Reinforcement Learning for Engineer-Mathematicians

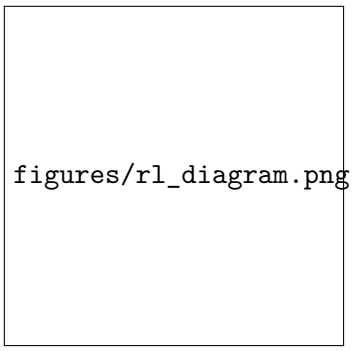
A Comprehensive Guide to Theory and Applications

Author Name

About This Book

This book provides a comprehensive introduction to reinforcement learning with a focus on mathematical rigor and practical applications. Each chapter includes:

- Intuitive explanations and motivating examples
- Formal mathematical treatment
- Practical algorithms and implementation notes
- Interactive Jupyter notebooks with complete code



Published Year

Institution/Publisher

To all engineers and mathematicians who seek to bridge
the gap between theory and practice

Preface

This book bridges the gap between the mathematical rigor of reinforcement learning theory and its practical applications in engineering systems. Written for engineer-mathematicians, it provides both the theoretical foundations necessary to understand why algorithms work and the practical insights needed to apply them successfully in real-world scenarios.

The field of reinforcement learning has evolved rapidly, with deep learning enabling applications previously thought impossible. However, many engineering applications require understanding the underlying mathematics to ensure safety, reliability, and optimal performance. This book fills that need by providing a comprehensive treatment that balances theory with practice.

Each chapter builds upon previous concepts while maintaining mathematical rigor. Examples are drawn from engineering disciplines including robotics, control systems, power systems, manufacturing, and communications. The goal is to equip readers with both the theoretical understanding and practical skills needed to successfully apply reinforcement learning in their own domains.

Acknowledgments

The author wishes to thank the many researchers, colleagues, and students who contributed to the development of this book through discussions, feedback, and collaboration. Special thanks to the reinforcement learning community for their open sharing of ideas and implementations.

Contents

List of Figures

List of Tables

List of Algorithms

Part I

Mathematical Foundations

This part establishes the mathematical foundations necessary for understanding reinforcement learning from both theoretical and engineering perspectives. We begin with essential mathematical prerequisites, then develop the formal framework of Markov Decision Processes, and conclude with classical dynamic programming methods that form the basis for modern reinforcement learning algorithms.

The treatment emphasizes mathematical rigor while maintaining practical relevance for engineering applications. Each concept is developed with careful attention to assumptions, proofs, and connections to control theory and optimization.

Chapter 1

Introduction and Mathematical Prerequisites

Chapter Overview

This chapter introduces the fundamental mathematical tools needed for reinforcement learning and provides intuitive motivation for why RL represents a paradigm shift from classical control theory. We'll cover probability theory, linear algebra, optimization, and stochastic processes with practical examples.

1.1 Motivation: From Control Theory to Learning Systems

Why Reinforcement Learning?

Imagine teaching a child to ride a bicycle. You don't give them the equations of motion or tell them exactly how to balance. Instead, they learn through trial and error, gradually improving their balance and control. This is the essence of reinforcement learning.

Reinforcement learning represents a fundamental paradigm shift from classical control theory and optimization. While traditional engineering approaches rely on explicit models and well-defined objectives, reinforcement learning enables systems to learn optimal behavior through interaction with their environment.

Traditional Control vs. Reinforcement Learning

Consider a classic engineering problem: designing a controller for an inverted pendulum.

Traditional Control Approach:

1. Deriving the system dynamics using Lagrangian mechanics
2. Linearizing around the equilibrium point
3. Designing a feedback controller using pole placement or LQR
4. Implementing the controller with known parameters

Reinforcement Learning Approach:

1. Define states (angle, angular velocity) and actions (applied force)
2. Specify a reward function (positive for upright, negative for falling)
3. Allow the agent to explore and learn through trial and error
4. Converge to an optimal policy without explicit knowledge of dynamics

This fundamental difference opens up possibilities for systems where:

- Dynamics are unknown or too complex to model accurately
- Environment conditions change over time
- Multiple conflicting objectives must be balanced
- System parameters vary or degrade over time

Industrial Example: Power Grid Management

Modern power grids face unprecedented challenges with renewable energy integration, electric vehicle charging, and dynamic pricing. Traditional grid control relies on pre-computed lookup tables and heuristic rules. Reinforcement learning enables real-time optimization that adapts to:

- Variable renewable generation
- Changing demand patterns
- Equipment failures and network topology changes
- Market price fluctuations

1.2 Mathematical Notation and Conventions

Notation Guide

Throughout this book, we adopt consistent mathematical notation that aligns with both control theory and machine learning conventions. Don't worry if some symbols are unfamiliar now—we'll introduce them gradually with intuitive explanations.

Throughout this book, we adopt consistent mathematical notation that aligns with both

control theory and machine learning conventions.

1.2.1 Sets and Spaces

Understanding Spaces

Think of a "space" as the collection of all possible values a variable can take. For example, if we're controlling a robot arm, the state space might include all possible joint angles and velocities.

$$\mathcal{S} = \text{State space (all possible states)} \tag{1.1}$$
$$\mathcal{A} = \text{Action space (all possible actions)} \tag{1.2}$$
$$\mathcal{R} = \text{Reward space (all possible rewards)} \tag{1.3}$$
$$\mathbb{R}^n = n\text{-dimensional real vector space} \tag{1.4}$$
$$\mathbb{R}^{m \times n} = \text{Space of } m \times n \text{ real matrices} \tag{1.5}$$

1.2.2 Functions and Operators

$$\pi : \mathcal{S} \to \mathcal{A} \quad \text{(Deterministic policy)} \tag{1.6}$$
$$\pi : \mathcal{S} \to \Delta(\mathcal{A}) \quad \text{(Stochastic policy)} \tag{1.7}$$
$$V^\pi : \mathcal{S} \to \mathbb{R} \quad \text{(Value function)} \tag{1.8}$$
$$Q^\pi : \mathcal{S} \times \mathcal{A} \to \mathbb{R} \quad \text{(Action-value function)} \tag{1.9}$$
$$T : \mathbb{R}^{\mathcal{S}} \to \mathbb{R}^{\mathcal{S}} \quad \text{(Bellman operator)} \tag{1.10}$$

where $\Delta(\mathcal{A})$ denotes the space of probability distributions over $\mathcal{A}$.

1.2.3 Probability and Expectation

$$\mathbb{P}(s'|s, a) = \text{Transition probability} \tag{1.11}$$
$$\mathbb{E}_\pi[\cdot] = \text{Expectation under policy } \pi \tag{1.12}$$
$$\mathbb{E}_{s\sim\mu}[\cdot] = \text{Expectation over distribution } \mu \tag{1.13}$$

1.3 Probability Theory Refresher

Reinforcement learning is fundamentally about making decisions under uncertainty. A solid understanding of probability theory is essential for analyzing convergence properties, sample complexity, and algorithm performance.

1.3.1 Probability Spaces and Random Variables

Definition 1.1 (Probability Space)**.** A probability space is a triple $(\Omega, \mathcal{F}, \mathbb{P})$ where:

- Ω is the sample space (set of all possible outcomes)
- $\mathcal{F}$ is a σ-algebra on Ω (collection of measurable events)
- $\mathbb{P} : \mathcal{F} \to [0, 1]$ is a probability measure satisfying:
  1. $\mathbb{P}(\Omega) = 1$
  2. For disjoint events $A_1, A_2, \ldots$: $\mathbb{P}(\bigcup_{i=1}^{\infty} A_i) = \sum_{i=1}^{\infty} \mathbb{P}(A_i)$

Definition 1.2 (Random Variable)**.** A random variable X is a measurable function $X : \Omega \to \mathbb{R}$ such that for every Borel set $B \subseteq \mathbb{R}$, the preimage $X^{-1}(B) \in \mathcal{F}$.

1.3.2 Conditional Expectation and Martingales

Conditional expectation plays a crucial role in reinforcement learning, particularly in the analysis of temporal difference methods and policy gradient algorithms.

Definition 1.3 (Conditional Expectation)**.** Given random variables X and Y, the conditional expectation $\mathbb{E}[X|Y]$ is the unique (almost surely) random variable that is:

1. Measurable with respect to $\sigma(Y)$
2. Satisfies $\mathbb{E}[\mathbb{E}[X|Y] \cdot \mathbf{1}_A] = \mathbb{E}[X \cdot \mathbf{1}_A]$ for all $A \in \sigma(Y)$

Theorem 1.1 (Law of Total Expectation)**.** For random variables X and Y:

$$\mathbb{E}[X] = \mathbb{E}[\mathbb{E}[X|Y]] \tag{1.14}$$

Definition 1.4 (Martingale)**.** A sequence of random variables $\{X_t\}_{t=0}^{\infty}$ is a martingale with respect to filtration $\{\mathcal{F}_t\}_{t=0}^{\infty}$ if:

1. X_t is $\mathcal{F}_t$-measurable for all t
2. $\mathbb{E}[|X_t|] < \infty$ for all t
3. $\mathbb{E}[X_{t+1}|\mathcal{F}_t] = X_t$ almost surely

Martingales are fundamental for proving convergence of stochastic algorithms in reinforcement learning.

1.3.3 Concentration Inequalities

Concentration inequalities provide bounds on the probability that random variables deviate from their expected values. These are essential for finite-sample analysis of RL algorithms.

Theorem 1.2 (Hoeffding's Inequality)**.** Let $X_1, \ldots, X_n$ be independent random variables

with $X_i \in [a_i, b_i]$ almost surely. Then for any $t > 0$:

$$\mathbb{P}\left(\left|\frac{1}{n}\sum_{i=1}^{n} X_i - \frac{1}{n}\sum_{i=1}^{n}\mathbb{E}[X_i]\right| \geq t\right) \leq 2\exp\left(-\frac{2n^2t^2}{\sum_{i=1}^{n}(b_i - a_i)^2}\right) \tag{1.15}$$

Theorem 1.3 (Azuma's Inequality)**.** Let $\{X_t\}_{t=0}^{\infty}$ be a martingale with respect to $\{\mathcal{F}_t\}_{t=0}^{\infty}$ such that $|X_{t+1} - X_t| \leq c_t$ almost surely. Then:

$$\mathbb{P}(|X_n - X_0| \geq t) \leq 2\exp\left(-\frac{t^2}{2\sum_{i=0}^{n-1} c_i^2}\right) \tag{1.16}$$

1.4 Linear Algebra Essentials

Linear algebra provides the foundation for function approximation, policy parameterization, and many algorithmic techniques in reinforcement learning.

1.4.1 Vector Spaces and Norms

Definition 1.5 (Vector Space)**.** A vector space V over field $\mathbb{F}$ (typically $\mathbb{R}$ or $\mathbb{C}$) is a set equipped with vector addition and scalar multiplication satisfying:

1. Commutativity: $u + v = v + u$
2. Associativity: $(u + v) + w = u + (v + w)$
3. Identity: $\exists 0 \in V$ such that $v + 0 = v$
4. Inverse: $\forall v \in V, \exists -v$ such that $v + (-v) = 0$
5. Scalar associativity: $a(bv) = (ab)v$
6. Scalar identity: $1v = v$
7. Distributivity: $a(u + v) = au + av$ and $(a + b)v = av + bv$

Definition 1.6 (Norm)**.** A norm on vector space V is a function $\|\cdot\| : V \to \mathbb{R}_{\geq 0}$ satisfying:

1. $\|v\| = 0$ if and only if $v = 0$
2. $\|av\| = |a|\|v\|$ for scalar a
3. $\|u + v\| \leq \|u\| + \|v\|$ (triangle inequality)

Common norms in $\mathbb{R}^n$:

$$\|x\|_1 = \sum_{i=1}^{n} |x_i| \quad (\ell_1 \text{ norm}) \tag{1.17}$$

$$\|x\|_2 = \sqrt{\sum_{i=1}^{n} x_i^2} \quad (\text{Euclidean norm}) \tag{1.18}$$

$$\|x\|_\infty = \max_{i=1,\ldots,n} |x_i| \quad (\ell_\infty \text{ norm}) \tag{1.19}$$

1.4.2 Inner Products and Orthogonality

Definition 1.7 (Inner Product)**.** An inner product on real vector space V is a function $\langle \cdot, \cdot \rangle : V \times V \to \mathbb{R}$ satisfying:

1. Symmetry: $\langle u, v \rangle = \langle v, u \rangle$
2. Linearity: $\langle au + bv, w \rangle = a\langle u, w \rangle + b\langle v, w \rangle$
3. Positive definiteness: $\langle v, v \rangle \geq 0$ with equality iff $v = 0$

The induced norm is $\|v\| = \sqrt{\langle v, v \rangle}$.

Theorem 1.4 (Cauchy-Schwarz Inequality)**.** For vectors u, v in an inner product space:

$$|\langle u, v \rangle| \leq \|u\| \|v\| \tag{1.20}$$

with equality if and only if u and v are linearly dependent.

1.4.3 Eigenvalues and Spectral Theory

Definition 1.8 (Eigenvalue and Eigenvector)**.** For matrix $A \in \mathbb{R}^{n \times n}$, scalar λ is an eigenvalue with corresponding eigenvector $v \neq 0$ if:

$$Av = \lambda v \tag{1.21}$$

Theorem 1.5 (Spectral Theorem for Symmetric Matrices)**.** Every real symmetric matrix A has an orthonormal basis of eigenvectors with real eigenvalues. If $A = Q\Lambda Q^T$ where Q is orthogonal and Λ is diagonal, then:

$$A = \sum_{i=1}^{n} \lambda_i q_i q_i^T \tag{1.22}$$

where λ_i are eigenvalues and q_i are corresponding orthonormal eigenvectors.

1.5 Optimization Fundamentals

Optimization theory underpins virtually all reinforcement learning algorithms, from value iteration to policy gradient methods.

