Class Notes for
ASU Course CSE 691; Spring 2023
Topics in Reinforcement Learning

CHAPTER 3
Version of April 24, 2023

Based on the Books

Dynamic Programming and Optimal Control,
Reinforcement Learning and Optimal Control, 2019,
Rollout, Policy Iteration, and
Distributed Reinforcement Learning, 2020,
Abstract Dynamic Programming, 3rd Ed., 2022,

and

Lessons from AlphaZero for Optimal,
Model Predictive, and Adaptive Control, 2022,

by

Dimitri P. Bertsekas
Arizona State University
dbertsek@asu.edu, dimitrib@mit.edu
# Learning Values and Policies

## Contents

3.1. Parametric Approximation Architectures . . . . . . . . p. 3
   3.1.1. Cost Function Approximation . . . . . . . . . . p. 4
   3.1.2. Feature-Based Architectures . . . . . . . . . . p. 5
   3.1.3. Training of Linear and Nonlinear Architectures . . p. 16
3.2. Neural Networks . . . . . . . . . . . . . . . . . . p. 23
   3.2.1. Training of Neural Networks . . . . . . . . . . p. 28
   3.2.2. Multilayer and Deep Neural Networks . . . . . p. 29
3.3. Training of Cost Functions in Approximate DP . . . . . p. 31
   3.3.1. Fitted Value Iteration . . . . . . . . . . . . . p. 31
   3.3.2. Q-Factor Parametric Approximation - Model-Free . .
       Implementation . . . . . . . . . . . . . . . . . . . . . . p. 33
   3.3.3. Parametric Approximation in Infinite Horizon . . . . .
       Problems - Approximate Policy Iteration . . . . . p. 36
   3.3.4. Optimistic Policy Iteration with Parametric Q-Factor . .
       Approximation - SARSA and DQN . . . . . . . . . . . . . . p. 39
   3.3.5. Approximate Policy Iteration for Infinite Horizon . . . . .
       POMDP . . . . . . . . . . . . . . . . . . . . . . . . . . . p. 42
   3.3.6. Advantage Updating - Approximating Q-Factor . . . . .
       Differences . . . . . . . . . . . . . . . . . . . . . . . . . . p. 46
   3.3.7. Differential Training of Cost Differences for Rollout . . . . . p. 49
3.4. Training of Policies in Approximate DP . . . . . . . . . . p. 51
   3.4.1. The Use of Classifiers for Approximation in Policy . . . . . . . .
       Space . . . . . . . . . . . . . . . . . . . . . . . . . . . p. 51
   3.4.2. Policy Iteration with Value and Policy Networks - . .
       Multiprocessor Parallelization . . . . . . . . . . . . . . p. 55
   3.4.3. Why Use On-Line Play and not Just Train a Policy . . . .
       Network to Emulate the Lookahead Minimization? . . . . . . . . p. 57
3.5. Aggregation ........................................ p. 59
3.5.1. Aggregation with Representative States .......... p. 59
3.5.2. Continuous Control Space Discretization .......... p. 65
3.5.3. Continuous State Space - POMDP Discretization . p. 67
3.5.4. General Aggregation ................................ p. 68
3.5.5. Hard Aggregation and Error Bounds ............... p. 72
3.5.6. Aggregation Using Features ...................... p. 74
3.5.7. Biased Aggregation ................................ p. 77
3.5.8. Asynchronous Distributed Multiagent Aggregation . p. 80
3.6. Notes, Sources, and Exercises ....................... p. 82
In this chapter, we will discuss the methods and objectives of off-line training through the use of parametric approximation architectures such as neural networks. We begin with a general discussion of parametric architectures and their training in Section 3.1. We then consider the training of neural networks in Section 3.2, and their use in the context of finite horizon approximate DP in Section 3.3. In Section 3.4, we discuss the training of policies. Finally, in Section 3.5, we discuss aggregation methods.

3.1 PARAMETRIC APPROXIMATION ARCHITECTURES

As we have noted earlier, for the success of approximation in value space, it is important to select a class of lookahead function approximations $\tilde{J}_k$ that is suitable for the problem at hand. In the preceding two chapters we discussed several methods for choosing $\tilde{J}_k$, based mostly on some form of rollout. We will now discuss how $\tilde{J}_k$ can be obtained by off-line training from a parametric class of functions, possibly involving a neural network, with the parameters “optimized” with the use of some algorithm.

A general structure for parametric cost function approximation is illustrated in Fig. 3.1.1. We have a target function $J(x)$ that we want to approximate with a member from a parametric class $\tilde{J}(x, r)$ that depend on a parameter vector $r$. To this end, we collect training data $(x^s, J(x^s)), \ s = 1, \ldots, q$, which we use to determine a parameter $\hat{r}$ that leads to a good “fit” between the data $J(x^s)$ and the predictions $\tilde{J}(x^s, \hat{r})$ of the parametrized function. This is usually done through some form of optimization that aims to minimize the size of the errors $J(x^s) - \tilde{J}(x^s, \hat{r}), \ s = 1, \ldots, q$.

The methodological ideas for parametric cost function approximation can also be used for approximation of a target policy $\mu$ with a policy from a parametric class $\tilde{\mu}(x, r)$. The training data may be obtained, for example, from rollout control calculations, thus enabling the construction of both value and policy networks that can be combined for use in a perpetual rollout...
scheme. However, there is an important difference: the approximate cost values \( \tilde{J}(x, r) \) are real numbers, whereas the approximate policy values \( \tilde{\mu}(x, r) \) are elements of a control space \( U \). Thus if \( U \) consists of \( m \) dimensional vectors, \( \tilde{\mu}(x, r) \) consists of \( m \) numerical components. In this case the parametric approximation problems for cost functions and for policies are fairly similar, and both involve continuous space approximations.

However, the case where the control space is finite \( U = \{u_1, \ldots, u_m\} \) is markedly different. In this case, for any \( x \), \( \tilde{\mu}(x, r) \) consists of one of the \( m \) possible controls \( u_1, \ldots, u_m \). This ushers a connection with traditional classification schemes, whereby objects \( x \) are classified as belonging to one of the categories \( u_1, \ldots, u_m \), so that \( \mu(x) \) defines the category of \( x \), and can be viewed as a classifier. Some of the most prominent classification schemes actually produce randomized outcomes, i.e., \( x \) is associated with a probability distribution

\[
\{\tilde{\mu}(u_1, r), \ldots, \tilde{\mu}(u_m, r)\},
\]

which is a randomized policy in our policy approximation context; see Fig. 3.1.2. This is done usually for reasons of algorithmic convenience, since many optimization methods, including least squares regression, require that the optimization variables are continuous. In this case, the randomized policy (3.1) can be converted to a nonrandomized policy using a maximization operation: associate \( x \) with the control of maximum probability (cf. Fig. 3.1.2),

\[
\tilde{\mu}(x, r) \in \arg \max_{i=1, \ldots, m} \tilde{\mu}_i(x, r).
\]

The use of classification methods for approximation in policy space will be discussed more fully in Section 3.4.

### 3.1.1 Cost Function Approximation

For the remainder of this section, as well as Sections 3.2 and 3.3, we will focus on approximation in value space schemes, where the approximate cost functions are selected from a parametric class of functions \( \tilde{J}_k(x_k, r_k) \) that
for each \( k \), depend on the current state \( x_k \) and a vector \( r_k = (r_{1,k}, \ldots, r_{m_k,k}) \) of \( m_k \) “tunable” scalar parameters. By adjusting the parameters, one can change the “shape” of \( J_k \) so that it is a reasonably good approximation to some target function, usually the true optimal cost-to-go function \( J_k^* \), or the cost-to-go function \( J_{k,\pi} \) of some policy \( \pi \). The class of functions \( \hat{J}_k(x_k, r_k) \) is called an approximation architecture, and the process of choosing the parameter vectors \( r_k \) is commonly called training or tuning the architecture.

We will focus initially on approximation of cost functions, hence the use of the \( \hat{J}_k \) notation. In Section 3.4 we will consider the other major use of parametric approximation architectures, of the form \( \hat{\mu}_k(x_k, r_k) \), where the target function is a control function \( \mu_k \) that is part of some policy.

The simplest training approach for parametric architectures is to do some form of semi-exhaustive or semi-random search in the space of parameter vectors and adopt the parameters that result in best performance of the associated one-step lookahead controller (according to some criterion). There are methods of this type that have been used primarily in cases where the number of parameters is relatively small.

Random search and Bayesian optimization methods have also been used to tune hyperparameters of an approximation architecture; for example, the number of layers in a neural network, or the number of clusters in the context of partitioning discrete spaces into clusters, etc. We refer to the research literature for further discussion.

Other systematic approaches are based on numerical optimization, such as a least squares fit that aims to match the cost approximation produced by the architecture to a “training set,” i.e., a large number of pairs of state and cost values that are obtained through some form of sampling process. Throughout Sections 3.1-3.3 we will focus primarily on this approach.

### 3.1.2 Feature-Based Architectures

There is a large variety of approximation architectures, based for example on polynomials, wavelets, radial basis functions, discretization/interpolation schemes, neural networks, and others. A particularly interesting type of cost approximation involves feature extraction, a process that maps the state \( x_k \) into some vector \( \phi_k(x_k) \), called the feature vector associated with \( x_k \) at time \( k \). The vector \( \phi_k(x_k) \) consists of scalar components

\[
\phi_{1,k}(x_k), \ldots, \phi_{m_k,k}(x_k),
\]

called features. A feature-based cost approximation has the form

\[
\hat{J}_k(x_k, r_k) = \hat{J}_k(\phi_k(x_k), r_k),
\]

where \( r_k \) is a parameter vector and \( \hat{J}_k \) is some function. Thus, the cost approximation depends on the state \( x_k \) through its feature vector \( \phi_k(x_k) \).
Note that we are allowing for different features \( \phi_k(x_k) \) and different parameter vectors \( r_k \) for each stage \( k \). This is necessary for nonstationary problems (e.g., if the state space changes over time), and also to capture the effect of proximity to the end of the horizon. On the other hand, for stationary problems with a long or infinite horizon, where the state space does not change with \( k \), it is common to use the same features and parameters for all stages. The subsequent discussion can easily be adapted to infinite horizon methods, as we will discuss later.

Features are often handcrafted, based on whatever human intelligence, insight, or experience is available, and are meant to capture the most important characteristics of the current state. There are also systematic ways to construct features, including the use of data and neural networks, which we will discuss shortly. In this section, we provide a brief and selective presentation of architectures.

One idea behind using features is that the optimal cost-to-go functions \( J_k^* \) may be complicated nonlinear mappings, so it is sensible to try to break their complexity into smaller, less complex pieces. In particular, if the features encode much of the nonlinearity of \( J_k^* \), we may be able to use a relatively simple architecture \( \tilde{J}_k \) to approximate \( J_k^* \). For example, with a well-chosen feature vector \( \phi_k(x_k) \), a good approximation to the cost-to-go is often provided by linearly weighting the features, i.e.,

\[
\tilde{J}_k(x_k, r_k) = \tilde{J}_k(\phi_k(x_k), r_k) = \sum_{\ell=1}^{m_k} r_{\ell,k} \phi_{\ell,k}(x_k) = r'_k \phi_k(x_k),
\]

(3.3)

where \( r_{\ell,k} \) and \( \phi_{\ell,k}(x_k) \) are the \( \ell \)th components of \( r_k \) and \( \phi_k(x_k) \), respectively, and \( r'_k \phi_k(x_k) \) denotes the inner product of \( r_k \) and \( \phi_k(x_k) \), viewed as column vectors of \( \mathbb{R}^{m_k} \) (a prime denotes transposition, so \( r'_k \) is a row vector); see Fig. 3.1.3.

This is called a linear feature-based architecture, and the scalar parameters \( r_{\ell,k} \) are also called weights. Among other advantages, these architectures admit simpler training algorithms than their nonlinear counterparts; see the NDP book [BeT96]. Mathematically, the approximating function \( \tilde{J}_k(x_k, r_k) \) can be viewed as a member of the subspace spanned by the features \( \phi_{\ell,k}(x_k) \), \( \ell = 1, \ldots, m_k \), which for this reason are also referred to as basis functions. We provide a few examples, where for simplicity we drop the index \( k \).
Sec. 3.1 Parametric Approximation Architectures

\[ \tilde{J}(x, r) = \sum_{\ell=1}^{m} r_\ell \phi_\ell(x) \]

Figure 3.1.4 Illustration of a piecewise constant architecture. The state space is partitioned into subsets \( S_1, \ldots, S_m \), with each subset \( S_\ell \) defining the feature

\[ \phi_\ell(x) = \begin{cases} 1 & \text{if } x \in S_\ell, \\ 0 & \text{if } x \notin S_\ell, \end{cases} \quad \ell = 1, \ldots, m, \]

with its own weight \( r_\ell \).

Example 3.1.1 (Piecewise Constant Approximation)

Suppose that the state space is partitioned into subsets \( S_1, \ldots, S_m \), so that every state belongs to one and only one subset. Let the \( \ell \)th feature be defined by membership to the set \( S_\ell \), i.e.,

\[ \phi_\ell(x) = \begin{cases} 1 & \text{if } x \in S_\ell, \\ 0 & \text{if } x \notin S_\ell, \end{cases} \quad \ell = 1, \ldots, m. \]

Consider the architecture

\[ \tilde{J}(x, r) = \sum_{\ell=1}^{m} r_\ell \phi_\ell(x), \]

where \( r \) is the vector consists of the \( m \) scalar parameters \( r_1, \ldots, r_m \). It can be seen that \( \tilde{J}(x, r) \) is the piecewise constant function that has value \( r_\ell \) for all states within the set \( S_\ell \); see Fig. 3.1.4.

The piecewise constant approximation is an example of a linear feature-based architecture that involves exclusively local features. These are features that take a nonzero value only for a relatively small subset of states. Thus a change of a single weight causes a change of the value of \( \tilde{J}(x, r) \) for relatively few states \( x \). At the opposite end we have linear
feature-based architectures that involve *global features*. These are features that take nonzero values for a large number of states. The following is a common example.

**Example 3.1.2 (Polynomial Approximation)**

An important case of linear architecture is one that uses polynomial basis functions. Suppose that the state consists of \( n \) components \( x_1, \ldots, x_n \), each taking values within some range of integers. For example, in a queueing system, \( x_i \) may represent the number of customers in the \( i \)th queue, where \( i = 1, \ldots, n \). Suppose that we want to use an approximating function that is quadratic in the components \( x_i \). Then we can define a total of \( 1 + n + n^2 \) basis functions that depend on the state \( x = (x_1, \ldots, x^n) \) via

\[
\phi_0(x) = 1, \quad \phi_i(x) = x^i, \quad \phi_{ij}(x) = x^i x^j, \quad i, j = 1, \ldots, n.
\]

A linear approximation architecture that uses these functions is given by

\[
\tilde{J}(x, r) = r_0 + \sum_{i=1}^{n} r_i x^i + \sum_{i=1}^{n} \sum_{j=1}^{n} r_{ij} x^i x^j,
\]

where the parameter vector \( r \) has components \( r_0, r_i, \) and \( r_{ij} \), with \( i, j = 1, \ldots, n \). Indeed, any kind of approximating function that is polynomial in the components \( x_1, \ldots, x^n \) can be constructed similarly.

A more general polynomial approximation may be based on some other known features of the state. For example, we may start with a feature vector

\[
\phi(x) = (\phi_1(x), \ldots, \phi_m(x))^t,
\]

and transform it with a quadratic polynomial mapping. In this way we obtain approximating functions of the form

\[
\tilde{J}(x, r) = r_0 + \sum_{i=1}^{m} r_i \phi_i(x) + \sum_{i=1}^{m} \sum_{j=1}^{m} r_{ij} \phi_i(x) \phi_j(x),
\]

where the parameter \( r \) has components \( r_0, r_i, \) and \( r_{ij} \), with \( i, j = 1, \ldots, m \). This can also be viewed as a linear architecture that uses the basis functions

\[
w_0(x) = 1, \quad w_i(x) = \phi_i(x), \quad w_{ij}(x) = \phi_i(x) \phi_j(x), \quad i, j = 1, \ldots, m.
\]

The preceding example architectures are generic in the sense that they can be applied to many different types of problems. Other architectures rely on problem-specific insight to construct features, which are then combined into a relatively simple architecture. We present two examples involving games.
Sec. 3.1  Parametric Approximation Architectures

Figure 3.1.5 The board of the tetris game. The squares fill up as blocks of different shapes fall from the top of the grid and are added to the top of the wall. The shapes are generated according to some stochastic process. As a given block falls, the player can move horizontally and rotate the block in all possible ways, subject to the constraints imposed by the sides of the grid and the top of the wall. When a row of full squares is created, this row is removed, the bricks lying above this row move one row downward, and the player scores a point. The player’s objective is to maximize the score attained (total number of rows removed) within \(N\) steps or up to termination of the game, whichever occurs first.

Example 3.1.3 (Tetris)

Let us consider the game of tetris, which we formulated in Example 1.6.2 as a stochastic shortest path problem with the termination state being the end of the game (see Fig. 3.1.5). The state is the pair of the board position \(x\) and the shape of the current falling block \(y\). We viewed as control, the horizontal positioning and rotation applied to the falling block. The optimal cost-to-go function is a vector of huge dimension (there are \(2^{200}\) board positions in a ‘standard’ tetris board of width 10 and height 20). However, it has been successfully approximated in practice by low-dimensional linear architectures.

In particular, the following features have been proposed in the paper by Bertsekas and Ioffe [BeI96]: the heights of the columns, the height differentials of adjacent columns, the wall height (the maximum column height), the number of holes of the board, and the constant 1 (the unit is often included as a feature in cost approximation architectures, as it allows for a constant shift in the approximating function). These features are readily recognized by tetris players as capturing important aspects of the board position.† There

† The use of feature-based approximate DP methods for the game of tetris was first suggested in the paper by Tsitsiklis and Van Roy [TsV96], which introduced just two features (in addition to the constant 1): the wall height and the number of holes of the board. Most studies have used the set of features of [BeI96] described here, but other sets of features have also been used; see [ThS09] and the discussion in [GGS13].
are a total of 22 features for a “standard” board with 10 columns. Of course the $2^{200} \times 22$ matrix of feature values cannot be stored in a computer, but for any board position, the corresponding row of features can be easily generated, and this is sufficient for implementation of the associated approximate DP algorithms. For recent works involving approximate DP methods and the preceding 22 features, see [Sch13], [GGS13], and [SGG15], which reference several other related papers.

In the works mentioned above the shapes of the falling blocks are stochastically independent. In a more challenging version of the problem, which has not been considered in the literature thus far, successive shapes are correlated. Then the state of the problem would become more complex, since past shapes would be useful in predicting future shapes. As a result, we may need to introduce state estimation and additional features in order to properly deal with the effects of correlations.

**Example 3.1.4 (Computer Chess)**

Computer chess programs that involve feature-based architectures have been available for many years, and are still used widely (they have been upstaged in the mid-2010s by alternative types of chess programs, which use neural network techniques that will be discussed later). These programs are based on approximate DP for minimax problems, a feature-based parametric architecture, and multistep lookahead.

The fundamental principles on which all computer chess programs (as well as most two-person game programs) are based were laid out by Shannon [Sha50], before Bellman started his work on DP. Shannon proposed multistep lookahead and evaluation of the end positions by means of a “scoring function” (in our terminology this plays the role of a cost function approximation). This function may involve, for example, the calculation of a numerical value for each of a set of major features of a position that chess players easily recognize (such as material balance, mobility, pawn structure, and other positional factors), together with a method to combine these numerical values into a single score. Shannon then went on to describe various strategies of exhaustive and selective search over a multistep lookahead tree of moves.

We may view the scoring function as a feature-based architecture for evaluating a chess position/state (cf. Fig. 3.1.6). In most computer chess programs, the features are weighted linearly, i.e., the architecture $J(x, r)$ that is used for multistep lookahead is linear [cf. Eq. (3.3)]. In many cases, the weights have been determined manually, by trial and error based on experi-
ence. However, in some programs, the weights have been determined with supervised learning techniques that use examples of grandmaster play, i.e., by adjustment to bring the play of the program as close as possible to the play of chess grandmasters. This is a technique that applies more broadly in artificial intelligence; see Tesauro [Tes89b], [Tes01].

In a recent computer chess breakthrough, the entire idea of extracting features of a position through human expertise was abandoned in favor of feature discovery through self-play and the use of neural networks. The first program of this type to attain supremacy over humans, as well as over the best computer programs that use human expertise-based features, was AlphaZero (Silver et al. [SHS17]). This program, described in Section 1.1, is based on DP principles of approximate policy iteration and multistep lookahead based on Monte Carlo tree search.

Our next example relates to a methodology for feature construction, where the number of features may increase as more data is collected. For a simple example, consider the piecewise constant approximation of Example 3.1.1, where more pieces are progressively added based on new data, possibly using some form of exploration-exploitation tradeoff.

**Example 3.1.5 (Feature Extraction from Data)**

We have viewed so far feature vectors \( \phi(x) \) as functions of \( x \), obtained through some unspecified process that is based on prior knowledge about the cost function being approximated. On the other hand, features may also be extracted from data. For example suppose that with some preliminary calculation using data, we have identified some suitable states \( x(\ell) \), \( \ell = 1, \ldots, m \), that can serve as “anchors” for the construction of Gaussian basis functions of the form

\[
\phi_\ell(x) = e^{-\frac{\|x - x(\ell)\|^2}{2\sigma^2}}, \quad \ell = 1, \ldots, m, \tag{3.4}
\]

where \( \sigma \) is a scalar “variance” parameter, and \( \| \cdot \| \) denotes the standard Euclidean norm. This type of function is known as a *radial basis function*. It is concentrated around the state \( x(\ell) \), and it is weighed with a scalar weight \( r_\ell \) to form a parametric linear feature-based architecture, which can be trained using additional data. Several other types of data-dependent basis functions, such as support vector machines, are used in machine learning, where they are often referred to as *kernels*.

While it is possible to use a preliminary calculation to obtain the anchors \( x(\ell) \) in Eq. (3.4), and then use additional data for training, one may also consider enrichment of the set of basis functions simultaneously with training. In this case the number of the basis functions increases as the training data is collected. A motivation here is that the quality of the approximation may increase with additional basis functions. This idea underlies a field of machine learning, known as *kernel methods* or sometimes *nonparametric methods*.

