REINFORCEMENT LEARNING AND STOCHASTIC OPTIMIZATION

A unified framework for sequential decisions

Warren B. Powell

August 22, 2021

A JOHN WILEY & SONS, INC., PUBLICATION

Copyright ©2021 by John Wiley & Sons, Inc. All rights reserved.

Published by John Wiley & Sons, Inc., Hoboken, New Jersey.
Published simultaneously in Canada.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning, or otherwise, except as permitted under Section 107 or 108 of the 1976 United States Copyright Act, without either the prior written permission of the Publisher, or authorization through payment of the appropriate per-copy fee to the Copyright Clearance Center, Inc., 222 Rosewood Drive, Danvers, MA 01923, (978) 750-8400, fax (978) 646-8600, or on the web at www.copyright.com. Requests to the Publisher for permission should be addressed to the Permissions Department, John Wiley & Sons, Inc., 111 River Street, Hoboken, NJ 07030, (201) 748-6011, fax (201) 748-6008.

Limit of Liability/Disclaimer of Warranty: While the publisher and author have used their best efforts in preparing this book, they make no representations or warranties with respect to the accuracy or completeness of the contents of this book and specifically disclaim any implied warranties of merchantability or fitness for a particular purpose. No warranty may be created ore extended by sales representatives or written sales materials. The advice and strategies contained herin may not be suitable for your situation. You should consult with a professional where appropriate. Neither the publisher nor author shall be liable for any loss of profit or any other commercial damages, including but not limited to special, incidental, consequential, or other damages.

For general information on our other products and services please contact our Customer Care Department with the U.S. at 877-762-2974, outside the U.S. at 317-572-3993 or fax 317-572-4002.

Wiley also publishes its books in a variety of electronic formats. Some content that appears in print, however, may not be available in electronic format.

Library of Congress Cataloging-in-Publication Data:

Optimization Under Uncertainty: A unified framework
Printed in the United States of America.

10 9 8 7 6 5 4 3 2 1

CHAPTER 15

BACKWARD APPROXIMATE DYNAMIC PROGRAMMING

Chapter 14 presented the most classical solution methods from discrete Markov decision processes, which are often referred to as "backward dynamic programming" since it is necessary to step backward in time, using the value $V_{t+1}(S_{t+1})$ to compute $V_t(S_t)$. While we can occasionally apply this strategy to problems with continuous states and decisions (as we did in section 14.4), most often this is used for problems with discrete states and decisions, and where the one-step transition matrix $P(S_{t+1} = s'|S_t = s, a)$ is known (that is, computable).

The field of discrete Markov decision processes has enjoyed a rich theoretical history, largely because of the elegance of discrete states and actions, and the assumption that we can compute expectations over W_{t+1}. This theory seems to have been self-perpetuating, since it is not supported by a class of well-motivated applications. However, as we see in this and later chapters, it has provided the foundation for powerful and practical approximation strategies.

The basic backward dynamic programming strategy used for discrete dynamic programming suffers from what we have identified as the three curses of dimensionality:

1) State variables - As the state variable grows past three or four dimensions, the number of states tends to become too large to enumerate. In particular, there are many applications where some (or all) of the dimensions of the state variable are continuous.

2) Decision variables - Enumerating all possible decisions tends to become intractable if there are more than three or four dimensions, unless it is possible to significantly prune the number of decision using constraints. Problems with more than three

Copyright © 2021 John Wiley & Sons, Inc.

or four dimensions tend to require special structure such as convexity. For this reason, we adopted the classical notation of discrete actions a in chapter 14.4, but for reasons we make clear shortly, this chapter reverts back to our standard notation x for decisions, where we are going to allow x to be multidimensional and continuous.

3) Exogenous information variables - We assume that our exogenous information $W_t \in \mathcal{W} = \{w_1, \ldots, w_L\}$ and let

$$p_t^W(w|s,x) = \mathbb{P}[W_t = w|s,x].$$

As we pointed out in section 9.7 finding the one-step transition matrix requires computing the expectation

$$\begin{aligned}\mathbb{P}(s'|S_t^x = (s,x)) &= \mathbb{E}_{W_{t+1}}\{\mathbb{1}_{\{s'=S^M(s,x,W_{t+1})\}}|S_t = s, x_t = x\} \\ &= \sum_{w\in\mathcal{W}} p_{t+1}^W(W_{t+1} = w|s,x)\mathbb{1}_{\{s'=S^M(s,x,w)\}}. \quad (15.1)\end{aligned}$$

However, if W_{t+1} is a vector or continuous (instead of the discrete outcomes in $\mathcal{W}$), this becomes computationally intractable.

These computational issues have motivated the development of fields with names like "approximate dynamic programming," "heuristic dynamic programming" (an older term used in engineering), "adaptive dynamic programming," (a term adopted in engineering after 2010), "neuro-dynamic programming," or "reinforcement learning," (the highly popular field that evolved within computer science). All of these approaches are effectively a form of "forward approximate dynamic programming" since they are all based on the principle of stepping forward in time. Many authors (including this author) have assumed that if you cannot do "backward dynamic programming" (that is, the method descriped in section 14.3), then you need to turn to "approximate dynamic programming" (which means forward approximate dynamic programming). This chapter challenges this notion.

This chapter presents a strategy known as *backward approximate dynamic programming*, which has the notable feature that it can handle multidimensional (and continuous) state variables and exogenous information variables. In addition, under the right conditions, it can also handle multidimensional (and continuous) decision variables. In other words, backward approximate dynamic programming overcomes all three curses of dimensionality. However, it still struggles with the same challenge of any method based on approximating the value function: The quality of the policy depends heavily on how well we can approximate the value function, and there are many problems where high quality approximations are simply not possible. At the end of this chapter, we are going to present some strong empirical evidence supporting its effectiveness.

15.1 BACKWARD APPROXIMATE DYNAMIC PROGRAMMING FOR FINITE HORIZON PROBLEMS

We are going to start by illustrating backward approximate dynamic programming for finite horizon problems, which parallels backward dynamic programming that we introduced in chapter 14. We begin using classical lookup tables for the value functions, and then transition to continuous approximations.

While we will see that forward ADP methods can be quite powerful, we are going to first present the idea of backward approximate dynamic programming, which has received

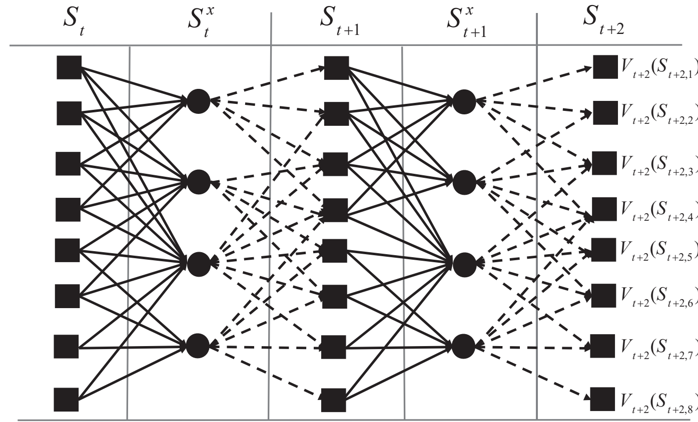


Figure 15.1 Illustration of transitions from pre-decision S_t to post-decision S_t^x to pre-decision S_{t+1} and so on..

comparatively little attention in the research literature. Backward ADP can be viewed as an implementation of classical backward dynamic programming (see the algorithm in figure 14.3) that uses sampling of states and exogenous information to avoid enumerating state spaces and information spaces. We still need to optimize over decisions, but this opens up the potential of exploiting structure such as concavity (convexity if minimizing) to use solvers for high-dimensional decisions.