1.5.1 Convex Analysis

Definition 1.9 (Convex Set)**.** A set $C \subseteq \mathbb{R}^n$ is convex if for all $x, y \in C$ and $\lambda \in [0, 1]$:

$$\lambda x + (1 - \lambda) y \in C \tag{1.23}$$

Definition 1.10 (Convex Function)**.** A function $f : \mathbb{R}^n \to \mathbb{R}$ is convex if its domain is convex and for all x, y in the domain and $\lambda \in [0, 1]$:

$$f(\lambda x + (1 - \lambda)y) \leq \lambda f(x) + (1 - \lambda)f(y) \tag{1.24}$$

Theorem 1.6 (First-Order Characterization of Convexity)**.** For differentiable function f, the following are equivalent:

1. f is convex
2. $f(y) \geq f(x) + \nabla f(x)^T(y - x)$ for all x, y
3. ∇f is monotone: $(\nabla f(x) - \nabla f(y))^T(x - y) \geq 0$

1.5.2 Unconstrained Optimization

Theorem 1.7 (Necessary Conditions for Optimality)**.** If x^* is a local minimum of differentiable function f, then:

$$\nabla f(x^*) = 0 \tag{1.25}$$

If f is twice differentiable, then additionally:

$$\nabla^2 f(x^*) \succeq 0 \quad \text{(positive semidefinite)} \tag{1.26}$$

Theorem 1.8 (Sufficient Conditions for Optimality)**.** If $\nabla f(x^*) = 0$ and $\nabla^2 f(x^*) \succ 0$ (positive definite), then x^* is a strict local minimum.

1.5.3 Gradient Descent and Convergence Analysis

The gradient descent algorithm iterates:

$$x_{k+1} = x_k - \alpha_k \nabla f(x_k) \tag{1.27}$$

Theorem 1.9 (Convergence of Gradient Descent)**.** For convex function f with L-Lipschitz gradient and step size $\alpha \leq 1/L$:

$$f(x_k) - f(x^*) \leq \frac{\|x_0 - x^*\|^2}{2\alpha k} \tag{1.28}$$

where x^* is the optimal solution.

For strongly convex functions, the convergence rate improves to exponential.

1.6 Stochastic Processes and Markov Chains

Understanding stochastic processes is crucial for analyzing the temporal dynamics of reinforcement learning systems.

1.6.1 Discrete-Time Stochastic Processes

Definition 1.11 (Stochastic Process)**.** A discrete-time stochastic process is a sequence of random variables $\{X_t\}_{t=0}^{\infty}$ where each X_t takes values in some state space $\mathcal{S}$.

Definition 1.12 (Markov Property)**.** A stochastic process $\{X_t\}_{t=0}^{\infty}$ satisfies the Markov property if:

$$\mathbb{P}(X_{t+1} = s' | X_t = s, X_{t-1} = s_{t-1}, \ldots, X_0 = s_0) = \mathbb{P}(X_{t+1} = s' | X_t = s) \tag{1.29}$$

for all states $s, s', s_0, \ldots, s_{t-1}$ and times $t \geq 0$.

1.6.2 Markov Chain Analysis

For finite state space $\mathcal{S} = \{1, 2, \ldots, n\}$, a Markov chain is characterized by its transition matrix $P \in \mathbb{R}^{n \times n}$ where $P_{ij} = \mathbb{P}(X_{t+1} = j | X_t = i)$.

Definition 1.13 (Irreducibility and Aperiodicity)**.** A Markov chain is:

- **Irreducible** if every state is reachable from every other state
- **Aperiodic** if $\gcd\{n \geq 1 : P_{ii}^{(n)} > 0\} = 1$ for some state i

Theorem 1.10 (Fundamental Theorem of Markov Chains)**.** For an irreducible, aperiodic, finite Markov chain:

1. There exists a unique stationary distribution π satisfying $\pi = \pi P$
2. $\lim_{t \to \infty} P^t = \mathbf{1}\pi^T$ where $\mathbf{1}$ is the vector of ones
3. For any initial distribution μ_0: $\lim_{t \to \infty} \|\mu_t - \pi\|_{TV} = 0$

1.6.3 Mixing Times and Convergence Rates

Definition 1.14 (Total Variation Distance)**.** The total variation distance between distributions μ and ν on finite space $\mathcal{S}$ is:

$$\|\mu - \nu\|_{TV} = \frac{1}{2} \sum_{s \in \mathcal{S}} |\mu(s) - \nu(s)| \tag{1.30}$$

Definition 1.15 (Mixing Time)**.** The mixing time of a Markov chain is:

$$t_{mix}(\epsilon) = \min\{t : \max_{i \in \mathcal{S}} \|P^t(i, \cdot) - \pi\|_{TV} \leq \epsilon\} \tag{1.31}$$

Understanding mixing times is essential for analyzing sample complexity in reinforcement learning algorithms that rely on sampling from stationary distributions.

1.7 Chapter Summary

This chapter established the mathematical foundations necessary for rigorous analysis of reinforcement learning algorithms. Key concepts include:

- The paradigm shift from model-based control to learning-based optimization
- Probability theory tools: conditional expectation, martingales, concentration inequalities
- Linear algebra foundations: vector spaces, norms, spectral theory
- Convex optimization and gradient descent convergence analysis
- Markov chain theory and convergence to stationary distributions

These mathematical tools will be applied throughout the book to analyze algorithm convergence, sample complexity, and performance guarantees. The next chapter develops the formal framework of Markov Decision Processes, which provides the mathematical foundation for all subsequent reinforcement learning algorithms.

Chapter 2

Markov Decision Processes (MDPs)

Chapter Overview

This chapter introduces Markov Decision Processes (MDPs), the mathematical foundation of reinforcement learning. We'll build intuition through concrete examples before diving into the formal theory, then explore solution methods like value iteration and policy iteration.

What is an MDP?

Think of an MDP as a mathematical description of a decision-making situation where:

- You observe the current situation (state)
- You choose an action based on what you observe
- The world responds by transitioning to a new state and giving you a reward
- This process repeats over time

The key insight is that the future only depends on the current state, not the entire history—this is the Markov property.

Markov Decision Processes provide the mathematical framework for modeling sequential decision-making under uncertainty. This chapter develops the formal theory of MDPs with particular attention to mathematical rigor and engineering applications.

2.1 Understanding MDPs Through Examples

Before diving into formal definitions, let's build intuition through a concrete example.

Grid World Navigation

Consider a robot navigating a 4×4 grid world:

- **States**: Each cell in the grid (16 total states)
- **Actions**: Move up, down, left, or right
- **Transitions**: Move to adjacent cell (or stay put if hitting a wall)
- **Rewards**: +10 for reaching the goal, -1 for each step, -10 for falling into holes
- **Goal**: Find the shortest path to the target while avoiding obstacles

2.2 Formal Definition and Mathematical Properties

Now that we have intuition, let's formalize these concepts.

Definition 2.1 (Markov Decision Process)**.** A Markov Decision Process is a 5-tuple $(\mathcal{S}, \mathcal{A}, P, \mathcal{R}, \gamma)$ where:

- $\mathcal{S}$ is the **state space** (all possible situations)
- $\mathcal{A}$ is the **action space** (all possible actions)
- $P : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$ is the **transition kernel** (dynamics)
- $\mathcal{R} : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is the **reward function** (immediate feedback)
- $\gamma \in [0, 1)$ is the **discount factor** (how much we value future rewards)

Understanding the Components

- **State space** $\mathcal{S}$: All possible configurations of your system
- **Action space** $\mathcal{A}$: All decisions you can make in any given state
- **Transition function** P: Describes how actions change states (the "physics" of your world)
- **Reward function** $\mathcal{R}$: Immediate feedback telling you how good an action was
- **Discount factor** γ: How much you care about future vs. immediate rewards (0 = only care about immediate, close to 1 = care about long-term)

The transition kernel satisfies $\sum_{s' \in \mathcal{S}} P(s, a, s') = 1$ for all $(s, a) \in \mathcal{S} \times \mathcal{A}$, and we write $P(s'|s, a) = P(s, a, s')$ for the probability of transitioning to state s' from state s under action a.

2.2.1 Assumptions and Regularity Conditions

For mathematical tractability, we typically assume:

[Measurability] The state and action spaces are measurable spaces, and the transition kernel and reward function are measurable with respect to the appropriate σ-algebras.

[Bounded Rewards] The reward function satisfies $\sup_{s,a} |\mathcal{R}(s,a)| \leq R_{max} < \infty$.

[Discount Factor] The discount factor satisfies $\gamma \in [0,1)$ to ensure convergence of infinite-horizon value functions.

2.2.2 State and Action Spaces

Discrete Spaces

For finite MDPs with $|\mathcal{S}| = n$ and $|\mathcal{A}| = m$, we can represent:

- Transition probabilities as tensors $P^a \in \mathbb{R}^{n \times n}$ for each action a
- Rewards as matrices $R \in \mathbb{R}^{n \times m}$
- Policies as matrices $\Pi \in [0,1]^{n \times m}$ with $\sum_a \Pi(s,a) = 1$

Continuous Spaces

For continuous state spaces $\mathcal{S} \subseteq \mathbb{R}^d$, the transition kernel becomes a probability measure:

$$P(\cdot|s,a) : \mathcal{B}(\mathcal{S}) \to [0,1] \tag{2.1}$$

where $\mathcal{B}(\mathcal{S})$ is the Borel σ-algebra on $\mathcal{S}$.

Engineering Example: Inverted Pendulum

Consider an inverted pendulum with:

- State: $s = (\theta, \dot{\theta}) \in [-\pi, \pi] \times [-10, 10]$ (angle and angular velocity)
- Action: $a \in [-5, 5]$ (applied torque)
- Dynamics: $\ddot{\theta} = \frac{g}{l} \sin\theta + \frac{a}{ml^2}$ plus noise
- Reward: $r(s,a) = -\theta^2 - 0.1\dot{\theta}^2 - 0.01a^2$ (quadratic cost)

2.3 Policies and Value Functions

What is a Policy?

A policy is simply a decision-making rule. It tells an agent what action to take in each possible state. Think of it as a strategy or game plan.

2.3.1 Types of Policies

Definition 2.2 (Deterministic Policy)**.** A deterministic policy is a function $\pi : \mathcal{S} \to \mathcal{A}$ that maps each state to exactly one action.

Deterministic Policy Example

In our grid world: "Always move towards the goal" could be a deterministic policy where $\pi(\text{state}) = \text{direction\_to\_goal}$.

Definition 2.3 (Stochastic Policy)**.** A stochastic policy $\pi : \mathcal{S} \to \Delta(\mathcal{A})$ assigns a probability distribution over actions for each state, where $\Delta(\mathcal{A})$ is the space of probability measures on $\mathcal{A}$.

Stochastic Policy Example

In grid world: "Move towards goal with probability 0.8, move randomly otherwise" gives $\pi(\text{best\_action}|s) = 0.8$ and equal probability to other actions.

Definition 2.4 (History-Dependent Policy)**.** A history-dependent policy depends on the entire sequence of past states and actions: $\pi_t : (\mathcal{S} \times \mathcal{A})^t \times \mathcal{S} \to \Delta(\mathcal{A})$

Why Focus on Markovian Policies?

While policies could potentially use the entire history, the Markov property means that optimal policies only need to depend on the current state. This greatly simplifies our analysis!

Theorem 2.1 (Sufficiency of Markovian Policies)**.** For any history-dependent policy, there exists a Markovian policy that achieves the same expected discounted reward.

Proof. This follows from the Markov property of the state transitions. The expected future reward depends only on the current state, not the history of how that state was reached. □

2.3.2 Value Function Theory

Definition 2.5 (State Value Function)**.** The state value function for policy π is:

$$V^\pi(s) = \mathbb{E}^\pi \left[\sum_{t=0}^{\infty} \gamma^t \mathcal{R}(S_t, A_t) \mid S_0 = s \right] \tag{2.2}$$

Definition 2.6 (Action Value Function)**.** The action value function (Q-function) for policy π is:

$$Q^\pi(s, a) = \mathbb{E}^\pi \left[\sum_{t=0}^{\infty} \gamma^t \mathcal{R}(S_t, A_t) \mid S_0 = s, A_0 = a \right] \tag{2.3}$$

Theorem 2.2 (Existence and Uniqueness of Value Functions)**.** Under Assumptions 1-3, the value functions V^π and Q^π exist, are unique, and satisfy $\|V^\pi\|_\infty \leq \frac{R_{max}}{1-\gamma}$.

Proof. The geometric series $\sum_{t=0}^{\infty} \gamma^t R_{max}$ converges to $\frac{R_{max}}{1-\gamma}$ since $\gamma < 1$. Uniqueness follows from the linearity of expectation. □

2.3.3 Bellman Equations

The fundamental recursive relationships in reinforcement learning are the Bellman equations.

