Reinforcement Learning

Lecture Notes

Master M2 — 2025–2026

Yaë Ulrich Gaba

"Can a machine think? This question is about as relevant as: can a submarine swim?"

— Edsger Dijkstra

March 25, 2026

Contents

Preface

Objectives of This Book

Deep Reinforcement Learning (DRL) combines classical reinforcement learning with deep learning. Over the past decade, it has enabled spectacular advances: defeating human champions at Go and Atari games, controlling robotic manipulators with remarkable dexterity, optimizing financial strategies, and designing new therapeutic molecules.

This book is intended for Master's and PhD students in computer science, applied mathematics, or engineering, as well as researchers and practitioners seeking a rigorous understanding of the theoretical foundations and algorithmic methods of DRL.

Course Organization

The course is structured in eleven chapters, organized in a pedagogical progression from mathematical foundations to recent applications:

1. **Markov Decision Processes (MDPs)**: the fundamental mathematical framework that formalizes agent-environment interaction.

2. **Dynamic Programming**: value iteration and policy iteration algorithms, the foundation of all RL methods.

3. **Monte Carlo and Temporal Difference Methods**: model-free approaches based on sampling.

4. **Q-Learning and SARSA**: fundamental tabular algorithms for off-policy and on-policy learning.

5. **Deep Q-Networks (DQN)**: the revolution of function approximation via neural networks applied to RL.

6. **Policy Gradient — REINFORCE**: direct optimization of the policy via gradient ascent.

7. **Actor-Critic Methods (A3C, PPO)**: combining value estimation and policy optimization.

8. **State of the Art (SAC, TD3, DDPG)**: modern algorithms for continuous action spaces.

9. **Multi-Agent Reinforcement Learning**: generalization to multi-agent systems.

10. **Constrained and Safe RL**: incorporating safety constraints into learning.

11. **Applications**: robotics, games, finance, and beyond.

Prerequisites

The reader should possess solid knowledge in:

- **Probability and Statistics**: random variables, conditional expectation, Markov chains, stochastic convergence.

- **Optimization**: gradient descent, convexity, Lagrange multipliers, numerical methods.

- **Linear Algebra**: vector spaces, matrix decompositions, eigenvalues.

- **Machine Learning**: neural networks, backpropagation, regularization, deep architectures (CNN, RNN).

- **Python Programming**: proficiency with NumPy, PyTorch, and Gymnasium environments (formerly OpenAI Gym).

Notation

We adopt the following conventions throughout this book:

Symbol	Meaning
$\mathcal{S}$	State space
$\mathcal{A}$	Action space
$\mathcal{R}$	Reward space
$\gamma \in [0, 1)$	Discount factor
$\pi(a \mid s)$	Stochastic policy
$V^\pi(s)$	State-value function under π
$Q^\pi(s, a)$	Action-value function under π
$A^\pi(s, a)$	Advantage function under π
$P(s' \mid s, a)$	Transition probability
$R(s, a, s')$	Reward function
$\mathbb{E}[\cdot]$	Expectation
$\mathbb{P}(\cdot)$	Probability
∇_θ	Gradient with respect to θ
θ	Policy or network parameters
α	Learning rate
ϵ	Exploration parameter (ϵ-greedy)
τ	Soft update coefficient
$\mathcal{D}$	Replay buffer

Typographic Conventions

- **Definitions** are presented in blue-bordered boxes.
- **Theorems** and **propositions** appear in formal boxes with proofs.
- **Algorithms** are presented in pseudocode in dedicated boxes.
- **Implementations** in PyTorch/Gymnasium are given in syntax-highlighted code blocks.
- **Key formulas** are highlighted in special boxes.
- Points requiring **special attention** are marked in warning boxes.

Acknowledgments

This book is the product of several years of teaching and research in reinforcement learning. We thank the students who, through their questions and curiosity, contributed to improving the presentation of these topics. We also express our gratitude to the open-source community, in particular the developers of PyTorch, Gymnasium, Stable-Baselines3, and CleanRL, whose tools make reinforcement learning accessible to all.