A further discussion is outside our scope. We refer to the literature; see e.g., books such as Cristianini and Shawe-Taylor [ChS00], [ShC04], Scholkopf and Smola [ScS02], Bishop [Bis06], Kung [Kun14], surveys such as Hofmann,
Learning Values and Policies  Chap. 3

Scholkopf, and Smola [HSS08], Pillonetto et al. [PDC14], RL-related discussions such as Dietterich and Wang [DiW02], Ormoneit and Sen [OrS02], Engel, Mannor, and Meir [EMM05], Jung and Polani [JuP07], Reisinger, Stone, and Miikkulainen [RSM08], Busoniu et al. [BBD10a], Bethke [Bet10], and recent developments such as Tu et al. [TRV16], Rudi, Carratino, and Rosasco [RCR17], Belkin, Ma, and Mandal [BMM18]. In what follows, for the sake of simplicity, we will focus on parametric architectures with a fixed and given feature vector, since the choice of approximation architecture is somewhat peripheral to our main focus.

The next example considers a feature extraction strategy that is particularly relevant to problems of partial state information.

Example 3.1.6 (Feature Extraction from Sufficient Statistics)

The concept of a sufficient statistic, which originated in inference methodologies, plays an important role in DP. As discussed in Section 1.6, it refers to quantities that summarize all the essential content of the state $x_k$ for optimal control selection at time $k$.

In particular, consider a partial information context where at time $k$ we have accumulated the information vector (also called the past history)

$$I_k = (z_0, \ldots, z_k, u_0, \ldots, u_{k-1})$$

which consists of the past controls $u_0, \ldots, u_{k-1}$ and the state-related measurements $z_0, \ldots, z_k$ obtained at the times $0, \ldots, k$. The control $u_k$ is allowed to depend only on $I_k$, and the optimal policy is a sequence of the form $\{\mu_0(I_0), \ldots, \mu_{N-1}(I_{N-1})\}$. We say that a function $S_k(I_k)$ is a sufficient statistic at time $k$ if the control function $\mu_k^*$ depends on $I_k$ only through $S_k(I_k)$, i.e., for some function $\hat{\mu}_k$, we have

$$\mu_k^*(I_k) = \hat{\mu}_k(S_k(I_k)),$$

where $\mu_k^*$ is optimal.

There are several examples of sufficient statistics, and they are typically problem-dependent. A trivial possibility is to view $I_k$ itself as a sufficient statistic, and a more sophisticated possibility is to view the belief state $b_k$ as a sufficient statistic (this is the conditional probability distribution of $x_k$ given $I_k$; cf. Section 1.6.4). For a proof that $b_k$ is indeed a sufficient statistic and for a more detailed discussion of other possible sufficient statistics, see [Ber17a], Chapter 4. For a mathematically more advanced discussion, see [BeS78], Chapter 10.

Since a sufficient statistic contains all the relevant information for optimal control purposes, an idea that suggests itself is to introduce features of a given sufficient statistic and to train a corresponding approximation architecture accordingly. As examples of potentially good features, one may consider some special characteristic of $I_k$ (such as whether some alarm-like "special" event has been observed), or a partial history (such as the last $m$ measurements and controls in $I_k$, or more sophisticated versions based on the concept
of a finite-state controller proposed by White [Whi91], and White and Scherer [WhS94], and further discussed by Hansen [Han98], Kaelbling, Littman, and Cassandra [KLC98], Meuleau et al. [MPK99], Poupart and Boutilier [PoB04], Yu and Bertsekas [YuB08], Saldi, Yuksel, and Linder [SYL17]. In the case where the belief state $b_k$ is used as a sufficient statistic, examples of good features may be a point estimate based on $b_k$, the variance of this estimate, and other quantities that can be simply extracted from $b_k$.

The paper by Bhattacharya et al. [BBW20] considers another type of feature vector that is related to the belief state. This is a sufficient statistic, denoted by $y_k$, which subsumes the belief state $b_k$, in the sense that $b_k$ can be computed exactly knowing $y_k$. One possibility is for $y_k$ to be the union of $b_k$ and some identifiable characteristics of the belief state, or some compact representation of the measurement history up to the current time (such as a number of most recent measurements, or the state of a finite-state controller). Even though the information content of $y_k$ is no different than the information content of $b_k$ for the purposes of exact optimization, a sufficient statistic that is specially designed for the problem at hand may lead to improved performance in the presence of cost and policy approximations.

We finally note a related idea, which is to supplement a sufficient statistic with features of other sufficient statistics, and thus obtain an enlarged/richer sufficient statistic. In problem-specific contexts, and in the presence of approximations, this may yield improved results.

**Example 3.1.7 (Feature-Based Dimensionality Reduction by Aggregation)**

The use of a feature vector $\phi(x)$ to represent the state $x$ in an approximation architecture of the form $\hat{J}(\phi(x), r)$ implicitly involves state aggregation, i.e., the grouping of states into subsets. We will discuss aggregation in some detail in Section 3.5. Here we will give a summary of a particular type of aggregation architecture.

In particular, let us assume that the feature vector can take only a finite number of values, and define for each possible value $v$, the subset of states $S_v$ whose feature vector is equal to $v$:

$$S_v = \{ i \mid \phi(x) = v \}.$$  

We refer to the sets $S_v$ as the aggregate states induced by the feature vector. These sets form a partition of the state space. An approximate cost-to-go function of the form $\hat{J}(\phi(x), r)$ is piecewise constant with respect to this partition; that is, it assigns the same cost-to-go value $\hat{J}(v, r)$ to all states in the set $S_v$.

An often useful approach to deal with problem complexity in DP is to introduce an “aggregate” DP problem, whose states are some suitably defined feature vectors $\phi(x)$ of the original problem. The precise form of the aggregate problem may depend on intuition and/or heuristic reasoning, based on our understanding of the original problem. Suppose now that the aggregate problem is simple enough to be solved exactly by DP, and let $\hat{J}(v)$
be its optimal cost-to-go when the initial value of the feature vector is \( v \). Then \( \hat{J}(\phi(x)) \) provides an approximation architecture for the original problem, i.e., the architecture that assigns to state \( x \) the (exactly) optimal cost-to-go \( J(\phi(x)) \) of the feature vector \( \phi(x) \) in the aggregate problem. There is considerable freedom on how one formulates and solves aggregate problems. We refer to the DP textbooks [Ber12], [Ber17a], and the RL textbook [Ber19a], Chapter 6, for a detailed treatment; see also the discussion of Section 3.5.

The next example relates to an architecture that is particularly useful when parallel computation is available.

**Example 3.1.8 (Feature-Based State Space Partitioning)**

A simple method to construct complex and sophisticated approximation architectures, is to partition the state space into several subsets and construct a separate approximation in each subset. For example, by using a separate linear or quadratic polynomial approximation in each subset of the partition, we can construct piecewise linear or piecewise quadratic approximations over the entire state space. Similarly, we may use a separate neural network architecture on each set of the partition. An important issue here is the choice of the method for partitioning the state space. Regular partitions (e.g., grid partitions) may be used, but they often lead to a large number of subsets and very time-consuming computations.

Generally speaking, each subset of the partition should contain “similar” states so that the variation of the optimal cost-to-go over the states of the subset is relatively smooth and can be approximated with smooth functions.

An interesting possibility is to use features as the basis for partition. In particular, one may use a more or less regular partition of the space of features, which induces a possibly irregular partition of the original state space. In this way, each subset of the irregular partition contains states with “similar features”; see Fig. 3.1.7.

As an illustration consider the game of chess. The state here consists of the board position, but the nature of the position progresses over time through opening, middlegame, and endgame phases. Moreover each of these
phases may be affected differently by special features of the position. For example there are several different types of endgames (rook endgames, king-and-pawn endgames, minor-piece endgames, etc), which are characterized by identifiable features and call for different playing strategies. It would thus make sense to partition the set of chess positions according to their features, and use a separate strategy on each set of the partition. Indeed this is done to some extent in a number of chess programs.

A potential difficulty with partitioned architectures is that there is discontinuity of the approximation along the boundaries of the partition. For this reason, a variant, called soft partitioning, is sometimes employed, whereby the subsets of the partition are allowed to overlap and the discontinuity is smoothed out over their intersection. In particular, once a function approximation is obtained in each subset, the approximate cost-to-go in the overlapping regions is taken to be a smoothly varying linear combination of the function approximations of the corresponding subsets.

Partitioning and local approximations can also be used to enhance the quality of approximation in parts of the space where the target function has some special character. For example, suppose that the state space $S$ is partitioned in subsets $S_1, \ldots, S_M$ and consider approximations of the form

$$\tilde{J}(x, r) = \hat{J}(x, \hat{r}) + \sum_{m=1}^{M} \sum_{k=1}^{K_m} r_m(k) \phi_{k,m}(x),$$

(3.5)

where each $\phi_{k,m}(x)$ is a basis function which is local, in the sense that it contributes to the approximation only on the set $S_m$; that is, it takes the value 0 for $x \notin S_m$. Here $\hat{J}(x, \hat{r})$ is an architecture of the type discussed earlier, and the parameter vector $r$ consists of $\hat{r}$ and the coefficients $r_m(k)$ of the basis functions. Thus the portion $\hat{J}(x, \hat{r})$ of the architecture is used to capture “global” aspects of the target function, while each portion

$$\sum_{k=1}^{K_m} r_m(k) \phi_{k,m}(i)$$

is used to capture aspects of the target function that are “local” to the subset $S_m$. The book [BeT96] (Section 3.1.3) discusses the training of local-global approximation architectures with methods that are tailored to their special structure.

Architectures with Automatic Feature Construction

Unfortunately, in practice we often do not know an adequate set of features, so it is important to have methods that construct features automatically, to supplement whatever features may already be available. Indeed, there are architectures that do not rely on the knowledge of good features. We have noted the kernel methods of Example 3.1.5 in this connection. Another very popular possibility is neural networks, which we will describe in Section 3.2.
Some of these architectures involve training that constructs simultaneously both the feature vectors $\phi(x)$ and the parameter vectors $r$ that weigh them.

Generally, architectures that construct features automatically do not preclude the use of additional features that are based on a priori knowledge or understanding of the problem at hand. In particular these architectures may, in addition to $x$, use as inputs additional hand-crafted features that are relevant for the problem at hand. Another possibility is to combine automatically constructed features with other a priori known good features into a (mixed) linear architecture that involves both types of features. The weights of the latter linear architecture may be obtained with a separate second stage training process, following the first stage training process that constructs automatically suitable features using a nonlinear architecture such as a neural network.

### 3.1.3 Training of Linear and Nonlinear Architectures

In this section, we discuss briefly the training process of choosing the parameter vector $r$ of a parametric architecture $\tilde{J}(x, r)$, focusing primarily on incremental gradient methods. The most common type of training is based on a least squares optimization, also known as least squares regression. Here a set of state-cost training pairs $(x^s, \beta^s)$, $s = 1, \ldots, q$, called the training set, is collected and $r$ is determined by solving the problem

$$
\min_r \sum_{s=1}^q \left( \tilde{J}(x^s, r) - \beta^s \right)^2.
$$

Thus $r$ is chosen to minimize the sum of squared errors between the sample costs $\beta^s$ and the architecture-predicted costs $\tilde{J}(x^s, r)$. Here there is some target cost function $J$ that we aim to approximate with $\tilde{J}(\cdot, r)$, and the sample cost $\beta^s$ is the value $J(x^s)$ plus perhaps some error or “noise.”

The cost function of the training problem (3.6) is generally nonconvex, and can be quite complicated. This may pose challenges, since there may exist multiple local minima. However, for a linear architecture the cost function is convex quadratic, and the training problem admits a closed-form solution. In particular, for the linear architecture $\tilde{J}(x, r) = r'\phi(x)$, the problem becomes

$$
\min_r \sum_{s=1}^q \left( r'\phi(x^s) - \beta^s \right)^2.
$$

By setting the gradient of the quadratic objective to 0, we obtain

$$
\sum_{s=1}^q \phi(x^s) \left( r'\phi(x^s) - \beta^s \right) = 0,
$$
Sec. 3.1 Parametric Approximation Architectures

or

\[ \sum_{s=1}^{q} \phi(x^s) \phi(x^s)' \mathbf{r} = \sum_{s=1}^{q} \phi(x^s) \beta^s. \]

Thus by matrix inversion we obtain the minimizing parameter vector

\[ \hat{\mathbf{r}} = \left( \sum_{s=1}^{q} \phi(x^s) \phi(x^s)' \right)^{-1} \sum_{s=1}^{q} \phi(x^s) \beta^s. \tag{3.7} \]

If the inverse above does not exist, an additional quadratic in \( r \), called a regularization function, is added to the least squares objective to deal with this, and also to help with other issues to be discussed later. A singular value decomposition approach may also be used to deal with the matrix inversion issue; see [BeT96], Section 3.2.2.

Thus a linear architecture has the important advantage that the training problem can be solved exactly and conveniently with the formula (3.7) (of course it may be solved by any other algorithm that is suitable for linear least squares problems, including iterative algorithms). By contrast, if we use a nonlinear architecture, such as a neural network, the associated least squares problem is nonquadratic and also nonconvex, so it is hard to solve in principle. Despite this fact, through a combination of sophisticated implementation of special gradient algorithms, called incremental, and powerful computational resources, neural network methods have been successful in practice.

Incremental Gradient Methods

We will now discuss briefly special methods for solution of the nonlinear least squares training problem (3.6), assuming a parametric architecture that is differentiable in the parameter vector. This methodology can be properly viewed as a subject in nonlinear programming and iterative algorithms, and as such it can be studied independently of the approximate DP methods of this book. Thus the reader who has already some exposure to the subject may skip to the next section. The author’s nonlinear programming textbook [Ber16] and the RL book [Ber19a] provide more detailed presentations.

We view the training problem (3.6) as a special case of the minimization of a sum of component functions

\[ f(y) = \sum_{i=1}^{m} f_i(y), \tag{3.8} \]

where each \( f_i \) is a differentiable scalar function of the \( n \)-dimensional column vector \( y \) (this is the parameter vector). Thus we use the more common
symbols $y$ and $m$ in place of $r$ and $q$, respectively, and we replace the squared error terms

$$(\tilde{J}(x^r, r) - \beta^r)^2$$

in the training problem (3.6) with the generic terms $f_i(y)$.

The (ordinary) gradient method for problem (3.8) generates a sequence $\{y^k\}$ of iterates, starting from some initial guess $y^0$ for the minimum of the cost function $f$. It has the form†

$$y^{k+1} = y^k - \gamma^k \nabla f(y^k) = y^k - \gamma^k \sum_{i=1}^{m} \nabla f_i(y^k), \quad (3.9)$$

where $\gamma^k$ is a positive stepsize parameter. The incremental gradient method is similar to the ordinary gradient method, but uses the gradient of a single component of $f$ at each iteration. It has the general form

$$y^{k+1} = y^k - \gamma^k \nabla f_{i_k}(y^k), \quad (3.10)$$

where $i_k$ is some index from the set $\{1, \ldots, m\}$, chosen by some deterministic or randomized rule. Thus a single component function $f_{i_k}$ is used at iteration $k$, with great economies in gradient calculation cost over the ordinary gradient method (3.9), particularly when $m$ is large. This is of course a radical simplification, which involves a large approximation error, yet it performs surprisingly well! The idea is to attain faster convergence when far from the solution as we will explain shortly; see the author’s books [BeT96], [Ber16], and [Ber19a] for a more detailed discussion.

The method for selecting the index $i_k$ of the component to be iterated on at iteration $k$ is important for the performance of the method. We describe three common rules, the last two of which involve randomization:‡

1. A cyclic order, the simplest rule, whereby the indexes are taken up in the fixed deterministic order $1, \ldots, m$, so that $i_k$ is equal to $(k \bmod m)$ plus 1. A contiguous block of iterations involving the components $f_1, \ldots, f_m$ in this order and exactly once is called a cycle.

2. A uniform random order, whereby the index $i_k$ chosen randomly by sampling over all indexes with a uniform distribution, independently of the past history of the algorithm. This rule may perform better than the cyclic rule in some circumstances.

† We use standard calculus notation for gradients; see, e.g., [Ber16], Appendix A. In particular, $\nabla f(y)$ denotes the $n$-dimensional column vector whose components are the first partial derivatives $\partial f(y)/\partial y_i$ of $f$ with respect to the components $y_1, \ldots, y_n$ of the column vector $y$.

‡ With these stepsize rules, the incremental gradient method is often called stochastic gradient or stochastic gradient descent method.
(3) A cyclic order with random reshuffling, whereby the indexes are taken up one by one within each cycle, but their order after each cycle is reshuffled randomly (and independently of the past). This rule is used widely in practice, particularly when the number of components \( m \) is modest, for reasons to be discussed later.

Note that in the cyclic cases, it is essential to include all components in a cycle; otherwise some components will be sampled more often than others, leading to a bias in the convergence process. Similarly, it is necessary to sample according to the uniform distribution in the random order case.

Focusing for the moment on the cyclic rule (with or without reshuffling), we note that the motivation for the incremental gradient method is faster convergence: we hope that far from the solution, a single cycle of the method will be as effective as several (as many as \( m \)) iterations of the ordinary gradient method (think of the case where the components \( f_i \) are similar in structure). Near a solution, however, the incremental method may not be as effective.

To be more specific, we note that there are two complementary performance issues to consider in comparing incremental and nonincremental methods:

(a) Progress when far from convergence. Here the incremental method can be much faster. For an extreme case take \( m \) large and all components \( f_i \) identical to each other. Then an incremental iteration requires \( m \) times less computation than a classical gradient iteration, but gives exactly the same result, when the stepsize is scaled to be \( m \) times larger. While this is an extreme example, it reflects the essential mechanism by which incremental methods can be much superior: far from the minimum a single component gradient will point to “more or less” the right direction, at least most of the time; see the following example.

(b) Progress when close to convergence. Here the incremental method can be inferior. In particular, the ordinary gradient method (3.9) can be shown to converge with a constant stepsize under reasonable assumptions, see e.g., [Ber16], Chapter 1. However, the incremental method requires a diminishing stepsize, and its ultimate rate of convergence can be much slower.

This type of behavior is illustrated in the following example.

**Example 3.1.9**

Assume that \( y \) is a scalar, and that the problem is

\[
\begin{align*}
\text{minimize} & \quad f(y) = \frac{1}{2} \sum_{i=1}^{m} (c_i y - b_i)^2 \\
\text{subject to} & \quad y \in \mathbb{R},
\end{align*}
\]
where $c_i$ and $b_i$ are given scalars with $c_i \neq 0$ for all $i$. The minimum of each of the components $f_i(y) = \frac{1}{2} (c_i y - b_i)^2$ is
\[ y_i^* = \frac{b_i}{c_i}, \]
while the minimum of the least squares cost function $f$ is
\[ y^* = \frac{\sum_{i=1}^{m} c_i b_i}{\sum_{i=1}^{m} c_i^2}. \]
It can be seen that $y^*$ lies within the range of the component minima
\[ R = \left[ \min_i y_i^*, \max_i y_i^* \right], \]
and that for all $y$ outside the range $R$, the gradient
\[ \nabla f_i(y) = c_i (c_i y - b_i) \]
has the same sign as $\nabla f(y)$ (see Fig. 3.1.8). As a result, when outside the region $R$, the incremental gradient method
\[ y^{k+1} = y^k - \gamma^k c_i (c_i y^k - b_i) \]
approaches $y^*$ at each step, provided the stepsize $\gamma^k$ is small enough. In fact it can be verified that it is sufficient that
\[ \gamma^k \leq \min_i \frac{1}{c_i^2}. \]

However, for $y$ inside the region $R$, the $i$th step of a cycle of the incremental gradient method need not make progress. It will approach $y^*$ (for small enough stepsize $\gamma^k$) only if the current point $y^k$ does not lie in the interval connecting $y_i^*$ and $y_i^*$. This induces an oscillatory behavior within the region $R$, and as a result, the incremental gradient method will typically not converge to $y^*$ unless $\gamma^k \to 0$. By contrast, the ordinary gradient method, which takes the form
\[ y^{k+1} = y^k - \gamma \sum_{i=1}^{m} c_i (c_i y^k - b_i), \]
can be verified to converge to $y^*$ for any constant stepsize $\gamma$ with
\[ 0 < \gamma \leq \frac{1}{\sum_{i=1}^{m} c_i^2}. \]
However, for $y$ outside the region $R$, a full iteration of the ordinary gradient method need not make more progress towards the solution than a single step of the incremental gradient method. In other words, with comparably intelligent stepsize choices, far from the solution (outside $R$), a single pass through the entire set of cost components by incremental gradient is roughly as effective as $m$ passes by ordinary gradient.

The preceding example assumes that each component function $f_i$ has a minimum, so that the range of component minima is defined. In cases where the components $f_i$ have no minima, a similar phenomenon may occur, as illustrated by the following example (the idea here is that we may combine several components into a single component that has a minimum).
Sec. 3.1 Parametric Approximation Architectures

Figure 3.1.8. Illustrating the advantage of incrementalism when far from the optimal solution. The region of component minima

\[ R = \left[ \min_i y_i^*, \max_i y_i^* \right], \]

is labeled as the “region of confusion.” It is the region where the method does not have a clear direction towards the optimum. The ith step in an incremental gradient cycle is a gradient step for minimizing \((c_i y - b_i)^2\), so if \(y\) lies outside the region of component minima \(R = \left[ \min_i y_i^*, \max_i y_i^* \right]\) (labeled as the “farout region”) and the stepsize is small enough, progress towards the solution \(y^*\) is made.

Example 3.1.10:

Consider the case where \(f\) is the sum of increasing and decreasing convex exponentials, i.e.,

\[ f_i(y) = a_i e^{b_i y}, \quad y \in \mathbb{R}, \]

where \(a_i\) and \(b_i\) are scalars with \(a_i > 0\) and \(b_i \neq 0\). Let

\[ I^+ = \{ i \mid b_i > 0 \}, \quad I^- = \{ i \mid b_i < 0 \}, \]

and assume that \(I^+\) and \(I^-\) have roughly equal numbers of components. Let also \(y^*\) be the minimum of \(\sum_{i=1}^{m} f_i\).

Consider the incremental gradient method that given the current point, call it \(y^k\), chooses some component \(f_{i_k}\) and iterates according to the incremental gradient iteration

\[ y^{k+1} = y^k - \alpha^k \nabla f_{i_k}(y^k). \]

Then it can be seen that if \(y^k \gg y^*\), \(y^{k+1}\) will be substantially closer to \(y^*\) if \(i \in I^+\), and negligibly further away than \(y^*\) if \(i \in I^-\). The net effect, averaged
over many incremental iterations, is that if $y^k >> y^*$, an incremental gradient iteration makes roughly one half the progress of a full gradient iteration, with $m$ times less overhead for calculating gradients. The same is true if $y^k << y^*$.

On the other hand as $y^k$ gets closer to $y^*$ the advantage of incrementalism is reduced, similar to the preceding example. In fact in order for the incremental method to converge, a diminishing stepsize is necessary, which will ultimately make the convergence slower than the one of the nonincremental gradient method with a constant stepsize.

