April 2018 (revised August 2018) **MIT/LIDS Report**

Feature-Based Aggregation and Deep Reinforcement Learning: A Survey and Some New Implementations

Dimitri P. Bertsekas†

Abstract

In this paper we discuss policy iteration methods for approximate solution of a finite-state discounted Markov decision problem, with a focus on feature-based aggregation methods and their connection with deep reinforcement learning schemes. We introduce features of the states of the original problem, and we formulate a smaller "aggregate" Markov decision problem, whose states relate to the features. We discuss properties and possible implementations of this type of aggregation, including a new approach to approximate policy iteration. In this approach the policy improvement operation combines feature-based aggregation with feature construction using deep neural networks or other calculations. We argue that the cost function of a policy may be approximated much more accurately by the nonlinear function of the features provided by aggregation, than by the linear function of the features provided by neural network-based reinforcement learning, thereby potentially leading to more effective policy improvement.

† Dimitri Bertsekas is with the Dept. of Electr. Engineering and Comp. Science, and the Laboratory for Information and Decision Systems, M.I.T., Cambridge, Mass., 02139. A version of this paper will appear in IEEE/CAA Journal of Automatica Sinica.

Contents

1. INTRODUCTION

We consider a discounted infinite horizon dynamic programming (DP) problem with n states, which we denote by $i = 1, \ldots, n$. State transitions (i, j) under control u occur at discrete times according to transition probabilities $p_{ij}(u)$, and generate a cost $\alpha^k g(i, u, j)$ at time k, where $\alpha \in (0, 1)$ is the discount factor. We consider deterministic stationary policies μ such that for each i, $\mu(i)$ is a control that belongs to a constraint set $U(i)$. We denote by $J_\mu(i)$ the total discounted expected cost of μ over an infinite number of stages starting from state i, and by $J^*(i)$ the minimal value of $J_\mu(i)$ over all μ. We denote by J_μ and J^* the n-dimensional vectors that have components $J_\mu(i)$ and $J^*(i)$, $i = 1, \ldots, n$, respectively. As is well known, J_μ is the unique solution of the Bellman equation for policy μ:

$$J_\mu(i) = \sum_{j=1}^{n} p_{ij}\big(\mu(i)\big)\Big(g\big(i, \mu(i), j\big) + \alpha J_\mu(j)\Big), \qquad i = 1, \ldots, n, \tag{1.1}$$

while J^* is the unique solution of the Bellman equation

$$J^*(i) = \min_{u \in U(i)} \sum_{j=1}^{n} p_{ij}(u)\big(g(i, u, j) + \alpha J^*(j)\big), \qquad i = 1, \ldots, n. \tag{1.2}$$

In this paper, we survey several ideas from aggregation-based approximate DP and deep reinforcement learning, all of which have been essentially known for some time, but are combined here in a new way. We will focus on methods of approximate policy iteration (PI for short), whereby we evaluate approximately the cost vector J_μ of each generated policy μ. Our cost approximations use a feature vector $F(i)$ of each state i, and replace $J_\mu(i)$ with a function that depends on i through $F(i)$, i.e., a function of the form

$$\hat{J}_\mu\big(F(i)\big) \approx J_\mu(i), \qquad i = 1, \ldots, n.$$

We refer to such $\hat{J}_\mu$ as a *feature-based approximation architecture*.

At the typical iteration of our approximate PI methodology, the cost vector J_μ of the current policy μ is approximated using a feature-based architecture $\hat{J}_\mu$, and a new policy $\hat{\mu}$ is then generated using a policy "improvement" procedure; see Fig. 1.1. The salient characteristics of our approach are two:

(a) The feature vector $F(i)$ may be obtained using a neural network or other calculation that automatically constructs features.

(b) The policy improvement, which generates $\hat{\mu}$ is based on a DP problem that involves feature-based aggregation.

By contrast, the standard policy improvement method is based on the one-step lookahead minimization

$$\hat{\mu}(i) \in \arg\min_{u \in U(i)} \sum_{j=1}^{n} p_{ij}(u)\Big(g(i, u, j) + \alpha \hat{J}_\mu\big(F(j)\big)\Big), \qquad i = 1, \ldots, n, \tag{1.3}$$

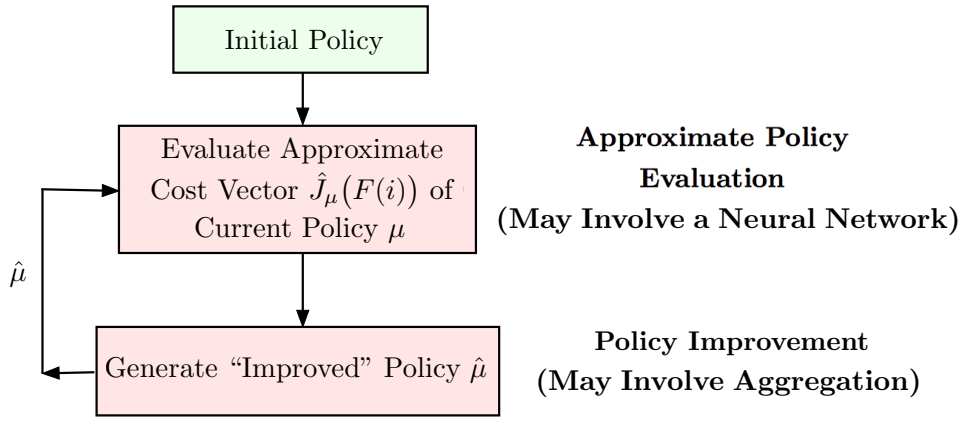


Figure 1.1 Schematic view of feature-based approximate PI. The cost $J_\mu(i)$ of the current policy μ starting from state i is replaced by an approximation $\hat{J}_\mu\big(F(i)\big)$ that depends on i through its feature vector $F(i)$. The feature vector is assumed independent of the current policy μ in this figure, but in general could depend on μ.

or alternatively, on multistep lookahead, possibly combined with Monte-Carlo tree search. We will argue that our feature-based aggregation approach has the potential to generate far better policies at the expense of a more computation-intensive policy improvement phase.

1.1. Alternative Approximate Policy Iteration Methods

A survey of approximate PI methods was given in 2011 by the author [Ber11a], and focused on *linear feature-based architectures*. These are architectures where $F(i)$ is an s-dimensional vector

$$F(i) = \big(F_1(i), \ldots, F_s(i)\big),$$

and $\hat{J}_\mu$ depends linearly on F, i.e.,

$$\hat{J}_\mu\big(F(i)\big) = \sum_{\ell=1}^{s} F_\ell(i) r_\ell, \qquad i = 1, \ldots, n,$$

for some scalar weights $r_1, \ldots, r_s$. We considered in [Ber11a] two types of methods:

(a) Projected equation methods, including temporal difference methods, where policy evaluation is based on simulation-based matrix inversion methods such as LSTD(λ), or stochastic iterative methods such as TD(λ), or variants of λ-policy iteration such as LSPE(λ)].

(b) General aggregation methods (not just the feature-based type considered here).

These methods will be briefly discussed in Section 2. The present paper is complementary to the survey [Ber11a], and deals with approximate PI with nonlinear feature-based architectures, including some where features are generated with the aid of neural networks or some other heuristic calculations.

An important advantage of linear feature-based architectures is that given the form of the feature vector $F(\cdot)$, they can be trained with linear least squares-type methods. However, determining good features may be a challenge in general. Neural networks resolve this challenge through training that constructs automatically features and simultaneously combines the components of the features linearly with weights. This is commonly done by cost fitting/nonlinear regression, using a large number of state-cost sample pairs, which are processed through a sequence of alternately linear and nonlinear layers (see Section 3). The outputs of the final nonlinear layer are the features, which are then processed by a final linear layer that provides a linear combination of the features as a cost function approximation.

The idea of representing cost functions in terms of features of the state in a context that we may now call "approximation in value space" or "approximate DP" goes back to the work of Shannon on chess [Sha50]. The work of Samuel [Sam59], [Sam67] on checkers extended some of Shannon's algorithmic schemes and introduced *temporal difference* ideas that motivated much subsequent research. The use of neural networks to simultaneously extract features of the optimal or the policy cost functions, and construct an approximation to these cost functions was also investigated in the early days of reinforcement learning; some of the original contributions that served as motivation for much subsequent work are Werbos [Wer77], Barto, Sutton, and Anderson [BSA83], Christensen and Korf [ChK86], Holland [Hol86], and Sutton [Sut88]. The use of a neural network as a cost function approximator for a challenging DP problem was first demonstrated impressively in the context of the game of backgammon by Tesauro [Tes92], [Tes94], [Tes95], [Tes02]. In Tesauro's work the parameters of the network were trained by using a form of temporal differences (TD) learning, and the features constructed by the neural network were supplemented by some handcrafted features.†

Following Tesauro's work, the synergistic potential of approximations using neural network or other architectures, and DP techniques had become apparent, and it was laid out in an influential survey paper by Barto, Bradtke, and Singh [BBS95]. It was then systematically developed in the neuro-dynamic programming book by Bertsekas and Tsitsiklis [BeT96], and the reinforcement learning book by Sutton and Barto [SuB98]. Subsequent books on approximate DP and reinforcement learning, which discuss approximate PI, among

† Tesauro also constructed a different backgammon player, trained by a neural network, but with a supervised learning approach, which used examples from human expert play [Tes89a], [Tes89b] (he called this approach "comparison learning"). However, his TD-based algorithm performed substantially better, and its success has been replicated by others, in both research and commercial programs. Tesauro and Galperin [TeG96] proposed still another approach to backgammon, based on a rollout strategy, which resulted in an even better playing program (see [Ber17] for an extensive discussion of rollout as a general approximate DP approach). At present, rollout-based backgammon programs are viewed as the most powerful in terms of performance, but are too time-consuming for real-time play. They have been used in a limited diagnostic way to assess the quality of neural network-based programs. A list of articles on computer backgammon may be found at http://www.bkgm.com/articles/page07.html.

other techniques, include Cao [Cao07], Busoniu et. al. [BBD10], Szepesvari [Sze10], Powell [Pow11], Chang, Fu, Hu, and Marcus [CFH13], Vrabie, Vamvoudakis, and Lewis [VVL13], and Gosavi [Gos15]. To these, we may add the edited collections by Si, Barto, Powell, and Wunsch [SBP04], Lewis, Liu, and Lendaris [LLL08], and Lewis and Liu [LeL12], which contain several survey papers.

The original ideas on approximate PI were enriched by further research ideas such as *rollout* (Abramson [Abr90], Tesauro and Galperin [TeG96], Bertsekas, Tsitsiklis, and Wu [BTW97], Bertsekas and Castanon [BeC99]; see the surveys in [Ber13], [Ber17]), *adaptive simulation and Monte Carlo tree search* (Chang, Hu, Fu, and Marcus [CFH05], [CFH13], Coulom [Cou06]; see the survey by Browne et al. [BPW12]), and *deep neural networks* (which are neural networks with many and suitably specialized layers; see for the example the book by Goodfellow, Bengio, and Courville [GBC16], the textbook discussion in [Ber17], Ch. 6, and the recent surveys by Schmidhuber [Sch15], Arulkumaran et al. [ADB17], Liu et al. [LWL17], and Li [Li17]).

A recent impressive success of the deep neural network-based approximate PI methodology is the AlphaZero program, which attained a superhuman level of play for the games of chess, Go, and others (see Silver et al. [SHS17]). A noteworthy characteristic of this program is that it does not use domain-specific knowledge (i.e., handcrafted features), but rather relies entirely on the deep neural network to construct features for cost function approximation (at least as reported in [SHS17]). Whether it is advisable to rely exclusively on the neural network to provide features is an open question, as other investigations, including the ones by Tesauro noted earlier, suggest that using additional problem-specific hand-crafted features can be very helpful in the context of approximate DP. Except for the use of deep rather than shallow neural networks (which are used in backgammon), the AlphaZero algorithm is similar to several other algorithms that have been proposed in the literature and/or have been developed in the past. It can be viewed as a conceptually straightforward implementation of approximate PI, using Monte Carlo tree search and a single neural network to construct a cost and policy approximation, and does not rely on any fundamentally new ideas or insightful theoretical analysis. Conceptually, it bears considerable similarity to Tesauro's TD-Gammon program. Its spectacular success may be attributed to the skillful implementation of an effective mix of known ideas, coupled with great computational power.

We note 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]), including the Group Method for Data Handling (GMDH), which is principally based on the use of polynomial (rather than sigmoidal) nonlinearities. The GMDH method 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. We

are unaware of any application of GMDH in the context of approximate DP, but we believe this to be a fruitful area of investigation. In any case, the feature-based PI ideas of the present paper apply equally well in conjunction with GMDH networks as with the neural networks described in Section 3.

While automatic feature extraction is a critically important aspect of neural network architectures, the linearity of the combination of the feature components at the final layer may be a limitation. A nonlinear alternative is based on aggregation, a dimensionality reduction approach to address large-scale problems. This approach has a long history in scientific computation and operations research (see for example Bean, Birge, and Smith [BBS87], Chatelin and Miranker [ChM82], Douglas and Douglas [DoD93], Mendelssohn [Men82], and Rogers et. al. [RPW91]). It was introduced in the simulation-based approximate DP context, mostly in the form of value iteration; see Singh, Jaakkola, and Jordan [SJJ95], Gordon [Gor95], Tsitsiklis and Van Roy [TsV96] (see also the book [BeT96], Sections 3.1.2 and 6.7). More recently, aggregation was discussed in a reinforcement learning context involving the notion of "options" by Ciosek and Silver [CiS15], and the notion of "bottleneck simulator" by Serban et. al. [SSP18]; in both cases encouraging computational results were presented. Aggregation architectures based on features were discussed in Section 3.1.2 of the neuro-dynamic programming book [BeT96], and in Section 6.5 of the author's DP book [Ber12] (and earlier editions), including the feature-based architecture that is the focus of the present paper. They have the capability to produce policy cost function approximations that are nonlinear functions of the feature components, thus yielding potentially more accurate approximations. Basically, in feature-based aggregation the original problem is approximated by a problem that involves a relatively small number of "feature states."

Feature-based aggregation assumes a given form of feature vector, so for problems where good features are not apparent, it needs to be modified or to be supplemented by a method that can construct features from training data. Motivated by the reported successes of deep reinforcement learning with neural networks, we propose a two-stage process: first use a neural network or other scheme to construct good features for cost approximation, and then use there features to construct a nonlinear feature-based aggregation architecture. In effect we are proposing a new way to implement approximate PI: *retain the policy evaluation phase which uses a neural network or alternative scheme, but replace the policy improvement phase with the solution of an aggregate DP problem*. This DP problem involves the features that are generated by a neural network or other scheme (possibly together with other handcrafted features). Its dimension may be reduced to a manageable level by sampling, while its cost function values are generalized to the entire feature space by linear interpolation. In summary, our suggested policy improvement phase may be more complicated, but may be far more powerful as it relies on the potentially more accurate function approximation provided by a nonlinear combination of features.

Aside from the power brought to bear by nonlinearly combining features, let us also note some other advantages that are generic to aggregation. In particular:

(a) Aggregation aims to solve an "aggregate" DP problem, itself an approximation of the original DP problem, in the spirit of coarse-grid discretization of large state space problems. As a result, aggregation methods enjoy the stability and policy convergence guarantee of exact PI. By contrast, temporal difference-based and other PI methods can suffer from convergence difficulties such as policy oscillations and chattering (see e.g., [BeT96], [Ber11a], [Ber12]). A corollary to this is that when an aggregation scheme performs poorly, it is easy to identify the cause: it is the quantization error due to approximating a large state space with a smaller "aggregate" space. The possible directions for improvement (at a computational cost of course) are then clear: introduce additional aggregate states, and increase/improve these features.

(b) Aggregation methods are characterized by error bounds, which are generic to PI methods that guarantee the convergence of the generated policies. These error bounds are better by a factor $(1-\alpha)$ compared to the corresponding error bounds for methods where policies need not converge, such as generic temporal difference methods with linear cost function approximation [see Eqs. (2.2) and (2.3) in the next section].

Let us finally note that the idea of using a deep neural network to extract features for use in another approximation architecture has been used earlier. In particular, it is central in the Deepchess program by David, Netanyahu, and Wolf [DNW16], which was estimated to perform at the level of a strong grandmaster, and at the level of some of the strongest computer chess programs. In this work the features were used, in conjunction with supervised learning and human grandmaster play selections, to train a deep neural network to compare any pair of legal moves in a given chess position, in the spirit of Tesauro's comparison training approach [Tes89b]. By contrast in our proposal the features are used to formulate an aggregate DP problem, which can be solved by exact methods, including some that are based on simulation.

The paper is organized as follows. In Section 2, we provide context for the subsequent developments, and summarize some of the implementation issues in approximate PI methods. In Section 3, we review some of the central ideas of approximate PI based on neural networks. In Section 4, we discuss PI ideas based on feature-based aggregation, assuming good features are known. In this section, we also discuss how features may be constructed based on one or more "scoring functions," which are estimates of the cost function of a policy, provided by a neural network or a heuristic. We also pay special attention to deterministic discrete optimization problems. Finally, in Section 5, we describe some of the ways to combine the feature extraction capability of deep neural networks with the nonlinear approximation possibilities offered by aggregation.

1.2. Reinforcement Learning and Optimal Control - Some Terminology

The success of approximate DP in addressing challenging large-scale applications owes much to an enormously beneficial cross-fertilization of ideas from decision and control, and from artificial intelligence. The boundaries between these fields are now diminished thanks to a deeper understanding of the foundational issues, and the

associated methods and core applications. Unfortunately, however, there have been substantial discrepancies of notation and terminology between the artificial intelligence and the optimization/decision/control fields, including the typical use of maximization/value function/reward in the former field and the use of minimization/cost function/cost per stage in the latter field. The notation and terminology used in this paper is standard in DP and optimal control, and in an effort to forestall confusion of readers that are accustomed to either the reinforcement learning or the optimal control terminology, we provide a list of selected terms commonly used in reinforcement learning (for example in the popular book by Sutton and Barto [SuB98], and its 2018 on-line 2nd edition), and their optimal control counterparts.

(a) Agent = Controller or decision maker.

(b) Action = Control.

(c) Environment = System.

(d) Reward of a stage = (Opposite of) Cost of a stage.

(e) State value = (Opposite of) Cost of a state.

(f) Value (or state-value) function = (Opposite of) Cost function.

(g) Maximizing the value function = Minimizing the cost function.

(h) Action (or state-action) value = Q-factor of a state-control pair.

(i) Planning = Solving a DP problem with a known mathematical model.

(j) Learning = Solving a DP problem in model-free fashion.

(k) Self-learning (or self-play in the context of games) = Solving a DP problem using policy iteration.

(l) Deep reinforcement learning = Approximate DP using value and/or policy approximation with deep neural networks.

(m) Prediction = Policy evaluation.

(n) Generalized policy iteration = Optimistic policy iteration.

(o) State abstraction = Aggregation.

(p) Episodic task or episode = Finite-step system trajectory.

(q) Continuing task = Infinite-step system trajectory.

(r) Afterstate = Post-decision state.

2. APPROXIMATE POLICY ITERATION: AN OVERVIEW

Many approximate DP algorithms are based on the principles of PI: the policy evaluation/policy improvement structure of PI is maintained, but the policy evaluation is done approximately, using simulation and some approximation architecture. In the standard form of the method, at each iteration, we compute an approximation $\tilde{J}_\mu(\cdot, r)$ to the cost function J_μ of the current policy μ, and we generate an "improved" policy $\hat{\mu}$ using†

$$\hat{\mu}(i) \in \arg\min_{u \in U(i)} \sum_{j=1}^{n} p_{ij}(u)\big(g(i,u,j) + \alpha \tilde{J}_\mu(j,r)\big), \qquad i = 1, \ldots, n. \tag{2.1}$$

Here $\tilde{J}_\mu$ is a function of some chosen form (the *approximation architecture*), which depends on the state and on a parameter vector $r = (r_1, \ldots, r_s)$ of relatively small dimension s.

The theoretical basis for the method was discussed in the neuro-dynamic programming book [BeT96], Prop. 6.2 (see also [Ber12], Section 2.5.6, or [Ber18a], Sections 2.4.1 and 2.4.2). It was shown there that if the policy evaluation is accurate to within δ (in the sup-norm sense), then for an α-discounted problem, the method, while not convergent, is stable in the sense that it will yield in the limit (after infinitely many policy evaluations) stationary policies that are optimal to within

$$\frac{2\alpha\delta}{(1-\alpha)^2}, \tag{2.2}$$

where α is the discount factor. Moreover, if the generated sequence of policies actually converges to some $\bar{\mu}$, then $\bar{\mu}$ is optimal to within

$$\frac{2\alpha\delta}{1-\alpha} \tag{2.3}$$

(see [BeT96], Section 6.4.1); this is a significantly improved error bound. In general, policy convergence may not be guaranteed, although it is guaranteed for the aggregation methods of this paper. Experimental evidence indicates that the bounds (2.2) and (2.3) are often conservative, with just a few policy iterations needed before most of the eventual cost improvement is achieved.

2.1. Direct and Indirect Approximation Approaches for Policy Evaluation

Given a class of functions $\mathcal{J}$ that defines an approximation architecture, there are two general approaches for approximating the cost function J_μ of a fixed policy μ within $\mathcal{J}$. The most straightforward approach,

† The minimization in the policy improvement phase may alternatively involve multistep lookahead, possibly combined with Monte-Carlo tree search. It may also be done approximately through Q-factor approximations. Our discussion extends straightfowardly to schemes that include multistep lookahead or approximate policy improvement.

referred to as *direct* (or cost fitting), is to find a $\tilde{J}_\mu \in \mathcal{J}$ that matches J_μ in some least squares error sense, i.e.,†

$$\tilde{J}_\mu \in \arg\min_{\tilde{J}\in\mathcal{J}} \|\tilde{J} - J_\mu\|^2. \tag{2.4}$$

Typically $\|\cdot\|$ is some weighted Euclidean norm with positive weights ξ_i, $i = 1, \ldots, n$, while $\mathcal{J}$ consists of a parametrized class of functions $\tilde{J}(i, r)$ where $r = (r_1, \ldots, r_s) \in \Re^s$ is the parameter vector, i.e.,‡

$$\mathcal{J} = \big\{\tilde{J}(\cdot, r) \mid r \in \Re^s\big\}.$$

Then the minimization problem in Eq. (2.4) is written as

$$\min_{r\in\Re^s} \sum_{i=1}^{n} \xi_i\big(\tilde{J}(i, r) - J_\mu(i)\big)^2, \tag{2.5}$$

and can be viewed as an instance of nonlinear regression.

In simulation-based methods, the preceding minimization is usually approximated by a least squares minimization of the form

$$\min_{r\in\Re^s} \sum_{m=1}^{M} \big(\tilde{J}(i_m, r) - \beta_m\big)^2, \tag{2.6}$$

where (i_m, β_m), $m = 1, \ldots, M$, are a large number of state-cost sample pairs, i.e., for each m, i_m is a sample state and β_m is equal to $J_\mu(i_m)$ plus some simulation noise. Under mild statistical assumptions on the sample collection process, the sample-based minimization (2.6) is equivalent in the limit to the exact minimization (2.5). Neural network-based approximation, as described in Section 3, is an important example of direct approximation that uses state-cost training pairs.

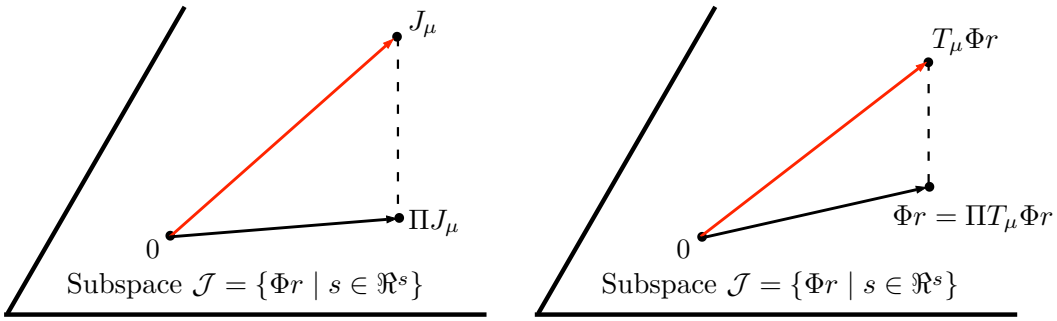
A common choice is to take $\mathcal{J}$ to be the subspace $\{\Phi r \mid r \in \Re^s\}$ that is spanned by the columns of an $n \times s$ matrix Φ, which can be viewed as basis functions (see the left side of Fig. 2.1). Then the approximation problem (2.6) becomes the linear least squares problem

$$\min_{(r_1,\ldots,r_s)\in\Re^s} \sum_{m=1}^{M} \left(\sum_{\ell=1}^{s} \phi_{i_m\ell} r_\ell - \beta_m\right)^2, \tag{2.7}$$

where $\phi_{i\ell}$ is the $i\ell$th entry of the matrix Φ and r_ℓ is the ℓth component of r. The solution of this problem can be obtained analytically and can be written in closed form (see e.g., [BeT96], Section 3.2.2). Note that the ith row of Φ may be viewed as a feature vector of state i, and Φr may be viewed as a linear feature-based architecture.

† Nonquadratic optimization criteria may also be used, although in practice the simple quadratic cost function has been adopted most frequently.

‡ We use standard vector notation. In particular, $\Re^s$ denotes the Euclidean space of s-dimensional real vectors, and $\Re$ denotes the real line.