How to Use This Book

Intuition

Each chapter follows a consistent structure:

1. **Motivation and Intuition** — Why this method? What problem does it solve?
2. **Theoretical Foundations** — Definitions, theorems, convergence proofs.
3. **Algorithms** — Detailed pseudocode with complexity analysis.
4. **Implementation** — Working, reproducible PyTorch code.
5. **Exercises** — Theoretical and practical problems of increasing difficulty.

Brief History

Reinforcement learning has its roots in Bellman's work on dynamic programming (1950s) and behavioral psychology (Skinner's operant conditioning). Major milestones include:

- **1989**: Watkins introduces Q-Learning, the first convergent off-policy algorithm for MDPs.
- **1992**: Tesauro's TD-Gammon learns backgammon through self-play with temporal differences.
- **2013**: Mnih et al. (DeepMind) combine DQN with convolutional networks to play Atari games from pixels.
- **2015**: DQN published in *Nature*; Silver et al. develop AlphaGo.
- **2016**: AlphaGo defeats Lee Sedol, world Go champion.
- **2017**: Schulman et al. introduce PPO; Haarnoja et al. propose SAC.
- **2019**: OpenAI Five defeats world champions at Dota 2.
- **2020–2024**: RL applied to drug discovery, nuclear fusion control (DeepMind/EPFL), and LLM alignment (RLHF).

Supplementary Resources

To deepen the topics covered in this book, we recommend:

- **Sutton & Barto** — *Reinforcement Learning: An Introduction* (2nd edition, 2018). The standard reference in classical RL.
- **Bertsekas** — *Dynamic Programming and Optimal Control* (4th edition). Rigorous treatment of dynamic programming.

- **Szepesvári** — *Algorithms for Reinforcement Learning.* Concise synthesis of fundamental algorithms.
- **Agarwal et al.** — *Reinforcement Learning: Theory and Algorithms* (2022). Modern theoretical treatment.
- **Online courses**: David Silver (UCL), Sergey Levine (UC Berkeley), Emma Brunskill (Stanford).

[Author Name]
March 2026

Chapter 1

Markov Decision Processes

Imagine a robot learning to walk. At each moment, it observes its posture (the *state*), chooses a force to apply to its joints (the *action*), and receives a signal — did it stay upright or fall over? (the *reward*). Its goal: find a strategy of actions that maximizes rewards accumulated over time. This scheme — state, action, reward, new state — is universal: it describes a chess player, an autonomous vehicle, and a trading algorithm alike. The mathematical formalization of this scheme has a name: the *Markov decision process* (MDP), formalized by Richard Bellman in the 1950s.

Intuition

A Markov Decision Process (MDP) is the central mathematical model of reinforcement learning. It formalizes how an agent interacts with an environment by making sequential decisions to maximize long-term cumulative reward.

1.1 Introduction

Reinforcement learning rests on a simple idea: an agent learns to act by interacting with an environment. At each time step t, the agent observes a state $s_t \in \mathcal{S}$, chooses an action $a_t \in \mathcal{A}$, receives a reward $r_{t+1} \in \mathbb{R}$, and transitions to a new state s_{t+1}. The mathematical formalism that captures this interaction is the *Markov Decision Process*.

1.2 Formal Definition

Definition 1.1 (Markov Decision Process)**.** An MDP is a five-tuple $\mathcal{M} = (\mathcal{S}, \mathcal{A}, P, R, \gamma)$ where:

- $\mathcal{S}$ is a finite or countable set of **states**,
- $\mathcal{A}$ is a finite or countable set of **actions**,
- $P : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$ is the **transition function**: $P(s' \mid s, a) = \mathbb{P}(S_{t+1} = s' \mid S_t = s, A_t = a)$,
- $R : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ is the **reward function**: $R(s, a, s')$ is the reward received when transitioning from s to s' under action a,
- $\gamma \in [0, 1)$ is the **discount factor**.

Markov Property

The Markov property states that the future state depends only on the present state and current action:

$$\mathbb{P}(S_{t+1} = s' \mid S_0, A_0, S_1, A_1, \ldots, S_t, A_t) = \mathbb{P}(S_{t+1} = s' \mid S_t, A_t) = P(s' \mid S_t, A_t).$$

1.3 Policy

Definition 1.2 (Policy)**.** A **policy** π is a decision rule that prescribes the agent's behavior.