The discussion of the preceding examples relies on $y$ being one-dimensional, but in many multidimensional problems the same qualitative behavior can be observed. In particular, a pass through the $i$th component $f_i$ by the incremental gradient method can make progress towards the solution in the region where the component gradient $\nabla f_{ik}(y^k)$ makes an angle less than 90 degrees with the cost function gradient $\nabla f(y^k)$. If the components $f_i$ are not “too dissimilar,” this is likely to happen in a region of points that are not too close to the optimal solution set. This behavior has been verified in many practical contexts, including the training of neural networks (cf. the next section), where incremental gradient methods have been used extensively, frequently under the name backpropagation methods.

**Stepsize Choice and Diagonal Scaling**

The choice of the stepsize $\gamma^k$ plays an important role in the performance of incremental gradient methods. In practice, it is common to use a constant stepsize for a (possibly prespecified) number of iterations, then decrease the stepsize by a certain factor, and repeat, up to the point where the stepsize reaches a prespecified floor value. An alternative possibility is to use a diminishing stepsize rule of the form

$$\gamma^k = \min \left\{ \gamma, \frac{\beta_1}{k + \beta_2} \right\},$$

where $\gamma$, $\beta_1$, and $\beta_2$ are some positive scalars. There are also variants of the method that use a constant stepsize throughout, and can be shown to converge to a stationary point of $f$ under reasonable assumptions. In one type of such method the degree of incrementalism gradually diminishes as the method progresses (see [Ber97a]). Another incremental approach with similar aims, is the aggregated gradient method, which is discussed in the author’s textbooks [Ber15a], [Ber16], [Ber19a].

Regardless of whether a constant or a diminishing stepsize is ultimately used, the incremental method must use a much larger stepsize than the corresponding nonincremental gradient method (as much as $m$ times larger, so that the size of the incremental gradient step is comparable to the size of the nonincremental gradient step).

One possibility is to use an adaptive stepsize rule, whereby, roughly speaking, the stepsize is reduced (or increased) when the progress of the
method indicates that the algorithm is (or is not) oscillating. There are formal ways to implement such stepsize rules with sound convergence properties (see [Tse98], [MYF03]).

The difficulty with stepsize selection may also be addressed with diagonal scaling, i.e., using a stepsize \( \gamma^k_j \) that is different for each of the components \( y_j \) of \( y \). Second derivatives can be very useful for this purpose. In generic nonlinear programming problems of unconstrained minimization of a function \( f \), it is common to use diagonal scaling with stepsizes

\[
\gamma^k_j = \gamma \left( \frac{\partial^2 f(y^k)}{\partial^2 y_j} \right)^{-1}, \quad j = 1, \ldots, n,
\]

where \( \gamma \) is a constant that is nearly equal 1 (the second derivatives may also be approximated by gradient difference approximations). However, in least squares training problems, this type of scaling is inconvenient because of the additive form of \( f \) as a sum of a large number of component functions:

\[
f(y) = \sum_{i=1}^{m} f_i(y),
\]

cf. Eq. (3.8). The neural network literature includes a number of practical scaling schemes, some of which have been incorporated in publicly and commercially available software.

The RL book [Ber19a] (Section 3.1.3) describes another type method that involves second derivatives and is based on Newton’s method. The idea here is to write Newton’s method in a format that is well suited to the additive character of the cost function \( f \), and involves low order matrix inversion. One can then implement diagonal scaling by setting to zero the off-diagonal terms of the inverted matrices, so that the algorithm involves no matrix inversion. There is also another related algorithm, which is based on the Gauss-Newton method and the extended Kalman filter; see the author’s paper [Ber96], and the books [BeT96] and [Ber16].

### 3.2 NEURAL NETWORKS

There are several different types of neural networks that can be used for a variety of tasks, such as pattern recognition, classification, image and speech recognition, natural language processing, and others. In this section, we focus on our finite horizon DP context, and the role that neural networks can play in approximating the optimal cost-to-go functions \( J^*_k \). As an example within this context, we may first use a neural network to construct an approximation to \( J^*_{N-1} \). Then we may use this approximation to approximate \( J^*_{N-2} \), and continue this process backwards in time, to obtain approximations to all the optimal cost-to-go functions \( J^*_k, k = 1, \ldots, N-1, \) as we will discuss in more detail in Section 3.3.
Throughout this section, we will focus on the type of neural network, known as a multilayer perceptron, which is the one most used at present in the RL applications discussed in these notes. Naturally, there are variations that are adapted to the problem at hand. For example AlphaZero uses a specialized neural network that can take advantage of the board-like structure of chess and Go to facilitate and expedite the associated computations.

To describe the use of neural networks in finite horizon DP, let us consider the typical stage $k$, and for convenience drop the index $k$; the subsequent discussion applies to each value of $k$ separately. We consider parametric architectures $\tilde{J}(x, v, r)$ of the form

$$\tilde{J}(x, v, r) = r'\phi(x, v)$$

(3.11)

that depend on two parameter vectors $v$ and $r$. Our objective is to select $v$ and $r$ so that $\tilde{J}(x, v, r)$ approximates some target cost function that can be sampled (possibly with some error). The process is to collect a training set that consists of a large number of state-cost pairs $(x^s, \beta^s), s = 1, \ldots, q$, and to find a function $\tilde{J}(x, v, r)$ of the form (3.11) that matches the training set in a least squares sense, i.e., $(v, r)$ minimizes

$$\sum_{s=1}^{q} (\tilde{J}(x^s, v, r) - \beta^s)^2.$$

We postpone for later the question of how the training pairs $(x^s, \beta^s)$ are generated. Notice the different roles of the two parameter vectors here: $v$ parametrizes $\phi(x, v)$, which in some interpretation may be viewed as a feature vector, and $r$ is a vector of linear weighting parameters for the components of $\phi(x, v)$.

**Single Layer Perceptron**

A neural network architecture provides a parametric class of functions $\tilde{J}(x, v, r)$ of the form (3.11) that can be used in the optimization framework just described. The simplest type of neural network is the single layer perceptron; see Fig. 3.2.1. Here the state $x$ is encoded as a vector of numerical values $y(x)$ with components $y_1(x), \ldots, y_n(x)$, which is then transformed linearly as

$$Ay(x) + b,$$

† The least squares training problem used here is based on nonlinear regression. This is a classical method for approximating the expected value of a function with a parametric architecture, and involves a least squares fit of the architecture to simulation-generated samples of the expected value. We refer to machine learning and statistics textbooks for more discussion.
Sec. 3.2 Neural Networks

25

Figure 3.2.1 Schematic illustration of a single layer perceptron, a neural network consisting of a linear layer and a nonlinear layer. It provides a way to compute features of the state, which can be used for cost function approximation. The state $x$ is encoded as a vector of numerical values $y(x)$, which is then transformed linearly as $Ay(x) + b$ in the linear layer. The $m$ scalar output components of the linear layer, become the inputs to nonlinear one-dimensional functions $\sigma : \mathbb{R} \mapsto \mathbb{R}$, thus producing the $m$ scalars

$$\phi_\ell(x, v) = \sigma((Ay(x) + b)_\ell),$$

which can be viewed as features that are in turn linearly weighted with parameters $r_\ell$.

where $A$ is an $m \times n$ matrix and $b$ is a vector in $\mathbb{R}^m$.† This transformation is called the linear layer of the neural network. We view the components of $A$ and $b$ as parameters to be determined, and we group them together into the parameter vector $v = (A, b)$.

Each of the $m$ scalar output components of the linear layer,

$$(Ay(x) + b)_\ell, \quad \ell = 1, \ldots, m,$$

becomes the input to a nonlinear differentiable and monotonically increasing function $\sigma$ that maps scalars to scalars. A simple and popular possibility is the rectified linear unit (ReLU for short), which is simply the function $\max\{0, \xi\}$, approximated by a differentiable function $\sigma$ by some form of smoothing operation; for example $\sigma(\xi) = \ln(1 + e^\xi)$, which is illustrated in Fig. 3.2.2. Other functions, used since the early days of neural networks, have the property

$$-\infty < \lim_{\xi \to -\infty} \sigma(\xi) < \lim_{\xi \to \infty} \sigma(\xi) < \infty;$$

† The method of encoding $x$ into the numerical vector $y(x)$ is generally problem-dependent, but it can be critical for the success of the training process. We should note also that some of the components of $y(x)$ could be known interesting features of $x$ that can be designed based on problem-specific knowledge.
The rectified linear unit $\sigma(\xi) = \ln(1 + e^\xi)$. It is the function $\max\{0, \xi\}$ with its corner “smoothed out.” Its derivative is $\sigma'(\xi) = e^\xi/(1 + e^\xi)$, and approaches 0 and 1 as $\xi \to -\infty$ and $\xi \to \infty$, respectively.

Some examples of sigmoid functions. The hyperbolic tangent function is on the left, while the logistic function is on the right.

see Fig. 3.2.3. Such functions are called sigmoid functions, and some common choices are the hyperbolic tangent function

$$\sigma(\xi) = \tanh(\xi) = \frac{e^\xi - e^{-\xi}}{e^\xi + e^{-\xi}},$$

and the logistic function

$$\sigma(\xi) = \frac{1}{1 + e^{-\xi}}.$$  

In what follows, we will ignore the character of the function $\sigma$ (except for differentiability), and simply refer to it as a “nonlinear unit” and to the corresponding layer as a “nonlinear layer.”

At the outputs of the nonlinear units, we obtain the scalars

$$\phi_\ell(x, v) = \sigma((Ay(x) + b)_\ell), \quad \ell = 1, \ldots, m.$$  

One possible interpretation is to view $\phi_\ell(x, v)$ as features of $x$, which are linearly combined using weights $r_\ell$, $\ell = 1, \ldots, m$, to produce the final
Figure 3.2.4 Nonlinear architecture with a view of the state encoding process as a feature extraction mapping preceding the neural network. The state encoder may also contain tunable parameters.

output

\[ \hat{J}(x, v, r) = \sum_{\ell=1}^{m} r_{\ell} \phi_{\ell}(x, v) = \sum_{\ell=1}^{m} r_{\ell} \sigma \left( (Ay(x) + b)_{\ell} \right). \]

Note that each value \( \phi_{\ell}(x, v) \) depends on just the \( \ell \)th row of \( A \) and the \( \ell \)th component of \( b \), not on the entire vector \( v \). In some cases this motivates placing some constraints on individual components of \( A \) and \( b \) to achieve special problem-dependent “handcrafted” effects.

**State Encoding and Direct Feature Extraction**

The state encoding operation that transforms \( x \) into the neural network input \( y(x) \) can be instrumental in the success of the approximation scheme. Examples of state encodings are components of the state \( x \), numerical representations of qualitative characteristics of \( x \), and more generally features of \( x \), i.e., functions of \( x \) that aim to capture “important nonlinearities” of the optimal cost-to-go function. With the latter view of state encoding, we may consider the approximation process as consisting of a feature extraction mapping, followed by a neural network with input the extracted features of \( x \), and output the cost-to-go approximation; see Fig. 3.2.4. In a more general view of the neural network, the state encoder may involve some tunable parameters.

The idea here is that with a good feature extraction mapping, the neural network need not be very complicated (may involve few nonlinear units and corresponding parameters), and may be trained more easily. This intuition is borne out by simple examples and practical experience. However, as is often the case with neural networks, it is hard to support it with a quantitative analysis.

**Universal Approximation Property of Neural Networks**

An important question is how well we can approximate the target function \( J_k^* \) with a neural network architecture, assuming we can choose the number of the nonlinear units \( m \) to be as large as we want. The answer to
this question is quite favorable and is provided by the so-called \textit{universal approximation theorem}.

Roughly, the theorem says that assuming that $x$ is an element of a Euclidean space $X$ and $y(x) \equiv x$, a neural network of the form described can approximate arbitrarily closely (in an appropriate mathematical sense), over a compact subset $S \subseteq X$, any piecewise continuous function $J : S \to \mathbb{R}$, provided the number $m$ of nonlinear units is sufficiently large. For proofs of the theorem, we refer to Cybenko \cite{Cyb89}, Funahashi \cite{Fun89}, Hornik, Stinchcombe, and White \cite{HSW89}, and Leshno et al. \cite{LLP93}. For additional sources and intuitive explanations we refer to Bishop (\cite{Bis95}, pp. 129-130), Jones \cite{Jon90}, and the RL textbook \cite{Ber19a}, Section 3.2.1.

While the universal approximation theorem provides some assurance about the adequacy of the neural network structure, it does not predict how many nonlinear units we may need for “good” performance in a given problem. Unfortunately, this is a difficult question to even pose precisely, let alone to answer adequately. In practice, one is often reduced to trying increasingly larger values of $m$ until one is convinced that satisfactory performance has been obtained for the task at hand. One may improve on trial-and-error schemes with more systematic hyperparameter search methods, such as Bayesian optimization, and in fact this has been used to tune the parameters of the deep network used by AlphaZero. Experience has shown that in many cases the number of required nonlinear units and corresponding dimension of $A$ can be very large, adding significantly to the difficulty of solving the training problem. This has given rise to many suggestions for modifications of the neural network structure. An important possibility is to concatenate multiple single layer perceptrons so that the output of the nonlinear layer of one perceptron becomes the input to the linear layer of the next, giving rise to deep neural networks, which we will discuss later.

\subsection*{3.2.1 Training of Neural Networks}

Given a set of state-cost training pairs $(x^s, \beta^s)$, $s = 1, \ldots, q$, the parameters of the neural network $A$, $b$, and $r$ are obtained by solving the problem

$$\min_{A,b,r} \sum_{s=1}^{q} \left( \sum_{\ell=1}^{m} r_{\ell} \sigma \left( (Ay(x^s) + b)^{\ell} \right) - \beta^s \right)^2. \quad (3.12)$$

Note that the cost function of this problem is generally nonconvex, so there may exist multiple local minima.

In practice it is common to augment the cost function of this problem with a \textit{regularization} function, such as a quadratic in the parameters $A$, $b$, and $r$. This is customary in least squares problems in order to make the problem easier to solve algorithmically. However, in the context of neural network training, regularization is primarily important for a different
reason: it helps to avoid overfitting, which occurs when the number of parameters of the neural network is relatively large (comparable to the size of the training set). In this case a neural network model matches the training data very well but may not do as well on new data. This is a known difficulty, which is the subject of much current research, particularly in the context of deep neural networks.

An important issue is to select a method to solve the training problem (3.12). While we can use any unconstrained optimization method that is based on gradients, in practice it is important to take into account the cost function structure of problem (3.12). The salient characteristic of this cost function is that it is the sum of a potentially very large number \( q \) of component functions. This makes the computation of the cost function value of the training problem and/or its gradient very costly. For this reason the incremental methods of Section 3.1.3 are universally used for training.† Experience has shown that these methods can be vastly superior to their nonincremental counterparts in the context of neural network training.

The implementation of the training process has benefited from experience that has been accumulated over time, and has provided guidelines for scaling, regularization, initial parameter selection, and other practical issues; we refer to books on neural networks such as Bishop [Bis95], Goodfellow, Bengio, and Courville [GBC16], and Haykin [Hay08], and to the overview paper on deep neural network training [Sun19] for related accounts. Still, incremental methods can be quite slow, and training may be a time-consuming process. Fortunately, the training is ordinarily done off-line, possibly using parallel computation, in which case computation time may not be a serious issue. Moreover, in practice the neural network training problem typically need not be solved with great accuracy. This is also supported by the Newton step view of approximation in value space, which suggests that great accuracy in the terminal cost function approximation is not critically important for good performance of the on-line play controller.

### 3.2.2 Multilayer and Deep Neural Networks

An important generalization of the single layer perceptron architecture involves a concatenation of multiple layers of linear and nonlinear functions; see Fig. 3.2.5. In particular the outputs of each nonlinear layer become the inputs of the next linear layer. In some cases it may make sense to add

† The incremental methods are valid for an arbitrary order of component selection within the cycle, but it is common to randomize the order at the beginning of each cycle. Also, in a variation of the basic method, we may operate on a batch of several components at each iteration, called a minibatch, rather than a single component. This has an averaging effect, which reduces the tendency of the method to oscillate and allows for the use of a larger stepsize; see the end-of-chapter references.
as additional inputs some of the components of the state $x$ or the state encoding $y(x)$.

In the early days of neural networks practitioners tended to use few nonlinear layers (say one to three). However, more recently a lot of success in certain problem domains (including image and speech processing, as well as approximate DP) has been achieved with deep neural networks, which involve a considerably larger number of layers.

There are a few questions to consider here. The first has to do with the reason for having multiple nonlinear layers, when a single one is sufficient to guarantee the universal approximation property. Here are some qualitative (and somewhat speculative) explanations:

(a) If we view the outputs of each nonlinear layer as features, we see that the multilayer network produces a hierarchy of features, where each set of features is a function of the preceding set of features [except for the first set of features, which is a function of the encoding $y(x)$ of the state $x$]. In the context of specific applications, this hierarchical structure can be exploited to specialize the role of some of the layers and to enhance some characteristics of the state.

(b) Given the presence of multiple linear layers, one may consider the possibility of using matrices $A$ with a particular sparsity pattern, or other structure that embodies special linear operations such as convolution, which may be well-matched to the training problem at hand. Moreover, when such structures are used, the training problem often becomes easier, because the number of parameters in the linear layers is drastically decreased.

(c) Overparametrization (more weights than data, as in a deep neural network) helps to mitigate the detrimental effects of overfitting, and the attendant need for regularization. The explanation for this fascinating phenomenon (observed as early as the late 90s) is the subject of much current research; see [ZBH16], [BMM18], [BRT18], [SJI18], [ADH19], [BLL19], [HMR19], [MVS19], [SuY19], [Sun19], [HaR21],

<table>
<thead>
<tr>
<th>State Encoding</th>
<th>Linear Layer</th>
<th>Nonlinear Layer</th>
<th>Linear Layer</th>
<th>Nonlinear Layer</th>
<th>Linear Weighting</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$r\phi(x,v)$</td>
</tr>
</tbody>
</table>

Figure 3.2.5 A deep neural network, with multiple layers. Each nonlinear layer constructs the inputs of the next linear layer.
We finally note that the use of deep neural networks has been an important factor for the success of the AlphaGo and AlphaZero programs that play Go and chess, respectively; see [SHM16], [SHS17]. By contrast, Tesauro’s backgammon program and its descendants have performed well with one or two nonlinear layers [PaR12]. Moreover, as new applications of approximate DP/RL are being considered, it is likely that different and/or specialized neural network architectures will be discovered, which may be better suited to the structure of these applications.

3.3 TRAINING OF COST FUNCTIONS IN APPROXIMATE DP

In the context of approximate DP/RL, architectures are mainly used to approximate either cost functions or policies. When a neural network is involved, the terms value network and policy network are commonly used, respectively.† In this section we will illustrate the use of value networks in finite horizon DP, while in the next section we will discuss the use of policy networks. We will also illustrate in Section 3.3.3 the combined use of policy and value networks within an approximate policy iteration context, whereby the policies and their cost functions are approximated by a policy and a value network, respectively, to generate a sequence of (approximately) improved policies. Finally, in Sections 3.3.4 and 3.4.5, we will describe how approximating Q-factor or cost differences (rather than Q-factors or costs) can be beneficial within our context of approximation in value space.

3.3.1 Fitted Value Iteration

Let us describe a popular approach for training an approximation architecture $J_k(x_k, r_k)$ for a finite horizon DP problem. The parameter vectors $r_k$ are determined sequentially, starting from the end of the horizon, and proceeding backwards as in the DP algorithm: first $r_{N-1}$ then $r_{N-2}$, and so on. The algorithm samples the state space for each stage $k$, and generates a large number of states $x^s_k$, $s = 1, \ldots, q$. It then determines sequentially the parameter vectors $r_k$ to obtain a good “least squares fit” to the DP algorithm. The method can also be used in the infinite horizon case, in essentially identical form, and it is commonly called fitted value iteration.

In particular, each $r_k$ is determined by generating a large number of sample states and solving a least squares problem that aims to minimize the error in satisfying the DP equation for these states at time $k$. At

† The alternative terms critic network and actor network are also used often. In these notes, we will use the terms “value network” and “policy network.”
the typical stage $k$, having obtained $r_{k+1}$, we determine $r_k$ from the least squares problem

$$r_k \in \arg \min_r \sum_{s=1}^q \left( \tilde{J}_k(x_k^s, r) - \min_{u \in U_k(x_k^s)} E\left\{ g_k(x_k^s, u, w_k) + \tilde{J}_{k+1}(f_k(x_k^s, u, w_k), r_{k+1}) \right\} \right)^2$$

where $x_k^s, i = 1, \ldots, q$, are the sample states that have been generated for the $k$th stage. Since $r_{k+1}$ is assumed to be already known, the complicated minimization term in the right side of this equation is the known scalar

$$\beta_k^s = \min_{u \in U_k(x_k^s)} E\left\{ g_k(x_k^s, u, w_k) + \tilde{J}_{k+1}(f_k(x_k^s, u, w_k), r_{k+1}) \right\}, \quad (3.13)$$

so that $r_k$ is obtained as

$$r_k \in \arg \min_r \sum_{s=1}^q (\tilde{J}_k(x_k^s, r) - \beta_k^s)^2. \quad (3.14)$$

The algorithm starts at stage $N - 1$ with the minimization

$$r_{N-1} \in \arg \min_r \sum_{s=1}^q \left( \tilde{J}_{N-1}(x_{N-1}^s, r) - \min_{u \in U_{N-1}(x_{N-1}^s)} E\left\{ g_{N-1}(x_{N-1}^s, u, w_{N-1}) + g_N(f_{N-1}(x_{N-1}^s, u, w_{N-1})) \right\} \right)^2$$

and ends with the calculation of $r_0$ at $k = 0$.

In the case of a linear architecture, where the approximate cost-to-go functions are

$$\tilde{J}_k(x_k, r_k) = r'_k \phi_k(x_k), \quad k = 0, \ldots, N - 1,$$

the least squares problem (3.14) greatly simplifies, and admits the closed form solution

$$r_k = \left( \sum_{s=1}^q \phi_k(x_k^s)\phi_k(x_k^s)' \right)^{-1} \sum_{s=1}^q \beta_k^s \phi_k(x_k^s);$$

cf. Eq. (3.7). For a nonlinear architecture such as a neural network, incremental gradient algorithms may be used.

An important implementation issue is how to select the sample states $x_k^s, s = 1, \ldots, q, k = 0, \ldots, N - 1$. In practice, they are typically obtained
by some form of Monte Carlo simulation, but the distribution by which they are generated is important for the success of the method. In particular, it is important that the sample states are “representative” in the sense that they are visited often under a nearly optimal policy. More precisely, the frequencies with which various states appear in the sample should be roughly proportional to the probabilities of their occurrence under an optimal policy.

Aside from the issue of selection of the sampling distribution that we have just described, a difficulty with fitted value iteration arises when the horizon $N$ is very long, since then the total number of parameters over the $N$ stages may become excessive. In this case, however, the problem is often stationary, in the sense that the system and cost per stage do not change as time progresses. Then it may be possible to treat the problem as one with an infinite horizon and bring to bear additional methods for training approximation architectures; see the relevant discussions in Chapter 5 of the book [Ber19a].