Direct Method: Projecting the cost vector J_μ

Indirect Method: Solving a projected form of Bellman's equation

Figure 2.1 Two methods for approximating the cost function J_μ as a linear combination of basis functions. The approximation architecture is the subspace $\mathcal{J} = \{\Phi r \mid r \in \Re^s\}$, where Φ is matrix whose columns are the basis functions. In the direct method (see the figure on the left), J_μ is projected on $\mathcal{J}$. In an example of the indirect method, the approximation is obtained by solving the projected form of Bellman's equation $\Phi r = \Pi T_\mu \Phi r$, where $T_\mu \Phi r$ is the vector with components

$$(T_\mu \Phi r)(i) = \sum_{j=1}^{n} p_{ij}\big(\mu(i)\big)\Big(g\big(i, \mu(i), j\big) + \alpha(\Phi r)(j)\Big), \qquad i = 1, \ldots, n,$$

and $(\Phi r)(j)$ is the jth component of the vector Φr (see the figure on the right).

In Section 3, we will see that neural network-based policy evaluation combines elements of both a linear and a nonlinear architecture. The nonlinearity is embodied in the features that the neural network constructs through training, but once the features are given, the neural network can be viewed as a linear feature-based architecture.

An often cited weakness of simulation-based direct approximation is excessive simulation noise in the cost samples β_m that are used in the least squares minimization (2.7). This has motivated alternative approaches for policy evaluation that inherently involve less noise. A major approach of this type, referred to as *indirect* (or equation fitting), is to approximate Bellman's equation for the policy μ,

$$J(i) = \sum_{j=1}^{n} p_{ij}\big(\mu(i)\big)\Big(g\big(i, \mu(i), j\big) + \alpha J(j)\Big), \qquad i = 1, \ldots, n, \tag{2.8}$$

with another equation that is defined on the set $\mathcal{J}$. The solution of the approximate equation is then used as an approximation of the solution of the original. The most common indirect methods assume a linear approximation architecture, i.e., $\mathcal{J}$ is the subspace $\mathcal{J} = \{\Phi r \mid r \in \Re^s\}$, and approximate Bellman's equation with another equation with fewer variables, the s parameters $r_1, \ldots, r_s$. Two major examples of this approach are *projected equation* methods and *aggregation* methods, which we proceed to discuss.

2.2. Indirect Methods Based on Projected Equations

Approximation using projected equations has a long history in numerical computation (e.g., partial differential equations) where it is known as *Galerkin approximation* [see e.g., [KVZ72], [Fle84], [Saa03], [Kir11]]. The projected equation approach is a special case of the so called Bubnov-Galerkin method, as noted in the papers [Ber11a], [Ber11b], and [YuB10]. In the context of approximate DP it is connected with *temporal difference methods*, and it is discussed in detail in many sources (see e.g., [BeT96], [BBD10], [Ber12], [Gos15]).

To state the projected equation, let us introduce the transformation T_μ, which is defined by the right-hand side of the Bellman equation (2.8); i.e., for any $J \in \Re^n$, $T_\mu J$ is the vector of $\Re^n$ with components

$$(T_\mu J)(i) = \sum_{j=1}^{n} p_{ij}\big(\mu(i)\big)\Big(g\big(i,\mu(i),j\big) + \alpha J(j)\Big), \qquad i = 1, \ldots, n. \tag{2.9}$$

Note that T_μ is a linear transformation from $\Re^n$ to $\Re^n$, and in fact in compact vector-matrix notation, it is written as

$$T_\mu J = g_\mu + \alpha P_\mu J, \qquad J \in \Re^n, \tag{2.10}$$

where P_μ is the transition probability matrix of μ, and g_μ is the expected cost vector of μ, i.e., the vector with components

$$\sum_{j=1}^{n} p_{ij}\big(\mu(i)\big)g\big(i,\mu(i),j\big), \qquad i = 1, \ldots, n.$$

Moreover the Bellman equation (2.8) is written as the fixed point equation

$$J = T_\mu J.$$

Let us denote by ΠJ the projection of a vector $J \in \Re^n$ onto $\mathcal{J}$ with respect to some weighted Euclidean norm, and consider $\Pi T_\mu \Phi r$, the projection of $T_\mu \Phi r$ (here $T_\mu \Phi r$ is viewed as a vector in $\Re^n$, and Π is viewed as an $n \times n$ matrix multiplying this vector). The projected equation takes the form

$$\Phi r = \Pi T_\mu \Phi r; \tag{2.11}$$

see the right-hand side of Fig. 2.1. With this equation we want to find a vector Φr of $\mathcal{J}$, which when transformed by T_μ and then projected back onto $\mathcal{J}$, yields itself. This is an overdetermined system of linear equations (n equations in the s unknowns $r_1, \ldots, r_s$), which is equivalently written as

$$\sum_{\ell=1}^{s} \phi_{i\ell} r_\ell = \sum_{m=1}^{n} \pi_{im} \sum_{j=1}^{n} p_{mj}\big(\mu(m)\big)\left(g\big(m,\mu(m),j\big) + \alpha \sum_{\ell=1}^{s} \phi_{j\ell} r_\ell\right), \qquad i = 1, \ldots, n; \tag{2.12}$$

here $\phi_{i\ell}$ is the $i\ell$th component of the matrix Φ and π_{im} is the imth component of the projection matrix Π. The system can be shown to have a unique solution under conditions that can be somewhat restrictive, e.g.,

assuming that the Markov chain corresponding to the policy μ has a unique steady-state distribution with positive components, that the projection norm involves this distribution, and that Φ has linearly independent columns (see e.g., [Ber12], Section 6.3).

An important extension is to replace the projected equation (2.11) with the equation

$$\Phi r = \Pi T_\mu^{(\lambda)} \Phi r, \tag{2.13}$$

where λ is a scalar with $0 \leq \lambda < 1$, and the transformation $T_\mu^{(\lambda)}$ is defined by

$$\big(T_\mu^{(\lambda)} J\big)(i) = (1-\lambda) \sum_{\ell=0}^{\infty} \lambda^\ell (T_\mu^{\ell+1} J)(i), \qquad i = 1, \ldots, n, \ J \in \Re^n, \tag{2.14}$$

and $T_\mu^\ell J$ is the ℓ-fold composition of T_μ applied to the vector J. This approach to the approximate solution of Bellman's equation is supported by extensive theory and practical experience (see the textbooks noted earlier). In particular, the TD(λ) algorithm, and other related temporal difference methods, such as LSTD(λ) and LSPE(λ), aim to solve by simulation the projected equation (2.13). The choice of λ embodies the important bias-variance tradeoff: larger values of λ lead to better approximation of J_μ, but require a larger number of simulation samples because of increased simulation noise (see the discussion in Section 6.3.6 of [Ber12]). An important insight is that the operator $T_\mu^{(\lambda)}$ is closely related to the proximal operator of convex analysis (with λ corresponding to the penalty parameter of the proximal operator), as shown in the author's paper [Ber16a] (see also the monograph [Ber18a], Section 1.2.5, and the paper [Ber18b]). In particular, TD(λ) can be viewed as a stochastic simulation-based version of the proximal algorithm.

A major issue in projected equation methods is whether the linear transformation ΠT_μ [or $\Pi T_\mu^{(\lambda)}$] is a contraction mapping, in which case Eq. (2.11) [or Eq. (2.13), respectively] has a unique solution, which may be obtained by iterative fixed point algorithms. This depends on the projection norm, and it turns out that there are special norms for which $\Pi T_\mu^{(\lambda)}$ is a contraction (these are related to the steady-state distribution of the system's Markov chain under μ; see the discussion of [Ber11a] or Section 6.3 of [Ber12]). An important fact is that for any projection norm, $\Pi T_\mu^{(\lambda)}$ is a contraction provided λ is sufficiently close to 1. Still the contraction issue regarding $\Pi T_\mu^{(\lambda)}$ is significant and affects materially the implementation of the corresponding approximate PI methods.

Another important concern is that the projection matrix Π may have some negative entries [i.e., some of the components π_{im} in Eq. (2.12) may be negative], and as a result the linear transformations ΠT_μ and $\Pi T_\mu^{(\lambda)}$ may lack the monotonicity property that is essential for the convergence of the corresponding approximate PI method. Indeed the lack of monotonicity (the possibility that we may not have $\Pi T_\mu J \geq \Pi T_\mu J'$ for two vectors J, J' with $J \geq J'$) is the fundamental mathematical reason for policy oscillations in PI methods that are based on temporal differences (see [Ber11a], [Ber12]). We refer to the literature for further details and analysis regarding the projected equations (2.11) and (2.13), as our focus will be on aggregation methods, which we discuss next.

2.3. Indirect Methods Based on Aggregation

Aggregation is another major indirect approach, which has originated in numerical linear algebra. Simple examples of aggregation involve finite-dimensional approximations of infinite dimensional equations, coarse grid approximations of linear systems of equations defined over a dense grid, and other related methods for dimensionality reduction of high-dimensional systems. In the context of DP, the aggregation idea is implemented by replacing the Bellman equation $J = T_\mu J$ [cf. Eq. (2.8)] with a lower-dimensional "aggregate" equation, which is defined on an approximation subspace $\mathcal{J} = \{\Phi r \mid r \in \Re^s\}$. The aggregation counterpart of the projected equation $\Phi r = \Pi T_\mu \Phi r$ is

$$\Phi r = \Phi D T_\mu \Phi r, \tag{2.15}$$

where Φ and D are some matrices, and T_μ is the linear transformation given by Eq. (2.9).† This is a vector-matrix notation for the linear system of n equations in the s variables $r_1, \ldots, r_s$

$$\sum_{k=1}^{s} \phi_{ik} r_k = \sum_{k=1}^{s} \phi_{ik} \sum_{m=1}^{n} d_{km} \sum_{j=1}^{n} p_{mj}\big(\mu(m)\big) \left(g\big(m, \mu(m), j\big) + \alpha \sum_{\ell=1}^{s} \phi_{j\ell} r_\ell \right), \qquad i = 1, \ldots, n,$$

where $\phi_{i\ell}$ is the $i\ell$th component of the matrix Φ and d_{km} is the kmth component of the matrix D.

A key restriction for aggregation methods as applied to DP is that *the rows of D and Φ should be probability distributions*. These distributions usually have intuitive interpretations in the context of specific aggregation schemes; see [Ber12], Section 6.5 for a discussion. Assuming that Φ has linearly independent columns, which is true for the most common types of aggregation schemes, Eq. (2.15) can be seen to be equivalent to

$$r = D T_\mu \Phi r, \tag{2.16}$$

or

$$r_k = \sum_{m=1}^{n} d_{km} \sum_{j=1}^{n} p_{mj}\big(\mu(m)\big) \left(g\big(m, \mu(m), j\big) + \alpha \sum_{\ell=1}^{s} \phi_{j\ell} r_\ell \right), \qquad k = 1, \ldots, s. \tag{2.17}$$

In most of the important aggregation methods, including the one of Section 4, D and Φ are chosen so that the product $D\Phi$ is the identity:

$$D\Phi = I.$$

† It turns out that under some widely applicable conditions, including the assumptions of Section 4, the projected and aggregation equations are closely related. In particular, it can be proved under these conditions that the matrix ΦD that appears in the aggregation equation (2.15) is a projection with respect to a suitable weighted Euclidean seminorm (see [YuB12], Section 4, or the book [Ber12]; it is a norm projection in the case of hard aggregation). Aside from establishing the relation between the two major indirect approximation methods, projected equation and aggregation, this result provides the basis for transferring the rich methodology of temporal differences methods such as TD(λ) to the aggregation context.

Assuming that this is true, the operator $I - DT_\mu\Phi$ of the aggregation equation (2.16) is obtained by pre-multiplying and post-multiplying the operator $I - T_\mu$ of the Bellman equation with D and Φ, respectively. Mathematically, this can be interpreted as follows:

(a) *Post-multiplying with* Φ: We replace the n variables $J(j)$ of the Bellman equation $J = T_\mu J$ with convex combinations of the s variables r_ℓ of the system (2.15), using the rows $(\phi_{j1}, \ldots, \phi_{js})$ of Φ:

$$J(j) \approx \sum_{\ell=1}^{s} \phi_{j\ell}\, r_\ell.$$

(b) *Pre-multiplying with* D: We form the s equations of the aggregate system by taking convex combinations of the n components of the $n \times n$ Bellman equation using the rows of D.

We will now describe how the aggregate system of Eq. (2.17) can be associated with a discounted DP problem that has s states, called the *aggregate states* in what follows. At an abstract level, the aggregate states may be viewed as entities associated with the s rows of D or the s columns of Φ. Indeed, since T_μ has the form $T_\mu J = g_\mu + \alpha P_\mu J$ [cf. Eq. (2.10)], the aggregate system (2.17) becomes

$$r = \hat{g}_\mu + \alpha \hat{P}_\mu r, \tag{2.18}$$

where

$$\hat{g}_\mu = Dg_\mu, \qquad \hat{P}_\mu = DP_\mu\Phi. \tag{2.19}$$

It is straightforward to verify that $\hat{P}_\mu$ is a transition probability matrix, since the rows of D and Φ are probability distributions. This means that the aggregation equation (2.18) [or equivalently Eq. (2.17)] represents a policy evaluation/Bellman equation for the discounted problem with transition matrix $\hat{P}_\mu$ and cost vector $\hat{g}_\mu$. This problem will be called the *aggregate DP problem* associated with policy μ in what follows. The corresponding aggregate state costs are $r_1, \ldots, r_s$. Some important consequences of this are:

(a) The aggregation equation (2.18)-(2.19) inherits the favorable characteristics of the Bellman equation $J = T_\mu J$, namely its monotonicity and contraction properties, and its uniqueness of solution.

(b) Exact DP methods may be used to solve the aggregate DP problem. These methods often have more regular behavior than their counterparts based on projected equations.

(c) Approximate DP methods, such as variants of simulation-based PI, may also be used to solve approximately the aggregate DP problem.

The preceding characteristics of the aggregation approach may be turned to significant advantage, and may counterbalance the restriction on the structure of D and Φ (their rows must be probability distributions, as stated earlier).

2.4. Implementation Issues

The implementation of approximate PI methods involves several delicate issues, which have been extensively investigated but have not been fully resolved, and are the subject of continuing research. We will discuss briefly some of these issues in what follows in this section. We preface this discussion by noting that all of these issues are addressed more easily and effectively within the direct approximation and the aggregation frameworks, than within the temporal difference/projected equation framework, because of the deficiencies relating to the lack of monotonicity and contraction of the operator ΠT_μ, which we noted in Section 2.2.

The Issue of Exploration

An important generic difficulty with simulation-based PI is that in order to evaluate a policy μ, we may need to generate cost samples using that policy, but this may bias the simulation by underrepresenting states that are unlikely to occur under μ. As a result, the cost-to-go estimates of these underrepresented states may be highly inaccurate, causing potentially serious errors in the calculation of the improved control policy $\hat{\mu}$ via the policy improvement equation (2.1).

The situation just described is known as *inadequate exploration* of the system's dynamics. It is a particularly acute difficulty when the system is deterministic [i.e., $p_{ij}(u)$ is equal to 1 for a single successor state j], or when the randomness embodied in the transition probabilities of the current policy is "relatively small," since then few states may be reached from a given initial state when the current policy is simulated.

One possibility to guarantee adequate exploration of the state space is to break down the simulation to multiple short trajectories (see [Ber11c], [Ber12], [YuB12]) and to ensure that the initial states employed form a rich and representative subset. This is naturally done within the direct approximation and the aggregation frameworks, but less so in the temporal difference framework, where the theoretical convergence analysis relies on the generation of a single long trajectory.

Another possibility for exploration is to artificially introduce some extra randomization in the simulation of the current policy, by occasionally generating random transitions using some policy other than μ (this is called an *off-policy approach* and its implementation has been the subject of considerable discussion; see the books [SuB98], [Ber12]). A Monte Carlo tree search implementation may naturally provide some degree of such randomization, and has worked well in game playing contexts, such as the AlphaZero architecture for playing chess, Go, and other games (Silver et al., [SHS17]). Other related approaches to improve exploration based on generating multiple short trajectories are discussed in Sections 6.4.1 and 6.4.2 of [Ber12].

Limited Sampling/Optimistic Policy Iteration

In the approximate PI approach discussed so far, the evaluation of the current policy μ must be fully carried out. An alternative is *optimistic PI*, where relatively few simulation samples are processed between successive

policy changes and corresponding parameter updates.

Optimistic PI with cost function approximation is frequently used in practical applications. In particular, extreme optimistic schemes, including nonlinear architecture versions, and involving a single or very few Q-factor updates between parameter updates have been widely recommended; see e.g., the books [BeT96], [SuB98], [BBD10] (where they are referred to as SARSA, a shorthand for State-Action-Reward-State-Action). The behavior of such schemes is very complex, and their theoretical convergence properties are unclear. In particular, they can exhibit fascinating and counterintuitive behavior, including a natural tendency for policy oscillations. This tendency is common to both optimistic and nonoptimistic PI, as we will discuss shortly, but in extreme optimistic PI schemes, oscillations tend to manifest themselves in an unusual form whereby we may have convergence in parameter space and oscillation in policy space (see [BeT96], Section 6.4.2, or [Ber12], Section 6.4.3).

On the other hand optimistic PI may in some cases deal better with the problem of exploration discussed earlier. The reason is that with rapid changes of policy, there may be less tendency to bias the simulation towards particular states that are favored by any single policy.

Policy Oscillations and Chattering

Contrary to exact PI, which converges to an optimal policy in a fairly regular manner, approximate PI may oscillate. By this we mean that after a few iterations, policies tend to repeat in cycles. The parameter vectors r that correspond to the oscillating policies may also tend to oscillate, although it is possible, in optimistic approximate PI methods, that there is convergence in parameter space and oscillation in policy space, a peculiar phenomenon known as *chattering*.

Oscillations and chattering have been explained with the use of the so-called "greedy partition" of the parameter space into subsets that correspond to the same improved policy (see [BeT96], Section 6.4.2, or [Ber12], Section 6.4.3). Policy oscillations occur when the generated parameter sequence straddles the boundaries that separate sets of the partition. Oscillations can be potentially very damaging, because there is no guarantee that the policies involved in the oscillation are "good" policies, and there is often no way to verify how well they compare to the optimal.

We note that oscillations are avoided and approximate PI can be shown to converge to a single policy under special conditions that arise in particular when aggregation is used for policy evaluation. These conditions involve certain monotonicity assumptions [e.g., the nonnegativity of the components π_{im} of the projection matrix in Eq. (2.12)], which are fulfilled in the case of aggregation (see [Ber11a]). However, for temporal difference methods, policy oscillations tend to occur generically, and often for very simple problems, involving few states (a two-state example is given in [Ber11a], and in [Ber12], Section 6.4.3). This is a potentially important advantage of the aggregation approach.

Model-Free Implementations

In many problems a mathematical model [the transition probabilities $p_{ij}(u)$ and the cost vector g] is unavailable or hard to construct, but instead the system and cost structure can be simulated far more easily. In particular, let us assume that there is a computer program that for any given state i and control u, simulates sample transitions to a successor state j according to $p_{ij}(u)$, and generates the transition cost $g(i,u,j)$.

As noted earlier, the direct and indirect approaches to approximate evaluation of a single policy may be implemented in model-free fashion, simply by generating the needed cost samples for the current policy by simulation. However, given the result $\tilde{J}_\mu(\cdot)$ of the approximate policy evaluation, the policy improvement minimization

$$\hat{\mu}(i) \in \arg\min_{u\in U(i)} \sum_{j=1}^{n} p_{ij}(u)\big(g(i,u,j) + \alpha \tilde{J}_\mu(j)\big), \qquad i = 1,\ldots,n, \tag{2.20}$$

still requires the transition probabilities $p_{ij}(u)$, so it is not model-free. To provide a model-free version we may use a parametric regression approach. In particular, suppose that for any state i and control u, state transitions (i,j), and corresponding transition costs $g(i,u,j)$ and values of $\tilde{J}_\mu(j)$ can be generated in a model-free fashion when needed, by using a simulator of the true system. Then we can introduce a parametric family/approximation architecture of Q-factor functions, $\tilde{Q}_\mu(i,u,\theta)$, where θ is the parameter vector, and use a regularized least squares fit/regression to approximate the expected value that is minimized in Eq. (2.20). The steps are as follows:

(a) Use the simulator to collect a large number of "representative" sample state-control pairs (i_m, u_m), and successor states j_m, $m = 1,\ldots,M$, and corresponding sample Q-factors

$$\beta_m = g(i_m,u_m,j_m) + \alpha \tilde{J}_\mu(j_m), \qquad m = 1,\ldots,M. \tag{2.21}$$

(b) Determine the parameter vector $\tilde{\theta}$ with the least-squares minimization

$$\tilde{\theta} \in \arg\min_{\theta} \sum_{m=1}^{M} \big(\tilde{Q}_\mu(i_m,u_m,\theta) - \beta_m\big)^2 \tag{2.22}$$

(or a regularized minimization whereby a quadratic regularization term is added to the above quadratic objective).

(c) Use the policy

$$\hat{\mu}(i) \in \arg\min_{u\in U(i)} \tilde{Q}_\mu(i,u,\tilde{\theta}), \qquad i = 1,\ldots,n. \tag{2.23}$$

This policy may be generated on-line when the control constraint set $U(i)$ contains a reasonably small number of elements. Otherwise an approximation in policy space is needed to represent the policy $\hat{\mu}$ using a policy approximation architecture. Such an architecture could be based on a neural network,

in which case it is commonly called an "action network" or "actor network" to distinguish from its cost function approximation counterpart, which is called a "value network" or "critic network."

Note some important points about the preceding approximation procedure:

(1) It does not need the transition probabilities $p_{ij}(u)$ to generate the policy $\hat{\mu}$ through the minimization (2.23). The simulator to collect the samples (2.21) suffices.

(2) The policy $\hat{\mu}$ obtained through the minimization (2.23) is not the same as the one obtained through the minimization (2.20). There are two reasons for this. One is the approximation error introduced by the Q-factor architecture $\tilde{Q}_\mu$, and the other is the simulation error introduced by the finite-sample regression (2.22). We have to accept these sources of error as the price to pay for the convenience of not requiring a mathematical model.

(3) Two approximations are potentially required: One to compute $\tilde{J}_\mu$, which is needed for the samples β_m [cf. Eq. (2.21)], and another to compute $\tilde{Q}_\mu$ through the least squares minimization (2.22), and the subsequent policy generation formula (2.23). The approximation methods to obtain $\tilde{J}_\mu$ and $\tilde{Q}_\mu$ may not be the same and in fact may be unrelated (for example $\tilde{J}_\mu$ need not involve a parametric approximation, e.g., it may be obtained by some type of problem approximation approach).

An alternative to first computing $\tilde{J}_\mu(\cdot)$ and then computing subsequently $\tilde{Q}_\mu(\cdot,\cdot,\theta)$ via the procedure (2.21)-(2.23) is to forgo the computation of $\tilde{J}_\mu(\cdot)$, and use just the parametric approximation architecture for the policy Q-factor, $\tilde{Q}_\mu(i,u,\theta)$. We may then train this Q-factor architecture, using state-control Q-factor samples, and either the direct or the indirect approach. Generally, algorithms for approximating policy cost functions can be adapted to approximating policy Q-factor functions.

As an example, a direct model-free approximate PI scheme can be defined by Eqs. (2.22)-(2.23), using M state-control samples (i_m, u_m), corresponding successor states j_m generated according to the probabilities $p_{i_m j}(u_m)$, and sample costs β_m equal to the sum of:

(a) The first stage cost $g(i_m, u_m, j_m)$.

(b) A α-discounted simulated sample of the infinite horizon cost of starting at j_m and using μ [in place of the term $\alpha\tilde{J}_\mu(j_m)$ in Eq. (2.21)].

A PI scheme of this type was suggested by Fern, Yoon, and Givan [FYG06], and has been discussed by several other authors; see [Ber17], Section 6.3.4. In particular, a variant of the method was used to train a tetris playing computer program that performs impressively better than programs that are based on other variants of approximate PI, and various other methods; see Scherrer [Sch13], Scherrer et al. [SGG15], and Gabillon, Ghavamzadeh, and Scherrer [GGS13], who also provide an analysis.

3. APPROXIMATE POLICY EVALUATION BASED ON NEURAL NETWORKS

In this section we will describe some of the basic ideas of the neural network methodology as it applies to the approximation of the cost vector J_μ of a fixed policy μ. Since μ is fixed throughout this section, we drop the subscript μ is what follows. A neural network provides an architecture of the form

$$\tilde{J}(i,v,r) = \sum_{\ell=1}^{s} F_\ell(i,v) r_\ell \tag{3.1}$$

that depends on a parameter vector v and a parameter vector $r = (r_1, \ldots, r_s)$. Here for each state i, $\tilde{J}(i,v,r)$ approximates $J_\mu(i)$, while the vector

$$F(i,v) = \big(F_1(i,v), \ldots, F_s(i,v)\big)$$

may be viewed as a feature vector of the state i. Notice the different roles of the two parameter vectors: v parametrizes $F(i,v)$, and r is a vector of weights that combine linearly the components of $F(i,v)$. The idea is to use training to obtain simultaneously both the features and the linear weights.