- **Deterministic policy**: $\pi : \mathcal{S} \to \mathcal{A}$, where $a = \pi(s)$.
- **Stochastic policy**: $\pi : \mathcal{S} \times \mathcal{A} \to [0, 1]$, where $\pi(a \mid s) = \mathbb{P}(A_t = a \mid S_t = s)$ with $\sum_{a \in \mathcal{A}} \pi(a \mid s) = 1$ for all s.

Definition 1.3 (Stationary Policy)**.** A policy π is **stationary** if it does not depend on time: $\pi_t = \pi$ for all $t \geq 0$. Unless stated otherwise, all policies considered in this course are stationary.

1.4 Value Functions

Definition 1.4 (Discounted Return)**.** The **discounted return** from time t is:

$$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}.$$

The factor $\gamma < 1$ ensures convergence of this sum when rewards are bounded.

Definition 1.5 (State-Value Function)**.** The **state-value function** under policy π is:

$$V^\pi(s) = \mathbb{E}_\pi \left[G_t \mid S_t = s \right] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \,\middle|\, S_t = s \right].$$

Definition 1.6 (Action-Value Function)**.** The **action-value function** (or Q-function) under policy π is:

$$Q^\pi(s, a) = \mathbb{E}_\pi \left[G_t \mid S_t = s, A_t = a \right].$$

Remark 1.7. The relationship between V^π and Q^π is:

$$V^\pi(s) = \sum_{a \in \mathcal{A}} \pi(a \mid s)\, Q^\pi(s, a).$$

1.5 Bellman Equations

Theorem 1.8 (Bellman Equation for V^π)**.** *For any policy π and any state $s \in \mathcal{S}$:*

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

Proof. By the definition of discounted return and the Markov property:

$$\begin{aligned} V^\pi(s) &= \mathbb{E}_\pi[G_t \mid S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} \mid S_t = s] \\ &= \sum_a \pi(a \mid s) \sum_{s'} P(s' \mid s, a) \left[R(s, a, s') + \gamma \, \mathbb{E}_\pi[G_{t+1} \mid S_{t+1} = s']\right] \\ &= \sum_a \pi(a \mid s) \sum_{s'} P(s' \mid s, a) \left[R(s, a, s') + \gamma \, V^\pi(s')\right]. \end{aligned}$$

□

Theorem 1.9 (Bellman Equation for Q^π)**.** *For any policy π, any state s, and any action a:*

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

Proof. Expanding similarly:

$$\begin{aligned} Q^\pi(s, a) &= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} \mid S_t = s, A_t = a] \\ &= \sum_{s'} P(s' \mid s, a) \left[R(s, a, s') + \gamma \, \mathbb{E}_\pi[G_{t+1} \mid S_{t+1} = s']\right] \\ &= \sum_{s'} P(s' \mid s, a) \left[R(s, a, s') + \gamma \sum_{a'} \pi(a' \mid s') Q^\pi(s', a')\right]. \end{aligned}$$

□

Bellman Equations — Summary

$$V^\pi(s) = \sum_a \pi(a|s) \sum_{s'} P(s'|s, a)[R(s, a, s') + \gamma V^\pi(s')] \tag{1.1}$$

$$Q^\pi(s, a) = \sum_{s'} P(s'|s, a)[R(s, a, s') + \gamma \sum_{a'} \pi(a'|s') Q^\pi(s', a')] \tag{1.2}$$

1.6 Optimal Policy and Bellman Optimality Equations

Definition 1.10 (Optimal Policy)**.** A policy π^* is **optimal** if for every state $s \in \mathcal{S}$:

$$V^{\pi^*}(s) \geq V^\pi(s) \quad \text{for all policies } \pi.$$

We write $V^*(s) = V^{\pi^*}(s)$ and $Q^*(s, a) = Q^{\pi^*}(s, a)$.