We finally note an important difficulty with the training method of this section: the calculation of each sample $\beta_s^k$ of Eq. (3.13) requires a minimization of an expected value, which can be very time consuming. In the next section, we describe an alternative type of fitted value iteration, which uses Q-factors, and involves a simpler minimization, whereby the order of the minimization and expectation operations in Eq. (3.13) is reversed.

### 3.3.2 Q-Factor Parametric Approximation - Model-Free Implementation

We will now consider an alternative form of approximation in value space and fitted value iteration, which involves approximation of the optimal Q-factors of state-control pairs $(x_k, u_k)$ at time $k$, with no intermediate approximation of cost-to-go functions. An important characteristic of this algorithm is that it allows for a model-free computation (i.e., the use of a computer model in place of a mathematical model).

We recall that the optimal Q-factors are defined by

$$Q^*_k(x_k, u_k) = E\{g_k(x_k, u_k, w_k) + J^*_k+1(f_k(x_k, u_k, w_k))\}, \quad k = 0, \ldots, N-1,$$

where $J^*_k+1$ is the optimal cost-to-go function for stage $k+1$. Thus $Q^*_k(x_k, u_k)$ is the cost attained by using $u_k$ at state $x_k$, and subsequently using an optimal policy.

As noted in Section 1.3, the DP algorithm can be written as

$$J^*_k(x_k) = \min_{u \in U_k(x_k)} Q^*_k(x_k, u_k),$$
and by using this equation, we can write Eq. (3.15) in the following equivalent form that relates $Q_k^*$ with $Q_{k+1}^*$:

$$Q_k^*(x_k, u_k) = E\left\{ g_k(x_k, u_k, w_k) + \min_{u \in U_{k+1}(f_k(x_k, u_k, w_k))} Q_{k+1}^*(f_k(x_k, u_k, w_k), u) \right\}. \quad (3.16)$$

This suggests that in place of the Q-factors $Q_k^*(x_k, u_k)$, we may use Q-factor approximations as the basis for suboptimal control.

We can obtain such approximations by using methods that are similar to the ones we have considered so far. Parametric Q-factor approximations $\tilde{Q}_k(x_k, u_k, r_k)$ may involve a neural network, or a feature-based linear architecture. The feature vector may depend on just the state, or on both the state and the control. In the former case, the architecture has the form

$$\tilde{Q}_k(x_k, u_k, r_k) = r_k'u_k^r \phi_k(x_k), \quad (3.17)$$

where $r_k(u_k)$ is a separate weight vector for each control $u_k$. In the latter case, the architecture has the form

$$\tilde{Q}_k(x_k, u_k, r_k) = r_k^' \phi_k(x_k, u_k), \quad (3.18)$$

where $r_k$ is a weight vector that is independent of $u_k$. The architecture (3.17) is suitable for problems with a relatively small number of control options at each stage. In what follows, we will focus on the architecture (3.18), but the discussion, with few modifications, also applies to the architecture (3.17) and to nonlinear architectures as well.

We may adapt the fitted value iteration approach of the preceding section to compute sequentially the parameter vectors $r_k$ in Q-factor parametric approximations, starting from $k = N - 1$. This algorithm is based on Eq. (3.16), with $r_k$ obtained by solving least squares problems similar to the ones of the cost function approximation case [cf. Eq. (3.14)]. As an example, the parameters $r_k$ of the architecture (3.18) are computed sequentially by collecting sample state-control pairs $(x_{s,k}, u_{s,k})$, $s = 1, \ldots, q$, and solving the linear least squares problems

$$r_k \in \arg \min_r \sum_{s=1}^q \left( r'^{r_s} \phi_k(x_{s,k}, u_{s,k}) - \beta_{k,s}^r \right)^2, \quad (3.19)$$

where

$$\beta_{k,s}^r = E \left\{ g_k(x_{s,k}^s, u_{s,k}^s, w_k) + \min_{u \in U_{k+1}(f_k(x_{s,k}^s, u_{s,k}^s, w_k))} r_{k+1}'^r \phi_{k+1}(f_k(x_{s,k}^s, u_{s,k}^s, w_k), u) \right\}. \quad (3.20)$$
Thus, having obtained $r_{k+1}$, we obtain $r_k$ through a least squares fit that aims to minimize the sum of the squared errors in satisfying Eq. (3.16). Note that the solution of the least squares problem (3.19) can be obtained in closed form as

$$r_k = \left( \sum_{s=1}^{q} \phi_k(x_s^k, u_s^k)\phi_k(x_s^k, u_s^k)' \right)^{-1} \sum_{s=1}^{q} \beta_s^k \phi_k(x_s^k, u_s^k);$$

[cf. Eq. (3.7)]. Once $r_k$ has been computed, the one-step lookahead control $\tilde{\mu}_k(x_k)$ is obtained on-line as

$$\tilde{\mu}_k(x_k) \in \arg \min_{u \in U_k(x_k)} \tilde{Q}_k(x_k, u, r_k),$$

without the need to calculate any expected value. This latter property is a primary incentive for using Q-factors in approximate DP, particularly when there are tight constraints on the amount of on-line computation that is possible in the given practical setting.

The samples $\beta_s^k$ of Eq. (3.20) involve the exact computation of an expected value. In an alternative implementation, we may replace $\beta_s^k$ with an average of just a few samples (even a single sample) of the random variable

$$g_k(x_s^k, u_s^k, w_k) + \min_{u \in U_{k+1}(f_k(x_s^k, u_s^k, w_k))} r_{k+1} \phi_{k+1}(f_k(x_s^k, u_s^k, w_k), u),$$

collected according to the probability distribution of $w_k$. This distribution may either be known explicitly, or in a model-free situation, through a computer simulator. In particular, to implement this scheme, we only need a simulator that for any pair $(x_k, u_k)$ generates a sample of the stage cost $g_k(x_k, u_k, w_k)$ and the next state $f_k(x_k, u_k, w_k)$ according to the distribution of $w_k$.

Note that the samples of the random variable (3.22) do not require the computation of an expected value like the samples (3.13) in the cost approximation method of the preceding chapter. Moreover the samples of (3.22) involve a simpler minimization than the samples (3.13). This is an important advantage of working with Q-factors rather than state costs.

Having obtained the weight vectors $r_0, \ldots, r_{N-1}$, and hence the one-step lookahead policy $\tilde{\pi} = \{\tilde{\mu}_0, \ldots, \tilde{\mu}_{N-1}\}$ through Eq. (3.21), a further possibility is to approximate this policy with a parametric architecture. This is approximation in policy space built on top of approximation in value space. The idea here is to simplify even further the on-line computation of the suboptimal controls by avoiding the minimization of Eq. (3.21).
3.3.3 Parametric Approximation in Infinite Horizon Problems
- Approximate Policy Iteration

In this section we will briefly discuss parametric approximation methods for infinite horizon problems, based on the policy iteration (PI) method. We will focus on the finite-state version of the \( \alpha \)-discounted problem of Section 1.4.1, and adopt notation that is more convenient for such problems. In particular, states and successor states will be denoted by \( i \) and \( j \), respectively. Moreover the system equation will be represented by control-dependent transition probabilities \( p_{ij}(u) \) (the probability that the system will move to state \( j \), given that it starts at state \( i \) and control \( u \) is applied). For a state-control pair \((i, u)\), the average cost per stage is denoted by \( g(i, u, j) \).

We recall that the PI algorithm in its exact form produces a sequence of stationary policies whose cost functions are progressively improving and converge in a finite number of iterations to the optimal. The corresponding convergence proof relies on the generic cost improvement property of PI, and depends on the finiteness of the state and control spaces. This proof, together with other PI-related convergence proofs, can be found in the author’s textbooks [Ber17a] or [Ber19a].

Let us state the exact form of the PI algorithm in terms of \( Q \)-factors, and in a form that is suitable for the use of approximations and simulation-based implementations. Given any policy \( \mu \), it generates the next policy \( \tilde{\mu} \) with a two-step process as follows (cf. Section 1.4.1):

(a) Policy evaluation: We compute the cost function \( J_{\mu} \) of \( \mu \) and its associated \( Q \)-factors, which are given by

\[
Q_{\mu}(i, u) = \sum_{j=1}^{n} p_{ij}(u)(g(i, u, j) + \alpha J_{\mu}(j)), \quad i = 1, \ldots, n, \ u \in U(i).
\]

Thus \( Q_{\mu}(i, u) \) is the cost of starting at \( i \), using \( u \) at the first stage, and then using \( \mu \) for the remaining stages.

(b) Policy improvement: We compute the new policy \( \tilde{\mu} \) according to

\[
\tilde{\mu}(i) \in \arg \min_{u \in U(i)} Q_{\mu}(i, u), \quad i = 1, \ldots, n.
\]

Let us now describe one way to approximate the two steps of the preceding process.

(a) Approximate policy evaluation: Here we introduce a parametric architecture \( \tilde{Q}_{\mu}(i, u, r) \) for the \( Q \)-factors of \( \mu \). We determine the value of the parameter vector \( r \) by generating (using a simulator of the system) a large number of training triplets \((i^s, u^s, \beta^s)\), \( s = 1, \ldots, q \), and
by using a least squares fit:

$$\tau \in \arg \min_r \sum_{s=1}^q (\tilde{Q}_\mu(i^s, u^s, r) - \beta^s)^2. \quad (3.23)$$

In particular, for a given pair \((i^s, u^s)\), the scalar \(\beta^s\) is generated by starting at \(i^s\), using \(u^s\) at the first stage, and simulating a trajectory of states and controls using \(\mu\) for some number \(k\) of subsequent stages. Thus, \(\beta^s\) is a sample of \(Q_k^\mu(i^s, u^s)\), the \(k\)-stage \(Q\)-factor of \(\mu\), which in the limit as \(k \to \infty\) yields the infinite horizon \(Q\)-factor of \(\mu\). The number of stages \(k\) may be either large, or fairly small. However, in the latter case some terminal cost function approximation should be added at the end of the \(k\)-stage trajectory, to compensate for the difference \(|Q_k(i, u) - Q_k^\mu(i, u)|\), which decreases in proportion to \(\alpha^k\), and may be large when \(k\) is small. Such a function may be obtained with additional training or from a previous iteration.

(b) Approximate policy improvement: Here we compute the new policy \(\tilde{\mu}\) according to

$$\tilde{\mu}(i) \in \arg \min_{u \in U(i)} \tilde{Q}_\mu(i, u, \tau), \quad i = 1, \ldots, n, \quad (3.24)$$

where \(\tau\) is the parameter vector obtained from the policy evaluation formula (3.23).

An important alternative for approximate policy improvement, is to compute a set of pairs \((i^s, \tilde{\mu}(i^s))\), \(s = 1, \ldots, q\), using Eq. (3.24), and fit these pairs with a policy approximation architecture (see the next section on approximation in policy space). The overall scheme then becomes policy iteration that is based on approximation in both value and policy spaces.

At the end of the last policy evaluation step of PI, we have obtained a final \(Q\)-factor approximation \(\tilde{Q}(i, u, \tilde{r})\). Then, in on-line play mode, we may apply the policy

$$\tilde{\mu}(i) \in \arg \min_{u \in U(i)} \tilde{Q}(i, u, \tilde{r}),$$

i.e., use the (would be) next policy iterate. Alternatively, we may apply the one-step lookahead policy

$$\tilde{\mu}(i) \in \arg \min_{u \in U(i)} \left[ \sum_{j=1}^n p_{ij}(u) \left( g(i, u, j) + \alpha \min_{u' \in U(j)} \tilde{Q}(j, u', \tilde{r}) \right) \right], \quad (3.25)$$

or its multistep lookahead version. The latter alternative implements a Newton step and will likely result in substantially better performance.
However, it is more time consuming, particularly if it is implemented by using a computer model and model-free simulation. Still another possibility, which also implements a Newton step, is to replace the function

$$\min_{u' \in U(j)} \tilde{Q}(j, u', \tilde{r})$$

in the preceding Eq. (3.25) with an off-line trained approximation.

Issues Relating to Approximate Policy Iteration

Approximate PI in its various forms has been the subject of extensive research, both theoretical and applied. A more detailed discussion is beyond our scope, and we refer to the literature, as well as Chapters 6 and 7 of the DP textbook [Ber12] or the RL textbook [Ber19a] for detailed accounts. Let us provide a few comments.

(a) Architectural issues: The architecture $\tilde{Q}_\mu(i, u, r)$ may involve the use of features, and it could be linear, or it could be nonlinear such as a neural network. A major advantage of a linear feature-based architecture is that the policy evaluation (3.23) is a linear least squares problem, which admits a closed-form solution. Moreover, when linear architectures are used, there is a broader variety of approximate policy evaluation methods with solid theoretical performance guarantees, such as TD($\lambda$), LSTD($\lambda$), and LSPE($\lambda$), which are not described in these notes, but are discussed extensively in the literature, including the DP textbook [Ber12] and the RL textbook [Ber19a].

(b) Exploration issues: Generating an appropriate set of training triplets ($i^*, u^*, \beta^*$) at the policy evaluation step poses considerable challenges, and the literature contains several related proposals. A generic difficulty has to do with inadequate exploration. In particular, to evaluate a policy $\mu$, we may need to generate $Q$-factor samples of $\mu$ starting from states frequently visited by $\mu$, but this may bias the simulation by underrepresenting states that are unlikely to occur under $\mu$. As a result, the $Q$-factor estimates of these underrepresented states may be highly inaccurate, causing potentially serious errors in the calculation of the improved control policy $\hat{\mu}$ via the policy improvement Eq. (3.24).

One possibility to improve the exploration of the state space is to use a large number of initial states to form a rich and representative subset. It may then be necessary to use relatively short trajectories to keep the cost of the simulation low. However, when using short trajectories it may be important to introduce a terminal cost function approximation in the policy evaluation step in order to make the cost sample $\beta^*$ more accurate. There have been other related approaches to improve exploration, such as using a so-called off-policy, i.e., a
policy $\mu'$ other than the currently evaluated policy $\mu$, to visit states that are unlikely to be visited using $\mu$. See the discussions in Section 6.4 of the DP textbook [Ber12].

(c) Oscillation issues: Contrary to exact PI, which is guaranteed to yield an optimal policy, approximate PI produces a sequence of policies, which are only guaranteed to lie asymptotically within a certain error bound from the optimal; see the books [BeT96], Section 6.2.2, and [Ber12], Section 2.5. Moreover, the generated policies may oscillate. By this we mean that after a few iterations, policies tend to repeat in cycles.

The associated parameter vectors $r$ may also tend to oscillate, although it is possible that there is convergence in parameter space and oscillation in policy space. This phenomenon, known as chattering, is explained in the author’s survey paper [Ber11b], and book [Ber12] (Section 6.4.3), and can be particularly damaging, because there is no guarantee that the policies involved in the oscillation are “good” policies, and there is often no way to verify how well they perform relative to the optimal. We note, however, that oscillations can be avoided and approximate PI can be shown to converge under special conditions, which arise in particular when an aggregation approach is used; see the approximate PI survey [Ber11b].

We refer to the literature for further discussion of the preceding issues, as well as a variety of other approximate PI methods.

### 3.3.4 Optimistic Policy Iteration with Parametric Q-Factor Approximation - SARSA and DQN

There are also “optimistic” approximate PI methods with Q-factor approximation, and/or a few samples in between policy updates. Because of the use of Q-factors and the limited number of samples between policy updates, these schemes have the potential of on-line play implementation, but a number of difficulties must be overcome in this case, as we will explain later in this section.

As an example, let us consider an extreme version of Q-factor parametric approximation that uses a single sample between policy updates. At the start of iteration $k$, we have the current parameter vector $r^k$, and we are at some state $i^k$, and we have chosen a control $u^k$. Then:

1. We simulate the next transition $(i^k, i^{k+1})$ using the transition probabilities $p_{i^k,j}(u^k)$.

2. We generate the control $u^{k+1}$ with the minimization

$$u^{k+1} \in \arg \min_{u \in U(i^{k+1})} \tilde{Q}(i^{k+1}, u, r^k).$$ (3.26)
In some schemes, to enhance exploration, \( u^{k+1} \) is chosen with a small probability to be a random element of \( U(i^{k+1}) \) or one that is “\( \epsilon \)-greedy,” i.e., attains within some \( \epsilon \) the minimum above. This is commonly referred to as the use of an off-policy.

We update the parameter vector via

\[
r^{k+1} = r^k - \gamma^k \nabla \tilde{Q}(i^k, u^k, r^k) \cdot (\tilde{Q}(i^k, u^k, r^k) - g(i^k, u^k, i^{k+1}) - \alpha \tilde{Q}(i^{k+1}, u^{k+1}, r^k)),
\]

where \( \gamma^k \) is a positive stepsize, and \( \nabla (\cdot) \) denotes gradient with respect to \( r \) evaluated at the current parameter vector \( r^k \). To get a sense for the rationale of this iteration, note that if \( \tilde{Q} \) is a linear feature-based architecture, \( \tilde{Q}(i^k, u^k, r^k) = \phi(i^k, u^k)'r \), then \( \nabla \tilde{Q}(i^k, u^k, r^k) \) is just the feature vector \( \phi(i^k, u^k) \), and iteration (3.27) becomes

\[
r^{k+1} = r^k - \gamma^k \phi(i^k, u^k) (\phi(i^k, u^k)'r^k - g(i^k, u^k, i^{k+1}) - \alpha \phi(i^{k+1}, u^{k+1})'r^k).
\]

Thus \( r^k \) is changed in an incremental gradient direction: the one opposite to the gradient (with respect to \( r \)) of the incremental error

\[
(\phi(i^k, u^k)'r - g(i^k, u^k, i^{k+1}) - \alpha \phi(i^{k+1}, u^{k+1})'r^k)^2,
\]

evaluated at the current iterate \( r^k \).

The process is now repeated with \( r^{k+1} \), \( i^{k+1} \), and \( u^{k+1} \) replacing \( r^k \), \( i^k \), and \( u^k \), respectively.

Extreme optimistic schemes of the type just described have received a lot of attention, in part because they admit a model-free implementation [i.e., the use of a computer simulator, which provides for each pair \( (i^k, u^k) \), the next state \( i^{k+1} \) and corresponding cost \( g(i^k, u^k, i^{k+1}) \) that are needed in Eq. (3.27)]. They are often referred to as SARSA (State-Action-Reward-State-Action): see e.g., the books [BeT96], [BBD10], [SuB18]. When Q-factor approximation is used, their behavior is very complex, their theoretical convergence properties are unclear, and there are no associated performance bounds in the literature. In practice, SARSA is more commonly used in a less extreme/optimistic form, whereby several (perhaps many) state-control-transition cost-next state samples are batched together and suitably averaged before updating the vector \( r^k \).

Other variants of the method attempt to save in sampling effort by storing the generated samples in a buffer and reusing them in some randomized fashion in subsequent iterations (cf. our earlier discussion of exploration. This is also called sometimes experience replay, an idea that has been used since the early days of RL, both to save in sampling effort and to enhance exploration. The DQN (Deep Q Network) scheme, championed by DeepMind (see Mnih et al. [MKS15]), is based on this idea (the term “Deep” is a reference to DeepMind’s affinity for deep neural networks, but experience replay does not depend on the use of a deep neural network architecture).
Q-Learning Algorithms and On-Line Play

Algorithms that approximate Q-factors, including SARSA and DQN, are fundamentally off-line training algorithms, primarily because their training process is long and requires the collection of many samples before reaching a stage that resembles parameter convergence. It can therefore be unreliable to use the interim approximate Q-factors for on-line decision making, particularly in an adaptive context that involves changing system parameters, thereby requiring on-line replanning.

On the other hand, compared to the approximate PI method of Section 3.3.3, SARSA and DQN are far better suited for on-line implementation, because the control generation process of Eq. (3.26) can also be used to select controls on-line, thereby facilitating the combination of training and on-line control selection. To this end, it is important, among others, to make sure that the parameters $r_k$ stay at reasonable levels during the on-line control process, which can be a challenge. Still, even if this difficulty can be overcome, there are a number of other difficulties that SARSA and DQN can encounter during on-line play.

(a) **On-line exploration issues**: The need to occasionally select controls using an off-policy in order to enhance exploration. Finding an off-line policy that adequately deals with exploration in a given practical context can be a challenge. Moreover, an additional concern is that the off-policy controls may improve exploration, but may be of poor quality, and in some contexts, may induce instability.

(b) **Robustness and replanning issues**: In an adaptive control context where the problem parameters are changing, the algorithm may be too slow to adapt to the changes.

(c) **Performance degradation issues**: Similar to our earlier discussion [cf. the comparison of Eqs. (3.24) and (3.25)], the minimization of Eq. (3.26) does not implement a Newton step, thereby resulting in performance loss. The alternative implementation

$$u^{k+1} = \arg \min_{u \in U^{(k+1)}} \left[ \sum_{j=1}^{n} p_{k+1,j}(u) \left( g(i^{k+1}, u, j) \right) + \alpha \min_{u' \in U(j)} \hat{Q}(j, u', r^k) \right],$$

which is patterned after Eq. (3.26), is better in this regard, but is computationally more costly, and thus less suitable for on-line implementation.

Generally speaking, the combination of off-line training and on-line play with the use of SARSA and DQN involves serious challenges. However,
in some specific contexts encouraging results have been obtained. Moreover, the methods have received a lot of attention, thanks in part to the availability of publicly available software, which also allow for a model-free implementation.

3.3.5 Approximate Policy Iteration for Infinite Horizon POMDP

In this section, we consider partial observation Markovian decision problems (POMDP) with a finite number of states and controls, and discounted additive cost over an infinite horizon. As discussed in Section 1.6.4, the optimal solution is typically intractable, so approximate DP/RL approaches must be used. In this section we focus on PI methods that are based on rollout, and approximations in policy and value space. They update a policy by using truncated rollout with that policy and a terminal cost function approximation. We focus on cost function approximation schemes, but Q-factor approximation is also possible.

Because of its simulation-based rollout character, the methodology of this section depends critically on the finiteness of the control space. It can be extended to POMDP with infinite state space but finite control space, although we will not consider this possibility in this section. In particular, we assume that there are $n$ states denoted by $i \in \{1, \ldots, n\}$ and a finite set of controls $U$ at each state. We denote by $p_{ij}(u)$ and $g(i, u, j)$ the transition probabilities and corresponding transition costs, from $i$ to $j$ under $u \in U$. The cost is accumulated over an infinite horizon and is discounted by $\alpha \in (0, 1)$. At each new generated state $j$, an observation $z$ from a finite set $Z$ is obtained with known probability $p(z \mid j, u)$ that depends on $j$ and the control $u$ that was applied prior to the generation of $j$. The objective is to select each control optimally as a function of the prior history of observations and controls.