In addition to scaling nicely to complex problems, we are going to close by presenting some empirical evidence supporting the use of backward ADP. However, as with any approximation method, we cannot make any broad statements about the performance of backward ADP over forward ADP methods (or any of the other classes of policies). It should be viewed as a powerful tool in the toolbox of any sequential decision scientist.

15.1.1 Some preliminaries

We start by writing Bellman's equation broken into two steps: from pre-decision state S_t to post-decision state S_t^x, and then from post-decision state S_t^x to the next pre-decision state S_{t+1}:

$$V_t(S_t) = \max_{x_t} \big(C(S_t, x_t) + V_t^x(S_t^x)\big), \tag{15.2}$$

$$V_t^x(S_t^x) = \mathbb{E}_{W_{t+1}} \{V_{t+1}(S_{t+1})|S_t^x\}, \tag{15.3}$$

where

$$S_t^x = S^{M,x}(S_t, x_t),$$
$$S_{t+1} = S^{M,W}(S_t^x, W_{t+1}).$$

These steps are illustrated in figure 15.1.

The computational challenges associated with these equations include:

- Computing $V_t(S_t)$ for each (presumably discrete) pre-decision state S_t in equation (15.2).

- Optimizing over x_t if x_t is a vector in equation (15.2).
- Computing $V_t^x(S_t^x)$ for each post-decision S_t^x in equation (15.3).
- Computing the expectation $\mathbb{E}_{W_{t+1}}$ over the random variable W_{t+1} in equation (15.3).

We are going to break down these computational challenges one step at a time, as follows:

1) Sampled states with lookup tables - The core idea of backward ADP is to avoid enumerating the entire state space by using a sampled set of states instead. In this first stage, we will still use a lookup table representation of the value functions, and we will also assume we can do full expectations, and maximize over all decisions (which generally means a not-too-large set of discrete decisions).

2) Sampled expectations - Here we are going to replace the exact expectation over W_{t+1} with a sampled approximation.

3) Parametric approximations of the value function - Here we replace the lookup table representation of the value function with a parametric (or nonparametric) approximation which helps with both the computation of value function approximations.

4) Decisions - There are two strategies for handling multidimensional (possibly high dimensional) decisions:

a) We can replace the maximization over decisions with a maximization over a sampled set.

b) If we use a parametric approximation for $V_t^x(S_t^x)$, we may be able to solve equation (15.2) using classical optimization methods (linear, nonlinear or integer programming).

We are going to start by describing backward ADP using lookup table models for the value function, and then we are going to transition to using continuous approximations.

15.1.2 Backward ADP using lookup tables

The basic idea of backward approximate dynamic programming is to perform classical backward dynamic programming, using equations (15.2) - (15.3), but instead of enumerating all the states $\mathcal{S}$, we work with a sampled set $\hat{\mathcal{S}}$. We begin by illustrating the strategy using lookup table approximations for the value function approximations. This closely parallels classical backward dynamic programming (see, for example, equation (14.3)).

For now we are going to make the assumption (true for some, but hardly all, applications) that the post-decision state space $\mathcal{S}^x$ is “not too large.” By contrast, we are going to allow the pre-decision state space $\mathcal{S}$ to be arbitrarily large. This situation arises frequently when there is information needed to make a decision, but which is no longer needed once a decision has been made. Some examples where this arises are:

EXAMPLE 15.1

As a car traverses from node i to node j on a transportation network, it incurs random costs $\hat{c}_{ij}$ which it learns when it first arrives at node i. The (pre-decision) state when it arrives at node i is then $S = (i, (\hat{c}_{ij})_j)$. After making the decision to traverse from i to some node j' (but before moving to j'), the post-decision state is $S^x = (j)$, since we no longer need the realization of the costs $(\hat{c}_{ij})_j$.

EXAMPLE 15.2

A truck driver arrives in city i and learns a set $\mathcal{L}_i$ of loads that need to be moved to other cities. This means when it arrives at i that the state of our dri5ver is $S = (i, \mathcal{L}_i)$. Once the driver chooses a load $\ell \in \mathcal{L}_i$, but before moving to the destination of load ℓ, the (post-decision) state is $S^x = (\ell)$ (or we might use the destination of load ℓ).

EXAMPLE 15.3

A cement truck is given a set of orders to deliver set to a set of work sites. Let R_t be the inventory of cement, and let $\mathcal{D}_t$ be the set of construction sites needing deliveries (the set includes how much cement is needed by each site). The decision that needs to be made by the cement plant is how much cement to make to replenish inventory. The pre-decision state is $S_t = (R_t, \mathcal{D}_t)$, while the post-decision state is $S_t^x = R_t^x$ which is the amount of inventory left over after making all the deliveries.

In each of these examples, the number of pre-decision states may be extremely large. Instead of looping over all states in $\mathcal{S}$ (as we had to do in figure 15.5), we are going to take a sample $\hat{\mathcal{S}}$ which is of manageable size. We see the power of Monte Carlo simulation in that the state variables can be both continuous and high-dimensional, since we control the number of samples in $\hat{\mathcal{S}}$. The only caveat is that we have to pre-specify a sampling region, which means we have to know something about the range of values of each dimension of S_t.

In addition to enumerating the post-decision states, we also assume (for now):

- There are a discrete set of decisions $x_t \in \{x_1, x_2, \ldots, x_K\}$ that we can search over.
- There are discrete outcomes $W_{t+1} \in \{w_1, \ldots, w_L\}$.
- We know the probability $p_t^W(w_\ell) = \mathbb{P}(W_{t+1} = w_\ell | S_t^x)$.

The steps of the algorithm are described in detail in figure 15.3, but we refer to figure 15.2 to explain the idea. The pre-decision states are depicted as squares while post-decision states are circles. We represent the states in our sampled set $\hat{\mathcal{S}}$ using the black squares. Assuming we know $\overline{V}_{t+2}(s)$ for states $s \in \hat{\mathcal{S}}$, we compute the value $\overline{V}^x_{t+1}(s)$ for each post-decision state s in $\mathcal{S}^x$ by taking the expectation over all random outcomes that take us to states in our sampled set $\hat{\mathcal{S}}$, given by the equation

$$V_{t+1}^x(S_{t+1}^x) = \frac{\sum_{\ell=1}^L p_{t+2}^W(w_\ell)\overline{V}_{t+2}(S_{t+2}(w_\ell))\mathbb{1}_{\{S_{t+2}(w_\ell)\in\hat{\mathcal{S}}\}}}{\sum_{\ell=1}^L p_{t+2}^W(w_\ell)\mathbb{1}_{\{S_{t+2}(w_\ell)\in\hat{\mathcal{S}}\}}}, \quad (15.4)$$

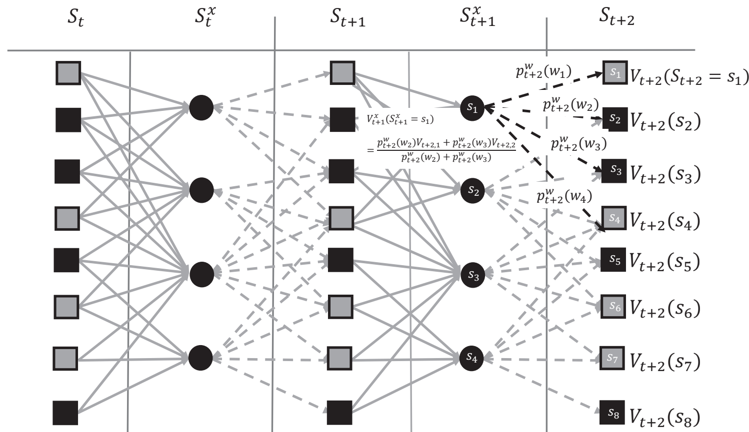


Figure 15.2 Calculation of the value of the post-decision state S_{t+1}^x using full expectation.

where $S_{t+2}(w) = S^M(S_{t+1}^x, w)$. Note that equation (15.4) only includes transitions to values of S_{t+2} in the sampled set $\hat{\mathcal{S}}$, which means that we have to normalize the probabilities so that the probabilities of the outcomes that transition to states in $\hat{\mathcal{S}}$ sum to one.

