Neuro-Dynamic Programming

Book · January 1996

DOI: 10.1007/978-0-387-74759-0_440

2 authors:

Dimitri P. Bertsekas
Massachusetts Institute of Technology
374 PUBLICATIONS **58,916** CITATIONS
SEE PROFILE

John N. Tsitsiklis
Massachusetts Institute of Technology
437 PUBLICATIONS **38,498** CITATIONS
SEE PROFILE

Neuro-Dynamic Programming

Dimitri P. Bertsekas and John N. Tsitsiklis (1996)

Massachusetts Institute of Technology

WWW site for book information and orders

http://world.std.com/~athenasc/

Athena Scientific, Belmont, Massachusetts

Learning without thought is labour lost;
thought without learning is perilous.

(Confucian Analects)

1

Introduction

Contents

This book considers systems where decisions are made in stages. The outcome of each decision is not fully predictable but can be anticipated to some extent before the next decision is made. Each decision results in some immediate cost but also affects the context in which future decisions are to be made and therefore affects the cost incurred in future stages. We are interested in decision making policies that minimize the total cost over a number of stages. Such problems are challenging primarily because of the tradeoff between immediate and future costs. Dynamic programming (DP for short) provides a mathematical formalization of this tradeoff.

Generally, in DP formulations we have a discrete-time dynamic system whose state evolves according to given transition probabilities that depend on a decision/control u. In particular, if we are in state i and we choose control u, we move to state j with given probability $p_{ij}(u)$. The control u depends on the state i and the rule by which we select the controls is called a *policy* or *feedback control policy* (see Fig. 1.1). Simultaneously with a transition from i to j under control u, we incur a cost $g(i,u,j)$. In comparing, however, the available controls u, it is not enough to look at the magnitude of the cost $g(i,u,j)$; we must also take into account the desirability of the next state j. We thus need a way to rank or rate states j. This is done by using the optimal cost (over all remaining stages) starting from state j, which is denoted by $J^*(j)$ and is referred to as the optimal *cost-to-go* of state j. These costs-to-go can be shown to satisfy some form of *Bellman's equation*

$$J^*(i) = \min_u E\big[g(i,u,j) + J^*(j) \mid i,u\big], \qquad \text{for all } i,$$

where j is the state subsequent to i, and $E[\cdot \mid i,u]$ denotes expected value with respect to j, given i and u. Generally, at each state i, it is optimal to use a control u that attains the minimum above. Thus, controls are ranked based on the sum of the expected cost of the present period and the optimal expected cost of all subsequent periods.

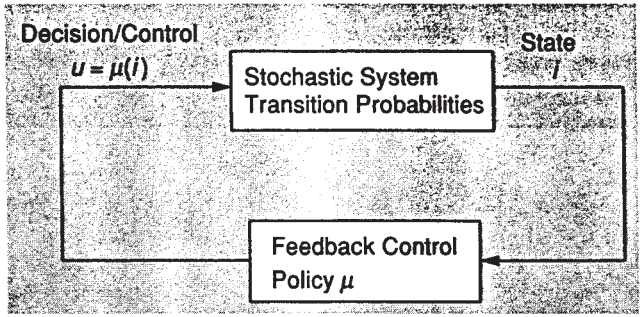

Figure 1.1: Structure of a discrete-time dynamic system under feedback control.

The objective of DP is to calculate numerically the optimal cost-to-go function J^*. This computation can be done off-line, i.e., before the real system starts operating. An optimal policy, that is, an optimal choice of u for each i, is computed either simultaneously with J^*, or in real time by minimizing in the right-hand side of Bellman's equation. It is well known, however, that for many important problems the computational requirements of DP are overwhelming, because the number of states and controls is very large (Bellman's "curse of dimensionality"). In such situations a suboptimal solution is required.