Consistent with the direct approximation framework of Section 2.1, to train a neural network, we generate a training set that consists of a large number of state-cost pairs (i_m, β_m), $m = 1, \ldots, M$, and we find (v,r) that minimizes

$$\sum_{m=1}^{M} \big(\tilde{J}(i_m,v,r) - \beta_m\big)^2. \tag{3.2}$$

The training pairs (i_m, β_m) are generated by some kind of calculation or simulation, and they may contain noise, i.e., β_m is the cost of the policy starting from state i_m plus some error.†

The simplest type of neural network is the *single layer perceptron*; see Fig. 3.1. Here the state i is encoded as a vector of numerical values $y(i)$ with components $y_1(i), \ldots, y_k(i)$, which is then transformed linearly as

$$Ay(i) + b,$$

where A is an $m \times k$ matrix and b is a vector in $\Re^m$. Some of the components of $y(i)$ may be known interesting features of i that can be designed based on problem-specific knowledge or prior training experience. This transformation will be referred to as 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 $v = (A,b)$.

† There are also neural network implementations of the indirect/projected equation approximation approach, which make use of temporal differences, such as for example nonlinear versions of TD(λ). We refer to the textbook literature on the subject, e.g., [SuB98]. In this paper, we will focus on neural network training that is based on minimization of the quadratic cost function (3.2).

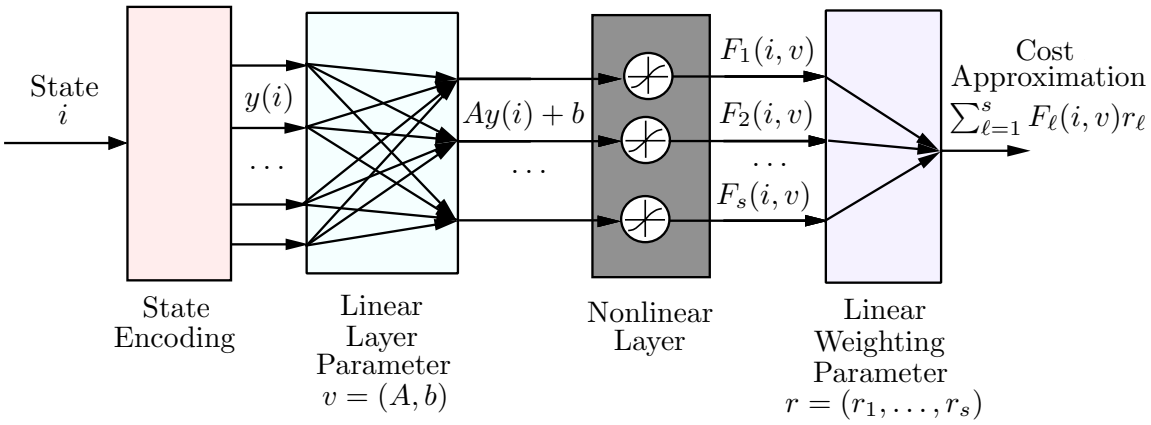


Figure 3.1 A perceptron consisting of a linear layer and a nonlinear layer. It provides a way to compute features of the state, which can be used for approximation of the cost function of a given policy. The state i is encoded as a vector of numerical values $y(i)$, which is then transformed linearly as $Ay(i)+b$ in the linear layer. The scalar output components of the linear layer, become the inputs to single input-single output nonlinear functions that produce the s scalars $F_\ell(i,v)=\sigma\big((Ay(i)+b)_\ell\big)$, which can be viewed as feature components that are in turn linearly weighted with parameters r_ℓ.

Each of the s scalar output components of the linear layer,

$$\big(Ay(i)+b\big)_\ell, \qquad \ell=1,\ldots,s,$$

becomes the input to a nonlinear differentiable function σ that maps scalars to scalars. Typically σ is monotonically increasing. A simple and popular possibility is the *rectified linear unit*, which is simply the function $\max\{0,\xi\}$, "rectified" to a differentiable function by some form of smoothing operation; for example

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

Other functions, used since the early days of neural networks, have the property

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

Such functions are referred to as *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 the differentiability requirement), and simply refer to it as a "nonlinear unit" and to the corresponding layer as a "nonlinear layer."

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

$$F_\ell(i,v)=\sigma\big((Ay(i)+b)_\ell\big), \qquad \ell=1,\ldots,s.$$

One possible interpretation is to view these scalars as features of state i, which are linearly combined using weights r_ℓ, $\ell = 1, \ldots, s$, to produce the final output

$$\sum_{\ell=1}^{s} F_\ell(i, v) r_\ell = \sum_{\ell=1}^{s} \sigma\Big(\big(Ay(i) + b\big)_\ell\Big)\, r_\ell. \tag{3.3}$$

Note that each value $F_\ell(i, v)$ 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.

Given a set of state-cost training pairs (i_m, β_m), $m = 1, \ldots, M$, the parameters of the neural network A, b, and r are obtained by solving the training problem (3.2), i.e.,

$$\min_{A,b,r} \sum_{m=1}^{M} \left(\sum_{\ell=1}^{s} \sigma\Big(\big(Ay(i_m) + b\big)_\ell\Big)\, r_\ell - \beta_m \right)^2. \tag{3.4}$$

The cost function of this problem is generally nonconvex, so it may have multiple local minima.

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 (3.4) is an unconstrained nonconvex differentiable optimization problem that can in principle be addressed with standard gradient-type methods.

Let us now discuss briefly two issues regarding the neural network formulation and training process just described:

(a) A major question is how to solve the training problem (3.4). The salient characteristic of the cost function of this problem is its form as the sum of a potentially very large number M of component functions. This structure can be exploited with a variant of the gradient method, called *incremental*,† which computes just the gradient of a *single* squared error component

$$\left(\sum_{\ell=1}^{s} \sigma\Big(\big(Ay(i_m) + b\big)_\ell\Big)\, r_\ell - \beta_m \right)^2$$

† Sometimes the more recent name "stochastic gradient descent" is used in reference to this method. However, once the training set has been generated, possibly by some deterministic process, the method need not have a stochastic character, and it also does not guarantee cost function descent at each iteration.

of the sum in Eq. (3.4) at each iteration, and then changes the current iterate in the opposite direction of this gradient using some stepsize; the books [Ber15], [Ber16b] provide extensive accounts, and theoretical analyses including the connection with stochastic gradient methods are given in the book [BeT96] and the paper [BeT00]. Experience has shown that the incremental gradient method can be vastly superior to the ordinary (nonincremental) gradient method in the context of neural network training, and in fact the methods most commonly used in practice are incremental.

(b) Another important question is how well we can approximate the cost function of the policy with a neural network architecture, assuming we can choose the number of the nonlinear units s 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 i is an element of a Euclidean space X and $y(i) \equiv i$, 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 s of nonlinear units is sufficiently large. For proofs of the theorem at different levels of generality, we refer to Cybenko [Cyb89], Funahashi [Fun89], Hornik, Stinchcombe, and White [HSW89], and Leshno et al. [LLP93]. For intuitive explanations we refer to Bishop ([Bis95], pp. 129-130) and Jones [Jon90].

While the universal approximation theorem provides some assurance about the adequacy of the neural network structure, it does not predict the number of nonlinear units that 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 numbers of units 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 motivated various 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, as we will now discuss.

Multilayer and Deep Neural Networks

An important generalization of the single layer perceptron architecture is deep neural networks, which involve multiple layers of linear and nonlinear functions. The number of layers can be quite large, hence the "deep" characterization. The outputs of each nonlinear layer become the inputs of the next linear layer; see Fig. 3.2. In some cases it may make sense to add as additional inputs some of the components of the state i or the state encoding $y(i)$.

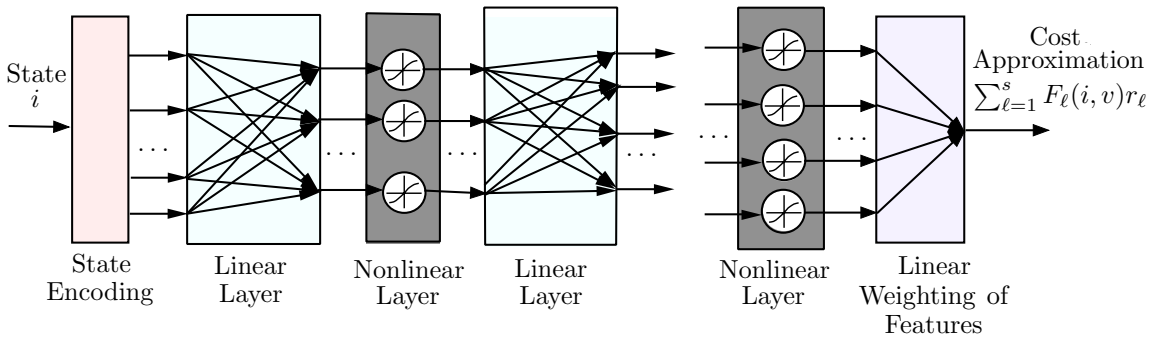


Figure 3.2 A neural network with multiple layers. Each nonlinear layer constructs a set of features as inputs of the next linear layer. The features are obtained at the output of the final nonlinear layer are linearly combined to yield a cost function approximation.

The training problem for multilayer networks has the form

$$\min_{v,r} \sum_{m=1}^{M} \left(\sum_{\ell=1}^{s} F_\ell(i,v) r_\ell - \beta_m \right)^2,$$

where v represents the collection of all the parameters of the linear layers, and $F_\ell(i,v)$ is the ℓth feature component produced at the output of the final nonlinear layer. Various types of incremental gradient methods can also be applied here, specially adapted to the multi-layer structure and they are the methods most commonly used in practice, in combination with techniques for finding good starting points, etc. An important fact is that the gradient with respect to v of each feature component $F_\ell(i,v)$ can be efficiently calculated using a special procedure known as *backpropagation*, which is just a computationally efficient way to apply the chain rule of differentiation. We refer to the specialized literature for various accounts (see e.g., [Bis95], [BeT96], [HOT06], [Hay08]).

In view of the universal approximation property, the reason for having multiple nonlinear layers is not immediately apparent. A commonly given explanation is that a multilayer network provides a hierarchical sequence of features, where each set of features in the sequence is a function of the preceding set of features in the sequence. 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. Another reason commonly given is that with 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 may be drastically decreased.

Deep neural networks also have another advantage, which is important for our aggregation-related purposes in this paper: *the final features obtained as output of the last nonlinear layer tend to be more complex, so their number can be made smaller as the number of nonlinear layers increases*. This tends to facilitate the implementation of the feature-based aggregation schemes that we will discuss in what follows.

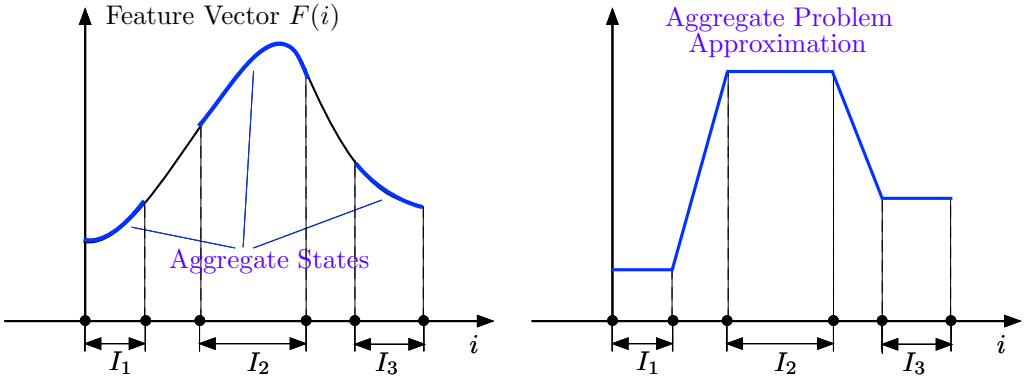


Figure 4.1 Illustration of aggregate states and a corresponding cost approximation, which is constant over each disaggregation set. Here there are three aggregate states, with disaggregation sets denoted I_1, I_2, I_3.

4. FEATURE-BASED AGGREGATION FRAMEWORK

In this section, we will specialize the general aggregation framework of Section 2.3 by introducing features in the definition of the matrices D and Φ. The starting point is a given *feature mapping*, i.e., a function F that maps a state i into its feature vector $F(i)$. We assume that F is constructed in some way (including hand-crafted, or neural network-based), but we leave its construction unspecified for the moment.

We will form a lower-dimensional DP approximation of the original problem, and to this end we introduce disjoint subsets $S_1, \ldots, S_q$ of state-feature pairs $\big(i, F(i)\big)$, which we call *aggregate states*. The subset of original system states I_ℓ that corresponds to S_ℓ,

$$I_\ell = \big\{i \mid (i, F(i)) \in S_\ell\big\}, \qquad \ell = 1, \ldots, q, \tag{4.1}$$

is called the *disaggregation set* of S_ℓ. An alternative and equivalent definition, given F, is to start with disjoint subsets of states I_ℓ, $\ell = 1, \ldots, q$, and define the aggregates states S_ℓ by

$$S_\ell = \big\{(i, F(i)) \mid i \in I_\ell\big\}, \qquad \ell = 1, \ldots, q. \tag{4.2}$$

Mathematically, the aggregate states are the restrictions of the feature mapping on the disaggregation sets I_ℓ. In simple terms, we may view the aggregate states S_ℓ as some "pieces" of the graph of the feature mapping F; see Fig. 4.1.

To preview our framework, we will aim to construct an aggregate DP problem whose states will be the aggregate states $S_1, \ldots, S_q$, and whose optimal costs, denoted $r_1^*, \ldots, r_q^*$, will be used to construct a function approximation $\tilde{J}$ to the optimal cost function J^*. This approximation will be constant over each disaggregation set; see Fig. 4.1. Our ultimate objective is that $\tilde{J}$ approximates closely J^*, which suggests as a general guideline that *the aggregate states should be selected so that J^* is nearly constant over each of the disaggregation sets $I_1, \ldots, I_q$*. This will also be brought out by our subsequent analysis.

To formulate an aggregation model that falls within the framework of Section 2.3, we need to specify the matrices Φ and D. We refer to the row of D that corresponds to aggregate state S_ℓ as the *disaggregation distribution of* S_ℓ and to its elements $d_{\ell 1}, \ldots, d_{\ell n}$ as the *disaggregation probabilities of* S_ℓ. Similarly, we refer to the row of Φ that corresponds to state j, $\{\phi_{j\ell} \mid \ell = 1, \ldots, q\}$, as the *aggregation distribution of* j, and to its elements as the *aggregation probabilities of* j. We impose some restrictions on the components of D and Φ, which we describe next.

Definition of a Feature-Based Aggregation Architecture:

Given the collection of aggregate states $S_1, \ldots, S_q$ and the corresponding disaggregation sets $I_1, \ldots, I_q$, the aggregation and disaggregation probabilities satisfy the following:

(a) The disaggregation probabilities map each aggregate state onto its disaggregation set. By this we mean that the row of the matrix D that corresponds to an aggregate state S_ℓ is a probability distribution $(d_{\ell 1}, \ldots, d_{\ell n})$ over the original system states that assigns zero probabilities to states that are outside the disaggregation set I_ℓ:

$$d_{\ell i} = 0, \qquad \forall\ i \notin I_\ell, \quad \ell = 1, \ldots, q. \tag{4.3}$$

(For example, in the absence of special problem-specific considerations, a reasonable and convenient choice would be to assign equal probability to all states in I_ℓ, and zero probability to all other states.)

(b) The aggregation probabilities map each original system state that belongs to a disaggregation set onto the aggregate state of that set. By this we mean that the row $\{\phi_{j\ell} \mid \ell = 1, \ldots, q\}$ of the matrix Φ that corresponds to an original system state j is specified as follows:

(i) If j belongs to some disaggregation set, say I_ℓ, then

$$\phi_{j\ell} = 1, \tag{4.4}$$

and $\phi_{j\ell'} = 0$ for all $\ell' \neq \ell$.

(ii) If j does not belong to any disaggregation set, the row $\{\phi_{j\ell} \mid \ell = 1, \ldots, q\}$ is an arbitrary probability distribution.

There are several possible methods to choose the aggregate states. Generally, as noted earlier, the idea will be to form disaggregation sets over which the cost function values [$J^*(i)$ or $J_\mu(i)$, depending on

the situation] vary as little as possible. We list three general approaches below, and we illustrate these approaches later with examples:

(a) *State and feature-based approach*: Sample in some way the set of original system states i, compute the corresponding feature vectors $F(i)$, and divide the pairs $\big(i, F(i)\big)$ thus obtained into subsets $S_1, \ldots, S_q$. Some problem-specific knowledge may be used to organize the state sampling, with proper consideration given to issues of sufficient exploration and adequate representation of what is viewed as important parts of the state space. This scheme is suitable for problems where states with similar feature vectors have similar cost function values, and is ordinarily the type of scheme that we would use in conjunction with neural network-constructed features (see Section 5).

(b) *Feature-based approach*: Start with a collection of disjoint subsets F_ℓ, $\ell = 1, \ldots, q$, of the set of all possible feature values

$$\mathcal{F} = \big\{F(i) \mid i = 1, \ldots, n\big\},$$

compute in some way disjoint state subsets $I_1, \ldots, I_q$ such that

$$F(i) \in F_\ell, \qquad \forall\, i \in I_\ell,\ \ell = 1, \ldots, q,$$

and obtain the aggregate states

$$S_\ell = \big\{(i, F(i)) \mid i \in I_\ell\big\}, \qquad \ell = 1, \ldots, q,$$

with corresponding disaggregation sets $I_1, \ldots, I_q$. This scheme is appropriate for problems where it can be implemented so that each disaggregation set I_ℓ consists of states with similar cost function values; examples will be given in Section 4.3.

(c) *State-based approach*: Start with a collection of disjoint subsets of states $I_1, \ldots, I_q$, and introduce an artificial feature vector $F(i)$ that is equal to the index ℓ for the states $i \in I_\ell$, $\ell = 1, \ldots, q$, and to some default index, say 0, for the states that do not belong to $\cup_{\ell=1}^q I_\ell$. Then use as aggregate states the subsets

$$S_\ell = \big\{(i, \ell) \mid i \in I_\ell\big\}, \qquad \ell = 1, \ldots, q,$$

with $I_1, \ldots, I_q$ as the corresponding disaggregation sets. In this scheme, the feature vector plays a subsidiary role, but the idea of using disaggregation subsets with similar cost function values is still central, as we will discuss shortly. (The scheme where the aggregate states are identified with subsets $I_1, \ldots, I_q$ of original system states has been called "aggregation with representative features" in [Ber12], Section 6.5, where its connection with feature-based aggregation has been discussed.)

The approaches of forming aggregate states just described cover most of the aggregation schemes that have been used in practice. Two classical examples of the state-based approach are the following:

Hard Aggregation:

The starting point here is a partition of the state space that consists of disjoint subsets $I_1, \ldots, I_q$ of states with $I_1 \cup \cdots \cup I_q = \{1, \ldots, n\}$. The feature vector $F(i)$ of a state i identifies the set of the partition that i belongs to:

$$F(i) = \ell, \qquad \forall\, i \in I_\ell,\ \ell = 1, \ldots, q. \tag{4.5}$$

The aggregate states are the subsets

$$S_\ell = \big\{(i, \ell) \mid i \in I_\ell\big\}, \qquad \ell = 1, \ldots, q,$$

and their disaggregation sets are the subsets $I_1, \ldots, I_q$. The disaggregation probabilities $d_{i\ell}$ are positive only for states $i \in I_\ell$ [cf. Eq. (4.3)]. The aggregation probabilities are equal to either 0 or 1, according to

$$\phi_{j\ell} = \begin{cases} 1 & \text{if } j \in I_\ell, \\ 0 & \text{otherwise,} \end{cases} \qquad j = 1, \ldots, n,\ \ell = 1, \ldots, q, \tag{4.6}$$

[cf. Eq. (4.4)].

The following aggregation example is typical of a variety of schemes arising in discretization or coarse grid schemes, where a smaller problem is obtained by discarding some of the original system states. The essence of this scheme is to solve a reduced DP problem, obtained by approximating the discarded state costs by interpolation using the nondiscarded state costs.

Aggregation with Representative States:

The starting point here is a collection of states $i_1, \ldots, i_q$ that we view as "representative." The costs of the nonrepresentative states are approximated by interpolation of the costs of the representative states, using the aggregation probabilities. The feature mapping is

$$F(i) = \begin{cases} \ell & \text{if } i = i_\ell,\ \ell = 1, \ldots, q, \\ 0 & \text{otherwise.} \end{cases} \tag{4.7}$$

The aggregate states are $S_\ell = \big\{(i_\ell, \ell)\big\}$, $\ell = 1, \ldots, q$, the disaggregation sets are $I_\ell = \{i_\ell\}$, $\ell = 1, \ldots, q$, and the disaggregation probabilities are equal to either 0 or 1, according to

$$d_{\ell i} = \begin{cases} 1 & \text{if } i = i_\ell, \\ 0 & \text{otherwise,} \end{cases} \qquad i = 1, \ldots, n,\ \ell = 1, \ldots, q,$$

[cf. Eq. (4.3)]. The aggregation probabilities must satisfy the constraint $\phi_{j\ell} = 1$ if $j = i_\ell$, $\ell = 1, \ldots, q$ [cf. Eq. (4.4)], and can be arbitrary for states $j \notin \{i_1, \ldots, i_q\}$.

An important class of aggregation frameworks with representative states arises in partially observed Markovian decision problems (POMDP), where observations from a controlled Markov chain become available sequentially over time. Here the states of the original high-dimensional DP problem are either information vectors (groups of past measurements) or "belief states" (conditional probability distributions of the state of the Markov chain given the available information). Features may be state estimates (given the information) and possibly their variances, or a relatively small number of representative belief states (see e.g., Section 5.1 of [Ber12] or the paper by Yu and Bertsekas [YuB04] and the references quoted there).

Choice of Disaggregation Probabilities

In both of the preceding aggregation schemes, the requirement $d_{\ell i} = 0$ for all $i \notin I_\ell$, cf. Eq. (4.3), leaves a lot of room for choice of the disaggregation probabilities. Simple examples show that the values of these probabilities can affect significantly the quality of aggregation-based approximations; the paper by Van Roy [Van06] provides a relevant discussion. Thus, finding a good set of disaggregation probabilities is an interesting issue.

Generally, problem-specific knowledge and intuition can be helpful in designing aggregation schemes, but more systematic methods may be desirable, based on some kind of gradient or random search optimization. In particular, for a given set of aggregate states and matrix Φ, we may introduce a parameter vector θ and a parametrized disaggregation matrix $D(\theta)$, which is differentiable with respect to θ. Then for a given policy μ, we may try to find θ that minimizes some cost function $F\big(\Phi r(\theta)\big)$, where $r(\theta)$ is defined as the unique solution of the corresponding aggregation equation $r = D(\theta)T_\mu \Phi r$. For example we may use as cost function F the squared Bellman equation residual

$$F\big(\Phi r(\theta)\big) = \big\|\Phi r(\theta) - \Phi D(\theta) T_\mu \Phi r(\theta)\big\|^2.$$

The key point here is that we can calculate the gradient of $r(\theta)$ with respect to each component of θ by using simulation and low-dimensional calculations based on aggregation. We can then use the chain rule to compute the gradient of F with respect to θ for use in some gradient-based optimization method. This methodology has been developed for a related projected equation context by Menache, Mannor, and Shimkin [MMS06], Yu and Bertsekas [YuB09], and Di Castro and Mannor [DiM10], but has never been tried in the context of aggregation. The paper [MMS06] also suggests the use of random search algorithms, such as the cross entropy method, in the context of basis function optimization. A further discussion of parametric

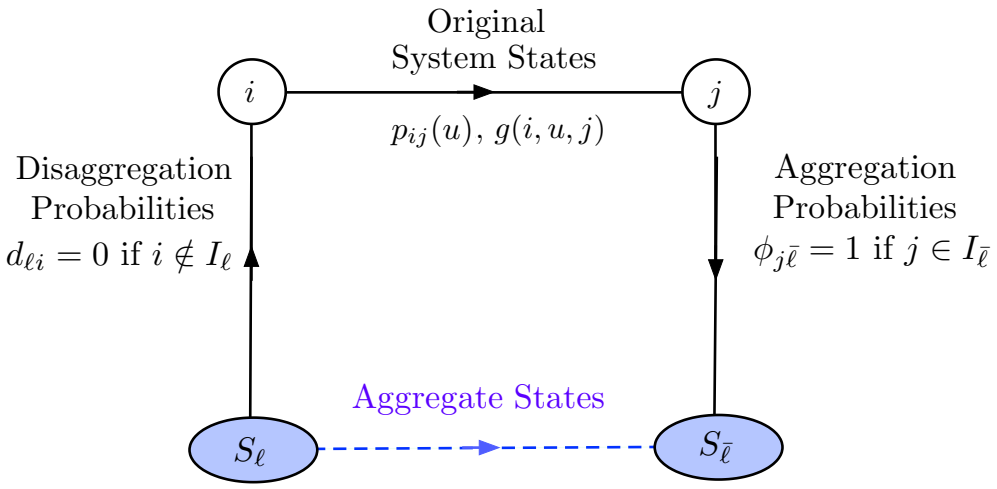


Figure 4.2. Illustration of the transition mechanism and the costs per stage of the aggregate problem.

optimization of the disaggregation probabilities or other structural elements of the aggregation framework is beyond the scope of the present paper, but may be an interesting subject for investigation.

4.1. The Aggregate Problem

Given a feature-based aggregation framework (i.e., the aggregate states $S_1, \ldots, S_q$, the corresponding disaggregation sets $I_1, \ldots, I_q$, and the aggregation and disaggregation distributions), we can consider an aggregate DP problem that involves transitions between aggregate states. In particular, the transition probabilities $p_{ij}(u)$, and the disaggregation and aggregation probabilities specify a controlled dynamic system involving both the original system states and the aggregate states (cf. Fig. 4.2).†

(i) From aggregate state S_ℓ, we generate a transition to original system state i according to $d_{\ell i}$ (note that i must belong to the disaggregation set I_ℓ, because of the requirement that $d_{\ell i} > 0$ only if $i \in I_\ell$).