Theorem 1.11 (Bellman Optimality Equations)**.** *The optimal value functions satisfy:*

$$V^*(s) = \max_{a \in \mathcal{A}} \sum_{s'} P(s'|s, a) \left[R(s, a, s') + \gamma V^*(s')\right] \tag{1.3}$$

$$Q^*(s, a) = \sum_{s'} P(s'|s, a) \left[R(s, a, s') + \gamma \max_{a'} Q^*(s', a')\right] \tag{1.4}$$

Proof. For Equation (1.3), we use the fact that the optimal policy selects the action maximizing value:

$$V^*(s) = \max_\pi V^\pi(s) = \max_a Q^*(s,a) = \max_a \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma V^*(s')].$$

For Equation (1.4), we substitute $V^*(s') = \max_{a'} Q^*(s',a')$ into the Bellman equation for Q^*. □

Theorem 1.12 (Existence of a Deterministic Optimal Policy)**.** *For any finite MDP, there exists at least one deterministic optimal policy $\pi^* : \mathcal{S} \to \mathcal{A}$ defined by:*

$$\pi^*(s) = \arg\max_{a \in \mathcal{A}} Q^*(s,a).$$

1.7 Bellman Operator and Contraction

Definition 1.13 (Bellman Operator)**.** The **Bellman operator** $\mathcal{T}^\pi$ for a policy π is defined by:

$$(\mathcal{T}^\pi V)(s) = \sum_a \pi(a|s) \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma V(s')].$$

The **Bellman optimality operator** is:

$$(\mathcal{T}^* V)(s) = \max_a \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma V(s')].$$

Theorem 1.14 (Contraction of the Bellman Operator)**.** *The operators $\mathcal{T}^\pi$ and $\mathcal{T}^*$ are contractions in the infinity norm with factor γ:*

$$\left\|\mathcal{T}^\pi V - \mathcal{T}^\pi U\right\|_\infty \leq \gamma \left\|V - U\right\|_\infty.$$

By Banach's fixed-point theorem, each has a unique fixed point: V^π for $\mathcal{T}^\pi$ and V^* for $\mathcal{T}^*$.

Proof. For any $s \in \mathcal{S}$:

$$\begin{aligned}
|(\mathcal{T}^* V)(s) - (\mathcal{T}^* U)(s)| &= \left| \max_a \sum_{s'} P(s'|s,a)[R + \gamma V(s')] - \max_a \sum_{s'} P(s'|s,a)[R + \gamma U(s')] \right| \\
&\leq \max_a \sum_{s'} P(s'|s,a)\gamma \left|V(s') - U(s')\right| \\
&\leq \gamma \left\|V - U\right\|_\infty.
\end{aligned}$$

The transition from the second to third line uses $|\max_a f(a) - \max_a g(a)| \leq \max_a |f(a) - g(a)|$. □

1.8 Advantage Function

Definition 1.15 (Advantage Function)**.** The **advantage function** under policy π is:

$$A^\pi(s,a) = Q^\pi(s,a) - V^\pi(s).$$

It measures the relative advantage of taking action a in state s compared to the average behavior under π.

Proposition 1.16. For any policy π and any state s:

$$\sum_{a \in \mathcal{A}} \pi(a \mid s)\, A^{\pi}(s, a) = 0.$$

Proof. $\sum_a \pi(a|s) A^\pi(s,a) = \sum_a \pi(a|s)[Q^\pi(s,a) - V^\pi(s)] = V^\pi(s) - V^\pi(s) = 0.$ □

1.9 MDP Examples

Example 1.17 (Gridworld)**.** Consider a 4×4 grid where an agent moves in four cardinal directions. The terminal state is cell $(4, 4)$ with reward $+1$. Each step costs -0.01. Moves against walls leave the agent in place. This formalizes as an MDP with:

- $\mathcal{S} = \{(i, j) : 1 \le i, j \le 4\}$, $|\mathcal{S}| = 16$,
- $\mathcal{A} = \{\text{up}, \text{down}, \text{left}, \text{right}\}$,
- Deterministic transitions (except at boundaries),
- $\gamma = 0.99$.

