Rollout, Approximate Policy Iteration, and Distributed Reinforcement Learning

by

Dimitri P. Bertsekas

Chapter 3
Learning Values and Policies

These notes represent “work in progress,” and will be periodically updated. They more than likely contain errors (hopefully not serious ones). Furthermore, the references to the literature are incomplete. Your comments and suggestions to the author at dbertsek@asu.edu are welcome.

February, 2020
Learning Values and Policies

Contents

3.1. Approximation Architectures .............................................. p. 126
   3.1.1. Feature-Based Architectures ....................................... p. 126
   3.1.2. Training of Linear and Nonlinear Architectures .......... p. 134
3.2. Neural Networks ............................................................... p. 149
   3.2.1. Training of Neural Networks ....................................... p. 153
   3.2.2. Multilayer and Deep Neural Networks ......................... p. 157
3.3. Training of Cost Functions in Approximate DP ................. p. 161
3.4. Training of Policies in Approximate DP .......................... p. 165
3.5. Notes and Sources ............................................................ p. 171
Clearly, for the success of approximation in value space, it is important to select a class of lookahead functions $\tilde{J}_k$ that is suitable for the problem at hand. In the preceding chapter we discussed several methods for choosing $\tilde{J}_k$ based mostly on rollout. In this chapter we discuss how $\tilde{J}_k$ is chosen to be a member of a parametric class of functions, including neural networks, with the parameters “optimized” or “trained” by using some algorithm.

The training methods used for parametric approximation in value space can also be used for approximation in policy space, as we will discuss in Section 3.4. The training data may be obtained, for example, from rollout calculations, thus enabling the construction of value and policy networks that can be used for perpetual rollout. This approach to learning an approximate cost function and policy will be reconsidered and extended in the context of infinite horizon problems and approximate policy iteration in Chapter 4.

3.1 APPROXIMATION ARCHITECTURES

The starting point for the schemes of this chapter is a 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, also called weights. By adjusting the weights, one can change the “shape” of $\tilde{J}_k$ so that it is a reasonably good approximation to the true optimal cost-to-go function $J^*_k$. The class of functions $\tilde{J}_k(x_k, r_k)$ 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 $\tilde{J}_k$ notation. In Section 3.4 we will consider the other major use of parametric approximation architectures, namely to represent policies in the form $\tilde{\mu}_k(x_k, r_k)$.

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 methods have also been used to tune hyperparameters of an approximation architecture, relating to the structure of the 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. In particular, one such method, known as Bayesian optimization, has been used in the training of the AlphaZero architecture [SHS17]. We refer to the literature for further discussion.

Other systematic approaches are based on numerical optimization, such as for example 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 this chapter we will focus primarily on this approach.

### 3.1.1 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

$$\phi_{1,k}(x_k), \ldots, \phi_{m,k}(x_k),$$

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

$$\tilde{J}_k(x_k, r_k) = \hat{J}_k(\phi_k(x_k), r_k),$$

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 \( \hat{J}_k \) 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.,

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

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. 3.1.1.

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. Mathematically, the approximating function \( \hat{J}_k(x_k, r_k) \) 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 basis functions. We provide a few examples, where for simplicity we drop the index \( k \).

**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.,

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

Consider the architecture

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

where \( r \) is the vector consists of the \( m \) scalar parameters \( r_1, \ldots, r_m \). It can be seen that \( \hat{J}(x, r) \) is the piecewise constant function that has value \( r_\ell \) for all states within the set \( S_\ell \); see Fig. 3.1.2.

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 \( \hat{J}(x, r) \) 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 an example.
Sec. 3.1 Approximation Architectures

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

**Figure 3.1.2** 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

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

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

**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

\[ \phi(x) = (\phi_1(x), \ldots, \phi_n(x))^\prime, \]
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.

**Example 3.1.3 (Tetris)**

Tetris is a popular video game played on a two-dimensional grid. Each square in the grid can be full or empty, making up a “wall of bricks” with “holes” and a “jagged top” (see Fig. 3.1.3). 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. 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. The falling blocks are generated independently according to some probability distribution, defined over a finite set of standard shapes. The game starts with an empty grid and ends when a square in the top row becomes full and the top of the wall reaches the top of the grid. 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. We can model the problem of finding an optimal playing strategy as a stochastic shortest path infinite horizon problem, where the termination state is the end of the game.

The state here 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 [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 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.

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 limited 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

† The use of feature-based approximate DP methods for the game of tetris was first suggested in the paper [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].

