Reinforcement Learning and Optimal Control

by

Dimitri P. Bertsekas

Massachusetts Institute of Technology

Chapter 3
Parametric Approximation

DRAFT

This is Chapter 3 of the draft textbook "Reinforcement Learning and Optimal Control." The chapter represents "work in progress," and it will be periodically updated. It more than likely contains errors (hopefully not serious ones). Furthermore, its references to the literature are incomplete. Your comments and suggestions to the author at dimitrib@mit.edu are welcome. The date of last revision is given below.

The date of last revision is given below. (A "revision" is any version of the chapter that involves the addition or the deletion of at least one paragraph or mathematically significant equation.)

May 16, 2019

3

Parametric Approximation

Contents

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 problem approximation and rollout. In this chapter we discuss an alternative approach, whereby $\tilde{J}_k$ 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 methods used for parametric approximation in value space can also be used for approximation in policy space, as we will discuss in Section 3.5.

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.

The simplest training approach 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). More 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 on this latter approach.

3.1.1 Linear and Nonlinear 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

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

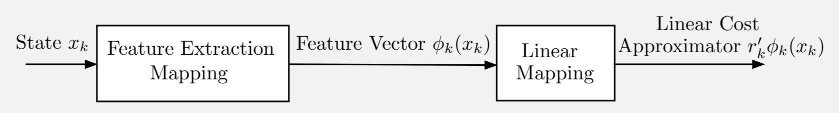
State x_k
Feature Extraction Mapping
Feature Vector $\phi_k(x_k)$
Linear Mapping
Linear Cost Approximator $r_k'\phi_k(x_k)$

Figure 3.1.1 The structure of a linear feature-based architecture. We use a feature extraction mapping to generate an input $\phi_k(x_k)$ to a linear mapping defined by a weight vector 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 discussion of architectures, and we refer to the specialized literature (e.g., Bertsekas and Tsitsiklis [BeT96], Bishop [Bis95], Haykin [Hay08], Sutton and Barto [SuB18]), and the author's [Ber12], Section 6.1.1, for more detailed presentations.

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.,

$$\tilde{J}_k(x_k, r_k) = \hat{J}_k\big(\phi_k(x_k), r_k\big) = \sum_{\ell=1}^{m_k} r_{\ell,k}\phi_{\ell,k}(x_k) = r_k'\phi_k(x_k), \tag{3.1}$$

where $r_{\ell,k}$ and $\phi_{\ell,k}(x_k)$ are the ℓth 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 $\Re^{m_k}$ (a prime denoting 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.

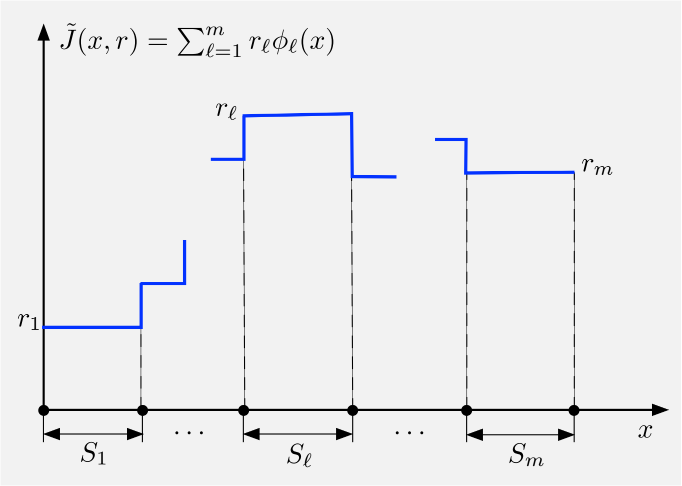

Figure 3.1.2 Illustration of a piecewise constant architecture. The state space is partitioned into subsets $S_1,\ldots,S_m$, with each subset S_ℓ defining the feature

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

with its own weight r_ℓ.

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 *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 ℓth feature be defined by membership to the set S_ℓ, 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

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

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_ℓ for all states within the set S_ℓ; 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 $\tilde{J}(x,r)$ 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 an 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 ith 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

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

A linear approximation architecture that uses these functions is given by

$$\tilde{J}(x,r)=r_0+\sum_{i=1}^{n} r_ix^i+\sum_{i=1}^{n}\sum_{j=1}^{n} r_{ij}x^ix^j,$$

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

$$\phi(x)=\big(\phi_1(x),\ldots,\phi_m(x)\big)',$$

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

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

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. The following are two examples involving games.

Example 3.1.3 (Tetris)

Let us revisit the game of tetris, which we discussed in Example 1.3.4. We can model the problem of finding an optimal playing strategy as a finite horizon problem with a very large horizon.

In Example 1.3.4 we viewed as state 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. However, the DP algorithm can be executed over the space of x only, since y is an uncontrollable state component. 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. For the most part, however, the computer chess training methodology has been qualitatively different from the parametric approximation methods that we consider in this book.

In particular, with few exceptions, the training of chess architectures has been done with ad hoc hand-tuning techniques (as opposed to some form of optimization). Moreover, the features have traditionally been hand-crafted

† 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 described here, but other sets of features have also been used; see [ThS09] and the discussion in [GGS13].

‡ We have not discussed DP for minimax problems and two-player games, but the ideas of approximation in value space apply to these contexts as well; see [Ber17] for a discussion that is focused on computer chess.

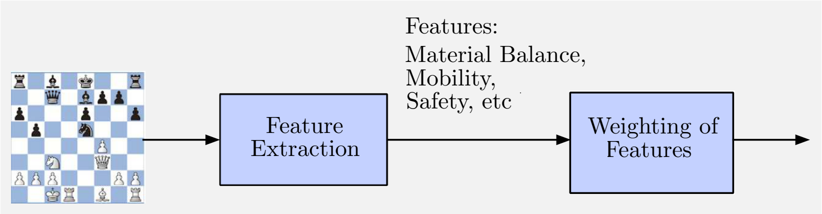

Figure 3.1.3 A feature-based architecture for computer chess.

based on chess-specific knowledge (as opposed to automatically generated through a neural network or some other method). Indeed, it has been argued that the success of chess programs in outperforming the best humans, can be more properly attributed to the brute-force calculating power of modern computers, which provides long and accurate lookahead, than to the ability of simulation-based algorithmic approaches, which can learn powerful playing strategies that humans have difficulty conceiving or executing. This assessment is changing rapidly following the impressive success of the AlphaZero chess program (Silver et al. [SHS17]). Still, however, computer chess with its use of long lookahead, provides an approximate DP paradigm that may be better suited for practical contexts where a lot of computational power is available, but innovative and sophisticated algorithms are hard to construct.

The fundamental principles on which all computer chess programs are based were laid out by Shannon [Sha50]. He 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 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 hand-crafted feature-based architecture for evaluating a chess position/state (cf. Fig. 3.1.3). In most computer chess programs, the features are weighted linearly, i.e., the architecture $\tilde J(x,r)$ 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. It will be discussed further later.

The 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

$$\phi_\ell(x) = e^{-\frac{\|x - x(\ell)\|^2}{2\sigma^2}}, \qquad \ell = 1, \ldots, m, \tag{3.2}$$

where σ is a scalar "variance" parameter. This type of function is known as a *radial basis function kernel*. It is concentrated around the state $x(\ell)$, and it is weighed with a scalar weight r_ℓ 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. (3.2), 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 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

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

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

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

where μ_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 $p_k(I_k)$ as a sufficient statistic (this is the conditional probability distribution of x_k given I_k; a more detailed discussion of these possibilities can be found in [Ber17], Chapter 4).

Since the sufficient statistic contains all the relevant information for control purposes, an idea that suggests itself is to introduce features of the 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 $p_k(I_k)$ is used as a sufficient statistic, examples of good features may be a point estimate based on $p_k(I_k)$, the variance of this estimate, and other quantities that can be simply extracted from $p_k(I_k)$ [e.g., simplified or certainty equivalent versions of $p_k(I_k)$].

Unfortunately, in many situations an adequate set of features is not known, 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 $\phi_k(x_k)$ and the parameter vectors r_k that weigh them.

3.1.2 Training of Linear and Nonlinear Architectures

In what follows in this section we discuss the process of choosing the parameter vector r of a parametric architecture $\tilde{J}(x,r)$. This is generally referred to as *training*. 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,β^s), $s=1,\ldots,q$, called the *training set*, is collected and r is determined by solving the problem

$$\min_r \sum_{s=1}^{q} \big(\tilde{J}(x^s,r)-\beta^s\big)^2. \tag{3.3}$$

Thus r is chosen to minimize the error between sample costs β^s and the architecture-predicted costs $\tilde{J}(x^s,r)$ in a least squares sense. Here typically, there is some "target" cost function J that we aim to approximate with $\tilde{J}(\cdot,r)$, and the sample cost β^s is the value $J(x^s)$ plus perhaps some error or "noise".