This quickly raises a potential problem. What if none of the random outcomes take us to states in $\hat{\mathcal{S}}$? When this happens, we choose a subset of random outcomes from a post-decision state, find the pre-decision states that these outcomes take us to, and then add these states to the sampled set $\hat{\mathcal{S}}$. We then repeat the calculation.

Once we have the value of being in each post-decision state, we then step back to find the value of being in each sampled pre-decision state, which is depicted in figure 15.4. Since we assume we have computed the value of being in each post-decision state, finding the value of being in any pre-decision state involves simply searching over all decisions and finding the decision with the highest one-period reward plus downstream value.

15.1.3 Backward ADP algorithm with continuous approximations

Now that we have sketched the basic idea of backward ADP, we are going to outline a fully scalable algorithm that can handle multidimensional and continuous state variables (pre-decision S_t and post-decision S_t^x), decisions x_t, and exogenous information W_{t+1}. We do this by using appropriately designed continuous approximations of the value function around the post-decision state variable.

A sketch of the algorithm is given in figure 15.5. This algorithm has some nice features:

- Both the pre- and post-decision states S_t and S_t^x can be multidimensional and continuous.
- The exogenous information W_t can also be multidimensional and continuous, as long as we have some mechanism for sampling the random variable. This may come from an underlying mathematical model, or it may come from historical observations.
- The decision x_t may be multidimensional and continuous (or discrete), but algorithms for solving multidimensional decision problems typically require concavity of $\left(C(\hat{s}_{t+1}^n, x) + \overline{V}_{t+1}^x(\hat{s}_{t+1}^{n,x})\right)$ (convexity if minimizing). This is where some care

Step 0. Initialization:

- **0a.** Initialize the terminal contribution $V_T(S_T)$.
- **0b.** Create a sampled set of pre-decision states $\hat{\mathcal{S}}$ (we assume we can use this same sample each time period).
- **0c.** Create a full set of post-decision states $\mathcal{S}^x$ (presumably a manageable size).
- **0d.** Set $t = T - 1$.

Step 1a. Step backward in time $t = T, T-1, \ldots, 0$:

Compute the value of each post-decision state:

- **Step 2a.** Initialize pre-decision value function approximation $\overline{V}_t(s) = -M$.
- **Step 2b.** Loop over the sampled set of pre-decision states $s \in \hat{\mathcal{S}}$.
- **Step 2c.** Loop over each decision $x \in \mathcal{X}(s)$:
  - **Step 3a.** Compute $Q_t(s,x) = C(s,x) + \overline{V}^x_t(s' = S^{M,x}(s,x))$.
  - **Step 3b.** If $Q_t(s,x) > \overline{V}_t(s)$ then set $\overline{V}_t(s) = Q_t(s,x)$.

Compute the value of each sampled pre-decision state:

- **Step 4a.** Loop over the full set of post-decision states $s^x \in \mathcal{S}^x$.
- **Step 4b.** Step back in time: $t = t - 1$.
- **Step 4b.** Initialize post-decision value function approximation $\overline{V}^x_t(s) = -M$.
  - **Step 4a.** Initialize $Q(s,x) = 0$.
  - **Step 4b.** Initialize total probability $\rho = 0$.
  - **Step 4c.** Loop over each $w \in \mathcal{W}$:
    - **Step 5a.** Compute $Q_t(s,x) = Q_t(s,x) + \mathbb{P}(w|s,x)\overline{V}_{t+1}(s' = S^M(s,x,w))$.
    - **Step 5b.** $\rho = \rho + \mathbb{P}(w|s,x)$.
  - **Step 4d.** If $\rho > 0$ then (we have to normalize $Q_t(s,x)$ in case $\rho < 1$):
    - **Step 5c.** $Q_t(s,x) = Q_t(s,x)/\rho$
  - **Else:** Get here if $\rho = 0$, which means there were no random transitions to states in $\hat{\mathcal{S}}$:
    - **Step 5d.** Choose a sample of outcomes $\hat{w}$ (at least one), find the downstream pre-decision state $\hat{s} = S^{M,W}(s,\hat{w})$, and add each $\hat{s}$ to $\hat{\mathcal{S}}$.
    - **Step 5e.** Return to step 4a.

Step 1b. Return the values $\overline{V}_t(s)$ for all $s \in \mathcal{S}$ and $t = 0, \ldots, T$.

Figure 15.3 A backward dynamic programming algorithm using lookup tables.

might have to be put into the choice of architecture for the value function approximation.

An open question is: how well does the method work? The approximation for time t depends on the approximation for $t+1$, which means the errors in the approximation for $t+1$ propagate backward to t and, in fact, accumulate. Section 15.4 reports on three sets of empirical benchmarking experiments that support the accuracy and efficiency of backward approximate dynamic programming. However, we can obtain stronger results when we apply these ideas in the context of a stationary (steady state) problem, an idea that has evolved in the literature under the name "fitted value iteration."

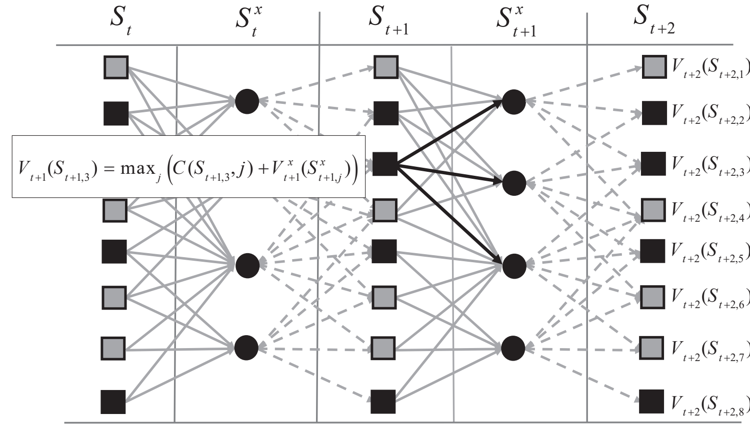


Figure 15.4 Calculation of value of pre-decision state S_{t+1} using full maximization.

---

0) Assume we have a value function approximation $\overline{V}_T(s)$.

1) Perform N samples for $n = 1, \ldots, N$:

- **1a)** Randomly sample a post-decision state $\hat{s}_t^{x,n}$ from the set $|Scalhat^x$.
- **1b)** Find a sample realization of $\hat{w}_{t+1}^n$ of the random variable W_{t+1} given that we are in state $\hat{s}_t^x$.
- **1c)** Simulate our way from $\hat{s}_t^{x,n}$ to $\hat{s}_{t+1}^n$ using $\hat{s}_{t+1}^n = S^{M,W}(\hat{s}_t^{x,n}, \hat{w}_{t+1}^n)$.
- **1d)** Compute a sample estimate $\hat{v}_{t+1}^n$ of the value of being in pre-decision state $\hat{s}_{t+1}^n$ using

$$\hat{v}_{t+1}^n = \max_x \left(C(\hat{s}_{t+1}^n, x) + \overline{V}_{t+1}^x(\hat{s}_{t+1}^{n,x})\right) \tag{15.5}$$

where $\hat{s}_{t+1}^{n,x} = S^{M,x}(\hat{s}_t^n, x_t^n)$, and where x_t^n is the value of x that optimizes equation (15.5). We are now going to associate the value $\hat{v}_{t+1}^n$ with the previous post-decision state $\hat{s}_t^{x,n}$.

