Foundations and Trends® in Machine Learning
Vol. 6, No. 4 (2013) 375–XXX
© 2013 A. Geramifard, T. J. Walsh, S. Tellex, G. Chowdhary,
N. Roy, and J. P. How
DOI: 10.1561/2200000042

now
the essence of knowledge

A Tutorial on Linear Function Approximators for Dynamic Programming and Reinforcement Learning

Alborz Geramifard Thomas J. Walsh Stefanie Tellex
Girish Chowdhary Nicholas Roy
Jonathan P. How

Massachusetts Institute of Technology

Contents

Abstract

A Markov Decision Process (MDP) is a natural framework for formulating sequential decision-making problems under uncertainty. In recent years, researchers have greatly advanced algorithms for learning and acting in MDPs. This article reviews such algorithms, beginning with well-known dynamic programming methods for solving MDPs such as policy iteration and value iteration, then describes approximate dynamic programming methods such as trajectory based value iteration, and finally moves to reinforcement learning methods such as Q-Learning, SARSA, and least-squares policy iteration. We describe algorithms in a unified framework, giving pseudocode together with memory and iteration complexity analysis for each. Empirical evaluations of these techniques with four representations across four domains, provide insight into how these algorithms perform with various feature sets in terms of running time and performance.

A. Geramifard, T. J. Walsh, S. Tellex, G. Chowdhary, N. Roy, and J. P. How. *A Tutorial on Linear Function Approximators for Dynamic Programming and Reinforcement Learning*. Foundations and Trends® in Machine Learning, vol. 6, no. 4, pp. 375–XXX, 2013.

DOI: 10.1561/2200000042.

1

Introduction

Designing agents to act near-optimally in stochastic sequential domains is a challenging problem that has been studied in a variety of settings. When the domain is known, analytical techniques such as *dynamic programming* (DP) [Bellman, 1957] are often used to find optimal policies for the agent. When the domain is initially unknown, *reinforcement learning* (RL) [Sutton and Barto, 1998] is a popular technique for training agents to act optimally based on their experiences in the world. However, in much of the literature on these topics, small-scale environments were used to verify solutions. For example the famous taxi problem has only 500 states [Dietterich, 2000]. This contrasts with recent success stories in domains previously considered unassailable, such as 9×9 Go [Silver et al., 2012], a game with approximately 10^{38} states. An important factor in creating solutions for such large-scale problems is the use of linear function approximation [Sutton, 1996, Silver et al., 2012, Geramifard et al., 2011]. This approximation technique allows the long-term utility (value) of policies to be represented in a low-dimensional form, dramatically decreasing the number of parameters that need to be learned or stored. This tutorial provides practical guidance for researchers seeking to extend DP and RL techniques to larger domains through linear value function approximation. We introduce DP and RL techniques in a unified frame-

work and conduct experiments in domains with sizes up to ~ 150 million state-action pairs.

Sequential decision making problems with full observability of the states are often cast as Markov Decision Processes (MDPs) [Puterman, 1994]. An MDP consists of a set of states, set of actions available to an agent, rewards earned in each state, and a model for transitioning to a new state given the current state and the action taken by the agent. Ignoring computational limitations, an *agent* with full knowledge of the MDP can compute an optimal policy that maximizes some function of its expected cumulative reward (which is often referred to as the expected *return* [Sutton and Barto, 1998]). This process is known as *planning*. In the case where the MDP is unknown, reinforcement learning agents *learn* to take optimal actions over time merely based on interacting with the world.

A central component for many algorithms that plan or learn to act in an MDP is a *value function*, which captures the long term expected return of a policy for every possible state. The construction of a value function is one of the few common components shared by many planners and the many forms of so-called *value-based* RL methods[1]. In the planning context, where the underlying MDP is known to the agent, the value of a state can be expressed recursively based on the value of successor states, enabling dynamic programming algorithms [Bellman, 1957] to iteratively estimate the value function. If the underlying model is unknown, value-based reinforcement learning methods estimate the value function based on observed state transitions and rewards. However, in either case, maintaining and manipulating the value of every state (*i.e.,* a *tabular representation*) is not feasible in large or continuous domains. In order to tackle practical problems with such large state-action spaces, a value function representation is needed that 1) does not require computation or memory proportional to the size of the number of states, and 2) *generalizes* learned values from data across states (*i.e.,* each new piece of data may change the value of more than one state).

One approach that satisfies these goals is to use linear function approximation to estimate the value function. Specifically, the full set of states is

[1]There are other MDP solving techniques not covered here (such as direct policy search) that do not directly estimate a value function and have been used successfully in many applications, including robotics [Williams, 1992, Sutton et al., 2000, Peters and Schaal, 2006, Baxter and Bartlett, 2000].

projected into a lower dimensional space where the value function is represented as a linear function. This representational technique has succeeded at finding good policies for problems with high dimensional state-spaces such as simulated soccer [Stone et al., 2005b] and Go [Silver et al., 2012]. This tutorial reviews the use of linear function approximation algorithms for decision making under uncertainty in DP and RL algorithms. We begin with classical DP methods for exact planning in decision problems, such as policy iteration and value iteration. Next, we describe approximate dynamic programming methods with linear value function approximation and "trajectory based" evaluations for practical planning in large state spaces. Finally, in the RL setting, we discuss learning algorithms that can utilize linear function approximation, namely: SARSA, Q-learning, and Least-Squares policy iteration. Throughout, we highlight the trade-offs between computation, memory complexity, and accuracy that underlie algorithms in these families.

In Chapter 3, we provide a more concrete overview of practical linear function approximation from the literature and discuss several methods for creating linear bases. We then give a thorough empirical comparison of the various algorithms described in the theoretical section paired with each of these representations. The algorithms are evaluated in multiple domains, several of which have state spaces that render tabular representations intractable. For instance, one of the domains we examine, Persistent Search and Track (PST), involves control of multiple unmanned aerial vehicles in a complex environment. The large number of properties for each robot (fuel level, location, etc.) leads to over 150 million state-action pairs. We show that the linear function approximation techniques described in this tutorial provide tractable solutions for this otherwise unwieldy domain. For our experiments, we used the RLPy framework [Geramifard et al., 2013a] which allows the reproduction of our empirical results.

There are many existing textbooks and reviews of reinforcement learning [Bertsekas and Tsitsiklis, 1996, Szepesvári, 2010, Buşoniu et al., 2010, Gosavi, 2009, Kaelbling et al., 1996, Sutton and Barto, 1998]. This tutorial differentiates itself by providing a narrower focus on the use of linear value function approximation and introducing many DP and RL techniques in a unified framework, where each algorithm is derived from the general concept of policy evaluation/improvement (shown in Figure 2.1). Also, our extensive

empirical evaluation covers a wider range of domains, representations, and algorithms than previous studies. The lessons from these experiments provide a guide to practitioners as they apply DP and RL methods to their own large-scale (and perhaps hitherto intractable) domains.

2

Dynamic Programming and Reinforcement Learning

This chapter provides a formal description of decision-making for stochastic domains, then describes linear value-function approximation algorithms for solving these decision problems. It begins with dynamic programming approaches, where the underlying model is known, then moves to reinforcement learning, where the underlying model is unknown. One of the main goals of this chapter is to show that approximations and sampling restrictions provide a path from exact dynamic programming methods towards reinforcement learning algorithms shown in Figure 2.1 (the terms used in the figure will be explained throughout this chapter). We give pseudocode together with memory and computation complexity analysis. The list of discussed algorithms and their computational complexities are available in Tables 2.2 and 2.3.

2.1 Markov Decision Processes (MDPs)

Following the convention of Sutton and Barto [1998], a Markov Decision Process (MDP) [Puterman, 1994] is a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}^a_{s\cdot}, \mathcal{R}^a_{ss'}, \gamma, S_0)$ where $\mathcal{S}$ is a set of states, $\mathcal{A}$ is a set of actions, $s, s' \in \mathcal{S}$, and $a \in \mathcal{A}$. $\mathcal{P}^a_{s\cdot}$ is a probability distribution over next states if action a is executed at state s. In what follows, for the sake of simplifying the presentation, we restrict our atten-

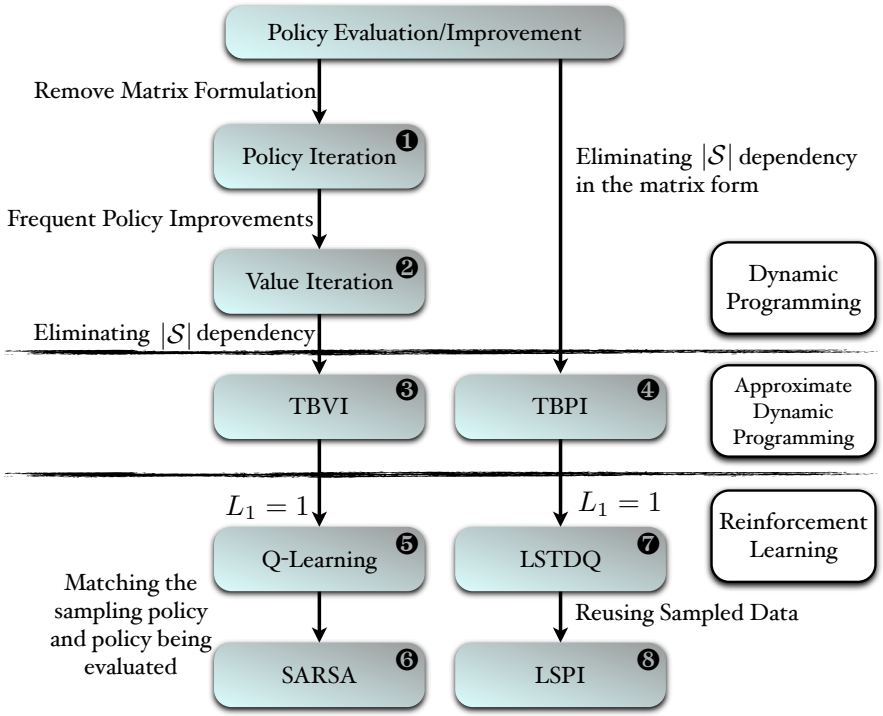

Figure 2.1: The roadmap we use to introduce various DP and RL techniques in a unified framework.

tion to MDPs with countable state spaces. For such MDPs, we denote the probability of getting to state s' by taking action a in state s as $\mathcal{P}^a_{ss'}$. Correspondingly, $\mathcal{R}^a_{ss'}$ is the reward the agent receives when the sequence s, a, s' occurs. The discount factor, $\gamma \in [0, 1)$, balances current and future rewards. The distribution over initial states is governed by S_0. Together, $\mathcal{P}^a_{s\cdot}$ and $\mathcal{R}^a_{ss'}$ constitute the *model* of an MDP. A *trajectory* is a sequence $(s_0, a_0, r_0, s_1, a_1, r_1, s_2, \ldots)$, where $s_0 \sim S_0$. Each state following the initial state in a trajectory is generated by the domain according to the transition model (*i.e.,* for $t \geq 1, s_{t+1} \sim \mathcal{P}^{a_t}_{s_t\cdot}$). Similarly, each reward $r_t = \mathcal{R}^{a_t}_{s_t s_{t+1}}$ for the selected action. Each action $a_t \in \mathcal{A}$ in the trajectory is chosen according to *policy*

$\pi : \mathcal{S} \rightarrow \mathcal{A}$. The policy maps each state to an action.[1] Given a policy π, the state-action value function, $Q^{\pi}(s,a)$ of each state-action pair is the expected sum of the discounted rewards for an agent starting at state s, taking action a, and then following policy π thereafter (denoted E_{π}) :

$$Q^{\pi}(s,a) = E_{\pi}\left[\sum_{t=0}^{\infty}\gamma^{t}r_{t}\middle|s_{0}=s,a_{0}=a\right]. \tag{2.1}$$

The discount factor bounds the summation and reduces the desirability of future rewards.[2] Similarly the state value function, (or called simply the value function), for a given policy π is defined as:

$$V^{\pi}(s) \triangleq Q^{\pi}(s,\pi(s)) \tag{2.2}$$

$$= E_{\pi}\left[\sum_{t=0}^{\infty}\gamma^{t}r_{t}\middle|s_{0}=s\right]. \tag{2.3}$$

The goal of *solving* an MDP is to, given the MDP model, find an optimal policy, one of which always exists [Puterman, 1994], that maximizes the expected cumulative discounted rewards in all states:

$$\pi^{*} = \underset{\pi}{\operatorname{argmax}}\, V^{\pi}(s), \forall s \in \mathcal{S}. \tag{2.4}$$

The optimal value function is defined accordingly as:

$$V^{*}(s) \triangleq V^{\pi^{*}}(s).$$

Similarly:

$$Q^{*}(s,a) \triangleq Q^{\pi^{*}}(s,a).$$

The optimal value function satisfies the Bellman Equation [Puterman, 1994]:

$$V^{*}(s) = \max_{a\in\mathcal{A}}\sum_{s'\in\mathcal{S}}\mathcal{P}^{a}_{ss'}\left[\mathcal{R}^{a}_{ss'}+\gamma V^{*}(s')\right] \quad s\in\mathcal{S}. \tag{2.5}$$

From a practical perspective, calculating the right hand side of the Bellman Equation for MDPs with infinite state and action spaces is challenging because 1) a unique value has to be stored for each state and 2) the maximization in the Equation 2.5 is over all actions. For the rest of the tutorial, we

[1] More generally policies can be based on histories of states but in this work we consider only Markovian policies.

[2] From an economical prospective, one can think that a dollar today is worth more than a dollar tomorrow.

focus our attention on MDPs with finite state and action spaces. However Chapter 3 introduces approximation techniques that can be used for MDPs with continuous state spaces.

2.2 MDP Solvers at a Glance

The rest of this chapter provides an overview of some popular algorithms for solving MDPs. These algorithms are categorized either as *planning*, where the MDP model is known, or *learning*, where the MDP model is not known.[3] This tutorial focuses on dynamic programming methods in the planning category and value-function based reinforcement learning techniques for the learning category. Throughout, we show these techniques are actually extremes on a spectrum: as one decreases the per-iteration complexity of DP methods and forces them to use samples in place of exact parameters, RL techniques naturally emerge. Since the common thread for all of these techniques is that they use a value function, we refer to them as *value-based solvers*.

As shown in Figure 2.2, value-based solvers tackle an MDP in two phases: 1) policy evaluation and 2) policy improvement. In the policy evaluation phase, the solver calculates the value function for some or all states given the fixed policy.[4] In the policy improvement step, the algorithm improves the previous policy based on values obtained in the policy evaluation step. The process of iteratively evaluating and improving the policy continues until either the policy remains unchanged, a time limit has been reached, or the change to the value function is below a certain threshold. Of course, Figure 2.2 does not prescribe exactly how these phases are completed or when an algorithm switches between them. For instance, we will see that some algorithms (like policy iteration) spend significant time in the evaluation phase while others (like value iteration) oscillate more quickly between the phases.

The chapter proceeds as follows: Section 2.3 explains dynamic programming techniques that have access to the MDP model (*i.e.,* $\mathcal{R}$ and $\mathcal{P}$). Sections 2.4-2.6 introduce approximate dynamic programming techniques by elimi-

[3] Mausam and Kolobov [2012] provide a broader overview of MDP solving methods.

[4] Sometime the policy evaluation step only improves the previous value estimates rather than calculating the exact values, known as *truncated policy evaluation* [see Sections 4.4 and 4.6 Sutton and Barto, 1998].

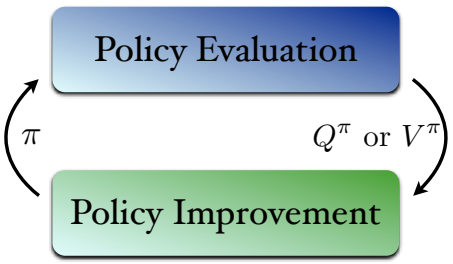

Figure 2.2: Policy evaluation/improvement loop: The convergent policy is guaranteed to be optimal, if the Q or V value functions are exact.

nating all $|\mathcal{S}|$ dependent memory and computation operators of dynamic programming techniques. Finally Section 2.7 shows how reinforcement learning techniques that do not have access to the MDP model can follow naturally from approximate dynamic programming techniques. These connections and the broad outline of the algorithm derivations of this tutorial are illustrated in Figure 2.1.

2.3 Dynamic Programming

Dynamic programming (DP) refers to a class of algorithms that solve complex problems by combining solutions from their subproblems. DP techniques can be used in the planning setting to solve a known MDP by finding the optimal value function and its corresponding optimal policy [Bellman, 1957, Bertsekas and Tsitsiklis, 1996, Sutton and Barto, 1998]. First, let us observe that given an MDP, policy evaluation (*i.e.,* finding the value function under a fixed policy) can be done in closed form. Looking back at the Equation 2.3, the value function can be derived recursively as explained by Sutton and Barto [1998]:

$$
\begin{aligned}
V^{\pi}(s) &= E_{\pi}\left[\sum_{t=0}^{\infty}\gamma^{t}r_{t}\middle|s_{0}=s\right] \\
&= E_{\pi}\left[r_{0}+\sum_{t=1}^{\infty}\gamma^{t}r_{t}\middle|s_{0}=s\right] \\
&= E_{\pi}\left[\mathcal{R}_{ss'}^{\pi(s)}+\gamma\sum_{t=1}^{\infty}\gamma^{t-1}r_{t}\middle|s_{0}=s\right] \\
&= \sum_{s'\in\mathcal{S}}\mathcal{P}_{ss'}^{\pi(s)}\left[\mathcal{R}_{ss'}^{\pi(s)}+\gamma V^{\pi}(s')\right]. \qquad (2.6)
\end{aligned}
$$

Notice the difference between Equations 2.5 and 2.6. The former is the Bellman optimality equation, which is independent of the policy, while the latter is the recursive form of the value function given a fixed policy. Since $\mathcal{S}$ is assumed to be finite, the state values can be calculated by solving $|\mathcal{S}|$ linear equations each specified by writing Equation 2.6 for every state of the MDP. The solution for a finite state MDP with $\mathcal{S}=\{s_1,s_2,...,s_{|\mathcal{S}|}\}$ for which the vector ${\boldsymbol{V}^{\pi}}_{|\mathcal{S}|\times 1}$ represents the value function of the policy π, can be calculated in closed form (our notation may exclude the π superscript for brevity, yet the policy dependency is always assumed). To write Equation 2.6 in the matrix form, let matrix $\boldsymbol{P}$ be defined using $\boldsymbol{P}_{ij}=\mathcal{P}_{s_is_j}^{\pi(s_i)}$, and let vector $\boldsymbol{R}$ be defined using $\boldsymbol{R}_i=\sum_j \mathcal{P}_{s_is_j}^{\pi(s_i)}\mathcal{R}_{s_is_j}^{\pi(s_i)}$. Then Equation 2.6 takes the form:

$$\boldsymbol{V} \;=\; \boldsymbol{R}+\gamma\boldsymbol{P}\boldsymbol{V}.$$

Let us define

$$\mathbf{T}(\boldsymbol{V}) \;\triangleq\; \boldsymbol{R}+\gamma\boldsymbol{P}\boldsymbol{V}, \qquad (2.7)$$

where $\mathbf{T}$ is known as the Bellman operator applied to the value function. The output of $\mathbf{T}$ is a vector with the same size as the input vector (*i.e.,* $\mathbf{T}:\mathbb{R}^{|\mathcal{S}|}\rightarrow\mathbb{R}^{|\mathcal{S}|}$). With the help of operator $\mathbf{T}$, we can write Equation 2.6 as $\boldsymbol{V}=\mathbf{T}(\boldsymbol{V})$. Thus, the problem of evaluating policy π translates into finding the fixed-point of operator $\mathbf{T}$. Solving this equation for $\boldsymbol{V}$ yields:

$$\boldsymbol{V} \;=\; (\boldsymbol{I}-\gamma\boldsymbol{P})^{-1}\boldsymbol{R}, \qquad (2.8)$$

where $\boldsymbol{I}$ is the identity matrix. Expanding the right hand side using the Neumann series [see *e.g.,* Barto and Duff, 1994], we obtain:

$$(\boldsymbol{I} - \gamma \boldsymbol{P})^{-1} = \sum_{k=0}^{\infty} (\gamma \boldsymbol{P})^k.$$

Note that the series converges, hence the inverse exists, because $\|\boldsymbol{P}\|_\infty \leq 1$, where $\|\boldsymbol{P}\|_\infty = \max_i \sum_j |\boldsymbol{P}_{ij}|$ is the ℓ_∞ operator norm of matrix $\boldsymbol{P}$.