The cost function of the training problem (3.3) 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

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

the problem becomes

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

By setting the gradient of the quadratic objective to 0, we have the linear equation

$$\sum_{s=1}^{q} \phi(x^s)\big(r'\phi(x^s)-\beta^s\big)=0,$$

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

$$\hat{r}=\left(\sum_{s=1}^{q} \phi(x^s)\phi(x^s)'\right)^{-1}\sum_{s=1}^{q} \phi(x^s)\beta^s. \tag{3.4}$$

(Since sometimes the inverse above may 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.)

Thus a linear architecture has the important advantage that the training problem can be solved exactly and conveniently with the formula (3.4) (of course it may be solved by any other algorithm that is suitable for linear least squares problems). 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 as we will discuss in Section 3.2.

3.1.3 Incremental Gradient and Newton Methods

We will now digress to discuss special methods for solution of the least squares training problem (3.3), assuming a parametric architecture that is differentiable in the parameter vector. This methodology is 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.

Incremental methods have a rich theory, and our presentation in this section is brief, focusing primarily on implementation and intuition. Some surveys and book references are provided at the end of the chapter, which include a more detailed treatment. In particular, the neuro-dynamic programming book [BeT96] contains convergence analysis for both deterministic and stochastic training problems, while the author's nonlinear programming book [Ber16a] contains an extensive discussion of incremental methods. Since, we want to cover problems that are more general than the specific least squares training problem (3.3), we will adopt a broader formulation and notation that are standard in nonlinear programming.

Incremental Gradient Methods

We view the training problem (3.3) as a special case of the minimization of a sum of component functions

$$f(y) = \sum_{i=1}^{m} f_i(y), \tag{3.5}$$

where each f_i is a differentiable scalar function of the n-dimensional 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 least squares terms $\big(\tilde J(x^s, r) - \beta^s\big)^2$ in the training problem (3.3) with the generic terms $f_i(y)$.

The (ordinary) gradient method for problem (3.5) 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†

$$y^{k+1} = y^k - \gamma^k \nabla f(y^k) = y^k - \gamma^k \sum_{i=1}^{m} \nabla f_i(y^k), \tag{3.6}$$

where γ^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

$$y^{k+1} = y^k - \gamma^k \nabla f_{i_k}(y^k), \tag{3.7}$$

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 (3.6), 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 modulo 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.

(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 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.

† We use standard calculus notation for gradients; see, e.g., [Ber16a], Appendix A. In particular, $\nabla f(y)$ denotes the n dimensional 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 vector y.

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 (3.6) can be shown to converge with a constant stepsize under reasonable assumptions, see e.g., [Ber16a], 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 3.1.7

Assume that y is a scalar, and that the problem is

$$\begin{aligned} &\text{minimize} \quad f(y) = \tfrac{1}{2}\sum_{i=1}^{m}(c_i y - b_i)^2 \\ &\text{subject to} \quad y \in \Re, \end{aligned}$$

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

$$y^* = \frac{\sum_{i=1}^{m} c_i b_i}{\sum_{i=1}^{m} c_i^2}.$$

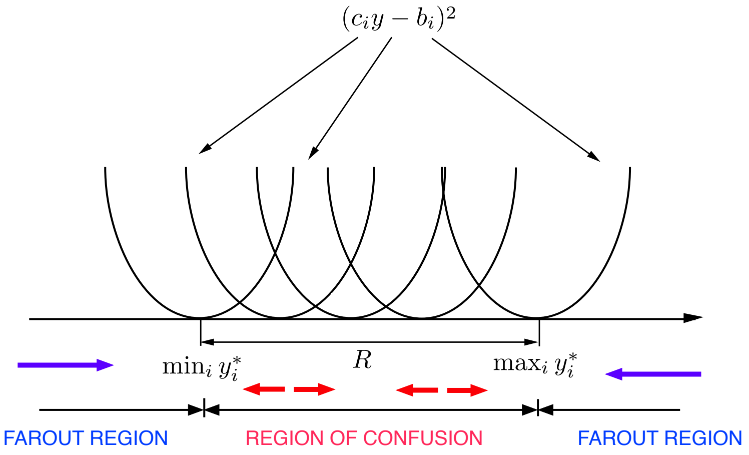

Figure 3.1.4. Illustrating the advantage of incrementalism when far from the optimal solution. The region of component minima

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

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 $(c_iy - b_i)^2$, 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.

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

$$\nabla f_i(y) = c_i(c_iy - b_i)$$

has the same sign as $\nabla f(y)$ (see Fig. 3.1.4). As a result, when outside the region R, the incremental gradient method

$$y^{k+1} = y^k - \gamma^k c_{i_k}(c_{i_k}y^k - b_{i_k})$$

approaches y^* at each step, provided the stepsize γ^k is small enough. In fact it can be verified that it is sufficient that

$$\gamma^k \le \min_i \frac{1}{c_i^2}.$$

However, for y *inside* the region R, the ith step of a cycle of the incremental gradient method need not make progress. It will approach y^* (for small enough stepsize γ^k) only if the current point y^k does not lie in the interval connecting y_i^* and y^*. 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 γ 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).

Example 3.1.8:

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

$$f_i(y) = a_i e^{b_i y}, \qquad y \in \Re,$$

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

$$I^+ = \{i \mid b_i > 0\}, \qquad I^- = \{i \mid b_i < 0\},$$

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 $y^k >> y^*$, 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 ith component f_i by the incremental gradient method can make progress towards the solution in the region where the component gradient $\nabla f_{i_k}(y^k)$ 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 γ^k plays an important role in the performance of incremental gradient methods. On close examination, it turns out that the direction used by the method differs from the gradient direction by an error that is proportional to the stepsize, and for this reason a diminishing stepsize is essential for convergence to a local minimum of f (convergence to a local minimum is the best we hope for since the cost function may not be convex).

However, it turns out that for the cyclic rule without reshuffling a peculiar form of convergence also typically occurs for a constant but sufficiently small stepsize. In this case, the iterates converge to a "limit cycle", whereby the ith iterates within the cycles converge to a different limit than the jth iterates for $i \neq j$. The sequence $\{y^k\}$ of the iterates obtained at the end of cycles converges, except that the limit obtained *need not* be a minimum of f, even when f is convex. The limit tends to be close to the minimum point when the constant stepsize is small (see Section 2.4 of [Ber16a] for analysis and examples). 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 γ, β_1, and β_2 are some positive scalars. There are also variants of the incremental gradient 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 will be discussed later in this section.

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], [GOP15a]).

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

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

where γ 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. (3.5). The neural network literature includes a number of practically useful scaling proposals, some of which have been incorporated in publicly and commercially available software. Also, later in this section, when we discuss incremental Newton methods, we will provide a type of diagonal scaling that uses second derivatives and is well suited to the additive character of f.

Stochastic Gradient Descent

Incremental gradient methods are related to methods that aim to minimize an expected value

$$f(y) = E\big\{F(y, w)\big\},$$

where w is a random variable, and $F(\cdot,w) : \Re^n \mapsto \Re$ is a differentiable function for each possible value of w. The *stochastic gradient method* for minimizing f is given by

$$y^{k+1} = y^k - \gamma^k \nabla_y F(y^k, w^k), \tag{3.8}$$

where w^k is a sample of w and $\nabla_y F$ denotes gradient of F with respect to y. This method has a rich theory and a long history, and it is strongly related to the classical algorithmic field of *stochastic approximation*; see the books [BeT96], [KuY03], [Spa03], [Mey07], [Bor08], [BPP13]. The method is also often referred to as *stochastic gradient descent*, particularly in the context of machine learning applications.

If we view the expected value cost $E\big\{F(y,w)\big\}$ as a weighted sum of cost function components, we see that the stochastic gradient method (3.8) is related to the incremental gradient method

$$y^{k+1} = y^k - \gamma^k \nabla f_{i_k}(y^k) \tag{3.9}$$

for minimizing a finite sum $\sum_{i=1}^m f_i$, when randomization is used for component selection. An important difference is that the former method involves stochastic sampling of cost components $F(y,w)$ from a possibly infinite population, under some probabilistic assumptions, while in the latter the set of cost components f_i is predetermined and finite. However, it is possible to view the incremental gradient method (3.9), with uniform randomized selection of the component function f_i (i.e., with i_k chosen to be any one of the indexes $1,\ldots,m$, with equal probability $1/m$, and independently of preceding choices), as a stochastic gradient method.

Despite the apparent similarity of the incremental and the stochastic gradient methods, the view that the problem

$$\begin{aligned} &\text{minimize} \quad f(y) = \sum_{i=1}^m f_i(y) \\ &\text{subject to} \ \ y \in \Re^n, \end{aligned} \tag{3.10}$$

can simply be treated as a special case of the problem

$$\begin{aligned} &\text{minimize} \quad f(y) = E\big\{F(y,w)\big\} \\ &\text{subject to} \ \ y \in \Re^n, \end{aligned}$$

is questionable.