2) From step 1, we compile a set of observations $(\hat{s}_t^n, \hat{v}_{t+1}^n), n = 1, \ldots, N$.

3) Use the dataset $(\hat{s}_t^n, \hat{v}_{t+1}^n), n = 1, \ldots, N$ to fit a statistical model $\overline{V}_t^x(s)$ using any of the statistical methods in chapter 3 (but here we are doing batch learning). Some of the methods that have proven successful in this context are described below in section 15.3.

4) Step back one time period and repeat until $t = 0$.

Figure 15.5 A backward dynamic programming algorithm for multidimensional applications.

15.2 FITTED VALUE ITERATION FOR INFINITE HORIZON PROBLEMS

Most of this book focuses on finite horizon problems, since these represent the problems most often encountered in practice. However, the literature on Markov decision processes, as can be seen in the presentation in chapter 14, has emphasized the steady state version of Bellman's equation which is written:

$$V(s) = \max_{x \in \mathcal{X}} \left(C(s,x) + \gamma \mathbb{E}_W\{V(S^M(s,x,W))|s\}\right).$$

where $s' = S^M(s,x,W)$ is the state we land in given that we are now in state s, make decision x, and then observe W. We write the value function explicitly as a function of the

transition function $S^M(s,x,W)$ to make the dependence on W explicit. Needless to say, computing this expectation is problematic, especially inside a max operator. Instead, we used a sampled estimate by choosing a random sample $\mathcal{W} = \{w_1, w_2, \ldots, w_L\}$.

The basic idea follows the steps of backward ADP. We choose a sample of states $\hat{\mathcal{S}} = \{\hat{s}_1, \ldots, \hat{s}_m, \ldots, \hat{s}_M\}$. Assume we have an approximate value function $\overline{V}^{n-1}(s)$. Then, given $\overline{V}^{n-1}(s)$, we sample $\hat{s}_m \in \hat{\mathcal{S}}$ and compute

$$\hat{v}^n_m = \max_{x \in \mathcal{X}} \left(C(\hat{s}_m, x) + \gamma \frac{1}{L} \sum_{\ell=1}^{L} \left(\overline{V}^{n-1}(S^M(\hat{s}_m, x, w_\ell)) | s \right) \right). \tag{15.6}$$

Repeat equation (15.6) for $m = 1, \ldots, M$ until we have compiled a dataset $(\hat{s}_m, \hat{v}^n_m)$ for $m = 1, \ldots, M$. Note that we index the value function approximation $\overline{V}^n(s)$ by iteration n, but the sampled states $\hat{s} \in \hat{\mathcal{S}}$ are the same from one iteration to the next.

The next step is to use the dataset $(\hat{s}_m, \hat{v}^n_m)_{m=1}^M$ to create an updated value function approximation $\overline{V}^n(s)$, using any of the approximation architectures in chapter 3. Of course, solving equation (15.6) is more difficult than solving for $\hat{v}^n$ in equation (15.5) because we have chosen to illustrate fitted value iteration using value functions that depend on the pre-decision state, forcing us to use the sampled representation of the expectation. We could use the same strategy as we did in the finite-horizon case and compute the value function around the post-decision state. Similarly, we could use the sampled representation of the expectation illustrated in this section in the finite-horizon setting. We have decided to illustrate both methods, but either can be used in either setting.

The only real difference between the finite and infinite horizon versions is that the finite horizon algorithm involves a single backward pass over the horizon. There is no notion of convergence. By contrast, we can repeat our process for updating $\overline{V}^n(s)$ in the infinite horizon case for as many iterations as we like, opening the door to questions about convergence. Recall that we could obtain strict bounds on the error when we were using lookup table representations and assuming that we could compute the one-step transition matrix (see section 14.12.2).

We made the point in chapter 4 that classical discrete Markov decision processes, where we assume that the one-step transition matrix is known, is actually a deterministic problem (see section 4.2.5), as is any stochastic problem where the expectation can be computed exactly. In fact, in section 4.3 we made the point that replacing the expectation with a sampled approximation, as we are doing in equation (15.6), is simply replacing the original expectation that we could not compute, with an approximate expectation that we can compute. Once we do so, we are effectively turning our exact "deterministic" problem into an approximate "deterministic" problem. But if we continue to use a lookup table representation, we still suffer from the curse of dimensionality in the state space.

It is possible to show convergence results similar to those for the exact, discrete dynamic programming methods, but it requires an approximating architecture that is sufficiently flexible to allow arbitrarily accurate fits at the sampled states. This would not be possible if we were to use a low-dimensional parametric architecture (such as a quadratic fit). Gaussian process regression, kernel regression and neural networks are all approximation methods that can produce very accurate approximations, but any time you use these high-dimensional architectures, you run the risk of overfitting to noisy observations unless you have exceptionally large samples. So, pick your poison.

15.3 VALUE FUNCTION APPROXIMATION STRATEGIES

We illustrated the basic idea of backward approximate dynamic programming using a standard lookup table representation for the value function, but this would quickly cause problems if we have a multidimensional state (the classic curse of dimensionality). In this section, we suggest three strategies for approximating value functions that mitigate this problem to some degree.

15.3.1 Linear models

Arguably the most natural strategy for approximating the value function is to fit a statistical model, where the most natural starting point is a linear model of the form

$$\overline{V}_t(S_t|\theta_t) = \sum_{f \in \mathcal{F}} \theta_{tf}\phi_f(S_t).$$

Here, $\phi_f(S_t)$ are a set of appropriately chosen features. For example, if S_t is a continuous scalar (such as price), we might use $\phi_1(S_t) = S_t$ and $\phi_2(S_t) = S_t^2$.

The idea is very simple. For each $\hat{s}$ in our sampled set of pre-decision states $\hat{\mathcal{S}}$, compute a sampled estimate $\hat{v}_t^n$ of the value of being in a state s^n

$$\hat{v}_t^n = \arg\max_x \left(C(\hat{s}^n, x) + \mathbb{E}\{\overline{V}_{t+1}(S_{t+1})|\hat{s}^n\}\right),$$

where $S_{t+1} = S^M(\hat{s}^n, x, W_{t+1})$.

Now we have a set of data $(\hat{s}^n, \hat{v}_t^n)$ for $n = 1, \ldots, |\hat{\mathcal{S}}|$. We can use this dataset to estimate any statistical model $\overline{V}_t(S_t|\theta_t)$ which gives us an estimate of the value of being in every state, not just the sampled states. For example, assume we have a linear model (remember this means linear in the parameters)

$$\begin{aligned}\overline{V}_t(S_t|\bar{\theta}_t) &= \bar{\theta}_{t1}\phi_1(S_t) + \bar{\theta}_{t2}\phi_2(S_t) + \bar{\theta}_{t3}\phi_3(S_t) + \ldots, \\ &= \sum_{f \in \mathcal{F}} \theta_{tf}\phi_f(S_t),\end{aligned}$$

where $\phi_f(S_t)$ is some feature of the state. This might be the inventory R_t (money in the bank, units of blood), or R_t^2, or $\ln(R_t)$. Create the (column) vector ϕ^n using

$$\phi^n = \begin{pmatrix} \phi_1^n \\ \phi_2^n \\ \vdots \\ \phi_F^n \end{pmatrix}$$

where $\phi_f^n = \phi_f(S_t^n)$.