As for policy improvement, the new policy is updated by selecting the action that is "greedy" with respect to the calculated values:

$$\pi(s) = \operatorname*{argmax}_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} \mathcal{P}^a_{ss'} \left[\mathcal{R}^a_{ss'} + \gamma V^\pi(s') \right]. \tag{2.9}$$

Notice the V^π values on the right hand side are the calculated ones from the policy evaluation step. Moreover, ties are broken in a systematic fashion, such as uniform random selection amongst the best available actions. Putting Equations 2.8 and 2.9 together, we arrive at the *policy iteration* algorithm (see Algorithm 1) [Howard, 1960].

While policy iteration is guaranteed to stop in a polynomial number of iterations in $|S| \times |A|$ for a fixed value of gamma [Ye, 2011] and reach the optimal solution [Howard, 1960], from a practical standpoint, this algorithm is not scalable because storing $\boldsymbol{P}$ requires $\mathcal{O}(|\mathcal{S}|^2)$ memory and solving Equation 2.8 takes $\mathcal{O}(|\mathcal{S}|^3)$ time.[5] Hence the exact policy evaluation step is often done iteratively within a given threshold η, as is done in Algorithm 1 (akin to Figure 4.3 of Sutton and Barto [1998]). Lines 1-2 initialize the policy and its corresponding value function. Without any prior domain knowledge, the policy can be selected to be uniformly random among possible actions, while state values are set initially to zero. There are other initialization schemes possible, such as *optimistic initialization*, where all values are set to (or near) the maximum value [see *e.g.,* Szita and Szepesvári, 2010]. Lines 6-11 solve Equation 2.8 approximately with complexity $\mathcal{O}(N|\mathcal{S}|^2)$, where N is the number of sweeps through the states (*i.e.,* lines 7-10). Smaller values of η provide better approximation at the cost of more computation (*i.e.,* larger values of N). Note that Algorithm 1 does not store $\boldsymbol{P}$ explicitly to avoid $\mathcal{O}(|\mathcal{S}|^2)$

[5] There are advanced techniques for matrix inversion with $\mathcal{O}(|\mathcal{S}|^{2.373})$ computational complexity, but most open-source math packages use common methods [*e.g.,* Golub and Loan, 1996] for matrix decomposition, amounting to $\mathcal{O}(|\mathcal{S}|^3)$ complexity.

Algorithm 1: Policy Iteration — Complexity

Input: MDP, η
Output: π

1 $\pi(s) \leftarrow$ Initialize arbitrary **for** $s \in \mathcal{S}$
2 $V^{\pi}(s) \leftarrow$ Initialize arbitrarily **for** $s \in \mathcal{S}$
3 `changed` $\leftarrow$ **True**
4 **while** `changed` **do**
5 $\Delta \leftarrow 0$
6 **repeat**
7 **for** $s \in S$ **do**
8 $v \leftarrow V^{\pi}(s)$
9 $V^{\pi}(s) \leftarrow \sum_{s'} \mathcal{P}_{ss'}^{\pi(s)}(\mathcal{R}_{ss'}^{\pi(s)} + \gamma V^{\pi}(s'))$ — $\mathcal{O}(|\mathcal{S}|)$
10 $\Delta \leftarrow \max(\Delta, |v - V^{\pi}(s)|)$
11 **until** $\Delta < \eta$
12 **for** $s \in \mathcal{S}$ **do**
13 $\pi^{+}(s) \leftarrow \operatorname{argmax}_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^{a} [\mathcal{R}_{ss'}^{a} + \gamma V^{\pi}(s')]$ — $\mathcal{O}(|\mathcal{A}||\mathcal{S}|)$
14 `changed` $\leftarrow (\pi^{+} \neq \pi)$
15 $\pi \leftarrow \pi^{+}$
16 return π

memory requirement. Instead lines 9 and 13 access transition probabilities on demand and store the value and the policy for each state, incurring $\mathcal{O}(|\mathcal{S}|)$ memory. Line 13 can be executed in memory size independent of $\mathcal{O}(|\mathcal{A}|)$ as maximization can be done incrementally. The algorithm also shows the computational complexity of selected lines for each iteration on the right side. Algorithm 1 requires $\mathcal{O}(|\mathcal{S}|)$ memory[6] and $\mathcal{O}((N + |\mathcal{A}|)|\mathcal{S}|^2)$ computation per iteration.

The loop shown in Figure 2.2 still converges to the optimal solution if policy improvement is executed before accurately evaluating the policy, as long as the value function gets closer to its optimal value after each policy evaluation step [Howard, 1960, Sutton and Barto, 1998]. This idea motivates a process that can potentially save substantial computation while still finding good policies: update the policy after every single "Bellman backup" (line 9 of Algorithm 1). Thus, the Bellman backup should use the best possible

[6] It is assumed that the transition probabilities and the rewards are not stored explicitly. Rather they are stored in a functional form that can be probed for arbitrary inputs.

Algorithm 2: Value Iteration — Complexity

Input: MDP, η
Output: π

$V(s) \leftarrow$ Initialize arbitrarily **for** $s \in \mathcal{S}$
repeat
  for $s \in \mathcal{S}$ **do**
    $v \leftarrow V(s)$
    $V(s) \leftarrow \max_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} \mathcal{P}^a_{ss'} \left[\mathcal{R}^a_{ss'} + \gamma V(s')\right]$ — $\mathcal{O}(|\mathcal{A}||\mathcal{S}|)$
    $\pi(s) \leftarrow \operatorname{argmax}_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} \mathcal{P}^a_{ss'} \left[\mathcal{R}^a_{ss'} + \gamma V(s')\right]$ — $\mathcal{O}(|\mathcal{A}||\mathcal{S}|)$
    $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \eta$
return π

action for backup rather than a fixed policy [Bellman, 1957]:

$$\forall s \in \mathcal{S}, V(s) \leftarrow \max_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} \mathcal{P}^a_{ss'} \left[\mathcal{R}^a_{ss'} + \gamma V(s')\right]. \tag{2.10}$$

Equation 2.10 implicitly unifies Equations 2.9 and 2.6, giving the *value iteration* method shown in Algorithm 2 (akin to Figure 4.5 of Sutton and Barto [1998]). Bertsekas [1976] has shown that if the ℓ_∞ difference of the value function between two successive iterations of value iteration (the Bellman residual) is less than η, then the ℓ_∞ difference between the value functions of the greedy policy with respect to the current values and the optimal policy is no more than $2\eta\gamma/(1-\gamma)$. Value iteration improves the policy much more frequently than policy iteration and reduces the main loop complexity from $\mathcal{O}((N+|\mathcal{A}|)|\mathcal{S}|^2)$ to $\mathcal{O}(|\mathcal{A}||\mathcal{S}|^2)$. The memory requirement of value iteration is the same as policy iteration (*i.e.,* $\mathcal{O}(|\mathcal{S}|)$). While in theory value iteration may require more iterations than policy iteration [Ye, 2011], in practice, value iteration often requires less total time to find the optimal solution compared to policy iteration; a finding that will be verified in Section 4.

2.4 Approximate Dynamic Programming

With the aim of scalability, we now describe ways to reduce the memory and computational complexities of the algorithms above. Attention is focused on MDPs with large, yet finite, state spaces with a small number of actions (*i.e.,* $|\mathcal{A}| \ll |\mathcal{S}|$). This assumption is often met in practice. For example in

9×9 Go, $|\mathcal{S}| = 10^{38}$ and $|\mathcal{A}| = 81$. Hence the rest of Section 2.4 focuses on eliminating $|\mathcal{S}|$ dependent memory sizes and computations (*i.e.,* scaling obstacles). These changes include moving away from the tabular representation where a unique value is stored for every state-action pair to a more compact representation. Looking back at Algorithm 2, there are four scaling problems:

1. π stores an action for each state (line 6).
2. There is a loop over all possible states (line 3).
3. Both the Bellman backup (line 5) and the policy update (line 6) consider all possible next states, which in the worst case can be $|\mathcal{S}|$.
4. V stores a unique parameter for every state of the MDP (line 5).

2.4.1 Solving Problem 1

The first problem can be solved by storing the policy implicitly through the use of action-value functions. If for each state s, an action-value $Q(s, a)$ is available for all actions, then the greedy policy can be retrieved simply by:

$$\pi(s) = \underset{a \in \mathcal{A}}{\operatorname{argmax}}\, Q(s, a), \tag{2.11}$$

which is also known as the *greedy* policy with respect to the action-value function Q. This change will eliminate the need for storing policies explicitly. Note that switching from V to Q increases the memory requirement for storing the value function by factor of $|\mathcal{A}|$. Yet, as long as the dependence of the value function on the state space (problem 4) is removed, this increase is not a major concern because $|\mathcal{A}|$ is assumed to be small.

2.4.2 Solving Problem 2

For MDPs with large state spaces, sweeping through all states and performing Bellman backups on each one is infeasible. Consequently, more advanced techniques have been proposed to focus Bellman backups in parts of the state-space that are more promising [Singh, 1992, Moore and Atkeson, 1993, Barto et al., 1995, Kuvayev and Sutton, 1996]. Currently, the best comprehensive source for these techniques is by Mausam and Kolobov [2012]. In this paper, we focus on the work of Barto et al. [1995] in which they proposed sampling

trajectories $\langle s_0, a_0, r_0, s_1, a_1 ... \rangle$ based on promising policies while performing Bellman backups on visited states. The core idea of this *trajectory-based sampling* is to execute Bellman backups on states that are visited under good policies. In practice, a form of exploration is required in producing these trajectories to ensure all states are visited infinitely often in the limit of infinite samples, so that Equation 2.6 holds for all states asymptotically. Here, we adopt a simple but common approach to generate trajectories named ϵ-greedy policy, which selects an action randomly with a small probability ϵ every time, and acts greedily with respect to the Q function otherwise [Sutton and Barto, 1998]:

$$\pi^{\epsilon}(s) \triangleq \begin{cases} \operatorname{argmax}_a Q^{\pi}(s,a), & \text{with probability } 1-\epsilon; \\ \texttt{UniformRandom}(\mathcal{A}), & \text{with probability } \epsilon. \end{cases} \tag{2.12}$$

2.4.3 Solving Problem 3

In most practical problems, there is a *locality* property meaning that given each state-action pair, the number of reachable next states is much smaller than the total number of states:

$$|\{s' | \mathcal{P}^a_{ss'} \neq 0\}| \ll |\mathcal{S}| \quad s \in \mathcal{S}, a \in \mathcal{A}.$$

This assumption naturally alleviates Problem 3. For problems where this assumption does not hold, L_1 samples can be used to approximate the expectations on the next state. For example, line 5 of Algorithm 2 can be changed to:

$$V(s) \leftarrow \max_a \frac{1}{L_1} \sum_{j=1}^{L_1} \left[\mathcal{R}^a_{ss'_j} + \gamma V(s'_j) \right], s'_j \sim \mathcal{P}^{\pi(s)}_s.$$

Notice that as $L_1 \to \infty$, the estimate becomes exact with probability one.

2.4.4 Solving Problem 4

Addressing the fourth problem is more challenging than the previous 3 problems. In general, holding a tabular representation of the value function $V(s)$, or its state-action version, $Q(s,a)$ (*i.e.,* storing a unique value for each state-action pair) is impractical for large state spaces. Even domains that are fairly simple to describe, like a multi-robot control scenario with 30 robots that each

can either be "busy" or "free" will require 2^{30} state values to be stored. Since it was shown that the Q function holds sufficient information for deriving a policy without needing extra $\mathcal{O}(|\mathcal{S}|)$ storage, what is needed is an approximate representation of $Q(s, a)$ whose size is not polynomial in $|\mathcal{S}|$. One such approach that has been successful in a number of RL domains is to approximate $Q(s, a)$ using a parametric function, where the savings comes from the fact that the number of parameters to store and update is less than the number of states [Sutton, 1996, Silver et al., 2012, Geramifard et al., 2011]. For instance, one could attempt to approximate the value function in the aforementioned multi-robot control domain by using a weighted combination of the "busy" and "free" state variables of each robot amounting to $30 \times 2 = 60$ parameters (*i.e., features*). This representation might have lower fidelity to the true value function in that it may not be able to capture the expected return of specific combinations of the robot states. Often, the parametric form used is linear in the parameters:

$$Q^{\pi}(s, a) \quad = \quad \phi(s, a)^{\top} \boldsymbol{\theta}, \tag{2.13}$$

with vectors $\phi(s, a)$ and $\boldsymbol{\theta}$ are defined as follows: Each element of the vector $\phi(s, a)$ is called a *feature*; $\phi_i(s, a)$ denotes the value of feature i for state-action pair (s, a). The feature function $\phi : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}^n$ maps each state-action pair to a vector of feature values; $\boldsymbol{\theta} \in \mathbb{R}^n$ is the weight vector specifying the contribution of each feature across all state-action pairs. Finding the right feature function (ϕ) is an important step, but for the rest of this chapter, it is assumed that ϕ is given. Various ways to build such functions are discussed in Chapter 3.

As defined in Equation 2.11, finding the optimal policy requires the correct ranking of Q values in a given state. Practitioners often avoid approximating the value of one state-action pair based on the value of other actions taken in the same state [Sutton, 1996, Lagoudakis and Parr, 2003, Buşoniu et al., 2010]. In other words, features used to approximate $Q(s, a)$ are different from the features used to approximate $Q(s, a')$, $\forall a \in \mathcal{A}, a' \neq a$. Given a state-action pair (s, a), this constraint is met by mapping s to a vector of feature values ($\phi : \mathcal{S} \rightarrow \mathbb{R}^m$), and then using this feature vector in the corresponding slot for action a while setting the feature values for the rest of the actions to zero. While this approach is impractical for domains with a large number of actions, it works well for domains with small $|\mathcal{A}|$, which are the

focus of this paper. Furthermore, based on our assumption of $|\mathcal{A}| \ll |\mathcal{S}|$, $m|\mathcal{A}| = n$ is always maintainable in the memory. Notice that $\boldsymbol{\theta}$ holds m separate weights for each action, for a total of $m|\mathcal{A}| = n$ values.

The following example shows this mechanism for an MDP with 2 actions and 3 features per action. Hence $3 \times 2 = 6$ features are used for linear function approximation.

$$\phi(s) = \begin{bmatrix} \phi_1(s) \\ \phi_2(s) \\ \phi_3(s) \end{bmatrix} \Rightarrow \phi(s, a_1) = \begin{bmatrix} \phi_1(s) \\ \phi_2(s) \\ \phi_3(s) \\ 0 \\ 0 \\ 0 \end{bmatrix}, \quad \phi(s, a_2) = \begin{bmatrix} 0 \\ 0 \\ 0 \\ \phi_1(s) \\ \phi_2(s) \\ \phi_3(s) \end{bmatrix} \tag{2.14}$$

For our empirical results, $\phi(s, a)$ is assumed to be generated from $\phi(s)$ following the above process, though a more general form of $\phi(s, a)$ could also be used.

2.5 Trajectory Based Value Iteration

We now have all the building blocks to introduce a scalable DP technique with memory and iteration computation independent of $|\mathcal{S}|$. However there is a caveat in putting all these blocks together. Looking back at Equation 2.10, for every state, the new value is calculated using the Bellman backups and stored in a separate location. Storing the value cannot be applied directly to the linear function approximation case because changing the weight corresponding to each feature can potentially change the value of multiple states.[7] So the question becomes, given a new target value, $Q^+(s, a)$, and our current approximate, $Q(s, a) = \phi(s, a)^\top \boldsymbol{\theta}$, how can $\boldsymbol{\theta}$ change to get $Q(s, a)$ closer to $Q^+(s, a)$? The answer is to move $\boldsymbol{\theta}$ in the opposite direction of the squared difference between $Q(s, a)$ and $Q^+(s, a)$:

$$\begin{aligned} C &\triangleq (Q^+(s, a) - Q(s, a))^2 \\ &= (Q^+(s, a) - \phi(s, a)^\top \boldsymbol{\theta})^2 \\ \frac{\partial C}{\partial \boldsymbol{\theta}} &= -2\phi(s, a)(Q^+(s, a) - \phi(s, a)^\top \boldsymbol{\theta}). \end{aligned}$$

[7] In the degenerate case where bases form the tabular representation, values can be stored directly.

Hence, the update rule takes the following form:

$$\begin{aligned}\boldsymbol{\theta} &\leftarrow \boldsymbol{\theta} - \beta\frac{\partial C}{\partial \boldsymbol{\theta}} \\ &\leftarrow \boldsymbol{\theta} + \alpha\phi(s,a)(Q^{+}(s,a) - Q(s,a)),\end{aligned}$$

where β is the step size parameter and $\alpha = 2\beta$. Combining the above gradient approach and the solutions to problems 1–4 with value iteration result in an algorithm we term *trajectory based value iteration* (TBVI) shown in Algorithm 3. TBVI can be viewed as an extension of the algorithm mentioned in Section 9.6 of Sutton and Barto [1998] and the Real-Time dynamic programming algorithm [Barto et al., 1995] to the linear function approximation setting. Problem 1 is addressed in line 3 by calculating the policy using Equation 2.11. Problem 2 is resolved in line 3 where samples are generated by following the ϵ-greedy policy instead of sweeping through the whole state space. Problem 3 is addressed in lines 4 and 5 by using a sampling technique to approximate the expectation. Problem 4 is addressed by using linear function approximation, specifically by calculating $Q(\cdot,\cdot)$ as $\phi(\cdot,\cdot)^{\top}\boldsymbol{\theta}$. An interesting observation on the TBVI algorithm is that if $L_1 = 1$, then instead of executing line 4, the next state and reward along the trajectory can be used to calculate the estimation on line 5. This special case is discussed further in Section 2.7.1. TBVI has an $\mathcal{O}(n + L_1)$ memory requirement, and $\mathcal{O}(nTL_1|\mathcal{A}|)$ iteration complexity, where T is the maximum length of a trajectory. One can follow the same steps to create a trajectory based policy iteration algorithm from Algorithm 1, but finding a reasonable condition to switch from policy evaluation to policy improvement is challenging because not all states are visited on each iteration.

Compared to Algorithm 2, the stopping condition (line 2) is defined more loosely based on a fixed planning time. In practice, more conditions can be employed to force the algorithm to exit the loop on line 2 earlier than the allocated planning horizon. For example, if through a certain number of consecutive trajectories the maximum δ value is less than a certain threshold, the algorithm can exit. As for the end result, TBVI can diverge with linear function approximation for reasons similar to those that cause the divergence of non-trajectory based dynamic programming methods with function approximation [see Tsitsiklis and Roy, 1997]. While this divergence was observed in some of the empirical domains discussed later, the algorithm is successful in

Algorithm 3: Trajectory Based Value Iteration (TBVI) — Complexity

Input: MDP, α, L_1
Output: π

$\boldsymbol{\theta} \leftarrow$ Initialize arbitrarily
while *time left* **do**
  for $\langle s, a \rangle$ *in a trajectory following* π^ϵ **do**
    Create L_1 samples: $s'_j \sim \mathcal{P}^a_{s\cdot}, j = 1, ..., L_1$
    $Q^+(s,a) \leftarrow \frac{1}{L_1} \sum_{j=1}^{L_1} \mathcal{R}^a_{ss'_j} + \gamma \max_{a'} Q(s'_j, a')$, — $\mathcal{O}(nL_1|\mathcal{A}|)$
    $\delta \leftarrow Q^+(s,a) - Q(s,a)$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha\delta\phi(s,a)$ — $\mathcal{O}(n)$
return π greedy with respect to Q

others, and is often a much faster alternative to full Value Iteration.