Example 1.18 (K-Armed Bandit)**.** A K-armed bandit is a degenerate MDP with $|\mathcal{S}| = 1$ (single state). The agent selects an arm $a \in \{1, \ldots, K\}$ at each step and receives a random reward $R_a \sim \mathcal{D}_a$. This special case allows studying the exploration-exploitation dilemma in isolation.

1.10 Python Implementation

Defining a Simple MDP with Gymnasium

```python
import numpy as np
import gymnasium as gym
from gymnasium import spaces

class GridWorldEnv(gym.Env):
    """Custom 4x4 Gridworld environment."""
    metadata = {"render_modes": ["human"]}

    def __init__(self, size=4, gamma=0.99):
        super().__init__()
        self.size = size
        self.gamma = gamma
        self.observation_space = spaces.Discrete(size * size)
        self.action_space = spaces.Discrete(4)  # up, down, left, right
        self.goal = (size - 1, size - 1)
        self._action_to_dir = {
            0: np.array([-1, 0]),  # up
            1: np.array([1, 0]),   # down
            2: np.array([0, -1]), # left
            3: np.array([0, 1]),  # right
```

```
        }
        self.reset()

    def _pos_to_state(self, pos):
        return pos[0] * self.size + pos[1]

    def reset(self, seed=None, options=None):
        super().reset(seed=seed)
        self.agent_pos = np.array([0, 0])
        return self._pos_to_state(self.agent_pos), {}

    def step(self, action):
        direction = self._action_to_dir[action]
        new_pos = np.clip(self.agent_pos + direction, 0, self.size - 1)
        self.agent_pos = new_pos
        terminated = tuple(self.agent_pos) == self.goal
        reward = 1.0 if terminated else -0.01
        return self._pos_to_state(self.agent_pos), reward, terminated,
        ↪  False, {}
```

Building the Transition Matrix

```
def build_transition_matrix(env):
    """Build P[s, a, s'] and R[s, a, s'] for a finite MDP."""
    nS = env.observation_space.n
    nA = env.action_space.n
    P = np.zeros((nS, nA, nS))
    R = np.zeros((nS, nA, nS))

    for s in range(nS):
        for a in range(nA):
            env.agent_pos = np.array([s // env.size, s % env.size])
            s_next, reward, _, _, _ = env.step(a)
            P[s, a, s_next] = 1.0
            R[s, a, s_next] = reward
    return P, R

env = GridWorldEnv(size=4)
P, R = build_transition_matrix(env)
print(f"P shape: {P.shape}")  # (16, 4, 16)
print(f"R shape: {R.shape}")  # (16, 4, 16)
```

1.11 Direct Algebraic Solution

Proposition 1.19 (Matrix Solution of the Bellman Equation)**.** For a finite MDP with $|\mathcal{S}| = n$, the Bellman equation for V^π can be written in matrix form:

$$\mathbf{v}^\pi = \mathbf{r}^\pi + \gamma\, \mathbf{P}^\pi \mathbf{v}^\pi$$

where $\mathbf{P}^{\pi}_{s,s'} = \sum_a \pi(a|s)P(s'|s,a)$ and $\mathbf{r}^{\pi}_s = \sum_a \pi(a|s)\sum_{s'} P(s'|s,a)R(s,a,s')$. The solution is:

$$\mathbf{v}^{\pi} = (I - \gamma \mathbf{P}^{\pi})^{-1}\mathbf{r}^{\pi}.$$

Complexity of Direct Solution

Matrix inversion has complexity $O(n^3)$, making it impractical for MDPs with large state spaces. This motivates the iterative methods of Chapter 2.

1.12 Extensions of the MDP Model

Definition 1.20 (POMDP)**.** A **Partially Observable MDP** (POMDP) augments an MDP with an observation set $\mathcal{O}$ and an observation function $O : \mathcal{S} \times \mathcal{A} \to \Delta(\mathcal{O})$. The agent does not directly observe state s_t but instead receives observation $o_t \sim O(\cdot|s_t, a_{t-1})$.

Definition 1.21 (Finite-Horizon MDP)**.** A **finite-horizon** MDP with horizon H terminates after exactly H time steps. The return is:

$$G_t = \sum_{k=0}^{H-t-1} \gamma^k R_{t+k+1}.$$

Definition 1.22 (Continuous MDP)**.** When $\mathcal{S} \subseteq \mathbb{R}^n$ and/or $\mathcal{A} \subseteq \mathbb{R}^m$, we speak of continuous state or action space MDPs. Sums are replaced by integrals in the Bellman equations.