‡ We have discussed DP and rollout for finite horizon minimax problems in Section 2.9. Two-player games admit a similar DP treatment, and the ideas of approximation in value space apply to these contexts as well; see [Ber17] for a discussion that is focused on computer chess.
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. 3.1.4). In most computer chess programs, the features are weighted linearly, i.e., the architecture $J(x,r)$ that is used for limited lookahead is linear [cf. Eq. (3.1)]. In many cases, the weights are determined manually, by trial and error based on experience. However, in some programs, the weights are 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 is based on DP principles of policy iteration and 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.

**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

$$\phi(\ell)(x) = e^{-\frac{\|x - x(\ell)\|^2}{2\sigma^2}}, \quad \ell = 1, \ldots, m,$$  \hspace{1cm} (3.2)
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 $x(t)$, 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 $x(t)$ in Eq. (3.2), 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-Tylor [ChS00], [ShC04], Scholkopf and Smola [ScS02], Bishop [Bis06], Kung [Kun14], surveys such as Hofmann, Scholkopf, and Smola [HSS08], Pillonetto et al. [PDC14], and 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. [BBS10], Bethke [Bet10]. In what follows, we will focus on parametric architectures with a fixed and given feature vector.

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 1.3, 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 an information record (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 $\widehat{\mu}_k$, we have

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

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 1.3.6). For a proof that $b_k$ is indeed a sufficient statistic and for a more detailed discussion of other possible sufficient statistics, see [Ber17], 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 [YuB06], 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 $b_k$.

Of course it is possible 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 Aggregation)**

The use of a feature vector $\phi(x)$ to represent the state $i$ in an approximation architecture of the form $\tilde{J}(\phi(x), r)$ implicitly involves state aggregation, that is, the grouping of states into subsets. In particular, let us assume that the feature vector can take only a finite number of values, and let us define for each possible value $v$, the subset of states $S_v$ whose feature vector is equal to $v$:

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

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 $\tilde{J}(\phi(x), r)$ is piecewise constant with respect to this partition; that is, it assigns the same cost-to-go value $\tilde{J}(v, r)$ 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 $\tilde{J}(v)$
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 \( \hat{J}(\phi(x)) \) of the feature vector \( f(i) \) in the aggregate problem. There is considerable freedom on how one formulates and solves aggregate problems. We refer to [Ber19a], Chapter 6, for a detailed treatment.

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

**Example 3.1.8 (Feature-Based 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, one can construct piecewise linear or piecewise quadratic approximations over the entire state space. Similarly, one 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. 3.1.5.

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
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 styles of play. 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 various 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 mapping we are trying to approximate 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

$$\tilde{J}(x, r) = \hat{J}(x, \hat{r}) + \sum_{m=1}^{M} \sum_{k=1}^{K_m} r_m(k) \phi_{k,m}(x), \quad (3.3)$$

where each $\phi_{k,m}(x)$ is a basis function which contributes to the approximation only on the set $S_m$; that is, it takes the value 0 for $i \notin S_m$. 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 mapping we are trying to approximate, while each portion

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

is used to capture aspects of the mapping that are “local” to the subset $S_m$.

The book [BeT96] describes some special methods for training local-global approximation architectures, which take advantage of their special structure.

**Mixed Architectures**

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.

### 3.1.2 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 \( \hat{J}(x, r) \), 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

\[
\min_r \sum_{s=1}^{q} (\hat{J}(x^s, r) - \beta^s)^2.
\]

Thus \( r \) is chosen to minimize the sum of squared errors between the sample costs \( \beta^s \) and the architecture-predicted costs \( \hat{J}(x^s, r) \). Here there is some “target” cost function \( J \) that we aim to approximate with \( \hat{J}(\cdot, r) \), 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 (3.4) is generally nonconvex, which 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 \( \hat{J}(x, r) = r^{'} \phi(x) \), the problem becomes

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

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

\[
\sum_{s=1}^{q} \phi(x^s)(r^{'} \phi(x^s) - \beta^s) = 0,
\]
or
\[
\sum_{s=1}^{q} \phi(x^s)\phi(x^s)'r = \sum_{s=1}^{q} \phi(x^s)\beta^s.
\]

Thus by matrix inversion we obtain the minimizing parameter vector
\[
\hat{r} = \left( \sum_{s=1}^{q} \phi(x^s)\phi(x^s)' \right)^{-1} \sum_{s=1}^{q} \phi(x^s)\beta^s. \tag{3.5}
\]

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 (3.5) (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. 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 special methods for solution of the least squares training problem (3.4), 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, and return later as needed.