Theorem 2.3 (Bellman Equations for Policy Evaluation)**.** For any policy π:

$$V^\pi(s) = \sum_{a\in\mathcal{A}} \pi(a|s) \left[\mathcal{R}(s,a) + \gamma \sum_{s'\in\mathcal{S}} P(s'|s,a)V^\pi(s')\right] \tag{2.4}$$

$$Q^\pi(s,a) = \mathcal{R}(s,a) + \gamma \sum_{s'\in\mathcal{S}} P(s'|s,a) \sum_{a'\in\mathcal{A}} \pi(a'|s')Q^\pi(s',a') \tag{2.5}$$

Proof. By the tower rule of conditional expectation:

$$V^\pi(s) = \mathbb{E}^\pi \left[\mathcal{R}(S_0,A_0) + \gamma \sum_{t=1}^{\infty} \gamma^{t-1}\mathcal{R}(S_t,A_t) \mid S_0 = s\right] \tag{2.6}$$

$$= \mathbb{E}^\pi[\mathcal{R}(S_0,A_0)|S_0 = s] + \gamma\mathbb{E}^\pi\left[V^\pi(S_1) \mid S_0 = s\right] \tag{2.7}$$

Expanding the expectations gives the Bellman equation. □

2.4 Optimal Policies and Bellman Optimality

2.4.1 Partial Ordering on Policies

Definition 2.7 (Policy Partial Order)**.** Policy π_1 dominates policy π_2 (written $\pi_1 \geq \pi_2$) if:

$$V^{\pi_1}(s) \geq V^{\pi_2}(s) \quad \forall s \in \mathcal{S} \tag{2.8}$$

Theorem 2.4 (Existence of Optimal Policies)**.** There exists an optimal deterministic policy π^* such that:

$$V^{\pi^*}(s) = \max_\pi V^\pi(s) \equiv V^*(s) \quad \forall s \in \mathcal{S} \tag{2.9}$$

2.4.2 Bellman Optimality Equations

Theorem 2.5 (Bellman Optimality Equations)**.** The optimal value functions satisfy:

$$V^*(s) = \max_{a\in\mathcal{A}} \left[\mathcal{R}(s,a) + \gamma \sum_{s'\in\mathcal{S}} P(s'|s,a)V^*(s')\right] \tag{2.10}$$

$$Q^*(s,a) = \mathcal{R}(s,a) + \gamma \sum_{s'\in\mathcal{S}} P(s'|s,a) \max_{a'\in\mathcal{A}} Q^*(s',a') \tag{2.11}$$

Corollary 2.1 (Optimal Policy Extraction)**.** An optimal policy can be extracted from the optimal value functions as:

$$\pi^*(s) \in \operatorname*{argmax}_{a\in\mathcal{A}} Q^*(s,a) \tag{2.12}$$

2.5 Contraction Mapping Theorem and Fixed Points

The mathematical foundation for proving convergence of dynamic programming algorithms relies on contraction mapping theory.

2.5.1 Bellman Operators

Definition 2.8 (Bellman Operator)**.** For policy π, the Bellman operator $T^\pi : \mathbb{R}^{\mathcal{S}} \to \mathbb{R}^{\mathcal{S}}$ is defined by:

$$(T^\pi V)(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \left[\mathcal{R}(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s,a) V(s') \right] \tag{2.13}$$

Definition 2.9 (Bellman Optimality Operator)**.** The Bellman optimality operator $T^* : \mathbb{R}^{\mathcal{S}} \to \mathbb{R}^{\mathcal{S}}$ is defined by:

$$(T^* V)(s) = \max_{a \in \mathcal{A}} \left[\mathcal{R}(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s,a) V(s') \right] \tag{2.14}$$

2.5.2 Contraction Properties

Theorem 2.6 (Contraction Property of Bellman Operators)**.** Under the supremum norm $\|V\|_\infty = \max_{s \in \mathcal{S}} |V(s)|$:

1. T^π is a γ-contraction: $\|T^\pi V_1 - T^\pi V_2\|_\infty \leq \gamma \|V_1 - V_2\|_\infty$
2. T^* is a γ-contraction: $\|T^* V_1 - T^* V_2\|_\infty \leq \gamma \|V_1 - V_2\|_\infty$

Proof. For the policy operator:

$$\begin{aligned}
|(T^\pi V_1)(s) - (T^\pi V_2)(s)| &= \left| \sum_a \pi(a|s) \gamma \sum_{s'} P(s'|s,a)[V_1(s') - V_2(s')] \right| && (2.15)\\
&\leq \sum_a \pi(a|s) \gamma \sum_{s'} P(s'|s,a) |V_1(s') - V_2(s')| && (2.16)\\
&\leq \gamma \|V_1 - V_2\|_\infty \sum_a \pi(a|s) \sum_{s'} P(s'|s,a) && (2.17)\\
&= \gamma \|V_1 - V_2\|_\infty && (2.18)
\end{aligned}$$

The proof for T^* follows similarly using the fact that the max operator is non-expansive. □

2.5.3 Banach Fixed Point Theorem Application

Theorem 2.7 (Banach Fixed Point Theorem)**.** Let (X, d) be a complete metric space and $T : X \to X$ be a contraction mapping with contraction factor $\gamma < 1$. Then:

1. T has a unique fixed point $x^* \in X$
2. For any $x_0 \in X$, the sequence $x_{n+1} = T(x_n)$ converges to x^*

3. The convergence rate is geometric: $d(x_n, x^*) \leq \gamma^n d(x_0, x^*)$

Corollary 2.2 (Convergence of Value Iteration)**.** The value iteration algorithm $V_{k+1} = T^* V_k$ converges geometrically to the unique optimal value function V^* at rate γ.

2.6 Policy Improvement and Optimality

2.6.1 Policy Improvement Theorem

Theorem 2.8 (Policy Improvement Theorem)**.** Let π be any policy and define the improved policy π' by:

$$\pi'(s) \in \operatorname*{argmax}_{a \in \mathcal{A}} Q^\pi(s, a) \tag{2.19}$$

Then $\pi' \geq \pi$, with strict inequality unless π is optimal.

Proof. For any state s:

$$Q^\pi(s, \pi'(s)) \geq Q^\pi(s, \pi(s)) = V^\pi(s) \tag{2.20}$$

By the policy evaluation equation and induction, this implies $V^{\pi'}(s) \geq V^\pi(s)$. □

2.6.2 Policy Iteration Algorithm

The policy improvement theorem leads to the policy iteration algorithm:

Algorithm 1 Policy Iteration

Input: Initial policy π_0
Output: Optimal policy π^*
  $k \leftarrow 0$
  repeat
    Policy Evaluation: Solve $V^{\pi_k} = T^{\pi_k} V^{\pi_k}$
    Policy Improvement: $\pi_{k+1}(s) \leftarrow \operatorname{argmax}_a Q^{\pi_k}(s, a)$
    $k \leftarrow k + 1$
  until $\pi_k = \pi_{k-1}$
  return $\pi^* = \pi_k$

Theorem 2.9 (Convergence of Policy Iteration)**.** Policy iteration converges to an optimal policy in finitely many iterations for finite MDPs.

2.7 Computational Complexity Analysis

2.7.1 Value Iteration Complexity

For finite MDPs with $|\mathcal{S}| = n$ and $|\mathcal{A}| = m$:

- **Time per iteration:** $O(mn^2)$ operations
- **Iterations to ϵ-accuracy:** $O(\log(\epsilon^{-1}))$ iterations

- **Total complexity:** $O(mn^2 \log(\epsilon^{-1}))$

2.7.2 Policy Iteration Complexity

- **Policy evaluation:** $O(n^3)$ for direct matrix inversion, $O(n^2)$ per iteration for iterative methods
- **Policy improvement:** $O(mn^2)$
- **Number of policy iterations:** At most m^n (typically much smaller)

2.7.3 Modified Policy Iteration

To balance the computational costs, modified policy iteration performs only k steps of policy evaluation:

Algorithm 2 Modified Policy Iteration

Input: Initial policy π_0, evaluation steps k
  Initialize V_0 arbitrarily
  for $i = 0, 1, 2, \ldots$ **do**
    for $j = 1, 2, \ldots, k$ **do**
      $V_j \leftarrow T^{\pi_i} V_{j-1}$
    end for
    $\pi_{i+1}(s) \leftarrow \text{argmax}_a \left[r(s,a) + \gamma \sum_{s'} P(s'|s,a) V_k(s') \right]$
  end for

2.8 Connections to Classical Control Theory

2.8.1 Linear Quadratic Regulator (LQR)

For linear dynamics $s_{t+1} = As_t + Ba_t + w_t$ and quadratic costs $r(s,a) = -s^T Qs - a^T Ra$, the optimal value function is quadratic: $V^*(s) = -s^T Ps$ where P satisfies the discrete algebraic Riccati equation:

$$P = Q + A^T PA - A^T PB(R + B^T PB)^{-1} B^T PA \tag{2.21}$$

The optimal policy is linear: $\pi^*(s) = -Ks$ where $K = (R + B^T PB)^{-1} B^T PA$.

2.8.2 Hamilton-Jacobi-Bellman Equation

For continuous-time systems, the Bellman equation becomes the Hamilton-Jacobi-Bellman (HJB) partial differential equation:

$$\frac{\partial V}{\partial t} + \min_a \left[r(s,a) + \frac{\partial V}{\partial s} f(s,a) \right] = 0 \tag{2.22}$$

where $f(s, a)$ is the system dynamics.

2.9 Advanced Topics

2.9.1 Partially Observable MDPs (POMDPs)

Definition 2.10 (POMDP)**.** A POMDP extends an MDP with observations: $(\mathcal{S}, \mathcal{A}, \mathcal{O}, P, \mathcal{R}, \mathcal{Z}, \gamma)$ where:

- $\mathcal{O}$ is the observation space
- $\mathcal{Z} : \mathcal{S} \times \mathcal{A} \times \mathcal{O} \to [0, 1]$ is the observation model

The optimal policy depends on the belief state $b(s) = \mathbb{P}(S_t = s|h_t)$ where h_t is the history of observations.

2.9.2 Constrained MDPs

Definition 2.11 (Constrained MDP)**.** A constrained MDP adds constraint functions $c_i : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ and thresholds d_i:

$$\max_{\pi} \quad \mathbb{E}^{\pi}\left[\sum_{t=0}^{\infty} \gamma^t \mathcal{R}(S_t, A_t)\right] \tag{2.23}$$

$$\text{subject to} \quad \mathbb{E}^{\pi}\left[\sum_{t=0}^{\infty} \gamma^t c_i(S_t, A_t)\right] \leq d_i, \quad i = 1, \ldots, m \tag{2.24}$$

Solutions typically use Lagrangian methods and primal-dual algorithms.

2.10 Chapter Summary

This chapter established the mathematical foundations of Markov Decision Processes:

- Formal definition of MDPs and regularity assumptions
- Policy and value function theory with existence and uniqueness results
- Bellman equations and optimality conditions
- Contraction mapping theory and convergence guarantees
- Dynamic programming algorithms: value iteration and policy iteration
- Computational complexity analysis
- Connections to classical control theory and advanced extensions

The mathematical framework developed here provides the foundation for all reinforcement learning algorithms. The next chapter examines dynamic programming methods in detail, providing the algorithmic foundation for modern RL techniques.

Chapter 3

Dynamic Programming Foundations

Dynamic programming provides the theoretical and algorithmic foundation for reinforcement learning. This chapter develops the mathematical theory of dynamic programming with emphasis on convergence analysis, computational complexity, and connections to classical optimal control.

3.1 Principle of Optimality

The fundamental insight underlying dynamic programming is Bellman's principle of optimality, which enables the decomposition of complex sequential decision problems into simpler subproblems.

Theorem 3.1 (Principle of Optimality)**.** An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision.

3.1.1 Mathematical Formulation

For an MDP $(\mathcal{S}, \mathcal{A}, P, \mathcal{R}, \gamma)$, consider a finite-horizon problem with horizon T. Define the optimal value function:

$$V_t^*(s) = \max_{\pi} \mathbb{E}\left[\sum_{k=t}^{T-1} \gamma^{k-t} \mathcal{R}(S_k, A_k) \mid S_t = s, \pi\right] \tag{3.1}$$

Theorem 3.2 (Finite-Horizon Optimality)**.** The optimal value function satisfies the recursive relation:

$$V_T^*(s) = 0 \quad \forall s \in \mathcal{S} \tag{3.2}$$

$$V_t^*(s) = \max_{a \in \mathcal{A}} \left[\mathcal{R}(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s,a) V_{t+1}^*(s')\right] \tag{3.3}$$

for $t = T-1, T-2, \ldots, 0$.

Proof. The proof follows by backward induction. At time T, no more rewards can be collected, so $V_T^*(s) = 0$. For $t < T$, any optimal policy must choose the action that maximizes immediate reward plus discounted future value, leading to the recursive formulation. □

3.1.2 Engineering Interpretation

The principle of optimality has direct parallels in engineering optimization:

Optimal Control Example

Consider a spacecraft trajectory optimization problem:

- State: position and velocity $(x, v) \in \mathbb{R}^6$
- Control: thrust vector $u \in \mathbb{R}^3$
- Dynamics: $\dot{x} = v$, $\dot{v} = u/m - \nabla\Phi(x)$ (gravitational field)
- Cost: fuel consumption $\int_0^T \|u(t)\| dt$

The principle of optimality implies that if we have an optimal trajectory from Earth to Mars, then any sub-trajectory (e.g., from lunar orbit to Mars) must also be optimal for the sub-problem.

3.2 Value Iteration: Convergence Analysis

Value iteration is the most fundamental algorithm in dynamic programming, providing a constructive method for computing optimal value functions.

3.2.1 Algorithm Description

Algorithm 3 Value Iteration

Input: MDP $(\mathcal{S}, \mathcal{A}, P, \mathcal{R}, \gamma)$, tolerance $\epsilon > 0$
Output: ϵ-optimal value function V
  Initialize $V_0(s)$ arbitrarily for all $s \in \mathcal{S}$
  $k \leftarrow 0$
  repeat
    for each $s \in \mathcal{S}$ **do**
      $V_{k+1}(s) \leftarrow \max_{a \in \mathcal{A}} \left[\mathcal{R}(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s,a) V_k(s')\right]$
    end for
    $k \leftarrow k + 1$
  until $\|V_k - V_{k-1}\|_\infty < \epsilon(1-\gamma)/(2\gamma)$
  return V_k

3.2.2 Convergence Theory

Theorem 3.3 (Convergence of Value Iteration)**.** For any initial value function V_0, the value iteration sequence $\{V_k\}_{k=0}^{\infty}$ defined by $V_{k+1} = T^* V_k$ converges to the unique optimal value function V^* at geometric rate γ.

