

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

Alborz Geramifard

Massachusetts Institute of Technology

Thomas J. Walsh

Massachusetts Institute of Technology

Stefanie Tellex

Massachusetts Institute of Technology

Girish Chowdhary

Massachusetts Institute of Technology

Nicholas Roy

Massachusetts Institute of Technology

Jonathan P. How

Massachusetts Institute of Technology

now

the essence of knowledge

Boston — Delft



Documents

Document Sharing Hub

Foundations and Trends[®] in Machine Learning

Published, sold and distributed by:

now Publishers Inc.
PO Box 1024
Hanover, MA 02339
United States
Tel. +1-781-985-4510
www.nowpublishers.com
sales@nowpublishers.com

Outside North America:

now Publishers Inc.
PO Box 179
2600 AD Delft
The Netherlands
Tel. +31-6-51115274

The preferred citation for this publication is

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. 375–454, 2013.

This Foundations and Trends[®] issue was typeset in L^AT_EX using a class file designed by Neal Parikh. Printed on acid-free paper.

ISBN: 978-1-60198-760-0

© 2013 A. Geramifard, T. J. Walsh, S. Tellex, G. Chowdhary, N. Roy, and J. P. How

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, mechanical, photocopying, recording or otherwise, without prior written permission of the publishers.

Photocopying. In the USA: This journal is registered at the Copyright Clearance Center, Inc., 222 Rosewood Drive, Danvers, MA 01923. Authorization to photocopy items for internal or personal use, or the internal or personal use of specific clients, is granted by now Publishers Inc for users registered with the Copyright Clearance Center (CCC). The 'services' for users can be found on the internet at: www.copyright.com

For those organizations that have been granted a photocopy license, a separate system of payment has been arranged. Authorization does not extend to other kinds of copying, such as that for general distribution, for advertising or promotional purposes, for creating new collective works, or for resale. In the rest of the world: Permission to photocopy must be obtained from the copyright owner. Please apply to now Publishers Inc., PO Box 1024, Hanover, MA 02339, USA; Tel. +1 781 871 0245; www.nowpublishers.com; sales@nowpublishers.com

now Publishers Inc. has an exclusive license to publish this material worldwide. Permission to use this content must be obtained from the copyright license holder. Please apply to now Publishers, PO Box 179, 2600 AD Delft, The Netherlands, www.nowpublishers.com; e-mail: sales@nowpublishers.com

Foundations and Trends® in Machine Learning

Volume 6, Issue 4, 2013

Editorial Board

Editor-in-Chief

Michael Jordan

University of California, Berkeley
United States

Editors

Peter Bartlett

UC Berkeley

Yoshua Bengio

University of Montreal

Avrim Blum

CMU

Craig Boutilier

University of Toronto

Stephen Boyd

Stanford University

Carla Brodley

Tufts University

Inderjit Dhillon

UT Austin

Jerome Friedman

Stanford University

Kenji Fukumizu

ISM, Japan

Zoubin Ghahramani

University of Cambridge

David Heckerman

Microsoft Research

Tom Heskes

Radboud University

Geoffrey Hinton

University of Toronto

Aapo Hyvarinen

HIIT, Finland

Leslie Pack Kaelbling

MIT

Michael Kearns

UPenn

Daphne Koller

Stanford University

John Lafferty

CMU

Michael Littman

Brown University

Gabor Lugosi

Pompeu Fabra University

David Madigan

Columbia University

Pascal Massart

University of Paris-Sud

Andrew McCallum

UMass Amherst

Marina Meila

University of Washington

Andrew Moore

CMU

John Platt

Microsoft Research

Luc de Raedt

University of Freiburg

Christian Robert

U Paris-Dauphine

Sunita Sarawagi

IIT Bombay

Robert Schapire

Princeton University

Bernhard Schoelkopf

MPI Tübingen

Richard Sutton

University of Alberta

Larry Wasserman

CMU

Bin Yu

UC Berkeley



Documents

Document Sharing Hub

Editorial Scope

Topics

Foundations and Trends[®] in Machine Learning publishes survey and tutorial articles on the theory, algorithms and applications of machine learning, including the following topics:

- Adaptive control and signal processing
- Applications and case studies
- Behavioral, cognitive, and neural learning
- Bayesian learning
- Classification and prediction
- Clustering
- Data mining
- Dimensionality reduction
- Evaluation
- Game theoretic learning
- Graphical models
- Independent component analysis
- Inductive logic programming
- Kernel methods
- Markov chain Monte Carlo
- Model choice
- Nonparametric methods
- Online learning
- Optimization
- Reinforcement learning
- Relational learning
- Robustness
- Spectral methods
- Statistical learning theory
- Variational inference
- Visualization

Information for Librarians

Foundations and Trends[®] in Machine Learning, 2013, Volume 6, 4 issues. ISSN paper version 1935-8237. ISSN online version 1935-8245. Also available as a combined paper and online subscription.