1.1 COST-TO-GO APPROXIMATIONS IN DYNAMIC PROGRAMMING

In this book, we primarily focus on suboptimal methods that center around the evaluation and approximation of the optimal cost-to-go function J^*, possibly through the use of neural networks and/or simulation. In particular, we replace the optimal cost-to-go $J^*(j)$ with a suitable approximation $\tilde{J}(j,r)$, where r is a vector of parameters, and we use at state i the (suboptimal) control $\tilde{\mu}(i)$ that attains the minimum in the (approximate) right-hand side of Bellman's equation, that is,

$$\tilde{\mu}(i) = \arg\min_u E\big[g(i,u,j) + \tilde{J}(j,r) \mid i,u\big].$$

The function $\tilde{J}$ will be called the *scoring function* or *approximate cost-to-go function*, and the value $\tilde{J}(j,r)$ will be called the *score* or *approximate cost-to-go* of state j (see Fig. 1.2). The general form of $\tilde{J}$ is known and is such that once the parameter vector r is fixed, the evaluation of $\tilde{J}(j,r)$ for any state j is fairly simple.

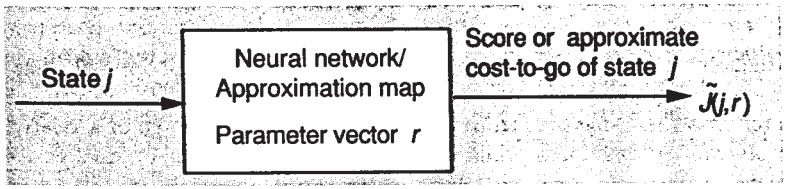

Figure 1.2: Structure of cost-to-go approximation.

We are interested in problems with a large number of states and in scoring functions $\tilde{J}$ that can be described with relatively few numbers (a vector r of small dimension). Scoring functions involving few parameters

will be called *compact representations*, while the tabular description of J^* will be called the *lookup table representation*. In a lookup table representation, the values $J^*(j)$ for all states j are stored in a table. In a typical compact representation, only the vector r and the general structure of the scoring function $\tilde{J}(\cdot, r)$ are stored; the scores $\tilde{J}(j, r)$ are generated only when needed. For example, if $\tilde{J}(j, r)$ is the output of some neural network in response to the input j, then r is the associated vector of weights or parameters of the neural network; or if $\tilde{J}(j, r)$ involves a lower dimensional description of the state j in terms of its "significant features," then r could be a vector of relative weights of the features. Naturally, we would like to choose r algorithmically so that $\tilde{J}(\cdot, r)$ approximates well $J^*(\cdot)$. Thus, determining the scoring function $\tilde{J}(j, r)$ involves two complementary issues: (1) deciding on the general structure of the function $\tilde{J}(j, r)$, and (2) calculating the parameter vector r so as to minimize in some sense the error between the functions $J^*(\cdot)$ and $\tilde{J}(\cdot, r)$.

We note that in some problems the evaluation of the expression

$$E\big[g(i, u, j) + \tilde{J}(j, r) \mid i, u\big],$$

for each u, may be too complicated or too time-consuming for making decisions in real-time, even if the scores $\tilde{J}(j, r)$ are simply calculated. There are a number of ways to deal with this difficulty (see Section 6.1). An important possibility is to approximate the expression minimized in Bellman's equation,

$$Q^*(i, u) = E\big[g(i, u, j) + J^*(j) \mid i, u\big],$$

which is known as the *Q-factor corresponding to* (i, u). In particular, we can replace $Q^*(i, u)$ with a suitable approximation $\tilde{Q}(i, u, r)$, where r is a vector of parameters. We can then use at state i the (suboptimal) control that minimizes the approximate Q-factor corresponding to i:

$$\tilde{\mu}(i) = \arg\min_u \tilde{Q}(i, u, r).$$

Much of what will be said about the approximation of the optimal costs-to-go also applies to the approximation of Q-factors. In fact, we will see later that the Q-factors can be viewed as optimal costs-to-go of a related problem. We thus focus primarily on approximation of the optimal costs-to-go.

Approximations of the optimal costs-to-go have been used in the past in a variety of DP contexts. Chess playing programs represent an interesting example. A key idea in these programs is to use a *position evaluator* to rank different chess positions and to select at each turn a move that results in the position with the best rank. The position evaluator assigns a numerical value to each position according to a heuristic formula that includes weights for the various features of the position (material balance,

piece mobility, king safety, and other factors); see Fig. 1.3. Thus, the position evaluator corresponds to the scoring function $\tilde{J}(j, r)$ above, while the weights of the features correspond to the parameter vector r. Usually, some general structure is selected for the position evaluator (this is largely an art that has evolved over many years, based on experimentation and human knowledge about chess), and the numerical weights are chosen by trial and error or (as in the case of the champion program Deep Thought) by "training" using a large number of sample grandmaster games. It should be mentioned that in addition to the use of sophisticated position evaluators, much of the success of chess programs can be attributed to the use of multimove lookahead, which has become deeper and more effective with the use of increasingly fast hardware.

