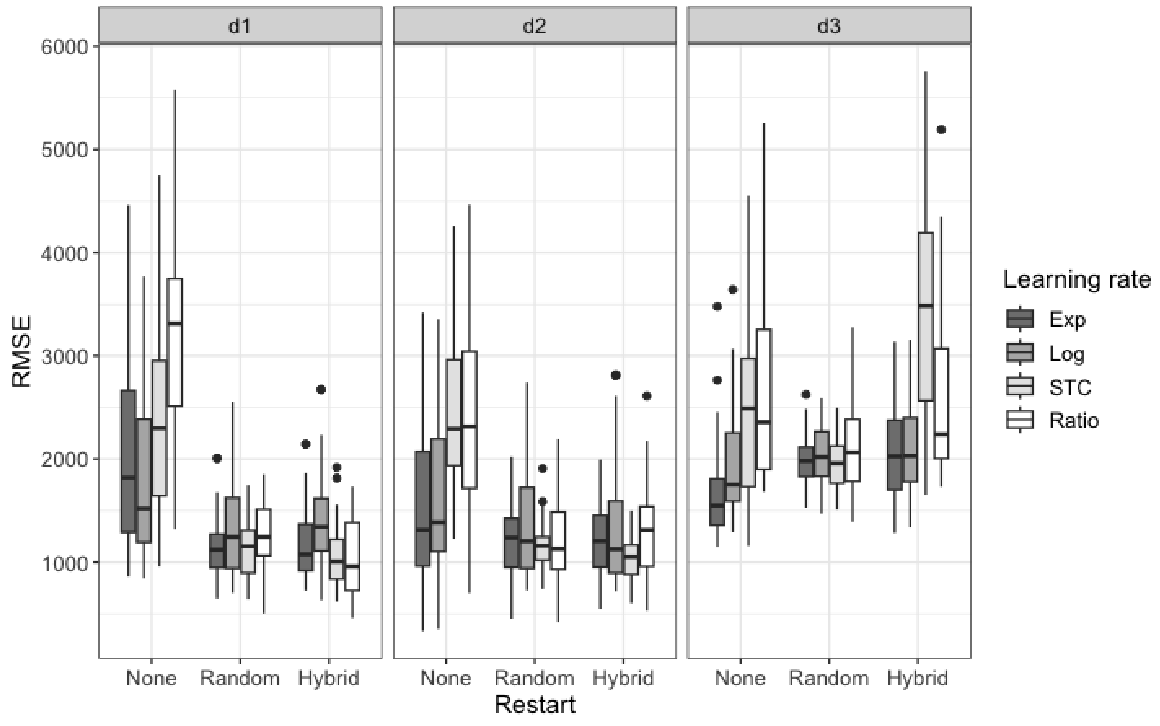


Figure 1.4: Comparison of the RMSE of value function estimates for the queuing control model in Example 1.3. As a frame of reference, Table 1.1 showed that a cubic polynomial approximation derived from Monte Carlo estimates had an average RMSE of 1029.04 for d_1 and 1098.57 for d_2. **(change Exp to Poly in plot)**

1.2.3 TD(γ) with linear value function approximation

This section generalizes TD(γ) to models with value function approximation. It provides it in algorithmic form, offers some comments and illustrates it with an example. It consider the case of linear approximations.

Algorithm 1.5. TD(γ) with linear value function approximation.

1. **Initialize:**
   (a) Specify a decision rule d, an initial vector $\boldsymbol{\beta}$ a state $s \in S$.
   (b) Specify the number of iterations N and the learning rate sequence $(\tau_n : n = 1, \ldots)$.

	Replicate				
Method	1	2	3	4	5
MC - initialization	1209	1259	794	973	2111
TD(0) - hybrid	967	1209	873	882	1620
TD(0) - random	1270	1070	1441	926	1131

Table 1.2: RMSE of fitted value functions for 5 replicates of Monte Carlo initialization and TD(0) with two state generation methods for decision rule d_1. Note that the cubic regression fit to the exact value function has RMSE equal to 344.73. Table 1.1 shows that the MC estimate with LS estimation had an average RMSE of 1029.94 over 40 replicates.

```
    (c) Set the $(K+1)$-dimensional eligibility trace vector $\mathbf{z} = \mathbf{0}$ and specify $\gamma \in [0,1]$.
    (d) $n \leftarrow 1$.
2. **Iterate:** For $n < N$:
    (a) Sample $a$ from $w_d(\cdot|s)$.
    (b) Generate $(s', r)$ or sample $s' \in S$ from $p(\cdot|s,a)$ and set $r = r(s,a,s')$.
    (c) **Compute the temporal difference:** Set $v(s;\boldsymbol{\beta}) = \mathbf{b}(s)^T\boldsymbol{\beta}$, $v(s';\boldsymbol{\beta}) = \mathbf{b}(s')^T\boldsymbol{\beta}$ and
            $\delta = r + \lambda v(s';\boldsymbol{\beta}) - v(s;\boldsymbol{\beta}).$    (1.20)
    (d) **Update eligibility trace:**
            $\mathbf{z}' = \gamma\lambda\mathbf{z} + \mathbf{b}(s)$    (1.21)
    (e) **Update $\boldsymbol{\beta}$:**
            $\boldsymbol{\beta}' \leftarrow \boldsymbol{\beta} - \tau_n \delta \mathbf{z}'$    (1.22)
    (f) $\boldsymbol{\beta} \leftarrow \boldsymbol{\beta}'$, $\mathbf{z} \leftarrow \mathbf{z}'$, $n \leftarrow n+1$, $s \leftarrow s'$.
3. Return $\boldsymbol{\beta}$.
```

Some comments follow:

1. We state the long-trajectory discounted variant of the algorithm only.
2. 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 non-linear value function approximations, the gradient of

the value-function (with respect to its parameters) replaces the feature vector in (1.21).

3. Note that when $\gamma = 0$, the algorithm reduces to TD(0).

4. We leave it as an exercise to show that this reduces to Algorithm **??** when features are indicators of states.

Example 1.4. This example investigates the application of the long-trajectory variant of TD(γ) to the queuing model analyzed in Example 1.3. That example showed the the best estimates occurred with polynomial and logarithmic learning rates. Consequently this example compares 40 replicates with common random seeds with these two learning rates and $\gamma = 0, 0.25, 0.5, 0.75$ for each of three decision rules. In each instance, N =20,000. Estimates are compared on the basis of the RMSE of the fitted values. Figure 1.5 summarizes results graphically. Observe that the effect of γ differed across decision rules:

1. For d_1, estimates using $\gamma = 0.75$ gave the most accurate estimates with those using the polynomial learning rate the least variable.

2. For d_2, the TD(0) estimates were least variable and had the lowest average[a] RMSE.

3. For d_3, the TD(0) estimates were the most accurate and least variable.

Thus in this example the benefits of using TD(γ) over TD(0) were small.

[a]Result not shown, the boxplot below shows the median RMSE.

1.3 Optimization based on value function approximation: Value iteration type algorithms

This section develops simulation-based methods in which value functions or state-action value functions are approximated by functions of features. It considers value iteration type algorithms such as Q-learning and policy iteration algorithms.

We restrict attention to discounted models. Algorithms can be easily generalized to episodic models by adding a mechanism to start the next episode after termination of the previous episode. Recall that value iteration is based on either the value function recursion

$$v(s) \leftarrow \max_{a \in A_s} \Big\{ \sum_{j \in S} p(j|s,a)(r(s,a,j) + \lambda v(j) \Big\} \tag{1.23}$$

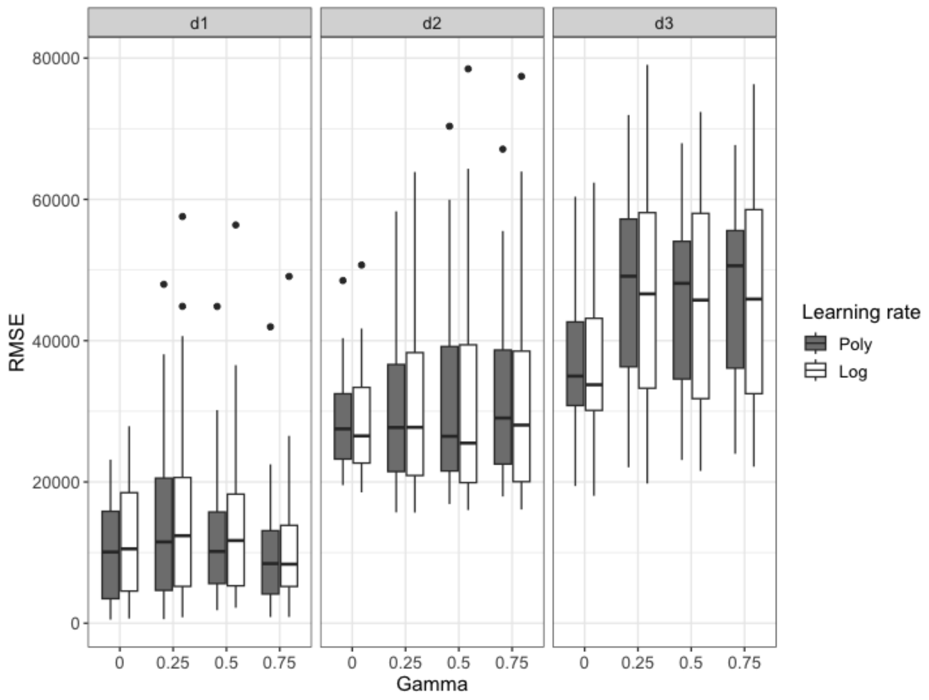


Figure 1.5: Boxplots of the RMSE of TD(γ) estimates based on 40 replicates of two learning rates and three decision rules.

or the state-action value function recursion

$$q(s,a) \leftarrow \sum_{j \in S} p(j|s,a)\big(r(s,a,j) + \lambda \max_{a \in A_s} q(j,a)\big). \tag{1.24}$$

Note that when a model is available, that is when we know $r(s,a,j)$, $p(j|s,a)$ and the optimal value-function $v^*(s)$, we can set

$$q(s,a) = \sum_{j \in S} p(j|s,a)\big(r(s,a,j) + \lambda v^*(j)\big). \tag{1.25}$$

however in model-free environments we must work directly with the state-action value function.

1.3.1 Q-learning with function approximation

(I'm not completely happy with this intro. I don't think the justification is sufficiently rigorous.) As in the case of a tabular model, Q-learning seeks to find a $\boldsymbol{\beta}$ that minimizes

$$E\Big[\Big(r(X,Y,X') + \lambda \max_{a' \in A_X} q(X',a';\boldsymbol{\beta}) - q(X,Y;\boldsymbol{\beta})\Big)^2 \Big| X = s, Y = a\Big] \tag{1.26}$$

over $\boldsymbol{\beta} \in \Re^K$ where the expectation is with respect to the distribution $p(\cdot|s,a)$.

The gradient[9] of this expression may be written as

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

which when $q(s,a)$ is approximated by a linear function of features $\mathbf{b}(s,a)$, becomes

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

Noting this expression for the gradient leads to the following recursion for estimating weights $\boldsymbol{\beta}$:

$$\boldsymbol{\beta}' \leftarrow \boldsymbol{\beta} + \tau\Big(r(s,a,s') + \lambda \max_{a \in A'_s} \boldsymbol{\beta}^T\mathbf{b}(s',a') - \boldsymbol{\beta}^T\mathbf{b}(s,a)\Big)\mathbf{b}(s,a). \tag{1.27}$$

where s' is the result of some form of sampling. Note that in the above expression the $\boldsymbol{\beta}$ and $\mathbf{b}(s,a)$ are column vectors and the expression in ()'s is a scalar.

An algorithm

The following Q-learning algorithm seeks to find the coefficients of a linear approximation to optimal state-action value function $q^*(s,a)$. Because we are implicitly assuming that the set of states is large, the output of the algorithm is a low dimension vector $\hat{\boldsymbol{\beta}}$ of weights which can be used in subsequent applications to determine action choice in state s by setting

$$\hat{d}(s) \in \arg\max_{a \in A_s} q(s,a;\hat{\boldsymbol{\beta}}). \tag{1.28}$$

Moreover the corresponding value function approximation can be obtained from

$$\hat{v}(s) = \max_{a \in A_s} q(s,a;\boldsymbol{\beta}). \tag{1.29}$$

over a range of states.

Algorithm 1.6. Q-Learning with linear state-action value function approximation in a discounted model

1. **Initialization:**
   (a) Initialize $\boldsymbol{\beta}$.
   (b) Specify the learning rate $\{\tau_n : n = 1,2,\ldots\}$ and a sequence $\{\epsilon_n\ n = 1,2,\ldots\}$ for ϵ-greedy action selection.

[9]Note that when computing this gradient, the state-action pair is regarded as fixed so that the gradient is evaluated at (s,a).

(c) Choose $s \in S$.

(d) Specify the number of iterations N and $n \leftarrow 1$.

2. **Re-evaluation** While $n < N$

   (a) Choose $a \in A_s$ ϵ_n-greedily.

   (b) Generate (s', r) or sample $s' \in S$ from $p(s'|s,a)$ and set $r = r(s,a,s')$.

   (c) Set
   $$q(s,a;\boldsymbol{\beta}) = \boldsymbol{\beta}^T \mathbf{b}(s,a) \tag{1.30}$$

   (d) Set
   $$\delta = \Big(r(s,a,s') + \lambda \max_{a' \in A_s} q(s',a';\boldsymbol{\beta}) - q(s,a;\boldsymbol{\beta}) \Big) \tag{1.31}$$

   (e) Update $\boldsymbol{\beta}$ according to
   $$\boldsymbol{\beta}' \leftarrow \boldsymbol{\beta} + \tau_n \mathbf{b}(s,a)\delta \tag{1.32}$$

   (f) $s \leftarrow s'$, $\boldsymbol{\beta} \leftarrow \boldsymbol{\beta}'$ and $n \leftarrow n+1$

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

We comment briefly on this algorithm.

1. The algorithm requires ϵ-greedy or softmax action selection to ensure exploration. Without it, the algorithm would 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.

3. As stated, the algorithm generates sequences of states and actions according to the underlying transition probabilities. Note that in step 2(f), $s \leftarrow s'$ can be replaced by "sample s from S" or the hybrid variant described previously.

4. The above algorithm never explicitly computes a decision rule. Action selection in step 2(b) implicitly corresponds to a policy but as $\boldsymbol{\beta}$ changes, the elusive policy shifts. We avoid this issue in the policy iteration algorithms below by fixing $\boldsymbol{\beta}$ for several iterations.

5. Note this is an off-policy algorithm since the update in (1.36) may be based on a different action than that followed by the trajectory. A SARSA variant, as in Chapter **??**, would provide an on-policy alternative.

6. Unlike its tabular counterpart, that corresponds to using state-action indicators as basis functions, there are no general convergence guarantees for Q-learning with function approximation.

7. The algorithm can be easily modified to allow for non-linear state-action value function approximations. This would require replacing 2(d) to allow for a non-linear functional form and replacing $\mathbf{b}(s, a)$ in 2(f) by the gradient of the non-linear function.

1.3.2 Q-learning in a discounted model

We now apply Q-learning to the infinite horizon discounted queuing service rate control model.

Example 1.5. This example again considers the queuing service rate control model on $S = \{0, \dots, 50\}$. Basis functions are of the form (1.3) where the terms in $f_k(s)$ equals the scaled powers of a cubic polynomial and the terms in $h_j(a)$ represent indicator functions of each possible action. This is a equivalent to representing $q(s, a)$ by a cubic polynomial in which the weights vary with the action.