2.6 Approximate Dynamic Programming in Matrix Format

Before moving on, let us review the path we have followed in deriving the first three algorithms (See Figure 2.1, boxes 1-3). The main idea was to evaluate/improve the policy in a loop (Figure 2.2). We started policy evaluation using Equation 2.8. Due to the cost of inverting $\boldsymbol{P}$ when $\boldsymbol{P}$ is large, we changed from the matrix form of policy evaluation to per-state policy evaluation and introduced both policy iteration and value iteration. Finally by eliminating all memory and computations with $\mathcal{O}(|\mathcal{S}|)$ complexity, we introduced TBVI. In this section, we take a different path (as shown on the right-side path in Figure 2.1) from the policy evaluation/improvement loop by investigating the benefits of using linear function approximation for more compact memory and computation in evaluating the policy (Equation 2.8). First we derive a policy evaluation technique to estimate V values using linear function approximation. Then we eliminate all $\mathcal{O}(|\mathcal{S}|)$ dependent memory and computational requirements and introduce a new algorithm.

Similar to Equation 2.13,

$$V(s) \quad = \quad \phi(s)^\top \boldsymbol{\theta}.$$

Note that for ease of readability, $\boldsymbol{\theta}$ is used to indicate the weight vector for both action-value functions and state-value functions. The only difference is

that the parameter vector, $\boldsymbol{\theta}$, will be of dimension m when approximating state-value functions, and n when approximating action-value functions. Define $\tilde{\boldsymbol{V}}_{\boldsymbol{\theta}}$ as an approximation of $\boldsymbol{V}$:

$$\tilde{\boldsymbol{V}}_{\boldsymbol{\theta}} = \begin{bmatrix} \text{—} \ \phi^\top(s_1) \ \text{—} \\ \text{—} \ \phi^\top(s_2) \ \text{—} \\ \vdots \\ \text{—} \ \phi^\top(s_{|\mathcal{S}|}) \ \text{—} \end{bmatrix} \times \begin{bmatrix} \theta_1 \\ \theta_2 \\ \vdots \\ \theta_m \end{bmatrix} \triangleq \boldsymbol{\Phi}_{|\mathcal{S}| \times m} \boldsymbol{\theta}_{m \times 1}. \tag{2.15}$$

For brevity, $\tilde{\boldsymbol{V}}$ will be used instead of $\tilde{\boldsymbol{V}}_{\boldsymbol{\theta}}$. Often the true value function does not lie in the space spanned by the basis functions (*i.e.,* column space of $\boldsymbol{\Phi}$). Hence a metric is required to define the best approximate value function in the span of $\boldsymbol{\Phi}$. Ideally, the goal would be to find the solution to the following minimization problem:

$$\min_{\boldsymbol{\theta}} ||\boldsymbol{V} - \tilde{\boldsymbol{V}}||^2_{\mathbf{d}} \quad = \quad \min_{\boldsymbol{\theta}} \sum_{i \in \{1, \cdots, |\mathcal{S}|\}} \left[\boldsymbol{V}_i - \tilde{\boldsymbol{V}}_i\right]^2 \mathbf{d}_i, \tag{2.16}$$

where $\mathbf{X}_i$ is the i^{th} element of vector $\mathbf{X}$ and $\mathbf{d}$ is a non-negative weight vector specifying the importance of each state. Intuitively states that are visited more often should have higher weights, penalizing the error correspondingly. One way to capture this intuition is to use the *steady state probability distribution* defined for any fixed policy π with transition $\boldsymbol{P}$ under that policy as a vector $\mathbf{d}_{1 \times |\mathcal{S}|}$, where

$$\begin{aligned} & \mathbf{d}\boldsymbol{P} = \mathbf{d} \\ \text{s.t.} \qquad & \sum_i \mathbf{d}_i = 1, \\ & \forall i \in \{1, \cdots, |\mathcal{S}|\}, \mathbf{d}_i \geq 0, \end{aligned}$$

where $\mathbf{d}_i$ indicates the probability of being at state i in the limit of following the fixed policy. Calculating the steady state distribution can be challenging, hence Section 2.6.3 will use a more practical weighting scheme.

Equation 2.16 defines an unconstrained quadratic optimization problem that has an analytic solution of the form: :

$$\begin{aligned} \tilde{\boldsymbol{V}} &= \boldsymbol{\Pi}\boldsymbol{V} \\ \boldsymbol{\Pi} &= \boldsymbol{\Phi}(\boldsymbol{\Phi}^\top \boldsymbol{D} \boldsymbol{\Phi})^{-1} \boldsymbol{\Phi}^\top \boldsymbol{D}, \end{aligned} \tag{2.17}$$

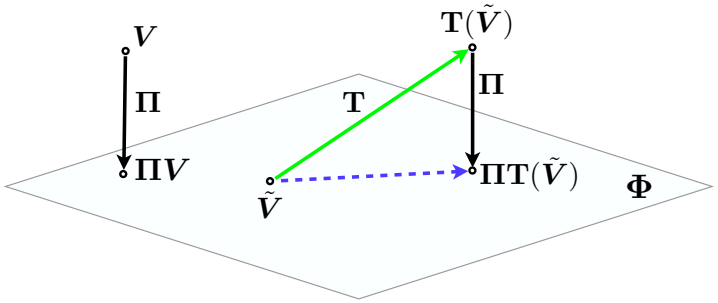

Figure 2.3: A geometric interpretation of what Bellman Residual Minimization (solid green) and Projected Bellman Residual Minimization (dashed blue) minimize [akin to Lagoudakis and Parr, 2003]. $\mathbf{\Pi}$ is the projection operator based on the steady state distribution $\mathbf{d}$, mapping every point to its orthogonal projection on the span of $\mathbf{\Phi}$ shown as a 2D plane. $\mathbf{T}$ is the Bellman operator. The ideal approximation is shown by $\mathbf{\Pi}\boldsymbol{V}$, yet it is impractical since $\boldsymbol{V}$ is not known.

where $\boldsymbol{D}_{|\mathcal{S}|\times|\mathcal{S}|}$ is defined as a matrix, with $\mathbf{d}$ on its diagonal ($\boldsymbol{D} = diag(\mathbf{d})$). Because $\boldsymbol{V}$ is not known, some other technique is needed to identify the weights that best approximate the value function of a policy. Recall from Section 2.3 that one can compute the value function of a policy using dynamic programming. So approximate dynamic programming can be used to compute the weights (and therefore the approximate value function) of the policy in the same way. However, this raises a question that is best illustrated geometrically.

Figure 2.3 shows the value function ($\boldsymbol{V}$) and its projection into the span of $\mathbf{\Phi}$ (shown as the 2D plane), using the orthogonal projection operator, $\mathbf{\Pi}$. When performing approximate DP to compute $\tilde{\boldsymbol{V}}$, the Bellman operator (Equation 2.7) is used to improve the approximation, but this operator can move the approximation out of the span of $\mathbf{\Phi}$. Hence the new approximation has to be projected back to the span of $\mathbf{\Phi}$ using $\mathbf{\Pi}$. There are two major metrics used in the literature to define the best approximated value function: 1) the projected Bellman error (blue dashed line in Figure 2.3) [Bradtke and Barto, 1996, Lagoudakis and Parr, 2003, Farahmand et al., 2008, Sutton et al., 2009, Scherrer, 2010] and 2) the Bellman error (green solid line in Figure 2.3). Correspondingly, there are two methods for solving for the best approximation to the true value function, based on each metric: 1) Projected

Bellman Residual Minimization, also known as the least-squares Temporal Difference solution (LSTD) [Bradtke and Barto, 1996], and 2) Bellman residual minimization (BRM) [Schweitzer and Seidman, 1985]. Both methods attempt to converge to a fixed point, with the former performing its operations directly in the projected linear space. At the fixed point, the projected Bellman error (*i.e.,* the length of the blue line in Figure 2.3) will be zero. This tutorial focuses on the LSTD solution (derived next) as it usually has better practical results [Scherrer, 2010]. For more information regarding the comparison of LSTD and BRM approaches refer to the work of Lagoudakis and Parr [2003] and Scherrer [2010].

2.6.1 Projected Bellman Residual Minimization

Our goal is to find the approximation that minimizes the norm of the blue dashed line in Figure 2.3 formulated as:

$$\min_{\boldsymbol{\theta}} \|\boldsymbol{\Pi}\mathbf{T}(\tilde{\boldsymbol{V}}) - \tilde{\boldsymbol{V}}\|^2.$$

The minimizer $\boldsymbol{\theta}$ can be found by forcing the two points $\boldsymbol{\Pi}\mathbf{T}(\tilde{\boldsymbol{V}})$ and $\tilde{\boldsymbol{V}}$ to be equal:

$$\begin{aligned}
\tilde{\boldsymbol{V}} &= \boldsymbol{\Pi}\mathbf{T}(\tilde{\boldsymbol{V}}) \\
\boldsymbol{\Phi}\boldsymbol{\theta} &= \boldsymbol{\Phi}(\boldsymbol{\Phi}^\top \boldsymbol{D}\boldsymbol{\Phi})^{-1}\boldsymbol{\Phi}^\top \boldsymbol{D}(\boldsymbol{R} + \gamma \boldsymbol{P}\boldsymbol{\Phi}\boldsymbol{\theta}) \\
\boldsymbol{\theta} &= (\boldsymbol{\Phi}^\top \boldsymbol{D}\boldsymbol{\Phi})^{-1}\boldsymbol{\Phi}^\top \boldsymbol{D}(\boldsymbol{R} + \gamma \boldsymbol{P}\boldsymbol{\Phi}\boldsymbol{\theta}) \\
\Big(\boldsymbol{\Phi}^\top \boldsymbol{D}(\boldsymbol{\Phi} - \gamma \boldsymbol{P}\boldsymbol{\Phi})\Big)\boldsymbol{\theta} &= \boldsymbol{\Phi}^\top \boldsymbol{D}\boldsymbol{R} \\
\boldsymbol{\theta} &= \underbrace{[\boldsymbol{\Phi}^\top \boldsymbol{D}(\boldsymbol{\Phi} - \gamma \boldsymbol{P}\boldsymbol{\Phi})]}_{\boldsymbol{A}}{}^{-1} \underbrace{\boldsymbol{\Phi}^\top \boldsymbol{D}\boldsymbol{R}}_{\boldsymbol{b}} \qquad (2.18) \\
&= \boldsymbol{A}^{-1}\boldsymbol{b}. \qquad (2.19)
\end{aligned}$$

Notice that $\boldsymbol{A}$ has to be invertible for the solution to exist. The $\tilde{\boldsymbol{V}}$ calculated above is known as the LSTD solution and can be far from $\boldsymbol{V}$. Yet it has been shown that under certain conditions the approximation error is bounded [Tsitsiklis and Roy, 1999]:

$$\|\boldsymbol{V} - \tilde{\boldsymbol{V}}\| \leq \frac{1}{\sqrt{1-\gamma^2}}\|\boldsymbol{V} - \boldsymbol{\Pi}\boldsymbol{V}\|.$$

This bound has been improved by Yu and Bertsekas [2010], but the discussion of their result is beyond the scope of this tutorial. In practice, when $\boldsymbol{A}$ is rank deficient, regularization techniques are used [Kolter and Ng, 2009]. Notice that in the tabular case where $\boldsymbol{\Phi} = \boldsymbol{D} = \boldsymbol{I}$, that is, features simply indicate which state the system is in, Equation 2.8 is retrieved:

$$\boldsymbol{\theta} \quad = \quad (\boldsymbol{I} - \gamma \boldsymbol{P})^{-1} \boldsymbol{R}.$$

2.6.2 State-Action Value Estimation using LSTD

The goal is to find optimal policies for MDPs. Hence the policy evaluation/improvement loop shown in Figure 2.2 has to be formed. LSTD can be integrated with Equation 2.9 to form such a loop, but this approach hinders scalability as π needs $\mathcal{O}(|\mathcal{S}|)$ memory (Section 2.4). To address this problem, Lagoudakis and Parr [2003] suggested to approximate action-values instead of state-values. Hence the LSTD solution is rederived to calculate Q values instead of V values. Similar to the derivation of Equation 2.6, Q^π can be written recursively as:

$$Q^\pi(s,a) = \sum_{s' \in \mathcal{S}} \mathcal{P}^a_{ss'} \left[\mathcal{R}^a_{ss'} + \gamma Q^\pi \left(s', \pi(s') \right) \right]. \tag{2.20}$$

The π notation will be dropped for the rest of the derivation, as the policy is assumed to be fixed. Now write the above equation in a matrix form to rederive LSTD. Notice that we overload our notation, such that $\boldsymbol{\Phi}, \boldsymbol{\theta}, \boldsymbol{P}, \boldsymbol{R}, \mathbf{d}$ hold extra information required to calculate Q rather than V:

$$\boldsymbol{Q} = \boldsymbol{R} + \gamma \boldsymbol{P} \boldsymbol{Q},$$

where,

$$\boldsymbol{Q}_{|\mathcal{S}||\mathcal{A}| \times 1} \quad = \quad \begin{bmatrix} Q(s_1, a_1) \\ \vdots \\ Q(s_{|\mathcal{S}|}, a_1) \\ \vdots \\ Q(s_{|\mathcal{S}|}, a_{|\mathcal{A}|}) \end{bmatrix}.$$

The $\boldsymbol{Q}$ vector is approximated by $\tilde{\boldsymbol{Q}} = \boldsymbol{\Phi\theta}, \boldsymbol{\theta} \in \mathbb{R}^n$, with

$$\boldsymbol{\Phi}_{|\mathcal{S}||\mathcal{A}|\times n} = \begin{bmatrix} \text{—} \ \phi(s_1, a_1)^\top \ \text{—} \\ \vdots \\ \text{—} \ \phi(s_{|\mathcal{S}|}, a_1)^\top \ \text{—} \\ \vdots \\ \text{—} \ \phi(s_{|\mathcal{S}|}, a_{|\mathcal{A}|})^\top \ \text{—} \end{bmatrix}, \boldsymbol{R}_{|\mathcal{S}||\mathcal{A}|\times 1} = \begin{bmatrix} \mathcal{R}_{s_1}^{a_1} \\ \vdots \\ \mathcal{R}_{s_{|\mathcal{S}|}}^{a_1} \\ \vdots \\ \mathcal{R}_{s_{|\mathcal{S}|}}^{a_{|\mathcal{A}|}} \end{bmatrix},$$

where $\phi(s, a)$ can be built from $\phi(s)$ as described in Section 2.4.4. Similarly the transition matrix and the reward vector are defined:

$$\boldsymbol{P}_{|\mathcal{S}||\mathcal{A}|\times|\mathcal{S}||\mathcal{A}|} = \begin{bmatrix} \mathcal{P}(s_1, a_1, s_1, a_1), \mathcal{P}(s_1, a_1, s_2, a_1) & \cdots & \mathcal{P}(s_1, a_1, s_{|\mathcal{S}|}, a_{|\mathcal{A}|}) \\ \vdots & \ddots & \vdots \\ \mathcal{P}(s_{|\mathcal{S}|}, a_{|\mathcal{A}|}, s_1, a_1), \mathcal{P}(s_{|\mathcal{S}|}, a_{|\mathcal{A}|}, s_2, a_1) & \cdots & \mathcal{P}(s_{|\mathcal{S}|}, a_{|\mathcal{A}|}, s_{|\mathcal{S}|}, a_{|\mathcal{A}|}) \end{bmatrix}$$

$$\begin{aligned} \mathcal{R}_s^a &= \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a \mathcal{R}_{ss'}^a \\ \mathcal{P}(s_i, a_j, s_k, a_l) &= \begin{cases} \mathcal{P}_{s_i s_k}^{a_j} & \text{if } a_l = \pi(s_k); \\ 0, & \text{otherwise.} \end{cases} \end{aligned}$$

In the tabular case:

$$\boldsymbol{Q} = (\boldsymbol{I} - \gamma \boldsymbol{P})^{-1} \boldsymbol{R}.$$

By overloading the weighting distribution to include state-action pairs instead of states, $\boldsymbol{D}$ becomes $|\mathcal{S}||\mathcal{A}| \times |\mathcal{S}||\mathcal{A}|$, with overloaded $\mathbf{d}_{1\times|\mathcal{S}||A|}$ on its diagonal. Note that each element of $\mathbf{d}$ highlights the importance of a state-action pair instead of a state. Consequently all derivations of LSTD (*i.e.,* Equations 2.18 and 2.19) remain intact by calculating $\tilde{\boldsymbol{Q}}$ instead of $\tilde{\boldsymbol{V}}$.

2.6.3 Trajectory Based Policy Iteration

The previously derived LSTD solution can be integrated with Equation 2.11 to form a policy iteration technique. Reviewing Equations 2.18 and 2.19 in the state-action (Q) case, the resulting algorithm will not be practical because $\boldsymbol{P}, \boldsymbol{R}, \boldsymbol{D}, \boldsymbol{\Phi}$ have $|\mathcal{S}||\mathcal{A}|$ rows.