We view the training problem (3.4) as a special case of the minimization of a sum of component functions
\[
f(y) = \sum_{i=1}^{m} f_i(y), \tag{3.6}
\]
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 \((\hat{J}(x^s, r) - \beta^s)^2\) in the training problem (3.4) with the generic terms \(f_i(y)\).
Sec. 3.1 Approximation Architectures

The (ordinary) gradient method for problem (3.6) 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†

\[
y^{k+1} = y^k - \gamma^k \nabla f(y^k) = y^k - \gamma^k \sum_{i=1}^m \nabla f_i(y^k), \tag{3.7}
\]

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

\[
y^{k+1} = y^k - \gamma^k \nabla f_{i_k}(y^k), \tag{3.8}
\]

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 (3.7), particularly when \( m \) is large.

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. Three common rules are:

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 \( (k \mod m) + 1 \). 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.

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, for otherwise some components will be sampled more often than

† We use standard calculus notation for gradients; see, e.g., [Ber16a], 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 \( y \).
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 generically 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 incremental gradient method, which is discussed in the author’s textbooks [Ber15a], [Ber16a], [Ber19a].

Regardless of whether a constant or a diminishing stepsize is ultimately used, to maintain the advantage of faster convergence when far from the solution, 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 $\gamma_j$ 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 stepsize

$$\gamma_j = \gamma \left( \frac{\partial^2 f(y_k)}{\partial^2 y_j} \right)^{-1}, \quad j = 1, \ldots, n,$$

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. (3.6). 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 of diagonal scaling with second derivatives, which is based on Newton’s method and is well suited to the additive character of the cost function $f$.

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, 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.

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 $	ilde{J}(x, v, r)$ of the form

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

(3.9)

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 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 (3.9) 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, and how the training problem is solved.† Notice the different

† 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.
Figure 3.2.1 A perceptron 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 functions that produce 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 $r_\ell$.

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 $\phi(x,v)$.

A neural network architecture provides a parametric class of functions $\tilde{J}(x,v,r)$ of the form (3.9) 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

$$Ay(x) + b,$$

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 function $\sigma$ that maps scalars to scalars. Typically $\sigma$ is differentiable and monotonically increasing. A simple and popular possibility is the rectified linear unit (ReLU for short),

† The method of encoding $x$ into the numerical vector $y(x)$ is problem-dependent, but it is important to note that some of the components of $y(x)$ could be some known interesting features of $x$ that can be designed based on problem-specific knowledge.
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 illustrated in Fig. 3.2.2. Other functions, used since the early days of neural networks, have the property

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

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

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 possible 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.

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 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 closed and bounded subset $S \subset X$, any piecewise continuous function $J : S \mapsto \mathbb{R}$, provided the number $m$ of nonlinear units is sufficiently large. For proofs of the theorem at different levels of generality, we refer to Cybenko [Cyb89], Funahashi [Fun89], Hornik, Stinchcombe, and White [HSW89], and Leshno et al. [LLP93]. For intuitive explanations we refer to Bishop ([Bis95], pp. 129-130) and Jones [Jon90].

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 reduced to trying increasingly larger values of $m$ until one is convinced that satisfactory performance has been obtained for the task at hand. 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. One 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.

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 training problem

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

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 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 well known difficulty in machine learning, 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 (3.10). 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 (3.10). 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 3.1.2 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] 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, 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.

### 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 as additional inputs some of the components of the state \( x \) or the state encoding \( y(x) \).

† 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.
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. 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 multilayer neural network) may help 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], [SJL18], [BLL19], [HMR19], [MVS19] for some representative works.

Note that while in the early days of neural networks practitioners tended to use few nonlinear layers (say one to three), more recently a lot of success in certain problem domains (including image and speech processing, as well as approximate DP) has been achieved with so called deep neural networks, which involve a considerably larger number of layers. In particular, 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 (cf. Section 2.4.2) have performed well with one or two nonlinear layers [PaR12].
3.3 TRAINING OF COST FUNCTIONS IN APPROXIMATE DP

In the context of 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. In Chapter 4, we will illustrate the combined use of policy and value networks in perpetual rollout, a form of repeated rollout, whereby the rollout policy and its cost function are approximated by a policy and a value network, to generate a sequence of (approximately) improved rollout policies.