As seen in Example 1.3, good starting values were required to ensure convergence. By using the above specification for the approximation, preliminary weight estimates for the cubic polynomial approximation can be obtained using Monte Carlo for each action separately or alternative randomizing over all actions with equal probability. Note that these differ from the q-functions sought by the algorithm because after choosing a state and action, they follow the value function of the specified policy, not the optimal value function.

The algorithm required considerable tuning to obtain convergence to reasonable policies. Results are described in the case N =250,000, learning rate

$$\tau_n = \frac{5000}{50000 + n}$$

and greedy parameter

$$\epsilon_n = \frac{100}{400 + n}.$$

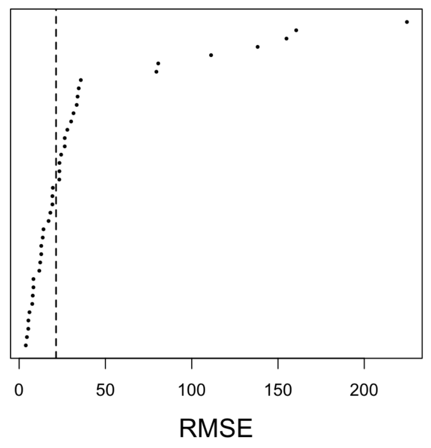
The indices of the learning rate and ϵ-greedy parameter were the iteration number.

Figure 1.6 summarizes output from 40 replicates on the basis of the RMSE of the value function of the greedy policy relative to the optimal value function, and the number of states at which the greedy policy differed from the optimal policy.

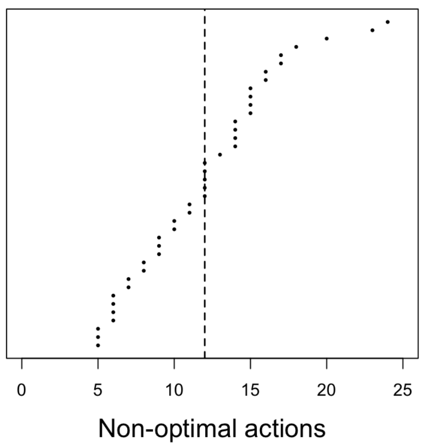
Figure 1.6a show that in 33 out of the 40 replicates the RMSE was less than 35.7 with a median of 21.4. To put this in context the the value function range was (221, 39,441) indicating a high degree of accuracy.

In spite of this, the greedy policy differed from the optimal policy in all replicates (Figure 1.6b). In 35 out of 40 replicates the greedy policy had the same structure as the optimal policy however the points at which actions changed differed. The median number of non-optimal actions was 12.

Consequently from the perspective of RMSE, the results were satisfactory however this approach failed to identify the optimal policy in all cases. Changing the starting values for the algorithm and basis functions did not improve the accuracy of the greedy-policy. We leave it to the reader to try to improve on these results.



(a) Plot of sorted RMSEs. Vertical dashed line denotes median.



(b) Plot of sorted number of non-optimal actions. Vertical dashed line denotes median.

Figure 1.6: Summary order statistics of the RMSE and number of non-optimal actions in 40 replicates obtained in Example 1.5.

1.3.3 Q-learning in an episodic model

This simple example applies Q-learning to a simple 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: $S = \{(i,j) : i = 1,\ldots,M, j = 1,\ldots,N\}$

Actions: For all $s = (i,j) \in S$, $A_s = A$

$$A = \begin{cases} \{\text{up, down, left, right}\} & (i,j) \neq (m^*, n^*) \\ \{\delta\} & (i,j) = (m^*, n^*) \end{cases}$$

Rewards:

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

Transition probabilities:

$$p((i',j')|(i,j),a) = \begin{cases} 1 & \text{if move from } (i,j) \text{ to } (i',j') \text{ is possible under action } a \\ 0 & \text{otherwise.} \end{cases}$$

Note that the transition probabilities are awkward to write down explicitly because there are many cases to enumerate[10], 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 exists (many) deterministic optimal policies that can be easily found by inspection. Although exact methods (from Chapter **??** converge, we found that it was necessary to introduce a discount factor to ensure reliable convergence.

Feature choice

This example compared three implementations of Q-learning differing by the choice of the form of the approximate state-action value function. All were based on representation (1.4) in which $h_j(a)$ was 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 on the form of features $f_k(i,j)$ as follows:

1. **Indicator functions of each cell (the tabular model):**

$$f_k(i',j') = I_{\{i,j\}}(i',j')$$

for each $(i,j) \in S$. For each action there are $M \times N$ features. In codes, it might be more convenient to index $f_k(i',j')$ by the row and column index.

[10]For example if the agent is at the left boundary, that is in state $(i,1)$, and tries to move left, it remains in state $(i,1)$ but if it tries, for example, to move right, a transition to $(i,2)$ occurs.

2. **Indicator functions of each row and column:**

$$f_k(i',j') = \begin{cases} I_{\{i\}}(i',j') & \text{for } i = 1,\dots,M \text{ and } k = 1,\dots,M, \\ I_{\{j\}}(i',j') & \text{for } j = 1,\dots,N \text{ and } k = N+1,\dots,N+M. \end{cases}$$

   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 calculations in the tabular case (described below) we hypothesized that a linear representation with an interaction term would well approximate the true value function.

$$f_0(i,j) := 1, \quad f_1(i,j) := j, \quad f_2(i,j) := j, \quad f_3(i,j) := ij.$$

   This means that for each action, $q(s,a;\boldsymbol{\beta})$ is approximated by a linear function of $(1, r, c, rc)$ 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 judicious to scale the row and column index.

Results: Tabular model

Calculations for the tabular model apply Algorithm 1.6 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 each the starting cell $(1, 1)$.

It uses ϵ-greedy action selection with $\epsilon_n = 10/(100 + n)$ and a learning rate of $\tau_n = 50/(1000+n)$ where n denotes the episode number. After completion of an episode the next episode again started from cell $(1, 1)$[11]. The experiments used a discount rate of $\lambda = 1$, 10,000 episodes and set all values $q((i,j),a) = 0$ for initialization.

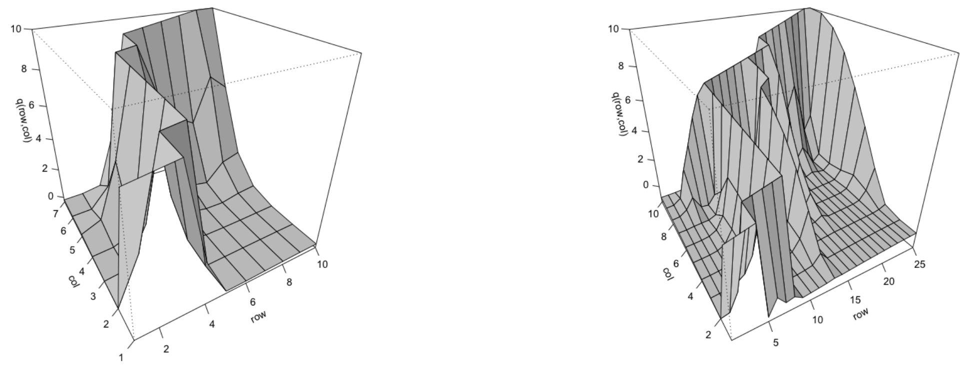
For small values of M and N, it was convenient to use the tabular representation of Q-learning to gain some insight into the form of the q-functions. For all replications of the experiment, using Q-learning in the tabular model found an optimal path through the grid.

Figure 1.7 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 that moved close to the main diagonal of the grid. This suggested that the parametric model above with a cross-product term might identify optimal policies.

Results: Indicator functions of rows and columns

We replicated the above calculations using indicator functions of grid row and grid column using the same choice for ϵ_n and learning rate τ_n and the two grid sizes. To reliably find an optimal policy required discounting; $(\lambda = 0.95)$ gave reliable results.

[11]If one sought a good policy for every starting state, episodes could be restarted at a random cell not equal to the target.



(a) 10 × 7 grid (b) 25 × 10 grid

Figure 1.7: Optimal q-values in grid-world model identified by Q-learning using a tabular representation.

Unlike the policies generated using the tabular representation, which tended to move down 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 that 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[12] used:

1. $\boldsymbol{\beta}$ initialized to be $\mathbf{0}$,
2. high initial exploration rates ($\epsilon_n = 20/(20+n)$),
3. an upper bound (1000) on the number of steps in an episode,
4. small learning rates ($\tau_n = 50/(1000+n)$),
5. a discount factor of 0.9, and
6. random starting states.

[12]The literature also suggests using experience replay to enhance convergence.

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 down through the interior of the region. They exhibited fast learning as shown in Figure 1.8.

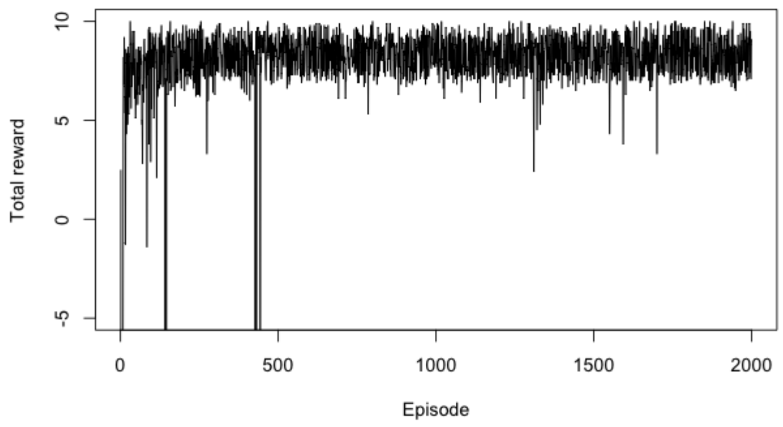


Figure 1.8: Reward per episode in a typical realization of Q-learning in a 10×25 grid. The eventual variability of the total reward per episode is due to the random starting state.

The beauty of using this parameterization was that it was not required to know the dimension of the grid before implementing Q-learning.

Summary

The above approximations used MN, $M + N$ and 4 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.

We leave it to the reader to explore parametric approximations based on other functional forms or neural networks.

1.4 Value-based policy iteration

Policy iteration algorithms require methods for both evaluating and improving stationary policies. This presents challenges in large models which in which policies are represented implicitly through vectors of weights.

It is more expedient to focus on state-action value functions than value functions to avoid determining an action in state s on the basis of

$$a' \in \arg\max_{a \in A_s} \left\{ \sum_{j \in S} p(j|s,a) \Big(r(s,a,j) + \lambda v(j; \hat{\boldsymbol{\beta}}) \Big) \right\}.$$

Evaluating such an expression would be prohibitive in a large model, even if both $p(j|s,a)$ and $r(s,a,j)$ were known.

We now describe a policy iteration type algorithm to achieve this.

1.4.1 Q-policy iteration

The following algorithm, which we refer to as *Q-policy iteration*[13] generalizes Q-learning with function approximation by running a policy that is close to a fixed implicitly-defined policy for several iterations before updating the weight vector. Essentially, it evaluates an ϵ-greedy variant [14] of the policy implicitly defined by the "current" weight vector to enable exploration. It may also be viewed as an extension of its tabular counterpart in Section **??** to an environment with state-action value function approximation.

Given a weight vector $\boldsymbol{\beta}$ and an $\epsilon \in (0,1)$, define the Markovian randomized decision rule d_0^ϵ by

$$w_{d_0^\epsilon}(a|s) = \begin{cases} 1-\epsilon & a = a_s \text{ where } a_s \in \arg\max_{a \in A_s} q(s,a;\boldsymbol{\beta}_0) \\ \frac{\epsilon}{|S|-1} & a \neq a_s. \end{cases} \tag{1.33}$$

We refer to the stationary randomized policy $(d^\epsilon)^\infty$ as the ϵ-greedy policy corresponding to $\boldsymbol{\beta}$. This policy can be implemented in a simulation by selecting action a_s with probability $1-\epsilon$ and sampling among other actions with probability ϵ.

The algorithm may be viewed as a simulation based version of modified[15] policy iteration (Algorithm **??**). We state the algorithm from the perspective of implicit policy specification in terms of a weight function and discuss why this should be regarded as a policy iteration type algorithm.

Algorithm 1.7. Q-policy iteration with linear state-action value function approximation in a discounted model

1. **Initialization:**

    (a) Initialize $\boldsymbol{\beta}_0$.

[13]This algorithm hasn't been widely discussed but seems to be a logical extension of policy iteration, see p.338 in Bertsekas and Tsitsiklis [1996] for a related discussion.

[14]Tabular models required the concept of ϵ-altered policies to account for the possibility that a decision rule being evaluated is not chosen greedily.

[15]Note that some authors, most notably, Bertseksas [2012] refer to this as *optimistic* policy iteration.

```
    (b) Specify a learning rate sequence $\{\tau_n\}$ and a sequence $\{\epsilon_n\}$ for $\epsilon$-altered action selection.
    (c) Specify the number of evaluation iterations $M_E$ and the number of evaluation loops N.
    (d) Choose $s \in S$ and $n \leftarrow 1$.
2. While $n < N$.
3. **Policy Evaluation for fixed** $\beta_0$**:** While $m < M_E$;
    (a) $m \leftarrow 1$ and $\boldsymbol{\beta} \leftarrow \boldsymbol{\beta}_0$
    (b) Generate $a \in A_s$ $\epsilon_n$-greedily.
    (c) Generate $(s', r)$ or sample $s' \in S$ from $p(\cdot|s,a)$ and set $r = r(s,a,s')$.
    (d) Set
            $q(s,a;\boldsymbol{\beta}_0) = \boldsymbol{\beta}_0^T \mathbf{b}(s,a)$    (1.34)
    (e) Generate $a' \in A_{s'}$ $\epsilon_n$-greedily.
    (f) Set
            $q(s',a';\boldsymbol{\beta}_0) = \boldsymbol{\beta}_0^T \mathbf{b}(s',a')$    (1.35)
    (g) Update $\boldsymbol{\beta}$ according to
            $\boldsymbol{\beta}' \leftarrow \boldsymbol{\beta} + \tau_n \mathbf{b}(s,a)\delta$    (1.36)
    (h) $s \leftarrow s'$, $\boldsymbol{\beta} \leftarrow \boldsymbol{\beta}'$ and $m \leftarrow m+1$
4. **Weight updating:**
    (a) $\boldsymbol{\beta}_0 \leftarrow \boldsymbol{\beta}$.
    (b) $n \leftarrow n+1$.
5. **Termination:** Output $\hat{\boldsymbol{\beta}} = \boldsymbol{\beta}$.
```

On the surface this does not look like a policy iteration algorithm since there is no clear improvement step. The algorithm as stated embeds improvement in the implicit generation of a new decision rule in steps 2(b) and 2(c) using fixed weights $\boldsymbol{\beta}_0$ updated after M_E evaluation iterations.

Suppose instead the algorithm starts with a decision rule d rather than a vector of weights $\boldsymbol{\beta}_0$. Then the policy evaluation step approximates an ϵ-greedy variant of that decision rule , d_ϵ, yielding a vector of weights $\boldsymbol{\beta}_{d_\epsilon}$. Then a full improvement step would choose

$$d'(s) \in \arg\max_{a \in A_s} \boldsymbol{\beta}_{d_\epsilon}^T \mathbf{b}(s,a)$$

for **all** $s \in S$ and then return to evaluating the ϵ-altered variant of this new decision rule.