Given the fixed policy π, samples can be gathered by following π, with each sample coming in the form of $\langle s_i, a_i \rangle, i \in \{1, \cdots, L_2\}$, where s_1 is the initial state, $a_i = \pi(s_i)$, and $s_{i+1} \sim \mathcal{P}_{s_i}^{a_i}$. Hence,

$$\widetilde{\boldsymbol{P}\boldsymbol{\Phi}}_{L_2 \times n} = \begin{bmatrix} \text{—} \, \varphi(s_1, a_1)^\top \, \text{—} \\ \text{—} \, \varphi(s_2, a_2)^\top \, \text{—} \\ \vdots \\ \text{—} \, \varphi(s_{L_2}, a_{L_2})^\top \, \text{—} \end{bmatrix} \tag{2.21}$$

$$\begin{aligned} \varphi(s_i, a_i) &= \sum_{s_i' \in \mathcal{S}} \mathcal{P}_{s_i s_i'}^{a_i} \phi\left(s_i', \pi(s_i')\right) \\ &\approx \frac{1}{L_1} \sum_{j=1}^{L_1} \phi\left(s_j', \pi(s_j')\right), \quad s_j' \sim \mathcal{P}_{s_i}^{a_i} \end{aligned} \tag{2.22}$$

$$\widetilde{\boldsymbol{R}}_{L_2 \times 1} = \begin{bmatrix} \rho(s_1, a_1) \\ \rho(s_2, a_2) \\ \vdots \\ \rho(s_{L_2}, a_{L_2}) \end{bmatrix} \tag{2.23}$$

$$\begin{aligned} \rho(s_i, a_i) &= \sum_{s_i' \in \mathcal{S}} \mathcal{P}_{s_i s_i'}^{a_i} \mathcal{R}_{s_i s_i'}^{a_i} \\ &\approx \frac{1}{L_1} \sum_{j=1}^{L_1} \mathcal{R}_{s_i s_j'}^{a_i}, \quad s_j' \sim \mathcal{P}_{s_i}^{a_i}. \end{aligned} \tag{2.24}$$

Notice φ and ρ, are estimates of expected values through L_1 samples identical to Section 2.4.3. Now, defining the feature matrix

$$\widetilde{\boldsymbol{\Phi}}_{L_2 \times n} = \begin{bmatrix} \text{—} \, \phi(s_1, a_1)^\top \, \text{—} \\ \text{—} \, \phi(s_2, a_2)^\top \, \text{—} \\ \vdots \\ \text{—} \, \phi(s_{L_2}, a_{L_2})^\top \, \text{—} \end{bmatrix}, \tag{2.25}$$

the weight vector can be estimated as follows:

$$\boldsymbol{\theta} \approx \widetilde{\boldsymbol{A}}^{-1} \widetilde{\boldsymbol{b}} \tag{2.26}$$

$$\widetilde{\boldsymbol{A}} = \frac{1}{L_2} \widetilde{\boldsymbol{\Phi}}^\top (\widetilde{\boldsymbol{\Phi}} - \gamma \widetilde{\boldsymbol{P}\boldsymbol{\Phi}}) \tag{2.27}$$

$$\widetilde{\boldsymbol{b}} = \frac{1}{L_2} \widetilde{\boldsymbol{\Phi}}^\top \widetilde{\boldsymbol{R}}. \tag{2.28}$$

Table 2.1: Memory and computation complexity involved in calculating $\widetilde{\boldsymbol{A}}$

Calculation	Computation Complexity	Memory	Equation
$Q(s,a)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	2.13
$\pi(s)$	$\mathcal{O}(n\|\mathcal{A}\|)$	$\mathcal{O}(n)$	2.11
$\varphi(s,\pi(s))$	$\mathcal{O}(nL_1\|\mathcal{A}\|)$	$\mathcal{O}(n)$	2.22
$\widetilde{\boldsymbol{P\Phi}}$	$\mathcal{O}(nL_1L_2\|\mathcal{A}\|)$	$\mathcal{O}(nL_2)$	2.21
$\widetilde{\boldsymbol{A}}$	$\mathcal{O}(n^2L_1 + nL_1L_2\|\mathcal{A}\|)$	$\mathcal{O}(n^2 + nL_2)$	2.27

Algorithm 4: Trajectory Based Policy Iteration (TBPI) — Complexity

Input: MDP, ϵ, L_1, L_2
Output: π

$\boldsymbol{\theta} \leftarrow$ Initialize arbitrarily
while *time left* **do**
  Create L_2 samples $\langle s_i, a_i \rangle$ following policy π^ϵ
  Calculate $\widetilde{\boldsymbol{A}}$ and $\widetilde{\boldsymbol{b}}$ using Equations 2.27, 2.28 — $\mathcal{O}(n^2L_2 + nL_2L_1|\mathcal{A}|)$
  $\boldsymbol{\theta} \leftarrow \widetilde{\boldsymbol{A}}^{-1}\widetilde{\boldsymbol{b}}$ (Use regularization if the system is ill-posed) — $\mathcal{O}(n^3)$
return π greedy w.r.t. Q

In MDPs with finite state and action sets, as the number of samples goes to infinity, the estimated weight vector $\hat{\theta}$ converges to $\boldsymbol{A}^{-1}\boldsymbol{b}$ if samples are gathered with the same distribution that defines $\boldsymbol{D}$ [see the discussion in Lagoudakis and Parr, 2003]. Then using the notation above yields:

$$\lim_{L_1,L_2\to\infty} \widetilde{\boldsymbol{A}}^{-1}\widetilde{\boldsymbol{b}} \;=\; \boldsymbol{A}^{-1}\boldsymbol{b}.$$

Furthermore, the $1/L_2$ term can be dropped from the calculations of $\widetilde{\boldsymbol{A}}$ and $\widetilde{\boldsymbol{b}}$ because $(cA)^{-1} = \frac{1}{c}A^{-1}$. The next step is to analyze the computation and memory complexity of policy evaluation using Equation 2.26. The memory and computational complexity required to calculate $\widetilde{\boldsymbol{A}}$ is shown in Table 2.1, in which, from top to bottom, calculating each building block of $\widetilde{\boldsymbol{A}}$ from scratch is analyzed. Calculating $\boldsymbol{\theta}$ requires inverting the $\widetilde{\boldsymbol{A}}$ matrix incurring $\mathcal{O}(n^3)$ computation. Consequently, performing policy iteration using Equation 2.26 requires $\mathcal{O}(n^3 + n^2L_1 + nL_1L_2|\mathcal{A}|)$ computation and $\mathcal{O}(n^2 + nL_2)$ memory, achieving our goal of eliminating all $\mathcal{O}(|\mathcal{S}|)$ dependencies.

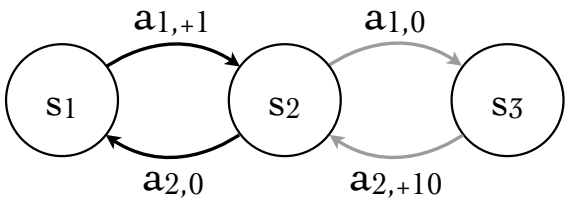

Figure 2.4: An MDP highlighting the importance of exploration for policy evaluation. Gray actions are not chosen by the initial policy. The result of policy evaluation assigns highest values to $Q(s_1, a_1)$ and $Q(s_2, a_2)$. The greedy policy with respect to this value function is sub-optimal because it ignores the $+10$ reward.

The combination of this policy evaluation scheme with the Equation 2.12 for policy iteration is shown as the *trajectory based policy iteration* algorithm (Algorithm 4). As opposed to Algorithm 1, the stopping condition in TBPI is based on the planning time because the computation involved in probing the policy for all states depends on $|\mathcal{S}|$. For regularization (line 5), $\lambda \boldsymbol{I}$ is added to the matrix $\widetilde{\boldsymbol{A}}$ before the inversion, where $\lambda > 0$ is a small scalar. Note that value iteration methods can also be used in the matrix format by following similar steps. One such example is the *fitted value iteration* algorithm [Boyan and Moore, 1995], which can diverge when a non-tabular representation is used [Gordon, 1995]. Unfortunately, a similar problem can cause TBPI to diverge with a non-tabular representation as well.

2.6.4 The Role of Exploration in TBPI

To collect L_2 samples, one can use the exploration strategy described in Section 2.4.2. Recall that this strategy collects state-action samples by selecting a random action on every time step with a small probability ϵ, and otherwise acts greedily with respect to the Q function. The value of ϵ plays a critical role in the TBPI algorithm for collecting samples. If ϵ is set to 0, some important transitions may not be visited regardless of the size of L_2. The lack of such samples can lead to poor policies in later iterations. As an example, Figure 2.4 shows an MDP with 3 states and 2 actions. Assume that $\pi(s_1) = a_1$ and $\pi(s_2) = a_2$. Rewards are highlighted as scalars on the right side of action la-

bels located on arrows. Samples gathered using policy π will not go through any of gray arrows, excluding the $+10$ reward. Hence using a tabular representation (one feature for each state), matrix $\widetilde{\boldsymbol{\Phi}}$ will have only two distinct rows[8] corresponding to $\phi(s_1, a_1)^\top$ and $\phi(s_2, a_2)^\top$:

$$\widetilde{\boldsymbol{\Phi}} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}, \qquad \widetilde{\boldsymbol{P\Phi}} = \begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}.$$

Setting $\gamma = 0.9$:

$$\widetilde{\boldsymbol{A}} = \begin{bmatrix} 0.5000 & 0 & 0 & 0 & -0.4500 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ -0.4500 & 0 & 0 & 0 & 0.5000 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}, \widetilde{\boldsymbol{b}} = \begin{bmatrix} 0.5000 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}.$$

Using regularization,

$$\tilde{\boldsymbol{Q}} = \begin{bmatrix} \tilde{Q}(s_1, a_1) \\ \tilde{Q}(s_2, a_1) \\ \tilde{Q}(s_3, a_1) \\ \tilde{Q}(s_1, a_2) \\ \tilde{Q}(s_2, a_2) \\ \tilde{Q}(s_3, a_2) \end{bmatrix} = \boldsymbol{\theta} = \widetilde{\boldsymbol{A}}^{-1}\widetilde{\boldsymbol{b}} = \begin{bmatrix} 5.2532 \\ 0 \\ 0 \\ 0 \\ 4.7269 \\ 0 \end{bmatrix}.$$

While $\widetilde{\boldsymbol{A}}$ is not full-rank, any non-zero regularization value leads to zero value estimates for all unseen state-action pairs. Applying policy improvement on the resulting Q function leads to the same policy π, which is sub-optimal, as it ignores the $+10$ reward.

2.6.5 Discussion

Table 2.2 provides an overview of the model-based MDP solvers discussed in this paper together with their per-iteration computational complexity and memory requirements. However, the choice of the "best" algorithm depends

[8] While more samples could be obtained (*i.e.,* $L_2 > 2$), the resulting value function does not change due to the use of a tabular representation.

Table 2.2: Model-based MDP solvers and their iteration computational complexity.

Algorithm	Iteration Complexity	Memory	Algorithm
Policy Iteration	$\mathcal{O}((N + \lvert\mathcal{A}\rvert)\lvert\mathcal{S}\rvert^2)$	$\mathcal{O}(\lvert\mathcal{S}\rvert)$	1
Value Iteration	$\mathcal{O}(\lvert\mathcal{A}\rvert\lvert\mathcal{S}\rvert^2)$	$\mathcal{O}(\lvert\mathcal{S}\rvert)$	2
TBVI	$\mathcal{O}(nTL_1\lvert\mathcal{A}\rvert)$	$\mathcal{O}(n)$	3
TBPI	$\mathcal{O}(n^3 + n^2L_1 + nL_1L_2\lvert\mathcal{A}\rvert)$	$\mathcal{O}(n^2 + nL_2)$	4

both on the underlying representation and the domain, as these methods trade off computational complexity with accuracy and approximate the value function within the class of functions dictated by the representation[9]. Often the size of the state space, $|\mathcal{S}|$, is very large for real-world problems, which eliminates methods with computational complexity or memory dependent on $|\mathcal{S}|$. However, more assumptions about a certain domain can help practitioners reduce the stated complexities.

While often $|\mathcal{A}| \ll |\mathcal{S}|$, for some domains with a large number or continuous actions, having a dependency on $|\mathcal{A}|$ is not sustainable either.[10] Hence further approximation is required to eliminate $|\mathcal{A}|$ from the above complexities [see *e.g.,* Antos et al., 2007]. Finally it is critical to note that Table 2.2 merely states the per-iteration complexity of each method, but it does not say anything about the number of iterations required to obtain a reasonable policy, nor about their behavior if they diverge. For example, some methods might require more time to finish one iteration, but they may require fewer iterations in total to find good policies. Recent methods have combined the idea of LSTD with kernelized representations to achieve good approximations of the value function with a small amount of data [Engel et al., 2003, Farahmand et al., 2008, Bethke and How, 2009, Taylor and Parr, 2009], but these techniques are outside the scope of this tutorial.

[9] In addition there is the possibility of divergence due to approximations or incorrect value functions due to sampling.

[10] Remember that $n = m|\mathcal{A}|$.

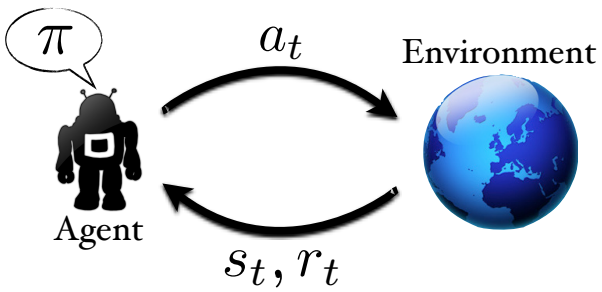

Figure 2.5: An agent interacting with an environment following policy π. The agent does not know the underlying reward or transition models of the MDP.

2.7 Reinforcement Learning

In practical domains, the MDP parameters (*i.e.,* $\mathcal{P}$ and $\mathcal{R}$) are often not known, cannot be sampled with random access, or may be too complicated to be captured analytically. These problems are addressed by the reinforcement learning (RL) framework shown in Figure 2.5, where an agent (left) interacts with the environment (right) on each time step, using policy π [Bertsekas and Tsitsiklis, 1996, Sutton and Barto, 1998, Buşoniu et al., 2010, Szepesvári, 2010]. The goal of RL methods is to achieve high reward, usually by solving MDPs with unknown models, solely through interaction with the environment. At first, the RL framework might look very different from the DP setting discussed in the previous section. Yet, as will be shown in this section, many RL techniques can be seen as dynamic programming methods where samples are obtained through sequential interaction.

RL methods are organized roughly into three categories: 1) model-free value-based methods [*e.g.,* Watkins and Dayan, 1992, Rummery and Niranjan, 1994]; 2) model-based value-based methods [*e.g.,* Brafman and Tennenholtz, 2002, Szita and Szepesvári, 2010]; and 3) model-free policy search techniques [*e.g.,* Williams, 1992, Bhatnagar et al., 2007]. This paper focuses on the first category because these methods require less computation than the model-based RL and are more closely connected to DP methods than policy search. Again, the core idea of value-based techniques is to follow the

policy evaluation/improvement loop, shown in Figure 2.2. Before continuing further, let us step back and see what extra restrictions an RL agent has to overcome compared to planning methods:

- **Restriction I:** Since sampling from the environment is only available through interaction, samples are gathered as part of a trajectory. That is, the agent does not have random access to all states at once. For example in a navigation task, the agent cannot ask the environment about the consequence of turning left at arbitrary junctions; it can only obtain samples in its current location, perhaps later reaching the junction of interest.

- **Restriction II:** In the RL setting, one-step look-ahead from the true model (*i.e.,* Equation 2.9) cannot be directly used for policy improvement since it requires knowledge of both $\mathcal{P}$ and $\mathcal{R}$. The agent can build its own model of the environment, value function, or optimal policy from experience, but it cannot directly access the model parameters (i.e. $\mathcal{P}$ and $\mathcal{R}$).

- **Restriction III:** In some settings, the time between interactions is limited. For example a flying robot can run out of power and crash while calculating the inverse of a large matrix. Hence the per-time-step complexity (as opposed to offline calculations) of the RL agent plays a critical role in its applicability to time sensitive domains.

The goal in this section is to introduce algorithms that satisfy Restrictions **I**-**III**. Restriction **I** is addressed by learning algorithms that collect data through sampled trajectories. RL methods hedge against Restriction **II** the same way the storage complexity of π was reduced to be independent of $|\mathcal{S}|$ in Section 2.4.1. This is done by estimating Q values instead of V values and relying on Equation 2.11 or Equation 2.12 to infer the policy. Finally, to deal with Restriction **III**, RL methods either provide computationally cheap learning during interaction with the environment (*i.e.,* online RL) or by carrying out learning after obtaining samples from the environment (*i.e.,* batch learning). The next section describes popular online and batch RL methods and shows that they can be viewed as approximate dynamic programming techniques. Specifically, we describe two classical RL methods (Q-Learning

and SARSA) derived from TBVI and two batch RL algorithms (LSTDQ and LSPI) derived from TBPI.

Note that due to their reliance on samples, RL methods are particularly sensitive to their exploration methods, which determine how they seek out new samples of the environment while still attempting to perform well. This trade-off between exploration and exploitation has a rich literature beyond the scope of this tutorial [Strehl et al., 2009, Nouri and Littman, 2009, Asmuth et al., 2009, Jaksch et al., 2010, Li, 2012]. Here, we will use one of the most basic exploration methods, namely: ϵ-greedy, the same method used for collecting samples in the previous sections. However, there are cases where this basic technique can take exponential time (in the size of the domain) to adequately explore [Whitehead, 1991] and more complicated exploration techniques must be used to ensure tractability [Strehl et al., 2009]. In domains where safety is a concern (*i.e.,* specific actions could damage the agent or humans), reliance on ϵ-greedy exploration can also be risky. Recent work has improved the use of RL techniques for risk sensitive agents by limiting the amount of exploration they perform [Mihatsch and Neuneier, 2002, Geibel and Wysotzki, 2005, Geramifard et al., 2012].

2.7.1 Q-Learning

To address Restriction **I**, trajectory based sampling is used, as described in Section 2.4.2. Such modifications blurs the line between planning and learning because, as shown, some planners also use samples for computational tractability, yet they allow random access to states for sampling. The following considers what happens to these algorithms if samples are restricted to be generated from connected trajectories, as enforced by Restriction **I**.

Let us focus on Algorithm 3, TBVI. Considering Restriction **I**, only one next state can be sampled from each state. Hence L_1 must be 1 to allow for learning from a trajectory. As discussed in Section 2.5 the Q^+ is now computable using the next state, s', and the received reward, r which are sampled through interaction with the environment (*i.e.,* setting $L_1 = 1$ in line 5 of Algorithm 3). The resulting *online* algorithm is known as Q-Learning [Watkins, 1989, 1992] and is shown in Algorithm 5. Hence Q-Learning can be seen as TBVI with $L_1 = 1$.

The complexities of online methods are described in terms of per-time-

Algorithm 5: Q-Learning — Complexity

Input: MDP$\setminus\{\mathcal{P}, \mathcal{R}\}, \alpha, \epsilon$
Output: π
1 $\boldsymbol{\theta} \leftarrow$ Initialize arbitrarily
2 $\langle s, a\rangle \leftarrow \langle s_0, \pi^\epsilon(s_0)\rangle$
3 **while** *time left* **do**
4 Take action a and receive reward r and next state s'
5 $Q^+(s,a) \leftarrow r + \gamma \max_{a'} Q(s', a')$ — $\mathcal{O}(n|\mathcal{A}|)$
6 $\delta \leftarrow Q^+(s,a) - Q(s,a)$
7 $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha\delta\phi(s,a)$ — $\mathcal{O}(n)$
8 $\langle s, a\rangle \leftarrow \langle s', \pi^\epsilon(s')\rangle$ — $\mathcal{O}(n|\mathcal{A}|)$
9 return π greedy w.r.t. Q

step computational complexity. As expected, Q-Learning has the same complexity as Algorithm 3 with $L_1 = 1$ and $T = 1$, which is $\mathcal{O}(n|\mathcal{A}|)$. The memory requirement of Q-Learning is the same: $\mathcal{O}(n)$. The δ calculated on line 6 of Algorithm 5 highlights the difference between the better estimate of the Q function based on a single interaction, $Q^+(s,a)$, and the current estimate, $Q(s,a)$. This quantity is called the *temporal difference* (TD) error in the literature [Sutton and Barto, 1998].

Q-Learning is an *off-policy* algorithm, that is, the policy it uses to generate the samples (*i.e.,* π^ϵ) is not necessarily the same as the policy corresponding to the current value function computed using the samples (*i.e.,* the policy that is greedy with respect to the current value function). Q-Learning is guaranteed to converge when a finite tabular representation is used, the reward variance is bounded, the α parameter is decayed at the proper rate, and a discount factor is used [Jaakkola et al., 1993]. Under certain conditions, Q-Learning has been shown to converge with function approximation [Melo et al., 2008], however those conditions may not easily be met in practice, and examples illustrating the potential divergence of Q-Learning are available [Baird, 1995]. Recent methods such as GQ [Maei et al., 2010, Maei and Sutton, 2010] have addressed the convergence issues of off-policy learning algorithms such as Q-Learning, but are beyond the scope of this tutorial.