Let us describe a popular approach for training an approximation architecture \( \tilde{J}_k(x_k, r_k) \) for a finite horizon DP problem. The parameter vectors \( r_k \) are determined sequentially, with an algorithm known as fitted value iteration, 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.

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 typical stage \( k \), having obtained \( r_{k+1} \), we determine \( r_k \) from the least squares problem

\[
\begin{align*}
    r_k &\in \arg \min_r \sum_{s=1}^q \left( \tilde{J}_k(x^s_k, r) - \min_{u \in U_k(x^s_k)} E \left\{ g(x^s_k, u, w_k) + \tilde{J}_{k+1}(f_k(x^s_k, u, w_k), r_{k+1}) \right\} \right)^2 \\
    \beta^s_k &= \min_{u \in U_k(x^s_k)} E \left\{ g(x^s_k, u, w_k) + \tilde{J}_{k+1}(f_k(x^s_k, u, w_k), r_{k+1}) \right\}, \quad (3.11)
\end{align*}
\]

where \( x^s_k \), \( i = 1, \ldots, q \), 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

\[
\beta^s_k = \min_{u \in U_k(x^s_k)} E \left\{ g(x^s_k, u, w_k) + \tilde{J}_{k+1}(f_k(x^s_k, u, w_k), r_{k+1}) \right\}, \quad (3.11)
\]

so that \( r_k \) is obtained as

\[
r_k \in \arg \min_r \sum_{s=1}^q \left( \tilde{J}_k(x^s_k, r) - \beta^s_k \right)^2. \quad (3.12)
\]
The algorithm starts at stage $N - 1$ with the minimization
\[
\begin{align*}
  r_{N-1} \in \arg \min_r \ & \sum_{s=1}^q \left( \hat{J}_{N-1}(x_{N-1}^s, r) \right. \\
  & \\
  \left. - \min_{u \in U_{N-1}(x_{N-1}^s)} \mathbb{E} \left\{ g_{N-1}(x_{N-1}^s, u, w_{N-1}) + g_N(f_{N-1}(x_{N-1}^s, u, w_{N-1})) \right\} \right)^2 \\
\end{align*}
\]
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
\[
\hat{J}_k(x_k, r_k) = r'_k \phi_k(x_k), \quad k = 0, \ldots, N - 1,
\]
the least squares problem (3.12) greatly simplifies, and admits the closed form solution
\[
 r_k = \left( \sum_{s=1}^q \phi_k(x_k^s) \phi_k(x_k^s)' \right)^{-1} \sum_{s=1}^q \beta_s \phi_k(x_k^s);
\]
cf. Eq. (3.5). 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. This point will be discussed later, in the context of infinite horizon problems, and the notion of “representative” state will be better quantified in probabilistic terms.

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 is very long, since then the number of parameters 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 4.

**Q-Factor Parametric Approximation**

We will now consider an alternative form of approximation in value space and fitted value iteration. It 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. The optimal Q-factors are defined by

\[
Q_k^* (x_k, u_k) = E \left\{ g_k(x_k, u_k, w_k) + J_{k+1}^* (f_k(x_k, u_k, w_k)) \right\}, \quad k = 0, \ldots, N-1,
\]

(3.13)

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 1.2, 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. (3.13) in the following equivalent form that relates \(Q_k^*\) with \(Q_{k+1}^*\):

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

(3.14)

This suggests that in place of the Q-factors \(Q_k^* (x_k, u_k)\), we may use Q-factor approximations and Eq. (3.14) 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

\[
\tilde{Q}_k^* (x_k, u_k, r_k) = r_k (u_k)' \phi_k (x_k),
\]

(3.15)

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

\[
\tilde{Q}_k^* (x_k, u_k, r_k) = r_k' \phi_k (x_k, u_k),
\]

(3.16)

where \(r_k\) is a weight vector that is independent of \(u_k\). The architecture (3.15) is suitable for problems with a relatively small number of control options at each stage. In what follows, we will focus on the architecture (3.16), but the discussion with few modifications, also applies to the architecture (3.15) 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. (3.14), with \(r_k\) obtained by solving least squares problems similar to the ones of the cost function approximation case [cf. Eq. (3.12)]. As
an example, the parameters $r_k$ of the architecture (3.16) are computed se-
quentially by collecting sample state-control pairs $(x^s_k, u^s_k)$, $s = 1, \ldots, q$, and solving the linear least squares problems

$$r_k \in \arg \min_r \sum_{s=1}^q (r^s \phi_k(x^s_k, u^s_k) - \beta^s_k)^2, \quad (3.17)$$

where

$$\beta^s_k = E \left\{ g_k(x^s_k, u^s_k, w_k) + \min_{u \in U_{k+1}(f_k(x^s_k, u^s_k, w_k))} r'_{k+1} \phi_{k+1} (f_k(x^s_k, u^s_k, w_k), u) \right\}. \quad (3.18)$$