Foundations and Trends® in Machine Learning
Vol. 6, No. 4 (2013) 375–454
© 2013 A. Geramifard, T. J. Walsh, S. Tellex, G. Chowdhary,
N. Roy, and J. P. How
DOI: 10.1561/22000000042



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

Alborz Geramifard
MIT LIDS
agf@mit.edu

Thomas J. Walsh
MIT LIDS
twalsh@mit.edu

Stefanie Tellex
MIT CSAIL
stefie10@csail.mit.edu

Girish Chowdhary
MIT LIDS
girishc@mit.edu

Nicholas Roy
MIT CSAIL
nickroy@mit.edu

Jonathan P. How
MIT LIDS
jhow@mit.edu



Documents
Document Sharing Hub

Contents

1	Introduction	2
2	Dynamic Programming and Reinforcement Learning	6
2.1	Markov Decision Processes (MDPs)	6
2.2	MDP Solvers at a Glance	9
2.3	Dynamic Programming	10
2.4	Approximate Dynamic Programming	14
2.5	Trajectory Based Value Iteration	18
2.6	Approximate Dynamic Programming in Matrix Format	20
2.7	Reinforcement Learning	31
2.8	Big Picture	38
3	Representations	40
3.1	Tabular Representation	42
3.2	Fixed Sparse Representation	43
3.3	Radial Basis Functions (RBFs)	44
3.4	Incremental Feature Dependency Discovery (iFDD) representation	45
4	Empirical Results	47
4.1	Algorithms	48
4.2	Domain Descriptions	48

4.3	Simulation Framework	52
4.4	Simulation Results	55
4.5	Discussion	62
5	Summary	65
	Acknowledgements	68
	References	69

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. 375–454, 2013.

DOI: 10.1561/22000000042.

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 unsolvable, 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 methods¹. 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

¹There 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 $(\mathcal{S}, \mathcal{A}, \mathcal{P}_s^a, \mathcal{R}_{ss'}^a, \gamma, S_0)$ where $\mathcal{S}$ is a set of states, $\mathcal{A}$ is a set of actions, $s, s' \in \mathcal{S}$, and $a \in \mathcal{A}$. $\mathcal{P}_s^a$ 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-

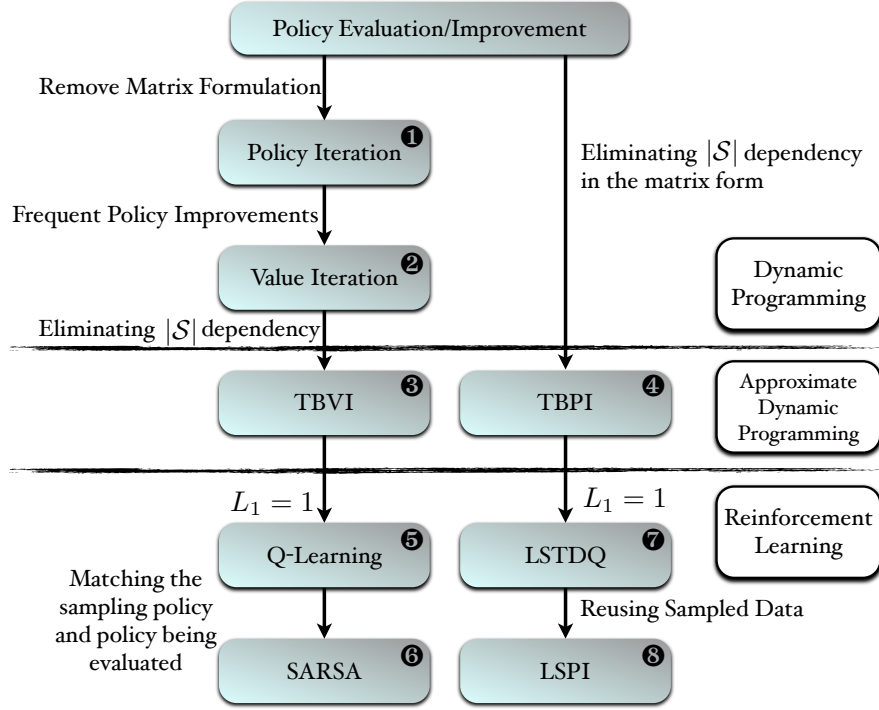


Figure 2.1: The roadmap we use to introduce various DP and RL techniques in a unified framework.