Some further comments follow:

1. When $M_E = 1$, this algorithm is equivalent to a SARSA variant of Q-learning. When $M_E > 1$ the evaluation step may be viewed as a TD(0) approximation to the state-action value function of the ϵ-greedy policy corresponding to $\boldsymbol{\beta}_0$.
2. No explicit stopping criterion is specified. One might terminate the algorithm when successive weight vectors differ by a small amount.
3. The algorithm is stated assuming long trajectory updates. Of course step 3(k) can update s in a different way to enhance coverage of the state space.
4. Softmax action selection can replace ϵ-greedy action selection in steps 3(b) and 3(f).
5. The reason ϵ-greedy policies are used in 3(c) and 3(f) are to obtain off-policy realizations of $q(s, a; \boldsymbol{\beta}_0)$.
6. The policy evaluation step can be replaced by Monte Carlo estimation of the ϵ-greedy policy.
7. Similarly to Q-learning, there is no guarantee of convergence of this algorithm.

The newsvendor model with approximate value functions

This section applies

1. Monte Carlo estimation
2. Q-learning
3. Q-policy iteration

to the newsvendor inventory model introduced in Section **??**. The reason why we consider this extremely simple model is that by removing the effect of state transitions we can distinguish the subtle differences between Q-learning and Q-policy iteration.

Since this model has a single state $s = 0$, the state-action value function $q(s, a)$ is a function **only** of the action a. We use a cubic polynomial approximation expressed in terms of scaled actions as follows

$$b_0(a) = 1, b_1(a) = \tilde{a},\ b_2(a) = \tilde{a}^2 \text{ and } b_3(a) = \tilde{a}^3$$

where $\tilde{a}$ represents a scaled by subtracting its mean and standard deviation[16] so that

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

[16]This means $\tilde{a} = (a - mean(a))/sd(a)$ where the mean and standard deviation are over $\{0, 1, \ldots, a_{max}\}$.

To implement Monte Carlo, for each element in a subset of $a \in A_0$, sample the demand N times to obtain $z^0, \ldots, z^N$ and set $\hat{q}(a)$ equal to the mean of $r(a, z^1), \ldots, r(a, z^N)$[17]. Then use least-squares to approximate $\hat{q}(a)$ by a linear function of its features to obtain $\hat{\boldsymbol{\beta}}_{MC}$.

The following algorithms apply Q-learning and Q-policy iteration apply directly to the newsvendor model.

Q-learning in the newsvendor model:

```
1. Initialize $\boldsymbol{\beta}$ and $n \leftarrow 1$.
2. Specify sequences $\{\epsilon_n : n = 1, 2, \ldots\}$ and $\{\tau_n : n = 1, 2, , \ldots)\}$.
3. Repeat N times:
    (a) Sample $a'$ using $\epsilon$-greedy (or softmax) sampling with respect to
            $q(a; \boldsymbol{\beta}) = \boldsymbol{\beta}^T \mathbf{b}(a)$
    (b) Generate demand $z^n$.
    (c) Update
            $\boldsymbol{\beta}' \leftarrow \boldsymbol{\beta} + \tau_n (r(a', z^n) - \boldsymbol{\beta}^T \mathbf{b}(a)) \mathbf{b}(a')$    (1.37)
    (d) $\boldsymbol{\beta} \leftarrow \boldsymbol{\beta}'$ and $n \leftarrow n + 1$.
4. Return $\hat{\boldsymbol{\beta}}_{QL} = \boldsymbol{\beta}$.
```

[17]Note the same realization of δ can be used for all a.

Q-policy iteration in the newsvendor model:

```
1. Initialize $\boldsymbol{\beta}_0$ and $n \leftarrow 1$.
2. Specify sequences $(\epsilon_n : n = 1, 2, \ldots)$ and $(\tau_n : n = 1, 2, , \ldots)$.
3. Repeat N times:
   (a) $m \leftarrow 1$
   (b) Repeat M times:
       i. Sample $a'$ using $\epsilon$-greedy (or softmax) sampling with respect to
              $q(a) = \boldsymbol{\beta}_0^T \mathbf{b}(a)$
       ii. Generate $z^m$.
       iii. Update
              $\boldsymbol{\beta}' \leftarrow \boldsymbol{\beta} + \tau_m (r(a', z^n) - \boldsymbol{\beta}^T \mathbf{b}(a'))\mathbf{b}(a')$
       iv. $\boldsymbol{\beta} \leftarrow \boldsymbol{\beta}'$ and $m \leftarrow m + 1$.
   (c) $\boldsymbol{\beta}_0 \leftarrow \boldsymbol{\beta}$ and $n \leftarrow n + 1$.
4. Return $\hat{\boldsymbol{\beta}}_{QPI}$.
```

While these algorithms appear very 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 $\max_{a \in A_0} E[r(Z, a)]$ so that the continuation cost is zero.

2. In Q-learning, action selection in step 2(a) is respect to a different $\boldsymbol{\beta}$ at each iteration, moreover this $\boldsymbol{\beta}$ is also updated in (1.37). 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 used in the update equation. This is because the inner loop seeks to evaluate a fixed policy based on the ϵ-altered decision rule corresponding $\boldsymbol{\beta}_0$.

3. When $M = 1$ these algorithms are equivalent.

Example 1.6. This example considers the newsvendor problem with two discrete demand distributions (rounded and truncated normal 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 $\frac{G}{G+L}$ were $0.0625, 0.400, 0.870$ which represent a wide range of quantiles of the demand distributions. The order quantities were $A_0 = \{0, 1, \ldots, 120\}$.

To obtain a baseline we generate 10,000 Monte Carlo replicates for each $a \in A_0$ and estimate the expected reward to be the average reward over the samples. Table 1.3 gives the optimal order quantity using the closed form representation (**??**), the maximum of the Monte Carlo estimates and the maximum obtained when fitting the Monte Carlo estimates with a cubic polynomial and a cubic spline with knots at 40 and 80.

Observe that order quantity that maximizes the Monte Carlo estimates well approximates the optimal order quantity. Observe also the maximum obtained from the spline approximation better approximates the Monte Carlo maximum than that based on a cubic approximation. We would expect this to be the case because the spline is a more flexible function with more parameters than a cubic polynomial.

We then applied Q-learning and Q-policy iteration using a cubic approximation based on scaled values of a. In each case the algorithm was initiated with $\boldsymbol{\beta} = \mathbf{0}$, used the STC learning rate $\tau_n = .05(1 + 10^{-5}n^2)^{-1}$ and a step-wise exploration rate ϵ_n that equaled 0.2 for $n < 10{,}000$, 0.15 for $n \geq 10{,}000$. We 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. We ran 40 replicates of each algorithm for each combination of demand distribution and price using common random seeds.

Figure 1.9 shows the variability of estimates of the optimal order quantity over replicates obtained by choosing

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

for $i = QL, QPI$. Comparison with Table 1.3 shows that the Q-policy iteration better approximated the optima obtained using a cubic approximation to the Monte Carlo estimates then the Q-learning estimates in **all** cases.
This example may be viewed as a learning experiment since no information was used regarding the underlying demand distribution or reward structure.

Q-policy iteration in the queuing service rate control model

We now apply Q-policy iteration to the queuing service rate control model.

Example 1.7. This example applies Algorithm 1.7 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.

Distribution	Price	Optimal	Monte Carlo	Cubic	Spline
	21	27	26	23	29
Normal	30	46	45	41	44
	120	67	65	76	69
	21	55	53	47	54
Gamma	30	74	76	76	75
	120	100	99	101	99

Table 1.3: Optimal order quantity in newsvendor model compared to three estimates obtained from 10,000 Monte Carlo replicates for each $a \in A_0$. The Monte Carlo estimate is is the maximum over the mean reward for each action while the Cubic and Spline estimates are the maximums of the least squares estimates using these approximations.

To obtain reasonable policy estimates required considerable experimentation with the number of evaluation iterations M_E and the learning rate and ϵ-greedy parameters. Table 1.4 summarizes results for $M_E = 50$, $N = 5{,}000$, $\tau_n = 5000/(50000 + n)$ and $\epsilon_n = 100/(400 + n)$ where the index for the parameters was the cumulative iteration number. Note that the total number of iterations was the same as in the experiment with Q-learning. States were sampled randomly at each iteration.

Table 1.4 summarizes results of the estimates and compares them to those obtained using Q-learning. Observe that the results using Q-policy iteration were quite similar to those obtained using Q-learning. Moreover the plots of order statistics (not shown) were similar to those in Figure 1.6.

Method	Quantity	min	25th pctl	median	75th pctl	max
Q-Policy iteration	Non-optimal actions	1	8	10.5	12.25	40
	RMSE	0.09	9.28	43.4	49.42	243.43
Q-learning	Non-optimal actions	5	8	12	15	24
	RMSE	3.86	12.07	21.39	33.48	224.70

Table 1.4: Summary statistics of 40 replicates of Q-policy iteration applied to the queuing control model.

Concluding remarks on examples

These numerical studies show that in the queuing control model, both Q-learning and Q-policy iteration produced similar results with Q-policy iteration slightly better at identifying a policy in agreement with the optimal policy. In the newsvendor model,

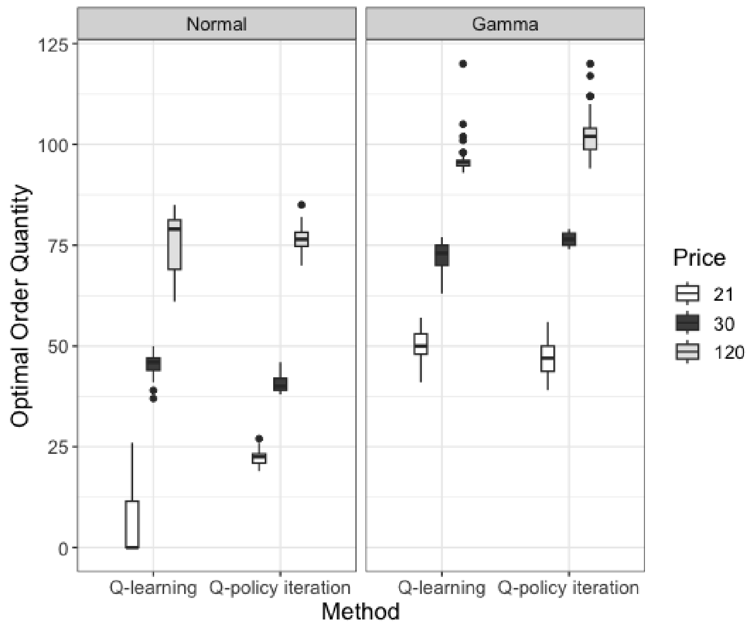


Figure 1.9: Boxplot comparing actions chosen by Q-learning and Q-policy iteration in the newsvendor model over 40 replicates for each demand distribution and price.

Q-policy iteration was better than Q-learning in identify a policy that agreed with that found using a cubic approximation to Monte Carlo estimates.

While these results were inconclusive, we believe that both Q-learning and Q-policy iteration provide the analyst with tools for analyzing models that use function approximation.

(Need to index from here on)

1.5 Policy space methods

This 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, this approach seeks to find a choice of parameter values that improves the policy directly.

It is advantage is that unlike state-action value focused methods (such as Q-learning) which may jump between deterministic policies, it allows for gradual changes in action selection probability leading to greater stability in values. This is especially attractive when controlling physical systems in real-time.

Two concepts underlie these methods:

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 $q(s,a)$ where s and a are inputs and the value is an output, but when approximating policies directly, the state s is the input and the probability of choosing action a is an output.

2. *Updating parameter values using (stochastic) gradient ascent.* This approach seeks to iteratively find a zero of the gradient of a value function[18] by stochastic approximation where estimates are obtained by sampling from a distribution with a specified expectation.

1.5.1 Motivation: A policy gradient algorithm for a simple one-period model

We first analyze a one period, single-state multi-action model such as the newsvendor problem (Example 1.6). Since this model doesn't include state-transitions, it is similar to a classical stochastic optimization model. We retain the same notation for approximation as used when approximating state-action value functions although as noted above, these are two conceptually different approaches. That is, we represent features by vectors $\mathbf{b}(s,a)$ and parameters by compatible vectors[19] $\boldsymbol{\beta}$. In the single-state model, we ignore the dependence on s and write $\mathbf{b}(a)$ for the vector of features corresponding to action a.

We represent policies in terms of the softmax[20]function. For single-state models:

$$w(a|\boldsymbol{\beta}) = \frac{e^{\boldsymbol{\beta}^T \mathbf{b}(a)}}{\sum_{a' \in A_s} e^{\boldsymbol{\beta}^T \mathbf{b}(a')}} \tag{1.38}$$

where $\boldsymbol{\beta}$ is a K-dimensional column vector of parameters and $\mathbf{b}(a)$ is a K-dimensional column vector of features associated with action a. In multi-state models (Markov decision processes),

$$w(a|s;\boldsymbol{\beta}) = \frac{e^{\boldsymbol{\beta}^T \mathbf{b}(s,a)}}{\sum_{a' \in A_s} e^{\boldsymbol{\beta}^T \mathbf{b}(s,a')}}. \tag{1.39}$$

The objective in the single state ($S = \{0\}$) model can be represented by

$$v^* := \max_{\boldsymbol{\beta} \in \Re^K} v(\boldsymbol{\beta}) = \max_{\boldsymbol{\beta} \in \Re^K} \sum_{a \in A_0} w(a|\boldsymbol{\beta}) r(a) = \max_{\boldsymbol{\beta} \in \Re^K} \sum_{a \in A_0} \frac{e^{\boldsymbol{\beta}^T \mathbf{b}(a)}}{\sum_{a' \in A_s} e^{\boldsymbol{\beta}^T \mathbf{b}(a')}} r(a) \tag{1.40}$$

where $r(a)$ denotes the (expected) reward for choosing action a, $w(a|\boldsymbol{\beta}) := w(a|0;\boldsymbol{\beta})$ and

$$v(\boldsymbol{\beta}) := \sum_{a \in A_0} w(a|\boldsymbol{\beta}) r(a) := E_{\boldsymbol{\beta}}[r(Y)]$$

[18]Of course, the value function varies over states. To obtain a single value, one can use a weighted average over states.

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

[20]This is also referred to as a *logistic* representation.

where Y denotes the random action selected by $w(\cdot|\boldsymbol{\beta})$.

We seek to maximize this expression by solving:

$$\nabla_{\boldsymbol{\beta}} v(\boldsymbol{\beta}) = \mathbf{0}. \tag{1.41}$$

In the newsvendor model, $r(a) = E[r(a,\delta)] = \sum_{k=0}^{\Delta} r(k)f(k)$ where $f(k) = P[\delta = k]$. In this simple model we can compute $v(\boldsymbol{\beta})$ analytically, numerically or through simulation. To simulate this model requires sampling from (both) $w(\cdot|\boldsymbol{\beta})$ and the demand distribution $f(\cdot)$.

The gradient of $v(\boldsymbol{\beta})$ in the newsvendor model.

We now develop the machinery to apply the approach in Figure ??. To do so, we derive a representation for the expected gradient of $v(\boldsymbol{\beta})$.