(ii) From original system state i, we generate a transition to original system state j according to $p_{ij}(u)$, with cost $g(i,u,j)$.

(iii) From original system state j, we generate a transition to aggregate state S_ℓ according to $\phi_{j\ell}$ [note here the requirement that $\phi_{j\ell} = 1$ if $j \in I_\ell$; cf. Eq. (4.4)].

† We will consider the aggregate problem for the case where there are multiple possible controls at each state. However, it is also possible to consider the aggregate problem for the purpose of finding an approximation to the cost function J_μ of a given policy μ; this is the special case where the control constraint set $U(i)$ consists of the single control $\mu(i)$ for every state i.

This is a DP problem with an enlarged state space that consists of two copies of the original state space $\{1,\ldots,n\}$ plus the q aggregate states. We introduce the corresponding optimal vectors $\tilde{J}_0$, $\tilde{J}_1$, and $r^* = \{r_1^*,\ldots,r_q^*\}$ where:

r_ℓ^* is the optimal cost-to-go from aggregate state S_ℓ.

$\tilde{J}_0(i)$ is the optimal cost-to-go from original system state i that has just been generated from an aggregate state (left side of Fig. 4.3).

$\tilde{J}_1(j)$ is the optimal cost-to-go from original system state j that has just been generated from an original system state (right side of Fig. 4.3).

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

These three vectors satisfy the following three Bellman's equations:

$$r_\ell^* = \sum_{i=1}^{n} d_{\ell i}\tilde{J}_0(i), \qquad \ell = 1,\ldots,q, \tag{4.8}$$

$$\tilde{J}_0(i) = \min_{u\in U(i)} \sum_{j=1}^{n} p_{ij}(u)\big(g(i,u,j) + \alpha\tilde{J}_1(j)\big), \qquad i = 1,\ldots,n, \tag{4.9}$$

$$\tilde{J}_1(j) = \sum_{m=1}^{q} \phi_{j\ell} r_m^*, \qquad j = 1,\ldots,n. \tag{4.10}$$

By combining these equations, we see that r^* satisfies

$$r_\ell^* = \sum_{i=1}^{n} d_{\ell i} \min_{u\in U(i)} \sum_{j=1}^{n} p_{ij}(u)\left(g(i,u,j) + \alpha\sum_{m=1}^{q}\phi_{jm}\, r_m^*\right), \qquad \ell = 1,\ldots,q, \tag{4.11}$$

or equivalently $r^* = Hr^*$, where H is the mapping that maps the vector r to the vector Hr with components

$$(Hr)(\ell) = \sum_{i=1}^{n} d_{\ell i} \min_{u\in U(i)} \sum_{j=1}^{n} p_{ij}(u)\left(g(i,u,j) + \alpha\sum_{m=1}^{q}\phi_{jm}\, r_m\right), \qquad \ell = 1,\ldots,q. \tag{4.12}$$

It can be shown that H is a contraction mapping with respect to the sup-norm and thus has r^* as its unique fixed point. This follows from standard contraction arguments, and the fact that $d_{\ell i}$, $p_{ij}(u)$, and $\phi_{j\ell}$ are probabilities. Note the nature of r_ℓ^*: it is the optimal cost of the aggregate state S_ℓ, which is the restriction of the feature mapping F on the disaggregation set I_ℓ. Thus, roughly, r_ℓ^* is an approximate optimal cost associated with states in I_ℓ.

Solution of the Aggregate Problem

While the aggregate problem involves more states than the original DP problem, it is in fact easier in some important ways. The reason is that *it can be solved with algorithms that execute over the smaller space*

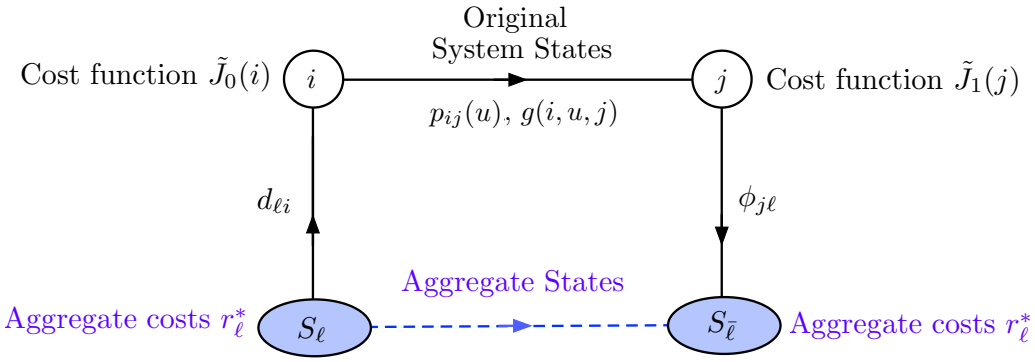


Figure 4.3. The transition mechanism and the cost functions of the aggregate problem.

of aggregate states. In particular, exact and approximate simulation-based algorithms, can be used to find the lower-dimensional vector r^* without computing the higher-dimensional vectors $\tilde{J}_0$ and $\tilde{J}_1$. We describe some of these methods in Section 4.2, and we refer to Chapter 6 of [Ber12] for a more detailed discussion of simulation-based methods for computing the vector r_μ of the costs of the aggregate states that correspond to a given policy μ. The simulator used for these methods is based on Figs. 4.2 and 4.3: transitions to and from the aggregate states are generated using the aggregation and disaggregation probabilities, respectively, while transitions (i, j) between original system states are generated using a simulator of the original system (which is assumed to be available).

Once r^* is found, the optimal-cost-to-go of the original problem may be approximated by the vector $\tilde{J}_1$ of Eq. (4.10). Note that $\tilde{J}_1$ is a "piecewise linear" cost approximation of J^*: it is constant over each of the disaggregation sets I_ℓ, $\ell = 1, \ldots, q$ [and equal to the optimal cost r^*_ℓ of the aggregate state S_ℓ; cf. Eqs. (4.4) and (4.10)], and it is interpolated/linear outside the disaggregation sets [cf. Eq. (4.10)]. In the case where $\cup_{\ell=1}^q I_\ell = \{1, \ldots, n\}$ (e.g., in hard aggregation), the disaggregation sets I_ℓ form a partition of the original system state space, and $\tilde{J}_1$ is piecewise constant. Figure 4.4 illustrates a simple example of approximate cost function $\tilde{J}_1$.

Let us also note that for the purposes of using feature-based aggregation to improve a given policy μ, it is not essential to solve the aggregate problem to completion. Instead, we may perform one or just a few PIs and adopt the final policy obtained as a new "improved" policy. The quality of such a policy depends on how well the aggregate problem approximates the original DP problem. While it is not easy to quantify the relevant approximation error, generally a small error can be achieved if:

(a) The feature mapping F "conforms" to the optimal cost function J^* in the sense that F varies little in regions of the state space where J^* also varies little.

(b) The aggregate states are selected so that F varies little over each of the disaggregation sets $I_1, \ldots, I_q$.

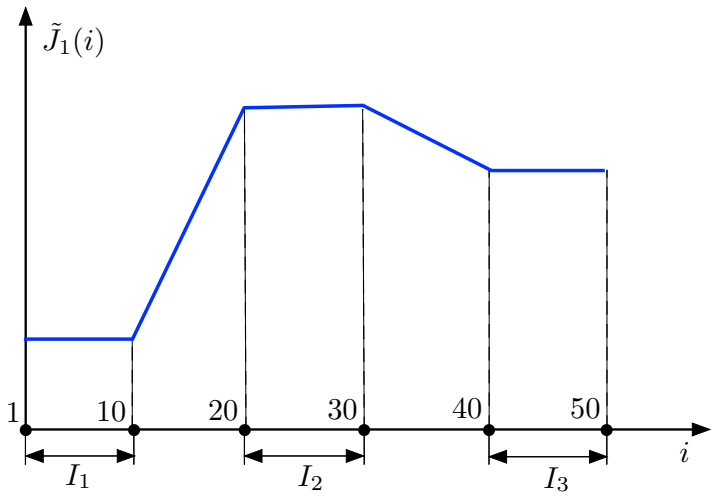


Figure 4.4. Schematic illustration of the approximate cost function $\tilde{J}_1$. Here the original states are the integers between 1 and 50. In this figure there are three aggregate states numbered $1, 2, 3$. The corresponding disaggregation sets are $I_1 = \{1, \ldots, 10\}$, $I_2 = \{20, \ldots, 30\}$, $I_3 = \{40, \ldots, 50\}$ are shown in the figure. The values of the approximate cost function $\tilde{J}_1(i)$ are constant within each disaggregation set I_ℓ, $\ell = 1, 2, 3$, and are obtained by linear interpolation for states i that do not belong to any one of the sets I_ℓ. If the sets I_ℓ, $\ell = 1, 2, 3$, include all the states $1, \ldots, 50$, we have a case of hard aggregation. If each of the sets I_ℓ, $\ell = 1, 2, 3$, consist of a single state, we have a case of aggregation with representative states.

This is intuitive and is supported by the subsequent discussion and analysis.

Given the optimal aggregate costs r^*_ℓ, $\ell = 1, \ldots, q$, the corresponding optimal policy is defined implicitly, using the one-step lookahead minimization

$$\hat{\mu}(i) \in \arg\min_{u \in U(i)} \sum_{j=1}^{n} p_{ij}(u) \left(g(i, u, j) + \alpha \sum_{\ell=1}^{q} \phi_{j\ell}\, r^*_\ell \right), \qquad i = 1, \ldots, n, \tag{4.13}$$

[cf. Eq. (4.9)] or a multistep lookahead variant. It is also possible to use a model-free implementation, as we describe next.

Model-Free Implementation of the Optimal Policy of the Aggregate Problem

The computation of the optimal policy of the aggregate problem via Eq. (4.13) requires knowledge of the transition probabilities $p_{ij}(u)$ and the cost function g. Alternatively, this policy may be implemented in model-free fashion using a Q-factor architecture $\tilde{Q}(i, u, \theta)$, as described in Section 2.4, i.e., compute sample approximate Q-factors

$$\beta_m = g(i_m, u_m, j_m) + \alpha \sum_{\ell=1}^{q} \phi_{j_m \ell}\, r^*_\ell, \qquad m = 1, \ldots, M, \tag{4.14}$$

cf. Eq. (2.21), compute $\tilde{\theta}$ via a least squares regression

$$\tilde{\theta} \in \arg\min_{\theta} \sum_{m=1}^{M} \big(\tilde{Q}(i_m, u_m, \theta) - \beta_m \big)^2 \tag{4.15}$$

(or a regularized version thereof), cf. Eq. (2.22), and approximate the optimal policy of the aggregate problem via

$$\hat{\mu}(i) \in \arg\min_{u \in U(i)} \tilde{Q}(i,u,\tilde{\theta}), \qquad i = 1, \ldots, n, \tag{4.16}$$

cf. Eq. (2.23).

Error Bounds

Intuitively, if the disaggregation sets nearly cover the entire state space (in the sense that $\cup_{\ell=1,\ldots,q} I_\ell$ contains "most" of the states $1, \ldots, n$) and J^* is nearly constant over each disaggregation set, then $\tilde{J}_0$ and $\tilde{J}_1$ should be close to J^*. In particular, in the case of hard aggregation, we have the following error bound, due to Tsitsiklis and VanRoy [TsV96]. We adapt their proof to the notation and terminology of this paper.

Proposition 4.1: In the case of hard aggregation, where $\cup_{\ell=1}^{q} I_\ell = \{1, \ldots, n\}$, and Eqs. (4.5), (4.6) hold, we have

$$\big|J^*(i) - r^*_\ell\big| \leq \frac{\epsilon}{1-\alpha}, \qquad \forall\ i \text{ such that } i \in I_\ell,\ \ell = 1, \ldots, q, \tag{4.17}$$

where

$$\epsilon = \max_{\ell=1,\ldots,q} \max_{i,j \in I_\ell} \big|J^*(i) - J^*(j)\big|. \tag{4.18}$$

Proof: Consider the mapping H defined by Eq. (4.12), and consider the vector $\overline{r}$ with components defined by

$$\overline{r}_\ell = \min_{i \in I_\ell} J^*(i) + \frac{\epsilon}{1-\alpha}, \qquad \ell \in 1, \ldots, q.$$

Denoting by $\ell(j)$ the index of the disaggregation set to which j belongs, i.e., $j \in I_{\ell(j)}$, we have for all ℓ,

$$\begin{aligned}
(H\overline{r})(\ell) &= \sum_{i=1}^{n} d_{\ell i} \min_{u \in U(i)} \sum_{j=1}^{n} p_{ij}(u)\Big(g(i,u,j) + \alpha \overline{r}_{\ell(j)}\Big) \\
&\leq \sum_{i=1}^{n} d_{\ell i} \min_{u \in U(i)} \sum_{j=1}^{n} p_{ij}(u)\left(g(i,u,j) + \alpha J^*(j) + \frac{\alpha\epsilon}{1-\alpha}\right) \\
&= \sum_{i=1}^{n} d_{\ell i}\left(J^*(i) + \frac{\alpha\epsilon}{1-\alpha}\right) \\
&\leq \min_{i \in I_\ell}\big(J^*(i) + \epsilon\big) + \frac{\alpha\epsilon}{1-\alpha} \\
&= \min_{i \in I_\ell} J^*(i) + \frac{\epsilon}{1-\alpha} \\
&= \overline{r}_\ell,
\end{aligned}$$

where for the second equality we used the Bellman equation for the original system, which is satisfied by J^*, and for the second inequality we used Eq. (4.18). Thus we have $H\overline{r} \leq \overline{r}$, from which it follows that $r^* \leq \overline{r}$ (since H is monotone, which implies that the sequence $\{H^k\overline{r}\}$ is monotonically nonincreasing, and we have

$$r^* = \lim_{k\to\infty} H^k\overline{r}$$

since H is a contraction). This proves one side of the desired error bound. The other side follows similarly. **Q.E.D.**

The scalar ϵ of Eq. (4.18) is the maximum variation of optimal cost within the sets of the partition of the hard aggregation scheme. Thus the meaning of the preceding proposition is that if the optimal cost function J^* varies by at most ϵ within each set of the partition, the hard aggregation scheme yields a piecewise constant approximation to the optimal cost function that is within $\epsilon/(1-\alpha)$ of the optimal. We know that for every approximation $\tilde{J}$ of J^* that is constant within each disaggregation set, the error

$$\max_{i=1,\ldots,n} \left|J^*(i) - \tilde{J}(i)\right|$$

is at least equal to $\epsilon/2$. Based on the bound (4.17), the actual value of this error for the case where $\tilde{J}$ is obtained by hard aggregation involves an additional multiplicative factor that is at most equal to $2/(1-\alpha)$, and depends on the disaggregation probabilities. In practice the bound (4.17) is typically conservative, and no examples are known where it is tight. Moreover, even for hard aggregation, the manner in which the error $J^* - \tilde{J}_1$ depends on the disaggregation distributions is complicated and is an interesting subject for research.

The following proposition extends the result of the preceding proposition to the case where the aggregation probabilities are all either 0 or 1, in which case the cost function $\tilde{J}_1$ obtained by aggregation is a piecewise constant function, but the disaggregation sets need not form a partition of the state space. Examples of this type of scheme include cases where the aggregation probabilities are generated by a "nearest neighbor" scheme, and the cost $\tilde{J}_1(j)$ of a state $j \notin \cup_{\ell=1}^q I_\ell$ is taken to be equal to the cost of the "nearest" state within $\cup_{\ell=1}^q I_\ell$.

Proposition 4.2: Assume that each aggregation probability $\phi_{j\ell}$, $j = 1,\ldots,n$, $\ell = 1,\ldots,q$, is equal to either 0 or 1, and consider the sets

$$\hat{I}_\ell = \{j \mid \phi_{j\ell} = 1\}, \qquad \ell = 1,\ldots,q.$$

Then we have

$$\big|J^*(i) - r^*_\ell\big| \leq \frac{\epsilon}{1-\alpha}, \qquad \forall\ i \in \hat{I}_\ell,\ \ell = 1, \ldots, q,$$

where

$$\epsilon = \max_{\ell=1,\ldots,q} \max_{i,j \in \hat{I}_\ell} \big|J^*(i) - J^*(j)\big|.$$

Proof: We first note that by the definition of a feature-based aggregation scheme, we have $I_\ell \subset \hat{I}_\ell$ for all $\ell = 1, \ldots, q$, while the sets $\hat{I}_\ell$, $\ell = 1, \ldots, q$, form a partition of the original state space, in view of our assumption on the aggregation probabilities. Let us replace the feature vector F with another feature vector $\hat{F}$ of the form

$$\hat{F}(i) = \ell, \qquad \forall\ i \in \hat{I}_\ell,\ \ell = 1, \ldots, q.$$

Since the aggregation probabilities are all either 0 or 1, the resulting aggregation scheme with I_ℓ replaced by $\hat{I}_\ell$, and with the aggregation and disaggregation probabilities remaining unchanged, is a hard aggregation scheme. When the result of Prop. 4.1 is applied to this hard aggregation scheme, the result of the present proposition follows. **Q.E.D.**

The preceding propositions suggest the principal guideline for a feature-based aggregation scheme. It should be designed so that states that belong to the same disaggregation set have nearly equal optimal costs. In Section 4.3 we will elaborate on schemes that are based on this idea. In the next section we discuss the solution of the aggregate problem by simulation-based methods.

4.2. Solving the Aggregate Problem with Simulation-Based Methods

We will now focus on methods to compute the optimal cost vector r^* of the aggregate problem that corresponds to the aggregate states. This is the unique solution of Eq. (4.11). We first note that since r^*, together with the cost functions $\tilde{J}_0$ and $\tilde{J}_1$, form the solution of the Bellman equations (4.8)-(4.10), they can all be computed with the classical (exact) methods of policy and value iteration (PI and VI for short, respectively). However, in this section, we will discuss specialized versions of PI and VI that compute just r^* (which has relatively low dimension), but not $\tilde{J}_0$ and $\tilde{J}_1$ (which may have astronomical dimension). These methods are based on stochastic simulation as they involve the aggregate problem, which is stochastic because of the disaggregation and aggregation probabilities, even if the original problem is deterministic.

We start with simulation-based versions of PI, where policy evaluation is done with lookup table versions of classical methods such as LSTD(0), LSPE(0), and TD(0), applied to a reduced size DP problem whose states are just the aggregate states.

Simulation-Based Policy Iteration

One possible way to compute r^* is a PI-like algorithm, which generates sequences of policies $\{\mu^k\}$ for the original problem and vectors $\{r^k\}$ that converge to an optimal policy and r^*, respectively. The algorithm does not compute any intermediate estimates of the high-dimensional vectors $\tilde{J}_0$ and $\tilde{J}_1$. It starts with a stationary policy μ^0 for the original problem, and given μ^k, it performs a policy evaluation step by finding the unique fixed point of the contraction mapping $H_{\mu^k} = DT_{\mu^k}\Phi$ that maps the vector r to the vector $H_{\mu^k}r$ with components

$$(H_{\mu^k}r)(\ell) = \sum_{i=1}^{n} d_{\ell i} \sum_{j=1}^{n} p_{ij}\big(\mu^k(i)\big)\left(g\big(i,\mu^k(i),j\big) + \alpha \sum_{m=1}^{q} \phi_{jm}\, r_m\right), \qquad \ell = 1,\ldots,q,$$

cf. Eq. (4.12). Thus the policy evaluation step finds $r^k = \{r^k_\ell \mid \ell = 1,\ldots,q\}$ satisfying

$$r^k = H_{\mu^k}r^k = DT_{\mu^k}\Phi r^k = D\big(g_{\mu^k} + \alpha P_{\mu^k}\Phi r^k\big), \tag{4.19}$$

where P_{μ^k} is the transition probability matrix corresponding to μ^k, g_{μ^k} is the expected cost vector of μ^k, i.e., the vector whose ith component is

$$\sum_{j=1}^{n} p_{ij}\big(\mu^k(i)\big)g\big(i,\mu^k(i),j\big), \qquad i = 1,\ldots,n,$$

and D and Φ are the matrices with rows the disaggregation and aggregation distributions, respectively. Following the policy evaluation step, the algorithm generates μ^{k+1} by

$$\mu^{k+1}(i) = \arg\min_{u\in U(i)} \sum_{j=1}^{n} p_{ij}(u)\left(g(i,u,j) + \alpha \sum_{m=1}^{q} \phi_{jm} r^k_m\right), \qquad i = 1,\ldots,n; \tag{4.20}$$

this is the policy improvement step. In the preceding minimization we use one step lookahead, but a multistep lookahead or Monte Carlo tree search can also be used.

It can be shown that this algorithm converges finitely to the unique solution of Eq. (4.11) [equivalently the unique fixed point of the mapping H of Eq. (4.12)]. An indirect way to show this is to use the convergence of PI applied to the aggregate problem to generate a sequence $\{\mu^k, r^k, \tilde{J}^k_0, \tilde{J}^k_1\}$. We provide a more direct proof, which is essentially a special case of a more general convergence proof for PI-type methods given in Prop. 3.1 of [Ber11a]. The key fact here is that the linear mappings $DT_\mu\Phi$ and ΦDT_μ are sup-norm contractions, and also have the monotonicity property of DP mappings, which is used in an essential way in the standard convergence proof of ordinary PI.

Proposition 4.3: Let μ^0 be any policy and let $\{\mu^k, r^k\}$ be a sequence generated by the PI algorithm (4.19)-(4.20). Then the sequence $\{r^k\}$ is monotonically nonincreasing (i.e., we have $r^k_\ell \geq r^{k+1}_\ell$ for all ℓ and k) and there exists an index $\bar{k}$ such that $r^{\bar{k}}$ is equal to r^*, the unique solution of Eq. (4.11).

Proof: For each policy μ, we consider the linear mapping $\Phi DT_\mu : \Re^n \mapsto \Re^n$ given by

$$\Phi DT_\mu J = \Phi D(g_\mu + \alpha P_\mu J), \qquad J \in \Re^n.$$

This mapping is monotone in the sense that for all vectors J and J' with $J \geq J'$, we have

$$\Phi DT_\mu J \geq \Phi DT_\mu J',$$

since the matrix $\alpha\Phi DP_\mu$ of the mapping has nonnegative components. Moreover, the mapping is a contraction of modulus α with respect to the sup-norm. The reason is that the matrix ΦDP_μ is a transition probability matrix, i.e., it has nonnegative components and its row sums are all equal to 1. This can be verified by a straightforward calculation, using the fact that the rows of Φ and D are probability distributions while P_μ is a transition probability matrix. It can also be intuitively verified from the structure of the aggregate problem: ΦDP_μ is the matrix of transition probabilities under policy μ for the Markov chain whose n states are the states depicted in the top righthand side of Fig. 4.2.

Since the mapping $DT_{\mu^k}\Phi$ has r^k as its unique fixed point [cf. Eq. (4.19)], we have $r^k = DT_{\mu^k}\Phi r^k$, so that the vector

$$\tilde{J}_{\mu^k} = \Phi r^k$$

satisfies

$$\tilde{J}_{\mu^k} = \Phi DT_{\mu^k}\tilde{J}_{\mu^k}.$$

It follows that $\tilde{J}_{\mu^k}$ is the unique fixed point of the contraction mapping ΦDT_{μ^k}. By using the definition

$$T_{\mu^{k+1}}\tilde{J}_{\mu^k} = T\tilde{J}_{\mu^k}$$

of μ^{k+1} [cf. Eq. (4.20)], we have

$$\tilde{J}_{\mu^k} = \Phi DT_{\mu^k}\tilde{J}_{\mu^k} \geq \Phi DT\tilde{J}_{\mu^k} = \Phi DT_{\mu^{k+1}}\tilde{J}_{\mu^k}. \tag{4.21}$$

Applying repeatedly the monotone mapping $\Phi DT_{\mu^{k+1}}$ to this relation, we have for all $m \geq 1$,

$$\tilde{J}_{\mu^k} \geq (\Phi DT_{\mu^{k+1}})^m\tilde{J}_{\mu^k} \geq \lim_{m\to\infty}(\Phi DT_{\mu^{k+1}})^m\tilde{J}_{\mu^k} = \tilde{J}_{\mu^{k+1}}, \tag{4.22}$$

where the equality follows from the fact that $\tilde{J}_{\mu^{k+1}}$ is the fixed point of the contraction mapping $\Phi DT_{\mu^{k+1}}$. It follows that $\tilde{J}_{\mu^k} \geq \tilde{J}_{\mu^{k+1}}$, or equivalently

$$\Phi r^k \geq \Phi r^{k+1}, \qquad k = 0, 1, \ldots.$$

By the definition of the feature-based aggregation architecture [cf. Eq. (4.4)], each column of Φ has at least one component that is equal to 1. Therefore we have for all k

$$r^k \geq r^{k+1},$$

and moreover, the equality $r^k = r^{k+1}$ holds if and only if $\tilde{J}_{\mu^k} = \tilde{J}_{\mu^{k+1}}$.