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 $\mathcal{P}_{ss'}^a$. Correspondingly, $\mathcal{R}_{ss'}^a$ 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, $\mathcal{P}_{ss'}^a$ and $\mathcal{R}_{ss'}^a$ constitute the *model* of an MDP. A *trajectory* is a sequence $(s_0, a_0, r_0, s_1, a_1, r_1, s_2, \dots)$, 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 $t \geq 1, s_{t+1} \sim \mathcal{P}_{s_t s_{t+1}}^{a_t}$). Similarly, each reward $r_t = \mathcal{R}_{s_t s_{t+1}}^{a_t}$ 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):

$$Q^\pi(s, a) = E_\pi \left[\sum_{t=0}^{\infty} \gamma^t r_t \mid s_0 = s, a_0 = a \right]. \quad (2.1)$$

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:

$$V^\pi(s) \triangleq Q^\pi(s, \pi(s)) \quad (2.2)$$

$$= E_\pi \left[\sum_{t=0}^{\infty} \gamma^t r_t \mid s_0 = s \right]. \quad (2.3)$$

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:

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

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]:

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

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

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

²From an economical perspective, 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-

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

⁴Sometime 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 $|S|$ 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]:

$$\begin{aligned}
V^\pi(s) &= E_\pi \left[\sum_{t=0}^{\infty} \gamma^t r_t \mid s_0 = s \right] \\
&= E_\pi \left[r_0 + \sum_{t=1}^{\infty} \gamma^t r_t \mid s_0 = s \right] \\
&= E_\pi \left[\mathcal{R}_{ss'}^{\pi(s)} + \gamma \sum_{t=1}^{\infty} \gamma^{t-1} r_t \mid s_0 = s \right] \\
&= \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^{\pi(s)} \left[\mathcal{R}_{ss'}^{\pi(s)} + \gamma V^\pi(s') \right]. \tag{2.6}
\end{aligned}$$

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 $\mathcal{S} = \{s_1, s_2, \dots, s_{|\mathcal{S}|}\}$ for which the vector $\mathbf{V}^\pi_{|\mathcal{S}| \times 1}$ 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 $\mathbf{P}$ be defined using $P_{ij} = \mathcal{P}_{s_i s_j}^{\pi(s_i)}$, and let vector $\mathbf{R}$ be defined using $\mathbf{R}_i = \sum_j \mathcal{P}_{s_i s_j}^{\pi(s_i)} \mathcal{R}_{s_i s_j}^{\pi(s_i)}$. Then Equation 2.6 takes the form:

$$\mathbf{V} = \mathbf{R} + \gamma \mathbf{P} \mathbf{V}.$$

Let us define

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

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 $\mathbf{V} = \mathbf{T}(\mathbf{V})$. Thus, the problem of evaluating policy π translates into finding the fixed-point of operator $\mathbf{T}$. Solving this equation for $\mathbf{V}$ yields:

$$\mathbf{V} = (\mathbf{I} - \gamma \mathbf{P})^{-1} \mathbf{R}, \tag{2.8}$$

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

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

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

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

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

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 $|\mathcal{S}| \times |\mathcal{A}|$ 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 $\mathbf{P}$ 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 $\mathbf{P}$ explicitly to avoid $\mathcal{O}(|\mathcal{S}|^2)$

⁵There 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 $s \in \mathcal{S}$ 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 $\mathcal{O}(|\mathcal{S}|)$ memory⁶ 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

⁶It 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	
1 $V(s) \leftarrow$ Initialize arbitrarily for $s \in \mathcal{S}$	
2 repeat	
3 for $s \in \mathcal{S}$ do	
4 $v \leftarrow V(s)$	
5 $V(s) \leftarrow \max_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a [\mathcal{R}_{ss'}^a + \gamma V(s')]$	$\mathcal{O}(\mathcal{A} \mathcal{S})$
6 $\pi(s) \leftarrow \operatorname{argmax}_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a [\mathcal{R}_{ss'}^a + \gamma V(s')]$	$\mathcal{O}(\mathcal{A} \mathcal{S})$
7 $\Delta \leftarrow \max(\Delta, v - V(s))$	
8 until $\Delta < \eta$	
9 return π	

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

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

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:

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

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 $\langle s_0, a_0, r_0, s_1, a_1, \dots \rangle$ 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]:

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

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:

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

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:

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

Notice that as $L_1 \rightarrow \infty$, 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:

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

with vectors $\phi(s, a)$ and θ 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; $\theta \in \mathbb{R}^n$ 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 θ 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.

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

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, $Q(s, a) = \phi(s, a)^\top \theta$, how can θ change to get $Q(s, a)$ closer to $Q^+(s, a)$? The answer is to move θ in the opposite direction of the squared difference between $Q(s, a)$ and $Q^+(s, a)$:

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

⁷In 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