Algorithm 6: SARSA — Complexity

Input: MDP$\setminus\{\mathcal{P}, \mathcal{R}\}, \alpha, \epsilon$
Output: π
$\boldsymbol{\theta} \leftarrow$ Initialize arbitrarily
$\langle s, a\rangle \leftarrow \langle s_0, \pi^{\epsilon}(s_0)\rangle$
while *time left* **do**
  Take action a and receive reward r and next state s'
  $a' \leftarrow \pi^{\epsilon}(s')$ — $\mathcal{O}(n|\mathcal{A}|)$
  $Q^{+}(s, a) \leftarrow r + \gamma Q(s', a')$ — $\mathcal{O}(n)$
  $\delta \leftarrow Q^{+}(s, a) - Q(s, a)$
  $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha\delta\phi(s, a)$ — $\mathcal{O}(n)$
  $\langle s, a\rangle \leftarrow \langle s', a'\rangle$
return π greedy w.r.t. Q

2.7.2 SARSA

The divergence of Q-Learning when function approximation is used stems from its off-policy property: the policy that is used to gather samples (in our case π^{ϵ}) is not the same as the policy used for learning (greedy w.r.t. Q values) [Sutton and Barto, 1998]. We now consider an online learning method, SARSA [Rummery and Niranjan, 1994], shown in Algorithm 6 that learns the value function of the policy it is currently implementing. The acronym SARSA stands for "state, action, reward, state, action," which is the subset of a trajectory used to compute updates. SARSA has convergence guarantees in the tabular case [Singh et al., 2000] and under milder conditions compared to Q-Learning when used with linear value function approximation [Melo et al., 2008]. SARSA has the same per-time-step computational complexity, $\mathcal{O}(n|\mathcal{A}|)$, and memory requirement, $\mathcal{O}(n)$, as Q-Learning. Notice that, compared to Algorithm 5, $Q^{+}(s, a)$ is now calculated based on $Q(s', \pi^{\epsilon}(s'))$ rather than $\max_{a'} Q(s', a')$, making both the sampling and learning policies identical. This technique is known as *on-policy* learning, meaning that the agent learns about the same policy used for generating trajectories.

2.7.3 Least-Squares Techniques

Policy iteration based methods can also be used in the context of RL by accounting for the Restrictions **I**-**III**. Let us revisit Algorithm 4. Suppose we

Algorithm 7: LSTDQ — Complexity

Input: MDP$\setminus\{\mathcal{P}, \mathcal{R}\}, \epsilon, L_2$
Output: π

1 $\boldsymbol{\theta} \leftarrow$ Initialize arbitrarily
2 **while** *time left* **do**
3 Create L_2 samples $\langle s_i, a_i, r_i, s'_i, a'_i \rangle$ following π^ϵ — $\mathcal{O}(nL_2|\mathcal{A}|)$
4 Calculate $\widetilde{\boldsymbol{A}}$ and $\widetilde{\boldsymbol{b}}$ using Equations 2.29, 2.27, and 2.28 — $\mathcal{O}(n^2 L_2)$
5 $\boldsymbol{\theta} \leftarrow \widetilde{\boldsymbol{A}}^{-1}\widetilde{\boldsymbol{b}}$ (Use regularization if the system is ill-posed) — $\mathcal{O}(n^3)$
6 return π greedy w.r.t. Q

set $L_1 = 1$ in Equations 2.22 and 2.24, because samples can be generated only by following a trajectory. Consequently, samples are gathered as $\langle s_i, a_i, r_i, s'_i, a'_i \rangle$, where $s_{i+1} = s'_i$ and $a_{i+1} = a'_i$. Then $\widetilde{\boldsymbol{P}\boldsymbol{\Phi}}$ and $\widetilde{\boldsymbol{R}}$ in Equation 2.27 and Equation 2.28 can be calculated by:

$$\widetilde{\boldsymbol{P}\boldsymbol{\Phi}} = \begin{bmatrix} \text{—} \; \phi^\top(s'_1, a'_1) \; \text{—} \\ \text{—} \; \phi^\top(s'_2, a'_2) \; \text{—} \\ \vdots \\ \text{—} \; \phi^\top(s'_{L_2}, a'_{L_2}) \; \text{—} \end{bmatrix}, \widetilde{\boldsymbol{R}} = \begin{bmatrix} r_1 \\ r_2 \\ \vdots \\ r_{L_2} \end{bmatrix}. \tag{2.29}$$

The number of samples collected, L_2, controls the accuracy of estimates for $\widetilde{\boldsymbol{A}}$ and $\widetilde{\boldsymbol{b}}$. Again, in the limit of infinite samples, approximations become exact provided that the sampling policy visits all state-action pairs infinitely often asymptotically. More discussions on exploration and data collection for least-squares techniques can be found in the work of Li et al. [2009a]. Algorithm 4 represents the policy using Q values, hence addressing Restriction **II**. Restriction **III**, on the other hand, is sidestepped here because of the *batch* nature of Algorithm 4, because the learning phase (lines 4-5) happens after the sample gathering phase (line 3).

The resulting method is known as LSTDQ [Lagoudakis and Parr, 2003] shown in Algorithm 7.[11] The computational complexity of important lines is also shown. This algorithm has $\mathcal{O}(n^3 + n^2 L_2 + nL_2|\mathcal{A}|)$ complexity per iteration and an $\mathcal{O}(n^2 + nL_2)$ memory requirement. It is easy to verify that

[11] The original LSTDQ algorithm was introduced to run on a fixed batch of data. Here we extended the algorithm allowing multiple collection of samples and calculating the weights.

Algorithm 8: Least-Squares Policy Iteration (LSPI) — Complexity

Input: MDP$\setminus\{\mathcal{P}, \mathcal{R}\}, \epsilon, L_2$
Output: π

$\boldsymbol{\theta} \leftarrow$ Initialize arbitrarily
Create L_2 samples $\langle s_i, a_i, r_i, s'_i, a'_i \rangle$ following ϵ-greedy policy π^ϵ
while *time left* **do**
  Calculate $\widetilde{\boldsymbol{A}}$ and $\widetilde{\boldsymbol{b}}$ using Equations 2.29, 2.27 and 2.28 — $\mathcal{O}(n^2 L_2)$
  $\boldsymbol{\theta} \leftarrow \widetilde{\boldsymbol{A}}^{-1}\widetilde{\boldsymbol{b}}$ (Use regularization if the system is ill-posed) — $\mathcal{O}(n^3)$
  $a'_i \leftarrow \text{argmax}_{a \in \mathcal{A}} Q(s'_i, a)$ **For** all i — $\mathcal{O}(nL_2|\mathcal{A}|)$
return π greedy w.r.t. Q

setting $L_1 = 1$ in Table 2.2 for Algorithm 4 yields the same complexity. An important fact about this method is that samples generated to estimate $\widetilde{\boldsymbol{A}}$ and $\widetilde{\boldsymbol{b}}$ are discarded after each cycle, making this algorithm inefficient for domains with an expensive cost of sample acquisition (*e.g.,* flying unmanned aerial vehicles with a chance of crashing).

Least-squares policy iteration (LSPI) [Lagoudakis and Parr, 2003] mitigates the problem of sample cost by optionally reusing the same set of samples over and over in each iteration. On each iteration, LSPI selects samples to evaluate the new policy by switching a'_i to $\pi(s'_i)$, where π is greedy with respect to the most recent Q values. The result is shown in Algorithm 8. While the per-iteration complexity remains unchanged, the same set of data can be reused through all iterations. Notice that given a fixed set of samples, LSPI does not necessarily improve the policy between each iteration, hence it lacks convergence guarantee, but it will not diverge either as in the worst case it will switch between policies. Antos et al. [2008] discuss the conditions under which performance bounds can be obtained for LSPI.

LSPI has been shown to work well in practice with sufficient data, but its strong bias based on the initial set of samples can hinder its performance. Specifically, if the initial sample distribution is far away from the sample distribution under good policies and the same samples are reused, then LSPI performs poorly. This drawback is currently handled by manual filtering of samples gathered by a domain expert [Lagoudakis and Parr, 2003, Petrik et al., 2010], by approximating a policy-independent model using the gathered samples [Bowling et al., 2008], or by obtaining new samples corresponding to the

Table 2.3: RL methods and their per-time-step/iteration computational complexity.

Algorithm	Iteration Complexity	Memory	Algorithm
Q-Learning	$\mathcal{O}(n\|\mathcal{A}\|)$	$\mathcal{O}(n)$	5
SARSA	$\mathcal{O}(n\|\mathcal{A}\|)$	$\mathcal{O}(n)$	6
LSTDQ	$\mathcal{O}(n^3 + n^2L_2 + nL_2\|\mathcal{A}\|)$	$\mathcal{O}(n^2 + nL_2)$	7
LSPI	$\mathcal{O}(n^3 + n^2L_2 + nL_2\|\mathcal{A}\|)$	$\mathcal{O}(n^2 + nL_2)$	8

new policy [Peters and Schaal, 2008, Li et al., 2009a].

2.7.4 Discussion

Table 2.3 provides an overview of RL methods discussed in this paper together with their iteration complexities and memory requirements. For the upper part of the table, iteration complexities correspond to the per-time-step computation. In general, the first two online methods provide cheap complexity per interaction, which is critical in dealing with domains where Restriction **III** allows only a short amount of interaction time. However, for some domains, if these methods are not paired with a powerful exploration strategy, their sample complexity may still make their total runtime prohibitive.

On the other hand, the last two batch algorithms often require fewer samples compared to online methods to produce good policies. Hence, if sample complexity is the only concern, batch methods are often preferred over online methods. Due to its popularity, LSPI has been extended to various algorithms [*e.g.,* Mahadevan, 2005, Li et al., 2009b, Farahmand et al., 2008]. There are also several other RL algorithms in the literature not discussed here. Interested readers are referred to the work of Bertsekas and Tsitsiklis [1996], Sutton and Barto [1998], Buşoniu et al. [2010] and Szepesvári [2010].

2.8 Big Picture

Figure 2.1 shows how all of the discussed dynamic programming and reinforcement learning algorithms were derived from the unified idea of policy evaluation/improvement. Each method is marked with its corresponding algo-

rithm number. By eliminating the matrix operator in Equation 2.8 for policy evaluation, and performing policy improvement after each full evaluation, we first introduced policy iteration. By increasing the frequency of policy improvements, we arrived at value iteration. In order to scale DP techniques to MDPs with large state-spaces, we addressed 4 problems to eliminate all $\mathcal{O}(|\mathcal{S}|)$ memory and computational dependencies. The result of these modifications was trajectory based value iteration (TBVI). Using the same techniques, we introduced the trajectory based policy iteration algorithm (TBPI), an approximate DP technique that uses matrix operations. Then by relaxing the assumption of having random access to the MDP model (*i.e.,* $\mathcal{P}$ and $\mathcal{R}$), we moved into the reinforcement learning setting. We first showed how the Q-Learning algorithm can be derived from TBVI by simply using a single sample collected along trajectories to estimate the Bellman backup (*i.e.,* setting $L_1 = 1$). We then matched the sampling policy with the policy being evaluated, arriving at the SARSA algorithm. On the right branch of Figure 2.1, we extended the TBPI algorithm and used one sample along the trajectory to estimate φ and ρ in Equations 2.22 and 2.24 to obtain the LSTDQ algorithm. Finally by changing a' to $\pi(s')$, we could reuse collected samples for policy iteration, arriving at the LSPI technique.

3

Representations

So far, we have described several techniques for planning and learning with a value function that is approximated based on a weighted combination of linear features $\phi : \mathcal{S} \rightarrow \mathbb{R}^n$. However, we sidestepped a question of crucial importance: "What are the *right* features for a given domain to represent the value function?". For some problems, domain experts can provide insight on defining useful features, but in general this is still an open problem within the planning and learning communities.

Many domains may have a natural set of *base features* that come directly from sensor inputs (such as the position and velocity of a vehicle), but these features may be inadequate to linearly model the value function. For instance, consider a domain with two Boolean base features x and y and a value function that was linear with respect to $x \oplus y$ (where $\oplus$ is an xor operator). A linear combination of the base features will lead to poor approximation of the Q-function, but adding an extra feature $z = x \wedge y$ will increase the accuracy. Choosing such features that will lead to high accuracy in the linear approximation of $Q(s, a)$ is one of the most important factors for obtaining good performance. Unfortunately, there is no "one size fits all" answer to this question. What may be a good representation of one domain may fail terribly in a similar environment, or may fail in the same environment after seemingly

minor modifications.

To quantify the benefits of certain representations over others, a natural measure is *coverage*, or the portion of the state space for which a feature's value is not zero (or in the case of continuous features, non-negligible). Features with low coverage provide better accuracy, while features with high coverage offer better generalization. Accuracy and generalization are both important, as the former defines the preciseness of the value function, while the latter affects the planning/learning time to reach good policies. For example, a tabular representation provides the most accurate representation for MDPs with finite state-action sizes, yet offers poor generalization in most cases.

As an example, consider the classical Inverted Pendulum domain illustrated in Figure 4.3. Base features can be formed for this domain by discretizing θ and $\dot{\theta}$ into a certain number of buckets separately. There are three discrete actions in this domain: 1) apply no force, 2) push the pendulum clockwise, and 3) push it counterclockwise. A negative reward is given when the pendulum hits the horizon and zero reward is given otherwise. Because of the physics of the domain, the value function in this task cannot be precisely represented as a linear combination of the base features. Nevertheless, a good approximation is possible by overlaying a very fine grid across the base features and using each grid cell as a feature in the domain, forming a tabular representation. These fine-grained features have small coverage, leading to an accurate value function, but potentially with the cost of maintaining a large representation. Furthermore, seemingly small modifications can make such a representation far better or worse. For instance, if the grid is too coarse (not enough cells) the fine-grained value distinctions that occur near the balancing point could be missed. Increasing the granularity (decreasing the feature coverage) in that region gives us a better representation with lower error, but also makes the representation much larger in terms of feature size, leading to longer computation/learning times and heavier memory requirements.

In addition to the grid-based tabular representation, this chapter introduces three representations that each attempt to balance the need for accuracy with the computational requirement of minimal features: 1) Fixed Sparse Representations, 2) Radial Basis Functions, and 3) Incremental Feature Dependency Discovery. The latter is of particular interest as adaptive represen-

tations tend to free practitioners from the burden of engineering exactly the "right" features for every domain. This list of possible representations is not exhaustive and is meant simply as a sample of possible representations and an introduction to those used in the experiments in the next chapter. Readers interested in comparisons of other representation methods in reinforcement learning with linear function approximation are directed to the works of Sutton [1996], Kretchmar and Anderson [1997] and Parr et al. [2008].

3.1 Tabular Representation

The tabular representation maintains a unique weight for each state-action pair. In discrete domains, this is an exact representation. In continuous domains, like the Inverted Pendulum above, it corresponds to a discretization of the state space by tiling the state space with uniform grid cells and using each cell as a feature. Since we would like to represent a linear approximation of the Q-function, features corresponding to state-action pairs can be built by first mapping the state to a set of discretized features $\phi : \mathcal{S} \rightarrow \mathbb{R}^n$ and then copying the features vector to the corresponding action slot while setting the rest of the features to zero (see the discussion surrounding equation 2.14 and Section 3.3. of [Buşoniu et al., 2010]).[1]

The main benefit of using a tabular representation is that feasible sufficient conditions exist to guarantee the convergence of most DP and RL algorithms to the optimal solution. However, since the size of this representation is just as large as the state-action space itself, the tabular representation does not scale well with domain size. Hence this approach can be the most demanding model in terms of memory and computational requirements. For instance, consider the value function in the Inverted Pendulum. A discretization of 50×50 could capture the value function with little error, but amounts to $2500 \times 3 = 7500$ features for a basic domain with only two dimensions and three actions. In general, discretizing each dimension of a d-dimensional continuous MDP with c cells leads to $c^d|\mathcal{A}|$ features, which, as we will see in our experiments, is intractable for high dimensional problems.

The tabular representation can be viewed as a linear function approxima-

[1] Notice that for continuous action spaces this approach is not feasible as the length of the feature vector will be infinite, though other methods [*e.g.,* Antos et al., 2007] may be successful.

tion with features that take on binary values. In particular, the feature $\phi(s, a)$ takes the value 1 (active) only at the state-action pair s, a and 0 (inactive) everywhere else. We now consider a different representation that does not explicitly enumerate the cross product of all the discretized base features, leading to a smaller set of features and better generalization across dimensions.

3.2 Fixed Sparse Representation

The Fixed Sparse Representation (FSR) is one of the simplest possible encodings where (unlike the tabular case) features can be active for multiple state-action pairs. Here we consider an FSR that represents each state with a binary encoding of its value in each dimension (a feature for each binary value of the base features). More formally, let the state s be represented by a d dimensional vector, where s_i corresponds to the i^{th} component, hence $s = (s_1, s_2, \cdots, s_d)$. Let n_i be the number of distinct values that the d^{th} dimension of the state space can take.[2] Consider the set $\{v_i^1, v_i^2, \cdots, v_i^{n_i}\}$, where v_i^j corresponds to the j^{th} possible value for the i^{th} dimension of the state space. The FSR features are created as follows:

$$\phi(s) = \left[\phi_{11}(s) \cdots \phi_{1n_1}(s), \phi_{21}(s) \cdots \phi_{2n_2}(s), \cdots, \phi_{d1}(s) \cdots \phi_{dn_d}(s)\right]^\top,$$

$$\phi_{ij}(s) = \begin{cases} 1 & s^i = v_i^j \\ 0 & \text{otherwise} \end{cases}, i = 1, \cdots, d, j = 1, \cdots, n_i,$$

amounting to a total of $m = \sum_{i=1}^{d} n_i$ features per action.

Notice that by allowing one feature from each dimension to contribute to the same state/action's value, we have exponentially decreased the size of the representation from the tabular case. For instance, consider the FSR representation of the Inverted Pendulum domain, even with a 50×50 discretization as before. With this FSR, only $100 \times 3 = 300$ weights need to be learned rather than the 7500 in the tabular case.

However, while FSRs generally try to balance coverage and generalization, their reliance on a static set of binary features makes the FSR one of the most sensitive representations to the choice of features. We will now see how

[2] Continuous dimensions can be discretized into buckets.

more complex representations using a distance metric can be built, leading to non-binary features.

3.3 Radial Basis Functions (RBFs)

Instead of using a binary representation of the discretized base features, as the previous methods have considered, we now describe a representation that works directly in the continuous space. Gaussian Radial Basis Functions (hereafter referred to as RBFs) [see *e.g.,* Moody and Darken, 1989, Haykin, 1994] are Gaussian functions spread across the domain of the Q function. Unlike binary FSRs, the feature activation of each RBF decays continuously away from the state-action pair where the RBF center is placed. Therefore, an RBF feature takes on significant nonzero values across multiple state values, and hence can be viewed to be active over a wide range of states. The output of the j^{th} RBF kernel centered around $\bar{s}_j$ is,

$$\phi_j(s) = e^{-\frac{\|s - \bar{s}_j\|^2}{2\mu_j^2}},$$

where μ_j is the bandwidth of the RBF that determines the rate of decay of RBF output when evaluated away from the center s_j. A larger bandwidth results in a flatter RBF. It is also possible to formulate RBFs to select a different bandwidth for each dimension [see *e.g.,* Buşoniu et al., 2010]. The output of an RBF is nearly zero when evaluated at a state s that is far from the center s_j. Therefore, the location of RBF centers greatly affects the accuracy and validity of the resulting representation. Hence, poorly placed RBFs can fail to capture the value function even in some simple domains. For instance, in the Inverted Pendulum domain, the centers need to be dense enough near the balance point to capture the rapidly changing values in that region, but this could lead to relatively low coverage elsewhere. Hence, there is a conflicting requirement that centers must also be placed sparse enough to adequately cover the full space. In our experiments, we generated many different possible RBF centers and bandwidths and found that uniform-random placement[3] often led to policies that could not balance the pole for more than a few steps,

[3]The bandwidth for each dimension was sampled uniformly between the middle and maximum dimension value.

while a hand-tuned basis selection that scatters the RBFs uniformly along the two dimensions was able to capture the value function sufficiently to balance the pole for thousands of steps.

The issues with center placement can be alleviated in many ways. Gaussian RBFs are Mercer kernels [see *e.g.,* Schölkopf and Smola, 2002]. Hence, nonparametric kernel placement techniques that rely on insights from Reproducing Kernel Hilbert Spaces can be used to place RBF centers online [Schölkopf and Smola, 2002, Rasmussen and Williams, 2006, Liu et al., 2008]. This approach can enhance the applicability of an RBF representation by ensuring that centers are placed wherever samples are obtained. There are other techniques for dynamically placing RBFs too, such as moving the centers of RBFs using gradient descent techniques [Barreto and Anderson, 2008]. However, we did not pursue these methods in this work. Instead, we explored an *adaptive* representation over discrete features, which is described in the next section.