A classical approach to this problem is to convert it to a perfect state information problem whose state is the current belief $b = (b_1, \ldots, b_n)$, where $b_i$ is the conditional distribution of the state $i$ given the prior history. As noted in Section 1.6.4, $b$ is a sufficient statistic, which can serve as a substitute for the set of available observations, in the sense that optimal control can be achieved with knowledge of just $b$.

In this section, we consider a more general form of sufficient statistic, which we call the feature state and we denote by $y$. We require that the feature state $y$ subsumes the belief state $b$. By this we mean that $b$ can be computed exactly knowing $y$. One possibility is for $y$ to be the union of $b$ and some identifiable characteristics of the belief state, or some compact representation of the measurement history up to the current time (such as a number of most recent measurements, or the state of a finite-state controller). We also make the additional assumption that $y$ can be
sequentially generated using a known feature estimator \( F(y, u, z) \). By this we mean that given that the current feature state is \( y \), control \( u \) is applied, and observation \( z \) is obtained, the next feature can be exactly predicted as \( F(y, u, z) \).

Clearly, since \( b \) is a sufficient statistic, the same is true for \( y \). Thus the optimal costs achievable by the policies that depend on \( y \) and on \( b \) are the same. However, specific suboptimal schemes may become more effective with the use of the feature state \( y \) instead of just the belief state \( b \).

The optimal cost \( J^*(y) \), as a function of the sufficient statistic/feature state \( y \), is the unique solution of the corresponding Bellman equation

\[
J^*(y) = \min_{u \in U} \left[ \hat{g}(y, u) + \alpha \sum_{z \in Z} \hat{p}(z | b_y, u) J^*(F(y, u, z)) \right].
\]

Here we use the following notation:

\( b_y \) is the belief state that corresponds to feature state \( y \), with components denoted by \( b_{y,i}, i = 1, \ldots, n \).

\( \hat{g}(y, u) \) is the expected cost per stage

\[
\hat{g}(y, u) = \sum_{i=1}^{n} b_{y,i} \sum_{j=1}^{n} p_{ij}(u) g(i, u, j).
\]

\( \hat{p}(z | b_y, u) \) is the conditional probability that the next observation will be \( z \) given the current belief state \( b_y \) and control \( u \).

\( F \) is the feature state estimator. In particular, \( F(y, u, z) \) is the next feature vector, when the current feature state is \( y \), control \( u \) is applied, and observation \( z \) is obtained.

The feature space reformulation of the problem can serve as the basis for approximation in value space, whereby \( J^* \) is replaced in Bellman’s equation by some function \( \tilde{J} \) after one-step or multistep lookahead. For example a one-step lookahead scheme yields the suboptimal policy \( \tilde{\mu} \) given by

\[
\tilde{\mu}(y) \in \arg \min_{u \in U} \left[ \hat{g}(y, u) + \alpha \sum_{z \in Z} \hat{p}(z | b_y, u) \tilde{J}(F(y, u, z)) \right].
\]

In \( \ell \)-step lookahead schemes, \( \tilde{J} \) is used as terminal cost function in an \( \ell \)-step horizon version of the original infinite horizon problem. In the standard form of a rollout algorithm, \( \tilde{J} \) is the cost function of some base policy. We will next discuss a rollout scheme with \( \ell \)-step lookahead, which involves rollout truncation and terminal cost approximation.
Figure 3.3.1 Composite system simulator for POMDP for a given policy. The starting state $i_k$ at stage $k$ of a trajectory is generated randomly using the belief state $b_k$, which is in turn computed from the feature state $y_k$.

**Truncated Rollout with Terminal Cost Function Approximation**

In the pure form of the rollout algorithm, the cost function approximation $\tilde{J}$ is the cost function $J_\mu$ of a known base policy $\mu$, and its value $\tilde{J}(y) = J_\mu(y)$ at any $y$ is obtained by first extracting $b$ from $y$, and then running a simulator starting from $b$, and using the system model, the feature generator, and $\mu$. In the truncated form of rollout, $\tilde{J}(y)$ is obtained by running the simulator of $\mu$ for a given number of steps $m$, and then adding a terminal cost approximation $\hat{J}(\bar{y})$ for each terminal feature state $\bar{y}$ that is obtained at the end of the $m$ steps of the simulation with $\mu$ (see Fig. 3.3.1).

Thus the rollout policy is defined by the base policy $\mu$, the terminal cost function approximation $\tilde{J}$, the number of steps $m$ after which the simulated trajectory with $\mu$ is truncated, as well as the lookahead size $\ell$. The choices of $m$ and $\ell$ are typically made by trial and error, based on computational tractability among other considerations, while $\tilde{J}$ may be chosen on the basis of problem-dependent insight or through the use of some off-line approximation method. In variants of the method, the multistep lookahead may be implemented approximately using a Monte Carlo tree search or adaptive sampling scheme.

Using $m$-step rollout between the $\ell$-step lookahead and the terminal cost approximation gives the method the character of a single PI. We will use repeated truncated rollout as the basis for constructing a PI algorithm, which we will discuss next.

**Supervised Learning of Rollout Policies and Cost Functions - Approximate Policy Iteration**

The rollout algorithm uses multistep lookahead and on-line simulation of the base policy to generate the rollout control at any feature state of interest. To avoid the cost of on-line simulation, we can approximate the rollout...
Sec. 3.3 Training of Cost Functions in Approximate DP

policy off-line by using some approximation architecture, which may involve a neural network. This is policy approximation built on top of the rollout scheme.

To this end, we may introduce a parametric family/architecture of policies of the form $\hat{\mu}(y, r)$, where $r$ is a parameter vector. We then construct a training set that consists of a large number of sample feature state-control pairs $(y^s, u^s), s = 1, \ldots, q,$ such that for each $s$, $u^s$ is the rollout control at feature state $y^s$. We use this data set to obtain a parameter $\vec{r}$ by solving a corresponding classification problem, which associates each feature state $y$ with a control $\hat{\mu}(y, \vec{r})$. The parameter $\vec{r}$ defines a classifier, which given a feature state $y$, classifies $y$ as requiring control $\hat{\mu}(y, \vec{r})$ (see Section 3.4).

We can also apply the rollout policy approximation to the context of PI. The idea is to view rollout as a single policy improvement, and to view the PI algorithm as a perpetual rollout process, which performs multiple policy improvements, using at each iteration the current policy as the base policy, and the next policy as the corresponding rollout policy.

In particular, we consider a PI algorithm where at the typical iteration we have a policy $\mu$, which we use as the base policy to generate many feature state-control sample pairs $(y^s, u^s), s = 1, \ldots, q,$ where $u^s$ is the rollout control corresponding to feature state $y^s$. We then obtain an “improved” policy $\hat{\mu}(y, \vec{r})$ with an approximation architecture and a classification algorithm, as described above. The “improved” policy is then used as a base policy to generate samples of the corresponding rollout policy, which is approximated in policy space, etc.

To use truncated rollout in this PI scheme, we must also provide a terminal cost approximation, which may take a variety of forms. Using zero is a simple possibility, which may work well if either the size $\ell$ of multistep lookahead or the length $m$ of the rollout is relatively large. Another possibility is to use as terminal cost in the truncated rollout an approximation of the cost function of some base policy, which may be obtained with a neural network-based approximation architecture.

In particular, at any policy iteration with a given base policy, once the rollout data is collected, one or two neural networks are constructed: A policy network that approximates the rollout policy, and (in the case of rollout with truncation) a value network that constructs a cost function approximation for that rollout policy. Thus, we may consider two types of methods:

(a) Approximate rollout and PI with truncation, where each generated policy as well as its cost function are approximated by a policy and a value network, respectively. The cost function approximation of the current policy is used to truncate the rollout trajectories that are used to train the next policy.

(b) Approximate rollout and PI without truncation, where each gener-
ated policy is approximated using a policy network, but the rollout trajectories are continued up to a large maximum number of stages (enough to make the cost of the remaining stages insignificant due to discounting) or upon reaching a termination state. The advantage of this scheme is that only a policy network is needed; a value network is unnecessary since there is no rollout truncation with cost function approximation at the end.

Note that as in all approximate PI schemes, the sampling of feature states used for training is subject to exploration concerns. In particular, for each policy approximation, it is important to include in the sample set \( \{ y^s \mid s = 1, \ldots, q \} \), a subset of feature states that are “favored” by the rollout trajectories; e.g., start from some initial subset of feature states \( y^s \) and selectively add to this subset feature states that are encountered along the rollout trajectories. This is a challenging issue, which must be approached with care.

An extensive case study of the methodology of this section was given in the paper by Bhattacharya et al. [BBW20], for the case of a pipeline repair problem. The implementation used there also includes the use of a partitioned state space architecture and an asynchronous distributed algorithm for off-line training; see Section 3.4.2.

3.3.6 Advantage Updating - Approximating Q-Factor Differences

Let us now focus on an important alternative to computing Q-factor approximations. It is motivated by the potential benefit of approximating Q-factor differences rather than Q-factors. In this method, called advantage updating, instead of computing and comparing \( Q^*_k(x_k, u_k) \) for all \( u_k \in U_k(x_k) \), we compute

\[
A_k(x_k, u_k) = Q^*_k(x_k, u_k) - \min_{u \in U_k(x_k)} Q^*_k(x_k, u).
\]

The function \( A_k(x_k, u_k) \) can serve to compare controls, i.e., at state \( x_k \) select

\[
\tilde{\mu}_k(x_k) \in \arg \min_{u \in U_k(x_k)} A_k(x_k, u),
\]

and this can also be done when \( A_k(x_k, u_k) \) is approximated with a value network.

Note that in the absence of approximations, selecting controls by advantage updating is clearly equivalent to selecting controls by comparing their Q-factors. By contrast, when approximation is involved, comparing advantages instead of Q-factors can be important, because the former may have a much smaller range of values than the latter. In particular, \( Q^*_k \) may embody sizable quantities that depend on \( x_k \) but are independent of \( u_k \), and which may interfere with algorithms such as the fitted value iteration.
Sec. 3.3 Training of Cost Functions in Approximate DP

(3.19)-(3.20). Thus, when training an architecture to approximate $Q^*_k$, the training algorithm may naturally try to capture the large scale behavior of $Q^*_k$, which may be irrelevant because it may not be reflected in the Q-factor differences $A_k$. However, with advantage updating, we may instead focus the training process on finer scale variations of $Q^*_k$, which may be all that matters. Here is an example (first given in the book [BeT96]) of what can happen when trained approximations of Q-factors are used.

**Example 3.3.1**

Consider the deterministic scalar linear system

$$x_{k+1} = x_k + \delta u_k,$$

and the quadratic cost per stage

$$g(x_k, u_k) = \delta (x_k^2 + u_k^2),$$

where $\delta$ is a very small positive constant [think of $\delta$-discretization of a continuous-time problem involving the differential equation $dx(t)/dt = u(t)$]. Let us focus on the stationary policy $\pi$, which applies at state $x$ the control

$$\mu(x) = -2x,$$

and view it as the base policy of a rollout algorithm. The Q-factors of $\pi$ over an infinite number of stages can be calculated to be

$$Q_\pi(x, u) = \frac{5x^2}{4} + \delta \left( \frac{9x^2}{4} + u^2 + \frac{5}{2} xu \right) + O(\delta^2).$$

(We omit the details of this calculation, which is based on the classical analysis of linear-quadratic optimal control problems; see e.g., Section 1.5, or [Ber17a], Section 3.1.) Thus the important part of $Q_\pi(x, u)$ for the purpose of rollout policy computation is

$$\delta \left( u^2 + \frac{5}{2} xu \right).$$

(3.28)

However, when a value network is trained to approximate $Q_\pi(x, u)$, the approximation will be dominated by $\frac{5x^2}{4}$, and the important part (3.28) will be “lost” when $\delta$ is very small. By contract, the advantage function can be calculated to be

$$A_\mu(x, u) = Q_\pi(x, u) - \min_v Q_\pi(x, v) + O(\delta^2)
= \delta \left( u^2 + \frac{5}{2} xu - \min_v \left( v^2 + \frac{5}{2} xv \right) \right) + O(\delta^2)
= \delta \left( u^2 + \frac{5}{2} xu + \frac{5^2}{4} x^2 \right) + O(\delta^2),$$

and when approximated with a value network, the approximation will be essentially unaffected by $\delta$. 
The Use of a Baseline

The idea of advantage updating is also related to the useful technique of subtracting a suitable constant (often called a baseline) from a quantity that is estimated; see Fig. 3.3.2 (in the case of advantage updating, the baselines depend on $x_k$, but the same general idea applies). This idea can also be used in the context of the fitted value iteration method given earlier, as well as in conjunction with other simulation-based methods in RL.

Example 3.1.1 also points to the connection between the ideas underlying advantage updating and the rollout methods for small stage costs relative to the cost function approximation, which we discussed in Section 2.6. In both cases it is necessary to avoid including terms of disproportionate size in the target function that is being approximated. The remedy in both cases is to subtract from the target function a suitable state-dependent baseline.
3.3.7 Differential Training of Cost Differences for Rollout

Let us now consider ways to approximate Q-factor differences (cf. our advantage updating discussion of the preceding section) by approximating cost function differences first. We recall here that given a base policy \( \pi = \{\mu_0, \ldots, \mu_{N-1}\} \), the off-line computation of an approximate rollout policy \( \tilde{\pi} = \{\tilde{\mu}_0, \ldots, \tilde{\mu}_{N-1}\} \) consists of two steps:

1. In a preliminary phase, we compute approximations \( \tilde{J}_k \) to the cost functions \( J_{k,\pi} \) of the base policy \( \pi \), possibly using simulation and a least squares fit from a parametrized class of functions.

2. Given \( \tilde{J}_k \) and a state \( x_k \) at time \( k \), we compute the approximate Q-factor \( \tilde{Q}_k(x_k, u) = E\left\{g_k(x_k, u, w_k) + \tilde{J}_{k+1}(f_k(x_k, u, w_k))\right\} \) for all \( u \in U_k(x_k) \), and we obtain the (approximate) rollout control \( \tilde{\mu}_k(x_k) \) from the minimization

\[
\tilde{\mu}_k(x_k) \in \arg \min_{u \in U_k(x_k)} \tilde{Q}_k(x_k, u).
\]

Unfortunately, this method also suffers from the error magnification inherent in the Q-factor differencing operation. This motivates an alternative approach, called differential training, which is based on cost-to-go difference approximations. To this end, we note that to compute the rollout control \( \tilde{\mu}_k(x_k) \), it is sufficient to have the differences of costs-to-go

\[
\tilde{J}_{k+1}(f_k(x_k, u, w_k)) - \tilde{J}_{k+1}(f_k(x_k, \mu_k(x_k), w_k)), \quad (3.29)
\]

where \( \mu_k(x_k) \) is the control applied by the base policy at \( x_k \).

We thus consider a function approximation approach, whereby given any two states \( x_{k+1} \) and \( \tilde{x}_{k+1} \), we obtain an approximation \( \tilde{G}_{k+1}(x_{k+1}, \tilde{x}_{k+1}) \) of the cost difference (3.29). We then compute the rollout control by

\[
\tilde{\mu}_k(x_k) \in \arg \min_{u \in U_k(x_k)} E\left\{g_k(x_k, u, w_k) - g_k(x_k, \mu_k(x_k), w_k) + \tilde{G}_{k+1}(f_k(x_k, u, w_k), f_k(x_k, \mu_k(x_k), w_k))\right\}, \quad (3.30)
\]

where \( \mu_k(x_k) \) is the control applied by the base policy at \( x_k \). Note that the minimization (3.30) aims to simply subtract the approximate Q-factor of the base policy control \( \mu_k(x_k) \) from the approximate Q-factor of every other control \( u \in U_k(x_k) \).

An important point here is that the training of an approximation architecture to obtain \( \tilde{G}_{k+1} \) can be done using any of the standard training
methods, and a “differential” system, whose “states” are pairs \((x_k, \hat{x}_k)\) and will be described shortly. To see this, let us denote for all \(k\) and pair of states \((x_k, \hat{x}_k)\)
\[
G_k(x_k, \hat{x}_k) = J_{k, \pi}(x_k) - J_{k, \pi}(\hat{x}_k)
\]
the cost function differences corresponding to the base policy \(\pi\). We consider the DP equations corresponding to \(\pi\), and to \(x_k\) and \(\hat{x}_k\):
\[
J_{k, \pi}(x_k) = E\{g_k(x_k, \mu_k(x), w_k) + J_{k+1, \pi}(f_k(x_k, \mu_k(x), w_k))\},
\]
\[
J_{k, \pi}(\hat{x}_k) = E\{g_k(\hat{x}_k, \mu_k(\hat{x}_k), w_k) + J_{k+1, \pi}(f_k(\hat{x}_k, \mu_k(\hat{x}_k), w_k))\},
\]
and we subtract these equations to obtain
\[
G_k(x_k, \hat{x}_k) = E\{g_k(x_k, \mu_k(x_k), w_k) - g(\hat{x}_k, \mu_k(\hat{x}_k), w_k)
+ G_{k+1}(f_k(x_k, \mu_k(x_k), w_k), f_k(\hat{x}_k, \mu_k(\hat{x}_k), w_k))\},
\]
for all \((x_k, \hat{x}_k)\) and \(k\). Therefore, \(G_k\) can be viewed as the cost-to-go function for a problem involving a fixed policy (the base policy), the state \((x_k, \hat{x}_k)\), the cost per stage
\[
g_k(x_k, \mu_k(x_k), w_k) - g(\hat{x}_k, \mu_k(\hat{x}_k), w_k), \quad (3.31)
\]
and the system equation
\[
(x_{k+1}, \hat{x}_{k+1}) = \left(f_k(x_k, \mu_k(x_k), w_k), f_k(\hat{x}_k, \mu_k(\hat{x}_k), w_k)\right). \quad (3.32)
\]

Thus, it can be seen that any of the standard methods that can be used to train architectures that approximate \(J_{k, \pi}\), can also be used for training architectures that approximate \(G_k\). For example, one may use simulation-based methods that generate pairs of trajectories starting at the pair of initial states \((x_k, \hat{x}_k)\), and generated according to Eq. (3.32) by using the base policy \(\pi\). Note that a single random sequence \(\{w_0, \ldots, w_{N-1}\}\) may be used to simultaneously generate samples of \(G_k(x_k, \hat{x}_k)\) for several triples \((x_k, \hat{x}_k, k)\), and in fact this may have a substantial beneficial effect.

A special case of interest arises when a linear, feature-based architecture is used for the approximator \(\tilde{G}_k\). In particular, let \(\phi_k\) be a feature extraction mapping that associates a feature vector \(\phi_k(x_k)\) with state \(x_k\) and time \(k\), and let \(\tilde{G}_k\) be of the form
\[
\tilde{G}_k(x_k, \hat{x}_k) = r_k^t(\phi_k(x_k) - \phi_k(\hat{x}_k)),
\]
where \(r_k\) is a tunable weight vector of the same dimension as \(\phi_k(x_k)\) and prime denotes transposition. The rollout policy is generated by
\[
\tilde{\mu}_k(x_k) \in \arg\min_{u \in U_k(x_k)} E\{g_k(x_k, u, w_k) + r_{k+1}^t(\phi_k(x_k) - \phi_k(\hat{x}_k))\},
\]
which corresponds to using $r'_{k+1} \phi_{k+1}(x_{k+1})$ (plus an unknown inconsequential constant) as an approximation to $J_{k+1}(x_{k+1})$. Thus, in this approach, we essentially use a linear feature-based architecture to approximate the cost functions $J_{k, \pi}(x_{k+1})$. This is done by selecting pairs of initial states, running in parallel the corresponding trajectories using the base policy, and subtracting the resulting trajectory costs from each other.

### 3.4 Training of Policies in Approximate DP

We have focused so far on approximation in value space using parametric architectures. In this section we will discuss briefly how the cost function approximation methods of this chapter can be suitably adapted for the purpose of approximation in policy space, whereby we select the policy by using optimization over a parametric family of some form.

In particular, suppose that for a given stage $k$, we have access to a dataset of sample state-control pairs $(x^s_k, u^s_k)$, $s = 1, \ldots, q$, obtained through some unspecified process, such as rollout or problem approximation. We may then wish to “learn” this process by training the parameter vector $r_k$ of a parametric family of policies $\mu_k(x_k, r_k)$, using least squares minimization/regression:

\[
\tau_k \in \arg \min_{r_k} \sum_{s=1}^{q} \| u^s_k - \tilde{\mu}_k(x^s_k, r_k) \|^2;
\]  

(3.33)

cf. our discussion of approximation in policy space in Section 1.3.3.

#### 3.4.1 The Use of Classifiers for Approximation in Policy Space

As we have noted in Section 3.1, in the case of a continuous control space, training of a parametric architecture for policy approximation is similar to training for a cost approximation. In the case where the control space is finite, however, it is useful to make the connection of approximation in policy space with classification; cf. Fig. 3.1.2 and the discussion of Section 3.1.

Classification is an important subject in machine learning. The objective is to construct an algorithm, called a classifier, which assigns a given “object” to one of a finite number of “categories” based on its “characteristics.” Here we use the term “object” generically. In some cases, the classification may relate to persons or situations. In other cases, an object may represent a hypothesis, and the problem is to decide which of the hypotheses is true, based on some data. In the context of approximation in policy space, objects correspond to states, and categories correspond to
controls to be applied at the different states. Thus in this case, we view each sample \((x_k^s, u_k^s)\) as an object-category pair.

Generally, in (multiclass) classification we assume that we have a population of objects, each belonging to one of \(m\) categories \(c = 1, \ldots, m\). We want to be able to assign a category to any object that is presented to us. Mathematically, we represent an object with a vector \(x\) (e.g., some raw description or a vector of features of the object), and we aim to construct a rule that assigns to every possible object \(x\) a unique category \(c\).

To illustrate a popular classification method, let us assume that if we draw an object \(x\) at random from this population, the conditional probability of the object being of category \(c\) is \(p(c \mid x)\). If we know the probabilities \(p(c \mid x)\), we can use a classical statistical approach, whereby we assign \(x\) to the category \(c^*(x)\) that has maximal posterior probability, i.e.,

\[
c^*(x) \in \arg \max_{c=1, \ldots, m} p(c \mid x).
\]

(3.34)

This is called the Maximum a Posteriori rule (or MAP rule for short; see for example the book [BeT08], Section 8.2, for a discussion).

When the probabilities \(p(c \mid x)\) are unknown, we may try to estimate them using a least squares optimization, based on the following property, whose proof is outlined in Exercise 4.1; see also [Ber19a], Section 3.5.