Let $\hat{v}_t^n$ be computed using (15.7), which we can think of as a sample realization of the estimate $\overline{V}_t^{n-1}(S_t)$. We can think of

$$\hat{\varepsilon}_t^n = \overline{V}_t^{n-1}(S_t) - \hat{v}_t^n$$

as the "error" in our estimate. Using the methods we first introduced in section 3.8.1, we can update our estimates of the parameter vector $\bar{\theta}_t^{n-1}$ using

$$\bar{\theta}_t^n = \bar{\theta}_t^{n-1} - H_t^n \phi_t^n \hat{\varepsilon}_t^n, \tag{15.7}$$

where H_t^n is a matrix computed using

$$H_t^n = \frac{1}{\gamma^n} M_t^{n-1}, \tag{15.8}$$

where M_t^{n-1} is an $|\mathcal{F}|$ by $|\mathcal{F}|$ matrix which is updated recursively using

$$M_t^n = M_t^{n-1} - \frac{1}{\gamma_t^n}(M_t^{n-1}\phi_t^n(\phi_t^n)^T M_t^{n-1}). \tag{15.9}$$

γ_t^n is a scalar computed using

$$\gamma_t^n \quad = \quad 1 + (\phi_t^n)^T M_t^{n-1} \phi_t^n. \tag{15.10}$$

Parametric approximations are particularly attractive because we get an estimate of the value of being in *every* state from a small sample. The price we pay for this generality is the errors introduced by our parametric approximation.

15.3.2 Monotone functions

There are a number of sequential decision problems where the state variable has three to six or seven dimensions, which tends to be the range where the state space is too large to estimate value functions using lookup tables. There are, however, a number of applications where the value function is monotone in each dimension, which is to say that as the state variable increases in each dimension, so does the value of being in the state. Some examples include:

- Optimal replacement of parts and equipment tend to exhibit value functions which are monotone in variables describing the age and/or condition of the parts.
- The problem of controlling the number of patients enrolled in clinical trials produces value functions that are monotone in variables such as the number of enrolled patients, the efficacy of the drug, and the rate at which patients drop out of the study.
- Initiation of drug treatments (statins for cholesteral, metformin for lowering blood sugar) result in value functions that are monotone in health metrics such as cholesterol or blood sugar, the age of a patient and their weight.
- Economic models of expenditures tend to be monotone in the resources available (e.g. personal savings), and other indices such as stock market, interest rates, and unemployment.

Monotonicity can be exploited when we are using a lookup table representation of a value function. Assume that a state s consists of four dimensions $(s_{t1}, s_{t2}, s_{t3}, s_{t4})$, where each dimension takes on one of a set of discrete values, such as $s_{t2} \in \{s_{t2,1}, s_{t2,2}, s_{t2,3}, \ldots, s_{t2,J_2}\}$. Assume we have a sampled estimate of the value of being in state $\hat{s}^n$, which we might compute using

$$\hat{v}_t^n(\hat{s}^n) = \max_x \big(C(\hat{s}^n, x) + \mathbb{E}_{W_{t+1}}\{\overline{V}_t^{n-1}(S_{t+1})|\hat{s}^n\}\big),$$

where $S_{t+1} = S^M(\hat{s}^n, x, W_{t+1})$. We might then use our sampled estimate (regardless of how it is found) to update the value function approximation at state $\hat{s}^n$ using

$$\overline{V}_t^n(\hat{s}^n) = (1-\alpha_n)\overline{V}_t^{n-1}(\hat{s}^n) + \alpha_n \hat{v}_t^n(\hat{s}^n).$$

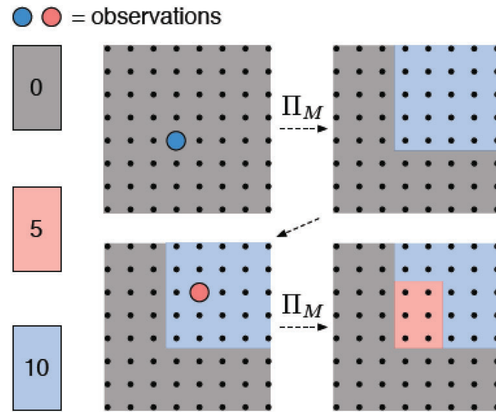


Figure 15.6 Illustration of the use of monotonicity. Starting from upper left: 1) Initial value function all 0, with observation (blue dot) of 10; 2) using observation to update all points to the right and above to 10; 3) new observation (pink dot) of 5; 4) updating all points to the left and below to 5. (Graphic due to Daniel Jiang)

We assume that $\overline{V}_t^{n-1}(s)$ is monotone in s before the update. Assume that $s' \succ s$ means that each element $s'_{ij} \geq s_{ij}$. Then if $\overline{V}_t^{n-1}(s)$ is monotone in s, then $s' \succ s$ means that $\overline{V}_t^{n-1}(s') \geq \overline{V}_t^{n-1}(s)$. However, we cannot assume that this is true of $\overline{V}_t^n(s)$ just after we have done an update for state s_t^n. We can quickly check if $\overline{V}_t^n(s) \leq \overline{V}_t^n(s')$ for each s' with at least one element that is larger than the corresponding element of s.

The idea is illustrated in 15.6. Starting with the upper left corner, we start with an initial value function $\overline{V}(s) = 0$, and make an observation (the blue dot) of 10 in the middle. We then use the monotone structure to make all points to the right and above of this point to equal 10. We then make an observation of 5, and use this observation to update all the points to the left and below the last observation.

Figure 15.7 shows snapshots from a video where monotonicity is being used to update a two-dimensional function. Again starting from the upper right, the first three screenshots were from the first 20 iterations, while the last one (lower right) was at the end, long after the function had stopped changing.

Monotonicity is an important structural property. When it holds, it dramatically speeds the process of learning the value functions. We have used this idea for matrices with as many as seven dimensions, although at that point a lookup representation of a seven-dimensional function becomes quite large.

There will be situations where a value function is monotone in some dimensions, but not in others. This can be handled (somewhat clumsily) but imposing monotonicity over the subset of states where monotonicity holds. For the remaining states, we have to resort to brute force lookup table methods. If $\bar{s}$ is the set of states where the value function is not monotone, while $\tilde{s}$ is the states over which the value function is monotone (of course, $s = (\tilde{s}, \bar{s})$), then we can think of a value function $\overline{V}(\tilde{s}, \bar{s})$ where we have a value function $\overline{V}(\tilde{s}, \bar{s})$ that is monotone in $\tilde{s}$ for each state $\bar{s}$ (we hope there are not too many of these).