3.4 Incremental Feature Dependency Discovery (iFDD) representation

Unlike a static representation, the Incremental Feature Dependency Discovery (iFDD) representation [Geramifard et al., 2011] is dynamic, in the sense that the set of features is updated online. iFDD is one of many methods that make these adaptations based on errors in the value function [*e.g.,* Parr et al., 2007, 2008]. The basic idea in all of these techniques is to analyze the temporal difference errors in the previous time steps (based on the current representation) and to create new features that helps mitigate these errors. Unlike methods that create arbitrary features solely based on the TD-errors, iFDD is restricted to making new features that are conjunctions of previous features, starting with an initial set of features (*i.e., base* features). Thus, iFDD can be thought of as an incremental search technique through the space of possible base-feature conjunctions, guided by the TD-error as a heuristic.

Given an initial set of features, iFDD is guaranteed to converge to the best possible representation achievable with conjunctions of the initial set of features [Geramifard et al., 2011]. Note that the best representation found by iFDD does not always include all possible feature conjunctions. Features whose combination does not improve the value function esttimate are not

added, avoiding a full expansion to the tabular case. In our experiments, we used the most recent extension of iFDD in an online setting [Geramifard et al., 2013b]. In the Inverted Pendulum domain, by discretizing both dimensions the FSR approach can be used to provide the base features. Then iFDD adds new features by generating "cells" that combine one feature from each dimension (θ and $\dot{\theta}$). So iFDD essentially takes the FSR features (potentially with too high coverage) and selectively adds features with more accuracy that help represent the value function better. This keeps the representation small, but allows it to overcome an imperfect initial selection of features, although iFDD cannot generate entirely new base features (e.g. changing the discretization scheme).

We have covered a variety of representations and some intuitions about their use. In Chapter 4, we provide a thorough empirical investigation of these different representations in four benchmark domains. Finally, we remind the reader that these representations are only a small sample of the possible linear function approximators that can be used to estimate the value function. Summaries and comparisons of these and other representations include those by Sutton and Barto [1998], Sutton [1996], Kretchmar and Anderson [1997], and Parr et al. [2008].

4

Empirical Results

This chapter empirically evaluates several algorithms' performance on four benchmark domains: 1) Gridworld, 2) Inverted Pendulum [Widrow and Smith, 1964], 3) BlocksWorld [Winograd, 1971], and 4) Persistent Search and Track [Geramifard et al., 2011]. We chose a mixture of standard benchmarks as well as benchmarks for larger problem domains with structured state spaces, which have more relevance to real-world problems. We first provide a brief description of each domain, including a formal model as an MDP. After describing the parameter settings for the algorithms and representations, we provide the empirical results of six solvers: three dynamic programming methods and three reinforcement learning techniques. Recently there has been extensive interest in evaluation for reinforcement learning [Riedmiller et al., 2007, Whiteson and Littman, 2011, Sanner, 2011, RLC, 2012, Dutech et al., 2005]. In contrast to these general RL evaluations, this section focuses specifically on a family of algorithms related to linear function approximation and dynamic programming. Moreover, we probe the performance of each algorithm and each domain across four different representations as opposed to similar works in the literature focusing on one representation per domain [*e.g.,* Kalyanakrishnan and Stone, 2011].

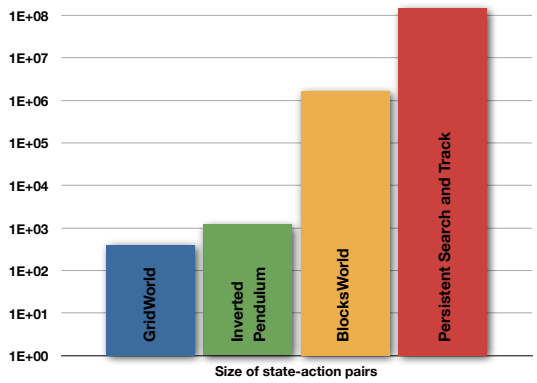

Figure 4.1: The size of state-action pairs for all 4 empirical domains in log-scale. The space size of the inverted pendulum is based on our discretization.

4.1 Algorithms

We evaluated six algorithms discussed in Section 2. From the DP algorithms, we evaluated policy iteration (Algorithm 1), value iteration (Algorithm 2), and TBVI (Algorithm 3). In the RL setting, we evaluated Q-Learning (Algorithm 5), SARSA (Algorithm 6), and LSPI (Algorithm 8).

4.2 Domain Descriptions

Four benchmark domains with varying state-action pair sizes are considered. We chose Gridworld (400 pairs) and Inverted Pendulum (1200 pairs after discretizing the states) because they are standard benchmark problems. Our additional test domains gradually increase the size and complexity of the state-action space: BlocksWorld ($\sim 1.7 \times 10^6$ pairs), and Persistent Search and Track ($\sim 150 \times 10^6$ pairs). Figure 4.1 depicts all the state/action pair sizes on a logarithmic scale. Notice that for the Inverted Pendulum domain, the number is based on the aggregated states generated by our discretization. The following sections define the MDP formulation of each of the above domains.

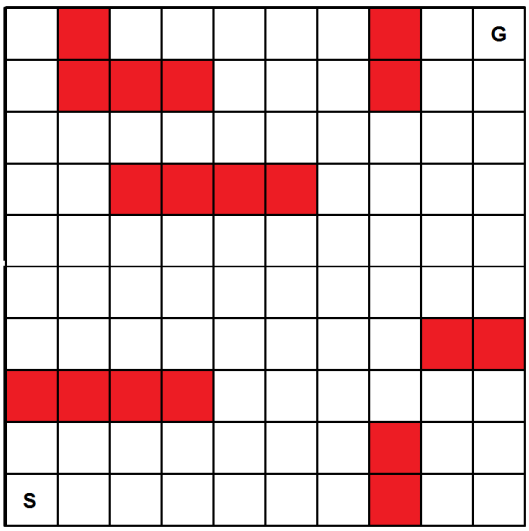

Figure 4.2: The GridWorld domain: the task is to navigate from the starting point (S) to the goal point (G). Red cells are blocked. Actions are the four cardinal directions. The movement is stochastic, with 20% probability of taking a random action.

4.2.1 GridWorld

The GridWorld domain simulates a path-planning problem for a mobile robot in an environment with obstacles. The small size of the state-space in this domain makes it relatively easy to use a tabular representation, thereby allowing comparison of approximate representations to a full tabular representation. Figure 4.2 depicts our simple grid world domain. The goal of the agent is to navigate from the starting point (S) to the goal state (G), using four cardinal directions: $\{\uparrow, \downarrow, \leftarrow, \rightarrow\}$. White cells are traversable, while red cells are blocked. There is a 30% chance that the intended action is replaced with a random action at each time-step. The agent cannot select actions that move it to a blocked grid. The reward on each step is -10^{-3} , except for actions that bring the agent to the goal with reward of $+1$. The size of the state-action space for this domain is $100 \times 4 = 400$.

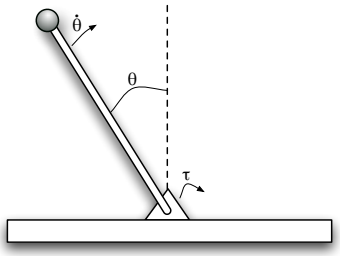

Figure 4.3: The Inverted Pendulum domain: The task is to maintain the balance of a pendulum using torque values applied to the base.

4.2.2 Inverted Pendulum

The inverted pendulum is a popular benchmark problem that is intended to evaluate the performance of the algorithm in the control of a continuous dynamical system [Widrow and Smith, 1964]. The goal is to balance a pendulum starting from a perturbed upright position up to 3000 steps. The state-space is a two dimensional continuous space (dimensions are shown in Figure 4.3). The dynamical model used to simulate state transition for the experiments is based on the work of Lagoudakis and Parr [2003]. The state of the system is described by the angular position of the pendulum $\in [-\frac{\pi}{2}, +\frac{\pi}{2}]$ and the angular rate $\in [-2, 2]$. In order to discretize the continuous state-space, each dimension is divided to 20 uniform intervals where the center of each cell represents all states in the cell. Available actions are three torque values of $[-50, 0, +50]$. In addition, stochasticity is introduced by adding a uniform random torque $\in [-10, 10]$. The reward is 0 for the steps where the angle of the pendulum $\in [-\frac{\pi}{2}, +\frac{\pi}{2}]$ and -1 outside of the interval. The resulting size of the approximated state-action space is $20^2 \times 3 = 1200$. We note that while this discretization can potentially introduce partial observability, the fineness of the partitioning and our later empirical results indicate this resolution is sufficient to recover a "good enough" policy.

4.2.3 BlocksWorld

The BlocksWorld domain is a classical domain for testing planning methods in the artificial intelligence community [Winograd, 1971]. Figure 4.4 shows a state in the BlocksWorld domain that consists of 6 labeled blocks on a table.

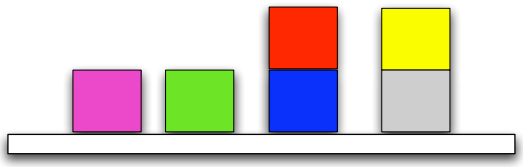

Figure 4.4: The Blocks World domain: the task is to build an ordered stack of blocks. Each block stacking attempt has an associated reward of -0.001 and a 30% chance of dropping the block on the table.

The objective is to put blocks on top of each other in a specific order to form a tower. Initially all blocks are unstacked and are on the table.

We consider a variant of BlocksWorld with 6 blocks. The state of the MDP is defined by 6 integer values $[s_1 \cdots s_6]$: $s_i = j$ indicates that block i is on top of j (for compactness $s_i = i$ indicates that the block i is on the table). At each step, the agent can take a block, and put it on top of another block or move it to the table, given that blocks do not have any other blocks on top of them prior to this action. We also added 30% probability of failure for each move, in which case the agent drops the moving block on the table. The reward is -10^{-3} for each step where the tower is not built and $+1.0$ when the tower is built. An estimate of the number of state-action pairs is $6^6 \times 36 \approx 1.7 \times 10^6$.

4.2.4 Persistent Search and Track (PST)

Persistent Search and Track (PST) is a multi-agent Unmanned Aerial Vehicle (UAV) mission planning problem (shown in Figure 4.5), where 3 UAVs perform surveillance on a target, in the presence of communication and health constraints [Geramifard et al., 2011]. This domain simulates a real-world planning problem with a relatively large state-action space. Each UAV's individual state has four dimensions: location, fuel, actuator status, and sensor status. The dimensionality of the state vector is therefore $4 \times 3 = 12$. The full state space is the combination of states for all UAVs. There are three available actions for each UAV: $\{advance, retreat, loiter\}$. Hence the size of the action space is $3^3 = 27$.

The objective of the mission is to fly to the surveillance node and perform surveillance on a target, while ensuring that a communication link with the

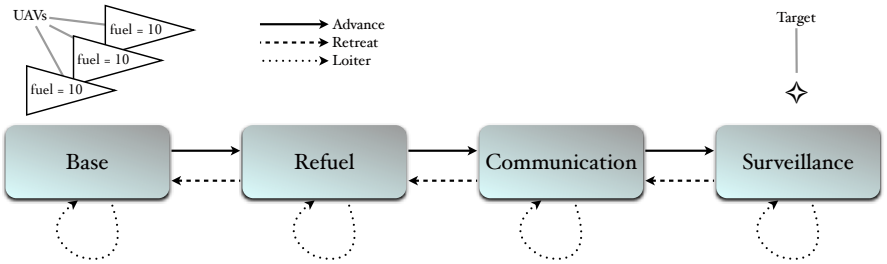

Figure 4.5: The Persistent Search and Track domain: the goal is to maintain surveillance on the target on the right, while facilitating a data link with the base by having a UAV with a functional actuator loitering in the communication area.

base is maintained by having a UAV with a working actuator loitering on the communication node. Each UAV starts with 10 units of fuel and burns one unit for all actions except when loitering in the base or refuel nodes. Executing "loiter" in the base and refuel nodes fixes all UAV failures and fully refuels the UAV, respectively. If a UAV runs out of fuel, it is assumed to crash, terminating the episode. Furthermore the sensor and actuator of each UAV can fail on each step with a 5% probability. A UAV with a failed sensor cannot perform surveillance. A UAV with a failed actuator cannot perform surveillance or communication. It can either retreat or loiter in refuel or base nodes. The reward function is:

$$\mathcal{R}(s,a) = +20 I_{comm} I_{ally} - 50 I_{crash} - f_b,$$

where, I_{comm} is a binary variable set to 1 if there is a UAV in the communication area with a working actuator and 0 otherwise, I_{ally} is similarly defined if a UAV with a working sensor is in the surveillance area. f_b is the total fuel burned by all UAVs, and I_{crash} is a binary variable set to 1 if any UAV crashes. The size of state-action pairs for this domain is $\sim 150 \times 10^6$.

4.3 Simulation Framework

This section describes the common setup used for our experiments, including the representations, learning and exploration schedules, and various algorithm parameters. All results were generated using the RLPy framework [Geramifard et al., 2013a] which is available online for download.

4.3.1 Representations

The convergence and the quality of the approximate solution obtained when using DP or RL algorithms depends on the representation used to represent the state-action value function, as described in Chapter 3. In our experiments, we used the four representations discussed in detail earlier: a *tabular* representation, the base features in a *Fixed Sparse Representation* (FSR), *Radial Basis Functions* (RBFs) and an adaptive technique, *Incremental Feature Dependency Discovery* (iFDD).

To move these representations into the state/action space, each $\langle s, a \rangle$ pair first had the state mapped to a set of features $\phi : \mathcal{S} \rightarrow \mathbb{R}^n$ and then the feature vector was copied to the corresponding action slot while setting the rest of the features to zero (see the discussion surrounding Equation 2.14 and Example 3.1 of the work of Buşoniu et al. [2010]). The use of tabular representation for finite state MDPs is straightforward. For the continuous Inverted Pendulum, each dimension was discretized into 20 buckets, leading to 400 states corresponding to the centers of the cells. For BlocksWorld and PST, due to the large number of state-action pairs, the tabular representation used a caching mechanism in order to create a unique feature for every unique visited state-action pair.[1]

For the FSR, all base features were captured using indicator functions for each dimension independent of the other dimensions. For example, in the pendulum domain, $20 + 20 = 40$ features were used to map each state s to $\phi(s)$.[2]

As for RBFs, for domains with a small number of dimensions, we distributed RBFs uniformly over the full state space while adding a constant feature with value of 1 [*e.g.,* Lagoudakis and Parr, 2003, Josemans, 2009]. For GridWorld, we placed $6 \times 6 = 36$ RBFs, while for Inverted Pendulum we placed $3 \times 3 = 9$ RBFs. For high dimensional problems this approach is unlikely to work if the number of RBFs is limited, hence we searched for a good performing set of 500 randomly placed RBFs [Hachiya et al., 2008].

[1] We did not apply this approach for value iteration and policy iteration methods to keep them consistent with the literature.

[2] For binary dimensions, the indicator functions were only used for the active values. For example, in the PST domain an indicator function realized $sensor = True$ but not $sensor = False$.

The difficulties we faced in finding a good placement of RBFs over larger domains that do not have a natural distance metric such as Blocksworld and PST re-emphasize the well-known fact that RBFs are not suited for spaces without a well defined distance metric. Despite our efforts, we could not find a set of RBFs with good resulting policies in our large domains (BlocksWorld and PST).

For iFDD, the best relevance threshold[3] was selected out of $\{0.02, 0.05, 0.1\}$ for the Gridworld, $\{0.1, 0.2, 0.5\}$ for Inverted Pendulum, $\{0.02, 0.05, 0.1\}$ for the BlocksWorld, and $\{75, 100, 150\}$ for the PST. For all techniques except LSPI, iFDD used the estimated Bellman error on each step for feature expansion (for RL techniques this estimate is based on one sample, while for DP methods more samples were used). For LSPI, iFDD expanded the feature with the highest TD-error over all samples and then reran LSPI. This loop continued until the TD-error for all potential features dropped below the relevance threshold.

4.3.2 Exploration and Learning Schedules

For RL techniques the learning rate took the following form

$$\alpha_t = \frac{\alpha_0}{k_t} \frac{N_0 + 1}{N_0 + \text{Episode\#}^{1.1}},$$

as used in the work of Boyan [1999] and Geramifard et al. [2011] in which k_t is the number of non-zero features at time step t, acting as a normalizer [Bradtke and Barto, 1996]. The best parameters were empirically found in the set $\alpha_0 \in \{0.1, 1\}$ and $N_0 \in \{100, 1000, 10^6\}$ for each algorithm, domain, and representation. For gradient-descent steps of TBVI, α was set to 1 and .1 for tabular and non-tabular representations respectively. Policies for both DP and RL were generated using an ϵ-greedy policy with $\epsilon = 0.1$, with the exception of the performance run (when the quality of the policy was probed) where ϵ was set to 0. We present results for the best parameter settings found in our experiments. Since many parameter settings will be unstable, this comparison allows us to judge the best performances of each algorithm and representation, and see their shortcomings even when all other factors are optimized.

[3] A constant used to determine if adding a feature would be beneficial for the representation [see Geramifard et al., 2011].

4.3.3 Algorithm Parameters

For both DP and RL techniques, $\boldsymbol{\theta}$ was initialized with the zero vector. For policy iteration, π was initialized to uniform randomly sampled actions. For the DP algorithms, we limit the computation time at 3 hours and set the convergence threshold $\eta = 10^{-3}$, stopping whenever one of these conditions was met.[4] For SARSA and Q-Learning, we used 10^5 samples. For LSPI the number of samples was capped at 10^4, where every 10^3 steps LSPI updated the policy and added 10^3 samples to the previous samples. We also ran LSPI in the case where the previous data was forgotten but the results were inferior. The discount factor was set to $\gamma = \{0.9, 0.95, 1, 0.9\}$ for the GridWorld, Inverted Pendulum, BlocksWorld, and PST domains respectively. On every run, the total return and the corresponding standard error were calculated based on 30 evaluations with 30 fixed random seeds across all combinations of algorithm/domain/representation. Episodes were capped at 1000 for all domains except for Inverted Pendulum, which was capped at 3000. For TBVI, the transition model was approximated with 100 next states (*i.e.,*$L_1 = 100$). For Inverted Pendulum, next states were generated by first sampling 10 states uniform randomly within the cell and then sampling 10 next states from each of those states. All samples were cached and reused for each DP method. In the GridWorld and BlocksWorld domains, due to small number of possible next states, the expectation was performed exactly. The $\boldsymbol{A}$ matrix was regularized by $\psi = 10^{-6}$. In other words, $\boldsymbol{A\theta} = \boldsymbol{b}$ was solved as $\boldsymbol{\theta} = (\boldsymbol{A}^T\boldsymbol{A} + \psi I)^{-1}\boldsymbol{A}^T\boldsymbol{b}$. The number of LSPI iterations was capped at 5.

4.4 Simulation Results

Our evaluation presents a comparison between various DP and RL techniques both in terms of performance and computation time based on 4 domains and 4 representations. All algorithms were evaluated with the representations described in Subsection 4.3.1, except for policy iteration and value iteration, which can only be applied to the tabular representation.

[4] TBVI assumed convergence if the maximum update for 5 consecutive trajectory was below η.

To check the performance of each algorithm, a set of states from a distribution can be sampled, and the policy to be evaluated can be run from each state with the average returns recorded (the Monte Carlo technique [*e.g.,* see Sutton and Barto, 1998]). The resulting weighted sum is indicative of the performance. However, in the literature, to simplify the simulation process and provide more intuitive results, the sum of undiscounted rewards for the initial state or the episode lengths are used as proxies [Sutton, 1996, Lagoudakis and Parr, 2003, Stone et al., 2005a, Jung and Stone, 2010]. This tutorial follows the same approach.