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

$$\begin{aligned}\nabla_{\boldsymbol{\beta}} v(\boldsymbol{\beta}) &= \sum_{a\in A_0} \nabla_{\boldsymbol{\beta}} w(a|\boldsymbol{\beta}) r(a) \\ &= \sum_{a\in A_0} w(a|\boldsymbol{\beta}) \nabla_{\boldsymbol{\beta}} \ln(w(a|\boldsymbol{\beta})) r(a) = E_{\boldsymbol{\beta}}[\nabla_{\boldsymbol{\beta}} \ln[w(Y|\boldsymbol{\beta})] r(Y)]. \end{aligned} \tag{1.42}$$

where the second equality follows from the generalizing the easily demonstrated calculus identity

$$\frac{dg(x)}{dx} = g(x)\frac{d\ln(g(x))}{dx}. \tag{1.43}$$

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, and
2. in multi-period models, expectations are based on products of probabilities so that logarithms transform a product to an easier to work with sum [21].

We leave it as an exercise that when $w(a|\boldsymbol{\beta})$ is defined by (1.38),

$$\nabla_{\boldsymbol{\beta}} \ln(w(a|\boldsymbol{\beta})) = \mathbf{b}(a) - \sum_{a'\in A_0} w(a'|\boldsymbol{\beta})\mathbf{b}(a') \tag{1.44}$$

where as above, $\mathbf{b}(a)$ is a vector of feature values. Thus we have the following closed-form expression for the gradient of $v(\boldsymbol{\beta})$ in the newsvendor model;

$$\nabla_{\boldsymbol{\beta}} v(\boldsymbol{\beta}) = \sum_{a\in A_0} w(a|\boldsymbol{\beta})\left[\mathbf{b}(a) - \sum_{a'\in A_0} w(a'|\boldsymbol{\beta})\mathbf{b}(a')\right] r(a) = E_{\boldsymbol{\beta}}[(\mathbf{b}(Y) - C)r(Y)] \tag{1.45}$$

where $C = \sum_{a'\in A_0} w(a'|\boldsymbol{\beta})\mathbf{b}(a')$.

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

Applying stochastic approximation

This section applies stochastic gradient ascent[22] to solve $\nabla_{\boldsymbol{\beta}} v(\boldsymbol{\beta}) = \mathbf{0}$ by sampling from a distribution with expected value $E_{\boldsymbol{\beta}}[(\mathbf{b}(Y) - C)r(Y)]$. Action a is sampled from $w(\cdot|\boldsymbol{\beta})$, a sample from the distribution $f(\cdot)$ is used to estimate $r(a)$ and the gradient is estimated by $\nabla_{\boldsymbol{\beta}} ln[w(a|\boldsymbol{\beta})]r(a)$.

The stochastic approximation recursion becomes:

$$\boldsymbol{\beta}' \leftarrow \boldsymbol{\beta} + \tau \nabla_{\boldsymbol{\beta}} \ln[w(a|\boldsymbol{\beta})]r(a) \tag{1.46}$$

where the gradient is computed using (1.45) (or by numerical differentiation). This observation leads to the following policy gradient algorithm for a one-period model (assuming that simulation of both actions and rewards):

Algorithm 1.8. Policy gradient algorithm for a single-state, one-period model:

```
1. Specify $\boldsymbol{\beta}$, $\{\tau_n : n = 1, 2, \ldots\}$ and $N$.
2. $n \leftarrow 1$
3. While $n < N$:
   (a) Sample action $a$ from $w(a|\boldsymbol{\beta})$.
   (b) Sample reward $r$ from $f(\cdot)$.
   (c) Estimate gradient:
          $\nabla_{\boldsymbol{\beta}} v(\boldsymbol{\beta}) = \Big(\mathbf{b}(a) - \sum_{a' \in A_0} w(a'|\boldsymbol{\beta})\mathbf{b}(a')\Big) r$      (1.47)
   (d) Update:
          $\boldsymbol{\beta} \leftarrow \boldsymbol{\beta} + \tau_n \nabla_{\boldsymbol{\beta}} v(\boldsymbol{\beta})$      (1.48)
   (e) $n \leftarrow n + 1$
4. Return $w(a|\boldsymbol{\beta})$.
```

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.

[22]We use the expression ”ascent” since we are maximizing.

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 a stationary randomized policy based on $w(a|\boldsymbol{\beta})$.

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.

An example

We now apply this algorithm to the newsvendor model. We take as features, indicator variables of actions:

$$b_i(a) = \begin{cases} 1 & a = a_i \\ 0 & a \neq a_i \end{cases}$$

for $i = \{0, 1, \ldots, a_{max}\}$ resulting in $a_{max} + 1$ features. Note this choice of features is equivalent to using a tabular representation. (We leave it to the reader to investigate other feature choices.)

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 $\boldsymbol{\beta} = [0, 0, 0, 0]^T$ so that $w(a|\boldsymbol{\beta}) = 0.25$ for $a \in \{0, 1, 2, 3\}$.

Suppose sampling generates action $a = 1$ and $r(a) = 17$ so that applying (1.47) 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(b) of Algorithm 1.8 the component of $\boldsymbol{\beta}$ corresponding to $a = 1$, $\phi_1' > \phi_1$ and all other components of $\boldsymbol{\beta}'$ will be smaller than the corresponding component of $\boldsymbol{\beta}$. Hence the policy $w(a|\boldsymbol{\beta}')$ 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.

We now apply this algorithm to some numerical instances.

Example 1.8. Policy gradient in the newsvendor model.

We set $a_{max} = 20$, cost equal to 20, salvage value equal to 5 and vary the price between $21, 30, 120$. We consider three demand distributions: truncated discrete normal with mean 10 and standard deviation 3, binomial with $N = 10$ and $p = 0.6$ and discrete uniform on $\{0, \dots, a_{max}\}$. We apply Algorithm 1.8 with N =10,000 and $\tau_n = 0.0001$ to each of 40 replicates for all nine combinations of price and demand distribution.

We observed that in **all** replicates $w(a|\boldsymbol{\beta})$ evaluated at the estimated $\boldsymbol{\beta}$ was unimodal with a single probability extremely close to 1 and all others near 0. This means that the algorithm converged to a deterministic policy.

Table 1.5 summarizes results. The column "Mean" denotes the average of $\arg\max_{a \in A_0} w(a|\boldsymbol{\beta})$ and the column "S.D." denotes the standard deviation of this quantity over 40 replicates. Observe that in all cases, the average over replicates well approximated the optimal value computed using (**??**). Estimates were most variable when price equalled 120 corresponding to the 87th percentile of the demand distribution.

Distribution	Price	Optimal	Mean	S. D.
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 1.5: Mean and standard deviation over 40 replicates of order quantity corresponding to maximum estimated probability obtained using Algorithm 1.8 . Note the optimal order quantity for the normal and uniform distributions is based on continuous representations of these distributions.

1.5.2 A policy gradient algorithm for a Markov decision process

We now use the intuition and results in the previous section to develop 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 **??**) that the expectation is respect to the probability distribution of sequences of states and actions generated by the stochastic process corresponding to π. 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

$$v^\pi = \sum_{s \in S} \rho(s) v^\pi(s). \tag{1.49}$$

As a result of Theorem **??** under mild conditions there is a stationary deterministic policy that maximizes (1.49). But instead of focusing on deterministic policies the methods herein operate within the larger family of randomized stationary policies. Recall that that a randomized stationary policy d^∞ chooses actions according to distribution $w_d(a|s)$.

As done previously in this chapter, let $\mathbf{b}(s, a)$ denote a K-dimensional state and action-dependent feature vector with corresponding K-dimensional weight vector $\boldsymbol{\beta}$ and let $w(a|s, \boldsymbol{\beta})$ denote the corresponding parameterized action-choice probability distribution. and let d_β represents 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 $\boldsymbol{\beta} \in \Re^K$ that maximizes

$$v(\boldsymbol{\beta}) := \sum_{s \in S} \rho(s) v^{(d_{\boldsymbol{\beta}})^\infty}(s) \tag{1.50}$$

or expressed differently, we seek

$$\hat{\boldsymbol{\beta}} \in \arg\max_{\boldsymbol{\beta} \in \Re^K} v(\boldsymbol{\beta}). \tag{1.51}$$

1.5.3 The gradient of the value function for an undiscounted infinite horizon Markov decision process

Chapter **??** 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

$$v(\boldsymbol{\beta}) = \sum_{s^1 \in S} \sum_{a^1 \in A_{s^1}} \sum_{s^2 \in S} \rho(s^1) w(a^1|s^1, \boldsymbol{\beta}) p(s^2|s^1, a^1) \Bigg(r(s^1, a^1, s^2) \tag{1.52}$$
$$+ \sum_{a^2 \in A_{s^2}} \sum_{s^3 \in S} w(a^2|s^2, \boldsymbol{\beta}) p(s^3|s^2, a^2) \big(r(s^2, a^2, s^3) + \ldots \big) \Bigg)$$

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

Now consider the expected reward in the first period only

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

Then

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

Observe that this gradient involves both the initial distribution and the transition probabilities and is not amenable (especially higher order terms) to direct computation or simulation. Instead using the "trick"

$$\frac{du(x)}{dx} = u(x) \frac{d \ln(u)}{dx} \tag{1.53}$$

yields

$$\begin{aligned} \nabla_\beta v_1(\boldsymbol{\beta}) = \sum_{s^1 \in S} \sum_{a^1 \in A_{s^1}} \sum_{s^2 \in S} &\rho(s^1) w(a^1|s^1, \boldsymbol{\beta}) p(s^2|s^1, a^1) r(s^1, a^1, s^2) \\ &\times \nabla_\beta \ln\big(\rho(s^1) w(a^1|s^1, \boldsymbol{\beta}) p(s^2|s^1, a^1) r(s^1, a^1, s^2)\big)\big). \end{aligned} \tag{1.54}$$

Since $\rho(s)$ and $p(s^2|s^1, a^1)$ don't involve $\boldsymbol{\beta}$,

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

Therefore it follows from (1.54) that

$$\nabla_\beta v_1(\boldsymbol{\beta}) = \sum_{s^1 \in S} \sum_{a^1 \in A_{s^1}} \sum_{s^2 \in S} \rho(s^1) w(a^1|s^1, \boldsymbol{\beta}) p(s^2|s^1, a^1) \nabla_\beta \Big(\ln\big(w(a^1|s^1, \boldsymbol{\beta})\big) r(s^1, a^1, s^2) \Big). \tag{1.55}$$

The significance of this expression is that it shows that $\nabla_\beta v_1(\boldsymbol{\beta})$ can be evaluated by averaging $\ln\big(w(a^1|s^1, \boldsymbol{\beta})\big) r(s^1, a^1, s^2)$ over replicates of (s^1, a^1, s^2). **(Add exercise that does this.)**

The following result, which is proved in the appendix to this chapter, generalizes the above argument. It provides the basis for estimating the gradient of the value function using a sampled trajectory.

Theorem 1.1. 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

$$\nabla_{\boldsymbol{\beta}} v(\boldsymbol{\beta}) = E^{(d_{\boldsymbol{\beta}})^\infty} \left[\sum_{j=1}^{N_\Delta} \nabla_{\boldsymbol{\beta}} \ln\big(w(Y_j|X_j; \boldsymbol{\beta})\big) \left(\sum_{i=j}^{N_\Delta} r(X_i, Y_i, X_{i+1}) \right) \right]. \tag{1.56}$$

Observe that the multiplier of the the gradient $\nabla_{\boldsymbol{\beta}} \ln \big(w(Y_j|X_j;\boldsymbol{\beta})$ only involves rewards after decision epoch j.

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 $q(X_j, Y_j)$ was replaced by the constant one, the above expression would correspond to the gradient used when applying gradient ascent to obtain the maximum likelihood estimator of the parameters of the decision rule.

1.5.4 A policy gradient algorithm

The following algorithm, referred to in the literature as REINFORCE[23], describes an early implementation of gradient ascent over the space of parameterized randomized policies. We frequently refer to this algorithm as "the policy gradient algorithm".

Algorithm 1.9. REINFORCE with baseline.

```
1. Initialize $\boldsymbol{\beta}$ and specify a learning rate sequence $\{\tau_n : n = 1, 2, \ldots\}$, a
   baseline $b(s)$ and the number of replicates $N'$.
2. Repeat $N'$ times:
   (a) Generate an episode and save a sequence of states, actions and rewards
       $(s^1, a^1, s^2, r^1, \ldots, s^N, a^N, s^{N+1}, r^N)$ in which $N$ denotes the termination time of the episode.
   (b) $N \leftarrow 1$
   (c) While $n \leq N$
       i. Update
          $\boldsymbol{\beta} \leftarrow \boldsymbol{\beta} + \tau_n \nabla_{\boldsymbol{\beta}} \ln \big(w(a^n|s^n;\boldsymbol{\beta})\big) \left(\sum_{k=n}^{N} r^k - b(s^n)\right)$    (1.57)
       ii. $n \leftarrow n + 1$.
3. Return $w(a|s, \boldsymbol{\beta})$.
```

Observe that:

1. Note that many implementations of REINFORCE are stated with $b(s) = 0$. Including it does not invalidate the gradient derivation in Theorem 1.1 since the baseline does not change the expectation in (1.56). Moreover choosing the baseline appropriately enhances convergence by reducing the variance of the estimates.

[23]REINFORCE is an acronym for the awkward expression "REward INcrement = Non-negative Factor × Offset Reinforcement × Characteristic Eligibility".

2. Representation (1.57) provides insight into the qualitative behavior of the policy gradient algorithm. When $\left(\sum_{k=n}^{N} r^k - b(s^n)\right)$ is positive, components of $\boldsymbol{\beta}$ that make action a^n more likely in state s^n will be increased.

   As a concrete example, consider the case in which the features are indicator functions of state-action pair and $w(a|s, \boldsymbol{\beta})$ is the softmax function. Then

$$\frac{\partial}{\partial \beta_{s,a}} \ln\left(w(a^n|s^n, \boldsymbol{\beta})\right) = \begin{cases} 1 - w(a|s; \boldsymbol{\beta}) & \text{for } a = a^n, s = s^n \\ -w(a|s; \boldsymbol{\beta}) & \text{otherwise.} \end{cases}$$

   so that the only positive component of $\nabla_\beta \ln\left(w(a|s; \boldsymbol{\beta})\right)$ in state s^n is a^n. Hence when $\left(\sum_{k=n}^{N} r^k - b(s^n)\right)$ is positive, β_{s^n,a^n} will increase and all other components will decrease resulting in an increased probability of choosing action a^n in state s^n at future iterations.

3. The algorithm requires an entire episode before updating the parameter estimates. Once the data is generated, REINFORCE updates the parameters stepwise.

4. The update in step 2(b) can be also be implemented in a single step by accumulating the terms of the gradient as in (1.56) prior to an update or even averaging the gradient over several trajectories.

5. The expression $\nabla_{\boldsymbol{\beta}} \ln\left(w(a_n|s_n; \boldsymbol{\beta})\right)$ 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. Step 2c. describes a direct application of gradient ascent. Enhanced versions are available[24].

7. The above algorithm is on policy, that is the evaluates the same policy that was used to generate the sequence of states, actions and rewards. An off-policy variant uses *importance sampling* to a adjust iterates appropriately.