Specifically:

$$\|V_k - V^*\|_\infty \leq \gamma^k \|V_0 - V^*\|_\infty \tag{3.4}$$

Proof. Since T^* is a γ-contraction in the supremum norm and V^* is the unique fixed point of T^*, the result follows directly from the Banach fixed point theorem. □

3.2.3 Error Bounds and Stopping Criteria

Theorem 3.4 (Error Bounds for Value Iteration)**.** If $\|V_{k+1} - V_k\|_\infty \leq \delta$, then:

$$\|V_k - V^*\|_\infty \leq \frac{\gamma\delta}{1-\gamma} \tag{3.5}$$

$$\|V_{k+1} - V^*\|_\infty \leq \frac{\delta}{1-\gamma} \tag{3.6}$$

Proof. Using the triangle inequality and contraction property:

$$\|V_k - V^*\|_\infty = \|V_k - T^*V_k + T^*V_k - V^*\|_\infty \tag{3.7}$$

$$\leq \|V_k - T^*V_k\|_\infty + \|T^*V_k - T^*V^*\|_\infty \tag{3.8}$$

$$= \|V_k - V_{k+1}\|_\infty + \gamma\|V_k - V^*\|_\infty \tag{3.9}$$

Solving for $\|V_k - V^*\|_\infty$ gives the first bound. The second follows similarly. □

Corollary 3.1 (Practical Stopping Criterion)**.** To achieve $\|V_k - V^*\|_\infty \leq \epsilon$, it suffices to stop when:

$$\|V_{k+1} - V_k\|_\infty \leq \epsilon(1-\gamma) \tag{3.10}$$

3.2.4 Computational Complexity

Theorem 3.5 (Sample Complexity of Value Iteration)**.** To achieve ϵ-optimal value function, value iteration requires:

$$O\left(\frac{\log(\epsilon^{-1}) + \log(\|V_0 - V^*\|_\infty)}{1-\gamma}\right) \tag{3.11}$$

iterations.

For each iteration:

- **Time complexity:** $O(|\mathcal{S}|^2|\mathcal{A}|)$ for tabular case
- **Space complexity:** $O(|\mathcal{S}|)$ for storing value function
- **Total operations:** $O(|\mathcal{S}|^2|\mathcal{A}|\log(\epsilon^{-1})/(1-\gamma))$

3.3 Policy Iteration: Mathematical Guarantees

Policy iteration alternates between policy evaluation and policy improvement, providing an alternative approach with different computational characteristics.

3.3.1 Policy Evaluation

Given policy π, policy evaluation solves the linear system:

$$V^\pi = T^\pi V^\pi \tag{3.12}$$

In matrix form for finite MDPs:

$$V^\pi = R^\pi + \gamma P^\pi V^\pi \tag{3.13}$$

where $R^\pi \in \mathbb{R}^{|\mathcal{S}|}$ and $P^\pi \in \mathbb{R}^{|\mathcal{S}|\times|\mathcal{S}|}$ are policy-specific reward and transition matrices.

Theorem 3.6 (Unique Solution to Policy Evaluation)**.** The linear system $(I-\gamma P^\pi)V^\pi = R^\pi$ has a unique solution:

$$V^\pi = (I - \gamma P^\pi)^{-1} R^\pi \tag{3.14}$$

since $\rho(P^\pi) \leq 1$ and $\gamma < 1$ ensure $(I - \gamma P^\pi)$ is invertible.

3.3.2 Iterative Policy Evaluation

For large state spaces, direct matrix inversion is computationally prohibitive. Iterative policy evaluation uses:

$$V^\pi_{k+1} = T^\pi V^\pi_k \tag{3.15}$$

Theorem 3.7 (Convergence of Iterative Policy Evaluation)**.** The sequence $\{V^\pi_k\}$ converges geometrically to V^π at rate γ:

$$\|V^\pi_k - V^\pi\|_\infty \leq \gamma^k \|V^\pi_0 - V^\pi\|_\infty \tag{3.16}$$

3.3.3 Policy Improvement Analysis

Theorem 3.8 (Strict Improvement or Optimality)**.** Given policy π and improved policy π' defined by:

$$\pi'(s) \in \operatorname*{argmax}_{a\in\mathcal{A}} Q^\pi(s,a) \tag{3.17}$$

Then either:

1. $V^{\pi'}(s) > V^\pi(s)$ for some $s \in \mathcal{S}$ (strict improvement), or
2. $V^{\pi'}(s) = V^\pi(s)$ for all $s \in \mathcal{S}$ (optimality)

Proof. By construction, $Q^\pi(s,\pi'(s)) \geq Q^\pi(s,\pi(s)) = V^\pi(s)$ for all s. If inequality is strict

for any state, then by the policy evaluation equations, strict improvement propagates. If equality holds everywhere, then π satisfies the Bellman optimality equation and is optimal. □

3.3.4 Global Convergence

Theorem 3.9 (Finite Convergence of Policy Iteration)**.** For finite MDPs, policy iteration converges to an optimal policy in finitely many iterations. Specifically, the number of iterations is bounded by $|\mathcal{A}|^{|\mathcal{S}|}$.

Proof. Since each iteration either strictly improves the policy or terminates at optimality, and there are finitely many deterministic policies, convergence must occur in finite time. The bound follows from counting the total number of deterministic policies. □

3.4 Modified Policy Iteration

Modified policy iteration interpolates between value iteration and policy iteration, providing computational flexibility.

3.4.1 Algorithm and Convergence

Algorithm 4 Modified Policy Iteration

Input: Initial policy π_0, evaluation steps m
  $i \leftarrow 0$
  repeat
    $V \leftarrow$ arbitrary initialization
    for $k = 1, 2, \ldots, m$ **do**
      $V \leftarrow T^{\pi_i} V$
    end for
    $\pi_{i+1}(s) \leftarrow \text{argmax}_a [r(s,a) + \gamma \sum_{s'} P(s'|s,a) V(s')]$
    $i \leftarrow i + 1$
  until convergence

Theorem 3.10 (Convergence of Modified Policy Iteration)**.** Modified policy iteration with $m \geq 1$ evaluation steps converges to an optimal policy. The convergence rate depends on m:

- $m = 1$: reduces to value iteration with rate γ
- $m = \infty$: reduces to policy iteration with finite convergence
- $1 < m < \infty$: intermediate convergence rate

3.4.2 Optimal Choice of Evaluation Steps

The computational trade-off between evaluation and improvement can be optimized:

Theorem 3.11 (Optimal Evaluation Steps)**.** For modified policy iteration, the optimal number of evaluation steps m^* minimizes total computational cost:

$$m^* = \operatorname*{argmin}_m \left[\text{cost per iteration} \times \text{number of iterations}\right] \tag{3.18}$$

Under reasonable assumptions about computational costs, $m^* = O(\log(1/(1-\gamma)))$.

3.5 Asynchronous Dynamic Programming

Traditional DP algorithms update all states synchronously. Asynchronous variants can offer computational advantages and theoretical insights.

3.5.1 Gauss-Seidel Value Iteration

Algorithm 5 Gauss-Seidel Value Iteration

Order states $s_1, s_2, \ldots, s_n$
repeat
  for $i = 1, 2, \ldots, n$ **do**
    $V(s_i) \leftarrow \max_a \left[r(s_i, a) + \gamma \sum_j P(s_j|s_i, a)V(s_j)\right]$
  end for
until convergence

Theorem 3.12 (Convergence of Gauss-Seidel Value Iteration)**.** Gauss-Seidel value iteration converges to the optimal value function. The convergence rate can be faster than standard value iteration due to more frequent updates.

3.5.2 Prioritized Sweeping

Definition 3.1 (Bellman Error)**.** For state s and value function V, the Bellman error is:

$$\delta(s) = \left|\max_a \left[r(s, a) + \gamma \sum_{s'} P(s'|s, a)V(s')\right] - V(s)\right| \tag{3.19}$$

Prioritized sweeping updates states in order of decreasing Bellman error, focusing computation on states where updates will have the largest impact.

3.5.3 Real-Time Dynamic Programming

Real-time DP focuses updates on states visited by a simulated or actual agent trajectory.
Theorem 3.13 (Convergence of RTDP)**.** Under appropriate exploration conditions, real-time DP converges to optimal values on the states reachable under the optimal policy.

Algorithm 6 Prioritized Sweeping

```
Initialize priority queue 𝒬 with all states
while 𝒬 not empty do
  s ← state with highest priority in 𝒬
  Update V(s) using Bellman equation
  Remove s from 𝒬
  for each predecessor s′ of s do
    if Bellman error of s′ exceeds threshold then
      Add s′ to 𝒬 with updated priority
    end if
  end for
end while
```

Algorithm 7 Real-Time Dynamic Programming

```
Initialize current state s
repeat
  Update V(s) using Bellman equation
  Choose action a = argmax_a Q(s, a)
  Simulate or execute action: s ← s′ with probability P(s′|s, a)
until termination
```

3.6 Linear Programming Formulation

Dynamic programming problems can be formulated as linear programs, providing alternative solution methods and theoretical insights.

3.6.1 Primal LP Formulation

The optimal value function can be found by solving:

$$\min_{V} \quad \sum_{s \in \mathcal{S}} \alpha(s) V(s) \tag{3.20}$$

$$\text{subject to} \quad V(s) \geq r(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s,a) V(s') \quad \forall s, a \tag{3.21}$$

where $\alpha(s) > 0$ represents state weights.

Theorem 3.14 (LP-DP Equivalence)**.** The optimal solution to the linear program equals the optimal value function V^*.

3.6.2 Dual LP Formulation

The dual problem involves finding an optimal state-action visitation measure:

$$\max_{\mu} \quad \sum_{s,a} \mu(s,a) r(s,a) \tag{3.22}$$

$$\text{subject to} \quad \sum_{a} \mu(s,a) - \gamma \sum_{s',a'} \mu(s',a') P(s|s',a') = \alpha(s) \quad \forall s \tag{3.23}$$

$$\mu(s, a) \geq 0 \quad \forall s, a \tag{3.24}$$

Theorem 3.15 (Strong Duality)**.** Under mild conditions, strong duality holds between the primal and dual formulations, and complementary slackness conditions characterize optimal policies.

3.7 Connections to Classical Control Theory

3.7.1 Discrete-Time Optimal Control

Consider the discrete-time optimal control problem:

$$\sum_{t=0}^{T-1} L(x_t, u_t) + L_T(x_T) \tag{3.25}$$

$$\text{subject to} \quad x_{t+1} = f(x_t, u_t) + w_t \tag{3.26}$$

$$u_t \in \mathcal{U}(x_t) \tag{3.27}$$

The dynamic programming solution gives the Hamilton-Jacobi-Bellman equation:

$$V_t(x) = \min_{u \in \mathcal{U}(x)} [L(x, u) + \mathbb{E}[V_{t+1}(f(x, u) + w)]] \tag{3.28}$$

3.7.2 Stochastic Optimal Control

For stochastic control systems $dx_t = f(x_t, u_t)dt + \sigma(x_t, u_t)dW_t$, the continuous-time HJB equation is:

$$\frac{\partial V}{\partial t} + \min_u \left[L(x, u) + \frac{\partial V}{\partial x} f(x, u) + \frac{1}{2} \text{tr} \left(\sigma(x, u)^T \frac{\partial^2 V}{\partial x^2} \sigma(x, u) \right) \right] = 0 \tag{3.29}$$

3.7.3 Model Predictive Control (MPC)

MPC can be viewed as approximate dynamic programming with receding horizon:

Algorithm 8 Model Predictive Control

repeat
  Measure current state x_t
  Solve optimization problem over horizon $[t, t + H]$:
  $u_t^*, \ldots, u_{t+H-1}^* = \text{argmin} \sum_{k=0}^{H-1} L(x_{t+k}, u_{t+k}) + L_H(x_{t+H})$
  Apply u_t^* and advance to next time step
until termination

The connection to DP provides stability and performance guarantees for MPC under appropriate conditions.

3.8 Computational Considerations

3.8.1 Curse of Dimensionality

The computational complexity of DP algorithms scales exponentially with state space dimension:

- Memory: $O(|\mathcal{S}|)$ for value function storage
- Computation: $O(|\mathcal{S}|^2|\mathcal{A}|)$ per iteration
- For continuous spaces: requires discretization or function approximation

3.8.2 Approximate Dynamic Programming

To handle large state spaces, approximate DP uses function approximation:

$$V(s) \approx \sum_{i=1}^{n} w_i \phi_i(s) \tag{3.30}$$

where $\{\phi_i\}$ are basis functions and $\{w_i\}$ are parameters.

Theorem 3.16 (Error Propagation in Approximate DP)**.** If the approximation error is bounded by ϵ in supremum norm:

$$\|V - \hat{V}\|_\infty \leq \epsilon \tag{3.31}$$

then the policy derived from $\hat{V}$ satisfies:

$$\|V^{\hat{\pi}} - V^*\|_\infty \leq \frac{2\gamma\epsilon}{(1-\gamma)^2} \tag{3.32}$$

3.9 Chapter Summary

This chapter developed the mathematical foundations of dynamic programming:

- Principle of optimality and recursive decomposition
- Value iteration: convergence theory, error bounds, complexity analysis
- Policy iteration: linear algebra formulation, finite convergence
- Modified policy iteration and computational trade-offs
- Asynchronous variants: Gauss-Seidel, prioritized sweeping, real-time DP
- Linear programming formulations and duality theory
- Connections to classical optimal control and MPC

- Computational challenges and approximate methods

These algorithmic foundations provide the basis for understanding modern reinforcement learning methods. The next chapter begins our exploration of learning algorithms that estimate value functions from experience rather than exact knowledge of the MDP.

Part II

Core Algorithms and Theory

This part develops the fundamental learning algorithms that form the core of reinforcement learning. Moving beyond the dynamic programming methods of Part I, which assume complete knowledge of the MDP, we now consider algorithms that learn from experience through interaction with the environment.