For each domain, there are two tables. The first table contains the mean cumulative reward obtained over 30 evaluation runs with the standard error highlighting the 95% confidence interval. The second table for each domain lists the average wall clock time required to reach 95% of the final performance reported in the first table for each algorithm. The results were all recorded under the same condition using a cluster where a single core was allocated to execute each run. Hence the wall clock time was measuring CPU time plus a constant shared among all runs. [5] The values in this table characterize the *speed* of our particular implementation of the algorithm to enlighten practitioners as to the relative speeds of the different techniques on common benchmark problems. These timing results can, of course, change with different implementations. For a theoretical comparison of the algorithm runtimes, the complexity results reported earlier are the worst case bounds.

4.4.1 Gridworld

In this domain, all DP and RL algorithms converged to the optimal policy using a tabular representation (Table 4.1). All algorithms using FSR performed poorly because it did not have enough representational power to capture the value function. However, the good performance obtained with iFDD indicates that iFDD was able to build a useful representation online starting from the FSR representation. RBF based methods could solve the task as well, yet due to the representation limitation could not fine tune the policy as well as tabular and iFDD techniques. In this domain, LSPI with RBFs did not perform well. We suspect that this is due to accumulated data gathered with various poli-

[5] We also used the CPU measure for some of the runs and observed the same trend among the results based on the wall clock time.

Table 4.1: Final Performance of various DP and RL methods in the GridWorld domain.

Algorithm/Rep.	tabular	FSR	RBFs	iFDD
Policy Iteration	$+0.973 \pm .001$	N/A	N/A	N/A
Value Iteration	$+0.976 \pm .001$	N/A	N/A	N/A
TBVI	$+0.973 \pm .001$	$-0.7 \pm .1$	$+0.84 \pm .09$	$+0.975 \pm .001$
SARSA	$+0.971 \pm .001$	$-1.0 \pm .0$	$+0.77 \pm .11$	$+0.973 \pm .001$
Q-Learning	$+0.973 \pm .001$	$+0.1 \pm .2$	$+0.91 \pm .07$	$+0.973 \pm .001$
LSPI	$+0.972 \pm .001$	$+0.3 \pm .2$	-0.42 ± 0.16	$+0.7 \pm .1$

Table 4.2: Time (sec) required to reach 95% of final performance for various DP and RL methods in the GridWorld domain.

Algorithm/Rep.	tabular	FSR	RBFs	iFDD
Policy Iteration	$5.1 \pm .2$	N/A	N/A	N/A
Value Iteration	$\mathbf{2.04 \pm .01}$	N/A	N/A	N/A
TBVI	5.2 ± 0.3	9 ± 4	39 ± 6	50 ± 7
SARSA	$\mathbf{10.5 \pm 0.6}$	9.2 ± 0.6	88 ± 11	$\mathbf{10 \pm 1}$
Q-Learning	21 ± 1	40 ± 6	62 ± 9	18.7 ± 0.9
LSPI	36 ± 2	25 ± 3	44 ± 8	197 ± 46

cies together with limited representation power of the RBFs. Notice that the choice of the representation played a major role in the performance compared to the choice of algorithm. Table 4.2 shows the timing results. Among DP algorithms, value iteration provided the answer in the shortest time. Among RL techniques, SARSA using tabular and iFDD techniques reached good policies in the shortest time. For all algorithms the running time of RBFs is greater than all other representations. We suspect this observation is due to real feature values of RBFs compared to all others cases where feature values are binary. Hence calculating $\phi(s,a)^\top \boldsymbol{\theta}$ involves multiplication of real numbers for RBFs as opposed to other representations.

4.4.2 Inverted Pendulum

Table 4.3 reports the number of steps each algorithm/representation could balance the pendulum. All of the DP techniques using the tabular representa-

Table 4.3: Final Performance of various DP and RL methods in the Inverted Pendulum domain.

Algorithm/Representation	tabular	FSR	RBFs	iFDD
Policy Iteration	3000 ± 0	N/A	N/A	N/A
Value Iteration	3000 ± 0	N/A	N/A	N/A
TBVI	3000 ± 0	105 ± 31	1466 ± 249	2909 ± 88
SARSA	2892 ± 85	1585 ± 223	1755 ± 247	3000 ± 0
Q-Learning	3000 ± 0	1156 ± 232	1782 ± 248	3000 ± 0
LSPI	3000 ± 0	94 ± 24	2807 ± 131	2725 ± 150

Table 4.4: Time (sec) required to converge to 95% of final performance for various DP and RL methods in the Inverted Pendulum domain.

Algorithm/Representation	tabular	FSR	RBFs	iFDD
Policy Iteration	464 ± 23	N/A	N/A	N/A
Value Iteration	176 ± 2	N/A	N/A	N/A
TBVI	$\mathbf{129 \pm 5}$	0.5 ± 0.02	2428 ± 513	372 ± 33
SARSA	25 ± 4	14 ± 2	42 ± 6	34 ± 1
Q-Learning	$\mathbf{17.1 \pm 0.8}$	21 ± 4	76 ± 11	$\mathbf{16.7 \pm 0.8}$
LSPI	23 ± 1	3.5 ± 0.5	$\mathbf{10 \pm 1}$	487 ± 38

tion found optimal policies, balancing the pendulum for the entire 3000 steps. By contrast, TBVI using FSR performed poorly. Using RBFs boosted the resulting performance by 10 fold over FSR because the RBFs yield a richer representation than FSR. The switch from RBFs to iFDD resulted in over a 100% improvement. Among RL techniques, we can see the same trend akin to DP techniques. The only exception is LSPI using RBFs and iFDD, where the difference is not statistically significant.

Table 4.4 shows the time required for all the algorithms to reach 95% of their final performance. As opposed to the previous domain where the number of possible next states for every state-action pair was 4, in this domain there is an infinite number of possible next states. Hence 100 samples (which empirically seemed to cover next states appropriately) were used to estimate the Bellman backup, increasing the computational complexity of all

Table 4.5: Final Performance of various DP and RL methods in the BlocksWorld domain.

Algorithm/Rep.	tabular	FSR	RBFs	iFDD
Policy Iteration	-0.87 ± 0.07	N/A	N/A	N/A
Value Iteration	-0.94 ± 0.06	N/A	N/A	N/A
TBVI	0.7 ± 0.1	-1.00 ± 0.00	-1.00 ± 0.00	$\mathbf{0.990 \pm 0.001}$
SARSA	0.7 ± 0.1	-0.93 ± 0.07	-1.00 ± 0.00	$\mathbf{0.93 \pm 0.07}$
Q-Learning	$\mathbf{0.93 \pm 0.07}$	-0.93 ± 0.07	-1.00 ± 0.00	$0.8 + 0.1$
LSPI	N/A	-1.00 ± 0.00	N/A	N/A

Table 4.6: Time (sec) required to converge to 95% of final performance for various DP and RL methods in the BlocksWorld domain.

Algorithm/Rep.	tabular	FSR	RBFs	iFDD
Policy Iteration	7324 ± 6	N/A	N/A	N/A
Value Iteration	7466 ± 16	N/A	N/A	N/A
TBVI	3836 ± 440	9.3 ± 0.2	41.5 ± 0.1	676 ± 101
SARSA	1444 ± 172	43 ± 11	254 ± 0.7	$\mathbf{141 \pm 14}$
Q-Learning	2622 ± 218	65 ± 12	665 ± 12	209 ± 23
LSPI	N/A	203 ± 6	N/A	N/A

the DP techniques.[6] Overall TBVI using the tabular representation achieved the best performance in the shortest time. For RL techniques, LSPI with RBFs achieved good policies in only 10 seconds. Among RL algorithms with optimal policies, Q-Learning with the tabular or iFDD representations was the fastest.

4.4.3 BlocksWorld

Table 4.5 depicts the final performance of the DP and RL techniques in the BlocksWorld domain. Notice that as the size of the planning space increased to over one million state-action pairs, most DP techniques had difficulty finding good policies within 3 hours of computation using a tabular representation. Among DP techniques using a tabular representation, only TBVI found

[6] To boost the calculation of Bellman backups, all s' were cached and reused.

a good policy. The FSR representation is not suitable for solving this problem as it ignores critical feature dependencies needed to solve the task of making a tower with a specific ordering of the blocks. We also noticed that TBVI diverged using this representation. Among 30 trials of randomly selecting 500 RBFs, none yielded good results. This is expected as RBFs are often used in domains with metric continuous state spaces, which is not the case in BlocksWorld. Finally iFDD helped TBVI, reaching the best performing policy.

A similar pattern is visible among the RL techniques. The best performances were achieved with Q-Learning using a tabular representation and SARSA using iFDD. LSPI became intractable when calculating the $\boldsymbol{A}$ matrix using any representation other than the FSR, with which it performed poorly.

The corresponding timing results to reach within 95% of the final performances are shown in Table 4.6. As expected for TBVI, SARSA and Q-Learning, using iFDD resulted in at least 5-times speed boost compared to the tabular representation. This is due to the fact the iFDD gradually expands the representation, resulting in good generalization, without representing every feature combination. Such effects are not as pronounced in the GridWorld and Inverted Pendulum domains, since the size of the tabular representation was at most 1200.

4.4.4 Persistent Search and Track

The large size of the PST domain makes it a challenging problem for both DP and RL methods. Table 4.7 shows the performance of all the algorithms in this environment. Using the tabular representation both policy iteration and value iteration reached poor policies resulting in crashing scenarios. TBVI reached a sub-optimal policy of holding UAVs in the base, achieving 0 return. TBVI using FSR again diverged, resulting in crashing scenarios. Again our search for a suitable set of RBFs for TBVI did not lead to any improvement over the tabular representation. Using iFDD also could not improve the performance of TBVI compared to using RBFs. We believe that iFDD was not effective because it only expands features corresponding to the states visited along the trajectory. Hence if the values of next states sampled to estimate the Bellman backup that are not along the trajectory are not accurate, iFDD does not

Table 4.7: Final Performance of various DP and RL methods in the Persistent Search and Track domain.

Algorithm/Representation	tabular	FSR	RBFs	iFDD
Policy Iteration	-66 ± 5	N/A	N/A	N/A
Value Iteration	-51 ± 6	N/A	N/A	N/A
TBVI	0 ± 0.0	-54 ± 8	-2 ± 2	-12 ± 9
SARSA	0 ± 0.0	79 ± 28	-5 ± 3	$\mathbf{328 \pm 77}$
Q-Learning	0 ± 0.0	106 ± 49	-6 ± 3	$\mathbf{639 \pm 218}$
LSPI	N/A	-66 ± 9	N/A	N/A

Table 4.8: Time (min) required to converge to 95% of final performance for various DP and RL methods in the Persistent Search and Track domain.

Algorithm/Representation	tabular	FSR	RBFs	iFDD
Policy Iteration	10481 ± 286	N/A	N/A	N/A
Value Iteration	8521 ± 100	N/A	N/A	N/A
TBVI	$\mathbf{24 \pm 1}$	758 ± 262	192 ± 60	1676 ± 362
SARSA	359 ± 5	434 ± 28	1961 ± 390	$\mathbf{625 \pm 32}$
Q-Learning	425 ± 9	536 ± 32	1066 ± 224	$\mathbf{921 \pm 51}$
LSPI	N/A	4203 ± 316	N/A	N/A

expand the representation to reduce those inaccuracies. In other words, applying iFDD with TBVI in domains with a large branching factor in $\mathcal{P}$ (in this case $4^3 = 64$) can be problematic. This phenomenon was not a seen in the BlocksWorld, because the branching factor was 2.

For RL techniques, the tabular representation led to the sub-optimal policy of holding UAVs at the base. FSR led to better policies with positive returns. We suspect this mismatch compared to DP techniques is due to the fact that RL techniques used the representation only to evaluate states along the trajectory not for states that were one step away. Hence they take better advantage of the limited representational capacity. Similarly, the LSPI algorithm tried to fit the limited representational power of FSR over all samples, which led to poor policies. The use of RBFs resulted in the policy of holding UAVs at the base most of the time. Finally, iFDD in this domain found im-

portant features allowing Q-Learning to find the best policies discovered by any of the algorithms. SARSA followed Q-Learning, outperforming all other techniques. Note that for online RL techniques the branching factor did not hinder the applicability of iFDD because states used to estimate the Bellman backup were also considered for feature expansion.

Table 4.8 reports the corresponding timing results. Among DP techniques, TBVI using the tabular representation found policies of holding UAVs in less than 30 seconds. Within RL techniques, both SARSA and Q-Learning could find good policies using iFDD within ~ 15 minutes.

4.5 Discussion

The most stark observation from our experiments is that the choice of the representation can often play a much more significant role in the final performance of the solver than the choice of the algorithm. In particular, the most accurate representation (*i.e.,* the tabular representation) often led to the best final performance for smaller domains but in larger domains it became intractable for both planning and learning.

In the larger domains, linear function approximation demonstrated its main benefit, making both the DP and RL algorithms tractable and still resulting in good performance, but these results strongly depend on the features used. In most domains, the simple FSR representation with base features was not capable of capturing the value function, leading to poor policies. We do note that there are other popular ways for creating FSRs with more than base features, such as tile coding, which may yield better performance [Sutton and Barto, 1998, Sutton, 1996]. Adding conjunctions of base features manually to a certain depth also led to good results in the game of Go [Silver et al., 2012].

For RBFs, our experiments illustrated that the performance of DP and RL algorithms with RBFs are highly sensitive to placement and bandwidth choices, and do not work well in spaces that do not contain a natural distance metric such as the BlocksWorld and PST. Our simple method of searching for well placed centers and well picked bandwidths was not able to ensure as good performance for RBFs as the adaptive approach (iFDD). We expect that adaptive RBF placement techniques could help alleviate the issues we faced

with RBFs.

Our experiments show that adaptive representations, like iFDD, can often lead to powerful yet sparse representations and very good performance in large domains like BlocksWorld and PST, where manual feature construction becomes too laborious. Hence, our empirical results reinforce that the use of adaptive representations is very important for successful scaling of MDP solvers to larger domains. Of course, iFDD is just one example of the many adaptive representations available. For example Ratitch and Precup [2004] introduced sparse distributed memories as a process for adaptively adding and adjusting RBFs in visited parts of the state space. Furthermore, the literature on kernel adaptive filtering offers ways of adaptively growing kernel representation [see *e.g.,* Schölkopf and Smola, 1998, Liu et al., 2010], which have been used for solving MDPs [see *e.g.,* Farahmand, 2009].

When the model of the MDP is present, TBVI using iFDD very quickly generated policies that performed well in domains with small branching factors. Value iteration and policy iteration were efficient for smaller domains, but their requirement for considering every possible transition became problematic in larger domains. By contrast, TBVI took advantage of sampling techniques that focus the computation on areas of the state space most visited under the current policy of the agent. Our findings coincide with the results in the literature verifying the advantage of focused sampling [Sutton and Barto, 1998, Silver et al., 2008, Ure et al., 2012].

In terms of sample complexity for the RL techniques, while batch methods are traditionally more sample efficient compared to the online techniques, in our case the complicated relationship between the approximate representations and exploration meant that choosing a memory management scheme for LSPI's data (what samples to keep and which to discard) is non-trivial. Often this meant that the incremental methods (Q-learning and SARSA) were able to outperform LSPI. However, Q-learning and SARSA also processed 10 times the number of samples compared to LSPI. Overall, SARSA and Q-Learning using iFDD were the most efficient among the RL solvers. It is also worth noting that SARSA and Q-learning were the most robust solvers; converging to a solution in all 4 domains using all 4 representations. In summary, the experiments lead to the following recommendations for practitioners:

- Spend time crafting the representation as it can often play a larger role

than algorithm choice for solving MDPs

- Consider adaptive representations for tackling large domains.
- Use algorithms with on-policy sampling for both trajectory-based planning and faster learning.
- Remember that using linear function approximation instead of tabular representation can cause the divergence of DP or RL techniques.

5

Summary

This article reviewed techniques for planning and learning in Markov Decision Processes (MDPs) with linear function approximation of the value function. Two major paradigms for finding optimal policies were considered: dynamic programming (DP) techniques for planning and reinforcement learning (RL). We also showed the deep connections between both paradigms by discussing algorithms that performed planning by sampling from a generative model and form a bridge between techniques from the DP and RL literature (Figure 2.1). Specifically, we showed how trajectory based DP techniques such as TBVI and TBPI are derived from value iteration and policy iteration by trading off accuracy with computational speed. We then described how model-free RL methods like SARSA, Q-Learning, LSTDQ, and LSPI are essentially approximate DP methods for a setting with the following restrictions: **I)** samples can only be gathered in the form of trajectories, **II)** one-step look-ahead of the model parameters (Equation 2.9) cannot be directly accessed, and **III)** the time between interactions is limited. Throughout the DP to RL spectrum, we saw that linear value function approximation allowed variants of these algorithms to scale to large domains where they would be ineffective with a tabular representation.

The presented algorithms were validated on four representative do-

mains with increasing level of complexity: Gridworld, Inverted Pendulum, Blocksworld, and Persistent Search and Track. For the first two small domains, both RL and DP techniques using a tabular representation achieved high quality performance. As the problem sizes grew, the use of a tabular representation became problematic; policy iteration and value iteration yielded poor policies in the next two large domains as they ran out of planning time which was capped at 3 hours. TBVI with the tabular representation could partially address the scalability problem by focusing the updates in important parts of the state space, reaching sub-optimal policies. The performance of SARSA and Q-Learning were similar to TBVI in the tabular setting. The LSPI implementation faced memory problems due to large sizes of matrixes. The FSR representation offered cheap computation, yet it did not offer enough representational power. As a result, all methods with FSR demonstrated fast convergence to noncompetitive policies. The RBF representation provided more powerful representation for the first two domains, but it did not work well for large domains with discrete state dimensions. Finally, the iFDD approach grew the representation as required. It worked well across both RL and DP methods in the first two small domains and led to high performing policies for the last two large domains. The main concern in using iFDD with DP techniques is when the branching factor of the MDP is large (*e.g.,* the PST domain), because in this case most states used to estimate the Bellman backup are not considered for feature expansion. LSPI offered good policies in the first two small domains, using 10 times fewer samples compared to online RL techniques. For example, LSPI reached very good policies in the Inverted Pendulum domain in about 10 seconds using RBFs. By contrast, Q-Learning required 76 seconds to reach a lower tier policy with the same representation. Using LSPI in the last two problems was challenging for all representations except FSR due to LSPI's memory requirement. These results suggest that, in large domains, one should use learning techniques with incremental updates.

The results also highlighted the pivotal role the choice of representation plays in ensuring good performance. Generally, our experiments showed that adaptive representations using iFDD out-performed static representations, both in terms of the effectiveness of the learned policy and the convergence time. This can be attributed to the ability of adaptive representations to overcome incorrect initial feature configurations and find new features that better

represent the value function. However, it is very important to note that a different choice of sparse features or a different number and location of Gaussian RBFs could have resulted in better performance. Unfortunately, it is difficult to choose the right representation *a priori*. A representation that employs a large number of basis functions is not desirable, as it can lead to high sample complexity (regularization can mitigate this effect to some extend [see *e.g.,* Farahmand et al., 2008]). On the other hand, an overly small representation may lead to poor generalization. Adaptive function approximation methods are being actively researched to address this problem by adjusting the basis of the representation in response to the data [*e.g.,* Ahmadi et al., 2007, Girgin and Preux, 2007, Parr et al., 2007, Kolter and Ng, 2009, Geramifard et al., 2011, Kroemer and Peters, 2011, Geramifard et al., 2013b]. The promise of these methods was exemplified by the positive results of iFDD.