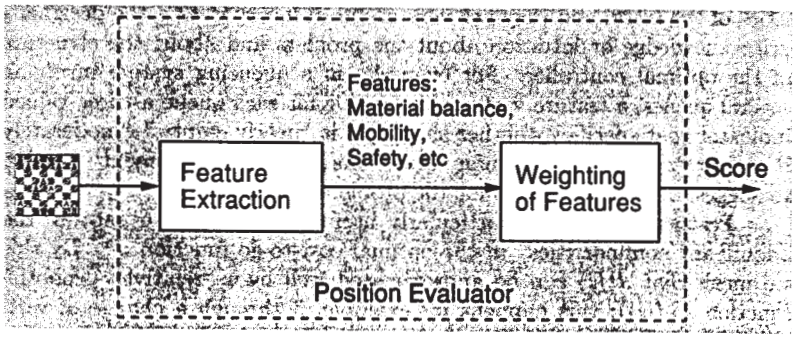

Figure 1.3: Structure of the position evaluator of a chess program.

As the chess program paradigm suggests, intuition about the problem, heuristics, and trial and error are all important ingredients for constructing cost-to-go approximations in DP. However, it is important to supplement heuristics and intuition with more systematic techniques that are broadly applicable and retain as much as possible of the nonheuristic characteristics of DP. This book will focus on several recent efforts to develop a methodological foundation for a rational approach to complex stochastic decision problems, which combines dynamic programming, function approximation, and simulation.

1.2 APPROXIMATION ARCHITECTURES

An important issue in function approximation is the *selection of an architecture*, that is, the choice of a parametric class of functions $\tilde{J}(\cdot, r)$ or

$\tilde{Q}(\cdot,\cdot,r)$ that suits the problem at hand. One possibility is to use a neural network architecture of some type. We should emphasize here that in this book we use the term "neural network" in a very broad sense, essentially as a synonym to "approximating architecture." In particular, we do not restrict ourselves to the classical multilayer perceptron structure with sigmoidal nonlinearities. Any type of universal approximator of nonlinear mappings could be used in our context. The nature of the approximating structure is left open in our discussion, and it could involve, for example, radial basis functions, wavelets, polynomials, splines, aggregation, etc.

Cost-to-go approximation can often be significantly enhanced through the use of *feature extraction*, a process that maps the state i into some vector $f(i)$, called the *feature vector* associated with i. Feature vectors summarize, in a heuristic sense, what are considered to be important characteristics of the state, and they are very useful in incorporating the designer's prior knowledge or intuition about the problem and about the structure of the optimal controller. For example, in a queueing system involving several queues, a feature vector may involve for each queue a three-valued indicator that specifies whether the queue is "nearly empty," "moderately busy," or "nearly full." In many cases, analysis can complement intuition to suggest the right features for the problem at hand.

Feature vectors are particularly useful when they can capture the "dominant nonlinearities" in the optimal cost-to-go function J^*. By this we mean that $J^*(i)$ can be approximated well by a "relatively smooth" function $\tilde{J}\big(f(i)\big)$; this happens for example, if through a change of variables from states to features, J^* becomes a (nearly) linear or low-order polynomial function of the features. When a feature vector can be chosen to have this property, it is appropriate to use approximation architectures where features and (relatively simple) neural networks are used together. In particular, the state is mapped to a feature vector, which is then used as input to a neural network that produces the score of the state (see Fig. 1.4). More generally, it is possible that both the state and the feature vector are provided as inputs to the neural network (see the second diagram in Fig. 1.4).