**Proposition 3.4.1: (Least Squares Property of Conditional Probabilities)** Let \(\xi(x)\) be any prior distribution of \(x\), so that the joint distribution of \((c, x)\) is

\[
\zeta(c, x) = \sum_x \xi(x) \sum_{c=1}^m p(c \mid x).
\]

Let \(z(c, x)\) be the function of \((c, x)\) defined by

\[
z(c, x) = \begin{cases} 
1 & \text{if } x \text{ is of category } c, \\
0 & \text{otherwise}.
\end{cases}
\]

For any function \(h(c, x)\) of \((c, x)\), consider

\[
E\left\{ (z(c, x) - h(c, x))^2 \right\},
\]

the expected value with respect to the distribution \(\zeta(c, x)\) of the random variable \((z(c, x) - h(c, x))^2\). Then \(p(c \mid x)\) minimizes this expected value over all functions \(h(c, x)\), i.e., for all functions \(h\), we have
The proposition states that \( p(c \mid x) \) is the function of \((c, x)\) that minimizes
\[
E \left\{ (z(c, x) - p(c \mid x))^2 \right\} \leq E \left\{ (z(c, x) - h(c, x))^2 \right\}.
\] over all functions \( h \) of \((c, x)\), independently of the prior distribution of \( x \). This suggests that we can obtain approximations to the probabilities \( p(c \mid x), c = 1, \ldots, m \), by minimizing an empirical/simulation based approximation of the expected value (3.36).

More specifically, let us assume that we have a training set consisting of \( q \) object-category pairs \((x^s, c^s), s = 1, \ldots, q\), and corresponding vectors
\[
z^s(c) = \begin{cases} 1 & \text{if } c^s = c, \\ 0 & \text{otherwise}, \end{cases} \quad c = 1, \ldots, m,
\]
and adopt a parametric approach. In particular, for each category \( c = 1, \ldots, m \), we approximate the probability \( p(c \mid x) \) with a function \( \tilde{h}(c, x, r) \) that is parametrized by a vector \( r \), and optimize over \( r \) the empirical approximation to the expected squared error of Eq. (3.36). Thus we can obtain \( r \) by the least squares regression:
\[
\bar{r} \in \arg \min_r \sum_{s=1}^q \sum_{c=1}^m \left( z^s(c) - \tilde{h}(c, x^s, r) \right)^2,
\] perhaps with some quadratic regularization added. The functions \( \tilde{h}(c, x, r) \) may be provided for example by a feature-based architecture or a neural network.

Note that each training pair \((x^s, c^s)\) is used to generate \( m \) examples for use in the regression problem (3.37): \( m - 1 \) “negative” examples of the form \((x^s, 0)\), corresponding to the \( m - 1 \) categories \( c \neq c^s \), and one “positive” example of the form \((x^s, 1)\), corresponding to \( c = c^s \). Note also that the incremental gradient method can be applied to the solution of this problem.

The regression problem (3.37) approximates the minimization of the expected value (3.36), so we conclude that its solution \( \tilde{h}(c, x, \bar{r}) \), \( c = 1, \ldots, m \), approximates the probabilities \( p(c \mid x) \). Once this solution is obtained, we may use it to classify a new object \( x \) according to the rule
\[
\text{Estimated Object Category} = \tilde{c}(x, \bar{r}) \in \arg \max_{c=1,\ldots,m} \tilde{h}(c, x, \bar{r}),
\] which approximates the MAP rule (3.34); cf. Fig. 3.4.1.
Learning Values and Policies Chap. 3

Figure 3.4.1 Illustration of the MAP classifier \( c^*(x) \) for the case where the probabilities \( p(c \mid x) \) are known [cf. Eq. (3.34)], and its data-trained version \( \hat{c}(x, \bar{r}) \) [cf. Eq. (3.38)]. The classifier may be obtained by using the data set \( (x_s^k, u_s^k) \), \( s = 1, \ldots, q \), and an approximation architecture such as a feature-based architecture or a neural network.

Figure 3.4.2 Illustration of classification-based approximation in policy space. The classifier, defined by the parameter \( r_k \), is constructed by using the training set \( (x_s^k, u_s^k) \), \( s = 1, \ldots, q \). It yields a randomized policy that consists of the probability \( \tilde{h}(u, x_k, r_k) \) of using control \( u \in U_k(x_k) \) at state \( x_k \). This policy is approximated by the deterministic policy \( \hat{\mu}_k(x_k, r_k) \) that uses at state \( x_k \) the control that maximizes over \( u \in U_k(x_k) \) the probability \( h(u, x_k, r_k) \) [cf. Eq. (3.38)].

Returning to approximation in policy space, for a given training set \( (x_s^k, u_s^k) \), \( s = 1, \ldots, q \), the classifier just described provides (approximations to) the “probabilities” of using the controls \( u_k \in U_k(x_k) \) at the states \( x_k \), so it yields a “randomized” policy \( \hat{h}(u, x_k, r_k) \) for stage \( k \) [once the values \( \hat{h}(u, x_k, r_k) \) are normalized so that, for any given \( x_k \), they add to 1]; cf. Fig. 3.4.2. In practice, this policy is usually approximated by the deterministic policy \( \hat{\mu}_k(x_k, r_k) \) that uses at state \( x_k \) the control of maximal probability at that state; cf. Eq. (3.38).

For the simpler case of a classification problem with just two categories, say \( A \) and \( B \), a similar formulation is to hypothesize a relation of
the following form between object $x$ and its category:

$$ \text{Object Category} = \begin{cases} A & \text{if } \tilde{h}(x,r) = 1, \\ B & \text{if } \tilde{h}(x,r) = -1, \end{cases} $$

where $\tilde{h}$ is a given function and $r$ is the unknown parameter vector. Given a set of $q$ object-category pairs $(x_1, z_1), \ldots, (x_q, z_q)$ where

$$ z_s = \begin{cases} 1 & \text{if } x \text{ is of category } A, \\ -1 & \text{if } x \text{ is of category } B, \end{cases} $$

we obtain $r$ by the least squares regression:

$$ \tilde{r} \in \arg \min_r \sum_{s=1}^{q} (z_s - \tilde{h}(x_s,r))^2. $$

The optimal parameter vector $\tilde{r}$ is used to classify a new object with data vector $x$ according to the rule

$$ \text{Estimated Object Category} = \begin{cases} A & \text{if } \tilde{h}(x,\tilde{r}) > 0, \\ B & \text{if } \tilde{h}(x,\tilde{r}) < 0. \end{cases} $$

In the context of DP and approximation in policy space, this classifier may be used, among others, in stopping problems where there are just two controls available at each state: stopping (i.e., moving to a termination state) and continuing (i.e., moving to some nontermination state).

There are several variations of the preceding classification schemes, for which we refer to the specialized literature. Moreover, there are several commercially and publicly available software packages for solving the associated regression problems and their variants. They can be brought to bear on the problem of parametric approximation in policy space using any training set of state-control pairs, regardless of how it was obtained.

### 3.4.2 Policy Iteration with Value and Policy Networks - Multiprocessor Parallelization

As we have already noted, in contrast to rollout, approximate policy iteration (PI) is fundamentally an off-line training algorithm, because for a large scale problem, it is necessary to represent the successively generated policies with an approximation architecture. Thus approximate PI involves the successive use of a value network to implement policy evaluation, and a policy network to represent policy improvement.

In particular, we can start with a base policy and a terminal cost approximation, and generate state-control samples of the corresponding truncated rollout policy. These samples can in turn be used with the
approximation in policy space scheme of this section to obtain a policy network that approximates the truncated rollout policy; cf. Fig. 3.4.3.

The cost function of the policy network can in turn be approximated with a value network using the methodology that we have discussed in this chapter. The value network can be used in turn as a terminal cost function approximation in a truncated rollout scheme where the previously obtained policy network can be used as a base policy. In this way a perpetual rollout scheme is obtained, which involves a sequence of value and policy networks.

Parallelization and distributed computation can be used in several different ways in such a scheme, including Q-factor, Monte Carlo, and multiagent parallelization. Moreover, when feature-based partitioning of the state space is used (cf. Example 3.1.8), we may consider a multiprocessor parallelization scheme, which involves multiple local value and policy networks, one per subset of the state space partition; see Fig. 3.4.4.

Let us finally note that multiprocessor parallelization leads to the idea of an approximation architecture that involves a graph. Each node of the graph consists of a neural network and each arc connecting a pair of nodes corresponds to data transfer between the corresponding neural networks. The question of how to train such an architecture is quite complex and one may think of several alternative possibilities. For example the training may be collaborative with the exchange of training results and/or training data communicated periodically or asynchronously; see the book [Ber20a], Section 5.8.
Each Set Has a Local Value Network and a Local Policy Network

Figure 3.4.4 Illustration of a perpetual truncated rollout scheme with a partitioned architecture. A local value network and a local policy network are used for each subset of the partition. The policy network is used as the base policy and the value network is used to provide a terminal cost function approximation.

State-control training pairs for the corresponding rollout policy are obtained by starting at an initial state within some subset of the partition, generating rollout trajectories using the local policy network, which are truncated once the state enters a different subset of the partition, with the corresponding terminal cost function approximation supplied by the value network of that subset.

When a separate processor is used for each subset of partition, the corresponding value networks are communicated between processors. This can be done asynchronously, with each processor sharing its value network as soon it becomes available. In a variation of this scheme, the local policy networks may also be shared selectively among processors for selective use in the truncated rollout process.

3.4.3 Why Use On-Line Play and not Just Train a Policy Network to Emulate the Lookahead Minimization?

This is a sensible and common question, which stems from the mindset that neural networks have extraordinary function approximation properties. In other words, why go through the arduous on-line process of lookahead minimization, if we can do the same thing off-line and represent the lookahead policy with a trained policy network? In particular, we can select the policy from a suitably restricted class of policies, such as a parametric class of the form $\mu(x, r)$, where $r$ is a parameter vector. We may then estimate $r$ using some type of off-line training process. Then the on-line computation of controls $\mu(x, r)$ can be much faster compared with on-line lookahead minimization.
On the negative side, because parametrized approximations often involve substantial calculations, they are not well suited for on-line replanning. From our point of view in these notes, there is another important reason why approximation in value space is needed on top of approximation in policy space: the off-line trained policy may not perform nearly as well as the corresponding one-step or multistep lookahead/rollout policy, because it lacks the extra power of the associated exact Newton step (cf. our discussion of AlphaZero and TD-Gammon in Section 1.1, and linear quadratic problems in Section 1.5).

Figure 3.4.5 illustrates this fact with a one-dimensional linear-quadratic example, and compares the performance of a linear policy with its corresponding one-step lookahead policy. In this example the system equation is

$$x_{k+1} = x_k + 2u_k,$$

and the quadratic cost function parameters are $q = 1$, $r = 0.5$. The optimal policy for this system and cost parameter values is

$$\mu^*(x) = L^*x,$$

with $L^* \approx -0.4$, and the optimal cost function is

$$J^*(x) = K^*x^2,$$

where $K^* \approx 1.1$. We want to to explore what happens when we use a policy of the form

$$\mu_L(x) = Lx,$$
where \( L \neq L^* \) (which may be optimal for another system equation or cost function parameters). The cost function of \( \mu_L \) has the form
\[
J_{\mu}(x) = K_L x^2,
\]
where \( K_L \) is obtained by using the formulas given in Section 1.5. The figure shows the quadratic cost coefficient differences \( K_L - K^* \) and \( \tilde{K}_L - K^* \) as a function of \( L \), where \( K_L \) and \( \tilde{K}_L \) are the quadratic cost coefficients of \( \mu \) (without one-step lookahead/Newton step) and the corresponding one-step lookahead policy \( \tilde{\mu} \) (with one-step lookahead/Newton step).

### 3.5 AGGREGATION

In this section we consider approximation in value space using a problem approximation approach that is based on aggregation. In particular, we construct a simpler and more tractable “aggregate” problem by creating special subsets of states, which we view as “aggregate states.” We then solve the aggregate problem exactly by DP. This is the off-line training part of the aggregation approach, and it may be carried out with a variety of DP methods, including simulation-based value and policy iteration; we refer to the RL book [Ber19a] for a detailed account. Finally, we use the optimal cost-to-go function of the aggregate problem to construct a terminal cost approximation in a one-step or multistep lookahead approximation scheme for the original problem. Additionally, we may also use the optimal policy of the aggregate problem to construct a base policy for a truncated rollout scheme.

In addition to problem approximation, aggregation is related to feature-based parametric approximation. In particular, it often produces a piecewise constant cost function approximation, which may be viewed as a linear feature-based parametrization, where the features are 0-1 membership functions; see Example 3.1.1. Aggregation can also be combined with other approximation schemes, to add a local correction to a cost function approximation \( \tilde{J} \), which is already available, possibly through the use of a neural network; see the discussion of biased aggregation later in Section 3.5.7.

Aggregation can be applied to both finite horizon and infinite horizon problems. In this section, we will focus primarily on the discounted infinite horizon problem. We will introduce aggregation in a simple intuitive form in Section 3.5.1, and generalize later to a more sophisticated form of feature-based aggregation, which we also discussed briefly in Example 3.1.7.

#### 3.5.1 Aggregation with Representative States

In this section we focus on a relatively simple form of aggregation, which involves a special subset of states, called *representative*. Our approach is to
Learning Values and Policies Chapter 3

States $i$ (fine grid)

Representative states $x$ (coarse grid)

Figure 3.5.1 Illustration of aggregation with representative states; cf. Example 3.5.1. A relatively small number of states are viewed as representative. We define transition probabilities between pairs of aggregate states and we also define the associated expected transition costs. These specify a smaller DP problem, called the aggregate problem, which is solved exactly. The optimal cost function $J^*$ of the original problem is approximated by interpolation from the optimal costs of the representative states $r_y^*$ in the aggregate problem:

$$\tilde{J}(j) = \sum_{y \in A} \phi_{jy} r_y^*, \quad j = 1, \ldots, n,$$

and is used in a one-step or multistep lookahead scheme.

We view these states as the states of a smaller optimal control problem, the aggregate problem, which we will formulate and solve exactly in place of the original. We will then use the optimal aggregate costs of the representative states to approximate the optimal costs of the original problem states by interpolation. In this chapter, whenever we consider a finite-state problem, we use notation that is more convenient for such a problem. In particular, states and successor states will be denoted by $i$ and $j$, respectively, and the system equation is represented by control-dependent transition probabilities $p_{ij}(u)$; cf. Section 1.4.1. Let us describe a classical example.

Example 3.5.1 (Coarse Grid Approximation)

Consider a discounted problem where the state space is a grid of points $i = 1, \ldots, n$ on the plane. We introduce a coarser grid that consists of a subset $A$ of the states/points, which we call representative and denote by $x$; see Fig. 3.5.1. We now wish to formulate a lower-dimensional DP problem just on the coarse grid of states. The difficulty here is that there may be positive transition probabilities $p_{ij}(u)$ from some representative states $x$ to some non-representative states $j$. To deal with this difficulty, we introduce artificial transition probabilities $\phi_{jy}$ from non-representative states $j$ to representative states $y$, which we call aggregation probabilities. In particular, a transition
from representative state $x$ to a nonrepresentative state $j$, is followed by a transition from $j$ to some other representative state $y$ with probability $\phi_{jy}$; see Fig. 3.5.2.

This process involves approximation but constructs a transition mechanism for an aggregate problem whose states are just the representative ones. The transition probabilities between representative states $x, y$ under control $u \in U(x)$ and the corresponding expected transition costs are

\[
\hat{p}_{xy}(u) = \sum_{j=1}^{n} p_{xj}(u)\phi_{jy}, \quad \hat{g}(x, u) = \sum_{j=1}^{n} p_{xj}(u)g(x, u, j). \tag{3.39}
\]

We can solve the aggregate problem by any suitable exact DP method. Let $A$ denote the set of representative states and let $r^*_x$ denote the corresponding optimal cost of representative state $x$. We can then approximate the optimal cost function of the original problem with the interpolation formula

\[
\tilde{J}(j) = \sum_{y \in A} \phi_{jy}r^*_y, \quad j = 1, \ldots, n. \tag{3.40}
\]

This function may in turn be used in a one-step or multistep lookahead scheme for approximation in value space of the original problem.

Note that there is a lot of freedom in selecting the aggregation probabilities $\phi_{jy}$. Intuitively, $\phi_{jy}$ should express a measure of proximity between $j$ and $y$, e.g., $\phi_{jy}$ should be relatively large when $y$ is geometrically close to $j$. For example, we could set $\phi_{yj} = 1$ for the representative state $y_j$ that is “closest” to $j$, and $\phi_{jy_j} = 0$ for all other representative states $y \neq y_j$. In this case, Eq. (3.40) yields a piecewise constant cost function approximation \(\tilde{J}\) (the constant values are the scalars $r^*_y$ of the representative states $y$).

We will now formalize our framework for aggregation with representative states by generalizing the preceding example; see Fig. 3.5.3. We first consider the $n$-state version of the $\alpha$-discounted problem of Section 1.4.1. We refer to this problem as the “original problem,” to distinguish from the “aggregate problem,” which we define next.

**Aggregation Framework with Representative States**

We introduce a finite subset $\mathcal{A}$ of the original system states, which we call *representative states*, and we denote them by symbols such as $x$ and $y$. We construct an aggregate problem, with state space $\mathcal{A}$, and transition probabilities and transition costs defined as follows:

(a) We relate the original system states $j$ to representative states $y \in \mathcal{A}$ with aggregation probabilities $\phi_{jy}$; these are scalar “weights” satisfying $\phi_{jy} \geq 0$ for all $y \in \mathcal{A}$, and $\sum_{y \in \mathcal{A}} \phi_{jy} = 1$.

(b) We define the transition probabilities between representative states $x$ and $y$ under control $u \in U(x)$ by
Representative States

Original State Space

Aggregation Probabilities

Original States to Representative States

Figure 3.5.2 Illustration of the use of aggregation probabilities \( \phi_{jy} \) from non-representative states \( j \) to representative states \( y \) in Example 3.5.1. A transition from a state \( x \) to a nonrepresentative state \( j \) is followed by a transition to aggregate state \( y \) with probability \( \phi_{jy} \). In this figure, from representative state \( x \), there are three possible transitions, to states \( j_1 \), \( j_2 \), and \( j_3 \), according to \( p_{xj_1}(u) \), \( p_{xj_2}(u) \), and \( p_{xj_3}(u) \), and each of these states is associated with a convex combination of representative states using the aggregation probabilities. For example, the state \( j_1 \) is associated with

\[
\phi_{j_1y_1} y_1 + \phi_{j_1y_2} y_2 + \phi_{j_1y_3} y_3.
\]

\[
\hat{p}_{xy}(u) = \sum_{j=1}^{n} p_{xj}(u) \phi_{jy}.
\]  

(3.41)

(c) We define the expected transition costs at representative states \( x \) under control \( u \in U(x) \) by

\[
\hat{g}(x, u) = \sum_{j=1}^{n} p_{xj}(u) g(x, u, j).
\]  

(3.42)

The optimal costs of the representative states \( y \in A \) in the aggregate problem are denoted by \( r^*_y \), and they define approximate costs for the original problem through the interpolation formula

\[
\hat{J}(j) = \sum_{y \in \mathcal{A}} \phi_{jy} r^*_y, \quad j = 1, \ldots, n.
\]  

(3.43)

Aside from the selection of representative states, an important consideration is the choice of the aggregation probabilities. These probabilities express “similarity” or “proximity” of original to representative states (as in the case of the coarse grid Example 3.5.1), but in principle they can be
Sec. 3.5 Aggregation

\[ p_{ij}(u), g(i, u, j) \]

\( \hat{p}_{xy}(u) = \sum_{j=1}^{n} p_{xj}(u) \phi_{jy} \)

\( \hat{g}(x, u) = \sum_{j=1}^{n} p_{xj}(u) g(x, u, j) \)

\[ \hat{J}(j) = \sum_{y \in A} \phi_{jy} r^*_y, \quad j = 1, \ldots, n, \]

One-step Lookahead with

\[ \hat{J}(j) = \sum_{y \in A} \phi_{jy} r^*_y \]

Original States

\( \phi_{jy} \) may be interpreted as some measure of “strength of relation” of \( j \) to \( y \). The vectors \( \{ \phi_{jy} \mid j = 1, \ldots, n \} \) may also be viewed as basis functions for a linear cost function approximation via Eq. (3.43).

Hard Aggregation and Error Bound

A special case of interest, called hard aggregation, is when for every state \( j \), we have \( \phi_{jy} = 0 \) for all representative states \( y \), except a single one, denoted \( y_j \), for which we have \( \phi_{jy_j} = 1 \). In this case, the one-step lookahead approximation

\[ \hat{J}(j) = \sum_{y \in A} \phi_{jy} r^*_y, \quad j = 1, \ldots, n, \]

is piecewise constant; it is constant and equal to \( r^*_y \) for all \( j \) in the set

\[ S_y = \{ j \mid \phi_{jy} = 1 \}, \quad y \in A, \]
$\tilde{J}(j) = \sum_{y \in A} \phi_{jy} r^*_y$

in the hard aggregation case where we have $\phi_{jy} = 0$ for all representative states $y$, except a single one. Here $\tilde{J}$ is constant and equal to $r^*_y$ for all $j$ in the footprint set

$S_y = \{ j \mid \phi_{jy} = 1 \}, \quad y \in A.$

called the \textit{footprint} of representative state $y$; see Fig. 3.5.4. Moreover the footprints of all the representative states are disjoint and form a partition of the state space, i.e.,

$\cup_{x \in A} S_x = \{1, \ldots, n\}.$

The footprint sets can be used to define a bound for the error $(J^* - \tilde{J})$. In particular, it can be shown that

$$|J^*(j) - \tilde{J}(j)| \leq \epsilon \frac{1}{1 - \alpha}, \quad j = 1, \ldots, n,$$

where

$$\epsilon = \max_{y \in A} \max_{i,j \in S_y} |J^*(i) - J^*(j)|$$

is the \textit{maximum variation of $J^*$ within the footprint sets $S_y$}. This error bound result can be extended to the more general aggregation framework that will be given in the next section. Note the primary intuition derived
from this bound: the error due to hard aggregation is small if $J^*$ varies little within each $S_y$.

For a special hard aggregation case of interest, consider the geometrical context of Example 3.5.1. There, aggregation probabilities are often based on a nearest neighbor approximation scheme, whereby each non-representative state $j$ takes the cost value of the “closest” representative state $y$, i.e.,

$$\phi_{jy} = 1 \quad \text{if } y_j \text{ is the closest representative state to } j.$$ 

Then all states $j$ for which a given representative state $y$ is the closest to $j$ (the footprint of $y$) are assigned equal approximate cost $\tilde{J}(j) = r_{y}^*$. 