The inequality $\tilde{J}_{\mu^k} \geq \tilde{J}_{\mu^{k+1}}$ implies that as long as $\tilde{J}_{\mu^k} \neq \tilde{J}_{\mu^{k+1}}$, a policy μ^k cannot be repeated. Since there is only a finite number of policies, it follows that we must eventually have $\tilde{J}_{\mu^k} = \tilde{J}_{\mu^{k+1}}$. In view of Eqs. (4.21)-(4.22), we see that

$$\tilde{J}_{\mu^k} = \Phi DT\tilde{J}_{\mu^k}$$

or

$$\Phi r^k = \Phi DT\Phi r^k.$$

Since each column of Φ has at least one component that is equal to 1, it follows that r^k is a fixed point of the mapping $H = DT\Phi$ of Eq. (4.12), which is r^* by Eq. (4.11). **Q.E.D.**

To avoid the n-dimensional calculations of the policy evaluation step in the PI algorithm (4.19)-(4.20), one may use simulation. In particular, the policy evaluation equation, $r = H_\mu r$, is linear of the form

$$r = Dg_\mu + \alpha DP_\mu \Phi r, \tag{4.23}$$

[cf. Eq. (4.19)]. Let us write this equation as $Cr = b$, where

$$C = I - \alpha DP_\mu \Phi, \qquad b = Dg_\mu,$$

and note that it is Bellman's equation for a policy with cost per stage vector equal to Dg_μ and transition probability matrix equal to $DP_\mu\Phi$. This is the transition matrix under policy μ for the Markov chain whose states are the aggregate states. The solution r_μ of the policy evaluation Eq. (4.23) is the cost vector corresponding to this Markov chain, and can be found by using simulation-based methods with lookup table representation.

In particular, we may use model-free simulation to approximate C and b, and then solve the system $Cr = b$ approximately. To this end, we obtain a sequence of sample transitions $\big\{(i_1, j_1), (i_2, j_2), \dots\big\}$ by first generating a sequence of states $\{i_1, i_2, \dots\}$ according to some distribution $\{\xi_i \mid i = 1, \dots, n\}$ (with $\xi_i > 0$ for all i), and then generate for each $m \geq 1$ a sample transition (i_m, j_m) according to the distribution $\{p_{i_m j} \mid j = 1, \dots, n\}$. Given the first M samples, we form the matrix $\widehat{C}_M$ and vector $\hat{b}_M$ given by

$$\widehat{C}_M = I - \frac{\alpha}{M}\sum_{m=1}^{M}\frac{1}{\xi_{i_m}}d(i_m)\phi(j_m)', \qquad \hat{b}_M = \frac{1}{M}\sum_{m=1}^{M}\frac{1}{\xi_{i_m}}d(i_m)g\big(i_m, \mu(i_m), j_m\big), \tag{4.24}$$

where $d(i)$ is the ith column of D and $\phi(j)'$ is the jth row of Φ. We can then show that $\widehat{C}_M \to C$ and $\hat{b}_M \to d$ by using law of large numbers arguments, i.e., writing

$$C = I - \alpha\sum_{i=1}^{n}\sum_{j=1}^{n}p_{ij}\big(\mu(i)\big)d(i)\phi(j)', \qquad b = \sum_{i=1}^{n}\sum_{j=1}^{n}p_{ij}\big(\mu(i)\big)d(i)g\big(i, \mu(i), j\big),$$

multiplying and dividing $p_{ij}\big(\mu(i)\big)$ by ξ_i in order to properly view these expressions as expected values, and using the relation

$$\lim_{M\to\infty}\frac{\text{Number of occurrences of the } i \text{ to } j \text{ transition from time } m=1 \text{ to } m=M}{M}=\xi_i\,p_{ij}\big(\mu(i)\big).$$

The corresponding estimates

$$\hat{r}_M=\widehat{C}_M^{-1}\hat{b}_M$$

converge to the unique solution of the policy evaluation Eq. (4.23) as $M\to\infty$, and provide the estimates $\Phi\hat{r}_M$ of the cost vector J_μ of μ:

$$\tilde{J}_\mu=\Phi\hat{r}_M.$$

This is the aggregation counterpart of the LSTD(0) method. One may also use an iterative simulation-based LSPE(0)-type method or a TD(0)-type method to solve the equation $Cr=b$; see [Ber12].

Note that instead of using the probabilities ξ_i to sample directly original system states, we may alternatively sample the aggregate states S_ℓ according to some distribution $\{\zeta_\ell \mid \ell=1,\ldots,q\}$, generate a sequence of aggregate states $\{S_{\ell_1},S_{\ell_2},\ldots\}$, and then generate a state sequence $\{i_1,i_2,\ldots\}$ using the disaggregation probabilities. In this case Eq. (4.24) should be modified as follows:

$$\widehat{C}_M=I-\frac{\alpha}{M}\sum_{m=1}^{M}\frac{1}{\zeta_{\ell_m}d_{\ell_m i_m}}d(i_m)\phi(j_m)',\qquad \hat{b}_M=\frac{1}{M}\sum_{m=1}^{M}\frac{1}{\zeta_{\ell_m}d_{\ell_m i_m}}d(i_m)g\big(i_m,\mu(i_m),j_m\big).$$

The main difficulty with the policy improvement step at a given state i is the need to compute the expected value in the Q-factor expression

$$\sum_{j=1}^{n}p_{ij}(u)\left(g(i,u,j)+\alpha\sum_{\ell=1}^{q}\phi_{j\ell}r_\ell^k\right)$$

that is minimized over $u\in U(i)$. If the transition probabilities $p_{ij}(u)$ are available and the number of successor states [the states j such that $p_{ij}(u)>0$] is small, this expected value may be easily calculated (an important case where this is so is when the system is deterministic). Otherwise, one may consider approximating this expected value using one of the model-free schemes described in Section 2.4.

Simulation-Based Value Iteration and Q-Learning

An exact VI algorithm for obtaining r^* is the fixed point iteration

$$r^{k+1}=Hr^k,$$

starting from some initial guess r^0, where H is the contraction mapping of Eq. (4.12). A stochastic approximation-type algorithm based on this fixed point iteration generates a sequence of aggregate states

$\{S_{\ell_0}, S_{\ell_1}, \ldots\}$ by some probabilistic mechanism, which ensures that all aggregate states are generated infinitely often. Given r^k and S_{ℓ_k}, it independently generates an original system state i_k according to the probabilities $d_{\ell i}$, and updates the component r_{ℓ_k} according to

$$r_{\ell_k}^{k+1} = (1-\gamma_k) r_{\ell_k}^k + \gamma_k \min_{u \in U(i)} \sum_{j=1}^{n} p_{i_k j}(u) \left(g(i_k, u, j) + \alpha \sum_{\ell=1}^{q} \phi_{j\ell} r_\ell^k \right),$$

where γ_k is a diminishing positive stepsize, and leaves all the other components unchanged:

$$r_\ell^{k+1} = r_\ell^k, \qquad \text{if } \ell \neq \ell_k.$$

This algorithm can be viewed as an asynchronous stochastic approximation version of VI. The stepsize γ_k should be diminishing (typically at the rate of $1/k$), and its justification and convergence mechanism are very similar to the ones for the Q-learning algorithm. We refer to the paper by Tsitsiklis and VanRoy [TsV96] for further discussion and analysis (see also [BeT96], Section 3.1.2 and 6.7).

A somewhat different algorithm is possible in the case of hard aggregation, assuming that for every ℓ, the set $U(i)$ is the same for all states i in the disaggregation set I_ℓ. Then, as discussed in [BeT96], Section 6.7.7, we can introduce Q-factors that are constant within each set I_ℓ and have the form

$$\tilde{Q}(i,u) = Q(\ell, u), \qquad i \in I_\ell, \ u \in U(i).$$

We then obtain an algorithm that updates the Q-factors $Q(\ell, u)$ one at a time, using a Q-learning-type iteration of the form

$$Q(\ell,u) := (1-\gamma) Q(\ell,u) + \gamma \Big(g(i,u,j) + \alpha \min_{v \in U(j)} Q\big(m(j), v\big) \Big),$$

where i is a state within I_ℓ that is chosen with probability $d_{\ell i}$, j is the outcome of a transition simulated according to the transition probabilities $p_{ij}(u)$, the index $m(j)$ corresponds to the aggregate state $S_{m(j)}$ to which j belongs, and γ is the stepsize. It can be seen that this algorithm coincides with Q-learning with lookup table representation, applied to a lower dimensional aggregate DP problem that involves just the aggregate states. With a suitably decreasing stepsize γ and assuming that each pair (ℓ, u) is simulated an infinite number of times, the standard convergence results for Q-learning [Tsi94] apply.

We note, however, that the Q-learning algorithm just described has a substantial drawback. It solves an aggregate problem that differs from the aggregate problem described in Section 4.1, because implicit in the algorithm is the restriction that the same control is applied at all states i that belong to the same disaggregation set. In effect, we are assigning controls to subsets of states (the disaggregation sets) and not to individual states of the original problem. Clearly this is a coarser form of control, which is inferior in terms of performance. However, the Q-learning algorithm may find some use in the context of initialization of another algorithm that aspires to better performance.

4.3. Feature Formation by Using Scoring Functions

The choice of the feature mapping F and the method to obtain aggregate states are clearly critical for the success of feature-based aggregation. In the subsequent Section 5 we will discuss how deep neural network architectures can be used for this purpose. In what follows in this section we consider some simple forms of feature mappings that can be used when we already have a reasonable estimate of the optimal cost function J^* or the cost function J_μ of some policy μ, which we can use to group together states with similar estimated optimal cost. Then the aggregation approach can provide an improved piecewise constant or piecewise linear cost approximation. We provide some simple illustrative examples of this approach in Section 4.5.

In particular, suppose that we have obtained in some way a real-valued *scoring function* $V(i)$ of the state i, which serves as an index of undesirability of state i as a starting state (smaller values of V are assigned to more desirable states, consistent with the view of V as some form of "cost" function). One possibility is to use as V an approximation of the cost function of some "good" (e.g., near-optimal) policy. Another possibility is to obtain V as the cost function of some reasonable policy applied to an approximation of the original problem (e.g., a related problem that can be solved more easily either analytically or computationally; see [Ber17], Section 6.2). Still another possibility is to obtain V by training a neural network or other architecture using samples of state-cost pairs obtained by using a software or human expert, and some supervised learning technique, such as for example Tesauro's comparison learning scheme [Tes89b], [Tes01]. Finally, one may compute V using some form of policy evaluation algorithm like TD(λ).

Given the scoring function V, we will construct a feature mapping that groups together states i with roughly equal scores $V(i)$. In particular, we let $R_1, \ldots, R_q$ be q disjoint intervals that form a partition of the set of possible values of V [i.e., are such that for any state i, there is a unique interval R_ℓ such that $V(i) \in R_\ell$]. We define a feature vector $F(i)$ of the state i according to

$$F(i) = \ell, \qquad \forall\ i \text{ such that } V(i) \in R_\ell, \quad \ell = 1, \ldots, q. \tag{4.25}$$

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

$$I_\ell = \big\{ i \mid F(i) = \ell \big\} = \big\{ i \mid V(i) \in R_\ell \big\}, \qquad \ell = 1, \ldots, q. \tag{4.26}$$

Assuming that all the sets I_ℓ are nonempty, we thus obtain a hard aggregation scheme, with aggregation probabilities defined by Eq. (4.6); see Fig. 4.5.

A related scoring function scheme may be based on representative states. Here the aggregate states and the disaggregation probabilities are obtained by forming a fairly large sample set of states $\{i_m \mid m = 1, \ldots, M\}$, by computing their corresponding scores

$$\big\{ V(i_m) \mid m = 1, \ldots, M \big\},$$

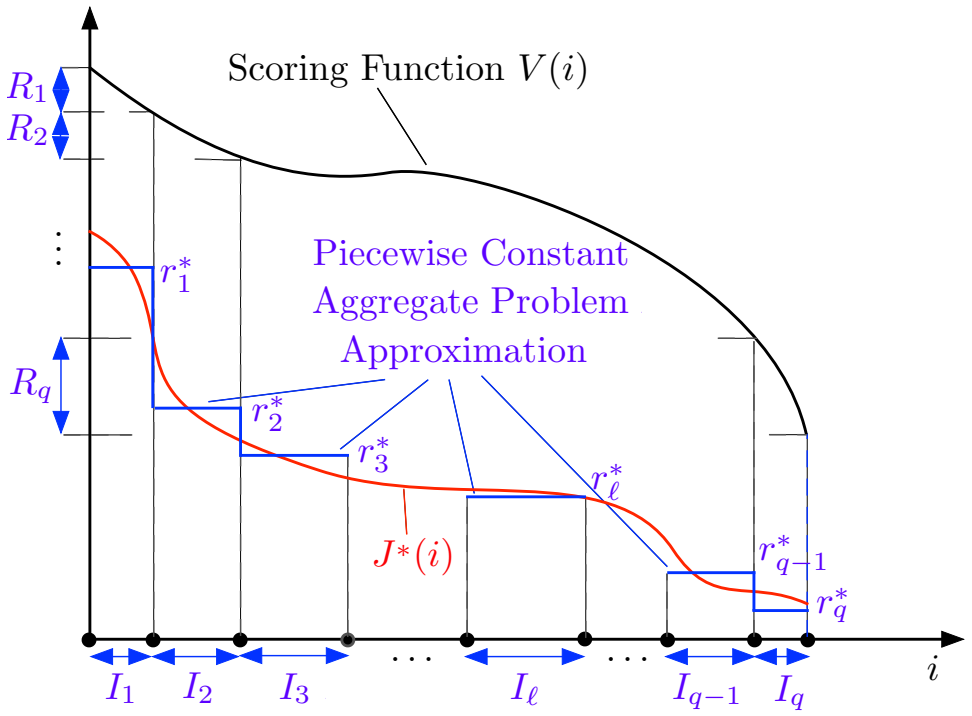


Figure 4.5. Hard aggregation scheme based on a single scoring function. We introduce q disjoint intervals $R_1, \ldots, R_q$ that form a partition of the set of possible values of V, and we define a feature vector $F(i)$ of the state i according to

$$F(i) = \ell, \qquad \forall\, i \text{ such that } V(i) \in R_\ell,\ \ell = 1, \ldots, q.$$

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

$$I_\ell = \big\{ i \mid F(i) = \ell \big\} = \big\{ i \mid V(i) \in R_\ell \big\}, \qquad \ell = 1, \ldots, q.$$

The solution of the aggregate problem provides a piecewise constant approximation of the optimal cost function of the original problem.

and by suitably dividing the range of these scores into disjoint intervals $R_1, \ldots, R_q$ to form the aggregate states, similar to Eqs. (4.25)-(4.26). Simultaneously we obtain subsets of sampled states $\hat{I}_\ell \subset I_\ell$ to which we can assign positive disaggregation probabilities. Figure 4.6 illustrates this idea for the case where each subset $\hat{I}_\ell$ consists of a single (representative) state. This is a form of "discretization" of the original state space based on the score values of the states. As the figure indicates, the role of the scoring function is to assist in forming a set of states that is small (to keep the aggregate DP problem computations manageable) but representative (to provide sufficient detail in the approximation of J^*, i.e., be dense in the parts of the state space where J^* varies a lot, and sparse in other parts).

The following proposition illustrates the important role of the *quantization error*, defined as

$$\delta = \max_{\ell=1,\ldots,q} \max_{i,j \in I_\ell} \big| V(i) - V(j) \big|. \tag{4.27}$$

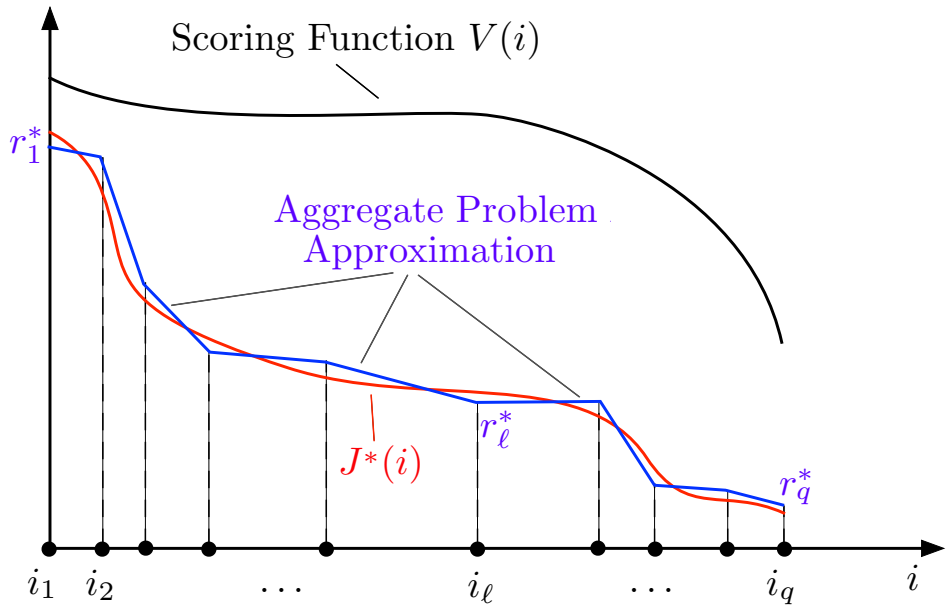


Figure 4.6. Schematic illustration of aggregation based on sampling states and using a scoring function V to form a representative set $i_1, \ldots, i_q$. A piecewise linear approximation of J^* is obtained by using the corresponding aggregate costs $r_1^*, \ldots, r_q^*$ and the aggregation probabilities.

It represents the maximum error that can be incurred by approximating V within each set I_ℓ with a single value from its range within the subset.

Proposition 4.4: Consider the hard aggregation scheme defined by a scoring function V as described above. Assume that the variations of J^* and V over the sets $I_1, \ldots, I_q$ are within a factor $\beta \geq 0$ of each other, i.e., that

$$\big|J^*(i) - J^*(j)\big| \leq \beta \big|V(i) - V(j)\big|, \qquad \forall\, i, j \in I_\ell,\ \ell = 1, \ldots, q.$$

(a) We have

$$\big|J^*(i) - r_\ell^*\big| \leq \frac{\beta\delta}{1-\alpha}, \qquad \forall\, i \in I_\ell,\ \ell = 1, \ldots, q,$$

where δ is the quantization error of Eq. (4.27).

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

$$J^*(i) = r_\ell^*, \qquad \forall\, i \in I_\ell,\ \ell = 1, \ldots, q.$$

Proof: (a) Since we are dealing with a hard aggregation scheme, the result of Prop. 4.1 applies. By our assumptions, the maximum variation of J^* over the disaggregation sets I_ℓ is bounded by $\epsilon = \beta\delta$, and the result of part (a) follows from Prop. 4.1.

(b) This is a special case of part (a) with $\delta = \epsilon = 0$. **Q.E.D.**

Examples of scoring functions that may be useful in various settings are cost functions of nearly optimal policies, or approximations to such cost functions, provided for example by a neural network or other approximation schemes. Another example, arising in the adaptive aggregation scheme proposed by Bertsekas and Castanon [BeC89], is to use as $V(i)$ the residual vector $(TJ)(i) - J(i)$, where J is some approximation to the optimal cost function J^*, or the residual vector $(T_\mu J)(i) - J(i)$, where J is some approximation to the cost function of a policy μ; see also Keller, Mannor, and Precup [KMP06]. Note that it is not essential that V approximates well J^* or J_μ. What is important is that states with similar values of J^* or J_μ also have similar values of V.

Scoring Function Scheme with a State Space Partition

Another useful scheme is based on a scoring function V, which is defined separately on each one of a collection of disjoint subsets $C_1, \ldots, C_m$ that form a partition of the state space. We define a feature vector $F(i)$ that depends not only on the value of $V(i)$ but also on the membership of i in the subsets of the partition. In particular, for each $\theta = 1, \ldots, m$, let $R_{1\theta}, \ldots, R_{q\theta}$ be q disjoint intervals that form a partition of the set of possible values of V over the set C_θ. We then define

$$F(i) = (\ell, \theta), \qquad \forall\ i \in C_\theta \text{ such that } V(i) \in R_{\ell\theta}. \tag{4.28}$$

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

$$I_{\ell\theta} = \big\{ i \mid F(i) = (\ell, \theta) \big\} = \big\{ i \in C_\theta \mid V(i) \in R_{\ell\theta} \big\}, \qquad \ell = 1, \ldots, q,\ \theta = 1, \ldots, m,$$

which represent the disaggregation sets of the resulting hard aggregation scheme. In this scheme the aggregate states depend not only on the values of V but also on the subset C_θ of the partition.

Using Multiple Scoring Functions

The approach of forming features using a single scoring function can be extended to the case where we have a vector of scoring functions $V(i) = \big(V_1(i), \ldots, V_s(i)\big)$. Then we can partition the set of possible values of $V(i)$ into q disjoint subsets $R_1, \ldots, R_q$ of the s-dimensional space $\Re^s$, define a feature vector $F(i)$ according to

$$F(i) = \ell, \qquad \forall\ i \text{ such that } V(i) \in R_\ell,\ \ell = 1, \ldots, q, \tag{4.29}$$

and proceed as in the case of a scalar scoring function, i.e., construct a hard aggregation scheme with disaggregation sets given by

$$I_\ell = \big\{ i \mid F(i) = \ell \big\} = \big\{ i \mid V(i) \in R_\ell \big\}, \qquad \ell = 1, \ldots, q.$$

One possibility to obtain multiple scoring functions is to start with a single fairly simple scoring function, obtain aggregate states as described earlier, solve the corresponding aggregate problem, and use the optimal cost function of that problem as an additional scoring function. This is reminiscent of *feature iteration*, an idea that has been suggested in several approximate DP works.

A related and complementary possibility is to somehow construct multiple policies, evaluate each of these policies (perhaps approximately, using a neural network), and use the policy cost function evaluations as scoring functions. This possibility may be particularly interesting in the case of a deterministic discrete optimization problem. The reason is that the deterministic character of the problem may obviate the need for expensive simulation and neural network training, as we discuss in the next section.

4.4. Using Heuristics to Generate Features - Deterministic Optimization and Rollout

An important context where it is natural to use multiple scoring functions is general deterministic optimization problems with a finite search space. For such problems simple heuristics are often available to obtain suboptimal solutions from various starting conditions, e.g., greedy algorithms of various kinds. The cost of each heuristic can then be used as a scoring function after the problem is converted to a finite horizon DP problem. The formulation that we will use in this section is very general and for this reason the number of states of the DP problem may be very large. Alternative DP reformulations with fewer states may be obtained by exploiting the structure of the problem. For example shortest path-type problems and discrete-time finite-state deterministic optimal control control problems can be naturally posed as DP problems with a simpler and more economical formulation than the one given here. In such cases the methodology to be described can be suitably adapted to exploit the problem-specific structural characteristics.

The general discrete optimization problem that we consider in this section is

$$\begin{aligned} &\text{minimize} \quad G(u) \\ &\text{subject to} \quad u \in U, \end{aligned} \tag{4.30}$$

where U is a finite set of feasible solutions and $G(u)$ is a cost function. We assume that each solution u has N components; i.e., it has the form $u = (u_1, \ldots, u_N)$, where N is a positive integer. We can then view the problem as a sequential decision problem, where the components $u_1, \ldots, u_N$ are selected one-at-a-time. An m-tuple $(u_1, \ldots, u_m)$ consisting of the first m components of a solution is called an *m-solution*. We associate

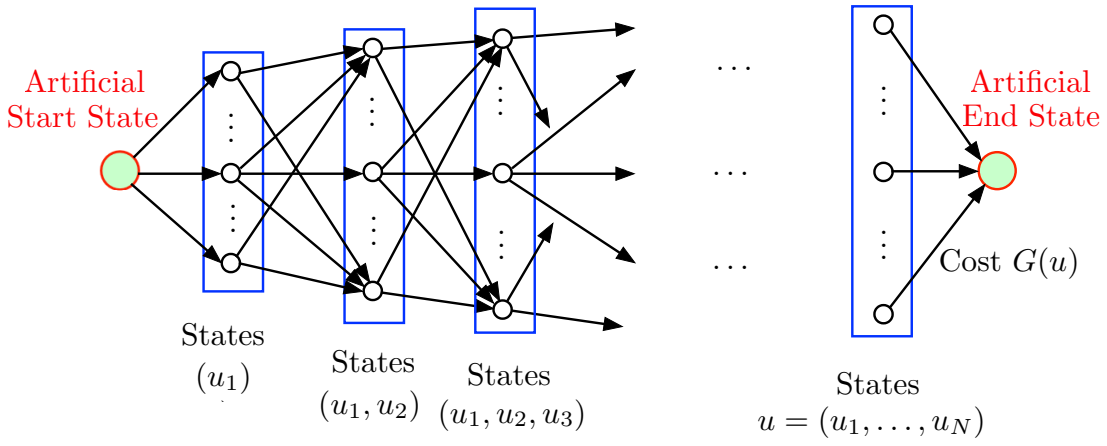


Figure 4.7. Formulation of a discrete optimization problem as a DP problem. There is a cost $G(u)$ only at the terminal stage on the arc connecting an N-solution $u = (u_1, \ldots, u_N)$ to the artificial terminal state. Alternative formulations may use fewer states by taking advantage of the problem's structure.

m-solutions with the mth stage of a finite horizon DP problem.† In particular, for $m = 1, \ldots, N$, the states of the mth stage are of the form $(u_1, \ldots, u_m)$. The initial state is a dummy (artificial) state. From this state we may move to any state (u_1), with u_1 belonging to the set

$$U_1 = \big\{\tilde{u}_1 \mid \text{there exists a solution of the form } (\tilde{u}_1, \tilde{u}_2, \ldots, \tilde{u}_N) \in U\big\}.$$

Thus U_1 is the set of choices of u_1 that are consistent with feasibility.