One reason is that once we convert the finite sum problem to a stochastic problem, we preclude the use of methods that exploit the finite sum structure, such as the incremental aggregated gradient methods to be discussed in the next subsection. Another reason is that the finite-component problem (3.10) is often genuinely deterministic, and to view it as a stochastic problem at the outset may mask some of its important characteristics, such as the number m of cost components, or the sequence in which the components are ordered and processed. These characteristics may potentially be algorithmically exploited. For example, with insight into the problem's structure, one may be able to discover a special deterministic or partially randomized order of processing the component functions that is superior to a uniform randomized order. On the other hand analysis indicates that in the absence of problem-specific knowledge that can be exploited to select a favorable deterministic order, a uniform randomized order (each component f_i chosen with equal probability $1/m$ at each iteration, independently of preceding choices) sometimes has superior worst-case complexity; see [NeB00], [NeB01], [BNO03], [Ber15a], [WaB16].

Finally, let us note the popular hybrid technique, which reshuffles randomly the order of the cost components after each cycle. Practical experience and recent analysis [GOP15b] indicates that it has somewhat better performance to the uniform randomized order when m is large. One possible reason is that random reshuffling allocates exactly one computation slot to each component in an m-slot cycle, while uniform sampling allocates one computation slot to each component *on the average*. A nonzero variance in the number of slots that any fixed component gets within a cycle, may be detrimental to performance. While it seems difficult to establish this fact analytically, a justification is suggested by the view of the incremental gradient method as a gradient method with error in the computation of the gradient. The error has apparently greater variance in the uniform sampling method than in the random reshuffling method, and heuristically, if the variance of the error is larger, the direction of descent deteriorates, suggesting slower convergence.

Incremental Aggregated Gradient Methods

Another algorithm that is well suited for least squares training problems is the *incremental aggregated gradient method*, which has the form

$$y^{k+1} = y^k - \gamma^k \sum_{\ell=0}^{m-1} \nabla f_{i_{k-\ell}}(y^{k-\ell}), \tag{3.11}$$

where f_{i_k} is the new component function selected for iteration k.† In the most common version of the method the component indexes i_k are selected in a cyclic order $[i_k = (k \text{ modulo } m) + 1]$. Random selection of the index i_k has also been suggested.

From Eq. (3.11) it can be seen that the method computes the gradient incrementally, one component per iteration. However, in place of the single

† In the case where $k < m$, the summation in Eq. (3.11) should go up to $\ell = k$, and the stepsize should be replaced by a corresponding larger value.

component gradient $\nabla f_{i_k}(y^k)$, used in the incremental gradient method (3.7), it uses the sum of the component gradients computed in the past m iterations, which is an approximation to the total cost gradient $\nabla f(y^k)$.

The idea of the method is that by aggregating the component gradients one may be able to reduce the error between the true gradient $\nabla f(y^k)$ and the incrementally computed approximation used in Eq. (3.11), and thus attain a faster asymptotic convergence rate. Indeed, it turns out that under certain conditions the method exhibits a linear convergence rate, just like in the nonincremental gradient method, without incurring the cost of a full gradient evaluation at each iteration (a strongly convex cost function and with a sufficiently small constant stepsize are required for this; see [Ber16a], Section 2.4.2, and the references quoted there). This is in contrast with the incremental gradient method (3.7), for which a linear convergence rate can be achieved only at the expense of asymptotic error, as discussed earlier.

A disadvantage of the aggregated gradient method (3.11) is that it requires that the most recent component gradients be kept in memory, so that when a component gradient is reevaluated at a new point, the preceding gradient of the same component is discarded from the sum of gradients of Eq. (3.11). There have been alternative implementations that ameliorate this memory issue, by recalculating the full gradient periodically and replacing an old component gradient by a new one; we refer to the specialized literature on the subject. Another potential drawback of the aggregated gradient method is that for a large number of terms m, one hopes to converge within the first cycle through the components f_i, thereby reducing the effect of aggregating the gradients of the components.

Incremental Newton Methods

We will now consider an incremental version of Newton's method for unconstrained minimization of an additive cost function of the form

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

where the functions f_i are convex and twice continuously differentiable.†

The ordinary Newton's method, at the current iterate y^k, obtains the next iterate y^{k+1} by minimizing over y the quadratic approximation/second

† We will denote by $\nabla^2 f(y)$ the $n \times n$ Hessian matrix of f at y, i.e., the matrix whose (i,j)th component is the second partial derivative $\partial^2 f(y)/\partial y^i \partial y^j$. A beneficial consequence of assuming convexity of f_i is that the Hessian matrices $\nabla^2 f_i(y)$ are positive semidefinite, which facilitates the implementation of the algorithms to be described. On the other hand, the algorithmic ideas of this section may also be adapted for the case where f_i are nonconvex.

order expansion

$$\tilde{f}(y; y^k) = \nabla f(y^k)'(y - y^k) + \tfrac{1}{2}(y - y^k)'\nabla^2 f(y^k)(y - y^k),$$

of f at y^k. Similarly, the incremental form of Newton's method minimizes a sum of quadratic approximations of components of the general form

$$\tilde{f}_i(y; \psi) = \nabla f_i(\psi)'(y - \psi) + \tfrac{1}{2}(y - \psi)'\nabla^2 f_i(\psi)(y - \psi), \qquad (3.12)$$

as we will now explain.

As in the case of the incremental gradient method, we view an iteration as a cycle of m subiterations, each involving a single additional component f_i, and its gradient and Hessian at the current point within the cycle. In particular, if y^k is the vector obtained after k cycles, the vector y^{k+1} obtained after one more cycle is

$$y^{k+1} = \psi_{m,k},$$

where starting with $\psi_{0,k} = y^k$, we obtain $\psi_{m,k}$ after the m steps

$$\psi_{i,k} \in \arg\min_{y \in \Re^n} \sum_{\ell=1}^{i} \tilde{f}_\ell(y; \psi_{\ell-1,k}), \qquad i = 1, \ldots, m, \qquad (3.13)$$

where $\tilde{f}_\ell$ is defined as the quadratic approximation (3.12). If all the functions f_i are quadratic, it can be seen that the method finds the solution in a single cycle.† The reason is that when f_i is quadratic, each $f_i(y)$ differs from $\tilde{f}_i(y; \psi)$ by a constant, which does not depend on y. Thus the difference

$$\sum_{i=1}^{m} f_i(y) - \sum_{i=1}^{m} \tilde{f}_i(y; \psi_{i-1,k})$$

is a constant that is independent of y, and minimization of either sum in the above expression gives the same result.

It is important to note that the computations of Eq. (3.13) can be carried out efficiently. For simplicity, let as assume that $\tilde{f}_1(y; \psi)$ is a positive definite quadratic, so that for all i, $\psi_{i,k}$ is well defined as the unique

† Here we assume that the m quadratic minimizations (3.13) to generate $\psi_{m,k}$ have a solution. For this it is sufficient that the first Hessian matrix $\nabla^2 f_1(y^0)$ be positive definite, in which case there is a unique solution at every iteration. A simple possibility to deal with this requirement is to add to f_1 a small positive regularization term, such as $\frac{\epsilon}{2}\|y - y^0\|^2$. A more sound possibility is to lump together several of the component functions (enough to ensure that the sum of their quadratic approximations at y^0 is positive definite), and to use them in place of f_1. This is generally a good idea and leads to smoother initialization, as it ensures a relatively stable behavior of the algorithm for the initial iterations.

solution of the minimization problem in Eq. (3.13). We will show that the incremental Newton method (3.13) can be implemented in terms of the incremental update formula

$$\psi_{i,k} = \psi_{i-1,k} - D_{i,k}\nabla f_i(\psi_{i-1,k}), \tag{3.14}$$

where $D_{i,k}$ is given by

$$D_{i,k} = \left(\sum_{\ell=1}^{i} \nabla^2 f_\ell(\psi_{\ell-1,k})\right)^{-1}, \tag{3.15}$$

and (assuming existence of the required inverses) is generated iteratively as

$$D_{i,k} = \Big(D_{i-1,k}^{-1} + \nabla^2 f_i(\psi_{i-1,k})\Big)^{-1}. \tag{3.16}$$

Indeed, from the definition of the method (3.13), the quadratic function $\sum_{\ell=1}^{i-1} \tilde f_\ell(y;\psi_{\ell-1,k})$ is minimized by $\psi_{i-1,k}$ and its Hessian matrix is $D_{i-1,k}^{-1}$, so we have

$$\sum_{\ell=1}^{i-1} \tilde f_\ell(y;\psi_{\ell-1,k}) = \tfrac{1}{2}(y-\psi_{i-1,k})'D_{i-1,k}^{-1}(y-\psi_{i-1,k}) + \text{constant}.$$

Thus, by adding $\tilde f_i(y;\psi_{i-1,k})$ to both sides of this expression, we obtain

$$\begin{aligned}\sum_{\ell=1}^{i}\tilde f_\ell(y;\psi_{\ell-1,k}) &= \tfrac{1}{2}(y-\psi_{i-1,k})'D_{i-1,k}^{-1}(y-\psi_{i-1,k}) + \text{constant}\\ &\quad + \tfrac{1}{2}(y-\psi_{i-1,k})'\nabla^2 f_i(\psi_{i-1,k})(y-\psi_{i-1,k}) + \nabla f_i(\psi_{i-1,k})'(y-\psi_{i-1,k}).\end{aligned}$$

Since by definition $\psi_{i,k}$ minimizes this function, we obtain Eqs. (3.14)-(3.16).