**Methods for Solving the Aggregate Problem**

The most straightforward way to solve the aggregate problem is to compute the aggregate problem transition probabilities $\hat{p}_{xy}(u)$ [cf. Eq. (3.41)] and transition costs $\tilde{g}(x, u)$ [cf. Eq. (3.42)] by either an algebraic calculation or by simulation. The aggregate problem may then be solved by any one of the standard methods, such as VI or PI. This exact calculation is plausible if the number of representative states is relatively small. An alternative possibility is to use a simulation-based VI or PI method. We refer to a discussion of these methods in the author’s books [Ber12], Section 6.5, and [Ber19a], Section 6.3. The idea is that a simulator for the original problem can be used to construct a simulator for the aggregate problem; cf. Fig. 3.5.3.

An important observation is that if the original problem is deterministic and hard aggregation is used, the aggregate problem is also deterministic, and can be solved by shortest-path like methods. This is true for both discounted problems and for undiscounted shortest path-type problems. In the latter case, the termination state of the original problem must be included as a representative state in the aggregate problem. However, if hard aggregation is not used, the aggregate problem will be stochastic, because of the introduction of the aggregation probabilities. Of course, once the aggregate problem is solved and the lookahead approximation $\tilde{J}$ is obtained, a deterministic structure in the original problem can be exploited to facilitate the lookahead minimizations.

### 3.5.2 Continuous Control Space Discretization

Aggregation with representative states extends without difficulty to problems with a continuous state space, as long as the control space is finite. Then once the representative states and the aggregation probabilities have been defined, the corresponding aggregate problem is a discounted problem with finite state and control spaces, which can be solved with the standard
methods. The only potential difficulty arises when the disturbance space is also infinite, in which case the calculation of the transition probabilities and expected stage costs of the aggregate problem must be obtained by some form of integration process.

The case where both the state and the control spaces are continuous is somewhat more complicated, because both of these spaces must be discretized using representative state-control pairs, instead of just representative states. The following example illustrates what may happen if we use representative state discretization only.

**Example 3.5.2 (Continuous Shortest Path Discretization)**

Suppose that we want to find the fastest route for a car to travel between two points A and B located at the opposite ends of a square with side 1000 meters, while avoiding some known obstacles. We assume a constant car speed of 1 meter per second and that the car can drive in any direction; cf. Fig. 3.5.5.

Let us consider discretizing the space with a square grid (a set of representative states), and restrict the directions of motion to horizontal and vertical, so that at each stage the car moves from a grid point to one of the four closest grid points. Thus in the discretized version of the problem the car travels with a sequence of horizontal and vertical moves as indicated in the right side of Fig. 3.5.5. Is it possible to approximate the fastest route arbitrarily closely with the optimal solution of the discretized problem, assuming a sufficiently fine grid?

The answer is no! To see this note that in the discretized problem the optimal travel time is 2000 secs, regardless of how fine the discretization is. On the other hand, in the continuous space/nondiscretized problem the optimal travel time can be as little as $\sqrt{2} \times 1000$ secs (this corresponds to the favorable case where the straight line from A to B does not meet an obstacle).
Sec. 3.5 Aggregation

The difficulty in the preceding example is that the state space is discretized finely but the control space is not. What is needed is to introduce a fine discretization of the control space as well, through some set of “representative controls.” We can deal with this situation with a suitable form of discretized aggregate problem, which when solved provides an appropriate form of cost function approximation for use with one-step lookahead. The discretized problem is a stochastic infinite horizon problem, even if the original problem is deterministic. Further discussion of this approach is outside our scope, and we refer to the sources cited at the end of the chapter. Under reasonable assumptions it is possible to show consistency, i.e., that the optimal cost function of the discretized problem converges to the optimal cost function of the original continuous spaces problem as the discretization of both the state and the control spaces becomes increasingly fine.

The type of difficulty illustrated in Example 3.5.2 does not arise if the state space is continuous but the control space is finite. In particular, this is true in partially observed finite spaces Markov decision problems (POMDP), which are defined over their belief space (the space of probability distributions over their states). We briefly discuss this case next.

3.5.3 Continuous State Space - POMDP Discretization

Let us consider any $\alpha$-discounted DP problem, where the state space is a bounded convex subset $B$ of a Euclidean space, such as the unit simplex, but the control space $U$ is finite. We use $b$ to denote the states, to emphasize the connection with belief states in POMDP and to distinguish them from $x$, which we will use to denote representative states. Bellman’s equation is $J = TJ$ with the Bellman operator $T$ defined by

$$(TJ)(b) = \min_{u \in U} E \{ g(b, u, w) + \alpha J(f(b, u, w)) \}, \quad b \in B.$$ 

We introduce a set of representative states $\{x_1, \ldots, x_m\} \subset B$. We assume that the convex hull of $\{x_1, \ldots, x_m\}$ is equal to $B$, so each state $b \in B$ can be expressed as

$$b = \sum_{i=1}^{m} \phi_{bx_i} x_i,$$

where $\{\phi_{bx_i} | i = 1, \ldots, m\}$ is a probability distribution:

$$\phi_{bx_i} \geq 0, \quad i = 1, \ldots, m, \quad \sum_{i=1}^{m} \phi_{bx_i} = 1, \quad \text{for all } b \in B.$$

We view $\phi_{bx_i}$ as aggregation probabilities.
Consider the operator $\hat{T}$ that transforms a vector $r = (r_{x_1}, \ldots, r_{x_m})$ into the vector $\hat{T}r$ with components $(\hat{T}r)(x_1), \ldots, (\hat{T}r)(x_m)$ defined by

$$(\hat{T}r)(x_i) = \min_{u \in U} \mathbb{E}_w \left\{ g(x_i, u, w) + \alpha \sum_{j=1}^{m} \phi_{f(x_i, u, w) x_j} r_{x_j} \right\}, \quad i = 1, \ldots, m,$$

where $\phi_{f(x_i, u, w) x_j}$ are the aggregation probabilities of the state $f(x_i, u, w)$. It can then be shown that $\hat{T}$ is a contraction mapping with respect to the maximum norm (we give the proof for a similar result in the next section). Bellman’s equation for an aggregate finite-state discounted DP problem whose states are $x_1, \ldots, x_m$ has the form

$$r_{x_i} = (\hat{T}r)(x_i), \quad i = 1, \ldots, m,$$

and has a unique solution.

The transitions in this problem occur as follows: from state $x_i$ under control $u$, we first move to $f(x_i, u, w)$ at cost $g(x_i, u, w)$, and then we move to a state $x_j$, $j = 1, \ldots, m$, according to the probabilities $\phi_{f(x_i, u, w) x_j}$. The optimal costs $r^*_i$, $i = 1, \ldots, m$, of this problem can often be obtained by standard VI and PI methods that may or may not use simulation. We may then approximate the optimal cost function of the original problem by

$$\tilde{J}(b) = \sum_{i=1}^{m} \phi_{b x_i} r^*_i, \quad \text{for all } b \in B,$$

and reasonably expect that the optimal discretized solution converges to the optimal as the number of representative states increases.

In the case where $B$ is the belief space of an $\alpha$-discounted POMDP, the representative states/beliefs and the aggregation probabilities define an aggregate problem, which is a finite-state $\alpha$-discounted problem with a perfect state information structure. This problem can be solved with exact DP methods if either the aggregate transition probabilities and transition costs can be obtained analytically (in favorable cases) or if the number of representative states is small enough to allow their calculation by simulation. The aggregate problem can also be addressed with approximate DP method that we have discussed earlier, such as problem approximation/certainty equivalence approaches. It can also be addressed with a rollout method, which is suitable for an on-line implementation.

### 3.5.4 General Aggregation

We will now discuss a more general aggregation framework for the infinite horizon $n$-state $\alpha$-discounted problem, by using subsets of states as aggregate states. In particular, we define our general aggregation framework by
essentially replacing the representative states \( x \) with subsets \( I_x \subset \{1, \ldots, n\} \) of the original state space as follows. These subsets are often constructed by using features, however, it is helpful to formulate our aggregation framework in a general form, and introduce features later.

**General Aggregation Framework**

We introduce a finite subset \( \mathcal{A} \) of aggregate states, which we denote by symbols such as \( x \) and \( y \). We define:

(a) A collection of disjoint subsets \( I_x \subset \{1, \ldots, n\} \), \( x \in \mathcal{A} \).

(b) A probability distribution over \( \{1, \ldots, n\} \) for each \( x \in \mathcal{A} \), denoted by \( \{d_{xi} \mid i = 1, \ldots, n\} \), and referred to the *disaggregation probabilities* of \( x \). We require that the distribution corresponding to \( x \) is concentrated on the subset \( I_x \):

\[
d_{xi} = 0, \quad \text{for all} \ i \notin I_x, \ x \in \mathcal{A}. \tag{3.44}
\]

(c) For each original system state \( j \in \{1, \ldots, n\} \), a probability distribution over \( \mathcal{A} \), denoted by \( \{\phi_{jy} \mid y \in \mathcal{A}\} \), and referred to as the *aggregation probabilities* of \( j \). We require that

\[
\phi_{jy} = 1, \quad \text{for all} \ j \in I_y, \ y \in \mathcal{A}. \tag{3.45}
\]

The aggregation and disaggregation probabilities specify a dynamic system involving both aggregate and original system states; cf. Fig. 3.5.6. In this system:

(i) From aggregate state \( x \), we generate an original system state \( i \in I_x \) according to \( d_{xi} \).

(ii) We generate transitions between original system states \( i \) and \( j \) according to \( p_{ij}(u) \), with cost \( g(i, u, j) \).

(iii) From original system state \( j \), we generate aggregate state \( y \) according to \( \phi_{jy} \). [Note that in view of Eq. (3.45), all states \( j \) within a set \( I_y \), are aggregated onto \( y \) with \( \phi_{jy} = 1 \).]

The optimal costs of the aggregate states \( y \in \mathcal{A} \) in the aggregate problem are denoted by \( r^*_y \), and they define approximate costs for the original problem through the interpolation formula

\[
\hat{J}(j) = \sum_{y \in \mathcal{A}} \phi_{jy} r^*_y, \quad j = 1, \ldots, n. \tag{3.46}
\]
Our general aggregation framework is illustrated in Fig. 3.5.6. Note that if each set $I_x$ consists of a single state, we obtain the representative states framework of the preceding section. In this case the disaggregation distribution $\{d_{xi} \mid i \in I_x\}$ is just the atomic distribution that assigns probability 1 to the unique state in $I_x$. Consistent with the special case of representative states, the disaggregation probability $d_{xi}$ may be interpreted as a "measure of the relation of $x$ and $i$."  

The aggregate problem is a DP problem with an enlarged state space that consists of two copies of the original state space $\{1, \ldots, n\}$ plus the set of aggregate states $A$. We introduce the corresponding optimal vectors $\tilde{J}_0$, $\tilde{J}_1$, and $r^* = \{r^*_x \mid x \in A\}$ where:

- $r^*_x$ is the optimal cost-to-go from aggregate state $x$.
- $\tilde{J}_0(i)$ is the optimal cost-to-go from original system state $i$ that has just been generated from an aggregate state (left side of Fig. 3.5.6).
- $\tilde{J}_1(j)$ is the optimal cost-to-go from original system state $j$ that has just been generated from an original system state (right side of Fig. 3.5.6).

Note that because of the intermediate transitions to aggregate states, $\tilde{J}_0$ and $\tilde{J}_1$ are different.

These three vectors satisfy the following three Bellman equations:

\begin{align*}
    r^*_x &= \sum_{i \in I_x} d_{xi} \tilde{J}_0(i), \quad x \in A, \quad (3.47) \\
    \tilde{J}_0(i) &= \min_{u \in U(i)} \sum_{j=1}^n p_{ij}(u) \left( g(i, u, j) + \alpha \tilde{J}_1(j) \right), \quad i = 1, \ldots, n, \quad (3.48)
\end{align*}
Sec. 3.5  Aggregation

\[ \tilde{J}_1(j) = \sum_{y \in \mathcal{A}} \phi_{jy} r_y^*, \quad j = 1, \ldots, n. \]  

(3.49)

The objective is to solve for the optimal costs \( r_y^* \) of the aggregate states in order to obtain approximate costs for the original problem through the interpolation formula

\[ \tilde{J}(j) = \sum_{y \in \mathcal{A}} \phi_{jy} r_y^*, \quad j = 1, \ldots, n; \]

cf. Eq. (3.46).

By combining the three Bellman equations (3.47)-(3.49), we see that \( r_y^* \) satisfies

\[ r_y^* = \sum_{i \in I_x} d_{xi} \min_{u \in U(i)} \sum_{j=1}^n p_{ij}(u) \left( g(i, u, j) + \alpha \sum_{y \in \mathcal{A}} \phi_{jy} r_y^* \right), \quad x \in \mathcal{A}, \]  

(3.50)

or equivalently \( r^* = Hr^* \), where \( H \) is the operator that maps the vector \( r \) to the vector \( Hr \) with components

\[ (Hr)(x) = \sum_{i \in I_x} d_{xi} \min_{u \in U(i)} \sum_{j=1}^n p_{ij}(u) \left( g(i, u, j) + \alpha \sum_{y \in \mathcal{A}} \phi_{jy} r_y^* \right), \quad x \in \mathcal{A}. \]  

(3.51)

It can be shown that \( H \) is a contraction mapping with respect to the maximum norm, and thus the composite Bellman equation (3.50) has \( r^* \) as its unique solution. To see this, we note for any vectors \( r \) and \( r' \), we have

\[ (Hr)(x) = \sum_{i \in I_x} d_{xi} \min_{u \in U(i)} \sum_{j=1}^n p_{ij}(u) \left( g(i, u, j) + \alpha \sum_{y \in \mathcal{A}} \phi_{jy} r_y^* \right) \]

\[ \leq \sum_{i \in I_x} d_{xi} \min_{u \in U(i)} \sum_{j=1}^n p_{ij}(u) \left( g(i, u, j) + \alpha \sum_{y \in \mathcal{A}} \phi_{jy} (r_y^* + \|r - r'\|) \right) \]

\[ = (Hr')(x) + \alpha \|r - r'\|, \]  

where \( \| \cdot \| \) is the maximum norm, and the equality follows from the definition of \( (Hr')(x) \), and the fact that \( d_{xi}, p_{ij}(u), \) and \( \phi_{jy} \) are probabilities. Thus we have

\[ (Hr)(x) - (Hr')(x) \leq \alpha \|r - r'\|, \quad x \in \mathcal{A}. \]

By reversing the roles of \( r \) and \( r' \), we also have

\[ (Hr')(x) - (Hr)(x) \leq \alpha \|r - r'\|, \quad x \in \mathcal{A}, \]
so that
\[ |(H r')(x) - (H r)(x)| \leq \alpha \| r - r' \|, \quad x \in \mathcal{A}. \]

By taking the maximum over \( x \in \mathcal{A} \), it follows that
\[ \| H r - H r' \| \leq \alpha \| r - r' \|, \]

and that \( H \) is a maximum norm contraction.

Note that the composite Bellman equation (3.50) has dimension equal to the number of aggregate states, which is potentially much smaller than \( n \). To apply the aggregation framework of this section, we may solve exactly this equation for the optimal aggregate costs \( r^*_x, x \in \mathcal{A} \), by simulation-based analogs of the VI and PI methods, and obtain a cost function approximation for the original problem through the interpolation formula (3.46). We will develop these methods later, but before doing so, we discuss various ways to formulate the aggregation framework, and in particular, how features can be used for this purpose.

### 3.5.5 Hard Aggregation and Error Bounds

Let us consider the special case of hard aggregation, where for every state \( j \), we have \( \phi_{jy} = 0 \) for all aggregate states \( y \), except a single one, denoted \( y_j \), for which we have \( \phi_{jy_j} = 1 \). In this case, the one-step lookahead approximation
\[ \tilde{J}(j) = \sum_{y \in \mathcal{A}} \phi_{jy} r^*_y, \quad j = 1, \ldots, n, \]
is piecewise constant; it is constant and equal to \( r^*_y \) for all \( j \) in the set
\[ S_y = \{ j \mid \phi_{jy} = 1 \}, \quad y \in \mathcal{A}, \quad (3.52) \]
called the footprint of aggregate state \( y \); see Fig. 3.5.4. Note that the footprints of all the aggregate states are disjoint and form a partition of the state space, i.e.,
\[ \cup_{x \in \mathcal{A}} S_x = \{ 1, \ldots, n \}. \]

We can show the following error bound, due to Tsitsiklis and Van Roy [TsV96]; a generalization of this error bound will be given later in this section.
Proposition 3.5.1: (Error Bound for Hard Aggregation) In the case of hard aggregation, we have

\[ |J^*(j) - \tilde{J}(j)| \leq \frac{\epsilon}{1 - \alpha}, \quad \text{for all } j \text{ such that } j \in S_y, \ y \in \mathcal{A}, \]

where \( \epsilon \) is the maximum variation of the optimal cost function \( J^* \) over the footprint sets \( S_y, \ y \in \mathcal{A} \):

\[ \epsilon = \max_{y \in \mathcal{A}} \max_{i,j \in S_y} |J^*(i) - J^*(j)|. \]

The meaning of the preceding proposition is that if the optimal cost function \( J^* \) varies by at most \( \epsilon \) within each set \( S_y \), the hard aggregation scheme yields a piecewise constant approximation to the optimal cost function that is within \( \epsilon/(1 - \alpha) \) of the optimal.

Aside from its intuitive nature and error bound properties, hard aggregation provides a connection with another major approach for approximation in value space, the so-called the projected equation approach, which we have not discussed here; see the books [Ber12] and [Ber19a]. In particular, it can be shown that for a given policy, the corresponding composite Bellman equation (3.50) for approximate evaluation of \( \mu \) can be viewed as a projected equation, where a projection seminorm is used that is defined by the disaggregation probabilities; see the paper by Yu and Bertsekas [YuB12] (Section 5.5), or the book [Ber12] (Exercise 6.10).

Selecting the Aggregate States

Generally, the method to select the aggregate states is an important issue, for which there is no mathematical theory at present. However, in practical problems, based on intuition and problem-specific knowledge, there are usually evident choices, which may be fine-tuned by experimentation. For example, suppose that the optimal cost function \( J^* \) is piecewise constant over a partition \( \{S_y \mid y \in \mathcal{A}\} \) of the state space \( \{1, \ldots, n\} \). By this we mean that for some vector \( r^* = \{r^*_y \mid y \in \mathcal{A}\} \), we have

\[ J^*(j) = r^*_y \quad \text{for all } j \in S_y, \ y \in \mathcal{A}. \]

Then from Prop. 3.5.1 it follows that the hard aggregation scheme with \( I_x = S_x \) for all \( x \in \mathcal{A} \) is exact, so \( r^*_y \) are the optimal costs of the aggregate states \( x \) in the aggregate problem. This suggests that in hard aggregation,
the states in the footprint set $S_y$ corresponding to an aggregate state $y$ should have roughly equal optimal cost, consistently with the error bound of Prop. 3.5.1.

As an extension of the preceding argument, suppose that through some special insight into the problem’s structure or some preliminary calculation, we know some features of the system’s state that can “predict well” its optimal cost when combined through some approximation architecture, e.g., one that is linear. Then it seems reasonable to form the set aggregate states $A$ of a hard aggregation scheme so that the sets $I_y$ and $S_y$ consist of states with “similar features” for every $y \in A$. This is called feature-based aggregation, and was suggested in the neuro-dynamic book [BeT96], Section 3.1.2. The next section considers this possibility, and provides a way to introduce features and nonlinearities into the aggregation architecture, without compromising its other favorable aspects.

### 3.5.6 Aggregation Using Features

Let us consider the guideline for hard aggregation that we just discussed: states $i$ that belong to the same footprint set $S_y$ should have nearly equal optimal costs, i.e.,

$$\max_{i,j \in S_y} |J^*(i) - J^*(j)| \approx 0, \quad \text{for all } y \in A.$$  

The question now is how to select the sets $S_y$ according to this guideline.

An idea that comes to mind is to use a feature mapping, i.e., a function $F$ that maps a state $i$ into an $m$-dimensional feature vector $F(i)$; cf. Example 3.1.7. In particular, suppose that $F$ has the property that states $i$ with nearly equal feature vector have nearly equal optimal cost $J^*(i)$. Then we can form the sets $S_y$ by grouping together states with nearly equal feature vector. In particular, given $F$, we introduce a more or less regular partition of the feature space [the subset of $\mathbb{R}^m$ that consists of all possible feature vectors $F(i)$]. The partition of the feature space induces a possibly irregular collection of subsets of the original state space. Each of these subsets is then used as the footprint of a distinct aggregate state; see Fig. 3.5.7.

Note that in the resulting aggregation scheme the number of aggregate states may become very large. On the other hand, there is a significant advantage over the linear feature-based architectures of Section 3.1, which assign a single weight to each feature: in feature-based hard aggregation we are assigning a weight to each subset of the feature space partition (possibly a weight to every possible feature value, in the extreme case where each feature value is viewed by itself as a distinct set of the partition). In effect we use aggregation to construct a nonlinear (piecewise constant) feature-based architecture, which may be much more powerful than the corresponding linear architecture.
Figure 3.5.7 Feature-based hard aggregation using a partition of the space of features. Each aggregate state $y$ has a footprint $S_y$ that consists of states with “similar” features, i.e., states that map into the same subset of a partition in the space of features.

The question now arises how to obtain a suitable feature vector when there is no obvious choice, based on problem-specific considerations. One possibility, discussed in the book [Ber19a] (Section 6.4), is to obtain “good” features by using a neural network. In fact any method that automatically generates features from data may be used. Here we will discuss a simple possibility.

Using Scoring Functions

Suppose that we have obtained in some way a real-valued scoring function $V(i)$ of the state $i$, which serves as an index of undesirability of state $i$ as a starting state (smaller values of $V$ are assigned to more desirable states, consistent with the view of $V$ as some form of “cost” function). One possibility is to use as $V$ an approximation of the cost function of some “good” (e.g., near-optimal) policy. Another possibility is to obtain $V$ by problem approximation, i.e., as the cost function of some reasonable policy applied to an approximation of the original problem. Still another possibility is to obtain $V$ by training a neural network or other architecture using samples of state-cost pairs obtained by using a software or human expert, and some supervised learning technique.

Given the scoring function $V$, we will construct a feature mapping that groups together states $i$ with roughly equal scores $V(i)$. In particular, we let $R_x$, $x = 1, \ldots, q$, be $q$ disjoint intervals that form a partition of the range of possible values of $V$ [i.e., are such that for any state $i$, there is a unique interval $R_x$ such that $V(i) \in R_x$]. We define a feature vector $F(i)$ of the state $i$ according to

$$F(i) = x, \quad \text{for all } i \text{ such that } V(i) \in R_x, \quad x = 1, \ldots, q. \quad (3.53)$$

This feature in turn defines a partition of the state space into the sets

$$I_x = \{ i \mid F(i) = x \} = \{ i \mid V(i) \in R_x \}, \quad x = 1, \ldots, q. \quad (3.54)$$
Figure 3.5.8. Hard aggregation scheme based on a single scoring function. We introduce $q$ disjoint intervals $R_1, \ldots, R_q$ that form a partition of the set of possible values of $V$, and we define a feature vector $F(i)$ of the state $i$ according to

$$F(i) = x, \quad \text{for all } i \text{ such that } V(i) \in R_x, \ x = 1, \ldots, q.$$ 

This feature vector in turn defines a partition of the state space into the sets

$$I_x = \{i \mid F(i) = x\} = \{i \mid V(i) \in R_x\}, \quad x = 1, \ldots, q.$$ 

The sets $I_x$ coincide with the footprint sets $S_x$, and the solution of the aggregate problem yields a piecewise constant approximation of the optimal cost function of the original problem.