1.5.5 An example

We consider the problem of optimal stopping in a reflecting random walk (Example **??**). In it, $S = \{-M, -(M-1), \ldots, M-1, M\}$, $A_s = \{a_0, a_1\}$ with a_0 corresponding to stopping and a_1 corresponding to continuing. Stopping in state s yield a reward of $g(s)$ while continuing costs c and results in a transition to state s' according to

$$p(s'|s, a_1) = \begin{cases} p & \text{if } s' = s+1, s < N \quad \text{or } s' = s = M \\ 1-p & \text{if } s' = s-1, s > 1 \quad \text{or } s' = s = -M \\ 0 & \text{otherwise} \end{cases}$$

[24]The smoothed gradient descent algorithm ADAM is described Kingma and Ba [2017].

for $0 < p < 1$. **(check this is consistent with optimal stopping formulation earlier)**. Let Δ denote the stopped state.

This is a stochastic shortest[25] 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 we can analyze it as an episodic model.

Illustrative calculations

Before providing computational results, we illustrate the update in step 3 of Algorithm 1.9 on a single hypothetical episode. Suppose an episode generates 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 output of the episode would be the realization[26] when a transition occurs to the stopped state.

$$(5, a_1, -2, 6, a_1, -2, 7, a_0, 49, \Delta, 0).$$

The following steps illustrate the calculation in (1.57) with $b = 0$. Ignoring "fake" transition in Δ, the quantity $\sigma_n := \left(\sum_{k=n}^N r^k - b\right)$ 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

$$b(s) = \beta_{0,0} I_{\{a=a_0\}} + \beta_{0,1} I_{\{a=a_1\}} + \beta_{1,0} s I_{\{a=a_0\}} + \beta_{1,1} s I_{\{a=a_1\}}$$

and the softmax function $w(a|s, \boldsymbol{\beta})$ becomes

$$w(a_0|s, \boldsymbol{\beta}) = \frac{e^{\beta_{0,0}+\beta_{1,0}s}}{e^{\beta_{0,0}+\beta_{1,0}s} + e^{\beta_{0,1}+\beta_{1,1}s}}$$

with $w(a_1|s, \boldsymbol{\beta}) = 1 - w(a_0|s, \boldsymbol{\beta})$. Some simple calculus establishes that the gradients are given by

$$\nabla_{\boldsymbol{\beta}} ln\big(w(a_0|s, \boldsymbol{\beta})\big) = \begin{bmatrix} 1 - w(a_0|s, \boldsymbol{\beta}) \\ -w(a_1|s, \boldsymbol{\beta}) \\ s(1 - w(a_0|s, \boldsymbol{\beta})) \\ -sw(a_1|s, \boldsymbol{\beta}) \end{bmatrix} = w(a_1|s, \boldsymbol{\beta}) \begin{bmatrix} 1 \\ -1 \\ s \\ -s \end{bmatrix}$$

and

$$\nabla_{\boldsymbol{\beta}} \ln\big(w(a_1|s, \boldsymbol{\beta})\big) = w(a_0|s, \boldsymbol{\beta}) \begin{bmatrix} -1 \\ 1 \\ -s \\ s \end{bmatrix}$$

[25]Since we are maximizing rewards, this is actually a *longest* path model.

[26]Assuming the reward is received immediately after the stopped action is chosen.

where $\boldsymbol{\beta} = (\beta_{0,0}, \beta_{0,1}, \beta_{1,0}, \beta_{1,1})^T$. 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 $a = a_0$, (1.57) becomes

$$\boldsymbol{\beta} \leftarrow \boldsymbol{\beta} + \tau_n \left(\sum_{k=n}^{N} r^k \right) w(a_1|s, \boldsymbol{\beta}) \begin{bmatrix} 1 \\ -1 \\ s \\ -s \end{bmatrix}. \tag{1.58}$$

Observe that the only stochastic element in this expression is the term $\left(\sum_{k=n}^{N} r^k\right)$ so that the variability of this quantity effects the stability of the estimates of $\boldsymbol{\beta}$. Standardizing rewards can reduce this variability.

Suppose that $\boldsymbol{\beta} = (0,0,0,0)^T$ and $\tau_n = 0.1$, then the iterates become

$$\boldsymbol{\beta} = 0.1 \cdot 49 \cdot 0.5 \begin{bmatrix} 1 \\ -1 \\ 7 \\ -7 \end{bmatrix} = \begin{bmatrix} 2.45 \\ -2.45 \\ 17.15 \\ -17.15 \end{bmatrix}, \ \boldsymbol{\beta} = \begin{bmatrix} -2.35 \\ 2.35 \\ -14.10 \\ 14.10 \end{bmatrix}, \text{and } \boldsymbol{\beta} = \begin{bmatrix} -2.35 \\ 2.35 \\ -14.10 \\ 14.10 \end{bmatrix}.$$

Note that corresponding to the last $\boldsymbol{\beta}$ above, the action choice probabilities in state 5 are $w(a_0|5, \boldsymbol{\beta}) = 1$ and $w(a_1|5, \boldsymbol{\beta}) = 0$.

Alternatively setting $b = 47$, which equals the mean of the σ_i, we obtain the sequence

$$\boldsymbol{\beta} = \begin{bmatrix} 0.30 \\ -0.30 \\ 1.70 \\ -1.60 \end{bmatrix}, \ \boldsymbol{\beta} = \begin{bmatrix} 0.30 \\ -0.30 \\ 1.70 \\ -1.60 \end{bmatrix}, \text{and } \boldsymbol{\beta} = \begin{bmatrix} 0.50 \\ -0.50 \\ 2.70 \\ -2.70 \end{bmatrix}.$$

Although the $\boldsymbol{\beta}$ estimates have changed and have become smaller in magnitude, the corresponding action selection probabilities in state 5 remain $w(a_0|5, \boldsymbol{\beta}) = 1$ and $w(a_1|5, \boldsymbol{\beta}) = 0$.

A numerical experiment

Now we provide a more detailed numerical study that implements Algorithm 1.9 in the context of an optimal stopping problem on $S = \{-50, \dots, 50\}$ and $g(s) = -s^2$. The objective is maximize the expected total (undiscounted) reward averaged over the initial state distribution. With this choice of $g(s)$ the goal is stop as close to zero as possible.

Figure 1.10 symbolically represents the *optimal* value functions and stopping region for three choices of the continuation cost. The optimal value function was found to a high degree accuracy using value iteration with $v^0(s) = -50^2$. This starting value, which was a lower bound on the value function, was chosen to ensure monotone convergence of value iteration (see Chapter **??**).

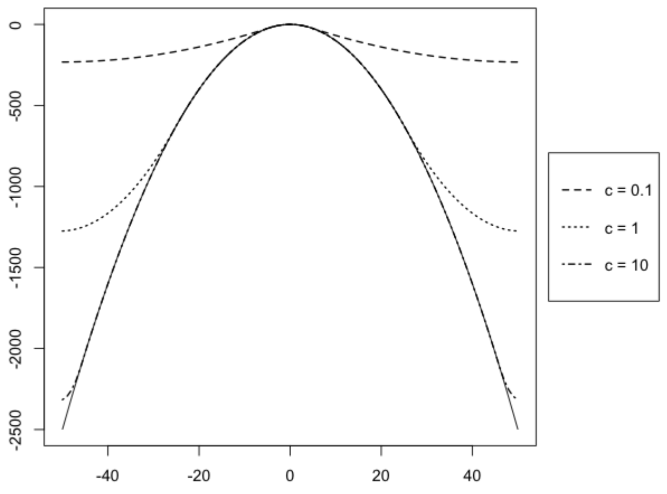


Figure 1.10: Optimal value functions (dashed lines) for three choices of c for the problem of optimally stopping in a random walk with $p = 0.5$ The solid line shows the reward of stopping in each state. The optimal policy is to stop when the optimal value function and the reward from stopping agree. Observe that the stopping region increases with respect to the continuation cost.

To apply the policy gradient algorithm above, first assume a tabular representation so that features have the form

$$b(s,a) = I_{\{s',a'\}}(s,a), \text{ for } s' \in S \text{ and } a' = a_0, a_1$$

so that

$$w(a'|s',\boldsymbol{\beta}) = \frac{e^{\eta\beta_{s',a'}}}{\sum_{s=-50}^{50}\sum_{j=0}^{1} e^{\eta\beta_{s,a_j}}} \tag{1.59}$$

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

For this model, (assuming[27] that $\eta = 1$) the gradient is available in closed form as:

$$\frac{\partial}{\partial\beta_{s,a}} \ln w(a'|s',\boldsymbol{\beta}) = \begin{cases} \big(1 - w(a|s,\boldsymbol{\beta})\big) & s = s', a = a' \\ -w(a|s,\boldsymbol{\beta}) & \text{otherwise.} \end{cases} \tag{1.60}$$

Since $0 \leq w(a|s,\boldsymbol{\beta}) \leq 1$, this means that the only positive term in the gradient corresponds to the state-action pair that is being evaluated.

Algorithm 1.9 was implemented with the following specifications:

- Initial distribution uniform on S;

[27]When $\eta \neq 1$ it will appear in each of the the derivative terms however it can be ignored as it can be absorbed in the learning rate.

- $c = 0.1, 10, 50$;
- $p = 0.5$;
- $\boldsymbol{\beta}$ initialized to $\mathbf{0}$;
- N' =500,000;
- episodes were truncated after 100 steps;
- softmax exponent $\eta = 1/100$;
- learning rate, $\tau_n = \frac{1000}{10000+n}$ where n represented the episode number, and
- baseline $b(s) = -50^2$.

The baseline was chosen so as to avoid overflow in probability estimates. A common random seed was used across all values of c.

Figure 1.11 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 lines correspond to 4th order polynomial fits to components of $\boldsymbol{\beta}$ corresponding to each action. The figures show that individual component values (dots) are quite noisy. Smoothing them with a fourth order polynomial in the state values for each action reveals some interesting and anticipated behavior.

To interpret these figures note that when 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 which is centered on $s = 0$ and increases with respect to c. When c is large, it's 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 above algorithm differs considerably form the optimum policy indicate in Figure 1.10.

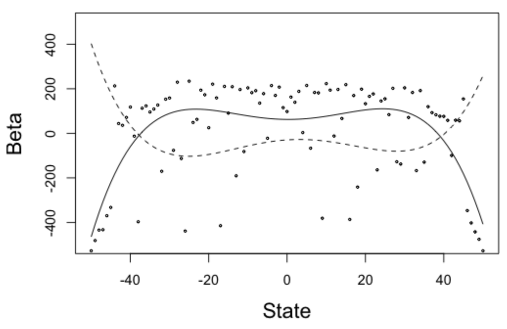
To conclude the analysis, this example fits a fourth-order polynomial using centered and standardized states as features for each action. That is

$$w(a_j|s, \boldsymbol{\beta}) = \frac{e^{\eta\left(\sum_{i=0}^{4} \beta_{i,j}\tilde{s}^i\right)}}{\sum_{k=0}^{1} e^{\eta\left(\sum_{i=0}^{4} \beta_{i,k}\tilde{s}^i\right)}}$$

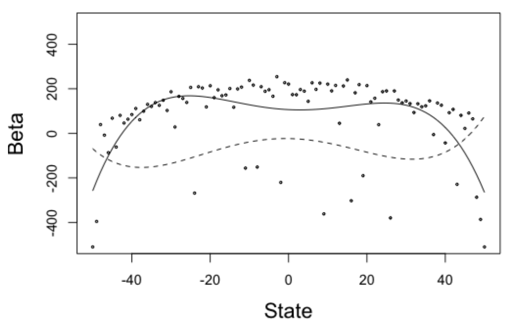
for $j = 0, 1$ where $\tilde{s}$ represents the standardized state. This model has 10 parameters as opposed to the tabular model above which has 202 parameters. It is convenient to represent $\boldsymbol{\beta}$ in matrix form as

$$\boldsymbol{\beta} = \begin{bmatrix} \beta_{0,0} & \beta_{0,1} \\ \beta_{1,0} & \beta_{1,1} \\ \beta_{2,0} & \beta_{2,1} \\ \beta_{3,0} & \beta_{3,1} \\ \beta_{4,0} & \beta_{4,1} \end{bmatrix} := \begin{bmatrix} \boldsymbol{\beta}_0 & \boldsymbol{\beta}_1 \end{bmatrix}$$

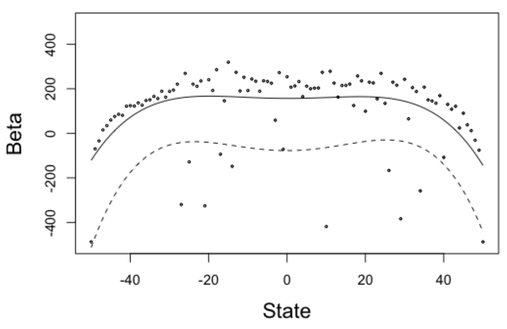
[28]Alternatively the estimates can be plotted on the probability scale



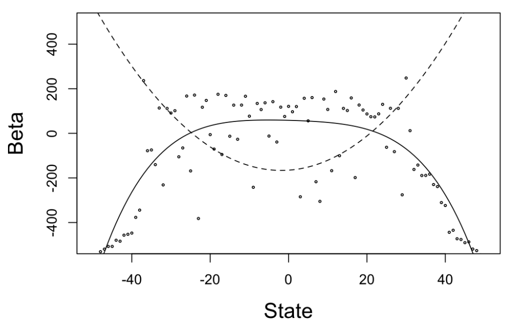
(a) $c = 0.1$ and $\lambda = 1$



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



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



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

Figure 1.11: Parameter estimates obtained applying Algorithm 1.9 to optimal stopping on a random walk in a single typical replicate. The dots correspond to the estimates of β_{s,a_0}, the solid line to the fit to these points using a fourth order polynomial and the dashed line indicates the fitted values to β_{s,a_1} using a fourth order polynomial.

Again (with $\eta = 1$) the gradient of $\ln w(a|s, \boldsymbol{\beta})$ is available in closed form with components

$$\frac{\partial}{\partial \beta_{i,j}} \ln w(a_0|s, \boldsymbol{\beta}) = \begin{cases} \big(1 - w(a_0|s, \boldsymbol{\beta})\big)\tilde{s}^i & j = 0 \\ -w(a_1|s, \boldsymbol{\beta})\tilde{s}^i & j = 1 \end{cases} \tag{1.61}$$

and

$$\frac{\partial}{\partial \beta_{i,j}} \ln w(a_1|s, \boldsymbol{\beta}) = \begin{cases} -w(a_0|s, \boldsymbol{\beta})\tilde{s}^i & j = 0 \\ \big(1 - w(a_1|s, \boldsymbol{\beta})\big)\tilde{s}^i & j = 1. \end{cases}$$

Note that in this model, $w(a_1) = 1 - w(a_0)$ so that the above expressions can be simplified further but we leave them in the above form in order to easily generalize to cases with more than two actions such as the queuing service rate control model analyzed earlier.

Figure 1.12 show the fitted exponents as a function of the state using a fourth degree polynomial with $\eta = 1/100$ and $b(s) = -.3(50)^2$. Observe that the stopping regions differ from the optimal (in the case $c = 0.1$ and $\lambda = 1.0$) and also from those using a polynomial fit applied to the tabular model. Note also that with this baseline choice and parameter choice, the estimates when $c = 10$ diverged resulting in overflow in the exponents of the softmax.