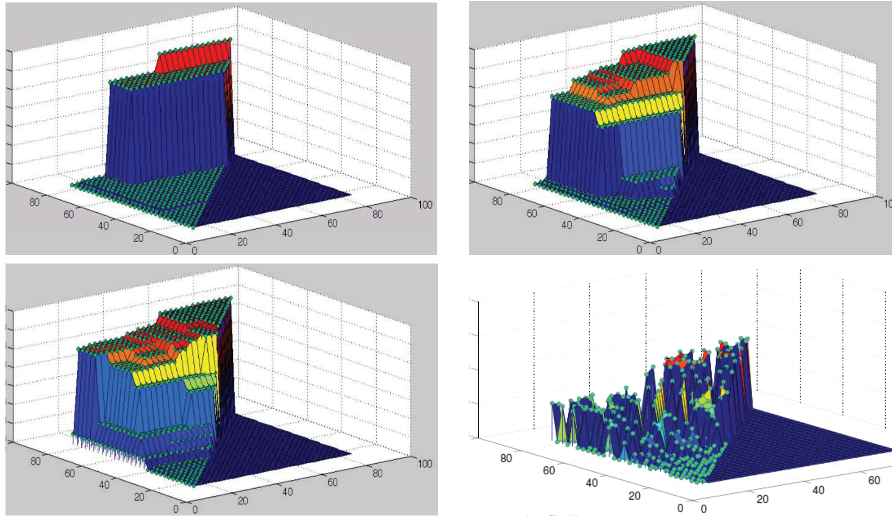


Figure 15.7 Video snapshot of use of monotonicity for a two-dimensional function for three updates; fourth snapshot (lower right) is a value function where monotonicity was not used.

15.3.3 Other approximation models

We encourage readers to experiment with other methods from chapter 3 (or your favorite book in statistics or machine learning). We note that approximation errors will accumulate with backward ADP, so you should not have much confidence that $\overline{V}_t(S_t)$ is actually a good approximation of the value of being in state S_t. However, we have found that even when there is a significant difference between $\overline{V}_t(S_t)$ and the true value function $V_t(S_t)$ (when we can find this), the approximation $\overline{V}_t(S_t)$ may still provide a high quality policy, but there are no guarantees.

15.4 COMPUTATIONAL OBSERVATIONS

As of the writing of this book, backward approximate dynamic programming is a relatively new algorithmic strategy, which is surprising given that it is the approximate analog of classical backward dynamic programming (from chapter 14). The first reference in the literature appears to be 2013. For this reason, we begin with a presentation of several projects we have been directly involved with which produced some form of benchmarking of the solutions produced by backward approximate dynamic programming. We then share some notes on the methodology.

15.4.1 Experimental benchmarking of backward ADP

In this section we report on the empirical benchmarking of backward ADP in three very different settings. The first uses comparisons against the exact, optimal solution computed using the techniques of chapter 14. The second two examples are more complex, making exact solutions impossible. Instead, we compare backward ADP against policies that are

already being used, one for the optimization of a battery storage system and the second for the allocation of resources in Africa by the International Monetary Fund.

Optimization of clinical trials

The problem faced by companies running clinical trials is to make the following decision at each point in time: the drug works (go to market, typically by selling the patent), the drug does not work (cancel the clinical trial), or continue testing. The state variable has three dimensions:

- The number of patients we have tested.
- A two-dimensional belief state, capturing the mean and variance of our estimate of the probability that the drug works.

This means our state variable has a single discrete dimension and two continuous dimensions. If we are willing to live with a discretized version of the continuous dimensions, this is a problem that can be solved optimally using the backward dynamic programming methods we presented in chapter 14. Optimal benchmarks for real problems are quite rare.

The results can be stated very simply:

- The optimal solution required 268 hours on a modern laptop.
- Backward approximate dynamic programming required 20 minutes, and the solution was within 1.2 percent of the performance of the policy produced by the optimal solution.

Optimizing a complex energy storage problem

We were given the problem of optimizing an energy storage device where we had to balance two revenue streams:

- We can use the battery to buy and sell energy from/to the grid. Electricity prices (known as LMPs, or "locational marginal prices") are updated every five minutes and can vary dramatically. Prices that might average $20/megawatt-hour can spike to $1000 or even $10,000.
- Grid operators will pay battery operators to help them with a process called "frequency regulation." Power over the grid fluctuates as a result of the random variations of loads placed on the grid. A grid operator might pay $30 per megawatt of power (each battery has a rated power rating, which gives how *fast* power moves into and out of the battery), but these prices also vary, and can increase to $500 per megawatt per hour or more.

When the grid operator is paying a battery to perform frequency regulation, the grid will send a signal every two seconds whether it wants the battery to charge or discharge (at some percentage of the battery's power rating), or do nothing. The grid never asks the battery to charge (or discharge) for extended periods, so these batteries do not have to be very large. Frequency regulation is purely for short-term smoothing of power variations.

When a battery operator is being paid to perform frequency regulation, then the expectation is that it will comply with the signals from the grid operator (these are known as "RegD" signals in the U.S.). In practice, limitations of the device (it is not just batteries

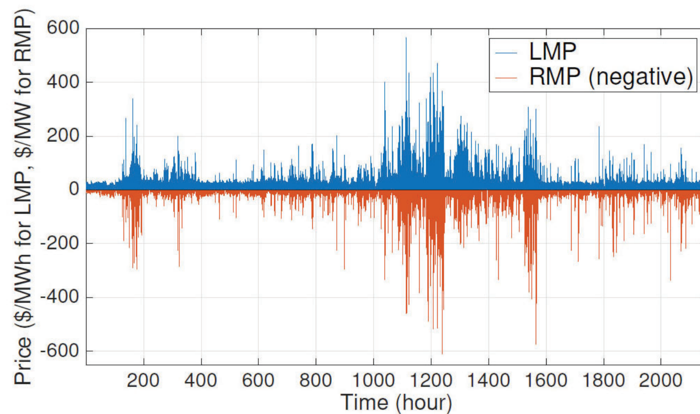


Figure 15.8 Real-time energy prices (blue), and regulation prices (red), January-March, 2015, revealing the high correlation between the two.

that perform this function - any generator, from natural gas turbines to coal plants may perform frequency regulation) mean that the device providing frequency regulation may not have perfect compliance with the RegD signal. For this reason, there are penalties for noncompliance. Figure 15.8 shows a plot of LMP and RegD prices over a period of several months. It indicates the degree of volatility, along with the correlations between the two prices. This is an example of a problem where the modeling of the exogenous information processes is particularly important.

This raised the question: What if a battery operator (batteries typically have perfect compliance) occasionally disobeyed the RegD signal? In particular, what if the grid operator is asking the battery to buy electricity at a time when electricity prices are very high? The battery operator might wish to go against the RegD signal, sell into a high-priced market (this may last for just five minutes), but paying the penalty for noncompliance.

The challenge here is that following a RegD signal is trivial; the battery operator simply follows the RegD signal from the grid operator which specifies when to charge, discharge or do nothing. However, choosing between simply doing what the RegD signal tells us to do, or running against the signal to take advantage of price spikes on the grid, requires optimization-based logic. For example, the grid price may rise, but has it risen enough to suffer the consequences of noncompliance with the RegD signal?

The distinguishing characteristic of this problem is the number of time periods. Decisions are made every 2 seconds, which means there are 43,200 time periods in a day. Standard backward dynamic programming was prohibitive because of the size of the state space as well as the number of time periods. Forward ADP methods, which we will introduce in chapters 16-18 (stay tuned for these presentations) require iterative learning which would simply be too slow. In fact, this was the problem that motivated our first use of backward ADP.

This problem is too hard to compute optimal benchmarks, as we did in the clinical trials problem. However, we have a different benchmark which is extremely demanding: instead of optimizing over the two signals, we can just follow the RegD signal. This is very difficult competition, since we anticipate that the benefits of optimizing across both revenue streams

Month	Backward ADP revenue	Pure RegD policy	Pct. improvement
January	22052	19131	10.27
February	51282	46331	10.68
March	36518	32329	12.95
April	24121	22272	8.3
May	31861	30232	5.39
June	18975	17999	5.42
July	18463	17152	7.64
August	15988	14750	8.39
September	22336	20462	9.16
October	17714	16553	7.01
November	15930	15033	5.97
December	15079	13901	8.47
Annual	290323	266151	9.08

Table 15.1 Comparison of revenues generated from backward ADP, combining revenues from frequency regulation and power purchase, to revenues from a pure frequency regulation policy.

will be modest. This means that we are not in a position to tolerate suboptimal performance since this would threaten the larger revenue stream from just following the RegD signal.