1.13 Exercises

Exercise 1.1 (Bellman Equation)**.** Consider an MDP with 3 states $\{s_1, s_2, s_3\}$, $\gamma = 0.9$, and the uniform policy $\pi(a|s) = 1/2$ for two actions. The transitions and rewards are:

- From s_1, action a_1: to s_2 with $R = 1$; action a_2: to s_3 with $R = 0$.
- From s_2, any action: to s_1 with $R = 2$.
- From s_3, any action: to s_3 with $R = 0$ (absorbing state).

Write the system of Bellman equations for V^{π} and solve it.

Exercise 1.2 (Contraction Operator)**.** Show that if $\gamma = 0$, the Bellman optimality operator converges in a single iteration. What is the interpretation of $\gamma = 0$?

Exercise 1.3 (Implementation)**.** Implement a custom `FrozenLake` environment with Gymnasium. Build the transition matrix and compute V^{π} by matrix inversion for the uniform policy. Verify your result by comparing with Gymnasium's `FrozenLake-v1`.

Exercise 1.4 (Advantage Function)**.** Show that for a deterministic optimal policy π^*:

$$A^{\pi^*}(s, \pi^*(s)) = 0 \quad \text{and} \quad A^{\pi^*}(s,a) \leq 0 \quad \forall a \neq \pi^*(s).$$

Chapter 2

Dynamic Programming

The name "dynamic programming" is one of the great misnomers in mathematics. Richard Bellman, who invented the framework in the 1950s at the RAND Corporation, later confessed that he chose the word "dynamic" to impress his Pentagon sponsors and "programming" because it sounded good—the actual content has nothing to do with computer programming in the modern sense. What Bellman discovered was something far deeper: a *principle of optimality* stating that an optimal policy has the property that, whatever the initial state and decision, the remaining decisions must constitute an optimal policy from the resulting state onward.

This recursive insight transforms an impossibly large optimisation problem—choosing the best action at every state for all time—into a sequence of local computations. When the environment is fully known (transition probabilities and rewards are given), dynamic programming solves Markov decision processes exactly, through two complementary algorithms: *policy iteration* and *value iteration*. These are not merely historical curiosities; they are the theoretical bedrock upon which all of reinforcement learning is built. Every modern RL algorithm, from Q-learning to policy gradient methods, can be understood as an approximation of the ideas in this chapter.

Intuition

Dynamic programming (DP) solves MDPs when the transition model $P(s'|s,a)$ and reward function $R(s,a,s')$ are perfectly known. It constitutes the theoretical foundation upon which all reinforcement learning methods rest.

2.1 Principle of Dynamic Programming

Dynamic programming, introduced by Richard Bellman in the 1950s, exploits the recursive structure of the Bellman equations to iteratively compute value functions. The two fundamental algorithms are:

1. **Policy evaluation**: computing V^π for a given policy π.

2. **Policy improvement**: constructing a better policy from V^π.

2.2 Policy Evaluation

Definition 2.1 (Iterative Policy Evaluation)**.** Iterative policy evaluation computes V^π by repeatedly applying the Bellman operator $\mathcal{T}^\pi$:

$$V_{k+1}(s) = \sum_a \pi(a|s) \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma V_k(s')] \quad \forall s \in \mathcal{S}.$$

Theorem 2.2 (Convergence of Policy Evaluation)**.** *For any initialization V_0, the sequence $(V_k)_{k\geq 0}$ defined by $V_{k+1} = \mathcal{T}^\pi V_k$ converges to V^π as $k \to \infty$. Moreover:*

$$\|V_k - V^\pi\|_\infty \leq \gamma^k \|V_0 - V^\pi\|_\infty .$$

Proof. This is a direct consequence of Theorem 1.14. The operator $\mathcal{T}^\pi$ is a γ-contraction in the infinity norm, so by Banach's fixed-point theorem, the sequence converges geometrically to the unique fixed point V^π. □