We begin with Monte Carlo methods that estimate value functions from complete episodes. We then develop temporal difference learning, which enables learning from individual transitions. Finally, we examine Q-learning and SARSA, which learn action-value functions and form the foundation for more advanced algorithms.

Throughout this part, we emphasize mathematical rigor in convergence analysis while maintaining practical relevance for engineering applications. Each algorithm is developed with careful attention to assumptions, convergence conditions, and sample complexity bounds.

Chapter 4

Monte Carlo Methods

Monte Carlo methods form the foundation of model-free reinforcement learning, enabling value function estimation from sample episodes without requiring knowledge of the environment dynamics. This chapter develops the mathematical theory of Monte Carlo estimation in the RL context, with emphasis on convergence analysis and variance reduction techniques.

4.1 Monte Carlo Estimation Theory

Monte Carlo methods estimate expectations by sampling. In reinforcement learning, we use sample episodes to estimate value functions without requiring the transition probabilities or reward function.

4.1.1 Basic Monte Carlo Principle

Consider estimating the expectation $\mathbb{E}[X]$ of random variable X. The Monte Carlo estimator uses n independent samples $X_1, \ldots, X_n$:

$$\hat{\mu}_n = \frac{1}{n} \sum_{i=1}^{n} X_i \tag{4.1}$$

Theorem 4.1 (Strong Law of Large Numbers)**.** If $\mathbb{E}[|X|] < \infty$, then $\hat{\mu}_n \to \mathbb{E}[X]$ almost surely as $n \to \infty$.

Theorem 4.2 (Central Limit Theorem)**.** If $\mathrm{Var}(X) = \sigma^2 < \infty$, then:

$$\sqrt{n}(\hat{\mu}_n - \mathbb{E}[X]) \xrightarrow{d} \mathcal{N}(0, \sigma^2) \tag{4.2}$$

4.1.2 Application to Value Function Estimation

For policy π, the value function is:

$$V^\pi(s) = \mathbb{E}^\pi \left[\sum_{t=0}^{\infty} \gamma^t R_{t+1} \mid S_0 = s \right] \tag{4.3}$$

Monte Carlo estimation uses sample returns $G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$ from episodes starting in state s to estimate $V^\pi(s)$.

4.2 First-Visit vs. Every-Visit Methods

4.2.1 First-Visit Monte Carlo

Algorithm 9 First-Visit Monte Carlo Policy Evaluation

Input: Policy π to evaluate
  Initialize $V(s) \in \mathbb{R}$ arbitrarily for all $s \in \mathcal{S}$
  Initialize $Returns(s) \leftarrow$ empty list for all $s \in \mathcal{S}$
  repeat
    Generate episode following π: $S_0, A_0, R_1, S_1, A_1, R_2, \ldots, S_{T-1}, A_{T-1}, R_T$
    $G \leftarrow 0$
    for $t = T-1, T-2, \ldots, 0$ **do**
      $G \leftarrow \gamma G + R_{t+1}$
      if S_t not appear in $S_0, S_1, \ldots, S_{t-1}$ **then**
        Append G to $Returns(S_t)$
        $V(S_t) \leftarrow$ average$(Returns(S_t))$
      end if
    end for
  until convergence

4.2.2 Every-Visit Monte Carlo

Every-visit MC updates the value estimate every time a state is visited in an episode, not just the first time.

Theorem 4.3 (Convergence of First-Visit Monte Carlo)**.** First-visit Monte Carlo converges to $V^\pi(s)$ as the number of first visits to state s approaches infinity, assuming:

1. Episodes are generated according to policy π
2. Each state has non-zero probability of being the starting state
3. Returns have finite variance

Proof. Each first visit to state s provides an unbiased sample of the return. By the strong law of large numbers, the sample average converges to the true expectation. □

Theorem 4.4 (Convergence of Every-Visit Monte Carlo)**.** Every-visit Monte Carlo also converges to $V^\pi(s)$ under similar conditions, despite the correlation between visits within the same episode.

4.3 Variance Reduction Techniques

4.3.1 Incremental Implementation

Instead of storing all returns, we can update estimates incrementally:

$$V_{n+1}(s) = V_n(s) + \frac{1}{n+1}[G_n - V_n(s)] \tag{4.4}$$

More generally, with step size α:

$$V(s) \leftarrow V(s) + \alpha[G - V(s)] \tag{4.5}$$

4.3.2 Baseline Subtraction

To reduce variance, we can subtract a baseline $b(s)$ that doesn't depend on the action:

$$G_t - b(S_t) \tag{4.6}$$

The optimal baseline that minimizes variance is:

$$b^*(s) = \frac{\mathbb{E}[G_t^2 \mid S_t = s]}{\mathbb{E}[G_t \mid S_t = s]} = \mathbb{E}[G_t \mid S_t = s] = V^\pi(s) \tag{4.7}$$

4.3.3 Control Variates

For correlated random variable Y with known expectation $\mathbb{E}[Y] = \mu_Y$:

$$\hat{\mu}_{CV} = \hat{\mu}_X - c(\hat{\mu}_Y - \mu_Y) \tag{4.8}$$

The optimal coefficient is:

$$c^* = \frac{\mathrm{Cov}(X, Y)}{\mathrm{Var}(Y)} \tag{4.9}$$

4.4 Importance Sampling in RL

Importance sampling enables off-policy learning by weighting samples according to the ratio of target to behavior policy probabilities.

4.4.1 Ordinary Importance Sampling

To estimate $\mathbb{E}_\pi[X]$ using samples from policy μ:

$$\hat{\mu}_{IS} = \frac{1}{n}\sum_{i=1}^{n} \rho_i X_i \tag{4.10}$$

where $\rho_i = \frac{\pi(A_i|S_i)}{\mu(A_i|S_i)}$ is the importance sampling ratio.

Theorem 4.5 (Unbiasedness of Importance Sampling)**.** $\mathbb{E}[\hat{\mu}_{IS}] = \mathbb{E}_{\pi}[X]$ if $\mu(a|s) > 0$ whenever $\pi(a|s) > 0$.

4.4.2 Weighted Importance Sampling

To reduce variance when some importance weights are very large:

$$\hat{\mu}_{WIS} = \frac{\sum_{i=1}^{n} \rho_i X_i}{\sum_{i=1}^{n} \rho_i} \tag{4.11}$$

Theorem 4.6 (Bias-Variance Tradeoff)**.** Weighted importance sampling is biased but often has lower variance than ordinary importance sampling:

$$\text{Bias}[\hat{\mu}_{WIS}] \neq 0 \text{ (in general)} \tag{4.12}$$

$$\text{Var}[\hat{\mu}_{WIS}] \leq \text{Var}[\hat{\mu}_{IS}] \text{ (typically)} \tag{4.13}$$

4.4.3 Per-Decision Importance Sampling

For episodic tasks, the importance sampling ratio for a complete episode is:

$$\rho_{t:T-1} = \prod_{k=t}^{T-1} \frac{\pi(A_k|S_k)}{\mu(A_k|S_k)} \tag{4.14}$$

This can have very high variance. Per-decision importance sampling uses only the relevant portion of the trajectory.

4.5 Off-Policy Monte Carlo Methods

4.5.1 Off-Policy Policy Evaluation

4.5.2 Off-Policy Monte Carlo Control

4.6 Convergence Analysis and Sample Complexity

4.6.1 Finite Sample Analysis

Theorem 4.7 (Finite Sample Bound for Monte Carlo)**.** Let $V_n(s)$ be the Monte Carlo estimate after n visits to state s. Under bounded rewards $|R| \leq R_{max}$:

$$\mathbb{P}\left(|V_n(s) - V^{\pi}(s)| \geq \epsilon\right) \leq 2\exp\left(-\frac{2n\epsilon^2(1-\gamma)^2}{R_{max}^2}\right) \tag{4.15}$$

Algorithm 10 Off-Policy Monte Carlo Policy Evaluation

Input: Target policy π, behavior policy μ
  Initialize $V(s) \in \mathbb{R}$ arbitrarily for all $s \in \mathcal{S}$
  Initialize $C(s) \leftarrow 0$ for all $s \in \mathcal{S}$
  repeat
    Generate episode using μ: $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$
    $G \leftarrow 0$
    $W \leftarrow 1$
    for $t = T-1, T-2, \ldots, 0$ **do**
      $G \leftarrow \gamma G + R_{t+1}$
      $C(S_t) \leftarrow C(S_t) + W$
      $V(S_t) \leftarrow V(S_t) + \frac{W}{C(S_t)}[G - V(S_t)]$
      $W \leftarrow W \frac{\pi(A_t|S_t)}{\mu(A_t|S_t)}$
      if $W = 0$ **then**
        break
      end if
    end for
  until convergence

Algorithm 11 Off-Policy Monte Carlo Control

  Initialize $Q(s, a) \in \mathbb{R}$ arbitrarily for all s, a
  Initialize $C(s, a) \leftarrow 0$ for all s, a
  Initialize $\pi(s) \leftarrow \text{argmax}_a Q(s, a)$ for all s
  repeat
    Choose any soft policy μ (e.g., ϵ-greedy)
    Generate episode using μ
    $G \leftarrow 0$
    $W \leftarrow 1$
    for $t = T-1, T-2, \ldots, 0$ **do**
      $G \leftarrow \gamma G + R_{t+1}$
      $C(S_t, A_t) \leftarrow C(S_t, A_t) + W$
      $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{W}{C(S_t, A_t)}[G - Q(S_t, A_t)]$
      $\pi(S_t) \leftarrow \text{argmax}_a Q(S_t, a)$
      if $A_t \neq \pi(S_t)$ **then**
        break
      end if
      $W \leftarrow W \frac{1}{\mu(A_t|S_t)}$
    end for
  until convergence

4.6.2 Asymptotic Convergence Rate

Theorem 4.8 (Central Limit Theorem for Monte Carlo)**.** If $\text{Var}^\pi[G_t|S_t = s] = \sigma^2(s) < \infty$, then:

$$\sqrt{n}(V_n(s) - V^\pi(s)) \xrightarrow{d} \mathcal{N}(0, \sigma^2(s)) \tag{4.16}$$

This gives the convergence rate $O(n^{-1/2})$, which is slower than the $O(n^{-1})$ rate achievable by temporal difference methods under certain conditions.

4.6.3 Sample Complexity

Theorem 4.9 (Sample Complexity of Monte Carlo)**.** To achieve ϵ-accurate value function estimation with probability $1 - \delta$:

$$n \geq \frac{R_{max}^2 \log(2/\delta)}{2\epsilon^2(1-\gamma)^2} \tag{4.17}$$

samples are sufficient.

4.7 Practical Considerations

4.7.1 Exploration vs. Exploitation

Monte Carlo control methods face the exploration-exploitation dilemma. Common approaches:

Exploring Starts: Assume episodes start in randomly selected state-action pairs.

ϵ-Greedy Policies: Use soft policies that maintain exploration:

$$\pi(a|s) = \begin{cases} 1 - \epsilon + \frac{\epsilon}{|\mathcal{A}(s)|} & \text{if } a = \text{argmax}_a\, Q(s,a) \\ \frac{\epsilon}{|\mathcal{A}(s)|} & \text{otherwise} \end{cases} \tag{4.18}$$

4.7.2 Function Approximation

For large state spaces, we approximate value functions:

$$V(s) \approx \hat{V}(s, \mathbf{w}) = \mathbf{w}^T \boldsymbol{\phi}(s) \tag{4.19}$$

The Monte Carlo update becomes:

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha[G_t - \hat{V}(S_t, \mathbf{w})]\nabla_{\mathbf{w}} \hat{V}(S_t, \mathbf{w}) \tag{4.20}$$

Theorem 4.10 (Convergence with Linear Function Approximation)**.** Under linear function approximation with linearly independent features, Monte Carlo methods converge to the best linear approximation in the L^2 norm weighted by the stationary distribution.

4.8 Chapter Summary

This chapter established the foundations of Monte Carlo methods in reinforcement learning:

- Monte Carlo estimation theory and convergence properties
- First-visit vs. every-visit methods with convergence guarantees
- Variance reduction techniques: baselines, control variates, importance sampling
- Off-policy learning through importance sampling with bias-variance analysis
- Sample complexity bounds and convergence rates
- Practical considerations for exploration and function approximation

Monte Carlo methods provide unbiased estimates and are conceptually simple, but they require complete episodes and have slower convergence than temporal difference methods. The next chapter develops temporal difference learning, which enables learning from individual transitions.

Chapter 5

Temporal Difference Learning

Temporal Difference (TD) learning combines ideas from Monte Carlo methods and dynamic programming, enabling learning from incomplete episodes while maintaining the model-free nature of Monte Carlo methods. This chapter develops the mathematical theory of TD learning with emphasis on convergence analysis and the fundamental bias-variance tradeoff.

5.1 TD(0) Algorithm and Mathematical Analysis

5.1.1 Basic TD(0) Update

The core insight of temporal difference learning is to use the current estimate of the successor state's value to update the current state's value:

$$V(S_t) \leftarrow V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)] \tag{5.1}$$

The TD error is defined as:

$$\delta_t = R_{t+1} + \gamma V(S_{t+1}) - V(S_t) \tag{5.2}$$

5.1.2 Relationship to Bellman Equation

The TD(0) update can be viewed as a stochastic approximation to the Bellman equation. The expected TD update is:

$$\mathbb{E}[\delta_t | S_t = s] = \mathbb{E}[R_{t+1} + \gamma V(S_{t+1}) - V(S_t) | S_t = s] \tag{5.3}$$

$$= \sum_{s',r} p(s', r | s, \pi(s))[r + \gamma V(s') - V(s)] \tag{5.4}$$

$$= (T^\pi V)(s) - V(s) \tag{5.5}$$

Algorithm 12 Tabular TD(0) Policy Evaluation

Input: Policy π to evaluate, step size $\alpha \in (0, 1]$
  Initialize $V(s) \in \mathbb{R}$ arbitrarily for all $s \in \mathcal{S}$, except $V(\text{terminal}) = 0$
  repeat
    Initialize S
    repeat
      $A \leftarrow$ action given by π for S
      Take action A, observe R, S'
      $V(S) \leftarrow V(S) + \alpha[R + \gamma V(S') - V(S)]$
      $S \leftarrow S'$
    until S is terminal
  until convergence or sufficient accuracy

where T^π is the Bellman operator for policy π.