More generally, from a state $(u_1, \ldots, u_m)$, we may move to any state of the form $(u_1, \ldots, u_m, u_{m+1})$, with u_{m+1} belonging to the set

$$U_{m+1}(u_1, \ldots, u_m) = \big\{\tilde{u}_{m+1} \mid \text{ there exists a solution of the form } (u_1, \ldots, u_m, \tilde{u}_{m+1}, \ldots, \tilde{u}_N) \in U\big\}. \quad (4.31)$$

The choices available at state $(u_1, \ldots, u_m)$ are $u_{m+1} \in U_{m+1}(u_1, \ldots, u_m)$. These are the choices of u_{m+1} that are consistent with the preceding choices $u_1, \ldots, u_m$, and are also consistent with feasibility. The terminal states correspond to the N-solutions $u = (u_1, \ldots, u_N)$, and the only nonzero cost is the terminal cost $G(u)$. This terminal cost is incurred upon transition from u to an artificial termination state; see Fig. 4.7.

Let $J^*(u_1, \ldots, u_m)$ denote the optimal cost starting from the m-solution $(u_1, \ldots, u_m)$, i.e., the optimal cost of the problem over solutions whose first m components are constrained to be equal to u_i, $i = 1, \ldots, m$, respectively. If we knew the optimal cost-to-go functions $J^*(u_1, \ldots, u_m)$, we could construct an optimal solution by a sequence of N single component minimizations. In particular, an optimal solution $(u_1^*, \ldots, u_N^*)$

† Our aggregation framework of Section 4.1 extends in a straightforward manner to finite-state finite-horizon problems. The main difference is that optimal cost functions, feature vectors, and scoring functions are not only state-dependent but also stage-dependent. In effect the states are the m-solutions for all values of m.

could be obtained sequentially, starting with u_1^* and proceeding forward to u_N^*, through the algorithm

$$u_{m+1}^* \in \arg \min_{u_{m+1} \in U_{m+1}(u_1^*,\ldots,u_m^*)} J^*(u_1^*,\ldots,u_m^*,u_{m+1}), \qquad m = 0,\ldots,N-1.$$

Unfortunately, this is seldom viable, because of the prohibitive computation required to obtain the functions $J^*(u_1,\ldots,u_m)$.

Suppose that we have s different heuristic algorithms, which we can apply for suboptimal solution. We assume that each of these algorithms can start from any m-solution $(u_1,\ldots,u_m)$ and produce an N-solution $(u_1,\ldots,u_m,u_{m+1},\ldots,u_N)$. The costs thus generated by the s heuristic algorithms are denoted by $V_1(u_1,\ldots,u_m),\ldots,V_s(u_1,\ldots,u_m)$, respectively, and the corresponding vector of heuristic costs is denoted by

$$V(u_1,\ldots,u_m) = \big(V_1(u_1,\ldots,u_m),\ldots,V_s(u_1,\ldots,u_m)\big).$$

Note that the heuristic algorithms can be quite sophisticated, and at a given partial solution $(u_1,\ldots,u_m)$, may involve multiple component choices from $(u_{m+1},\ldots,u_N)$ and/or suboptimizations that may depend on the previous choices $u_1,\ldots,u_m$ in complicated ways. In fact, the heuristic algorithms may require some preliminary experimentation and training, using for example, among others, neural networks.

The main idea now is to use the heuristic cost functions as scoring functions to construct a feature-based hard aggregation framework.† In particular, for each $m = 1,\ldots,N-1$, we partition the set of possible values of $V(u_1,\ldots,u_m)$ into q disjoint subsets $R_1^m,\ldots,R_q^m$, we define a feature vector $F(u_1,\ldots,u_m)$ according to

$$F(u_1,\ldots,u_m) = \ell, \qquad \forall\ (u_1,\ldots,u_m) \text{ such that } V(u_1,\ldots,u_m) \in R_\ell^m,\ \ell = 1,\ldots,q, \tag{4.32}$$

and we construct a hard aggregation scheme with disaggregation sets for each $m = 1,\ldots,N-1$, given by

$$I_\ell^m = \big\{(u_1,\ldots,u_m) \mid V(u_1,\ldots,u_m) \in R_\ell^m\big\}, \qquad \ell = 1,\ldots,q.$$

Note that the number of aggregate states is roughly similar for each of the $N-1$ stages. By contrast the number of states of the original problem may increase very fast (exponentially) as N increases; cf. Fig. 4.7.

The aggregation scheme is illustrated in Fig. 4.8. It involves $N-1$ successive transitions between m-solutions to $(m+1)$-solutions ($m = 1,\ldots,N-1$), interleaved with transitions to and from the corresponding aggregate states. The aggregate problem is completely defined once the aggregate states and the disaggregation probabilities have been chosen. The transition mechanism of stage m involves the following steps.

† There are several variants of this scheme, involving for example a state space partition as in Section 4.3. Moreover, the method of partitioning the decision vector u into its components $u_1,\ldots,u_N$ may be critically important in specific applications.

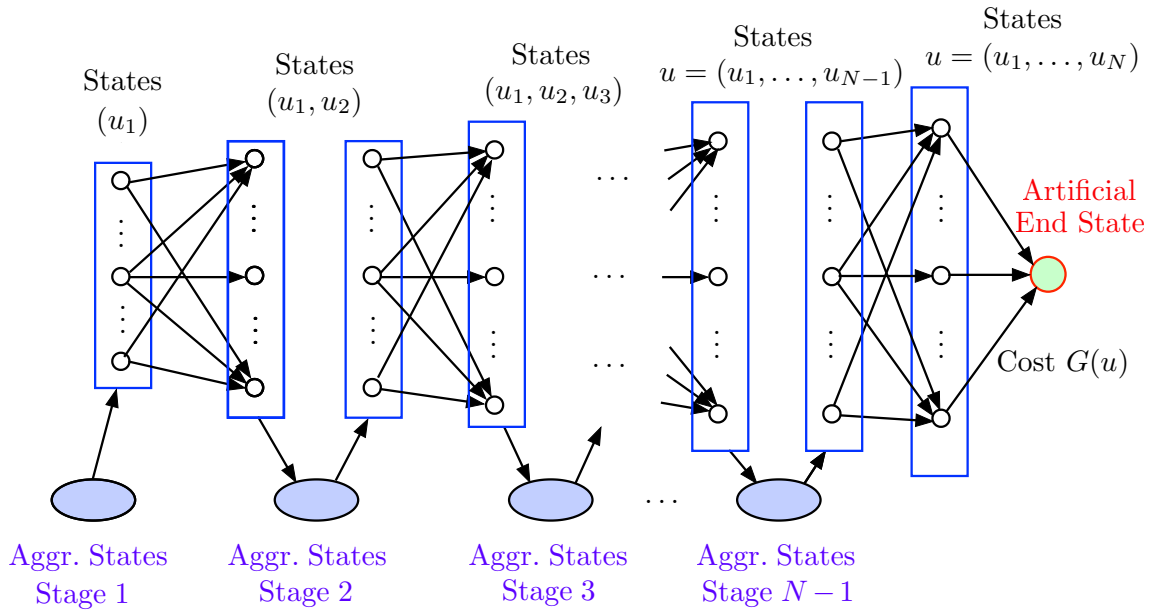


Figure 4.8. Schematic illustration of the heuristics-based aggregation scheme for discrete optimization. The aggregate states are defined by the scoring functions/heuristics, and the optimal aggregate costs are obtained by DP stating from the last stage and proceeding backwards.

(1) From an aggregate state ℓ at stage m, we generate some state $(u_1, \ldots, u_m) \in I_\ell^m$ according to the disaggregation probabilities.

(2) We transition to the next state $(u_1, \ldots, u_m, u_{m+1})$ by selecting the control u_{m+1}.

(3) We run the s heuristics from the $(m+1)$-solution $(u_1, \ldots, u_m, u_{m+1})$ to determine the next aggregate state, which is the index of the set of the partition of stage $m+1$ to which the vector

$$V(u_1, \ldots, u_m, u_{m+1}) = \big(V_1(u_1, \ldots, u_m, u_{m+1}), \ldots, V_s(u_1, \ldots, u_m, u_{m+1})\big)$$

belongs.

A key issue is the selection of the disaggregation probabilities for each stage. This requires, for each value of m, the construction of a suitable sample of m-solutions, where the disaggregation sets I_ℓ^m are adequately represented.

The solution of the aggregate problem by DP starts at the last stage to compute the corresponding aggregate costs $r^*_{\ell(N-1)}$ for each of the aggregate states ℓ, using $G(u)$ as terminal cost function. Then it proceeds with the next-to-last stage to compute the corresponding aggregate costs $r^*_{\ell(N-2)}$, using the previously computed aggregate costs $r^*_{\ell(N-1)}$, etc.

The optimal cost function $J^*(u_1, \ldots, u_m)$ for stage m is approximated by a piecewise constant function,

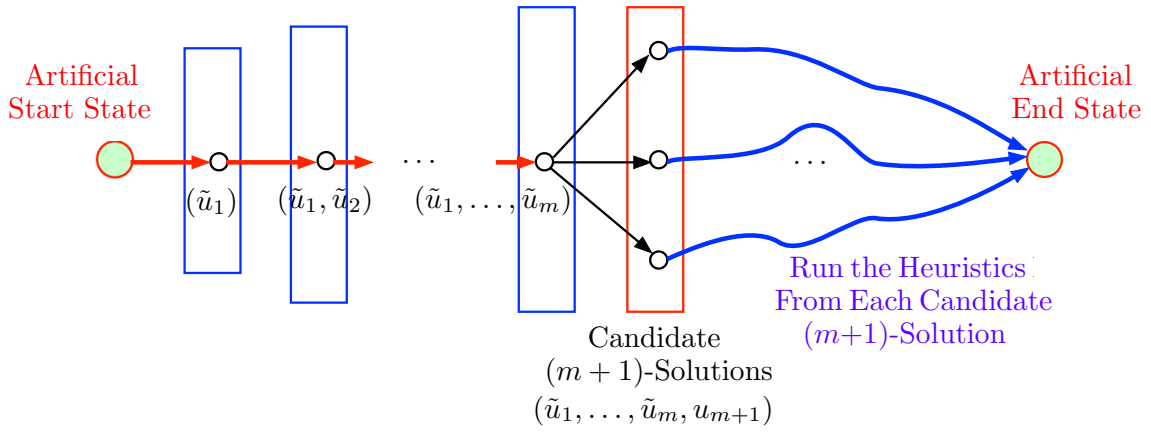


Figure 4.9. Sequential construction of a suboptimal N-solution $(\tilde{u}_1, \ldots, \tilde{u}_N)$ for the original problem, after the aggregate problem has been solved. Given the m-solution $(\tilde{u}_1, \ldots, \tilde{u}_m)$, we run the s heuristics from each of the candidate $(m+1)$-solution $(\tilde{u}_1, \ldots, \tilde{u}_m, u_{m+1})$, and compute the aggregate state and aggregate cost of this candidate $(m+1)$-solution. We then select as $\tilde{u}_{m+1}$ the one that corresponds to the candidate $(m+1)$-solution with minimal aggregate cost.

which is derived by solving the aggregate problem. This is the function

$$\tilde{J}(u_1, \ldots, u_m) = r^*_{\ell m}, \qquad \forall\ (u_1, \ldots, u_m) \text{ with } V(u_1, \ldots, u_m) \in R^m_\ell, \tag{4.33}$$

where $r^*_{\ell m}$ is the optimal cost of aggregate state ℓ at stage m of the aggregate problem.

Once the aggregate problem has been solved for the costs $r^*_{\ell m}$, a suboptimal N-solution $(\tilde{u}_1, \ldots, \tilde{u}_N)$ for the original problem is obtained sequentially, starting from stage 1 and proceeding to stage N, through the minimizations

$$\tilde{u}_1 \in \arg\min_{u_1} \tilde{J}(u_1), \tag{4.34}$$

$$\tilde{u}_{m+1} \in \arg \min_{u_{m+1} \in U_{m+1}(\tilde{u}_1, \ldots, \tilde{u}_m)} \tilde{J}(\tilde{u}_1, \ldots, \tilde{u}_m, u_{m+1}), \quad m = 1, \ldots, N-1. \tag{4.35}$$

Note that to evaluate each of the costs $\tilde{J}(\tilde{u}_1, \ldots, \tilde{u}_m, u_{m+1})$ needed for this minimization, we need to do the following (see Fig. 4.9):

(1) Run the s heuristics from the $(m+1)$-solution $(\tilde{u}_1, \ldots, \tilde{u}_m, u_{m+1})$ to evaluate the scoring vector of heuristic costs

$$V(\tilde{u}_1, \ldots, \tilde{u}_m, u_{m+1}) = \big(V_1(\tilde{u}_1, \ldots, \tilde{u}_m, u_{m+1}), \ldots, V_s(\tilde{u}_1, \ldots, \tilde{u}_m, u_{m+1})\big).$$

(2) Set $\tilde{J}(\tilde{u}_1, \ldots, \tilde{u}_m, u_{m+1})$ to the aggregate cost $r^*_{\ell(m+1)}$ of the aggregate state $S_{\ell(m+1)}$ corresponding to this scoring vector, i.e., to the set $R^{(m+1)}_\ell$ such that

$$V(\tilde{u}_1, \ldots, \tilde{u}_m, u_{m+1}) \in R^{(m+1)}_\ell.$$

Once $\tilde{J}(\tilde{u}_1, \ldots, \tilde{u}_m, u_{m+1})$ has been computed for all $u_{m+1} \in U_{m+1}(\tilde{u}_1, \ldots, \tilde{u}_m)$, we select $\tilde{u}_{m+1}$ via the minimization (4.35), and repeat starting from the $(m+1)$-solution $(\tilde{u}_1, \ldots, \tilde{u}_m, \tilde{u}_{m+1})$. Note that even if there is only one heuristic, $\tilde{u}_{m+1}$ minimizes the aggregate cost $r^*_{\ell(m+1)}$, which is not the same as the cost corresponding to the heuristic.

We finally mention a simple improvement of the scheme just described for constructing an N-solution. In the course of the algorithm many other N-solutions are obtained, during the training and final solution selection processes. It is possible that some of these solutions are actually better [have lower cost $G(u)$] than the final N-solution $(\tilde{u}_1, \ldots, \tilde{u}_N)$ that is constructed by using the aggregate problem formulation. This can happen because the aggregation scheme is subject to quantization error. Thus it makes sense to maintain the best of the N-solutions generated in the course of the algorithm, and compare it at the end with the N-solution obtained through the aggregation scheme. This is similar to the so-called "fortified" version of the rollout algorithm (see [BTW97] or [Ber17]).

Relation to the Rollout Algorithm

The idea of using one or more heuristic algorithms as a starting point for generating an improved solution of a discrete optimization problem is shared by other suboptimal control approaches. A prime example is the rollout algorithm, which in some contexts can be viewed as a single policy iteration; see [BTW97] for an analysis of rollout for discrete optimization problems, and the textbook [Ber17] for an extensive discussion and many references to applications, including the important model predictive control methodology for control system design.

Basically the rollout algorithm uses the scheme of Fig. 4.9 to construct a suboptimal solution $(\tilde{u}_1, \ldots, \tilde{u}_N)$ in N steps, one component at a time, but adds a new decision $\tilde{u}_{m+1}$ to the current m-solution $(\tilde{u}_1, \ldots, \tilde{u}_m)$ in a simpler way. It runs the s heuristics from each candidate $(m+1)$-solution $(\tilde{u}_1, \ldots, \tilde{u}_m, u_{m+1})$ and computes the corresponding heuristic costs

$$V_1(\tilde{u}_1, \ldots, \tilde{u}_m, u_{m+1}), \ldots, V_s(\tilde{u}_1, \ldots, \tilde{u}_m, u_{m+1}).$$

It then selects as the next decision $\tilde{u}_{m+1}$ the one that minimizes over $u_{m+1} \in U_{m+1}(\tilde{u}_1, \ldots, \tilde{u}_m)$ the best heuristic cost

$$\hat{V}(\tilde{u}_1, \ldots, \tilde{u}_m, u_{m+1}) = \min\big\{V_1(\tilde{u}_1, \ldots, \tilde{u}_m, u_{m+1}), \ldots, V_s(\tilde{u}_1, \ldots, \tilde{u}_m, u_{m+1})\big\},$$

i.e., it uses $\hat{V}$ in place of $\tilde{J}$ in Eqs. (4.34)-(4.35). In practice, the rollout algorithm's heuristics may involve sophisticated suboptimizations that may make sense in the context of the problem at hand.

Note that the construction of the final N-solution is similar and equally complicated in the rollout and the scoring vector-based aggregation approach. However, the aggregation approach requires an extra layer

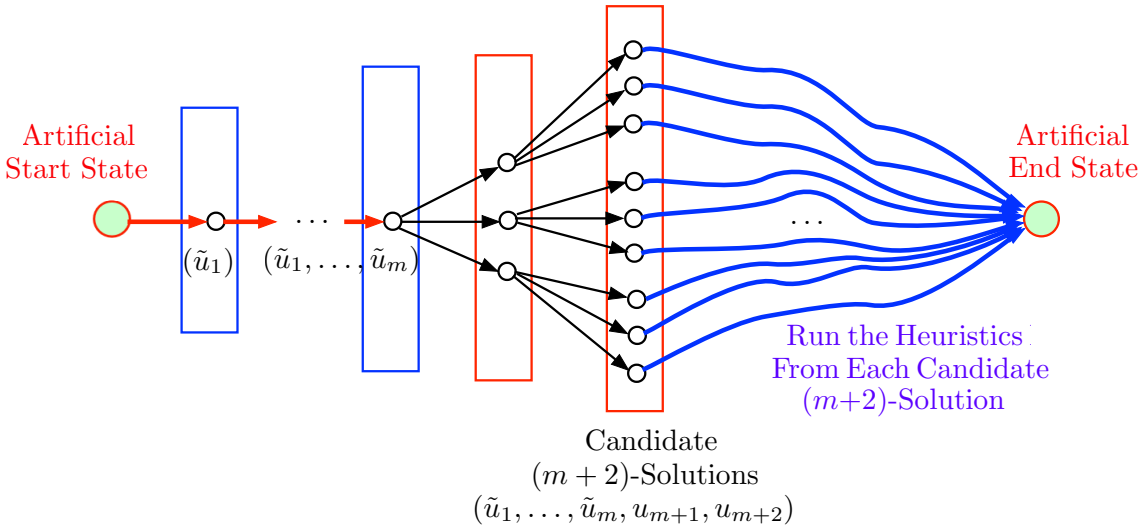


Figure 4.10. Sequential construction of a suboptimal N-solution $(\tilde{u}_1, \ldots, \tilde{u}_N)$ by using two-step lookahead, after the aggregate problem has been solved. Given the m-solution $(\tilde{u}_1, \ldots, \tilde{u}_m)$, we run the s heuristics from all the candidate $(m+2)$-solutions $(\tilde{u}_1, \ldots, \tilde{u}_m, u_{m+1}, u_{m+2})$, and select as $\tilde{u}_{m+1}$ the first component of the two-step sequence that corresponds to minimal aggregate cost.

of computation *prior to constructing the N-solution*, namely the solution of an aggregate problem. This may be a formidable problem, because it is stochastic (due to the use of disaggregation probabilities) and must be solved exactly (at least in principle). Still, the number of states of the aggregate problem may be quite reasonable, and its solution is well suited for parallel computation.

On the other hand, setting aside the issue of computational solution of the aggregate problem, the heuristics-based aggregation algorithm has the potential of being far superior to the rollout algorithm, for the same reason that approximate policy improvement based on aggregation can be far superior to policy improvement based on one-step lookahead. In particular, with sufficiently large number of aggregate states to eliminate the effects of the quantization error, feature-based aggregation will find an optimal solution, regardless of the quality of the heuristics used. By contrast, policy iteration and rollout can only aspire to produce a solution that is better than the one produced by the heuristics.

Using Multistep Lookahead and Monte Carlo Tree Search

Once the aggregate problem that is based on multiple scoring functions has been solved, the final N-solution can be constructed in more sophisticated ways than the one described in Fig. 4.9. It can be seen that the scheme of Eqs. (4.34)-(4.35) and Fig. 4.9 is based on one-step lookahead. It is possible instead to use multistep lookahead or randomized versions such as Monte Carlo tree search.

As an example, in a two-step lookahead scheme, we again obtain a suboptimal solution $(\tilde{u}_1, \ldots, \tilde{u}_N)$ for the original problem in N stages, starting from stage 1 and proceeding to stage N. At stage 1, we carry

out the two-step minimization

$$(\tilde{u}_1, \tilde{u}_2) \in \arg\min_{u_1,u_2} \tilde{J}(u_1, u_2), \tag{4.36}$$

and fix the first component $\tilde{u}_1$ of the result, cf. Fig. 4.10. We then proceed sequentially: for $m = 1, \ldots, N-2$, given the current m-solution $(\tilde{u}_1, \ldots, \tilde{u}_m)$, we carry out the two-step minimization

$$(\tilde{u}_{m+1}, \tilde{u}_{m+2}) \in \arg\min_{u_{m+1},u_{m+2}} \tilde{J}(\tilde{u}_1, \ldots, \tilde{u}_m, u_{m+1}, u_{m+2}), \quad m = 1, \ldots, N-2, \tag{4.37}$$

and fix the first component $\tilde{u}_{m+1}$ of the result, cf. Fig. 4.10. At the final stage, given the $(N-1)$-solution $(\tilde{u}_1, \ldots, \tilde{u}_{N-1})$, we carry out the one-step minimization

$$\tilde{u}_N \in \arg\min_{u_N} \tilde{J}(\tilde{u}_1, \ldots, \tilde{u}_{N-1}, u_N), \tag{4.38}$$

and obtain the final N-solution $(\tilde{u}_1, \ldots, \tilde{u}_N)$.

Multistep lookahead generates a tree of fixed depth that is rooted at the last node $\tilde{u}_m$ of the current m-solution, and then runs the heuristics from each of the leaf nodes of the tree. We can instead select only a subset of these leaf nodes from which to run the heuristics, thereby economizing on computation. The selection may be based on some heuristic criterion. Monte Carlo tree search similarly uses multistep lookahead but selects only a random sample of leaf nodes to search based on some criterion.

In a more general version of Monte Carlo tree search, instead of a single partial solution, we maintain multiple partial solutions, possibly of varying length. At each step, a one-step or multistep lookahead tree is generated from the most "promising" of the current partial solutions, selected by using a randomization mechanism. The heuristics are run from the leafs of the lookahead trees similar to Fig. 4.10. Then some of the current partial solutions are expanded with an additional component based on the results produced by the heuristics. This type of Monte Carlo tree search has been suggested for use in conjunction with rollout (see the paper [RSS12]), and it can be similarly used with feature-based aggregation.

4.5. Stochastic Shortest Path Problems - Illustrative Examples

Our aggregation framework extends straightforwardly to stochastic shortest path (SSP for short) problems, where there is no discounting and in addition to the states $1, \ldots, n$, there is an additional cost-free and absorbing termination state, denoted 0 (the text references given earlier discuss in detail such problems). The principal change needed is to account for the termination state by introducing an additional aggregate state with corresponding disaggregation set $\{0\}$. As before there are also other aggregate states $S_1, \ldots, S_q$ whose disaggregation sets $I_1, \ldots, I_q$ are subsets of $\{1, \ldots, n\}$. With this special handling of the termination state, the aggregate problem becomes a standard SSP problem whose termination state is the aggregate state corresponding to 0. The Bellman equation of the aggregate problem is given by

$$r_\ell = \sum_{i=1}^{n} d_{\ell i} \min_{u \in U(i)} \sum_{j=1}^{n} p_{ij}(u) \left(g(i,u,j) + \sum_{m=1}^{q} \phi_{jm}\, r_m \right), \qquad \ell = 1, \ldots, q, \tag{4.39}$$

[cf. Eq. (4.12)]. It has a unique solution under some well-known conditions that date to the paper by Bertsekas and Tsitsiklis [BeT91] (there exists at least one proper policy, i.e., a stationary policy that guarantees eventual termination from each initial state with probability 1; moreover all stationary policies that are not proper have infinite cost starting from some initial state). In particular, these conditions are satisfied if all stationary policies are proper.

We will now provide two simple illustrative SSP examples, which were presented in the author's paper [Ber95] as instances of poor performance of TD(λ) and other methods that are based on projected equations and temporal differences (see also the book [BeT96], Section 6.3.2). In these examples the cost function of a policy will be approximated by using feature-based aggregation and a scoring function obtained using either the TD(1) or the TD(0) algorithms. The approximate cost function computed by aggregation will be compared with the results of TD(1) and TD(0). We will show that aggregation provides a much better approximation, suggesting better policy improvement results in a PI context.

Our examples involve a problem with a single policy μ where the corresponding Markov chain is deterministic with n states plus a termination state 0. Under μ, when at state $i = 1, \ldots, n$, we move to state $i-1$ at a cost g_i. Thus starting at state i we traverse each of the states $i-1, \ldots, 1$ and terminate at state 0 at costs $g_i, g_{i-1}, \ldots, g_1$, respectively, while accumulating the total cost

$$J_\mu(i) = g_i + \cdots + g_1, \qquad i = 1, \ldots, n,$$

with $J_\mu(0) = 0$. We consider a linear approximation to this cost function, which we denote by V:

$$V(i) = ri, \qquad i = 1, \ldots, n,$$

where r is a scalar parameter. This parameter may be obtained by using any of the simulation-based methods that are available for training linear architectures, including TD(λ). In the subsequent discussion we will assume that TD(λ) is applied in an idealized form where the simulation samples contain no noise.