Assuming that all the sets $I_x$ are nonempty, we thus obtain a hard aggregation scheme, where the aggregate states are $x = 1, \ldots, q$, and the aggregation probabilities are defined by

$$\phi_{jx} = \begin{cases} 1 & \text{if } j \in I_x, \\ 0 & \text{otherwise}, \end{cases} \quad j = 1, \ldots, n, \ x = 1, \ldots, q, \quad (3.55)$$

see Fig. 3.5.8. Note that the sets $I_x$ coincide with the footprint sets $S_x$.

The following proposition (due to Tsitsiklis and VanRoy [TsV96]) illustrates the important role of the quantization error, defined as

$$\delta = \max_{x=1,\ldots,q} \max_{i,j \in I_x} |V(i) - V(j)|. \quad (3.56)$$

It represents the maximum error that can be incurred by approximating $V$ within each set $I_x$ with a single value from its range within the subset. Its
proof with additional discussion can be found in Chapter 6 of the author’s RL book [Ber19a].

**Proposition 3.5.2:** Consider the hard aggregation scheme defined by a scoring function $V$ as described above. Assume that the variations of $J^*$ and $V$ over the sets $I_1, \ldots, I_q$ are within a factor $\beta \geq 0$ of each other, i.e., that

$$|J^*(i) - J^*(j)| \leq \beta |V(i) - V(j)|,$$

for all $i, j \in I_x$, $x = 1, \ldots, q$.

(a) We have

$$|J^*(i) - r^*_x| \leq \frac{\beta \delta}{1 - \alpha},$$

for all $i \in I_x$, $x = 1, \ldots, q$,

where $\delta$ is the quantization error of Eq. (3.56).

(b) Assume that there is no quantization error, i.e., $V$ and $J^*$ are constant within each set $I_x$. Then the aggregation scheme yields the optimal cost function $J^*$ exactly, i.e.,

$$J^*(i) = r^*_x,$$

for all $i \in I_x$, $x = 1, \ldots, q$.

### 3.5.7 Biased Aggregation

In this section we will introduce an extension of the preceding aggregation framework. This extension involves a vector

$$V = (V(1), \ldots, V(n))$$

called the *bias vector* or *bias function*, which affects the cost structure of the aggregate problem, and biases the values of its optimal cost function towards their correct levels. When $V = 0$, we will obtain the aggregation scheme of Section 3.5.4. When $V \neq 0$, we will obtain a different aggregation scheme, which yields an approximation to $J^*$ that is equal to $V$ plus a local correction; see Fig. 3.5.10. In this case the aggregate DP problem aims to provide a correction/improvement to $V$, which may itself be a reasonably good estimate of $J^*$.

An obvious context where biased aggregation can be used is to improve on an approximation to $J^*$ obtained using a different method, such as for example by neural network-based approximate PI, by rollout, or by problem approximation. Generally, we may speculate that if $V$ captures
a fair amount of the nonlinearity of $J^*$, we may reduce the number of aggregate states needed for adequate performance.

Let us now formulate the aggregate problem in biased aggregation. It is a discounted infinite horizon problem that is similar to the (unbiased) aggregate problem of Section 3.5.4. It involves three sets of states: two copies of the original state space, denoted $I_0$ and $I_1$, as well as a finite set $A$ of aggregate states, as depicted in Fig. 3.5.11. The state transitions in the aggregate problem go from a state in $A$ to a state in $I_0$, according to disaggregation probabilities, then to a state in $I_1$, and then back to a state in $A$, according to aggregation probabilities, and the process is repeated. At state $i \in I_0$ we must choose a control $u \in U(i)$, and then transition to a state $j \in I_1$ at a cost $g(i, u, j)$ according to the original system transition probabilities $p_{ij}(u)$.

The salient new characteristic of the biased aggregation scheme is a (possibly nonzero) cost $-V(i)$ for transition from any aggregate state to a state $i \in I_0$, and of a cost $V(j)$ from a state $j \in I_1$ to any aggregate state; cf. Fig. 3.5.11. The function $V$ is the bias function, and we will argue that $V$ should be chosen as close as possible to $J^*$. Moreover, for practical purposes its values at various states should be easily computable.

A key insight is that biased aggregation can be viewed as unbiased aggregation applied to a modified DP problem, which is equivalent to the original DP problem in the sense that it has the same optimal policies. The modified DP problem is obtained from the original by changing its cost per stage from $g(i, u, j)$ to

$$g(i, u, j) - V(i) + \alpha V(j), \quad i, j = 1, \ldots, n, \ u \in U(i). \quad (3.57)$$

In particular, by comparing Figs. 3.5.6 and 3.5.11 it can be seen that unbiased aggregation applied to the modified DP problem gives the same
state-control trajectories as biased aggregation applied to the original DP problem, while the incurred transition costs (from aggregate state to aggregate state) are equal.

Moreover, there is a close connection between the optimal cost functions of the modified DP problem with cost per stage given by Eq. (3.57), and the original DP problem. In particular, the optimal cost function of the modified problem, call it $\tilde{J}$, satisfies the corresponding Bellman equation:

$$
\tilde{J}(i) = \min_{u \in U(i)} \sum_{j=1}^{n} p_{ij}(u) \left( g(i, u, j) - V(i) + \alpha V(j) + \alpha \tilde{J}(j) \right), \quad i = 1, \ldots, n,
$$

or equivalently

$$
\tilde{J}(i) + V(i) = \min_{u \in U(i)} \sum_{j=1}^{n} p_{ij}(u) \left( g(i, u, j) + \alpha (\tilde{J}(j) + V(j)) \right), \quad i = 1, \ldots, n.
$$

By comparing this equation with the Bellman equation for the original problem, we see that the optimal cost functions of the modified and the original problems are related by

$$
J^*(i) = \tilde{J}(i) + V(i), \quad i = 1, \ldots, n,
$$

and that the two problems have the same optimal policies. This of course assumes that the original and modified problems are solved exactly. If instead they are solved approximately using aggregation or another approximation architecture, such as a neural network, the policies obtained may be substantially different. In particular, the choice of $V$ and the approximation architecture may affect substantially the quality of suboptimal policies obtained.
To summarize, any unbiased aggregation scheme and algorithm, when applied to the modified DP problem with cost per stage given by Eq. (3.57), yields a biased aggregation scheme and algorithm for the original DP problem. Thus, we can straightforwardly transfer results, algorithms, and intuition from our earlier unbiased aggregation analysis to the biased aggregation framework, by applying them to the unbiased aggregation framework that corresponds to the modified stage cost (3.57). Moreover, we may use simulation-based algorithms for policy evaluation, policy improvement, and Q-learning for the aggregate problem, with the only requirement that the value $V(i)$ for any state $i$ is available when needed.

### 3.5.8 Asynchronous Distributed Multiagent Aggregation

Let us now discuss the distributed solution of large-scale discounted DP problems using cost function approximation, multiple agents/processors, and hard aggregation. Here we partition the original system states into aggregate states/subsets $x \in A = \{x_1, \ldots, x_m\}$, and we envision a network of processors/agents, each updating asynchronously a detailed/exact local cost function, defined on a single aggregate state/subset. Each processor also maintains an aggregate cost for its aggregate state, which is a weighted average of the detailed cost of the (original system) states in the processor’s subset, weighted by the corresponding disaggregation probabilities. These aggregate costs are communicated between processors and are used to perform the local updates.

In a synchronous VI method of this type, each processor $\ell = 1, \ldots, m$, maintains/updates a (local) cost $J(i)$ for every original system state $i \in x_\ell$, and an aggregate cost

$$R(\ell) = \sum_{i \in x_\ell} d_{x_\ell i} J(i),$$

where $d_{x_\ell i}$ are the corresponding disaggregation probabilities. We generically denote by $J$ and $R$ the vectors with components $J(i), i = 1, \ldots, n$, and $R(\ell), \ell = 1, \ldots, m$, respectively. These components are updated according to

$$J_{k+1}(i) = \min_{u \in U(i)} H_{\ell}(i, u, J_k, R_k), \quad \forall \ i \in x_\ell, \quad (3.58)$$

with

$$R_k(\ell) = \sum_{i \in x_\ell} d_{x_\ell i} J_k(i), \quad \ell = 1, \ldots, m, \quad (3.59)$$

where the mapping $H_{\ell}$ is defined for all $\ell = 1, \ldots, m$, $i \in x_\ell$, $u \in U(i)$, and $J \in \mathbb{R}^n$, $R \in \mathbb{R}^m$, by

$$H_{\ell}(i, u, J, R) = \sum_{j=1}^n p_{ij}(u) g(i, u, j) + \alpha \sum_{j \in x_\ell} p_{ij}(u) J(j) + \alpha \sum_{j \notin x_\ell} p_{ij}(u) R(x(j)), \quad (3.60)$$
Sec. 3.5 Aggregation

81

and where for each original system state \( j \), we denote by \( x(j) \) the subset to which \( j \) belongs [i.e., \( j \in x(j) \)]. Thus the iteration (3.58) is the same as ordinary VI, except that instead of \( J(j) \), we use the aggregate costs \( R(x(j)) \) for the states \( j \) whose costs are updated by other processors.

It is possible to show that the iteration (3.58)-(3.59) involves a sup-norm contraction mapping of modulus \( \alpha \), so it converges to the unique solution of the system of equations in \((J, R)\)

\[
J(i) = \min_{u \in U(i)} H_\ell(i, u, J, R), \quad R(\ell) = \sum_{i \in x_\ell} d_{x_\ell i} J(i),
\]

\[\forall i \in x_\ell, \quad \ell = 1, \ldots, m. \tag{3.61}\]

This follows from the fact that \( \{d_{x_\ell i} \mid i = 1, \ldots, n\} \) is a probability distribution. We may view the equations (3.61) as a set of Bellman equations for an "aggregate" DP problem, which similar to our earlier discussion, involves both the original and the aggregate system states. The difference from the Bellman equations (3.47)-(3.49) is that the mapping (3.60) involves \( J(j) \) rather than \( R(x(j)) \) for \( j \in x_\ell \).

In the algorithm (3.58)-(3.59), all processors \( \ell \) must be updating their local costs \( J(i) \) and aggregate costs \( R(\ell) \) synchronously, and communicate the aggregate costs to the other processors before a new iteration may begin. This is often impractical and time-wasting. In a more practical asynchronous version of the method, the aggregate costs \( R(\ell) \) may be outdated to account for communication "delays" between processors. Moreover, the costs \( J(i) \) need not be updated for all \( i \); it is sufficient that they are updated by each processor \( \ell \) only for a (possibly empty) subset of \( I_{\ell,k} \) of the aggregate state/set \( x_\ell \). In this case, the iteration (3.58)-(3.59) is modified to take the form

\[
J_{k+1}(i) = \min_{u \in U(i)} H_\ell(i, u, J_k, R_{\tau_{1,k}}(1), \ldots, R_{\tau_{m,k}}(m)), \quad \forall i \in I_{\ell,k}, \tag{3.62}\]

with \( 0 \leq \tau_{\ell,k} \leq k \) for \( \ell = 1, \ldots, m \), and

\[
R_\tau(\ell) = \sum_{i \in x_\ell} d_{x_\ell i} J_\tau(i), \quad \forall \ell = 1, \ldots, m.
\]

The differences \( k - \tau_{\ell,k}, \ell = 1, \ldots, m \), in Eq. (3.62) may be viewed as "delays" between the current time \( k \) and the times \( \tau_{\ell,k} \) when the corresponding aggregate costs were computed at other processors. For convergence, it is of course essential that every \( i \in x_\ell \) belongs to \( I_{\ell,k} \) for infinitely many \( k \) (so each cost component is updated infinitely often), and \( \lim_{k \to \infty} \tau_{\ell,k} = \infty \) for all \( \ell = 1, \ldots, m \) (so that processors eventually communicate more recently computed aggregate costs to other processors).

The convergence of this type of method based on the sup-norm contraction property of the mapping underlying Eq. (3.61), can be established
using an asynchronous convergence theory for DP developed by the author in the paper \cite{Ber82} (see also the books \cite{BeT89}, \cite{Ber12}). The monotonicity property is also sufficient to establish convergence, and this is useful in the convergence analysis of related aggregation algorithms for nondiscounted DP models (see the paper by Bertsekas and Yu \cite{BeY10}).

### 3.6 NOTES AND SOURCES

#### Section 3.1: Our discussion of approximation architectures, neural networks, and training has been limited, and aimed just to provide the connection with approximate DP. The literature on the subject is vast, and some of the textbooks mentioned in the references to Chapter 1 provide detailed accounts and many sources, in addition to the ones given in Sections 3.1 and 3.2.

There are two broad directions of inquiry in parametric architectures:

1. The design of architectures, either in a general or a problem-specific context.
2. The training of neural networks, as well as other linear and nonlinear architectures.

Research along both of these directions has been extensive and is continuing.

Methods for selection of basis functions have received much attention, particularly in the context of neural network research and deep reinforcement learning (see e.g., the book by Goodfellow, Bengio, and Courville \cite{GBC16}). For discussions that are focused outside the neural network area, see Bertsekas and Tsitsiklis \cite{BeT96}, Keller, Mannor, and Precup \cite{KMP06}, Jung and Polani \cite{JuP07}, Bertsekas and Yu \cite{BeY09}, and Bhatnagar, Borkar, and Prashanthi \cite{BBP13}. Moreover, there has been considerable research on optimal feature selection within given parametric classes (see Menache, Mannor, and Shimkin \cite{MMS05}, Yu and Bertsekas \cite{YuB09}, Busoniu et al. \cite{BBD10a}, and Di Castro and Mannor \cite{DiM10}).

Incremental algorithms are the principal methods for training approximation architectures. They are supported by substantial theoretical analysis, which addresses issues of convergence, rate of convergence, step-size selection, and component order selection. Moreover, incremental algorithms have been extended to constrained optimization settings, where the constraints are also treated incrementally, first by Nedić \cite{Ned11}, and then by several other authors: Bertsekas \cite{Ber11a}, Wang and Bertsekas \cite{WaB15}, \cite{WaB16}, Bianchi \cite{Bia16}, Iusem, Jofre, and Thompson \cite{IJT18}. It is beyond our scope to cover this analysis. The author’s surveys \cite{Ber10a} and \cite{Ber15b}, and convex optimization and nonlinear programming textbooks \cite{Ber15a}, \cite{Ber16}, collectively contain an extensive account of incremental methods, including the Kaczmarz, incremental gradient, subgradient, ag-
Section 3.6: Notes and Sources

Aggregated gradient, Newton, Gauss-Newton, and extended Kalman filtering methods, and give many references. The book [BeT96] and paper [BeT00] by Bertsekas and Tsitsiklis, and the survey by Bottou, Curtis, and Nocedal [BCN18] provide theoretically oriented treatments.

**Section 3.2:** The publicly and commercially available neural network training programs incorporate heuristics for scaling and preprocessing data, stepsize selection, initialization, etc, which can be very effective in specialized problem domains. We refer to books on neural networks such as Bishop [Bis95], Goodfellow, Bengio, and Courville [GBC16], and Haykin [Hay08].

Deep neural networks have created a lot of excitement in the machine learning field, in view of some high profile successes in image and speech recognition, and in RL with the AlphaGo and AlphaZero programs. One question is whether and for what classes of target functions we can enhance approximation power by increasing the number of layers while keeping the number of weights constant. For analysis and speculation around this question, see Bengio [Ben09], Liang and Srikant [LiS16], Yarotsky [Yar17], Daubechies et al. [DDF19], and the references quoted there.

Another important research question relates to the role of overparameterization in the success of deep neural networks. With more weights than training data, the training problem has infinitely many solutions, each providing an architecture that fits the training data perfectly. The question then is how to select a solution that works well on test data (i.e., data outside the training set); see Zhang et al. [ZBH16], [ZBH21], Belkin, Ma, and Mandal [BMM18], Belkin, Rakhlin, and Tsybakov [BRT18], Soltanolkotabi, Javanmard, and Lee [SLL18], Bartlett et al. [BLL19], Hastie et al. [HMR19], Muthukumar, Vodrahalli, and Sahai [MVS19], Su and Yang [SuY19], Sun [Sun19], Vaswani et al. [VLK21], Zhang et al. [ZBH21] and the discussion in the book by Hardt and Recht [HaR21].

**Section 3.3:** Fitted value iteration has a long history; it was mentioned by Bellman among others. It has interesting properties, and at times exhibits pathological/unstable behavior due to accumulation of errors over a long horizon (see [Ber19a], Section 5.2).

The approximate policy iteration method of Section 3.3.3 has been proposed by Fern, Yoon, and Givan [FYG06], and variants have also been discussed and analyzed by several other authors. The method (with some variations) has been used to train a tetris playing computer program that performs impressively better than programs that are based on other variants of approximate policy iteration, and various other methods; see Scherrer [Sch13], Scherrer et al. [SGG15], and Gabillon, Ghavamzadeh, and Scherrer [GGS13], who also provide an analysis of the method. The RL and approximate DP books collectively describe several alternative simulation-based methods for policy evaluation, including the popular temporal difference methods; see e.g., [BeT96], [SuB18], [Ber12], Chapters 6 and 7. The book [Ber20a] describes distributed versions of approximate policy itera-
tion, which are based on partitioning of the state space.

The original proposal of SARSA (Section 3.3.4) is attributed to Rum-mery and Niranjan [RuN94], with related work presented in the papers by Peng and Williams [FeW96], and Wiering and Schmidhuber [WiS98]. The ideas of the DQN algorithm attracted much attention following the paper by Mnih et al. [MKS15], which reported impressive test results on a suite of 49 classic Atari 2600 games.

The rollout and approximate PI methodology for POMDP of Section 3.3.5 was described in the author’s RL book [Ber19a]. It was extended and tested in the paper by Bhattacharya et al. [BBW20] in the context of a challenging pipeline repair problem.

Advantage updating (Section 3.3.6) was proposed by Baird [Bai93], [Bai94], and is discussed further in Section 6.6 of the neuro-dynamic programming book [BeT96]. The differential training methodology (Section 3.3.7) was proposed by the author in the paper [Ber97b], and followup work was presented by Weaver and Baxter [WeB99].

Section 3.4: Classification (sometimes called “pattern classification” or “pattern recognition”) is a major subject in machine learning, for which there are many approaches, an extensive literature, and an abundance of public domain and commercial software; see e.g. the textbooks by Bishop [Bis95], [Bis06], Duda, Hart, and Stork [DHS12], and Hardt and Recht [HaR21]. Approximation in policy space was formulated as a classification problem in the context of DP by Lagoudakis and Parr [LaP03], and was followed up by several other authors (see e.g., Dimitrakakis and Lagoudakis [DiL08], Lazaric, Ghavamzadeh, and Munos [LGM10], Gabil-lon et al. [GLG11], Liu and Wei [LiW14], Furahmand et al. [FPB15], and the references quoted there). While we have focused on a classification approach that makes use of least squares regression and a parametric architecture, other classification methods may also be used. For example the paper [LaP03] discusses the use of nearest neighbor schemes, support vector machines, as well as neural networks.

Section 3.5: The aggregation approach has a long history in scientific computation and operations research (see for example Bean, Birge, and Smith [BBS87], Chatelin and Miranker [ChM82], Douglas and Douglas [DoD93], and Rogers et al. [RPW91]). It was introduced in the simulation-based approximate DP context, mostly in the form of VI; see Singh, Jaakkola, and Jordan [SJJ95], Gordon [Gor95], and Tsitsiklis and Van Roy [TsV96]. It was further discussed in the neuro-dynamic programming book [BeT96], Sections 3.1.2 and 6.7.

The aggregation framework with representative features was introduced in the author’s book [Ber12], was discussed in detail in the RL textbook [Ber19a] (Chapter 6), and was further developed in the author’s survey paper [Ber18b], which provides an expanded view of the methodology. Biased aggregation (Section 3.5.7) was first proposed in the author’s
paper [Ber18c], which contains further discussion, connections with rollout algorithms, and additional methods.

Distributed asynchronous aggregation (Section 3.5.8) was first proposed in the paper by Bertsekas and Yu [BeY10] (Example 2.5); see also the discussions in author’s DP books [Ber12] (Section 6.5.4) and [Ber22b] (Example 1.2.11). A recent computational study related to distributed traffic routing is given by Vertovec and Margellos [VeM23].
Complete the details of the following proof of Prop. 3.4.1. Consider for any pair \((c, x)\) the conditional expected value \(E\{ (z(c, x) - y)^2 \mid c, x \}\), where \(y\) is any scalar. Given \((c, x)\), the random variable \(z(c, x)\) takes the value \(z(c, x) = 1\) with probability \(p(c \mid x)\) and the value \(z(c, x) = 0\) with probability \(1 - p(c \mid x)\), so we have
\[
E\{ (z(c, x) - y)^2 \mid c, x \} = p(c \mid x)(y - 1)^2 + (1 - p(c \mid x))y^2.
\]
We minimize this expression with respect to \(y\), by setting to 0 its derivative, i.e.,
\[
0 = 2p(c \mid x)(y - 1) + 2(1 - p(c \mid x))y = 2\left( -p(c \mid x) + y \right).
\]
We thus obtain the minimizing value of \(y\), namely \(y^* = p(c \mid x)\), so that
\[
E\{ (z(c, x) - p(c \mid x))^2 \mid c, x \} \leq E\{ (z(c, x) - h(c, x))^2 \mid c, x \},
\]
for all scalars \(y\).

For any function \(h\) of \((c, x)\) we set \(y = h(c, x)\) in the above expression and obtain
\[
E\{ (z(c, x) - p(c \mid x))^2 \mid c, x \} \leq E\{ (z(c, x) - h(c, x))^2 \mid c, x \}.
\]
Since this is true for all \((c, x)\), we also have
\[
\sum_{(c, x)} \zeta(c, x)E\{ (z(c, x) - p(c \mid x))^2 \mid c, x \} \leq \sum_{(c, x)} \zeta(c, x)E\{ (z(c, x) - h(c, x))^2 \mid c, x \},
\]
for all functions \(h\). By using the theorem of total probability (see e.g., [BeT08], Chapter 1), this is equivalent to
\[
E\{ (z(c, x) - p(c \mid x))^2 \} \leq E\{ (z(c, x) - h(c, x))^2 \},
\]
i.e., Eq. (3.35) holds.