1.3 SIMULATION AND TRAINING

Some of the most successful applications of neural networks are in the areas of pattern recognition, nonlinear regression, and nonlinear system identification. In these applications the neural network is used as a universal approximator: the input-output mapping of the neural network is matched to an unknown nonlinear mapping F of interest using a least-squares optimization, known as *training the network*. To perform training, one must

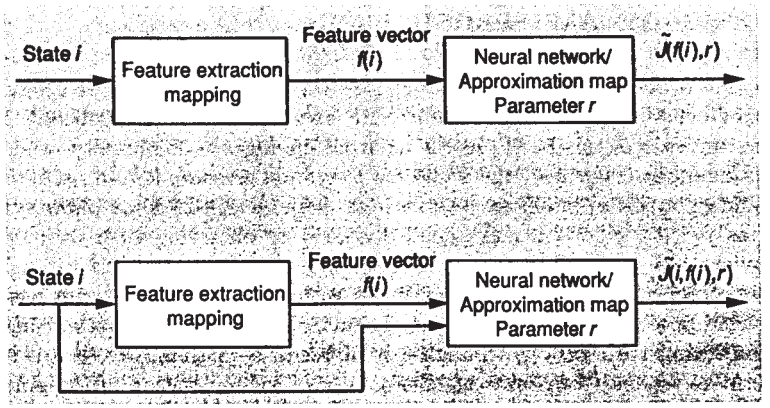

Figure 1.4: Approximation architectures involving feature extraction and neural networks.

have some training data, that is, a set of pairs $\big(i, F(i)\big)$, which is representative of the mapping F that is approximated.

It is important to note that in contrast with these neural network applications, in the DP context there is no readily available training set of input-output pairs $\big(i, J^*(i)\big)$ that could be used to approximate J^* with a least squares fit. The only possibility is to evaluate (exactly or approximately) by simulation the cost-to-go functions of given (suboptimal) policies, and to try to iteratively improve these policies based on the simulation outcomes. This creates analytical and computational difficulties that do not arise in classical neural network training contexts. Indeed the use of simulation to evaluate approximately the optimal cost-to-go function is a key new idea that distinguishes the methodology of this book from earlier approximation methods in DP.

Simulation offers another major advantage: it allows the methods of this book to be used for systems that are hard to model but easy to simulate, i.e., problems where a convenient explicit model is not available, and the system can only be observed, either through a software simulator or as it operates in real time. For such problems, the traditional DP techniques are inapplicable, and estimation of the transition probabilities to construct a detailed mathematical model is often cumbersome or impossible.

There is a third potential advantage of simulation: it can implicitly identify the "most important" or "most representative" states of the system. It appears plausible that these states are the ones most often visited during the simulation, and for this reason the scoring function will tend to approximate better the optimal cost-to-go for these states, and the suboptimal policy obtained will on the average perform better.

1.4 NEURO-DYNAMIC PROGRAMMING

In view of the reliance on both DP and neural network concepts, we use the name *neuro-dynamic programming* (NDP for short) to describe collectively the methods of this book. In the artificial intelligence community, where the methods originated, the name *reinforcement learning* is also used. In common artificial intelligence terms, the methods of this book allow systems to "learn how to make good decisions by observing their own behavior, and use built-in mechanisms for improving their actions through a reinforcement mechanism." In the less anthropomorphic DP terms used in this book, "observing their own behavior" relates to simulation, and "improving their actions through a reinforcement mechanism" relates to iterative schemes for improving the quality of approximation of the optimal costs-to-go, or the Q-factors, or the optimal policy. There has been a gradual realization that reinforcement learning techniques can be fruitfully motivated and interpreted in terms of classical DP concepts such as value and policy iteration.

In this book, we attempt to clarify some aspects of the current NDP methodology, we suggest some new algorithmic approaches, and we identify some open questions. Despite the great interest in NDP, the theory of the subject is only now beginning to take shape, and the corresponding literature is often confusing. Yet, there have been many reports of successes with problems too large and complex to be treated in any other way. A particularly impressive success that greatly motivated subsequent research, was the development of a backgammon playing program by Tesauro [Tes92] (see Section 8.6). Here a neural network was trained to approximate the optimal cost-to-go function of the game of backgammon by using simulation, that is, by letting the program play against itself. After training for several months, the program nearly defeated the human world champion. Unlike chess programs, this program did not use lookahead of many stages, so its success can be attributed primarily to the use of a properly trained approximation of the optimal cost-to-go function.