The TD(1) algorithm is based on minimizing the sum of the squares of the differences between J_μ and V over all states, yielding the approximation

$$\hat{V}_1(i) = \hat{r}_1 i, \qquad i = 0, 1, \ldots, n,$$

where

$$\hat{r}_1 \in \arg\min_{r \in \Re} \sum_{i=1}^{n} \big(J_\mu(i) - ri\big)^2. \tag{4.40}$$

Here, consistent with our idealized setting of noise-free simulation, we assume that $J_\mu(i)$ is computed exactly for all i. The TD(0) algorithm is based on minimizing the sum of the squares of the errors in satisfying the Bellman equation $V(i) = g_i + V(i-1)$ (or temporal differences) over all states, yielding the approximation

$$\hat{V}_0(i) = \hat{r}_0 i, \qquad i = 0, 1, \ldots, n,$$

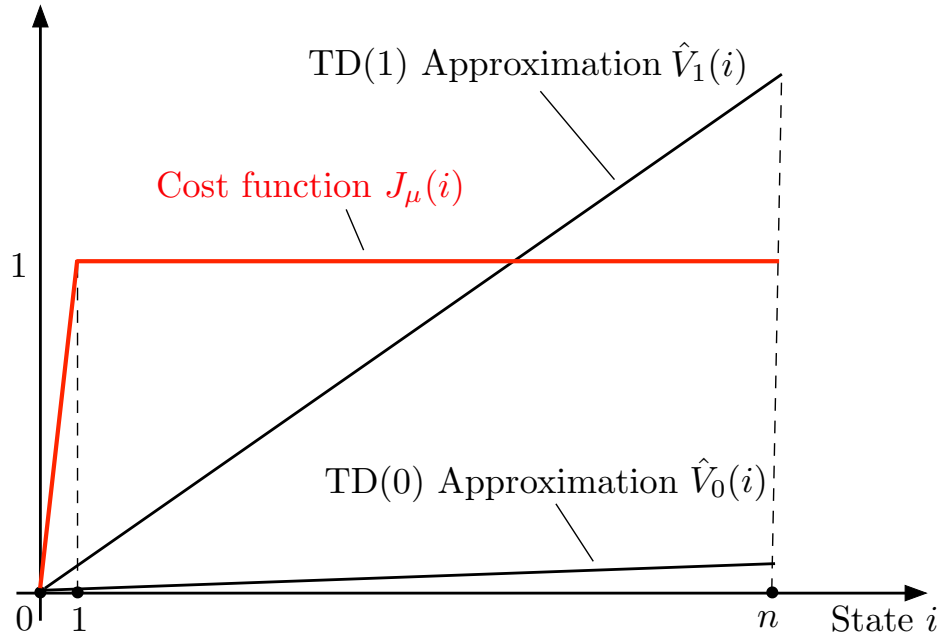


Figure 4.11. Form of $J_\mu(i)$ and the linear approximations $\hat{V}_1(i)$ and $\hat{V}_0(i)$ for case (a): $g_1 = 1$, and $g_i = 0$ for all $i = 2, \ldots, n$.

where

$$\hat{r}_0 = \in \arg\min_{r \in \Re} \sum_{i=1}^{n} \big(g_i + r(i-1) - ri\big)^2. \tag{4.41}$$

Again, we assume that the temporal differences $\big(g_i + r(i-1) - ri\big)$ are computed exactly for all i.

The straightforward solution of the minimization problems in Eqs. (4.40) and (4.41) yields

$$\hat{r}_1 = \frac{n(g_1 + \cdots + g_n) + (n-1)(g_1 + \cdots + g_{n-1}) + \cdots + g_1}{n^2 + (n-1)^2 + \cdots + 1},$$

and

$$\hat{r}_0 = \frac{ng_n + (n-1)g_{n-1} + \cdots + g_1}{n + (n-1) + \cdots + 1}.$$

Consider now two different choices of the one-stage costs g_i:

(a) $g_1 = 1$, and $g_i = 0$ for all $i \neq 1$.

(b) $g_n = -(n-1)$, and $g_i = 1$ for all $i \neq n$.

Figures 4.11 and 4.12 provide plots of $J_\mu(i)$, and the approximations $\hat{V}_1(i)$ and $\hat{V}_0(i)$ for these two cases (these plots come from [Ber95] where the number of states used was $n = 50$). It can be seen that $\hat{V}_1(i)$ and particularly $\hat{V}_0(i)$ are poor approximations of $J_\mu(i)$, suggesting that if used for policy improvement, they may yield a poor successor policy.

We will now consider a hard aggregation scheme based on using $\hat{V}_1$ and $\hat{V}_0$ as scoring functions. The aggregate states of such a scheme in effect consist of disaggregation subsets $I_1, \ldots, I_q$ with $\cup_{\ell=1}^{q} I_\ell = \{1, \ldots, n\}$

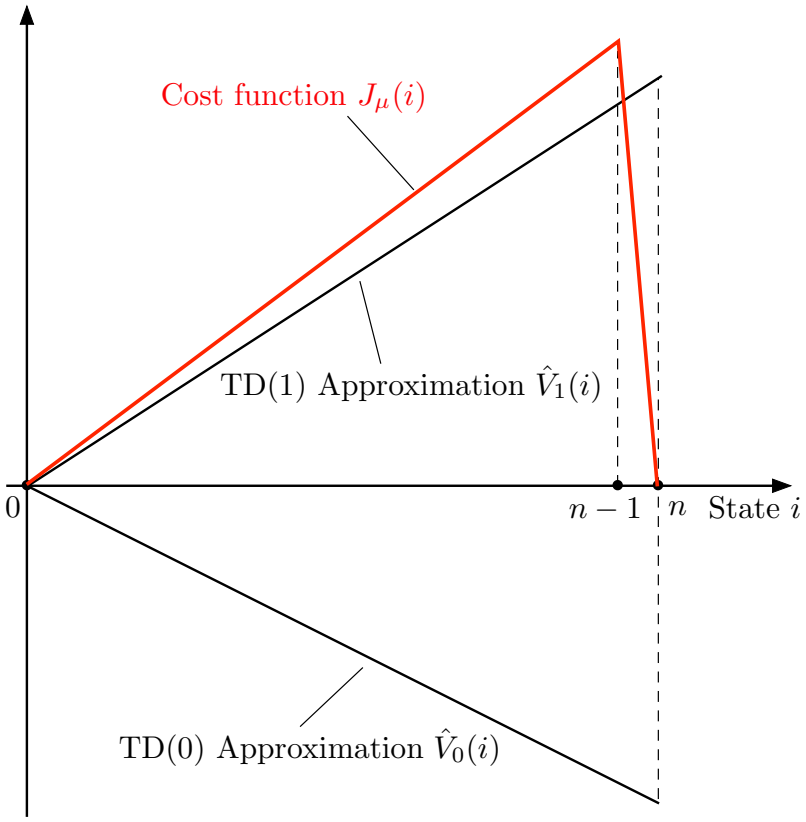


Figure 4.12. Form of $J_\mu(i)$ and the linear approximations $\hat{V}_1(i)$ and $\hat{V}_0(i)$ for case (b): $g_n = -(1-n)$, and $g_i = 1$ for all $i = 1, \ldots, n-1$.

plus the subset $\{0\}$ that serves as the termination state of the aggregate problem. With either $\hat{V}_1$ or $\hat{V}_0$ as the scoring function, the subsets $I_1, \ldots, I_q$ consist of contiguous states. In order to guarantee that the termination state is eventually reached in the aggregate problem, we assume that the disaggregation probability of the smallest state within each of the subsets $I_1, \ldots, I_q$ is strictly positive; this is a mild restriction, which is naturally satisfied in typical schemes that assign equal probability to all the states in a disaggregation set.

Consider first case (a) (cf. Fig. 4.11). Then, because the policy cost function J_μ is constant within each of the subsets $I_1, \ldots, I_q$, the scalar ϵ in Prop. 4.1 is equal to 0, implying that the hard aggregation scheme yields the optimal cost function, i.e., $r^*_\ell = J_\mu(i)$ for all $i \in I_\ell$. To summarize, in case (a) the TD(0) approach yields a very poor linear cost function approximation, the TD(1) approach yields a poor linear cost function approximation, but the aggregation scheme yields the nonlinear policy cost function J_μ exactly.

Consider next case (b) (cf. Fig. 4.12). Then, the hard aggregation scheme yields a piecewise constant approximation to the optimal cost function. The quality of the approximation is degraded by quantization effects. In particular, as the variations of J_μ, and $\hat{V}_1$ or $\hat{V}_0$ increase over the disaggregation sets $I_1, \ldots, I_q$, the quality of the approximation deteriorates, as predicted by Prop. 4.4. Similarly, as the number of states in the disaggregation sets $I_1, \ldots, I_q$ is reduced, the quality of the approximation improves, as illustrated in Fig.

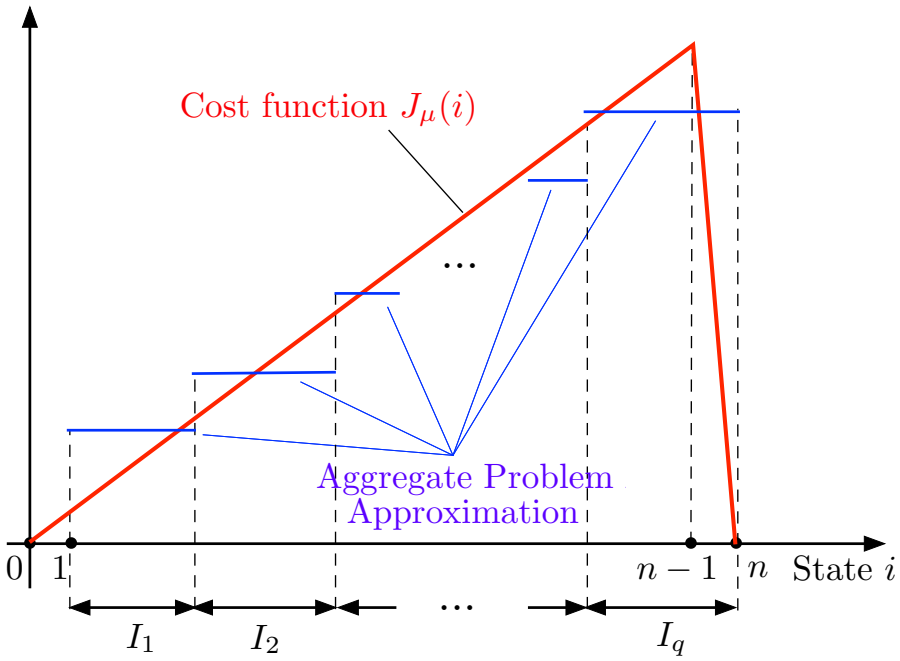


Figure 4.13. Schematic illustration of the piecewise constant approximation of J_μ that is provided by hard aggregation based on the scoring functions $\hat{V}_1$ and $\hat{V}_0$ in case (b).

4.13. In the extreme case where there is only one state in each of the disaggregation sets, the aggregation scheme yields exactly J_μ.

To summarize, in case (b) the TD(0) approach yields a very poor linear cost function approximation, the TD(1) approach yields a reasonably good linear cost function approximation, while the aggregation scheme yields a piecewise constant approximation whose quality depends on the coarseness of the quantization that is implicit in the selection of the number q of disaggregation subsets. The example of case (b) also illustrates how the quality of the scoring function affects the quality of the approximation provided by the aggregation scheme. Here both $\hat{V}_1$ and $\hat{V}_0$ work well as scoring functions, despite their very different form, because states with similar values of J_μ also have similar values of $\hat{V}_1$ as well as $\hat{V}_0$ (cf. Prop. 4.4).

4.6. Multistep Aggregation

The aggregation methodology discussed so far is based on the aggregate problem Markov chain of Fig. 4.2, which returns to an aggregate state after a single transition of the original chain. We may obtain alternative aggregation frameworks by considering a different Markov chain, which starting from an aggregate state, involves multiple original system state transitions before return to an aggregate state. We discuss two possibilities:

(a) *k-Step Aggregation*: Here we require a fixed number k of transitions between original system states before returning to an aggregate state.

(b) λ-*Aggregation*: Here the number k of transitions prior to returning to an aggregate state is controlled by some randomization mechanism. In the case where k is geometrically distributed with parameter $\lambda \in (0,1)$, this method involves multistep mappings that arise in temporal difference contexts and facilitate the use of temporal differences methodology.

Another related possibility, which we do not discuss in this paper, is to introduce *temporal abstractions* (faster/multistep "macro-actions" and transitions between selected states with suitably computed transition costs) into the upper (original system) portion of the aggregate problem Markov chain of Fig. 4.2. There have been many proposals of this type in the reinforcement learning literature, under various names; for some representative works, see Hauskrecht et al. [HMK98], Sutton, Precup, and Singh [SPS99], Parr and Russell [PaR98], Dietterich [Die00], Konidaris and Barto [KoB09], Ciosek and Silver [CiS15], Mann, Mannor, and Precup [MMP15], Serban et al. [SSP18], and the references cited there. It is likely that some of these proposals can be fruitfully adapted to our feature-based aggregation context, and this is an interesting subject for further research.

k-Step Aggregation

This scheme, suggested in [Ber11a] and illustrated in Fig. 4.14, is specified by disaggregation and aggregation probabilities as before, but involves $k > 1$ transitions between original system states in between transitions from and to aggregate states. The aggregate DP problem for this scheme involves $k+1$ copies of the original state space, in addition to the aggregate states. We accordingly introduce vectors $\tilde{J}_0, \tilde{J}_1, \ldots, \tilde{J}_k$, and $r^* = \{r_1^*, \ldots, r_q^*\}$ where:

r_ℓ^* is the optimal cost-to-go from aggregate state S_ℓ.

$\tilde{J}_0(i)$ is the optimal cost-to-go from original system state i that has just been generated from an aggregate state (left side of Fig. 4.14).

$\tilde{J}_1(j_1)$ is the optimal cost-to-go from original system state j_1 that has just been generated from an original system state i.

$\tilde{J}_m(j_m)$, $m = 2, \ldots, k$, is the optimal cost-to-go from original system state j_m that has just been generated from an original system state j_{m-1}.

These vectors satisfy the following set of Bellman equations:

$$r_\ell^* = \sum_{i=1}^{n} d_{\ell i} \tilde{J}_0(i), \qquad \ell = 1, \ldots, q,$$

$$\tilde{J}_0(i) = \min_{u \in U(i)} \sum_{j_1=1}^{n} p_{ij_1}(u)\big(g(i,u,j_1) + \alpha \tilde{J}_1(j_1)\big), \qquad i = 1, \ldots, n, \tag{4.42}$$

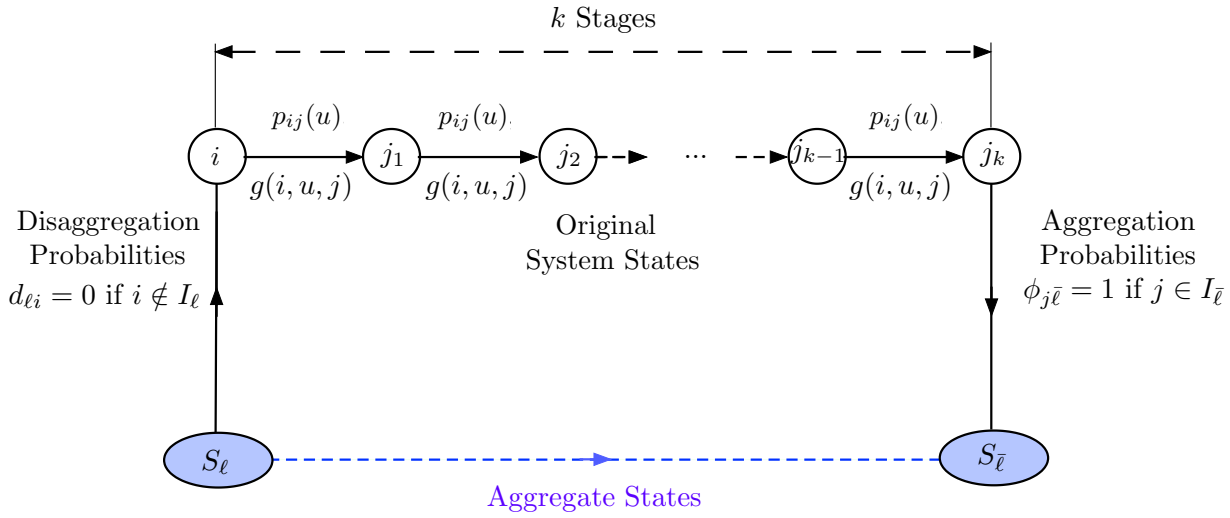


Figure 4.14 The transition mechanism for multistep aggregation. It is based on a dynamical system involving k transitions between original system states interleaved between transitions from and to aggregate states.

$$\tilde{J}_m(j_m) = \min_{u \in U(j_m)} \sum_{j_{m+1}=1}^{n} p_{j_m j_{m+1}}(u)\big(g(j_m, u, j_{m+1}) + \alpha \tilde{J}_{m+1}(j_{m+1})\big), \qquad j_m = 1, \ldots, n, \ m = 1, \ldots, k-1, \tag{4.43}$$

$$\tilde{J}_k(j_k) = \sum_{\ell=1}^{q} \phi_{j_k \ell} r^*_\ell, \qquad j_k = 1, \ldots, n. \tag{4.44}$$

By combining these equations, we obtain an equation for r^*:

$$r^* = DT^k(\Phi r^*),$$

where T is the usual DP mapping of the original problem [the case $k = 1$ corresponds to Eqs. (4.11)-(4.12)]. As earlier, it can be seen that the associated mapping $DT^k\Phi$ is a contraction mapping with respect to the sup-norm, but its contraction modulus is α^k rather than α.

There is a similar mapping corresponding to a fixed policy and it can be used to implement a PI algorithm, which evaluates a policy through calculation of a corresponding parameter vector r and then improves it. However, there is a major difference from the single-step aggregation case: a policy involves a set of k control functions $\{\mu_0, \ldots, \mu_{k-1}\}$, and while a known policy can be easily simulated, its improvement involves multistep lookahead using the minimizations of Eqs. (4.42)-(4.44), and may be costly. Thus the preceding implementation of multistep aggregation-based PI is a useful idea only for problems where the cost of this multistep lookahead minimization (for a single given starting state) is not prohibitive.

On the other hand, from a theoretical point of view, a multistep aggregation scheme provides a means of better approximation of the true optimal cost vector J^*, independent of the use of a large number of

aggregate states. This can be seen from Eqs. (4.42)-(4.44), which by classical value iteration convergence results, show that $\tilde{J}_0(i) \to J^*(i)$ as $k \to \infty$, regardless of the choice of aggregate states. Moreover, because the modulus of the underlying contraction is α^k, we can verify an improved error bound in place of the bound (4.17) of Prop. 4.1, which corresponds to $k = 1$:

$$\left|J^*(i) - r^*_\ell\right| \leq \frac{\epsilon}{1-\alpha^k}, \qquad \forall\ i \text{ such that } i \in I_\ell,\ \ell = 1, \ldots, q,$$

where ϵ is given by Eq. (4.18). The proof is very similar to the one of Prop. 4.1.

λ-Aggregation

Multistep aggregation need not involve sequences of a fixed number of transitions between original system states. The number of transitions may be state-dependent or may be controlled by some randomized mechanism. In one such possibility, called λ-*aggregation*, we introduce a parameter $\lambda \in (0, 1)$ and consider a Markov chain that makes a transition with probability $1 - \lambda$ from an original system state to an aggregate state at each step, rather than with certainty after k steps as in Fig. 4.14. Then it can be shown that the cost vector of a given stationary policy μ, may be evaluated approximately by Φr_μ, where r_μ is the solution of the equation

$$r = DT^{(\lambda)}_\mu(\Phi r), \tag{4.45}$$

where $T^{(\lambda)}_\mu$ is the mapping given by Eq. (2.14). This equation has a unique solution because the mapping $DT^{(\lambda)}_\mu\Phi$ can be shown to be a contraction mapping with respect to the sup-norm.

As noted earlier, the aggregation equation

$$\Phi r = \Phi DT_\mu(\Phi r)$$

is a projected equation because ΦD is a projection mapping with respect to a suitable weighted Euclidean seminorm (see [YuB12], Section 4; it is a norm projection in the case of hard aggregation). Similarly, the λ-aggregation equation

$$\Phi r = \Phi DT^{(\lambda)}_\mu(\Phi r)$$

is a projected equation, which is related to the proximal algorithm [Ber16a], [Ber18b], and may be solved by using temporal differences. Thus we may use exploration-enhanced versions of the LSTD(λ) and LSPE(λ) methods in an approximate PI scheme to solve the λ-aggregation equation. We refer to [Ber12] for further discussion.

5. POLICY ITERATION WITH FEATURE-BASED AGGREGATION AND A NEURAL NETWORK

We noted in Section 3 that neural networks can be used to construct features at the output of the last nonlinear layer. The neural network training process also yields linear weighting parameters for the feature

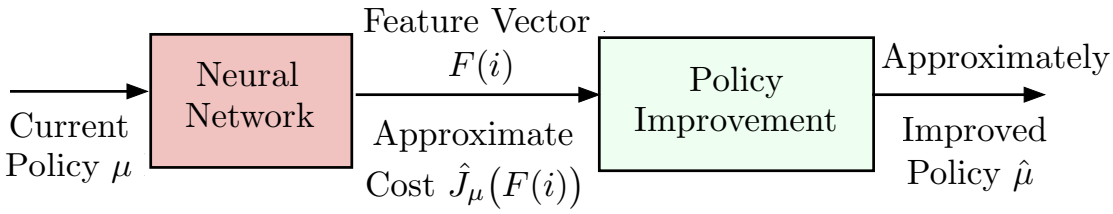


Figure 5.1. Schematic illustration of PI using a neural network-based cost approximation. Starting with a training set of state-cost pairs generated using the current policy μ, the neural network yields a set of features and an approximate cost evaluation $\hat{J}_\mu$ using a linear combination of the features. This is followed by policy improvement using $\hat{J}_\mu$ to generate the new policy $\hat{\mu}$.

vector $F(i)$ at the output of the last layer, thus obtaining an approximation $\hat{J}_\mu\big(F(i)\big)$ to the cost function of a given policy μ. Thus given the current policy μ, the typical PI produces the new policy $\hat{\mu}$ using the approximate policy improvement operation (1.3) or a multistep variant, as illustrated in Fig. 5.1.

A similar PI scheme can be constructed based on feature-based aggregation with features supplied by the same neural network; see Fig. 5.2. The main idea is to replace the (approximate) policy improvement operation with the solution of an aggregate problem, which provides the (approximately) improved policy $\hat{\mu}$. This is a more complicated policy improvement operation, but computes the new policy $\hat{\mu}$ based on a more accurate cost function approximation: one that is a nonlinear function of the features rather than linear. Moreover, $\hat{\mu}$ *not only aspires to be an improved policy relative to* μ*, but also to be an optimal policy based on the aggregate problem*, an approximation itself of the original DP problem. In particular, suppose that the neural network approximates J_μ perfectly. Then the scheme of Fig. 5.1 will replicate a single step of the PI algorithm starting from μ, while the aggregation scheme of Fig. 5.2, with sufficient number of aggregate states, will produce a policy that is arbitrarily close to optimal.

Let us now explain each of the steps of the aggregation-based PI procedure of Fig. 5.2, starting with the current policy μ.

(a) *Feature mapping construction*: We train the neural network using a training set of state-cost pairs that are generated using the current policy μ. This provides a feature vector $F(i)$ as described in Section 3.

(b) *Sampling to obtain the disaggregation sets*: We sample the state space, generating a subset of states $I \subset \{1, \ldots, n\}$. We partition the corresponding set of state-feature pairs

$$\big\{(i, F(i)) \mid i \in I\big\}$$

into a collection of subsets $S_1, \ldots, S_q$. We then consider the aggregation framework with $S_1, \ldots, S_q$ as the aggregate states, and the corresponding aggregate problem as described in Section 4. The sampling to obtain the set of states I may be combined with exploration to ensure that a sufficiently representative set of states is included.

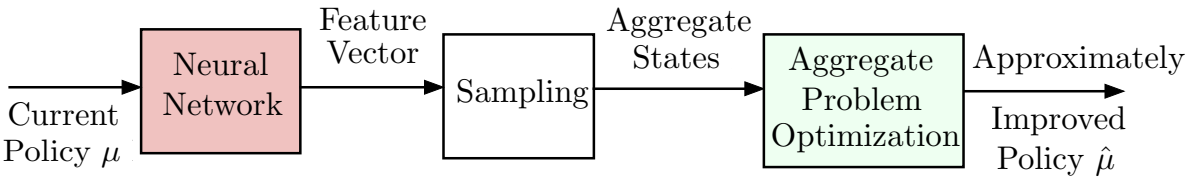


Figure 5.2. Illustration of PI using feature-based aggregation with features supplied by a neural network. Starting with a training set of state-cost pairs generated using the current policy μ, the neural network yields a set of features, which are used to construct a feature-based aggregation framework. The optimal policy of the corresponding aggregate problem is used as the new policy $\hat{\mu}$.

(c) *Aggregate problem solution*: The aggregate DP problem is solved by using a simulation-based method to yield (perhaps approximately) the aggregate state optimal costs r^*_ℓ, $\ell = 1, \ldots, q$ (cf. Section 4.2).

(d) *Definition of the improved policy*: The "improved" policy is simply the optimal policy of the aggregate problem (or an approximation thereof, obtained for example after a few iterations of approximate simulation-based PI). This policy is defined implicitly by the aggregate costs r^*_ℓ, $\ell = 1, \ldots, q$, using the one-step lookahead minimization

$$\hat{\mu}(i) \in \arg\min_{u \in U(i)} \sum_{j=1}^{n} p_{ij}(u) \left(g(i,u,j) + \alpha \sum_{\ell=1}^{q} \phi_{j\ell}\, r^*_\ell \right), \qquad i = 1, \ldots, n,$$

[cf. Eq. (4.9)] or a multistep lookahead variant. Alternatively, the "improved" policy can implemented in model-free fashion using a Q-factor architecture $\tilde{Q}(i,u,\theta)$, as described in Sections 2.4 and 4.1, cf. Eqs. (4.14)-(4.16).