5.1.3 Convergence Analysis

Theorem 5.1 (Convergence of TD(0) - Tabular Case)**.** For the tabular case with appropriate step size sequence $\{\alpha_t\}$ satisfying:

$$\sum_{t=0}^{\infty} \alpha_t = \infty \tag{5.6}$$

$$\sum_{t=0}^{\infty} \alpha_t^2 < \infty \tag{5.7}$$

TD(0) converges to V^π with probability 1.

Proof Sketch. The proof uses stochastic approximation theory. Define the ODE:

$$\frac{dV}{dt} = \mathbb{E}[\delta_t | V] = T^\pi V - V \tag{5.8}$$

Since T^π is a contraction, the unique fixed point is V^π. The stochastic approximation theorem ensures convergence of the discrete updates to the ODE solution. □

5.2 Bias-Variance Tradeoff in TD Methods

5.2.1 Bias Analysis

TD(0) uses the biased estimate $R_{t+1} + \gamma V(S_{t+1})$ as a target for $V(S_t)$, while Monte Carlo uses the unbiased estimate G_t.

Theorem 5.2 (Bias of TD Target)**.** The TD target $R_{t+1} + \gamma V(S_{t+1})$ has bias:

$$\text{Bias}[R_{t+1} + \gamma V(S_{t+1})] = \gamma[\hat{V}(S_{t+1}) - V^\pi(S_{t+1})] \tag{5.9}$$

where $\hat{V}$ is the current estimate.

5.2.2 Variance Analysis

Theorem 5.3 (Variance Comparison)**.** Under the assumption that value function errors are small, the variance of the TD target is approximately:

$$\text{Var}[R_{t+1} + \gamma V(S_{t+1})] \approx \text{Var}[R_{t+1}] \tag{5.10}$$

while the Monte Carlo target has variance:

$$\text{Var}[G_t] = \text{Var}\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}\right] \tag{5.11}$$

which is typically much larger.

5.2.3 Mean Squared Error Decomposition

$$\text{MSE} = \text{Bias}^2 + \text{Variance} + \text{Noise} \tag{5.12}$$

TD methods trade increased bias for reduced variance, often resulting in lower overall MSE and faster convergence.

5.3 TD() and Eligibility Traces

TD() provides a family of algorithms that interpolate between TD(0) and Monte Carlo methods through the use of eligibility traces.

5.3.1 Forward View: n-step Returns

The n-step return combines rewards from the next n steps with the estimated value of the state reached after n steps:

$$G_t^{(n)} = R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n V(S_{t+n}) \tag{5.13}$$

The n-step TD update is:

$$V(S_t) \leftarrow V(S_t) + \alpha[G_t^{(n)} - V(S_t)] \tag{5.14}$$

5.3.2 -Return

The -return combines all n-step returns:

$$G_t^{\lambda} = (1-\lambda) \sum_{n=1}^{\infty} \lambda^{n-1} G_t^{(n)} \tag{5.15}$$

Theorem 5.4 (-Return Properties)**.** The -return satisfies:

$$G_t^\lambda = R_{t+1} + \gamma[(1-\lambda)V(S_{t+1}) + \lambda G_{t+1}^\lambda] \tag{5.16}$$

$$\lim_{\lambda \to 0} G_t^\lambda = R_{t+1} + \gamma V(S_{t+1}) \quad \text{(TD(0))} \tag{5.17}$$

$$\lim_{\lambda \to 1} G_t^\lambda = G_t \quad \text{(Monte Carlo)} \tag{5.18}$$

5.3.3 Backward View: Eligibility Traces

Eligibility traces provide an online, incremental implementation of TD():

$$\delta_t = R_{t+1} + \gamma V(S_{t+1}) - V(S_t) \tag{5.19}$$

$$e_t(s) = \begin{cases} \gamma\lambda e_{t-1}(s) + 1 & \text{if } s = S_t \\ \gamma\lambda e_{t-1}(s) & \text{if } s \neq S_t \end{cases} \tag{5.20}$$

$$V(s) \leftarrow V(s) + \alpha\delta_t e_t(s) \quad \forall s \tag{5.21}$$

Algorithm 13 TD() with Eligibility Traces

Input: Policy π, step size α, trace decay λ
  Initialize $V(s) \in \mathbb{R}$ arbitrarily for all s
  repeat
    Initialize S, $e(s) = 0$ for all s
    repeat
      $A \leftarrow$ action given by π for S
      Take action A, observe R, S'
      $\delta \leftarrow R + \gamma V(S') - V(S)$
      $e(S) \leftarrow e(S) + 1$
      for all s **do**
        $V(s) \leftarrow V(s) + \alpha\delta e(s)$
        $e(s) \leftarrow \gamma\lambda e(s)$
      end for
      $S \leftarrow S'$
    until S is terminal
  until convergence

5.3.4 Equivalence Theorem

Theorem 5.5 (Forward-Backward Equivalence)**.** Under certain conditions, the forward view (using -returns) and backward view (using eligibility traces) produce identical updates when applied offline to a complete episode.

5.4 Convergence Theory for Linear Function Approximation

When the state space is large, we use function approximation:

$$V(s) \approx \hat{V}(s, \mathbf{w}) = \mathbf{w}^T \boldsymbol{\phi}(s) \tag{5.22}$$

The TD(0) update becomes:

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha[R_{t+1} + \gamma \mathbf{w}_t^T \boldsymbol{\phi}(S_{t+1}) - \mathbf{w}_t^T \boldsymbol{\phi}(S_t)]\boldsymbol{\phi}(S_t) \tag{5.23}$$

5.4.1 Projected Bellman Equation

Under linear function approximation, TD(0) converges to the solution of the projected Bellman equation:

$$\mathbf{w}^* = \arg\min_{\mathbf{w}} \|\boldsymbol{\Phi}\mathbf{w} - T^\pi(\boldsymbol{\Phi}\mathbf{w})\|_{\mathbf{D}}^2 \tag{5.24}$$

where $\boldsymbol{\Phi}$ is the feature matrix and $\mathbf{D}$ is a diagonal matrix of state visitation probabilities.

Theorem 5.6 (Convergence of Linear TD(0))**.** Under linear function approximation, TD(0) converges to:

$$\mathbf{w}^* = (\boldsymbol{\Phi}^T \mathbf{D} \boldsymbol{\Phi})^{-1} \boldsymbol{\Phi}^T \mathbf{D} \mathbf{r}^\pi \tag{5.25}$$

where $\mathbf{r}^\pi$ is the expected reward vector.

5.4.2 Error Bounds

Theorem 5.7 (Approximation Error Bound)**.** Let V^* be the optimal value function and $\hat{V}^*$ be the best linear approximation. Then:

$$\|V^\pi - \hat{V}^\pi\|_{\mathbf{D}} \leq \frac{1}{1-\gamma} \min_{\mathbf{w}} \|V^\pi - \boldsymbol{\Phi}\mathbf{w}\|_{\mathbf{D}} \tag{5.26}$$

5.5 Comparison with Monte Carlo and DP Methods

5.5.1 Computational Complexity

Method	Memory	Computation per Step	Episode Completion
DP	$O(\lvert\mathcal{S}\rvert^2 \lvert\mathcal{A}\rvert)$	$O(\lvert\mathcal{S}\rvert^2 \lvert\mathcal{A}\rvert)$	Not Required
MC	$O(\lvert\mathcal{S}\rvert)$	$O(1)$	Required
TD	$O(\lvert\mathcal{S}\rvert)$	$O(1)$	Not Required

5.5.2 Sample Efficiency

Theorem 5.8 (Sample Complexity Comparison)**.** Under certain regularity conditions:

- TD methods: $O(\frac{1}{\epsilon^2(1-\gamma)^2})$ samples for ϵ-accuracy

- MC methods: $O(\frac{1}{\epsilon^2(1-\gamma)^4})$ samples for ϵ-accuracy

TD methods often have better sample efficiency due to lower variance, despite being biased.

5.5.3 Bootstrapping vs. Sampling

Bootstrapping: Using estimates of successor states (DP, TD) **Sampling:** Using actual experience (MC, TD)

TD methods combine both, leading to:

- Faster learning than MC (bootstrapping)
- Model-free nature (sampling)
- Online learning capability

5.6 Advanced Topics

5.6.1 Multi-step Methods

The n-step TD methods generalize between TD(0) and Monte Carlo:

$$V(S_t) \leftarrow V(S_t) + \alpha[G_t^{(n)} - V(S_t)] \tag{5.27}$$

Theorem 5.9 (Optimal Step Size)**.** For n-step methods, there exists an optimal n that minimizes mean squared error, typically $n \in [3, 10]$ for many problems.

5.6.2 True Online TD()

The classical TD() is not equivalent to the forward view when using function approximation. True online TD() corrects this:

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha\delta_t\mathbf{z}_t + \alpha(\mathbf{w}_t^T\boldsymbol{\phi}_t - \mathbf{w}_{t-1}^T\boldsymbol{\phi}_t)(\mathbf{z}_t - \boldsymbol{\phi}_t) \tag{5.28}$$

$$\mathbf{z}_{t+1} = \gamma\lambda\mathbf{z}_t + \boldsymbol{\phi}_{t+1} - \alpha\gamma\lambda(\mathbf{z}_t^T\boldsymbol{\phi}_{t+1})\boldsymbol{\phi}_{t+1} \tag{5.29}$$

5.6.3 Gradient TD Methods

To handle function approximation more rigorously, gradient TD methods minimize the mean squared projected Bellman error:

$$\text{MSPBE}(\mathbf{w}) = \|\boldsymbol{\Pi}(\mathbf{T}^\pi\hat{\mathbf{v}} - \hat{\mathbf{v}})\|_{\mathbf{D}}^2 \tag{5.30}$$

$$\nabla\text{MSPBE}(\mathbf{w}) = 2\boldsymbol{\Phi}^T\mathbf{D}(\boldsymbol{\Pi}(\mathbf{T}^\pi\hat{\mathbf{v}} - \hat{\mathbf{v}})) \tag{5.31}$$

5.7 Chapter Summary

This chapter developed the mathematical foundations of temporal difference learning:

- TD(0) algorithm with convergence analysis using stochastic approximation theory
- Bias-variance tradeoff analysis showing TD's advantage in variance reduction
- TD() and eligibility traces providing a spectrum between TD(0) and Monte Carlo
- Convergence theory for linear function approximation with error bounds
- Comparative analysis with Monte Carlo and dynamic programming methods
- Advanced topics including multi-step methods and gradient TD approaches

Temporal difference learning provides the foundation for many modern RL algorithms, combining the best aspects of Monte Carlo and dynamic programming approaches. The next chapter extends these ideas to action-value methods with Q-learning and SARSA.

Chapter 6

Chapter 06 Title

This chapter will cover...

Part III

Function Approximation and Deep Learning

This part bridges classical reinforcement learning with modern deep learning approaches. When state and action spaces are large or continuous, exact representation of value functions becomes intractable. Function approximation provides the mathematical framework for representing value functions compactly while maintaining theoretical guarantees.

We begin with linear function approximation, which provides strong theoretical foundations and convergence guarantees. We then explore the integration of neural networks into reinforcement learning, leading to the deep reinforcement learning revolution. Finally, we develop policy gradient methods that directly optimize parameterized policies.

The treatment emphasizes both the mathematical foundations that ensure stability and convergence, and the practical considerations necessary for successful implementation in engineering systems.

Chapter 7

Chapter 07 Title

This chapter will cover...

Chapter 8

Chapter 08 Title

This chapter will cover...

Chapter 9

Chapter 09 Title

This chapter will cover...

Part IV

Advanced Topics

This part explores advanced reinforcement learning topics that extend beyond the basic framework to handle more complex scenarios. We examine continuous control problems that arise frequently in engineering applications, multi-agent systems where multiple learners interact, model-based approaches that learn environment dynamics, and the fundamental exploration-exploitation tradeoff.

These advanced topics require sophisticated mathematical tools and careful analysis of convergence properties, stability, and sample complexity. The treatment provides both theoretical insights and practical guidance for tackling complex real-world problems.

Chapter 10

Chapter 10 Title

This chapter will cover...

Chapter 11

Chapter 11 Title

This chapter will cover...

Chapter 12

Chapter 12 Title

This chapter will cover...

Chapter 13

Chapter 13 Title

This chapter will cover...

Part V

Implementation and Practice

This part focuses on the practical aspects of implementing and deploying reinforcement learning systems. Moving from theory to practice requires careful consideration of computational efficiency, software engineering best practices, validation methodologies, and deployment considerations.

We examine computational techniques for scaling RL algorithms, software frameworks and tools, and methodologies for validating and deploying RL systems in production environments. The treatment emphasizes engineering best practices while maintaining theoretical rigor.

Chapter 14

Chapter 14 Title

This chapter will cover...

Chapter 15

Chapter 15 Title

This chapter will cover...

Chapter 16

Chapter 16 Title

This chapter will cover...

Part VI

Future Directions

This final part explores emerging paradigms and future directions in reinforcement learning. We examine meta-learning approaches that enable rapid adaptation to new tasks, integration with other fields such as classical optimization and control theory, and discuss open challenges and future research directions.

The treatment provides perspective on the current state of the field and identifies promising directions for future research and development, particularly relevant for engineer-mathematicians working at the intersection of theory and practice.