Our own experience has been that NDP methods can be impressively effective in problems where traditional DP methods would be hardly applicable and other heuristic methods would have limited potential. In this book, we outline some engineering applications, and we use a few computational studies for illustrating the methodology and some of the art that is often essential for success.

We note, however, that the practical application of NDP is computationally very intensive, and often requires a considerable amount of trial and error. Furthermore, success is often obtained using methods whose properties are not fully understood. Fortunately, all of the computation and experimentation with different approaches can be done off-line. Once the approximation is obtained off-line, it can be used to generate decisions fast enough for use in real time. In this context, we mention that in the artificial intelligence literature, reinforcement learning is often viewed as an "on-line" method, whereby the cost-to-go approximation is improved as the system operates in real time. This is reminiscent of the methods of traditional adaptive control. We will not discuss this viewpoint in this book, as we prefer to focus on applications involving a large and complex system. A lot of training data are required for such systems. These data often cannot be obtained in sufficient volume as the system is operating; even if they can, the corresponding processing requirements are often too large for effective use in real time.

We finally mention an alternative approach to NDP, known as *approximation in policy space*, which, however, we will not consider in this book. In this approach, in place of an overall optimal policy, we look for an optimal policy within some restricted class that is parametrized by some vector s of relatively low dimension. In particular, we consider policies of a given form $\tilde{\mu}(i,s)$. We then minimize over s the expected cost $E_i\big[J^{\tilde{\mu}(\cdot,s)}(i)\big]$, where the expectation is with respect to some suitable probability distribution of the initial state i. This approach applies to complex problems where there is no explicit model for the system and the cost, as long as the cost corresponding to a given policy can be calculated by simulation. Furthermore, insight and analysis can sometimes be used to select simple and effective parametrizations of the policies. On the other hand, there are many problems where such parametrizations are not easily obtained. Furthermore, the minimization of $E_i\big[J^{\tilde{\mu}(\cdot,s)}(i)\big]$ can be very difficult because the gradient of the cost with respect to s may not be easily calculated; while methods that require cost values (and not gradients) may be used, they tend to require many cost function evaluations and to be slow in practice.

The general organizational plan of the book is to first develop some essential background material on DP, and on deterministic and stochastic iterative optimization algorithms (Chs. 2-4), and then to develop the main algorithmic methods of NDP in Chs. 5 and 6. Various extensions of the methodology are discussed in Ch. 7. Finally, we present case studies in Ch. 8. Many of the ideas of the book extend naturally to continuous-state systems, although the NDP theory is far from complete for such systems. To keep the exposition simple, we have restricted ourselves to the case where the number of states is finite and the number of available controls at each state is also finite. This is consistent with the computational orientation of the book.

1.5 NOTES AND SOURCES

1.1. The origins of our subject can be traced to the early works on DP by Bellman, who used the term "approximation in value space," and

to the works by Shannon [Sha50] on computer chess and by Samuel [Sam59], [Sam67] on computer checkers.

1.4. The works by Barto, Sutton, and Anderson [BSA83] on adaptive critic systems, by Sutton [Sut88] on temporal difference methods, and by Watkins [Wat89] on Q-learning initiated the modern developments which brought together the ideas of function approximation, simulation, and DP. The work of Tesauro [Tes92], [Tes94], [Tes95] on backgammon was the first to demonstrate impressive success on a very complex and challenging problem. Much research followed these seminal works. The extensive survey by Barto, Bradtke, and Singh [BBS95], and the overviews by Werbös [Wer92a], [Wer92b], and other papers in the edited volume by White and Sofge [WhS92] point out the connections between the artificial intelligence/reinforcement learning viewpoint and the control theory/DP viewpoint, and give many references. The DP textbook by Bertsekas [Ber95a] describes a broad variety of suboptimal control methods, including some of the NDP approaches that are treated in much greater depth in the present book.

Philosophy is written in this grand book, the universe,
which stands continually open to our gaze.
But the book cannot be understood
unless one first learns to comprehend the language
and read the letters in which it is composed.
It is written in the language of mathematics.

(Galileo)

2

Dynamic Programming

Contents