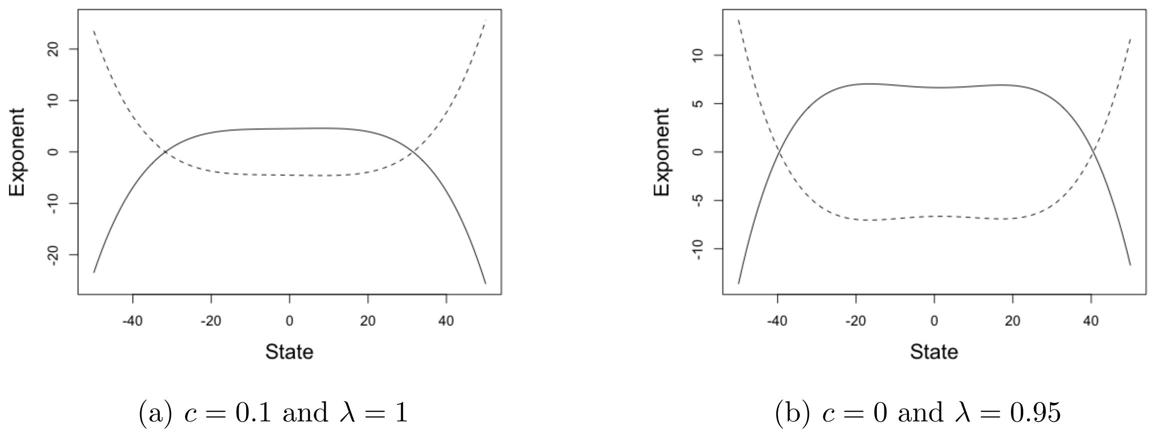


Figure 1.12: Parameter estimates obtained applying Algorithm 1.9 to optimal stopping on a random walk in a single typical replicate using a fourth order polynomial. Solid line corresponds to stopping and the dashed line to continuing.

1.5.6 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 in with 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.

Algorithm 1.9 applies directly to the first representation where the stopping time is either be sampled before each episode or realized by sampling from a Bernoulli distribution at each decision epoch within an episode. To apply truncation, (1.57) must be modified to include the discount factor as follows:

$$\boldsymbol{\beta} \leftarrow \boldsymbol{\beta} + \tau_n \nabla_{\boldsymbol{\beta}} \ln \big(w(a^n|s^n; \boldsymbol{\beta})\big) \left(\sum_{k=n}^{N} \lambda^{k-n} r^k - b(s) \right) \tag{1.62}$$

Otherwise the algorithm is applied as is.

A numerical example

We illustrate the truncation approach in the context of the frequently analyzed two-state model (Example **??**). Experiments used softmax action selections probabilities and tabular representations for the policy so that $\boldsymbol{\beta} \in \{\beta_{1,1}, \beta_{1,2}, \beta_{2,1}, \beta_{2,2}\}$ where the first subscript corresponds to the state and the second subscript corresponds to the action. The implementation used a discount rate $\lambda = 0.9$, $N' = 1000$ episodes, truncation at $N = 100$, learning rate $\tau_n = 50/(1000 + n)$ where n 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 softmax exponent equal to $1/10$, 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 seeds. For example, for a specific random number seed

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

Using a softmax exponent equal to 1 resulted in frequent convergence to a non-optimal policy.

1.5.7 Policy gradient: concluding remarks

In the discrete domain numerical examples in this section, we have found that the policy gradient algorithm is extremely sensitive algorithmic specification including: initialization, baseline specification, state and reward scaling, softmax scaling, and learning rate.

We observed that a configuration that identifies an optimal policy for one random number seed may converge to a non-optimal policy for another random number seed. These results are consistent with observations in the literature for instance [29] noted:

- 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 less achieved 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 methods herein.

[29] Gronauer et al. [2021]

1.6 Actor-Critic Algorithms: Combining policy and value function approximation

We saw in the previous section that the expression:

$$\delta(s) := \left(\sum_{k=n}^{N} r^k - b(s^n) \right)$$

in (1.57) is fundamental to behavior of iterates of the policy gradient algorithm. In this expression N represented a realization of the episode length and $\sum_{k=n}^{N} r^k$ a realization of the total reward over the episode starting in in state s^n and choosing action a^n. In other words, the sum represents a realization of the state-action value function $q^{(d_\beta)^\infty}(s^n, a^n)$ under the randomized stationary policy $(d_\beta)^\infty$ corresponding to weights $\boldsymbol{\beta}$. To simplify notation let $q(s, a, \beta) := q^{(d_\beta)^\infty}(s, a)$.

This observation sheds light on choosing the baseline. Suppose we choose the baseline to be the expected value of the policy $(d_\beta)^\infty$ starting in s^n, $v^{(d_\beta)^\infty}(s^n)$. To simplify notation let $v(s, \boldsymbol{\beta}) := v^{(d_\beta)^\infty}(s)$.

Then reminiscent of policy iteration algorithms in Chapters **??** - **??**:

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 above will generate a new weight vector that corresponds to a policy with an increased value. We include the expression ”on average” in the previous statement to account for stochastic variation in the course of a simulation.

It is convenient to introduce the *advantage function*[30]

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

defined for $s \in S$, $a \in A_s$ and $\boldsymbol{\beta} \in \Re^K$. Then the sign and magnitude of the advantage function impact the change in parameter values and the probability of selecting the designated action.

Note that one benefit of using the advantage function is that it may avoid explicit calculation of the two individual components. Since $q(s, a, \boldsymbol{\beta})$ and $v(s, \boldsymbol{\beta})$ are not known, the challenge is to incorporate these expressions in algorithms. Of course in a simple model, $v(s, \beta)$ can be computed exactly using appropriate policy evaluation

[30]Hopefully the use of the symbol A to represent a function will not be confused with the notation A_s for the set of actions in state A.

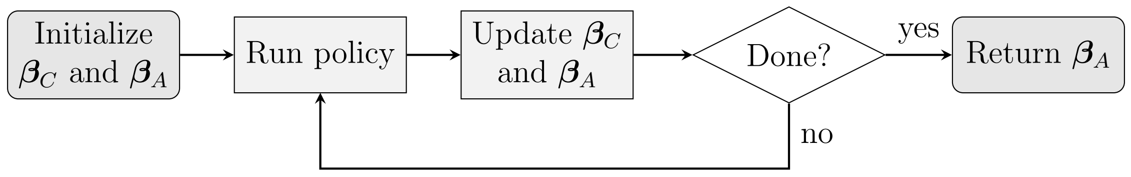

Initialize $\boldsymbol{\beta}_C$ and $\boldsymbol{\beta}_A$
Run policy
Update $\boldsymbol{\beta}_C$ and $\boldsymbol{\beta}_A$
Done?
yes
Return $\boldsymbol{\beta}_A$
no

Figure 1.13: Steps in an actor-critic algorithm.

methods[31], however in doing so we found that convergence to an optimal policy was not guaranteed, especially in the random walk model of the previous section.

Note that in this notation the gradient of the value function can be represented by

$$\nabla_{\boldsymbol{\beta}} v(\boldsymbol{\beta}) = E^{(d_{\boldsymbol{\beta}})^\infty}\left[\sum_{j=1}^{N_\Delta} \nabla_{\boldsymbol{\beta}} \ln\big(w(Y_j|X_j;\boldsymbol{\beta})\big) A(X_i,Y_i,\boldsymbol{\beta})\right]. \tag{1.63}$$

We digress regarding nomenclature. The literature refers to algorithms which use the advantage function or state-action value function combined with policy gradient algorithm as *actor-critic algorithms.* The *actor* corresponds to a policy and the *critic* an evaluator. Feedback from the critic enables to actor to improve the policy. Policy gradient algorithms are frequently referred to as actor-only algorithms and q-learning algorithms as critic-only algorithms.

Actor-critic algorithm overview

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 features and $\boldsymbol{\beta}_A$ denote the corresponding vector of weights. Similarly let $\mathbf{b}_C(s,a)$ denote the vector of critic features and $\boldsymbol{\beta}_C$ denote the corresponding vector of critic weights. Note the features may be either the same or different.

Figure **??** provides a high level schematic of the 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 total discounted (or average reward) model while the batch mode is designed for episodic task however it can be adopted for discounted models by 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. In batch mode, 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 a policy gradient with an estimate of the critic based on the current rewards or previous rewards in the advantage function.

[31]either value iteration of solving a evaluation equations

Note that an actor-critic algorithm avoids the need to use ϵ-altered policy in Q-policy iteration.

1.6.1 An online actor-critic algorithm

The online actor-critic algorithm is the easiest to describe and applies to discounted infinite horizon applications. We state it assuming a linear value function approximation

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

Algorithm 1.10. On-line actor critic algorithm[a]

1. **Initialization:**
    (a) Initialize $\boldsymbol{\beta}_C$ 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 number of iterations N and set $n = 1$.
    (d) Choose $s \in S$.
2. **Loop:** While $n < N$:
    (a) Sample a from $w(\cdot|s, \boldsymbol{\beta}_A)$.
    (b) Generate (s', r).
    (c) **Update critic:**

$$\boldsymbol{\beta}_C \leftarrow \boldsymbol{\beta}_C - \tau_n^C \big(r + \lambda v(s', \boldsymbol{\beta}_C) - v(s, \boldsymbol{\beta}_C)\big)\mathbf{b}_C(s) \tag{1.64}$$

    (d) **Evaluate advantage:**

$$A(s, a, \boldsymbol{\beta}_C) := r + \lambda v(s', \boldsymbol{\beta}_C) - v(s, \boldsymbol{\beta}_C). \tag{1.65}$$

    (e) **Update actor:**

$$\boldsymbol{\beta}_A \leftarrow \boldsymbol{\beta}_A + \tau_n^A \Big(\nabla_{\boldsymbol{\beta}_A} w(a|s, \boldsymbol{\beta}_A)\Big) A(s, a, \boldsymbol{\beta}_C). \tag{1.66}$$

    (f) $n \leftarrow n + 1$, $s \leftarrow s'$.
3. Return $\boldsymbol{\beta}_A$.

[a]Some authors refer to this as *one-shot* actor critic.

Some comments about this algorithm follow:

1. As stated, the critic update uses TD(0). A TD(γ) update may be preferable however the critic update step will be less transparent.

2. Note that the advantage equals the temporal difference applied at the updated value function parameter. To avoid this extra calculation, the advantage can be set equal to the temporal difference in which case the updated critic parameter is not used until the next iterate.

3. If a nonlinear value function approximation is used, $\nabla_{\boldsymbol{\beta}_C} v(s, \boldsymbol{\beta}_C)$ replaces the expression $\mathbf{b}_C(s)$ in 1.64.

4. 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.

5. All the earlier comments about exploration and model tuning the apply.

6. Many variants in the literature consider updates of the critics based on parallel sampling states and rewards from parallel implementations.

7. **(Make more precise)**Since the algorithm is a variant of a policy gradient algorithm it will converge under standard conditions on the learning rate for the actor. By adaptively choosing the baseline using the advantage function, the convergence should be to an optimal policy.

An example

This example illustrates the online actor-critic algorithm in the context of the two-state example.

Example 1.9. This example applies Algorithm 1.10 to a discounted ($\lambda = 0.9$) version of the two-state example analyzed frequently throughout the book. It uses state-action indicators as features for the actor and indicators of the state as features for the critic. Consequently the updates for the critic correspond to TD(0) recursion

$$v(s) \leftarrow v(s) - \tau_n^C \Big(r + \lambda v(s') - v(s) \Big),$$

updates for the advantage are given by

$$A(s,a) = r + \lambda v(s') - v(s),$$

and updates for components of the actor weights are given by

$$\beta_{(s',a')} \leftarrow \begin{cases} \beta_{(s,a)} + \tau_n^A \Big(1 - w(a|s, \boldsymbol{\beta}_A) \Big) A(s,a) & \text{for } (s',a') = (s,a) \\ \beta_{(s',a')} + \tau_n^A \Big(- w(a|s, \boldsymbol{\beta}_A) \Big) A(s,a) & \text{otherwise.} \end{cases}$$

Note that at each iteration, the critic is updated only in the observed state s while the actor weights are updated for all state-action pairs.

The algorithm converged to the optimal value function and policy for a wide range of learning rates and numbers of iterations. Figure 1.14 shows the sequences of values for a typical replicate with N =20,000 iterations and $\tau_n^A = \tau_n^C = 100/(1000 + n)$. In this replicate the corresponding policy used action selection probabilities

$$w(a_{1,2}|s_1, \boldsymbol{\beta}_A) = 0.994 \quad \text{and} \quad w(a_{2,2}|s_2, \boldsymbol{\beta}_A) = 1.000$$

in agreement with the optimal policy.

We also observed that the algorithm converged to the optimal policy for random number seeds in which the policy gradient algorithm (Algorithm 1.9 converged to a sub-optimal policy. Thus by replacing an arbitrary baseline by a critic that tracked the "current policy", the algorithm avoided convergence to a sub-optimal policy.

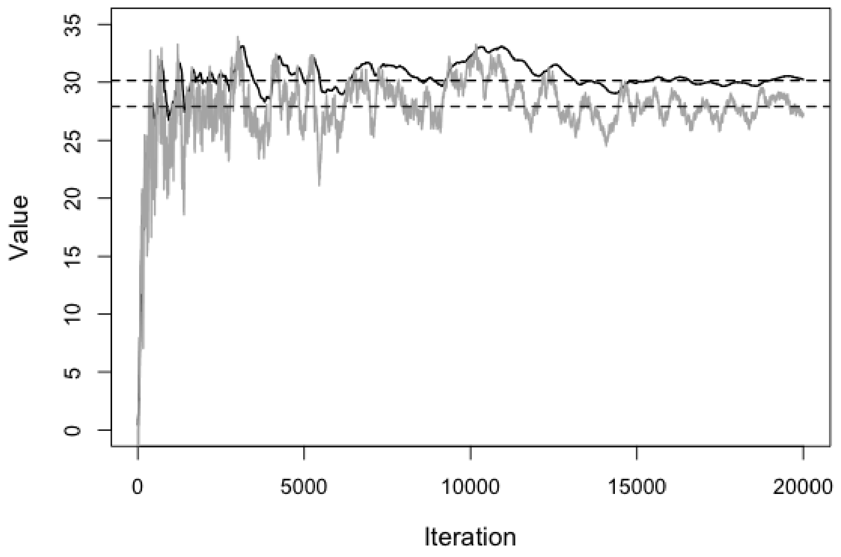


Figure 1.14: Value function estimates in two-state example using Algorithm 1.10 obtained in a typical replicate. Black lines correspond to $v^*(s_1)$ and grey lines to $v^*(s_2)$; solid lines correspond to estimates and dashed lines to true value.