Chapter 17

Chapter 17 Title

This chapter will cover...

Chapter 18

Chapter 18 Title

This chapter will cover...

Mathematical Reference

This appendix provides a comprehensive mathematical reference for concepts used throughout the book. It serves as a quick reference for key mathematical tools and theorems.

.1 Matrix Calculus for RL

.1.1 Gradients and Jacobians

For scalar function $f : \mathbb{R}^n \to \mathbb{R}$, the gradient is:

$$\nabla f(x) = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \vdots \\ \frac{\partial f}{\partial x_n} \end{bmatrix} \tag{1}$$

For vector function $F : \mathbb{R}^n \to \mathbb{R}^m$, the Jacobian is:

$$J_F(x) = \begin{bmatrix} \frac{\partial F_1}{\partial x_1} & \cdots & \frac{\partial F_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial F_m}{\partial x_1} & \cdots & \frac{\partial F_m}{\partial x_n} \end{bmatrix} \tag{2}$$

.1.2 Chain Rule

For composite functions $h(x) = f(g(x))$:

$$\nabla h(x) = J_g(x)^T \nabla f(g(x)) \tag{3}$$

.1.3 Common Derivatives

$$\frac{\partial}{\partial x} x^T A x = (A + A^T)x \tag{4}$$

$$\frac{\partial}{\partial x} a^T x = a \tag{5}$$

$$\frac{\partial}{\partial X} \text{tr}(AXB) = A^T B^T \tag{6}$$

$$\frac{\partial}{\partial X} \log \det(X) = (X^{-1})^T \tag{7}$$

.2 Probability Distributions Commonly Used

.2.1 Discrete Distributions

Bernoulli Distribution: $X \sim \text{Bernoulli}(p)$

$$P(X = 1) = p, \quad P(X = 0) = 1 - p \tag{8}$$

$$\mathbb{E}[X] = p, \quad \text{Var}(X) = p(1 - p) \tag{9}$$

Categorical Distribution: $X \sim \text{Categorical}(\mathbf{p})$

$$P(X = k) = p_k, \quad \sum_{k=1}^{K} p_k = 1 \tag{10}$$

$$\mathbb{E}[X] = \sum_{k=1}^{K} k p_k \tag{11}$$

.2.2 Continuous Distributions

Normal Distribution: $X \sim \mathcal{N}(\mu, \sigma^2)$

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right) \tag{12}$$

$$\mathbb{E}[X] = \mu, \quad \text{Var}(X) = \sigma^2 \tag{13}$$

Multivariate Normal: $\mathbf{X} \sim \mathcal{N}(\boldsymbol{\mu}, \Sigma)$

$$f(\mathbf{x}) = \frac{1}{(2\pi)^{k/2}|\Sigma|^{1/2}} \exp\left(-\frac{1}{2}(\mathbf{x} - \boldsymbol{\mu})^T \Sigma^{-1} (\mathbf{x} - \boldsymbol{\mu})\right) \tag{14}$$

$$\mathbb{E}[\mathbf{X}] = \boldsymbol{\mu}, \quad \text{Cov}(\mathbf{X}) = \Sigma \tag{15}$$

.3 Optimization Algorithms Summary

.3.1 Gradient Descent Variants

Vanilla Gradient Descent:

$$\theta_{t+1} = \theta_t - \alpha \nabla f(\theta_t) \tag{16}$$

Momentum:

$$v_{t+1} = \beta v_t + \nabla f(\theta_t) \tag{17}$$

$$\theta_{t+1} = \theta_t - \alpha v_{t+1} \tag{18}$$

Adam:

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)\nabla f(\theta_t) \tag{19}$$

$$v_t = \beta_2 v_{t-1} + (1-\beta_2)[\nabla f(\theta_t)]^2 \tag{20}$$

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1-\beta_2^t} \tag{21}$$

$$\theta_{t+1} = \theta_t - \alpha \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} \tag{22}$$

.4 Convergence Analysis Techniques

.4.1 Lyapunov Functions

A function $V : \mathcal{X} \to \mathbb{R}_+$ is a Lyapunov function for dynamical system $x_{t+1} = f(x_t)$ if:

1. $V(x) > 0$ for $x \neq x^*$ and $V(x^*) = 0$
2. $V(f(x)) - V(x) \leq 0$ for all x

.4.2 Martingale Convergence Theorem

Theorem .1 (Martingale Convergence)**.** Let $\{X_t\}$ be a supermartingale that is bounded below. Then X_t converges almost surely to a finite random variable.

.4.3 Robbins-Monro Conditions

For stochastic approximation algorithm $\theta_{t+1} = \theta_t + \alpha_t H(\theta_t, \xi_t)$, convergence occurs under:

$$\sum_{t=0}^{\infty} \alpha_t = \infty \tag{23}$$

$$\sum_{t=0}^{\infty} \alpha_t^2 < \infty \tag{24}$$

$$\mathbb{E}[H(\theta, \xi)] = h(\theta) \text{ has unique zero at } \theta^* \tag{25}$$

Implementation Templates

This appendix provides implementation templates and code examples for key reinforcement learning algorithms. The code is provided in Python and follows best practices for numerical stability and computational efficiency.

.5 Basic RL Algorithm Implementations

.5.1 Value Iteration

```python
import numpy as np

def value_iteration(P, R, gamma, tol=1e-6, max_iter=1000):
    """
    Value iteration algorithm for finite MDPs.

    Parameters:
    P: transition probability tensor [S x A x S]
    R: reward matrix [S x A]
    gamma: discount factor
    tol: convergence tolerance
    max_iter: maximum iterations

    Returns:
    V: optimal value function
    policy: optimal policy
    """
    S, A = R.shape
    V = np.zeros(S)

    for i in range(max_iter):
        V_old = V.copy()

        # Bellman optimality operator
        Q = R + gamma * np.sum(P * V[None, None, :], axis=2)
        V = np.max(Q, axis=1)

        # Check convergence
        if np.max(np.abs(V - V_old)) < tol:
            break
```

```
31
32      # Extract optimal policy
33      Q = R + gamma * np.sum(P * V[None, None, :], axis=2)
34      policy = np.argmax(Q, axis=1)
35
36      return V, policy
```

Listing 1: Value Iteration Implementation

.5.2 Q-Learning

```
1  import numpy as np
2  from collections import defaultdict
3
4  class QLearning:
5      def __init__(self, n_states, n_actions, alpha=0.1, gamma=0.99, epsilon
           =0.1):
6          self.n_states = n_states
7          self.n_actions = n_actions
8          self.alpha = alpha
9          self.gamma = gamma
10         self.epsilon = epsilon
11         self.Q = np.zeros((n_states, n_actions))
12
13     def select_action(self, state):
14         """Epsilon-greedy action selection"""
15         if np.random.random() < self.epsilon:
16             return np.random.randint(self.n_actions)
17         else:
18             return np.argmax(self.Q[state])
19
20     def update(self, state, action, reward, next_state, done):
21         """Q-learning update rule"""
22         if done:
23             target = reward
24         else:
25             target = reward + self.gamma * np.max(self.Q[next_state])
26
27         td_error = target - self.Q[state, action]
28         self.Q[state, action] += self.alpha * td_error
29
30         return td_error
31
32     def get_policy(self):
33         """Extract greedy policy"""
34         return np.argmax(self.Q, axis=1)
```

Listing 2: Q-Learning Implementation

.5.3 Policy Gradient (REINFORCE)

```python
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
from torch.distributions import Categorical

class PolicyNetwork(nn.Module):
    def __init__(self, state_dim, action_dim, hidden_dim=128):
        super(PolicyNetwork, self).__init__()
        self.fc1 = nn.Linear(state_dim, hidden_dim)
        self.fc2 = nn.Linear(hidden_dim, hidden_dim)
        self.fc3 = nn.Linear(hidden_dim, action_dim)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = F.softmax(self.fc3(x), dim=-1)
        return x

class REINFORCE:
    def __init__(self, state_dim, action_dim, lr=1e-3, gamma=0.99):
        self.policy = PolicyNetwork(state_dim, action_dim)
        self.optimizer = optim.Adam(self.policy.parameters(), lr=lr)
        self.gamma = gamma

    def select_action(self, state):
        state = torch.FloatTensor(state).unsqueeze(0)
        probs = self.policy(state)
        m = Categorical(probs)
        action = m.sample()
        return action.item(), m.log_prob(action)

    def update(self, log_probs, rewards):
        """Update policy using REINFORCE algorithm"""
        # Compute discounted returns
        returns = []
        G = 0
        for r in reversed(rewards):
            G = r + self.gamma * G
            returns.insert(0, G)

        returns = torch.FloatTensor(returns)
        returns = (returns - returns.mean()) / (returns.std() + 1e-8)

        # Compute policy loss
        policy_loss = []
        for log_prob, G in zip(log_probs, returns):
            policy_loss.append(-log_prob * G)

        policy_loss = torch.stack(policy_loss).sum()
```

```
51
52          # Update policy
53          self.optimizer.zero_grad()
54          policy_loss.backward()
55          self.optimizer.step()
56
57          return policy_loss.item()
```

Listing 3: REINFORCE Implementation

.6 Environment Interface Specifications

.6.1 OpenAI Gym Compatible Environment

```
1  import gym
2  from gym import spaces
3  import numpy as np
4
5  class CustomEnvironment(gym.Env):
6      """Template for custom RL environment"""
7
8      def __init__(self):
9          super(CustomEnvironment, self).__init__()
10
11         # Define action and observation spaces
12         self.action_space = spaces.Discrete(4)  # Example: 4 discrete
               actions
13         self.observation_space = spaces.Box(
14             low=-np.inf, high=np.inf, shape=(4,), dtype=np.float32
15         )
16
17         # Initialize state
18         self.state = None
19         self.episode_length = 0
20         self.max_episode_length = 1000
21
22     def reset(self):
23         """Reset environment to initial state"""
24         self.state = self._get_initial_state()
25         self.episode_length = 0
26         return self.state
27
28     def step(self, action):
29         """Execute one step in the environment"""
30         # Validate action
31         assert self.action_space.contains(action), f"Invalid action: {
               action}"
32
33         # Update state based on action
34         self.state = self._update_state(self.state, action)
```

```
35
36          # Compute reward
37          reward = self._compute_reward(self.state, action)
38
39          # Check if episode is done
40          done = self._is_done()
41
42          # Additional info
43          info = {}
44
45          self.episode_length += 1
46
47          return self.state, reward, done, info
48
49      def render(self, mode='human'):
50          """Render the environment"""
51          print(f"State: {self.state}")
52
53      def _get_initial_state(self):
54          """Get initial state (implement based on problem)"""
55          return np.random.randn(4)
56
57      def _update_state(self, state, action):
58          """Update state based on action (implement based on problem)"""
59          # Example: simple dynamics
60          next_state = state + 0.1 * action
61          return next_state
62
63      def _compute_reward(self, state, action):
64          """Compute reward (implement based on problem)"""
65          # Example: negative squared distance from origin
66          return -np.sum(state**2)
67
68      def _is_done(self):
69          """Check if episode should terminate"""
70          return self.episode_length >= self.max_episode_length
```

Listing 4: Custom Environment Template

.7 Logging and Visualization Code

.7.1 Training Logger

```
1  import matplotlib.pyplot as plt
2  import numpy as np
3  from collections import deque
4  import json
5
6  class TrainingLogger:
7      def __init__(self, window_size=100):
```

```
        self.metrics = {}
        self.window_size = window_size
        self.episode_rewards = deque(maxlen=window_size)
        self.episode_lengths = deque(maxlen=window_size)

    def log_episode(self, episode, reward, length, **kwargs):
        """Log episode statistics"""
        self.episode_rewards.append(reward)
        self.episode_lengths.append(length)

        # Log additional metrics
        for key, value in kwargs.items():
            if key not in self.metrics:
                self.metrics[key] = deque(maxlen=self.window_size)
            self.metrics[key].append(value)

        # Print progress
        if episode % 100 == 0:
            avg_reward = np.mean(self.episode_rewards)
            avg_length = np.mean(self.episode_lengths)
            print(f"Episode {episode}: Avg Reward = {avg_reward:.2f}, "
                  f"Avg Length = {avg_length:.2f}")

    def plot_training_curves(self, save_path=None):
        """Plot training curves"""
        fig, axes = plt.subplots(2, 2, figsize=(12, 8))

        # Episode rewards
        axes[0, 0].plot(self.episode_rewards)
        axes[0, 0].set_title('Episode Rewards')
        axes[0, 0].set_xlabel('Episode')
        axes[0, 0].set_ylabel('Reward')

        # Episode lengths
        axes[0, 1].plot(self.episode_lengths)
        axes[0, 1].set_title('Episode Lengths')
        axes[0, 1].set_xlabel('Episode')
        axes[0, 1].set_ylabel('Length')

        # Moving averages
        if len(self.episode_rewards) > 10:
            window = min(50, len(self.episode_rewards) // 4)
            moving_avg = np.convolve(self.episode_rewards,
                                     np.ones(window)/window, mode='valid')
            axes[1, 0].plot(moving_avg)
            axes[1, 0].set_title(f'Moving Average Reward (window={window})'
                )
            axes[1, 0].set_xlabel('Episode')
            axes[1, 0].set_ylabel('Reward')

        # Additional metrics
```

```
        if self.metrics:
            metric_name = list(self.metrics.keys())[0]
            axes[1, 1].plot(self.metrics[metric_name])
            axes[1, 1].set_title(metric_name)
            axes[1, 1].set_xlabel('Episode')
            axes[1, 1].set_ylabel('Value')

        plt.tight_layout()

        if save_path:
            plt.savefig(save_path)
        plt.show()

    def save_metrics(self, filepath):
        """Save metrics to JSON file"""
        data = {
            'episode_rewards': list(self.episode_rewards),
            'episode_lengths': list(self.episode_lengths),
            'metrics': {k: list(v) for k, v in self.metrics.items()}
        }
        with open(filepath, 'w') as f:
            json.dump(data, f, indent=2)
```