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. (3.14).
Note that the solution of the least squares problem (3.17) can be ob-
tained in closed form as

$$r_k = \left( \sum_{s=1}^q \phi_k(x^s_k, u^s_k) \phi_k(x^s_k, u^s_k)' \right)^{-1} \sum_{s=1}^q \beta^s_k \phi_k(x^s_k, u^s_k); \quad [\text{cf. Eq. (3.5)}].$$

Once $r_k$ has been computed, the one-step lookahead control

$$\tilde{\mu}_k(x_k) \in \arg \min_u Q_k(x_k, u, r_k), \quad (3.19)$$

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. (3.18) involve the 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

$$g_k(x^s_k, u^s_k, w_k) + \min_{u \in U_{k+1}(f_k(x^s_k, u^s_k, w_k))} r'_{k+1} \phi_{k+1} (f_k(x^s_k, u^s_k, w_k), u), \quad (3.20)$$

collected according to the probability distribution of $w_k$. This distribu-
tion may either be known explicitly, or in a model-free situation, through
a computer simulator; cf. the discussion of Section 2.1.4. 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 (3.20) do not require
the computation of an expected value like the samples (3.11) in the cost
approximation method of the preceding chapter. Moreover the samples of
(3.20) involve a simpler minimization than the samples (3.11). 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. (3.19), 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; see the discussion of Section 2.1.5. The idea here is to simplify even further the on-line computation of the suboptimal controls by avoiding the minimization of Eq. (3.19).

### Advantage Updating

Let us finally note an 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

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

The function \( A_k(x_k, u_k) \) can serve just as well as \( Q^*_k(x_k, u_k) \) for the purpose of comparing controls, but may have a much smaller range of values than \( Q^*_k(x_k, u_k) \).

In the absence of approximations, selecting controls by advantage updating is clearly equivalent to selecting controls by comparing their Q-factors. However, when approximation is involved, using \( A_k \) can be important, because \( Q^*_k \) may embody sizable quantities that are independent of \( u \), and which may interfere with algorithms such as the fitted value iteration (3.17)-(3.18). In particular, 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. This question is discussed further and is illustrated with an example in the neuro-dynamic programming book [BeT96], Section 6.6.2.

The technique of subtracting a suitable constant (often called a baseline) from a quantity that is estimated by simulation is a useful idea; see Fig. 3.3.1 (in the case of advantage updating, the constants depend on \( x_k \), but the same general idea applies). It can also be used in the context of the sequential DP approximation method of Section 3.3, as well as in conjunction with other simulation-based methods in RL.
1. \( b = 0 \)

\[ \bar{b} \in \arg \min_r \sum_{s=1}^q \left( (h(u^s) - b) - ru^s \right)^2. \]

By properly adjusting \( b \), we can improve the quality of the approximation, which after subtracting \( b \) from all the sample values, takes the form \( \hat{h}(u, b, r) = b + ru \).

Conceptually, \( b \) serves as an additional weight (multiplying the basis function \( 1 \)), which enriches the approximation architecture.

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 dataset of “good” sample state-control pairs \( (x_k^s, u_k^s), s = 1, \ldots, q \), 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 \( \tilde{\mu}_k(x_k, r_k) \), using least squares minimization/regression:

\[ \bar{r}_k \in \arg \min_{r_k} \sum_{s=1}^q \| u_k^s - \tilde{\mu}_k(x_k^s, r_k) \|^2; \quad (3.21) \]
Learning Values and Policies Chap. 3

cf. our discussion of approximation in policy space in Section 2.1.

It is useful to make the connection of this regression approach with classification, an important problem in machine learning. This is the problem of constructing 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 \((x^*_k, u^*_k)\) 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 \(p(c|x)\). If we know the probabilities \(p(c|x)\), we can use a classical statistical approach, whereby we assign \(x\) to the category \(c^*(x)\) that has maximal posterior probability, i.e.,

\[
c^*(x) \in \arg \max_{c=1,\ldots,m} p(c|x).
\] (3.22)

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 \(p(c|x)\) are unknown, we may try to estimate them using a least squares optimization, based on the following property, which is proved in the book [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

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

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
\[
E\left\{ (z(c, x) - p(c \mid x))^2 \right\} \leq E\left\{ (z(c, x) - h(c, x))^2 \right\}. \tag{3.23}
\]

The proposition states that \(p(c \mid x)\) is the function of \((c, x)\) that minimizes
\[
E\left\{ (z(c, x) - h(c, x))^2 \right\} \tag{3.24}
\]
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 (3.24).