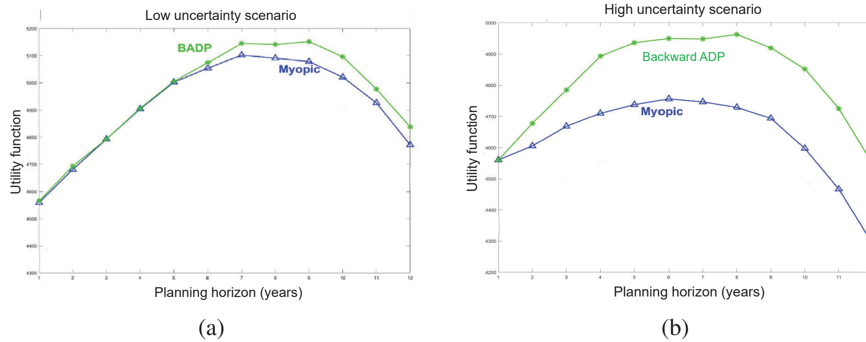
The results are shown in table 15.1. These results show consistent, if modest, improvements from the combined signal produced using backward approximate dynamic programming. Again, we emphasize the challenge of competing against a pure frequency regulation policy which produces over 90 percent of the revenue using a very simple rule that is easy to follow.

Resource allocation in Africa

Our last demonstration involves a complex resource allocation problem faced by the International Monetary Fund (IMF) among projects within Africa. The widely-used approach for solving this problem is a single-period linear program that optimized a complex utility function for capturing the state of a country over the course of a year. The utility function would capture metrics about the economy, social metrics such as poverty, investments in infrastructure, and measures of instability (such as assassinations). The decisions were how much to invest in different projects, such as roads, education, health, and power generators. Given that these decisions cover resources being allocated across all countries in Africa, and all projects, it is a high-dimensional decision, with a very high-dimensional (and largely continuous) state vector.

The state of the art for this problem is the use of a linear program that optimizes the benefits within a single year, although it was clear that some investments had multiyear horizons. This was also a problem with tremendous uncertainties. In any given year insurgencies could arise and challenge the stability of a country. The emergence of diseases, or discoveries of natural resources, were frequent examples of high-impact sources of uncertainty.

This problem was solved using backward approximate dynamic programming, and compared to a myopic policy that is widely used in practice. The results are shown in figure



(a) (b)

Figure 15.9 Performance of the policy produced by backward ADP using (a) low uncertainty and (b) high uncertainty in future forecasts.

15.9, which reports on two sets of simulations. Figure 15.9(a) shows the results of backward ADP for a simulation with relatively low noise, while figure 15.9(b) shows the results for a setting with significant sources of uncertainty. Backward ADP outperformed the standard myopic policy for both the low noise and high noise situations. It did particularly well in the high noise environment, which is precisely the conditions where someone might say "we have so much uncertainty about the future, why plan for it?"

This application is a nice demonstration of backward ADP in a complex, high-dimensional resource allocation problem. In fact, it is a problem which clearly needs a policy in the lookahead class, but where direct lookahead policies (which we introduced in chapter 11, and cover in much more detail in chapter 19) are not an obvious approach.

15.4.2 Computational notes

Some thoughts to keep in mind while designing and testing algorithms using backward approximate dynamic programming:

Approximation architectures - It is possible to use any of the statistical learning methods described in chapter 3.8.1 (or your favorite book on statistics/machine learning). We note that most of the methods in this book involve adaptive learning (this is the focus of chapter 3.8.1), but with backward ADP, we actually return to the more familiar setting (in the statistical learning community) of batch learning. Following standard advice in the specification of any statistical model, make sure that the dimensionality of the model (measured by the number of parameters) is much smaller than the number of observations to avoid overfitting.

Tuning - Virtually all adaptive learning algorithms have tunable parameters, and this is the Achilles heel of this entire approach to solving stochastic optimization problem. In chapter 9, section 9.11 summarizes four problem classes (see table 9.3), where classes (1) and (4) are posed as finding the best learning policy. These "learning policies" represent the process of finding the best search algorithm, which includes tuning the parameters that govern a particular class of algorithm. In practice, this search for the best learning policy (or equivalently, the search for the best search algorithm) is typically done in an ad hoc way. There are thousands of papers which will prove asymptotic convergence, but the actual design of an algorithm depends on ad hoc testing.

Validating - A major challenge with any approximation strategy, backward ADP included, is validation. Backward ADP can work extremely well on problems where the value function is a fairly good approximation of the true value function, but there are no guarantees. It helps to have a good benchmark (in this case, the widely accepted myopic policy served this role) for comparison.

Performance - We have obtained exceptionally good performance on some problem classes, including energy storage problems with thousands of time periods. In comparisons against optimal policies (obtained using the methods from chapter 14 for low-dimensional problem instances), we have obtained solutions that were over 95 percent of optimality, but on occasions the performance was as low as 70 percent when we did a poor job with the approximations.

15.5 BIBLIOGRAPHIC NOTES

Section 15.1 - The first use of the term "backward approximate dynamic programming" in the published literature is in Senn et al. (2014), which is based on Senn's Ph.D. dissertation (in German), which appeared in 2013. This work was for a finite-horizon deterministic control problem. Cheng et al. (2018*a*) used backward ADP for a stochastic energy storage problem using the idea of a low-rank approximation for the value function. Cheng et al. (2018*b*) used a simpler linear architecture for an energy storage problem and showed that it was quite effective.

Section 15.2 - Fitted value iteration is basically backward approximate dynamic programming for infinite horizon problems. Szepesvári & Munos (2005) and Munos & Szepesv (2008) were the earliest papers that use the term "fitted value iteration." Fitted value iteration is a form of approximate value iteration which we consider in depth in chapter 17, which focuses on forward algorithms.

Section 15.4.1 - The work on backward ADP for clinical trials is taken from Tian et al. (2021). The experimental work for energy storage is taken from Cheng et al. (2018*b*). The work on allocating aid in Africa is taken from Aboagye & Powell (2018), which extended the seminal paper by Collier & Dollar (2002) which proposed the myopic policy for the same problem.

EXERCISES

Review questions

15.1 Contrast backward approximate dynamic programming for finite horizon problems versus infinite horizon problems in terms of the concept of "convergence" for each one.

Computational exercises

15.2 We are going to solve the continuous budgeting problem presented in section 14.4.2 using backward approximate dynamic programming. The problem starts with R_0 resources which are then allocated over periods 0 to T. Let x_t be the amount allocated in period t

with contribution

$$C_t(x_t) = \sqrt{x_t}.$$

Assume that $T = 20$ time periods.

a) Use the results of section 14.4.2 to solve this problem optimally. Evaluate your simulation by simulating your optimal policy 1000 times.

b) Use the backward ADP algorithm described in figure 15.5 to obtain the value function approximations using

$$\overline{V}_t(R_t) = \theta_{t0} + \theta_{t1}\sqrt{x_t}.$$

Use linear regression (either the methods in section 3.7.1, or a package) to fit $\overline{V}_t(R_t)$. Then, simulate this policy 1000 times (ideally using the same sample paths as you used for part (a)). How do you think θ_{t0} and θ_{t1} should behave?

c) Use the backward ADP algorithm described in figure 15.5 to obtain the value function approximations using

$$\overline{V}_t(R_t) = \theta_{t0} + \theta_{t1}R_t^x + \theta_{t2}(R_t^x)^2,$$

where R_t^x is the post-decision resource state $R_t^x = R_t - x_t$ (which is the same as R_{t+1} since transitions are deterministic).