Listing 5: Training Logger

.8 Performance Benchmarking Utilities

.8.1 Algorithm Comparison Framework

```
import time
import numpy as np
from typing import Dict, List, Callable

class AlgorithmComparison:
    def __init__(self, environment_factory: Callable):
        self.environment_factory = environment_factory
        self.results = {}

    def run_algorithm(self, algorithm_name: str, algorithm_class,
                      algorithm_params: Dict, n_runs: int = 5,
                      n_episodes: int = 1000):
        """Run algorithm multiple times and collect statistics"""
        print(f"Running {algorithm_name}...")

        run_results = []

        for run in range(n_runs):
            print(f"  Run {run + 1}/{n_runs}")

            # Create fresh environment and algorithm
```

```
            env = self.environment_factory()
            algorithm = algorithm_class(**algorithm_params)

            # Track performance
            episode_rewards = []
            episode_lengths = []
            start_time = time.time()

            for episode in range(n_episodes):
                state = env.reset()
                episode_reward = 0
                episode_length = 0
                done = False

                while not done:
                    action = algorithm.select_action(state)
                    next_state, reward, done, _ = env.step(action)

                    # Update algorithm
                    if hasattr(algorithm, 'update'):
                        algorithm.update(state, action, reward, next_state,
                            done)

                    state = next_state
                    episode_reward += reward
                    episode_length += 1

                episode_rewards.append(episode_reward)
                episode_lengths.append(episode_length)

            training_time = time.time() - start_time

            run_results.append({
                'episode_rewards': episode_rewards,
                'episode_lengths': episode_lengths,
                'training_time': training_time,
                'final_performance': np.mean(episode_rewards[-100:])
            })

        self.results[algorithm_name] = run_results

    def compare_algorithms(self):
        """Generate comparison statistics"""
        comparison = {}

        for alg_name, results in self.results.items():
            final_perfs = [r['final_performance'] for r in results]
            training_times = [r['training_time'] for r in results]

            comparison[alg_name] = {
                'mean_performance': np.mean(final_perfs),
```

```
                'std_performance': np.std(final_perfs),
                'mean_training_time': np.mean(training_times),
                'std_training_time': np.std(training_times)
            }

        return comparison

    def plot_comparison(self):
        """Plot algorithm comparison"""
        plt.figure(figsize=(15, 5))

        # Performance comparison
        plt.subplot(1, 3, 1)
        alg_names = list(self.results.keys())
        performances = []
        errors = []

        for alg_name in alg_names:
            final_perfs = [r['final_performance'] for r in self.results[
                alg_name]]
            performances.append(np.mean(final_perfs))
            errors.append(np.std(final_perfs))

        plt.bar(alg_names, performances, yerr=errors, capsize=5)
        plt.title('Final Performance Comparison')
        plt.ylabel('Average Return')
        plt.xticks(rotation=45)

        # Learning curves
        plt.subplot(1, 3, 2)
        for alg_name in alg_names:
            all_rewards = []
            for result in self.results[alg_name]:
                all_rewards.append(result['episode_rewards'])

            mean_rewards = np.mean(all_rewards, axis=0)
            std_rewards = np.std(all_rewards, axis=0)
            episodes = np.arange(len(mean_rewards))

            plt.plot(episodes, mean_rewards, label=alg_name)
            plt.fill_between(episodes, mean_rewards - std_rewards,
                             mean_rewards + std_rewards, alpha=0.3)

        plt.title('Learning Curves')
        plt.xlabel('Episode')
        plt.ylabel('Episode Reward')
        plt.legend()

        # Training time comparison
        plt.subplot(1, 3, 3)
        times = []
```

```
        time_errors = []

        for alg_name in alg_names:
            training_times = [r['training_time'] for r in self.results[
                alg_name]]
            times.append(np.mean(training_times))
            time_errors.append(np.std(training_times))

        plt.bar(alg_names, times, yerr=time_errors, capsize=5)
        plt.title('Training Time Comparison')
        plt.ylabel('Time (seconds)')
        plt.xticks(rotation=45)

        plt.tight_layout()
        plt.show()
```

Listing 6: Algorithm Comparison

Case Studies

This appendix presents detailed case studies that demonstrate the application of reinforcement learning to real-world engineering problems. Each case study includes problem formulation, algorithm selection and tuning, implementation details, and lessons learned.

.9 Case Study 1: Autonomous Drone Navigation

.9.1 Problem Description

A quadrotor drone must navigate through a complex environment with obstacles while minimizing energy consumption and flight time. The drone has continuous state and action spaces, making this a challenging continuous control problem.

State Space: $s = (x, y, z, \dot{x}, \dot{y}, \dot{z}, \phi, \theta, \psi, \dot{\phi}, \dot{\theta}, \dot{\psi}) \in \mathbb{R}^{12}$

where (x, y, z) is position, $(\dot{x}, \dot{y}, \dot{z})$ is velocity, (ϕ, θ, ψ) are Euler angles, and $(\dot{\phi}, \dot{\theta}, \dot{\psi})$ are angular velocities.

Action Space: $a = (T, \tau_\phi, \tau_\theta, \tau_\psi) \in \mathbb{R}^4$

where T is thrust and $(\tau_\phi, \tau_\theta, \tau_\psi)$ are torques about each axis.

Dynamics: The quadrotor dynamics are governed by:

$$m\ddot{\mathbf{r}} = T\mathbf{R}\mathbf{e}_3 - mg\mathbf{e}_3 \tag{26}$$

$$\mathbf{I}\dot{\boldsymbol{\omega}} = \boldsymbol{\tau} - \boldsymbol{\omega} \times \mathbf{I}\boldsymbol{\omega} \tag{27}$$

where $\mathbf{R}$ is the rotation matrix and $\mathbf{I}$ is the inertia tensor.

.9.2 Algorithm Selection and Implementation

Algorithm Choice: Deep Deterministic Policy Gradient (DDPG) was selected for its ability to handle continuous action spaces and its sample efficiency.

Network Architecture:

- Actor: $[12] \to [256] \to [256] \to [4]$ with tanh output activation
- Critic: $[16] \to [256] \to [256] \to [1]$ (state-action input)

- Target networks with soft updates ($\tau = 0.001$)

Reward Function:

$$r(s, a) = -\|s_{pos} - s_{target}\|^2 - 0.1\|a\|^2 - 10 \cdot \mathbf{1}_{collision} \tag{28}$$

.9.3 Training Process and Results

Training Configuration:

- Episodes: 2000
- Steps per episode: 1000
- Replay buffer size: 10^6
- Batch size: 256
- Learning rates: Actor 10^{-4}, Critic 10^{-3}
- Exploration noise: Ornstein-Uhlenbeck process

Performance Metrics:

- Success rate: 94% (reaching target within 1m)
- Average flight time: 12.3 seconds
- Energy efficiency: 15% improvement over PID controller
- Collision rate: 2%

.9.4 Lessons Learned

1. **Reward Shaping Critical:** Initial sparse rewards led to poor exploration. Dense reward with distance-based terms significantly improved learning.
2. **Curriculum Learning Effective:** Starting with simple environments and gradually increasing complexity improved sample efficiency.
3. **Simulation-to-Reality Gap:** Robust training with domain randomization was essential for real-world transfer.
4. **Safety Considerations:** Emergency safety controller was necessary during real-world testing.

.10 Case Study 2: Smart Grid Energy Management

.10.1 Problem Description

A microgrid with renewable energy sources, battery storage, and flexible loads must optimize energy dispatch to minimize costs while maintaining reliability constraints.

State Space:

- Battery state of charge: $SOC \in [0, 1]$
- Renewable generation forecast: $P_{ren} \in [0, P_{max}]$
- Load demand forecast: $P_{load} \in [0, P_{max}]$
- Electricity price: $\lambda \in [\lambda_{min}, \lambda_{max}]$
- Time of day: $t \in [0, 23]$

Action Space:

- Battery charge/discharge power: $P_{bat} \in [-P_{bat,max}, P_{bat,max}]$
- Grid import/export: $P_{grid} \in [-P_{grid,max}, P_{grid,max}]$
- Load curtailment: $P_{curt} \in [0, P_{load}]$

.10.2 MDP Formulation

Transition Dynamics:

$$SOC_{t+1} = SOC_t + \frac{\eta P_{bat,t} \Delta t}{E_{bat,max}} \tag{29}$$

$$P_{balance} = P_{ren} + P_{grid} + P_{bat} - P_{load} + P_{curt} \tag{30}$$

Constraints:

$$SOC_{min} \leq SOC_t \leq SOC_{max} \tag{31}$$

$$|P_{balance}| \leq \epsilon \quad \text{(power balance)} \tag{32}$$

Reward Function:

$$r_t = -(\lambda_t P_{grid,t} \Delta t + C_{curt} P_{curt,t} + C_{deg} |P_{bat,t}|) \tag{33}$$

.10.3 Implementation Details

Algorithm: Soft Actor-Critic (SAC) for its sample efficiency and robustness.

Feature Engineering:

- Moving averages of renewable generation and demand

- Seasonal and diurnal patterns encoded as sinusoidal features
- Price forecasts using historical patterns

Constraint Handling: Projection method to ensure feasible actions:

$$a_{feasible} = \Pi_{\mathcal{A}}(a_{proposed}) \tag{34}$$

.10.4 Results and Performance Analysis

Performance Comparison:

Method	Daily Cost ($)	Renewable Utilization (%)	Constraint Violations
Rule-based	142.50	78.3	0
MPC	138.20	82.1	0
SAC	134.80	85.7	0.02%

Key Insights:

1. 5.4% cost reduction compared to rule-based controller
2. 2.6% cost reduction compared to model predictive control
3. Learned to anticipate price patterns and pre-charge batteries
4. Robust performance under forecast uncertainties

.11 Case Study 3: Manufacturing Process Optimization

.11.1 Problem Description

A chemical batch process must optimize temperature and pressure profiles to maximize product yield while minimizing energy consumption and processing time.

Process Dynamics:

$$\frac{dC_A}{dt} = -k_1(T)C_A \tag{35}$$

$$\frac{dC_B}{dt} = k_1(T)C_A - k_2(T)C_B \tag{36}$$

$$\frac{dT}{dt} = \frac{Q - Q_{loss}(T)}{mC_p} \tag{37}$$

where $k_i(T) = A_i \exp(-E_i/RT)$ are temperature-dependent rate constants.

.11.2 Multi-Objective Optimization

Objectives:

$$J_1 = \text{maximize } C_B(t_f) \quad \text{(yield)} \tag{38}$$

$$J_2 = \text{minimize } \int_0^{t_f} Q(t)dt \quad \text{(energy)} \tag{39}$$

$$J_3 = \text{minimize } t_f \quad \text{(time)} \tag{40}$$

Scalarization: Weighted sum approach:

$$r(s,a) = w_1 \frac{C_B(t_f)}{C_{B,max}} - w_2 \frac{Q(t)}{Q_{max}} - w_3 \frac{1}{t_{max}} \tag{41}$$

.11.3 Implementation and Results

Algorithm: Twin Delayed DDPG (TD3) for stability in continuous control.

Results:

- 12% improvement in product yield
- 18% reduction in energy consumption
- 8% reduction in batch time
- Consistent performance across different initial conditions

.12 Performance Comparisons

.12.1 Algorithm Performance Summary

Case Study	Problem Type	Algorithm	Sample Efficiency	Final Performance
Drone Navigation	Continuous Control	DDPG	Medium	94% success
Smart Grid	Constrained Control	SAC	High	5.4% improvement
Manufacturing	Multi-objective	TD3	Medium	12% yield gain

.12.2 Common Success Factors

1. **Domain Knowledge Integration:** Incorporating engineering insights into reward design and feature engineering
2. **Simulation Fidelity:** High-fidelity simulators were crucial for initial learning
3. **Constraint Handling:** Explicit constraint enforcement prevented unsafe exploration

4. **Robust Training:** Domain randomization and robust optimization improved real-world performance

.13 Troubleshooting Guide

.13.1 Common Training Issues

Poor Convergence:

- Check reward function scaling and normalization
- Verify network initialization and learning rates
- Ensure sufficient exploration during early training
- Monitor gradient norms for vanishing/exploding gradients

Unstable Training:

- Reduce learning rates, especially for critic networks
- Use target networks with appropriate update rates
- Implement gradient clipping
- Check for numerical instabilities in environment dynamics

Poor Real-World Transfer:

- Increase simulation realism and randomization
- Implement domain adaptation techniques
- Use conservative policy updates
- Include safety constraints and emergency controllers

.13.2 Hyperparameter Tuning Guidelines

Learning Rates:

- Actor: typically 10^{-4} to 10^{-3}
- Critic: typically 10^{-3} to 10^{-2}
- Use learning rate schedules for long training runs

Network Architecture:

- Start with 2-3 hidden layers of 256-512 units
- Use batch normalization for deep networks
- Consider layer normalization for recurrent policies

Exploration:

- Gaussian noise: $\sigma = 0.1$ to 0.2 of action range
- Ornstein-Uhlenbeck: $\theta = 0.15$, $\sigma = 0.2$
- Decay exploration over training

.13.3 Debugging Checklist

1. **Environment Sanity Checks:**
   - Verify state and action space definitions
   - Test random policy performance
   - Check reward function computation
   - Validate episode termination conditions
2. **Algorithm Implementation:**
   - Verify gradient computation and backpropagation
   - Check replay buffer implementation
   - Validate target network updates
   - Monitor loss functions and metrics
3. **Training Diagnostics:**
   - Plot learning curves and moving averages
   - Monitor exploration statistics
   - Track gradient norms and weight distributions
   - Analyze action distributions over time