In conclusion, linear value function approximation is a powerful tool that allows DP and RL algorithms to be used in domains where enumerating the full tabular state space might be intractable. However, one always has to consider the potential for divergence with these approximate techniques against their potential savings. We have shown how DP and RL algorithms trade-off accuracy for computational tractability when using linear function approximation, and how sensitive the resulting behaviors are to the chosen representation. However, the practical algorithms and empirical successes outlined in this paper form a guide for practitioners trying to weigh computational costs, accuracy requirements, and representational concerns. Decision making in large domains will always be challenging, but with the tools presented here this challenge is not insurmountable.

Acknowledgements

We thank Ronald Parr, Michael Littman, Amir-massoud Farahmand, Joelle Pineau, Jan Peters, and Csaba Szepesvári for their constructive feedback on this article. We also recognize the help of Robert Klein and Kemal Ure with the implementation of RLPy framework and brainstorming around the empirical results.

References

RL competition. http://www.rl-competition.org/, 2012. Accessed: 20/08/2012.

M. Ahmadi, M. E. Taylor, and P. Stone. IFSA: incremental feature-set augmentation for reinforcement learning tasks. In *International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pages 1–8, New York, NY, USA, 2007. ACM. .

A. Antos, R. Munos, and C. Szepesvári. Fitted Q-iteration in continuous action-space MDPs. In *Proceedings of Neural Information Processing Systems Conference (NIPS)*, 2007.

A. Antos, C. Szepesvári, and R. Munos. Learning near-optimal policies with bellman-residual minimization based fitted policy iteration and a single sample path. *Machine Learning*, 71(1):89–129, 2008.

J. Asmuth, L. Li, M. Littman, A. Nouri, and D. Wingate. A Bayesian sampling approach to exploration in reinforcement learning. In *International Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 19–26, Arlington, Virginia, United States, 2009. AUAI Press.

L. C. Baird. Residual algorithms: Reinforcement learning with function approximation. In *ICML*, pages 30–37, 1995.

A. d. M. S. Barreto and C. W. Anderson. Restricted gradient-descent algorithm for value-function approximation in reinforcement learning. *Artificial Intelligence*, 172:454 – 482, 2008.

A. Barto and M. Duff. Monte carlo matrix inversion and reinforcement learning. In *Neural Information Processing Systems (NIPS)*, pages 687–694. Morgan Kaufmann, 1994.

A. Barto, S. Bradtke, and S. Singh. Learning to act using real-time dynamic programming. *Artificial Intelligence*, 72:81–138, 1995.

J. Baxter and P. Bartlett. Direct gradient-based reinforcement learning. In *Circuits and Systems, 2000. Proceedings. ISCAS 2000 Geneva. The 2000 IEEE International Symposium on*, volume 3, pages 271–274. IEEE, 2000.

R. E. Bellman. *Dynamic Programming*. Princeton University Press, Princeton, NJ, 1957.

D. Bertsekas and J. Tsitsiklis. *Neuro-Dynamic Programming*. Athena Scientific, Belmont, MA, 1996.

D. P. Bertsekas. *Dynamic Programming and Optimal Control*. AP, 1976.

D. P. Bertsekas and J. N. Tsitsiklis. *Neuro-Dynamic Programming (Optimization and Neural Computation Series, 3)*. Athena Scientific, May 1996.

B. Bethke and J. How. Approximate Dynamic Programming Using Bellman Residual Elimination and Gaussian Process Regression. In *American Control Conference (ACC)*, St. Louis, MO, 2009.

S. Bhatnagar, R. S. Sutton, M. Ghavamzadeh, and M. Lee. Incremental natural actor-critic algorithms. In J. C. Platt, D. Koller, Y. Singer, and S. T. Roweis, editors, *Advances in Neural Information Processing Systems (NIPS)*, pages 105–112. MIT Press, 2007.

M. Bowling, A. Geramifard, and D. Wingate. Sigma Point Policy Iteration. In *International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, volume 1, pages 379–386, Richland, SC, 2008.

J. Boyan and A. Moore. Generalization in reinforcement learning: Safely approximating the value function. In G. Tesauro, D. Touretzky, and T. Lee, editors, *Neural Information Processing Systems (NIPS)*, pages 369–376, Cambridge, MA, 1995. The MIT Press.

J. A. Boyan. Least-squares temporal difference learning. In *International Conference on Machine Learning (ICML)*, pages 49–56. Morgan Kaufmann, San Francisco, CA, 1999.

S. J. Bradtke and A. G. Barto. Linear least-squares algorithms for temporal difference learning. *Journal of Machine Learning Research (JMLR)*, 22:33–57, 1996.

R. Brafman and M. Tennenholtz. R-Max - A General Polynomial Time Algorithm for Near-Optimal Reinforcement Learning. *Journal of Machine Learning Research (JMLR)*, 3:213–231, 2002.

L. Buşoniu, R. Babuška, B. De Schutter, and D. Ernst. *Reinforcement Learning and Dynamic Programming Using Function Approximators*. CRC Press, Boca Raton, Florida, 2010.

T. G. Dietterich. Hierarchical reinforcement learning with the MAXQ value function decomposition. *Journal of Artificial Intelligence and Research (JAIR)*, 13(1):227–303, Nov. 2000.

A. Dutech, T. Edmunds, J. Kok, M. Lagoudakis, M. Littman, M. Riedmiller, B. Russell, B. Scherrer, R. Sutton, S. Timmer, et al. Reinforcement learning benchmarks and bake-offs II. In *Advances in Neural Information Processing Systems (NIPS) 17 Workshop*, 2005.

Y. Engel, S. Mannor, and R. Meir. Bayes meets bellman: The gaussian process approach to temporal difference learning. In *International Conference on Machine Learning (ICML)*, pages 154–161, 2003.

A. Farahmand. *Regularities in Sequential Decision-Making Problems*. PhD thesis, Department of Computing Science, University of Alberta, 2009.

A. Farahmand, M. Ghavamzadeh, C. Szepesvári, and S. Mannor. Regularized policy iteration. In D. Koller, D. Schuurmans, Y. Bengio, and L. Bottou, editors, *Advances in Neural Information Processing Systems (NIPS)*, pages 441–448. MIT Press, 2008.

P. Geibel and F. Wysotzki. Risk-sensitive reinforcement learning applied to chance constrained control. *Journal of Artificial Intelligence and Research (JAIR)*, 24, 2005.

A. Geramifard, F. Doshi, J. Redding, N. Roy, and J. How. Online discovery of feature dependencies. In L. Getoor and T. Scheffer, editors, *International Conference on Machine Learning (ICML)*, pages 881–888. ACM, June 2011.

A. Geramifard, J. Redding, J. Joseph, N. Roy, and J. P. How. Model estimation within planning and learning. In *American Control Conference (ACC)*, June 2012.

A. Geramifard, R. H. Klein, and J. P. How. RLPy: The Reinforcement Learning Library for Education and Research. `http://acl.mit.edu/RLPy`, April 2013a.

A. Geramifard, T. J. Walsh, N. Roy, and J. How. Batch iFDD: A Scalable Matching Pursuit Algorithm for Solving MDPs. In *Proceedings of the 29th Annual Conference on Uncertainty in Artificial Intelligence (UAI)*, Bellevue, Washington, USA, 2013b. AUAI Press.

S. Girgin and P. Preux. Feature Discovery in Reinforcement Learning using Genetic Programming. Research Report RR-6358, INRIA, 2007.

G. H. Golub and C. F. V. Loan. *Matrix Computations*. The John Hopkins University Press, 1996.

G. Gordon. Stable function approximation in dynamic programming. In *International Conference on Machine Learning (ICML)*, page 261, Tahoe City, California, July 9-12 1995. Morgan Kaufmann.

A. Gosavi. Reinforcement learning: A tutorial survey and recent advances. *INFORMS J. on Computing*, 21(2):178–192, April 2009.

H. Hachiya, T. Akiyama, M. Sugiyama, and J. Peters. Adaptive importance sampling with automatic model selection in value function approximation. In *Association for the Advancement of Artificial Intelligence (AAAI)*, pages 1351–1356, 2008.

S. Haykin. *Neural Networks: A Comprehensive Foundation*. Macmillan, New York, 1994.

R. A. Howard. *Dynamic Programming and Markov Processes*. MIT Press, Cambridge, Massachusetts, 1960.

T. Jaakkola, M. Jordan, and S. Singh. on the convergence of stochastic iterative dynamic programming algorithms. Technical report, Massachusetts Institute of Technology, Cambridge, MA, August 1993.

T. Jaksch, R. Ortner, and P. Auer. Near-optimal regret bounds for reinforcement learning. *Journal of Machine Learning Research (JMLR)*, 11:1563–1600, 2010.

W. Josemans. *Generalization in Reinforcement Learning*. PhD thesis, University of Amsterdam, 2009.

T. Jung and P. Stone. Gaussian processes for sample efficient reinforcement learning with RMAX-like exploration. In *European Conference on Machine Learning (ECML)*, September 2010.

L. P. Kaelbling, M. L. Littman, and A. W. Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence and Research (JAIR)*, 4:237–285, 1996.

S. Kalyanakrishnan and P. Stone. Characterizing reinforcement learning methods through parameterized learning problems. *Machine Learning*, 2011.

J. Z. Kolter and A. Y. Ng. Regularization and feature selection in least-squares temporal difference learning. In *International Conference on Machine Learning (ICML)*, pages 521–528, New York, NY, USA, 2009. ACM.

R. Kretchmar and C. Anderson. Comparison of cmacs and radial basis functions for local function approximators in reinforcement learning. In *International Conference on Neural Networks*, volume 2, pages 834–837 vol.2, 1997.

O. Kroemer and J. Peters. A non-parametric approach to dynamic programming. In *Advances in Neural Information Processing Systems (NIPS)*, pages 1719–1727, 2011.

L. Kuvayev and R. Sutton. Model-based reinforcement learning with an approximate, learned model. In *Proceeding of the ninth Yale workshop on adaptive and learning systems*, pages 101–105, 1996.

M. G. Lagoudakis and R. Parr. Least-squares policy iteration. *Journal of Machine Learning Research (JMLR)*, 4:1107–1149, 2003.

L. Li. Sample complexity bounds of exploration. In M. Wiering and M. van Otterlo, editors, *Reinforcement Learning: State of the Art*. Springer Verlag, 2012.

L. Li, M. L. Littman, and C. R. Mansley. Online exploration in least-squares policy iteration. In *International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pages 733–739, Richland, SC, 2009a. International Foundation for Autonomous Agents and Multiagent Systems.

L. Li, J. D. Williams, and S. Balakrishnan. Reinforcement learning for dialog management using least-squares policy iteration and fast feature selection. In *New York Academy of Sciences Symposium on Machine Learning*, 2009b.

W. Liu, P. Pokharel, and J. Principe. The kernel least-mean-square algorithm. *IEEE Transactions on Signal Processing*, 56(2):543–554, 2008.

W. Liu, J. C. Principe, and S. Haykin. *Kernel Adaptive Filtering: A Comprehensive Introduction*. Wiley, Hoboken, New Jersey, 2010.

H. R. Maei and R. S. Sutton. GQ(λ): A general gradient algorithm for temporal-difference prediction learning with eligibility traces. In *Proceedings of the Third Conference on Artificial General Intelligence (AGI)*, Lugano, Switzerland, 2010.

H. R. Maei, C. Szepesvári, S. Bhatnagar, and R. S. Sutton. Toward off-policy learning control with function approximation. In J. Fürnkranz and T. Joachims, editors, *International Conference on Machine Learning (ICML)*, pages 719–726. Omnipress, 2010.

S. Mahadevan. Representation policy iteration. *International Conference on Uncertainty in Artificial Intelligence (UAI)*, 2005.

Mausam and A. Kolobov. *Planning with Markov Decision Processes: An AI Perspective*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2012.

F. S. Melo, S. P. Meyn, and M. I. Ribeiro. An analysis of reinforcement learning with function approximation. In *International Conference on Machine Learning (ICML)*, pages 664–671, 2008.

O. Mihatsch and R. Neuneier. Risk-sensitive reinforcement learning. *Journal of Machine Learning Research (JMLR)*, 49(2-3):267–290, 2002.

J. Moody and C. J. Darken. Fast learning in networks of locally-tuned processing units. *Neural Computation*, 1(2):281–294, June 1989.

A. W. Moore and C. G. Atkeson. Prioritized sweeping: Reinforcement learning with less data and less time. In *Machine Learning*, pages 103–130, 1993.

A. Nouri and M. L. Littman. Multi-resolution exploration in continuous spaces. In D. Koller, D. Schuurmans, Y. Bengio, and L. Bottou, editors, *Advances in Neural Information Processing Systems (NIPS)*, pages 1209–1216. MIT Press, 2009.

R. Parr, C. Painter-Wakefield, L. Li, and M. Littman. Analyzing feature generation for value-function approximation. In *International Conference on Machine Learning (ICML)*, pages 737–744, New York, NY, USA, 2007. ACM.

R. Parr, L. Li, G. Taylor, C. Painter-Wakefield, and M. L. Littman. An analysis of linear models, linear value-function approximation, and feature selection for reinforcement learning. In *International Conference on Machine Learning (ICML)*, pages 752–759, New York, NY, USA, 2008. ACM.

J. Peters and S. Schaal. Policy gradient methods for robotics. In *2006 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2219–2225. IEEE, October 2006.

J. Peters and S. Schaal. Natural actor-critic. *Neurocomputing*, 71:1180–1190, March 2008.

M. Petrik, G. Taylor, R. Parr, and S. Zilberstein. Feature selection using regularization in approximate linear programs for Markov decision processes. In *International Conference on Machine Learning (ICML)*, 2010.

M. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Wiley, 1994.

C. Rasmussen and C. Williams. *Gaussian Processes for Machine Learning*. MIT Press, Cambridge, MA, 2006.

B. Ratitch and D. Precup. Sparse distributed memories for on-line value-based reinforcement learning. In *European Conference on Machine Learning (ECML)*, pages 347–358, 2004.

M. Riedmiller, J. Peters, and S. Schaal. Evaluation of policy gradient methods and variants on the Cart-Pole benchmark. In *IEEE International Symposium on Approximate Dynamic Programming and Reinforcement Learning (ADPRL)*, pages 254–261, April 2007.

G. A. Rummery and M. Niranjan. Online Q-learning using connectionist systems (tech. rep. no. cued/f-infeng/tr 166). *Cambridge University Engineering Department*, 1994.

S. Sanner. International Probabilistic Planning Competition (IPPC) at International Joint Conference on Artificial Intelligence (IJCAI). http://users.cecs.anu.edu.au/~ssanner/IPPC_2011/, 2011. Accessed: 26/09/2012.

B. Scherrer. Should one compute the temporal difference fix point or minimize the bellman residual? the unified oblique projection view. In *International Conference on Machine Learning (ICML)*, 2010.

B. Schölkopf and A. Smola. Nonlinear component analysis as a kernel eigenvalue problem. *Neural Computations*, 10(5):1299–1319, 1998.

B. Schölkopf and A. Smola. *Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond*. MIT Press, Cambridge, MA, 2002.

P. Schweitzer and A. Seidman. Generalized polynomial approximation in Markovian decision processes. *Journal of mathematical analysis and applications*, 110:568–582, 1985.

D. Silver, R. S. Sutton, and M. Müller. Sample-based learning and search with permanent and transient memories. In *International Conference on Machine Learning (ICML)*, pages 968–975, New York, NY, USA, 2008. ACM.

D. Silver, R. S. Sutton, and M. Müller. Temporal-difference search in computer go. *Machine Learning*, 87(2):183–219, 2012.

S. P. Singh. Reinforcement learning with a hierarchy of abstract models. In *Proceeding of the Tenth National Conference on Artificial Intelligence*, pages 202–207. MIT/AAAI Press, 1992.

S. P. Singh, T. Jaakkola, M. L. Littman, and C. Szepesvári. Convergence results for single-step on-policy reinforcement-learning algorithms. *Journal of Machine Learning Research (JMLR)*, 38:287–308, 2000.

P. Stone, R. S. Sutton, and G. Kuhlmann. Reinforcement learning for RoboCup-soccer keepaway. *International Society for Adaptive Behavior*, 13(3):165–188, 2005a.

P. Stone, R. S. Sutton, and G. Kuhlmann. Reinforcement learning for RoboCup soccer keepaway. *Adaptive Behavior*, 13(3):165–188, September 2005b.

A. L. Strehl, L. Li, and M. L. Littman. Reinforcement learning in finite mdps: Pac analysis. *Journal of Machine Learning Research (JMLR)*, 10:2413–2444, Dec. 2009.

R. S. Sutton. Generalization in reinforcement learning: Successful examples using sparse coarse coding. In *Neural Information Processing Systems (NIPS)*, pages 1038–1044. The MIT Press, 1996.

R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 1998.

R. S. Sutton, D. McAllester, S. Singh, and Y. Mansour. Policy gradient methods for reinforcement learning with function approximation. *Advances in Neural Information Processing Systems (NIPS)*, 12(22):1057–1063, 2000.

R. S. Sutton, H. R. Maei, D. Precup, S. Bhatnagar, D. Silver, C. Szepesvári, and E. Wiewiora. Fast gradient-descent methods for temporal-difference learning with linear function approximation. In *International Conference on Machine Learning (ICML)*, pages 993–1000, New York, NY, USA, 2009. ACM.

C. Szepesvári. *Algorithms for Reinforcement Learning*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2010.

I. Szita and C. Szepesvári. Model-based reinforcement learning with nearly tight exploration complexity bounds. In *International Conference on Machine Learning (ICML)*, pages 1031–1038, 2010.

G. Taylor and R. Parr. Kernelized value function approximation for reinforcement learning. In *International Conference on Machine Learning (ICML)*, pages 1017–1024, New York, NY, USA, 2009. ACM.

J. N. Tsitsiklis and B. V. Roy. An analysis of temporal difference learning with function approximation. *IEEE Transactions on Automatic Control*, 42(5):674–690, May 1997.

J. N. Tsitsiklis and B. V. Roy. Average cost temporal-difference learning. *Automatica*, 35(11):1799 – 1808, 1999.

N. K. Ure, A. Geramifard, G. Chowdhary, and J. P. How. Adaptive Planning for Markov Decision Processes with Uncertain Transition Models via Incremental Feature Dependency Discovery. In *European Conference on Machine Learning (ECML)*, 2012.

C. J. Watkins. *Models of Delayed Reinforcement Learning*. PhD thesis, Cambridge Univ., 1989.

C. J. Watkins. Q-learning. *Machine Learning*, 8(3):279–292, 1992.

C. J. C. H. Watkins and P. Dayan. Q-learning. *Machine Learning*, 8(3):279–292, May 1992.

S. D. Whitehead. A complexity analysis of cooperative mechanisms in reinforcement learning. In *Association for the Advancement of Artificial Intelligence (AAAI)*, pages 607–613, 1991.

S. Whiteson and M. Littman. Introduction to the special issue on empirical evaluations in reinforcement learning. *Machine Learning*, pages 1–6, 2011.

B. Widrow and F. Smith. Pattern-recognizing control systems. In *Computer and Information Sciences: Collected Papers on Learning, Adaptation and Control in Information Systems, COINS symposium proceedings*, volume 12, pages 288–317, Washington DC, 1964.

R. J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. In *Machine Learning*, pages 229–256, 1992.

T. Winograd. Procedures as a representation for data in a computer program for understanding natural language. Technical Report 235, Massachusetts Institute of Technology, 1971.

Y. Ye. The simplex and policy-iteration methods are strongly polynomial for the markov decision problem with a fixed discount rate. *Math. Oper. Res.*, 36(4): 593–603, 2011.

H. Yu and D. P. Bertsekas. Error bounds for approximations from projected linear equations. *Math. Oper. Res.*, 35(2):306–329, 2010.