Use linear regression (either the methods in section 3.7.1, or a package) to fit $\overline{V}_t(R_t)$. Then, simulate this policy 1000 times (ideally using the same sample paths as you used for part (a)).

15.3 Repeat exercise 15.2, but this time use

$$C(x_t) = \ln(x_t).$$

For part (b), use

$$\overline{V}_t(R_t) = \theta_{t0} + \theta_{t1}\ln(x_t).$$

15.4 In this exercise you are going to solve a simple inventory problem using Bellman's equations, to obtain an optimal policy. Then, the exercises that follow will have you implement various backward ADP policies that you can compare against the optimal policy you obtain in this exercise. Your inventory problem will span T time periods, with an inventory equation governed by

$$R_{t+1} = \max\{0, R_t - \hat{D}_{t+1}\} + x_t.$$

Here we are assuming that product ordered at time t, x_t, arrive at $t+1$. Assume that $\hat{D}_{t+1}$ is described by a discrete uniform distribution between 1 and 20.

Next assume that our contribution function is given by

$$C(S_t, x_t) = 50\min\{R_t, \hat{D}_{t+1}\} - 10x_t.$$

a) Find an optimal policy by solving this dynamic program exactly using classical backward dynamic programming methods from chapter 14 (specifically equation

(14.3)). Note that your biggest challenge will be computing the one-step transition matrix. Simulate the optimal policy 1,000 times starting with $R_0 = 0$ and report the performance.

b) Now solve the problem using backward ADP using a simple quadratic approximation for the value function approximation:

$$\overline{V}_t^x(R_t^x) = \theta_{t0} + \theta_{t1} R_t^x + \theta_{t2}(R_t^x)^2.$$

where R_t^x is the post-decision resource state which we might represent using

$$R_t^x = \max\{0, R_t - \mathbb{E}\{\hat{D}_{t+1}\}\} + x_t.$$

Having found $\overline{V}_t^x(R_t^x)$, simulate the resulting policy 1,000 times, and compare your results to your optimal policy.

Sequential decision analytics and modeling

These exercises are drawn from the online book *Sequential Decision Analytics and Modeling* available at `http://tinyurl.com/sdaexamplesprint`.

15.5 We are going to perform experiments for an energy storage problem that we can solve exactly using backward approximate dynamic programming. Download the code "EnergyStorage_I" from `http://tinyurl.com/sdagithub`. This code is set up to solve the problem exactly using backward dynamic programming, where we have to enumerate the state space. Here, you will be asked to create a version of the code that uses backward approximate dynamic programming.

Assume that the price process evolves according to

$$p_{t+1} = \min\{100, \max\{0, p_t + \varepsilon_{t+1}\}\}$$

where ε_{t+1} follows a discrete uniform distribution given by

$$\varepsilon_{t+1} = \begin{cases} -2 & \text{with prob. } 1/5 \\ -1 & \text{with prob. } 1/5 \\ 0 & \text{with prob. } 1/5 \\ +1 & \text{with prob. } 1/5 \\ +2 & \text{with prob. } 1/5 \end{cases}$$

Assume that $p_0 = \$50$.

a) Solve for an optimal policy by using the backward dynamic programming strategy in section 14.3 of the text (the algorithm has already been implemented in the Python module).

i) Run the algorithm where prices are discretized in increments of \$1, then \$0.50 and finally \$0.25. Compute the size of the state space for each of the three levels of discretization, and plot the run times against the size of the state space.

ii) Using the optimal value function for the discretization of \$1, simulate the policy for each level of discretization of the prices using 100 forward simulations, and report the estimated objective functions.

b) Modify the code to solve the problem using the approximate dynamic programming with lookup tables given in figure 15.3. Simulate the resulting policy (for each of the three levels of price discretization) and report the results.

c) Modify the code to solve the problem using the approximate dynamic programming using a continuous approximation given in figure 15.5. Simulate the resulting policy (for each of the three levels of price discretization) and report the results. Use the linear model of the post-decision value function

$$\overline{V}^x_t(S^x_t) = \sum_{f\in\mathcal{F}} \theta_f \phi_f(S^x_t)$$

with features

$$\begin{aligned}
\phi_0(S^x_t) &= 1, \\
\phi_1(S^x_t) &= R^x_t, \\
\phi_2(S^x_t) &= (R^x_t)^2, \\
\phi_3(S^x_t) &= p_t, \\
\phi_4(S^x_t) &= p_t^2, \\
\phi_5(S^x_t) &= R^x_t p_t.
\end{aligned}$$

Simulate the policy you obtain with your approximate value function (using 100 simulations) and compare the results to the optimal policy.

d) Repeat (c), but now assume that the price process evolves according to

$$p_{t+1} = .5p_t + .5p_{t-1} + \varepsilon_{t+1}$$

where ε_{t+1} follows the distribution above. Remember that you now have to include p_{t-1} in your state variable. Just use the single price discretization of \$1. Please do the following:

i) First compute the optimal policy following your approach in part (a). You have to modify the code to handle an extra dimension of the state variable. Compare the run times using the price models assumed in part (a) and part (b). How did the more complex state variable affect the solution time for the optimal algorithm and the backward approximate dynamic programming algorithm?

ii) Compare the performance of the optimal solution to the solution obtained using backward approximate dynamic programming.

Diary problem

The diary problem is a single problem you chose (see chapter 1 for guidelines). Answer the following for your diary problem.

15.6 Take your formulation of your diary problem that you developed for the diary problem exercise in chapter 14, and sketch a backward ADP algorithm for the problem. Specify a value function approximation that you think might work.

Bibliography

Aboagye, N. K. & Powell, W. B. (2018), 'Stochastic optimization of Official Development Assistance allocation'.

Cheng, B., Asamov, T. & Powell, W. B. (2018*a*), 'Low-rank value function approximation for co-optimization of battery storage', *IEEE Transactions on Smart Grid* **9**(6), 6590–6598.

Cheng, B., Member, S. & Powell, W. B. (2018*b*), 'Transactions on Smart Grid Co-optimizing Battery Storage for the Frequency Regulation and Energy Arbitrage Using Multi-Scale Dynamic Programming', *IEEE Transactionson the Smart Grid* **9**(3), 1997 – 2005.

Collier, P. & Dollar, D. (2002), 'Aid allocation and poverty reduction', *European Economic Review* **46**(8), 1475–1500.

Munos, R. & Szepesv, C. (2008), 'Finite-Time Bounds for Fitted Value Iteration', *Journal of Machine Learning Research* **1**, 815–857.

Senn, M., Link, N., Pollak, J. & Lee, J. H. (2014), 'Reducing the computational effort of optimal process controllers for continuous state spaces by using incremental learning and post-decision state formulations', *J. of Process Control2* **24**, 133–143.

Szepesvári, C. & Munos, R. (2005), 'Finite time bounds for sampling based fitted value iteration', *Proceedings of the 22nd international conference on Machine learning - ICML '05* pp. 880–887.

Tian, Z., Han, W. & Powell, W. B. (2021), 'Adaptive Learning of Drug Quality and Optimization of Patient Recruitment for Clinical Trials with Dropouts', *Manufacturing & Service Operations Management*.