The recursion (3.16) for the matrix $D_{i,k}$ can often be efficiently implemented by using convenient formulas for the inverse of the sum of two matrices. In particular, if f_i is given by

$$f_i(y) = h_i(a_i'y - b_i),$$

for some twice differentiable convex function $h_i : \Re \mapsto \Re$, vector a_i, and scalar b_i, we have

$$\nabla^2 f_i(\psi_{i-1,k}) = \nabla^2 h_i(a_i'\psi_{i-1,k} - b_i)\, a_i a_i',$$

and the recursion (3.16) can be written as

$$D_{i,k} = D_{i-1,k} - \frac{D_{i-1,k}a_i a_i' D_{i-1,k}}{\nabla^2 h_i(a_i'\psi_{i-1,k} - b_i)^{-1} + a_i' D_{i-1,k} a_i};$$

this is the well-known Sherman-Morrison formula for the inverse of the sum of an invertible matrix and a rank-one matrix.

We have considered so far a single cycle of the incremental Newton method. Similar to the case of the incremental gradient method, we may cycle through the component functions multiple times. In particular, we may apply the incremental Newton method to the extended set of components

$$f_1, f_2, \ldots, f_m,\ f_1, f_2, \ldots, f_m,\ f_1, f_2, \ldots. \tag{3.17}$$

The resulting method asymptotically resembles an incremental gradient method with diminishing stepsize of the type described earlier. Indeed, from Eq. (3.15), the matrix $D_{i,k}$ diminishes roughly in proportion to $1/k$. From this it follows that the asymptotic convergence properties of the incremental Newton method are similar to those of an incremental gradient method with diminishing stepsize of order $O(1/k)$. Thus its convergence rate is slower than linear.

To accelerate the convergence of the method one may employ a form of restart, so that $D_{i,k}$ does not converge to 0. For example $D_{i,k}$ may be reinitialized and increased in size at the beginning of each cycle. For problems where f has a unique nonsingular minimum y^* [one for which $\nabla^2 f(y^*)$ is nonsingular], one may design incremental Newton schemes with restart that converge linearly to within a neighborhood of y^* (and even superlinearly if y^* is also a minimum of all the functions f_i, so there is no region of confusion). Alternatively, the update formula (3.16) may be modified to

$$D_{i,k} = \Big(\beta_k D_{i-1,k}^{-1} + \nabla^2 f_\ell(\psi_{i,k})\Big)^{-1}, \tag{3.18}$$

by introducing a parameter $\beta_k \in (0,1)$, which can be used to accelerate the practical convergence rate of the method; cf. the discussion of the incremental Gauss-Newton methods.

Incremental Newton Method with Diagonal Approximation

Generally, with proper implementation, the incremental Newton method is often substantially faster than the incremental gradient method, in terms of numbers of iterations (there are theoretical results suggesting this property for stochastic versions of the two methods; see the end-of-chapter references). However, in addition to computation of second derivatives, the incremental Newton method involves greater overhead per iteration due to the matrix-vector calculations in Eqs. (3.14), (3.16), and (3.18), so it is suitable only for problems where n, the dimension of y, is relatively small.

A way to remedy in part this difficulty is to approximate $\nabla^2 f_i(\psi_{i,k})$ by a diagonal matrix, and recursively update a diagonal approximation of $D_{i,k}$ using Eqs. (3.16) or (3.18). In particular, one may set to 0 the off-diagonal components of $\nabla^2 f_i(\psi_{i,k})$. In this case, the iteration (3.14) becomes a diagonally scaled version of the incremental gradient method, and involves comparable overhead per iteration (assuming the required diagonal second derivatives are easily computed or approximated). As an additional scaling option, one may multiply the diagonal components with a stepsize parameter that is close to 1 and add a small positive constant (to bound them away from 0). Ordinarily this method is easily implementable, and requires little experimentation with stepsize selection.

Example 3.1.9: (Diagonalized Newton Method for Linear Least Squares)

Consider the problem of minimizing the sum of squares of linear scalar functions:

$$\begin{aligned} &\text{minimize} \quad f(y) = \tfrac{1}{2}\sum_{i=1}^{m}(c_i'y - b_i)^2 \\ &\text{subject to} \quad y \in \Re^n, \end{aligned}$$

where for all i, c_i is a given nonzero vector in $\Re^n$ and b_i is a given scalar. In this example, we will assume that we apply the incremental Newton method to the extended set of components

$$f_1, f_2, \ldots, f_m,\ f_1, f_2, \ldots, f_m,\ f_1, f_2, \ldots,$$

c.f. Eq. (3.17).

The inverse Hessian matrix for the kth cycle is given by

$$D_{i,k} = \left((k-1)\sum_{\ell=1}^{m} c_\ell c_\ell' + \sum_{\ell=1}^{i} c_\ell c_\ell' \right)^{-1},$$

and its jth diagonal term (the one to multiply the partial derivative $\partial f_i / \partial x^j$ within cycle $k+1$) is

$$\frac{1}{(k-1)\sum_{\ell=1}^{m}(c_\ell^j)^2 + \sum_{\ell=1}^{i}(c_\ell^j)^2}. \tag{3.19}$$

The diagonalized version of the incremental Newton method is given for each of the n components of $\psi_{i,k}$ by

$$\psi_{i,k}^j = \psi_{i-1,k}^j - \frac{c_i'\psi_{i-1,k} - b_i}{(k-1)\sum_{\ell=1}^{m}(c_\ell^j)^2 + \sum_{\ell=1}^{i}(c_\ell^j)^2} c_i^j, \qquad j = 1, \ldots, n,$$

where $\psi_{i,k}^j$ and c_i^j are the jth components of the vectors $\psi_{i,k}$ and c_i, respectively. Contrary to the incremental Newton iteration, it does not involve matrix inversion. Note that the diagonal scaling term (3.19) tends to 0 as k increases, for each component j. As noted earlier, this diagonal term may be multiplied by a constant that is less or equal to 1 for a better guarantee of asymptotic convergence.

In an alternative implementation of the diagonal Hessian approximation idea, we reinitialize the inverse Hessian matrix to 0 (or to small multiple of the identity) at the beginning of each cycle. Then instead of (3.19), the jth diagonal term of the inverse Hessian will be

$$\frac{1}{\sum_{\ell=1}^{i}(c_\ell^j)^2},$$

and will not converge to 0. It may be modified by multiplication with a suitable diminishing scalar for practical purposes.

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. We focus here 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 $\tilde J(x, v, r)$ of the form

$$\tilde J(x, v, r) = r'\phi(x, v) \tag{3.20}$$

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, β^s), $s = 1, \ldots, q$, and to find a function $\tilde J(x, v, r)$ of the form (3.20) that matches the training set in a least squares sense, i.e., (v, r) minimizes

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

We postpone for later the question of how the training pairs (x^s, β^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.

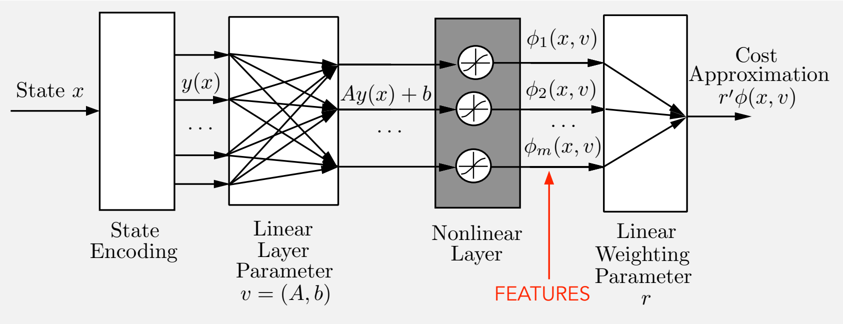

Figure 3.2.1 A perceptron consisting of a linear layer and a nonlinear layer. It provides a way to compute features of the state, which can be used for cost function approximation. The state x is encoded as a vector of numerical values $y(x)$, which is then transformed linearly as $Ay(x) + b$ in the linear layer. The m scalar output components of the linear layer, become the inputs to nonlinear functions that produce the m scalars $\phi_\ell(x, v) = \sigma\big((Ay(x) + b)_\ell\big)$, which can be viewed as features that are in turn linearly weighted with parameters r_ℓ.

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 (3.20) 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 $\Re^m$.† 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,

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

becomes the input to a nonlinear differentiable function σ that maps scalars to scalars. Typically σ 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.

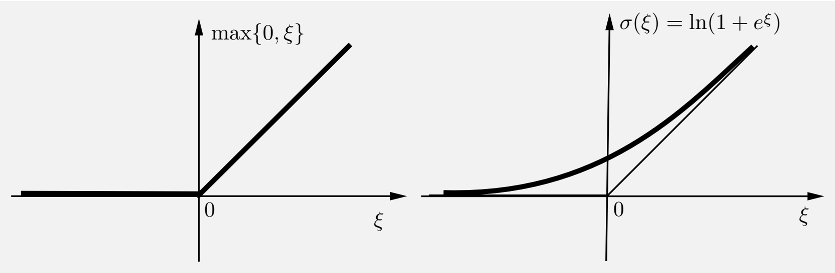

Figure 3.2.2 The rectified linear unit $\sigma(\xi) = \ln(1 + e^{\xi})$. It is the function $\max\{0, \xi\}$ with its corner "smoothed out." Its derivative is $\sigma'(\xi) = e^{\xi}/(1 + e^{\xi})$, and approaches 0 and 1 as $\xi \to -\infty$ and $\xi \to \infty$, respectively.