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. (3.24). In particular, we obtain \(r\) by the least squares regression:
\[
\bar{r} \in \arg\min_r \sum_{s=1}^{q} \sum_{c=1}^{m} (z^s(c) - \tilde{h}(c, x^s, r))^2. \tag{3.25}
\]
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 (3.25): \(m - 1\) “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 described in Section 3.1.2 can be applied to the solution of this problem.
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. (3.22)], and its data-trained version \( \tilde{c}(x, \bar{r}) \) [cf. Eq. (3.26)]. 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.

The regression problem (3.25) approximates the minimization of the expected value (3.24), 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

\[
\text{Estimated Object Category} = \tilde{c}(x, \bar{r}) \in \arg \max_{c=1,\ldots,m} \tilde{h}(c, x, \bar{r}), \quad (3.26)
\]

which approximates the MAP rule (3.22); cf. Fig. 3.4.1.

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 \( \tilde{h}(u, x_k, r_k) \) for stage \( k \) [once the values \( \tilde{h}(u, x_k, r_k) \) 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 \( \tilde{\mu}_k(x_k, r_k) \) that uses at state \( x_k \) the control of maximal probability at that state; cf. Eq. (3.26).

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:

\[
\bar{r} \in \arg \min_r \sum_{s=1}^q (z^s - \tilde{h}(x^s, r))^2.
\]

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

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

In the context of DP, 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.

### 3.5 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 and a starting point for further exploration. The literature on the subject is vast, and 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. Research in this area has intensified following the increased popularity of deep neural networks.

(2) The training of neural networks as well as linear architectures. Research on both of these issues has been extensive and is continuing. An important direction, not discussed here, is how to take advantage of distributed computation, particularly in conjunction with partitioned architectures (see [BeT96], Section 3.1.3, [BeY10], [BeY12]).

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 [GBC16]). For discussions that are focused outside the neural network area, see Bertsekas and Tsitsiklis [BeT96], Keller, Mannor, and Precup [KMP06], Jung and Polani [JnP07], Bertsekas and Yu [BeY09], and Bhatnagar, Borkar, and Prashanth [BBP13]. Moreover, there has been considerable research on the optimal feature selection within given parametric classes (see Menache, Mannor, and Shimkin [MMS05], Yu and Bertsekas [YuB09], Busoniu et al. [BBD10], and Di Castro and Mannor [DiM10]).

Incremental algorithms are supported by substantial theoretical analysis, which addresses issues of convergence, rate of convergence, stepsize 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 [Ned11], and then by several other authors: Bertsekas [Ber11c], Wang and Bertsekas [WaB14], [WabB16], Bianchi [Bia16], Iusem, Jofre, and Thompson [IJT18]. It is beyond our scope to cover this analysis. The author’s surveys [Ber10] and [Ber15b], and convex optimization and nonlinear programming textbooks [Ber15a], [Ber16a], collectively contain an extensive account of incremental methods, including the Kaczmarz, and incremental gradient, subgradient, aggregated gradient, Newton, Gauss-Newton, and extended Kalman filtering methods, and give many references. We also refer to the book [BeT96] and paper [BeT00] by Bertsekas and Tsitsiklis, and the survey by Bottou, Curtis, and Nocedal [BCN18] for a 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 research question relates to the role of overparametrization in the success of deep neural networks. With more weights than training data, the training problem has infinitely many solutions. The question then is how to select a solution that works well on test data (i.e., data outside the training set); see [ZBH16], [BMM18], [BRT18], [SJL18], [BLL19], [HMR19], [MVS19].

Section 3.3: Fitted value iteration has a long history; it has been 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). Advantage updating was proposed by Baird [Bai93], [Bai94], and is discussed in Section 6.6 of the book [BeT96].

Section 3.4: Classification (sometimes called “pattern classification” or “pattern recognition”) is a major subject in machine learning, for which there are many approaches and an extensive literature; see e.g. the textbooks by Bishop [Bis95], [Bis06], and Duda, Hart, and Stork [DHS12]. 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 our subsequent discussion of rollout-based policy iteration for infinite horizon problems in Chapter 4. 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.