1.6.2 A batch actor-critic algorithm

The following algorithm describes actor-critic implementations suitable for an undiscounted episodic models. It generalizes the (batch) policy gradient algorithm above (Algorithm 1.9) by using the critic to update the baseline.

```
Algorithm 1.11. Batch actor-critic

1. Initialization:
   (a) Initialize $\boldsymbol{\beta}_A$.
   (b) Specify learning rate sequence $\{\tau_n^A; n = 1, 2, \ldots\}$.
   (c) Specify number of episodes $N'$.
2. Repeat $N'$ times:
   (a) Generate an episode and save a sequence of states, actions and rewards $(s^1, a^1, s^2, r^1, \ldots, s^N, a^N, s^{N+1}, r^N)$ in which $N$ denotes the termination time of the episode.
   (b) Update critic: Choose
       $$\boldsymbol{\beta}_C \in \operatorname*{arg\,min}_{\boldsymbol{\beta} \in \Re^K} \sum_{n=1}^{N} \Big( \sum_{k=n}^{N} r^k - v(s^n, \boldsymbol{\beta}) \Big)^2 \qquad (1.67)$$
   (c) $n \leftarrow 1$
   (d) While $n \leq N$
       i. Estimate advantage:
          $$A(s^n, a^n, \boldsymbol{\beta}_C) = \sum_{k=n}^{N} r^k - v(s^n; \boldsymbol{\beta}_C) \qquad (1.68)$$
       ii. Update actor:
          $$\boldsymbol{\beta}_A \leftarrow \boldsymbol{\beta}_A + \tau_n^A \nabla_{\boldsymbol{\beta}} \ln \big( w(a^n | s^n; \boldsymbol{\beta}_A) A(s^n, a^n, \boldsymbol{\beta}_C) \qquad (1.69)$$
       iii. $n \leftarrow n + 1$.
3. Return $w(s|a, \boldsymbol{\beta}_A)$.
```

Some comments on the algorithm follow.

1. Critic weights $\boldsymbol{\beta}_C$ can be 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 uses regression to estimate the the critic weights based derived from an entire episode. An alternative is to minimize

$$h(\boldsymbol{\beta}) := \sum_{n=1}^{N} \Big(\sum_{k=n}^{N} r^k - v(s^n, \boldsymbol{\beta}) \Big)^2$$

by gradient descent. To do so, an algorithm would compute

$$\nabla_{\boldsymbol{\beta}} h(\boldsymbol{\beta}) = \nabla_{\beta} \sum_{n=1}^{N} \Big(\sum_{k=n}^{N} r^k - v(s^n, \boldsymbol{\beta}) \Big)^2$$

which in the linear case equals

$$\nabla_{\boldsymbol{\beta}} h(\boldsymbol{\beta}) = -2 \sum_{n=1}^{N} (\Big(\sum_{k=n}^{N} r^k - v(s^n, \boldsymbol{\beta}) \Big) \mathbf{b}^C(s^n)$$

where $\mathbf{b}^C(s)$ **(check notation for critic features)** denotes the critic weights evaluated at s. Absorbing the constant into the learning rate gives the recursion for $\boldsymbol{\beta}_C$:

$$\boldsymbol{\beta}_C \leftarrow \boldsymbol{\beta}_B - \tau_n^C \nabla_{\boldsymbol{\beta}} h(\boldsymbol{\beta}). \tag{1.70}$$

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:

$$A(s^n, a^n, \boldsymbol{\beta}_C) := r^n + v(s^{n+1}, \boldsymbol{\beta}_C) - v(s^n, \boldsymbol{\beta}_C) \tag{1.71}$$

4. The actor weights can be updated iteratively as in the algorithm or accumulated and then updated using a single gradient ascent step.

5. Instead of using value functions, one could evaluate the advantage in terms of the state action function $q(s, a; \boldsymbol{\beta}_C)$ as follows:

$$A(s^n, a^n, \boldsymbol{\beta}_C) := q(s^{n+1}, a^{n+1}, \boldsymbol{\beta}_C) - q(s^n, a^n, \boldsymbol{\beta}_C). \tag{1.72}$$

(Check this makes sense)

An example

This simple example applies the batch actor critic algorithm to the shortest path model in Section 1.3.3

This example applies Algorithm 1.11 to an instance 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

$$v\big((i,j), \boldsymbol{\beta}^C\big) = \beta_0^C + \beta_{1,0}^C i + \beta_{0,1}^C j + \beta_{1,1}^C ij + \beta_{2,0}^C i^2 + \beta_{0,2}^C j^2$$

so that $\boldsymbol{\beta} = (\beta_0^C, \beta_{1,0}^C, \beta_{0,1}^C, \beta_{1,1}^C, \beta_{2,0}^C, \beta_{0,2}^C)$ where critic parameters are computed using ordinary least squares to obtain an estimate of $\boldsymbol{\beta}$ and compute the advantage. The actor was represented in tabular form with $\boldsymbol{\beta}^A = \{\beta_{(i,j,a)}^A : (i,j) \in S \text{ and } a \in A_{(i,j)}\}$.

Implementation details follow: The algorithm was initiated with components of $\boldsymbol{\beta}^A$ sampled from a standard normal distribution to so as to avoid too many long episodes. Episodes were truncated at 200 iterations and the algorithm was run for 50000 episodes, each starting from a random cell not equal to the target. The learning rate, $\tau_n^A = 40/(400 + n)$.

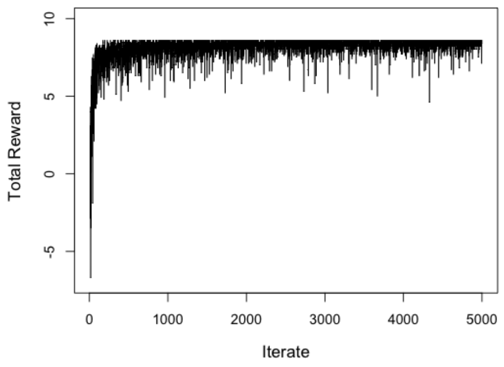
Figure 1.15 shows how the total reward per episode varies within a single replicate starting in cell $(1, 1)$ and at random. Observe that the reward increases and stabilizes with variability in total rewards when starting in a random cell more variable then when starting in cell $(1, 1)$. The advantage 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[32] 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.

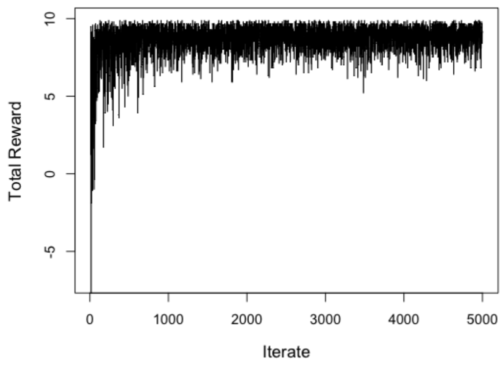
Note that when the critic was evaluated *after* the actor, the algorithm converged to sub-optimal policies that cycled between states and never reached the target cell.

We also applied the policy gradient algorithm (Algorithm 1.9) to this example. Our implementation was the same as above but instead of using the critic, it set the baseline (bl in Table 1.6 equal to $0, -8$ and the (running) mean of the previous total rewards accumulated from the starting state. Observe form Table 1.6 that the quality of estimates varied significantly with the baseline with the running mean giving the best results. In fact for this replicate the policy gradient with the mean of total rewards as the baseline 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.

[32]For example 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 non-terminal random cell.

Figure 1.15: A typical replicate of the total reward per episode in a model with a 10×7 grid with the target cell in lower right.

Method	Total Reward mean	Total Reward std dev
PG ($bl = 0$)	6.31	6.14
PG ($bl = -8$)	5.80	8.06
PG (bl =mean)	8.52	0.47
AC	8.24	0.78

Table 1.6: Mean and standard deviation of total rewards over episodes for a single replicate in which all four methods converged. The results used a common random number seeds obtained using the policy gradient algorithm with the specified baseline (bl) and the actor-critic algorithm as described above.

Delivering coffee

This section applies the actor-critic algorithm to find an optimal policy in the coffee delivering robot problem from Section **??** on a 5×3 grid. We divide rewards and costs by 10 to 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. However, randomness is introduced through $w(a|s, \boldsymbol{\beta}^A)$.

Algorithmic details follow. The value function is approximated by

$$v\big((i,j), \boldsymbol{\beta}^C\big) = \boldsymbol{\beta}^C_{0,0} + \beta^C_{1,0} i + \beta^C_{0,1} j$$

where i denotes the row (indexed from the bottom) and j denotes the column (indexed from the left)[33] Its parameters are estimated in step 2(b) of Algorithm 1.11 using linear regression. The actor is again modelled 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 20000 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 seeds. When it did not, it identified a policy that cycled without delivering the coffee. For example Figure 1.16a shows a typical replicate when the algorithm identified a good policy. For this replicate, the policy assigned probabilities exceeding 0.9 to each step on the "conservative" path $13 \to 14 \to 15 \to 12 \to 9 \to 6 \to 3 \to 2 \to 1$. Note that for a few earlier episodes, the robot incurred a cost of -10 when the robot fell down the stairs but eventually it learned to avoid such incidents. For this model the value function estimate was

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

So that 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.

We also applied the algorithm to a similar problem on a 10×5 grid with the stairs occupying 4 cells on the left hand side. Results from a single replicate are shown in Figure 1.16b. Observe that in several episodes the robot fell down the stairs more than once, with this event occurring less frequently in later episodes. The optimal path was similar to that in the smaller model, moving down to the edge of the stairwell, then moving right to column 3 and then downward toward the lower boundary.

In both models, estimating the critic weights by gradient descent failed to identify an optimal policy.

1.6.3 Actor-critic: remarks and enhancements

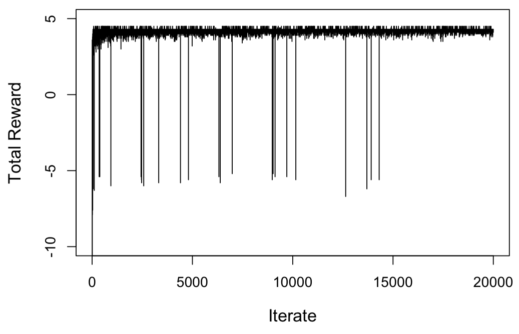
Policy gradient and actor-critic algorithms have been researched extensively in the literature. Various sources have suggested improvements to enhance convergence. They include:

Normalization: Centering and scaling states and rewards, especially when they are highly variable) can enhance convergence.

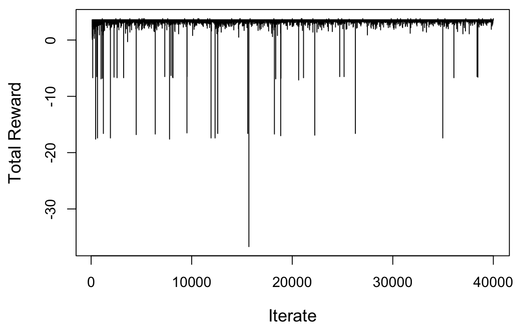
Parallelization: Running many episodes in parallel (on several robots) can provide better critic estimates.

Natural gradients: The gradients described above are sensitive to both the parameterization and the geometry (curvature) of the underlying surface. The

[33]For example, cell 13 corresponds to row 1 and column 1.



(a) 5×3 grid; the median total reward equalled 4.3 with range between -20.9 and 4.5.



(b) 10×5 grid; the median total reward equalled 3.6 with range between -89.2 and 3.8.

Figure 1.16: Total reward by iterate for a single replicate of the coffee delivering robot model obtained using Algorithm 1.11 for two different grid sizes.

natural gradient accounts for this in the actor by multiplying the the gradient by the inverse of the Fisher information matrix given by

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

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:

$$r^n + \ldots + r^{n+m} + v(s^{n+m+1}, \boldsymbol{\beta}_C) - v(s^n, \boldsymbol{\beta}_C)$$

Improved gradient descent: Adaptive moment estimation (ADAM) 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 $v(a|s, \boldsymbol{\beta}_C)$ one can replaced 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. Considerable experimentation is required to obtain good results. It is our intent that the introduction above provides an entry in to understanding and applying these concepts.

1.7 Further topics

As the intent of this part of the book is provide an introduction to reinforcement learning methods, it has been necessary to omit many topics. Here we provide a brief overview of further areas for study:

1.7.1 Simultaneous learning and control

Earlier we have emphasized the distinction between model-based and model-free methods. Methods referred to as *model-based Rl* learn transition probabilities and rewards at the same time as they they update a policy or value-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 Section **??**), 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 benefits 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 start episodes in simulations at later points in the maze that would be visited 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 grid-world 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. Moreover

Algorithms that implement this approach include Dyna (Sutton and Barto [2018]) and DreamerV3 (Hafner et al. [2024]). The latter algorithm was used to find solutions to the Brio Labyrinth.

1.7.2 Experience replay

Experience replay is an offline method developed to improve the efficiency and stability of learning algorithms. It applies to both Q-learning and policy gradient type 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?)e. 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 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.

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.

1.7.3 Deep learning

Deep learning refers to the use of high-dimensional neural networks in reinforcement learning. 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 learninguse neural networks to approximate value functions, state-action values functions and policies.

1.7.4 Importance sampling

Importance sampling is a statistical technique used to estimate properties of a specific distribution while sampling from a different distribution. It is particularly useful when direct sampling from the specific distribution is difficult or computationally expensive. In reinforcement learning, importance sampling is often used for off-policy learning, where the agent learns from data generated by a different policy than the one currently being evaluated or improved.

It works as follows. Suppose we wish to estimate the expected value of a real-valued function $f(\cdot)$ with respect to a known distribution $q(\cdot)$ but want to do so using samples obtained from a different known distribution $p(\cdot)$. Assuming both distributions have the same domain X,

$$E_q[f(X)] := \sum_{x \in X} f(x)q(x) = \sum_{x \in X} f(x)\frac{q(x)}{p(x)}p(x). \tag{1.73}$$

That is

$$E_q[f(X)] = E_p\left[f(X)\frac{q(X)}{p(X)}\right].$$

The quantities $\frac{q(x)}{p(x)}$ are referred to as *importance weights*.

Empirically, if $\{x^1, \cdots, x^N\}$ represent samples from $p(\cdot)$ an estimate of $E_q[f(X)]$ is obtained given by

$$\widehat{E_q[f(X)]} = \frac{1}{N}\sum_{n=1}^{N} f(x^n)\frac{q(x^n)}{p(x^n)}.$$

In reinforcement learning, importance sampling is to evaluate one policy while using data obtained from another 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

$$E^{\gamma}[r(s, Y_1, X_2)] = E^{\delta}\left[r(s, Y_1, X_2)\frac{p(X_2|s, Y_1)w_{\gamma}(Y_1|s)}{p(X_2|s, Y_1)w_{\delta}(Y_1|s)}\right].$$

In this expression, the quantity

$$\frac{p(X_2|s, Y_1)w_{\gamma}(Y_1|s)}{p(X_2|s, Y_1)w_{\delta}(Y_1|s)}$$

is the importance weight. Subsequent expectations require more complex joint probabilities.

Such a calculation is useful in value function estimation and Q-learning when δ is the ϵ-greedy policy used to generate data and γ is the policy on which it is based. See Sutton and Barto [2018] for an example of calculations based on this approach.

1.7.5 Monte Carlo tree search

Monte Carlo tree search (MCTS) is a simulation based approach 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*), decision trees and bandit models.

It works as follows. Given as specific state, the MCTS algorithm consists of loops of the following form:

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 is used in n-arm bandit models.
3. **Generate next state:** The action is implemented (in real life or a simulation) to obtain the next state. This may correspond to the opponents 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 well estimated is encountered. This is sometimes referred to as a *roll out*.
5. **Update values:** Step back through the sample path and update values in all states encountered. This is often referred to as *back-propagation*.

When applied to models with large state and action spaces approximations are required. Often, neural networks are used to represent the values and opponents action choice probabilities.

MCTS underlies Google's AlphaGo and AlphaZero that obtained outstanding successes in chess and Go. MCTS is closely related to the adaptive multistage sampling algorithm of Chang et al. [2005]. Fu [2018] provides an easy to read overview of this material.