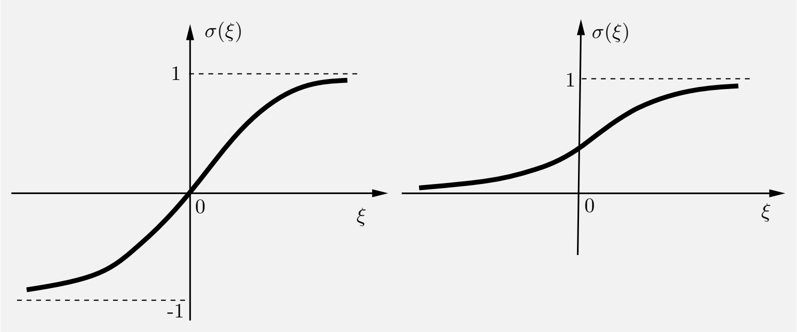

Figure 3.2.3 Some examples of sigmoid functions. The hyperbolic tangent function is on the left, while the logistic function is on the right.

which is simply the function $\max\{0, \xi\}$, approximated by a differentiable function σ 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 *sigmoids*, 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 σ (except for differentiability), and simply refer to it as a "nonlinear unit" and to the corresponding layer as a "nonlinear layer."

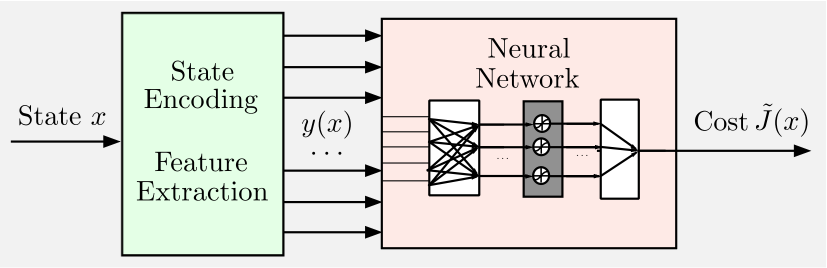

Figure 3.2.4 Nonlinear architecture with a view of the state encoding process as a feature extraction mapping preceding the neural network.

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

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

One possible interpretation is to view $\phi_\ell(x,v)$ as features of x, which are linearly combined using weights r_ℓ, $\ell = 1,\ldots,m$, to produce the final output

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

Note that each value $\phi_\ell(x,v)$ depends on just the ℓth row of A and the ℓth 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.

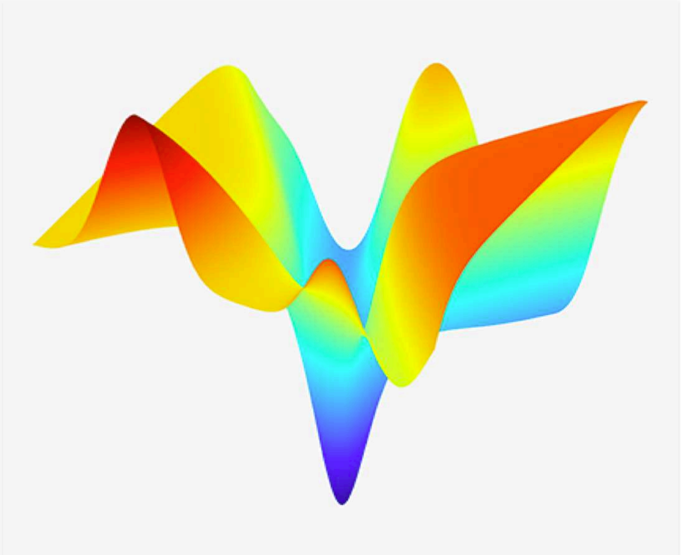

Figure 3.2.5 Illustration of the graph of the cost function of a two-dimensional neural network training problem. The cost function is nonconvex, and it becomes flat for large values of the weights. In particular, the cost tends asymptotically to a constant as the vector of weights is changed towards ∞ along rays emanating from the origin. The graph in the figure corresponds to an extremely simple training problem: a single input, a single nonlinear unit, two weights, and five input-output data pairs (corresponding to the five "petals" of the graph).

3.2.1 Training of Neural Networks

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

$$\min_{A,b,r} \sum_{s=1}^{q} \left(\sum_{\ell=1}^{m} r_\ell \sigma \left(\big(Ay(x^s) + b\big)_\ell \right) - \beta^s \right)^2 . \tag{3.21}$$

Note that the cost function of this problem is generally nonconvex, so there may exist multiple local minima. In fact the graph of cost function of the above training problem can be very complicated; see Fig. 3.2.5, which illustrates a very simple special case.

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 refers to a situation where a neural network model matches the training data very well but does not do as well on new data. This is a well known difficulty in machine learning, which may occur when the number of parameters of the neural network is relatively large (roughly comparable to the size of the training set). We refer to machine learning and neural network textbooks for a discussion of algorithmic questions regarding regularization and other issues that relate to the practical implementation of the training process. In any case, the training problem (3.21) is an unconstrained nonconvex differentiable optimization problem that can in principle be addressed with any of the standard gradient-type methods. Significantly, it is well-suited for the incremental methods discussed in Section 3.1.3 (a sense of this can be obtained by comparing the cost structure illustrated in Fig. 3.2.5 with the one of Fig. 3.1.4, which explains the advantage of incrementalism).

Let us now consider a few issues regarding the neural network formulation and training process just described:

(a) The first issue is to select a method to solve the training problem (3.21). 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 (3.21). 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] 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.

(b) Another important question is how well we can approximate the op-

† 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.

timal cost-to-go functions 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 $J : S \mapsto \Re$, 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].

(c) 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 in Section 3.2.2.

What are the Features that can be Produced by Neural Networks?

The short answer is that just about any feature that can be of practical interest can be produced or be closely approximated by a neural network. What is needed is a single layer that consists of a sufficiently large number of nonlinear units, preceded and followed by a linear layer. This is a consequence of the universal approximation theorem. In particular it is not necessary to have more than one nonlinear layer (although it is possible that fewer nonlinear units may be needed with a deep neural network, involving more than one nonlinear layer).

To illustrate this fact, we will consider features of a scalar state variable x, and a neural network that uses the rectifier function

$$\sigma(\xi) = \max\{0, \xi\}$$

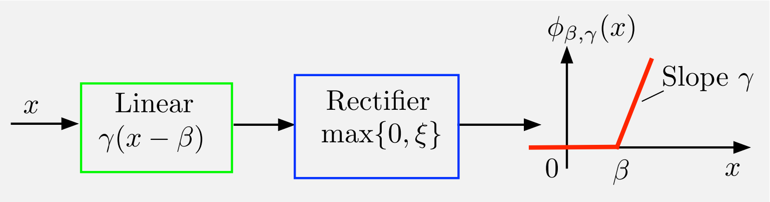

Figure 3.2.6 The feature

$$\phi_{\beta,\gamma}(x) = \max\big\{0, \gamma(x-\beta)\big\},$$

produced by a rectifier preceded by the linear function $L(x) = \gamma(x-\beta)$.

as the basic nonlinear unit. Thus a single rectifier preceded by a linear function

$$L(x) = \gamma(x-\beta),$$

where β and γ are scalars, produces the feature

$$\phi_{\beta,\gamma}(x) = \max\big\{0, \gamma(x-\beta)\big\}, \tag{3.22}$$

illustrated in Fig. 3.2.6.

We can now construct more complex features by adding or subtracting single rectifier features of the form (3.22). In particular, subtracting two rectifier functions with the same slope but two different horizontal shifts β_1 and β_2, we obtain the feature

$$\phi_{\beta_1,\beta_2,\gamma}(x) = \phi_{\beta_1,\gamma}(x) - \phi_{\beta_2,\gamma}(x)$$

shown in Fig. 3.2.7(a). By subtracting again two features of the preceding form, we obtain the "pulse" feature

$$\phi_{\beta_1,\beta_2,\beta_3,\beta_4,\gamma}(x) = \phi_{\beta_1,\beta_2,\gamma}(x) - \phi_{\beta_3,\beta_4,\gamma}(x), \tag{3.23}$$

shown in Fig. 3.2.7(b). The "pulse" feature can be used in turn as a fundamental block to approximate any desired feature by a linear combination of "pulses." This explains how neural networks produce features of any shape by using linear layers to precede and follow nonlinear layers, at least in the case of a scalar state x. In fact, the mechanism for feature formation just described can be extended to the case of a multidimensional x, and is at the heart of the universal approximation theorem and its proof (see Cybenko [Cyb89]).