Let us also note that there are several options for implementing the algorithmic ideas of this section.

(1) The neural network-based feature construction process may be performed any number of times, each time followed by an aggregate problem solution that constructs a new policy, which is then used to generate new training data for the neural network. Alternatively, the neural network training and feature construction process may be done only once, followed by the solution of the corresponding feature-based aggregate problem.

(2) Several deep neural network-based PI cycles may be performed, a subset of the features thus generated may be selected, and the corresponding aggregate problem is solved just once, as a way of improving the final policy generated by the deep reinforcement learning process.

(3) Following each cycle of neural network-based feature evaluation, the generated features may be supplemented with additional problem-specific handcrafted features, and/or features from previous cycles. This is a form of feature iteration that was noted in the preceding section.

Finally, let us mention a potential weakness of using the features obtained at the output of the last

nonlinear layer of the neural network in the context of aggregation: the sheer number of these features may be so large that the resulting number of aggregate states may become excessive. To address this situation one may consider pruning some of the features, or reducing their number using some form of regression, at the potential loss of some approximation accuracy. In this connection let us also emphasize a point made earlier in connection with an advantage of deep (rather than shallow) neural networks: *because with each additional layer, the generated features tend to be more complex, their number at the output of the final nonlinear layer of the network can be made smaller as the number of layers increases*. An extreme case is to use the cost function approximation obtained at the output of the neural network as a single feature/scoring function, in the spirit of Section 4.3.

Using Neural Networks in Conjunction with Heuristics

We noted at the end of Section 4.3 another use of neural networks in conjunction with aggregation: somehow construct multiple policies, evaluate each of these policies using a neural network, and use the policy cost function evaluations as multiple scoring functions in a feature-based aggregation scheme. In Section 4.4, we elaborated on this idea for the case of the deterministic discrete optimization problem

$$\text{minimize} \quad G(u_1, \ldots, u_N)$$

$$\text{subject to} \quad (u_1, \ldots, u_N) \in U,$$

where U is a finite set of feasible solutions and G is a cost function [cf. Eq. (4.30)]. We described the use of multiple heuristics to construct corresponding scoring functions. At any given m-solution, the scoring function values are computed by running each of the heuristics. A potential time-saving alternative is to approximate these scoring functions using neural networks.

In particular, for each of the heuristics, we may train a separate neural network by using a training set consisting of pairs of m-solutions and corresponding heuristic costs. In this way we can obtain approximate scoring functions

$$\tilde{V}_1(u_1, \ldots, u_m; \theta_1), \ldots, \tilde{V}_s(u_1, \ldots, u_m; \theta_s),$$

where $\theta_1, \ldots, \theta_s$ are the corresponding neural network weight vectors. We may then use the approximate scoring functions as features in place of the exact heuristic cost functions to construct an aggregate problem similar to the one described in Section 4.4. The solution of the aggregate problem can be used in turn to define a new policy, which may optionally be added to the current set of heuristics, as discussed earlier.

Note that a separate neural network is needed for each heuristic and stage, so assembling the training data together with the training itself can be quite time consuming. However, both the data collection and the training processes can benefit greatly from parallelization.

Finally, let us note that the approach of using a neural network to obtain approximate scoring functions may also be used in conjunction with a rollout scheme that uses a limited horizon. In such a scheme, starting

from an m-solution, we may evaluate all possible subsequent $(m+1)$-solutions by running each of the s heuristics up to a certain horizon depth of d steps [rather than the full depth of $(N-m-1)$ steps], and then approximate the subsequent heuristic cost [from stage $(m+1+d)$ to stage N] by using the neural network estimates.

6. CONCLUDING REMARKS

We have surveyed some aspects of approximate PI methods with a focus on a new idea for policy improvement: feature-based aggregation that uses features provided by a neural network or a heuristic scheme, perhaps in combination with additional handcrafted features. We have argued that this type of policy improvement, while more time-consuming, may yield more effective policies, owing to the DP character of the aggregate problem and the use of a nonlinear feature-based architecture. The algorithmic idea of this paper seems to work well on small examples. However, tests with challenging problems are needed to fully evaluate its merits, particularly since solving the aggregate DP problem is more time-consuming than the standard one-step lookahead policy improvement scheme of Eq. (2.20) or its multistep lookahead variants.

In this paper we have focused on finite-state discounted Markov decision problems, but our approach clearly extends to other types of finite-state DP involving stochastic uncertainty, including finite horizon, stochastic shortest path, and semi-Markov decision problems. It is also worth considering extensions to infinite-state problems, including those arising in the context of continuous spaces optimal control, shortest path, and partially observed Markov decision problems. Generally, the construction of aggregation frameworks for continuous spaces problems is conceptually straightforward, and follows the pattern discussed in this paper for finite-state problems. For example a hard aggregation scheme involves a partition of the continuous state space into a finite number of subsets/aggregate states, while a representative states scheme involves discretization of the continuous state space using a finite number of states. Note, however, that from a mathematical point of view, there may be a substantial issue of consistency, i.e., whether the solution of the aggregate problem "converges" to the solution of the continuous spaces problem as the number of aggregate states increases. Part of the reason has to do with the fact that the Bellman equation of continuous spaces problems need not have a unique solution. The author's monograph [Ber18a], Sections 4.5 and 4.6, provides an analysis of this question for shortest path and optimal control problems with a continuous state space, and identifies classes of problems that are more amenable to approximate DP solution approaches.

Finally, we note that the key issue of feature construction can be addressed in a number of ways. In this paper we have focused on the use of deep neural networks and heuristics for approximating the optimal cost function or the cost functions of policies. However, we may use instead any methodology that automatically constructs good features at reasonable computational cost.

7. REFERENCES

[ADB17] Arulkumaran, K., Deisenroth, M. P., Brundage, M. and Bharath, A. A., 2017. "A Brief Survey of Deep Reinforcement Learning," arXiv preprint arXiv:1708.05866.

[Abr90] Abramson, B., 1990. "Expected-Outcome: A General Model of Static Evaluation," IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. 12, pp. 182-193.

[BBD10] Busoniu, L., Babuska, R., De Schutter, B., and Ernst, D., 2010. Reinforcement Learning and Dynamic Programming Using Function Approximators, CRC Press, N. Y.

[BBS95] Barto, A. G., Bradtke, S. J., and Singh, S. P., 1995. "Real-Time Learning and Control Using Asynchronous Dynamic Programming," Artificial Intelligence, Vol. 72, pp. 81-138.

[BPW12] Browne, C., Powley, E., Whitehouse, D., Lucas, L., Cowling, P. I., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., and Colton, S., 2012. "A Survey of Monte Carlo Tree Search Methods," IEEE Trans. on Computational Intelligence and AI in Games, Vol. 4, pp. 1-43.

[BSA83] Barto, A. G., Sutton, R. S., and Anderson, C. W., 1983. "Neuronlike Elements that Can Solve Difficult Learning Control Problems," IEEE Trans. on Systems, Man, and Cybernetics, Vol. 13, pp. 835-846.

[BTW97] Bertsekas, D. P., Tsitsiklis, J. N., and Wu, C., 1997. "Rollout Algorithms for Combinatorial Optimization," Heuristics, Vol. 3, pp. 245-262.

[BeC89] Bertsekas, D. P., and Castanon, D. A., 1989. "Adaptive Aggregation Methods for Infinite Horizon Dynamic Programming," IEEE Trans. on Aut. Control, Vol. AC-34, pp. 589-598.

[BeC99] Bertsekas, D. P., and Castanon, D. A., 1999. "Rollout Algorithms for Stochastic Scheduling Problems," Heuristics, Vol. 5, pp. 89-108.

[BeT91] Bertsekas, D. P., and Tsitsiklis, J. N., 1991. "An Analysis of Stochastic Shortest Path Problems," Math. Operations Research, Vol. 16, pp. 580-595.

[BeT96] Bertsekas, D. P., and Tsitsiklis, J. N., 1996. Neuro-Dynamic Programming, Athena Scientific, Belmont, MA.

[BeT00] Bertsekas, D. P., and Tsitsiklis, J. N., 2000. "Gradient Convergence in Gradient Methods," SIAM J. on Optimization, Vol. 10, pp. 627-642.

[Ber95] Bertsekas, D. P., 1995. "A Counterexample to Temporal Differences Learning," Neural Computation, Vol. 7, pp. 270-279.

[Ber11a] Bertsekas, D. P., 2011. "Approximate Policy Iteration: A Survey and Some New Methods," J. of Control Theory and Applications, Vol. 9, pp. 310-335.

[Ber11b] Bertsekas, D. P., 2011. "Temporal Difference Methods for General Projected Equations," IEEE

Trans. on Aut. Control, Vol. 56, pp. 2128-2139.

[Ber11c] Bertsekas, D. P., 2011. "λ-Policy Iteration: A Review and a New Implementation," Lab. for Information and Decision Systems Report LIDS-P-2874, MIT; in *Reinforcement Learning and Approximate Dynamic Programming for Feedback Control*, by F. Lewis and D. Liu (eds.), IEEE Press, Computational Intelligence Series, 2012.

[Ber12] Bertsekas, D. P., 2012. Dynamic Programming and Optimal Control, Vol. II: Approximate Dynamic Programming, 4th edition, Athena Scientific, Belmont, MA.

[Ber13] Bertsekas, D. P., 2013. "Rollout Algorithms for Discrete Optimization: A Survey," Handbook of Combinatorial Optimization, Springer.

[Ber15] Bertsekas, D. P., 2015. Convex Optimization Algorithms, Athena Scientific, Belmont, MA.

[Ber16a] Bertsekas, D. P., 2016. "Proximal Algorithms and Temporal Differences for Large Linear Systems: Extrapolation, Approximation, and Simulation," Report LIDS-P-3205, MIT; arXiv preprint arXiv:1610.05427.

[Ber16b] Bertsekas, D. P., 2016. Nonlinear Programming, 3rd edition, Athena Scientific, Belmont, MA.

[Ber17] Bertsekas, D. P., 2017. Dynamic Programming and Optimal Control, Vol. I, 4th edition, Athena Scientific, Belmont, MA.

[Ber18a] Bertsekas, D. P., 2018. Abstract Dynamic Programming, Athena Scientific, Belmont, MA.

[Ber18b] Bertsekas, D. P., 2018. "Proximal Algorithms and Temporal Difference Methods for Solving Fixed Point Problems," Computational Optimization and Applications J., Vol. 70, pp. 709-736.

[Bis95] Bishop, C. M, 1995. Neural Networks for Pattern Recognition, Oxford University Press, N. Y.

[CFH05] Chang, H. S., Hu, J., Fu, M. C., and Marcus, S. I., 2005. "An Adaptive Sampling Algorithm for Solving Markov Decision Processes," Operations Research, Vol. 53, pp. 126-139.

[CFH13] Chang, H. S., Hu, J., Fu, M. C., and Marcus, S. I., 2013. Simulation-Based Algorithms for Markov Decision Processes, (2nd Ed.), Springer, N. Y.

[Cao07] Cao, X. R., 2007. Stochastic Learning and Optimization: A Sensitivity-Based Approach, Springer, N. Y.

[ChK86] Christensen, J., and Korf, R. E., 1986. "A Unified Theory of Heuristic Evaluation Functions and its Application to Learning," in Proceedings AAAI-86, pp. 148-152.

[ChM82] Chatelin, F., and Miranker, W. L., 1982. "Acceleration by Aggregation of Successive Approximation Methods," Linear Algebra and its Applications, Vol. 43, pp. 17-47.

[CiS15] Ciosek, K., and Silver, D., 2015. "Value Iteration with Options and State Aggregation," Report, Centre for Computational Statistics and Machine Learning University College London.

[Cou06] Coulom, R., 2006. "Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search," International Conference on Computers and Games, Springer, pp. 72-83.

[Cyb89] Cybenko, 1989. "Approximation by Superpositions of a Sigmoidal Function," Math. of Control, Signals, and Systems, Vol. 2, pp. 303-314.

[DNW16] David, O. E., Netanyahu, N. S., and Wolf, L., 2016. "Deepchess: End-to-End Deep Neural Network for Automatic Learning in Chess," in International Conference on Artificial Neural Networks, pp. 88-96, Springer.

[DiM10] Di Castro, D., and Mannor, S., 2010. "Adaptive Bases for Reinforcement Learning," Machine Learning and Knowledge Discovery in Databases, Vol. 6321, pp. 312-327.

[Die00] Dietterich, T., 2000. "Hierarchical Reinforcement Learning with the MAXQ Value Function Decomposition," J. of Artificial Intelligence Research, Vol. 13, pp. 227-303.

[FYG06] Fern, A., Yoon, S. and Givan, R., 2006. "Approximate Policy Iteration with a Policy Language Bias: Solving Relational Markov Decision Processes," J. of Artificial Intelligence Research, Vol. 25, pp. 75-118.

[DoD93] Douglas, C. C., and Douglas, J., 1993. "A Unified Convergence Theory for Abstract Multigrid or Multilevel Algorithms, Serial and Parallel," SIAM J. Num. Anal., Vol. 30, pp. 136-158.

[Fle84] Fletcher, C. A. J., 1984. Computational Galerkin Methods, Springer, N. Y.

[Fun89] Funahashi, K., 1989. "On the Approximate Realization of Continuous Mappings by Neural Networks," Neural Networks, Vol. 2, pp. 183-192.

[GBC16] Goodfellow, I., Bengio, J., and Courville, A., Deep Learning, MIT Press, Cambridge, MA.

[GGS13] Gabillon, V., Ghavamzadeh, M., and Scherrer, B., 2013. "Approximate Dynamic Programming Finally Performs Well in the Game of Tetris," in Advances in Neural Information Processing Systems, pp. 1754-1762.

[Gor95] Gordon, G. J., 1995. "Stable Function Approximation in Dynamic Programming," in Machine Learning: Proceedings of the 12th International Conference, Morgan Kaufmann, San Francisco, CA.

[Gos15] Gosavi, A., 2015. Simulation-Based Optimization: Parametric Optimization Techniques and Reinforcement Learning, 2nd Edition, Springer, N. Y.

[HMK98] Hauskrecht, M., Meuleau, N., Kaelbling, L. P., Dean, T., and Boutilier, C., 1998. "Hierarchical Solution of Markov Decision Processes Using Macro-Actions," in Proceedings of the 14th Conference on Uncertainty in Artificial Intelligence, pp. 220-229.

[HOT06] Hinton, G. E., Osindero, S., and Teh, Y. W., 2006. "A Fast Learning Algorithm for Deep Belief Nets," Neural Computation, Vol. 18, pp. 1527-1554.

[HSW89] Hornick, K., Stinchcombe, M., and White, H., 1989. "Multilayer Feedforward Networks are Uni-

versal Approximators," Neural Networks, Vol. 2, pp. 359-159.

[Hay08] Haykin, S., 2008. Neural Networks and Learning Machines, (3rd Edition), Prentice-Hall, Englewood-Cliffs, N. J.

[Hol86] Holland, J. H., 1986. "Escaping Brittleness: the Possibility of General-Purpose Learning Algorithms Applied to Rule-Based Systems," in Machine Learning: An Artificial Intelligence Approach, Michalski, R. S., Carbonell, J. G., and Mitchell, T. M., (eds.), Morgan Kaufmann, San Mateo, CA, pp. 593-623.

[Iva68] Ivakhnenko, A. G., 1968. "The Group Method of Data Handling: A Rival of the Method of Stochastic Approximation," Soviet Automatic Control, Vol. 13, pp. 43-55.

[Iva71] Ivakhnenko, A. G., 1971. "Polynomial Theory of Complex Systems," IEEE Transactions on Systems, Man and Cybernetics, Vol. 4, pp. 364-378.

[Jon90] Jones, L. K., 1990. "Constructive Approximations for Neural Networks by Sigmoidal Functions," Proceedings of the IEEE, Vol. 78, pp. 1586-1589.

[KMP06] Keller, P. W., Mannor, S., and Precup, D., 2006. "Automatic Basis Function Construction for Approximate Dynamic Programming and Reinforcement Learning," in Proc. of the 23rd International Conference on Machine Learning, ACM, pp. 449-456.

[KVZ72] Krasnoselskii, M. A., Vainikko, G. M., Zabreyko, R. P., and Ruticki, Ya. B., 1972. Approximate Solution of Operator Equations, Translated by D. Louvish, Wolters-Noordhoff Pub., Groningen.

[Kir11] Kirsch, A., 2011. An Introduction to the Mathematical Theory of Inverse Problems, (2nd Edition), Springer, N. Y.

[KoB09] Konidaris, G., and Barto, A., 2009. "Efficient Skill Learning Using Abstraction Selection," in 21st International Joint Conference on Artificial Intelligence.

[LLL08] Lewis, F. L., Liu, D., and Lendaris, G. G., 2008. Special Issue on Adaptive Dynamic Programming and Reinforcement Learning in Feedback Control, IEEE Trans. on Systems, Man, and Cybernetics, Part B, Vol. 38, No. 4.

[LLP93] Leshno, M., Lin, V. Y., Pinkus, A., and Schocken, S., 1993. "Multilayer Feedforward Networks with a Nonpolynomial Activation Function can Approximate any Function," Neural Networks, Vol. 6, pp. 861-867.

[LWL17] Liu, W., Wang, Z., Liu, X., Zeng, N., Liu, Y., and Alsaadi, F. E., 2017. "A Survey of Deep Neural Network Architectures and their Applications," Neurocomputing, Vol. 234, pp. 11-26.

[LeL12] Lewis, F. L., and Liu, D., 2012. Reinforcement Learning and Approximate Dynamic Programming for Feedback Control, IEEE Press Computational Intelligence Series, N. Y.

[Li17] Li, Y., 2017. "Deep Reinforcement Learning: An Overview," arXiv preprint ArXiv: 1701.07274v5.

[MMP15] Mann, T.A., Mannor, S. and Precup, D., 2015. "Approximate Value Iteration with Temporally Extended Actions," J. of Artificial Intelligence Research, Vol. 53, pp. 375-438.

[MMS06] Menache, I., Mannor, S., and Shimkin, N., 2005. "Basis Function Adaptation in Temporal Difference Reinforcement Learning," Ann. Oper. Res., Vol. 134, pp. 215-238.

[Men82] Mendelssohn, R., 1982. "An Iterative Aggregation Procedure for Markov Decision Processes," Operations Research, Vol. 30, pp. 62-73.

[PaR98] Parr, R., and Russell, S. J., 1998. "Reinforcement Learning with Hierarchies of Machines," in Advances in Neural Information Processing Systems, pp. 1043-1049.

[Pow11] Powell, W. B., 2011. Approximate Dynamic Programming: Solving the Curses of Dimensionality, 2nd Edition, J. Wiley and Sons, Hoboken, N. J.

[RPW91] Rogers, D. F., Plante, R. D., Wong, R. T., and Evans, J. R., 1991. "Aggregation and Disaggregation Techniques and Methodology in Optimization," Operations Research, Vol. 39, pp. 553-582.

[RSS12] Runarsson, T. P., Schoenauer, M., and Sebag, M., 2012. "Pilot, Rollout and Monte Carlo Tree Search Methods for Job Shop Scheduling," in Learning and Intelligent Optimization (pp. 160-174), Springer, Berlin, Heidelberg.

[SBP04] Si, J., Barto, A., Powell, W., and Wunsch, D., (Eds.) 2004. Learning and Approximate Dynamic Programming, IEEE Press, N. Y.

[SGG15] Scherrer, B., Ghavamzadeh, M., Gabillon, V., Lesner, B., and Geist, M., 2015. "Approximate Modified Policy Iteration and its Application to the Game of Tetris," J. of Machine Learning Research, Vol. 16, pp. 1629-1676.

[SHS17] Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T. and Lillicrap, T., 2017. "Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm," arXiv preprint arXiv:1712.01815.

[SJJ95] Singh, S. P., Jaakkola, T., and Jordan, M. I., 1995. "Reinforcement Learning with Soft State Aggregation," in Advances in Neural Information Processing Systems 7, MIT Press, Cambridge, MA.

[SPS99] Sutton, R., Precup, D., and Singh, S., 1999. "Between MDPs and Semi-MDPs: A Framework for Temporal Abstraction in Reinforcement Learning," Artificial Intelligence, Vol. 112, pp. 181-211.

[SSP18] Serban, I. V., Sankar, C., Pieper, M., Pineau, J., Bengio, J., 2018. "The Bottleneck Simulator: A Model-Based Deep Reinforcement Learning Approach," arXiv preprint arXiv:1807.04723.v1.

[Saa03] Saad, Y., 2003. Iterative Methods for Sparse Linear Systems, SIAM, Phila., Pa.

[Sam59] Samuel, A. L., 1959. "Some Studies in Machine Learning Using the Game of Checkers," IBM J. of Research and Development, pp. 210-229.

[Sam67] Samuel, A. L., 1967. "Some Studies in Machine Learning Using the Game of Checkers. II – Recent Progress," IBM J. of Research and Development, pp. 601-617.

[Sch13] Scherrer, B., 2013. "Performance Bounds for Lambda Policy Iteration and Application to the Game of Tetris," J. of Machine Learning Research, Vol. 14, pp. 1181-1227.

[Sch15] Schmidhuber, J., 2015. "Deep Learning in Neural Networks: An Overview," Neural Networks, Vol. 61, pp. 85-117.

[Sha50] Shannon, C., 1950. "Programming a Digital Computer for Playing Chess," Phil. Mag., Vol. 41, pp. 356-375.

[SuB98] Sutton, R. S., and Barto, A. G., 1998. Reinforcement Learning, MIT Press, Cambridge, MA. (A draft 2nd edition is available on-line.)

[Sut88] Sutton, R. S., 1988. "Learning to Predict by the Methods of Temporal Differences," Machine Learning, Vol. 3, pp. 9-44.

[Sze10] Szepesvari, C., 2010. Algorithms for Reinforcement Learning, Morgan and Claypool Publishers, San Franscisco, CA.

[TeG96] Tesauro, G., and Galperin, G. R., 1996. "On-Line Policy Improvement Using Monte Carlo Search," presented at the 1996 Neural Information Processing Systems Conference, Denver, CO; also in M. Mozer et al. (eds.), Advances in Neural Information Processing Systems 9, MIT Press (1997).

[Tes89a] Tesauro, G. J., 1989. "Neurogammon Wins Computer Olympiad," Neural Computation, Vol. 1, pp. 321-323.

[Tes89b] Tesauro, G. J., 1989. "Connectionist Learning of Expert Preferences by Comparison Training," in Advances in Neural Information Processing Systems, pp. 99-106.

[Tes92] Tesauro, G. J., 1992. "Practical Issues in Temporal Difference Learning," Machine Learning, Vol. 8, pp. 257-277.

[Tes94] Tesauro, G. J., 1994. "TD-Gammon, a Self-Teaching Backgammon Program, Achieves Master-Level Play," Neural Computation, Vol. 6, pp. 215-219.

[Tes95] Tesauro, G. J., 1995. "Temporal Difference Learning and TD-Gammon," Communications of the ACM, Vol. 38, pp. 58-68.

[Tes01] Tesauro, G. J., 2001. "Comparison Training of Chess Evaluation Functions," in Machines that Learn to Play Games, Nova Science Publishers, pp. 117-130.

[Tes02] Tesauro, G. J., 2002. "Programming Backgammon Using Self-Teaching Neural Nets," Artificial Intelligence, Vol. 134, pp. 181-199.

[TsV96] Tsitsiklis, J. N., and Van Roy, B., 1996. "Feature-Based Methods for Large-Scale Dynamic Pro-

gramming," Machine Learning, Vol. 22, pp. 59-94.

[Tsi94] Tsitsiklis, J. N., 1994. "Asynchronous Stochastic Approximation and Q-Learning," Machine Learning, Vol. 16, pp. 185-202.

[VVL13] Vrabie, V., Vamvoudakis, K. G., and Lewis, F. L., 2013. Optimal Adaptive Control and Differential Games by Reinforcement Learning Principles, The Institution of Engineering and Technology, London.

[Van06] Van Roy, B., 2006. "Performance Loss Bounds for Approximate Value Iteration with State Aggregation," Mathematics of Operations Research, Vol. 31, pp. 234-244.

[Wer77] Werbös, P. J., 1977. "Advanced Forecasting Methods for Global Crisis Warning and Models of Intelligence," General Systems Yearbook, Vol. 22, pp. 25-38.

[YuB04] Yu, H., and Bertsekas, D. P., 2004. "Discretized Approximations for POMDP with Average Cost," Proc. of the 20th Conference on Uncertainty in Artificial Intelligence, Banff, Canada.

[YuB09] Yu, H., and Bertsekas, D. P., 2009. "Basis Function Adaptation Methods for Cost Approximation in MDP," Proceedings of 2009 IEEE Symposium on Approximate Dynamic Programming and Reinforcement Learning (ADPRL 2009), Nashville, Tenn.

[YuB10] Yu, H., and Bertsekas, D. P., 2010. "Error Bounds for Approximations from Projected Linear Equations," Mathematics of Operations Research, Vol. 35, pp. 306-329.

[YuB12] Yu, H., and Bertsekas, D. P., 2012. "Weighted Bellman Equations and their Applications in Dynamic Programming," Lab. for Information and Decision Systems Report LIDS-P-2876, MIT.