Wordle

As an example of MCTS, consider the popular online game, Wordle created by Wardle [2021] and 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 tries by using feedback on the quality of the guess. The game provides feedback by coloring letters in the current guess as follows:

- Green if the letter is in the same position as in the target word,
- Yellow if the letter is in the word but in the wrong position, and

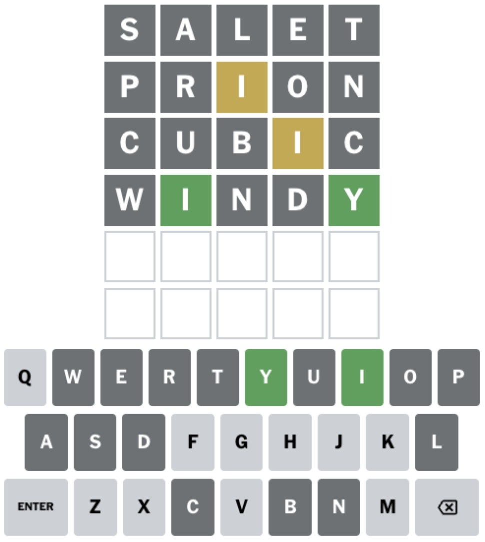


Figure 1.17: Typical Wordle output

- Gray if the letter is not in the word.

In Figure 1.17, the first guess ”SALET” had no correct letters, the second guess ”PRION” contained one correct letter but in the wrong position and the fourth guess ”WINDY” had two letters ”I” and ”Y” that were in the same position in the target word. The keyboard below the guesses indicate the previously guessed letters in black, green or yellow in accordance with there positions in the target word.

Note that the target word in this instance was ”JIFFY”. Data provided by Wordle after completing the completing the game showed that there were only 3 remaining words after guessing ”WINDY”, namely ”DIZZY”, ”DITZY” and ”JIFFY”. Moreover the average score 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 2316 words and only 6 guesses are allowed however in modelling 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.

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, this can be solved using Q-learning.

1.7.6 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 the value function, that is instead of using $w(a|s)$ to represent the policy and $v(s)$ to represent the value, one uses $w(a|o)$ and $v(o)$ where o denotes the observation.

The consequence of doing so is that action chosen by a policy is function of an observation, rather than of the belief distribution over the unobservable states as analyzed in Chapter **??**.

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 instructive to carry out a comprehensive study comparing these two approaches.

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. We subsequently discuss extensions to infinite horizon models.

Theorem A.1. Let N be finite and suppose $w(a|s;\boldsymbol{\beta})$ is differentiable with respect to each component of $\boldsymbol{\beta}$. Then

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

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

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

$$\sum_{i=1}^{N}\left[\sum_{j=1}^{i} b_j\right] a_i = \sum_{j=1}^{N} b_j \left[\sum_{i=j}^{N} a_i\right]. \tag{A.2}$$

This lemma and its proof for N infinite series as Theorem 8.3 in Rudin [1964]. Next we provide a representation for the expected value of an Markov decision process that only receives a reward $r(s, a, s')$ at the end of period n.

Lemma A.2. Consider an n-period Markov decision process that in which

$$r_i(s, a, j) = \begin{cases} 0 & i < n \\ 1 & i = n \end{cases}$$

and let $v_n(\phi)$ denotes the expected reward when using randomized decision rule $d_{\boldsymbol{\beta}}$ selects actions according to $w(a|s; \boldsymbol{\beta})$ averaged over initial states according to $\rho(\cdot)$.
Then if $w(a|s; \phi)$ is differentiable with respect to each component of ϕ:

$$\nabla_{\boldsymbol{\beta}} v_n(\boldsymbol{\beta}) = E^{(d_{\boldsymbol{\beta}})^\infty} \left\{ \left(\sum_{j=1}^{n} \nabla_{\boldsymbol{\beta}} \ln(w(Y_j|X_j; \boldsymbol{\beta})) \right) r(X_n, Y_n, X_{n+1}) \right\}. \tag{A.3}$$

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

$$v_n(\phi) := \sum_{h \in H_n} P_\phi(h) r(s_n, a_n, s_{n+1}) = E^{(d_{\boldsymbol{\beta}})^\infty} \{r(X_n, Y_n, X_{n+1})\} \tag{A.4}$$

where H_n denotes the set of all histories of the form $(s_1, a_1, \dots, s_n, a_n, s_{n+1})$ and $P_\phi(h)$ denotes the probability of each history where the probability distribution is parameterized by $\boldsymbol{\beta}$. By the Markov property, for a specific history,

$$P_\phi(h) = \rho(s_1) w(a_1|s_1; \boldsymbol{\beta}) p(s_2|s_1, a_1) \cdots w(a_n|s_n; \boldsymbol{\beta}) p(s_{n+1}|s_n, a_n). \tag{A.5}$$

Since $w(a|s; \boldsymbol{\beta})$ is differentiable with respect to each component of $\boldsymbol{\beta}$, we can show using (1.43) that

$$\nabla_{\boldsymbol{\beta}} P_{\boldsymbol{\beta}}(h) = P_{\boldsymbol{\beta}}(h) \nabla_{\boldsymbol{\beta}} \ln(P_{\boldsymbol{\beta}}(h)). \tag{A.6}$$

Therefore

$$\nabla_{\boldsymbol{\beta}} v_n(\phi) = \sum_{h \in H_n} P_{\boldsymbol{\beta}(h)} \nabla_{\boldsymbol{\beta}} \ln(P_{\boldsymbol{\beta}}(h)) r(s_n, a_n, s_{n+1}). \tag{A.7}$$

From (A.5)

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

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

$$\nabla_{\boldsymbol{\beta}} \ln(P_{\boldsymbol{\beta}}(h)) = \sum_{j=1}^{n} \nabla_{\boldsymbol{\beta}} \ln(w(a_j|s_j;\boldsymbol{\beta})). \tag{A.8}$$

Hence combining (A.7) and (A.8) gives

$$\nabla_{\boldsymbol{\beta}} v_n(\boldsymbol{\beta}) = \sum_{h\in H_n} P_{\boldsymbol{\beta}(h)} \left(\sum_{j=1}^{n} \nabla_{\boldsymbol{\beta}} \ln(w(a_j|s_j;\boldsymbol{\beta})) \right) r(s_n,a_n,s_{n+1}). \tag{A.9}$$

Rewriting this in expectation form yields the desired representation. □

Proof of Theorem A.1. 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 (A.3) into the above result and applying Lemma A.1 with $b_j = \ln w(Y_j|X_j;\boldsymbol{\beta})$ and $a_i = r(X_i,Y_i,X_{i+1})$. □

Extensions*

(I think this is correct)

Above we prove the policy gradient theorem for finite N. If we use the same proof for a random stopping time N which 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 geometric case.

To analyze infinite horizon discounted models directly requires some additional assumptions, including the variant of Lemma A.1 which allows $N=\infty$. For this to hold when N is infinite, the series must converge absolutely. Requiring $\lambda<1$ and bounded rewards ensures this holds.

Bibliographic remarks

Out discussion just touches the surface of the vast literature of reinforcement learning with function approximation including value-based methods, policy based method and combinations thereof. Szepesvari [2010] and Bertsekas and Tsitsiklis [1996] provide comprehensive overviews of RL methods with references to theoretical results. Bertsekas and Tsitsiklis [1996] also contains a comprehensive historical perspective on the development of RL methods. Sutton and Barto [2018], and Kochendorfer et al. [2022] provide accessible and insightful presentations of this material.

Tsitsiklis and van Roy [1997] rigorously analyzed TD(γ) motivating many subsequent results.

Sutton et al. [1999] establish convergence of Williams [1992] policy gradient algorithm REINFORCE for models with function approximation. Widrow et al. [1973] introduces the idea of using a critic and Barto et al. [1983] formalize this concept by showing how a system with *"two neuronlike elements can solve a difficult learning control problem."* Lehmann [2024] provides a state-of the art reference on policy gradient methods.

The vanilla policy-gradient algorithm is discussed in OpenAI [2024]. Course lecture notes available on the internet provide excellent guides for learning this material. We especially liked the notes of Levine [2024] who provides an insightful discussion of policy gradient and actor-critic algorithms and those of Katselis [2019] on Q-learning.

Lagoudakis and Parr [2003] describe a least squares policy iteration method that avoids the using of learning rates and is suitable for linear function approximations.

Exercises

You should try to replicate the computational results in this chapter and as well try out all the methods herein on examples you are interested in. Chapter **??** provides a wide range of possibilities. We think that advanced appointment scheduling **??** offers an excellent vehicle for testing these methods. Analysing it with neural networks offers considerable research potential.

The exercises below often some alternative suggestions.

1. Specify first-visit and every-visit Monte Carlo approximation algorithms for estimating the value of a policy in an episodic model. Apply them to the shortest path grid-world model in Section **??**.

2. Consider the problem of optimal stopping in a random walk in Example 1.1.

    (a) 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.

    (b) Find a suitable value function approximation in cases in which a linear fit is inadequate.

    (c) Which fit would you use if you wished to incorporate this approximation in an optimization algorithm in which required value functions approximations for several policies.

   (d) Obtain approximations of state-action value functions.

3. (a) State a Monte Carlo 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 instead of the discounted reward as was the case in the truncation version.

   (b) Use it to approximate in the value function in the two instances described in Example 1.2.

4. Show that when features are indicators of states, Algorithm 1.5 reduces to Algorithm **??**

5. Apply gradient descent and stochastic gradient decent to estimate the parameters in the regression model $y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \varepsilon$ using methods similar to Example **??**.

6. **Hybrid SGD** Develop an SGD policy evaluation algorithm that randomly restarts in a random state after M transitions. Apply this approach to the model in Example **??**.

7. **SGD in an episodic model**. Develop an apply a version of SGD for policy evaluation in an episodic model. Apply it to obtain a linear approximation to the value function for π_2 in Example 1.1.

8. Provide a TD(γ) version of Algorithm **??** and apply it to the queuing control model in Example 1.3

9. Fit a cubic spline with two knots to the queuing control model of Example and use Algorithm **??** to estimate its parameters.

10. Show that when features are indicator functions of state-action pairs, the tabular Q-learning algorithm of Chapter **??** is equivalent to the Q-learning algorithm in this chapter.

11. Write out features when $f_k(s)$ and $h_j(a)$ are polynomials in s and a respectively.

12. Consider the episodic shortest path model in Section 1.3.3.

    (a) Verify the conclusions in Section 1.3.3 by developing your own codes.

    (b) Repeat the calculations using a neural network and other parametric state-action value function approximations.

    (c) Specify a SARSA variant of Q-learning algorithm 1.6 and apply it to this problem.

    (d) Solve the model using the Q-policy iteration algorithm, Algorithm 1.7.

13. One of the challenges of using Q-learning in an episodic model is that large rewards are not realized until a goal state is reached. Consider an alternative shortest path model in which the reward when reaching cell (i,j) is of the form $ci + dj$ so that rewards are generated along the path to the goal state.

    Apply Q-learning with function approximation to this variant of the shortest path model and summarize your conclusions.

14. **State aggregation in optimal stopping:** Apply policy gradient and actor-critic to the the optimal stopping model as follows. Let $N = 60$ and compare approximations that aggregate consecutive states into groups of $M = 1, 2, 4$. 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 $j = 1, 2$. We can represent these features as 30 component vectors with the first 15 components corresponding to a_1 and the next 15 components corresponding to a_2.

15. Derive (1.44).

16. Apply policy gradient and actor-critic to the discounted queuing service rate control model.

17. Consider the shortest path through a grid analyzed in Section 1.6.2.

    (a) Replicate the analysis in the text using function approximations for both the actor and critic. See Section 1.3.3 for guidance on specificaiton of basis functions.

    (b) Repeat you analysis assuming the cost per step $c(i, j)$ in cell (i, j) depends on the row i and column j. For example suppose $c(i, j) = -0.1 - 0.1i$. How do you think modifying the costs in this way effects policy choice. (Note we found actor-critic and policy gradient frequently diverged or converged to sub-optimal policies for this modification.)

18. Partial observable models and rl

Bibliography

A. G. Barto, R.S̃. Sutton, and C. W. Anderson. Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-13(5):834–846, 1983.

D. Bertsekas and J. Tsitsiklis. *Neuro-Dynamic Programming*. Athena Scientific, 1996.

D.P. Bertseksas. *Dynamic Programming and Optimal Control, Volume 2, 4th Edition*. Athena Scientific, 2012.

H.S. Chang, M.C. Fu, J. Hu, and S.I. Marcus. An adaptive sampling algorithm for solving Markov decision processes. *Operations Research*, 53(1):126–139, 2005.

M. C. Fu. Monte carlo tree search: A tutorial. In *Proceedings of the 2018 Winter Simulation Conference*, pages 222–236, 2018.

S. Gronauer, M. Gottwald, and K. Diepold. The sucessful ingredients of policy gradient algorithms. In *Proceedings of the Thirtieth Joint Conference on Artificial Intelligence (IIJCAI-21)*, pages 2455–2461, 2021.

D. Hafner, J. Pasukonis, J. Ba, and T. Lillicrap. Mastering diverse domains through world models, 2024. URL `https://arxiv.org/abs/2301.04104`.

D. Katselis. Ece586: Mdps and reinforcement learning: Lecture 10, 2019. URL `http://katselis.web.engr.illinois.edu/ECE586/Lecture10.pdf`.

D. P. Kingma and J. Ba. Adam: A method for stochastic optimization, 2017.

M. J. Kochendorfer, T. A. Wheeler, and K. H. Wray. *Algorithms for Decision Making*. MIT Press, 2022.

M. Lagoudakis and R. Parr. Least-squares policy evaluation. *J. Mach. Learning Res.*, 4:1107–1149, 2003.

M. Lehmann. The definitive guide to policy gradients in deep reinforcement learning: Theory, algorithms and implementations, 2024.

S. Levine. Cs285: Deep reinforcement learning; lectures 4-9, 2024. URL `https://rail.eecs.berkeley.edu/deeprlcourse/`.

OpenAI. Vanilla policy gradient, 2024. URL `https://spinningup.openai.com/en/latest/algorithms/vpg.html`. Accessed: 2024-06-11.

R. *R: A Language and Environment for Statistical Computing.* R Foundation for Statistical Computing, Vienna, Austria, 2021. URL `https://www.R-project.org/`.

W. Rudin. *Principles of Mathematical Analysis, 2nd Edition.* McGraw-Hill, Inc., 1964.

R. Sutton, D. MacAllester, S. Singh, and Y.Manoour. Policy gradient methods for reinforcement learning with function approximation. In *NIPS Proceedings*, pages 1057–1063, 1999.

R. S. Sutton and A. G. Barto. *Reinforcement Learning: An introduction, second edition.* MIT Press, Cambridge, MA, 2018.

C. Szepesvari. *Algorithms for Reinforcement Learning* . Morgan and Claypool, 2010.

J.N. Tsitsiklis and B. van Roy. An analysis of temporal-difference learning with function approximations. *IEEE Trans. of Auto. Cont.*, 42:674–690, 1997.

Josh Wardle. Wordle, 2021. URL `https://www.nytimes.com/games/wordle/index.html`. Accessed: 2024-07-12.

B. Widrow, N. K. Gupta, and S. Maitra. Punish/reward: learning with a critic in adaptive threshold systems. *IEEE Trans. Syst. Man Cybern.*, 3(5):455–465, 1973.

R. Williams. Some statistical gradient-following allgorithms for connectionist reinforcement learning. *Machine Learning*, 8:229–256, 1992.

Index