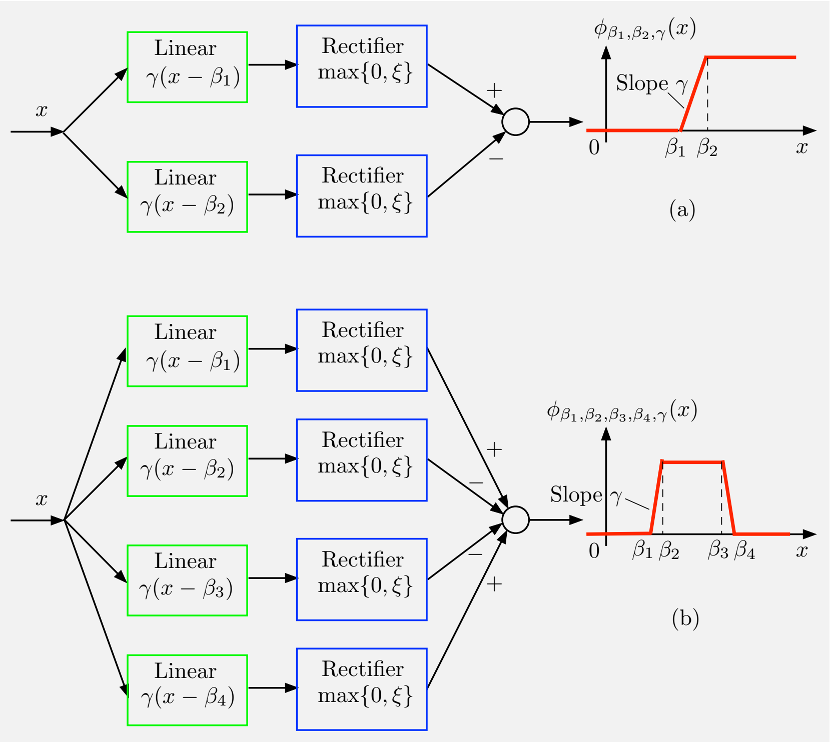

Figure 3.2.7 (a) Illustration of how the feature

$$\phi_{\beta_1,\beta_2,\gamma}(x) = \phi_{\beta_1,\gamma}(x) - \phi_{\beta_2,\gamma}(x)$$

is formed by a neural network of a two-rectifier nonlinear layer, preceded by a linear layer.

(b) Illustration of how the "pulse" feature

$$\phi_{\beta_1,\beta_2,\beta_3,\beta_4,\gamma}(x) = \phi_{\beta_1,\beta_2,\gamma}(x) - \phi_{\beta_3,\beta_4,\gamma}(x)$$

is formed by a neural network of a four-rectifier nonlinear layer, preceded by a linear layer.

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. 3.2.8. 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 $y(x)$.

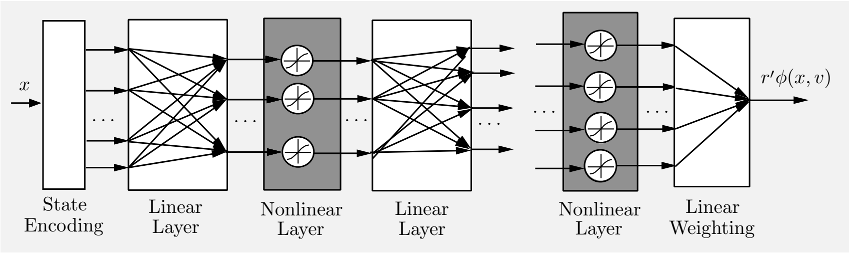

Figure 3.2.8 A deep neural network, with multiple layers. Each nonlinear layer constructs the inputs of the next linear layer.

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 provides a sequence of features, where each set of features in the sequence is a function of the preceding set of features in the sequence [except for the first set of features, which is a function of the encoding $y(x)$ of the state x]. Thus the network produces a hierarchy of features. In the context of specific applications, this hierarchical structure can be exploited in order to specialize the role of some of the layers and to enhance particular 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.

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, in conjunction with Monte Carlo tree search, has been an important factor for the success of the computer programs AlphaGo and AlphaZero, which perform better than the best humans in the games of Go and chess; see the papers by Silver et al. [SHM16], [SHS17]. By contrast, Tesauro's backgammon program and its descendants (cf. Section 2.4.2) do not require multiple nonlinear layers for good performance at present.

Training and Backpropagation

Let us now consider the training problem for multilayer networks. It has the form

$$\min_{v,r} \sum_{s=1}^{q} \left(\sum_{\ell=1}^{m} r_\ell \phi_\ell(x^s, v) - \beta^s \right)^2,$$

where v represents the collection of all the parameters of the linear layers, and $\phi_\ell(x,v)$ is the ℓth feature produced at the output of the final nonlinear layer. Various types of incremental gradient methods, which modify the weight vector in the direction opposite to the gradient of a single sample term

$$\left(\sum_{\ell=1}^{m} r_\ell \phi_\ell(x^s, v) - \beta^s \right)^2,$$

can also be applied here. They are the methods most commonly used in practice. An important fact is that the gradient with respect to v of each feature $\phi_\ell(x,v)$ can be efficiently calculated using a special procedure known as *backpropagation*. This is just an intelligent way of applying the chain rule of differentiation, as we will explain now.

Multilayer perceptrons can be represented compactly by introducing certain mappings to describe the linear and the nonlinear layers. In particular, let $L_1, \ldots, L_{m+1}$ denote the matrices representing the linear layers; that is, at the output of the 1st linear layer we obtain the vector $L_1 x$ and at the output of the kth linear layer ($k > 1$) we obtain $L_k \xi$, where ξ is the output of the preceding nonlinear layer. Similarly, let $\Sigma_1, \ldots, \Sigma_m$ denote the mappings representing the nonlinear layers; that is, when the input of the kth nonlinear layer ($k > 1$) is the vector y with components $y(j)$, we obtain at the output the vector $\Sigma_k y$ with components $\sigma\big(y(j)\big)$. The output of the multilayer perceptron is

$$F(L_1, \ldots, L_{m+1}, x) = L_{m+1} \Sigma_m L_m \cdots \Sigma_1 L_1 x.$$

The special nature of this formula has an important computational consequence: the gradient (with respect to the weights) of the squared error between the output and a desired output y,

$$E(L_1, \ldots, L_{m+1}) = \frac{1}{2}\big(y - F(L_1, \ldots, L_{m+1}, x)\big)^2,$$

can be efficiently calculated using a special procedure known as *backpropagation*, which is just an intelligent way of using the chain rule.† In particular, the partial derivative of the cost function $E(L_1, \ldots, L_{m+1})$ with

† The name *backpropagation* is used in several different ways in the neural networks literature. For example feedforward neural networks of the type shown in Fig. 3.2.8 are sometimes referred to as *backpropagation networks*. The somewhat abstract derivation of the backpropagation formulas given here comes from Section 3.1.1 of the book [BeT96].

respect to $L_k(i,j)$, the ijth component of the matrix L_k, is given by

$$\frac{\partial E(L_1,\ldots,L_{m+1})}{\partial L_k(i,j)} = -e' L_{m+1}\overline{\Sigma}_m L_m \cdots L_{k+1}\overline{\Sigma}_k I_{ij}\Sigma_{k-1}L_{k-1}\cdots \Sigma_1 L_1 x, \tag{3.24}$$

where e is the error vector

$$e = y - F(L_1,\ldots,L_{m+1},x),$$

$\overline{\Sigma}_n$, $n=1,\ldots,m$, is the diagonal matrix with diagonal terms equal to the derivatives of the nonlinear functions σ of the nth hidden layer evaluated at the appropriate points, and I_{ij} is the matrix obtained from L_k by setting all of its components to 0 except for the ijth component which is set to 1. This formula can be used to obtain efficiently all of the terms needed in the partial derivatives (3.24) of E using a two-step calculation:

(a) Use a forward pass through the network to calculate sequentially the outputs of the linear layers

$$L_1x, L_2\Sigma_1L_1x, \ldots, L_{m+1}\Sigma_mL_m\cdots\Sigma_1L_1x.$$

This is needed in order to obtain the points at which the derivatives in the matrices $\overline{\Sigma}_n$ are evaluated, and also in order to obtain the error vector $e = y - F(L_1,\ldots,L_{m+1},x)$.

(b) Use a backward pass through the network to calculate sequentially the terms

$$e'L_{m+1}\overline{\Sigma}_mL_m\cdots L_{k+1}\overline{\Sigma}_k$$

in the derivative formulas (3.24), starting with $e'L_{m+1}\overline{\Sigma}_m$, proceeding to $e'L_{m+1}\overline{\Sigma}_mL_m\overline{\Sigma}_{m-1}$, and continuing to $e'L_{m+1}\overline{\Sigma}_m\cdots L_2\overline{\Sigma}_1$.

As a final remark, we mention that the ability to simultaneously extract features and optimize their linear combination is not unique to neural networks. Other approaches that use a multilayer architecture have been proposed (see the survey by Schmidhuber [Sch15]), and they admit similar training procedures based on appropriately modified forms of backpropagation. An example of an alternative multilayer architecture approach is the Group Method for Data Handling (GMDH), which is principally based on the use of polynomial (rather than sigmoidal) nonlinearities. The GMDH approach was investigated extensively in the Soviet Union starting with the work of Ivakhnenko in the late 60s; see e.g., [Iva68]. It has been used in a large variety of applications, and its similarities with the neural network methodology have been noted (see the survey by Ivakhnenko [Iva71], and the large literature summary at the web site http://www.gmdh.net). Most of the GMDH research relates to inference-type problems, and there have not been any applications to approximate DP to date. However, this may be a fruitful area of investigation, since in some applications it may turn out that polynomial nonlinearities are more suitable than sigmoidal or rectified linear unit nonlinearities.

3.3 SEQUENTIAL DYNAMIC PROGRAMMING APPROXIMATION

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_k^s, $s = 1, \ldots, q$. 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

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

where x_k^s, $i = 1, \ldots, q$, are the sample states that have been generated for the kth 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

$$\beta_k^s = \min_{u \in U_k(x_k^s)} E\Big\{ g(x_k^s, u, w_k) + \tilde{J}_{k+1}\big(f_k(x_k^s, u, w_k), r_{k+1}\big) \Big\}, \tag{3.25}$$

so that r_k is obtained as

$$r_k \in \arg\min_r \sum_{s=1}^q \big(\tilde{J}_k(x_k^s, r) - \beta_k^s \big)^2. \tag{3.26}$$

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

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

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

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

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

the least squares problem (3.26) 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. (3.4). 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. 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 subsequent chapters.

3.4 Q-FACTOR PARAMETRIC APPROXIMATION

We will now consider an alternative form of approximation in value space. It involves approximation of the optimal Q-factors of state-control pairs (x_k, u_k) at time k, without an intermediate approximation of cost-to-go functions. The optimal Q-factors are defined by

$$Q_k^*(x_k, u_k) = E\Big\{ g_k(x_k, u_k, w_k) + J_{k+1}^*\big(f_k(x_k, u_k, w_k)\big) \Big\}, \quad k = 0, \ldots, N-1, \tag{3.27}$$

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

$$J_k^*(x_k) = \min_{u \in U_k(x_k)} Q_k^*(x_k, u_k), \tag{3.28}$$

and by using this equation, we can write Eq. (3.27) in the following equivalent form that relates Q_k^* with Q_{k+1}^*:

$$\begin{aligned} Q_k^*(x_k, u_k) = E\Big\{ & g_k(x_k, u_k, w_k) \\ & + \min_{u \in U_{k+1}(f_k(x_k,u_k,w_k))} Q_{k+1}^*\big(f_k(x_k, u_k, w_k), u\big)\Big\}. \end{aligned} \tag{3.29}$$

This suggests that in place of the Q-factors $Q_k^*(x_k, u_k)$, we may use Q-factor approximations and Eq. (3.29) 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 approximation, enforced decomposition, certainty equivalent control, etc). 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

$$\tilde{Q}_k(x_k, u_k, r_k) = r_k(u_k)'\phi_k(x_k), \tag{3.30}$$

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

$$\tilde{Q}_k(x_k, u_k, r_k) = r_k'\phi_k(x_k, u_k), \tag{3.31}$$

where r_k is a weight vector that is independent of u_k. The architecture (3.30) is suitable for problems with a relatively small number of control options at each stage. In what follows, we will focus on the architecture (3.31), but the discussion with few modifications, also applies to the architecture (3.30) 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. (3.29), with r_k obtained by solving least squares problems similar to the ones of the cost function approximation case [cf. Eq. (3.26)]. As an example, the parameters r_k of the architecture (3.31) are computed sequentially by collecting sample state-control pairs (x_k^s, u_k^s), $s = 1, \ldots, q$, and solving the linear least squares problems

$$r_k \in \arg\min_r \sum_{s=1}^{q} \big(r'\phi_k(x_k^s, u_k^s) - \beta_k^s\big)^2, \tag{3.32}$$

where

$$\beta_k^s = E\Big\{ g_k(x_k^s, u_k^s, w_k) + \min_{u \in U_{k+1}(f_k(x_k^s, u_k^s, w_k))} r_{k+1}' \phi_{k+1}\big(f_k(x_k^s, u_k^s, w_k), u\big) \Big\}. \tag{3.33}$$

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. (3.29). Note that the solution of the least squares problem (3.32) can be obtained in closed form as

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

[cf. Eq. (3.4)]. Once r_k has been computed, the one-step lookahead control $\tilde{\mu}_k(x_k)$ is obtained on-line as

$$\tilde{\mu}_k(x_k) \in \arg \min_{u \in U_k(x_k)} \tilde{Q}_k(x_k, u, r_k), \tag{3.34}$$

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 β_k^s of Eq. (3.33) involve the computation of an expected value. In an alternative implementation, we may replace β_k^s with an average of just a few samples (even a single sample) of the random variable

$$g_k(x_k^s, u_k^s, w_k) + \min_{u \in U_{k+1}(f_k(x_k^s, u_k^s, w_k))} r_{k+1}' \phi_{k+1}\big(f_k(x_k^s, u_k^s, w_k), u\big), \tag{3.35}$$

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; 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 (3.35) do not require the computation of an expected value like the samples (3.25) in the cost approximation method of the preceding chapter. Moreover the samples of (3.35) involve a simpler minimization than the samples (3.25). This is an important advantage of working with Q-factors rather than state costs.

Having obtained the one-step lookahead policy $\tilde{\pi} = \{\tilde{\mu}_0, \ldots, \tilde{\mu}_{N-1}\}$, a further possibility is to approximate it with a parametric architecture. This is *approximation in policy space built on top of approximation in value space*; see the discussion of Section 2.1.3. The idea here is to simplify even further the on-line computation of the suboptimal controls by avoiding the minimization of Eq. (3.34).

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 (3.32)-(3.33). 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. 3.4.1 (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.

3.5 PARAMETRIC APPROXIMATION IN POLICY SPACE BY CLASSIFICATION

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 with some unspecified process, such as rollout or problem approximation. We may then wish to “learn” this process by training the parameter vector

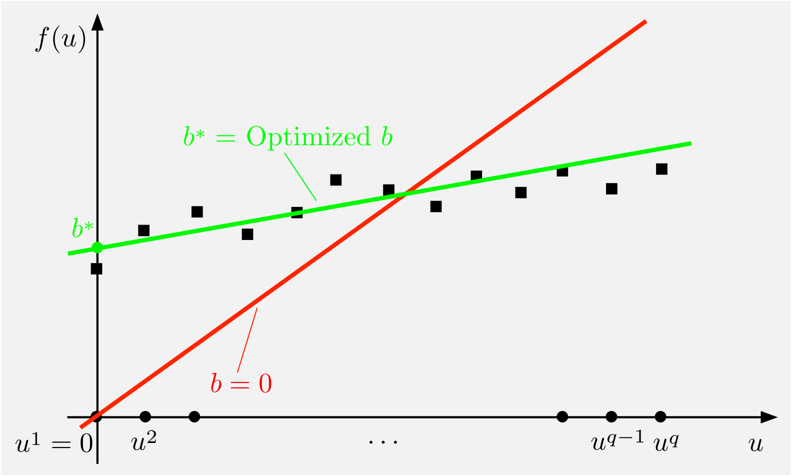

Figure 3.4.1. Illustration of the idea of subtracting a baseline constant from a cost or Q-factor approximation. Here we have samples $h(u^1), \ldots, h(u^q)$ of a scalar function $h(u)$ at sample points $u^1, \ldots, u^q$, and we want to approximate $h(u)$ with a linear function $\tilde{h}(u, r) = ru$, where r is a scalar tunable weight. We subtract a baseline constant b from the samples, and we solve the problem

$$\bar{r} \in \arg\min_r \sum_{s=1}^{q} \Big(\big(h(u^s) - b \big) - r u^s \Big)^2 .$$

By properly adjusting b, we can improve the quality of the approximation, which after subtracting b from all the sample values, takes the form $\tilde{h}(u, b, r) = b + ru$. Conceptually, b serves as an additional weight that enriches the approximation architecture.

r_k of a parametric family of policies $\tilde{\mu}_k(x_k, r_k)$, using least squares minimization/regression:

$$\overline{r}_k \in \arg\min_{r_k} \sum_{s=1}^{q} \big\| u_k^s - \tilde{\mu}_k(x_k^s, r_k) \big\|^2; \tag{3.36}$$

cf. our discussion of approximation in policy space in Section 2.1.5.

It is useful to make the connection of the regression approach of Eq. (3.36) 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 $\big(x_k^s, u_k^s\big)$ as an object-category pair.

Generally, in 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.,

$$c^*(x) \in \arg \max_{c=1,\ldots,m} p(c \,|\, x). \tag{3.37}$$

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 by using the following well-known fact from Bayesian estimation.

Proposition 3.5.1: (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

$$E\Big\{\big(z(c, x) - h(c, x)\big)^2\Big\},$$

the expected value with respect to the distribution $\zeta(c, x)$ of the random variable $\big(z(c, x) - h(c, x)\big)^2$. Then $p(c \,|\, x)$ minimizes this expected value over all functions $h(c, x)$, i.e., for all functions h, we have

$$E\Big\{\big(z(c, x) - p(c \,|\, x)\big)^2\Big\} \leq E\Big\{\big(z(c, x) - h(c, x)\big)^2\Big\}. \tag{3.38}$$

The proof of the proposition may be found in textbooks that deal with Bayesian least squares estimation (see, e.g., [BeT08], Section 8.3).† The proposition states that $p(c\,|\,x)$ is the function of (c,x) that minimizes

$$E\Big\{\big(z(c,x)-h(c,x)\big)^2\Big\} \tag{3.39}$$

over all functions h of (c,x), *independently of the prior distribution of* x. This suggests that we can obtain approximations to the probabilities $p(c\,|\,x)$, $c=1,\ldots,m$, by minimizing an empirical/simulation based approximation of the expected value (3.39).

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

$$z^s(c)=\begin{cases}1 & \text{if } c^s=c,\\ 0 & \text{otherwise,}\end{cases}\qquad c=1,\ldots,m,$$

and adopt a parametric approach. In particular, for each category $c=1,\ldots,m$, we approximate the probability $p(c\,|\,x)$ with a function $\tilde h(c,x,r)$

† For a quick argument, consider for any pair (c,x) the conditional expected value $E\Big\{\big(z(c,x)-y\big)^2\;\big|\;c,x\Big\}$, where y is any scalar. Given (c,x), the random variable $z(c,x)$ takes the value $z(c,x)=1$ with probability $p(c\,|\,x)$ and the value $z(c,x)=0$ with probability $1-p(c\,|\,x)$, so we have

$$E\Big\{\big(z(c,x)-y\big)^2\;\big|\;c,x\Big\}=p(c\,|\,x)(y-1)^2+\big(1-p(c\,|\,x)\big)y^2.$$

We minimize this expression with respect to y, by setting to 0 its derivative, i.e., $0=2p(c\,|\,x)(y-1)+2\big(1-p(c\,|\,x)\big)y=2\big(-p(c\,|\,x)+y\big)$. We thus obtain the minimizing value of y, namely $y^*=p(c\,|\,x)$, so that

$$E\Big\{\big(z(c,x)-p(c\,|\,x)\big)^2\mid c,x\Big\}\le E\Big\{\big(z(c,x)-y\big)^2\mid c,x\Big\},\qquad \text{for all scalars } y.$$

For any function h of (c,x) we set $y=h(c,x)$ in the above expression and obtain

$$E\Big\{\big(z(c,x)-p(c\,|\,x)\big)^2\mid c,x\Big\}\le E\Big\{\big(z(c,x)-h(c,x)\big)^2\mid c,x\Big\}.$$

Since this is true for all (c,x), we also have

$$\sum_{(c,x)}\zeta(c,x)E\Big\{\big(z(c,x)-p(c\,|\,x)\big)^2\;\big|\;c,x\Big\}\le\sum_{(c,x)}\zeta(c,x)E\Big\{\big(z(c,x)-h(c,x)\big)^2\;\big|\;c,x\Big\},$$

for all functions h, i.e., Eq. (3.38) holds.

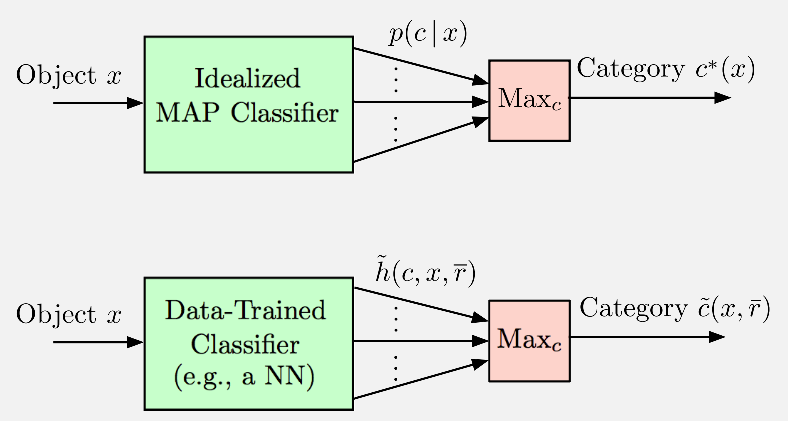

Figure 3.5.1. Illustration of the MAP classifier $c^*(x)$ for the case where the probabilities $p(c\,|\,x)$ are known [cf. Eq. (3.37)], and its data-trained version $\tilde c(x,\bar r)$ [cf. Eq. (3.41)]. The classifier may be obtained by using the data set (x_k^s,u_k^s), $s=1,\dots,q$, and an approximation architecture such as a feature-based architecture or a neural network.

that is parametrized by a vector r, and optimize over r the empirical approximation to the expected squared error of Eq. (3.39). In particular, we obtain r by the least squares regression:

$$\bar r \in \arg\min_r \sum_{s=1}^{q}\sum_{c=1}^{m}\big(z^s(c)-\tilde h(c,x^s,r)\big)^2. \tag{3.40}$$

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 (3.40): $m-1$ "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 training methods described in Sections 3.1 and 3.2 can be applied to the solution of this problem.

The regression problem (3.40) approximates the minimization of the expected value (3.39), so we conclude that its solution $\tilde h(c,x,\bar r)$, $c=1,\dots,m$, approximates the probabilities $p(c\,|\,x)$. Once this solution is obtained, we may use it to classify a new object x according to the rule

$$\text{Estimated Object Category} = \tilde c(x,\bar r) \in \arg\max_{c=1,\dots,m} \tilde h(c,x,\bar r), \tag{3.41}$$

which approximates the MAP rule (3.37); cf. Fig. 3.5.1.

Returning to approximation in policy space, for a given training set (x_k^s,u_k^s), $s=1,\dots,q$, the classifier just described provides (approximations

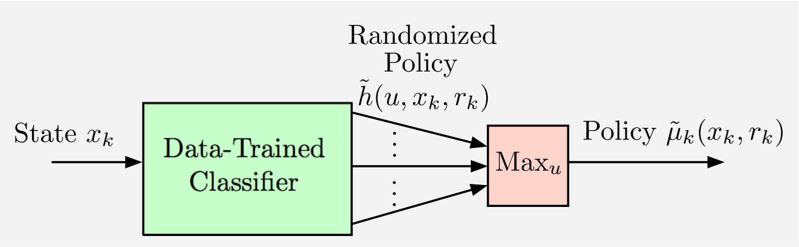

Figure 3.5.2 Illustration of classification-based approximation in policy space. The classifier, defined by the parameter r_k, is constructed by using the training set (x_k^s, u_k^s), $s = 1, \ldots, q$. It yields a randomized policy that consists of the probability $\tilde h(u, x_k, r_k)$ of using control $u \in U_k(x_k)$ at state x_k. This policy is approximated by the deterministic policy $\tilde\mu_k(x_k, r_k)$ that uses at state x_k the control that maximizes over $u \in U_k(x_k)$ the probability $\tilde h(u, x_k, r_k)$ [cf. Eq. (3.41)].

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. 3.5.2. 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. (3.41).

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:

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

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.

3.6 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 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 references in addition to the ones given in Sections 3.1.3 and 3.2.1.

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 algorithms for training of neural networks as well as linear architectures.

Both of these issues have been extensively investigated and research relating to these and other related issues 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 [GBC16]). For discussions that are focused outside the neural network area, see Bertsekas and Tsitsiklis [BeT96], Keller, Mannor, and Precup [KMP06], Jung and Polani [JuP07], Bertsekas and Yu [BeY09], and Bhatnagar, Borkar, and Prashanth [BBP13]. Moreover, there has been considerable research on the optimization of feature selection within given parametric classes (see Menache, Mannor, and Shimkin [MMS05], Yu and Bertsekas [YuB09b], 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], [Wab16], Iusem, Jofre, and Thompson [IJT18]. It is beyond our scope to cover this analysis, and we refer to the book [BeT96] and paper [BeT00] by Bertsekas and Tsitsiklis, and the recent survey by Bottou, Curtis, and Nocedal [BCN18] for theoretically oriented accounts. 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, incremental gradient, incremental subgradient, incremental aggregated gradient, incremental Newton, and incremental Gauss-Newton methods.

Section 3.2: The publicly and commercially available neural network training programs incorporate a number of heuristics for scaling and preprocessing data, stepsize selection, initialization, etc, which can be very effective in specialized problem domains. We also 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. A key theoretical question regarding deep neural networks is whether and for what classes of functions approximation power can be enhanced 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.

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 behavior, such as instability over a long horizon. We will discuss this behavior in the context of infinite horizon problems.

Section 3.4: Advantage updating was proposed by Baird [Bai93], [Bai94], and is discussed in Section 6.6 of the book [BeT96].

Section 3.5: 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 (cf. Section 5.7.3). While we have focused in this section 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 classification based on the use of support vector machines, as well as neural networks.