Iterative Policy Evaluation

1. Initialize $V(s) \leftarrow 0$ for all $s \in \mathcal{S}$
2. **Repeat** until convergence ($\Delta < \theta$):
   (a) $\Delta \leftarrow 0$
   (b) For each $s \in \mathcal{S}$:
      i. $v \leftarrow V(s)$
      ii. $V(s) \leftarrow \sum_a \pi(a|s) \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma V(s')]$
      iii. $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
3. Return $V \approx V^\pi$

2.3 Policy Improvement

Theorem 2.3 (Policy Improvement Theorem)**.** *Let π be a policy and π' the greedy policy with respect to V^π:*

$$\pi'(s) = \arg\max_{a \in \mathcal{A}} \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma V^\pi(s')].$$

Then $V^{\pi'}(s) \geq V^\pi(s)$ for all $s \in \mathcal{S}$, with equality if and only if π is already optimal.

Proof. For any state s:

$$\begin{aligned} V^\pi(s) &\leq \max_a Q^\pi(s,a) = Q^\pi(s, \pi'(s)) \\ &= \sum_{s'} P(s'|s,\pi'(s))[R(s,\pi'(s),s') + \gamma V^\pi(s')] \\ &\leq \sum_{s'} P(s'|s,\pi'(s))[R(s,\pi'(s),s') + \gamma Q^\pi(s',\pi'(s'))] \\ &\leq \cdots \leq V^{\pi'}(s). \end{aligned}$$

The result follows by recursive application of the inequality. □

2.4 Policy Iteration

Policy Iteration

1. Initialize $\pi(s)$ arbitrarily for all s
2. **Repeat**:
   (a) **Evaluation**: compute V^π via iterative policy evaluation
   (b) **Improvement**: for each s:
   $$\pi'(s) \leftarrow \arg\max_a \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma V^\pi(s')]$$
   (c) If $\pi' = \pi$, **stop** (optimal policy found)
   (d) $\pi \leftarrow \pi'$
3. Return $\pi^* = \pi$

Theorem 2.4 (Convergence of Policy Iteration)**.** *Policy iteration converges to the optimal policy π^* in a finite number of iterations (at most $|\mathcal{A}|^{|\mathcal{S}|}$ iterations, since the number of deterministic policies is finite and each iteration strictly improves the policy).*

2.5 Value Iteration

Definition 2.5 (Value Iteration)**.** Value iteration combines evaluation and improvement in a single update:

$$V_{k+1}(s) = \max_{a\in\mathcal{A}} \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma V_k(s')] \quad \forall s \in \mathcal{S}.$$

Theorem 2.6 (Convergence of Value Iteration)**.** *The sequence (V_k) defined by value iteration converges to V^*:*

$$\|V_k - V^*\|_\infty \leq \gamma^k \|V_0 - V^*\|_\infty .$$

The number of iterations to reach precision ϵ is:

$$k \geq \frac{\log(\|V_0 - V^*\|_\infty / \epsilon)}{\log(1/\gamma)}.$$

Value Iteration

1. Initialize $V(s) \leftarrow 0$ for all $s \in \mathcal{S}$
2. **Repeat** until convergence ($\Delta < \theta$):
   (a) $\Delta \leftarrow 0$
   (b) For each $s \in \mathcal{S}$:
      i. $v \leftarrow V(s)$

ii. $V(s) \leftarrow \max_a \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma V(s')]$
iii. $\Delta \leftarrow \max(\Delta, |v - V(s)|)$

3. Extract policy: $\pi^*(s) = \arg\max_a \sum_{s'} P(s'|s,a)[R + \gamma V(s')]$
4. Return V^*, π^*

2.6 Comparison of Methods

	Policy Iteration	**Value Iteration**
Cost per iteration	$O(\|\mathcal{S}\|^3 + \|\mathcal{S}\|^2\|\mathcal{A}\|)$	$O(\|\mathcal{S}\|^2\|\mathcal{A}\|)$
Number of iterations	Few (often < 10)	More ($O(1/\log(1/\gamma))$)
Memory	$V + \pi$	V only
Convergence	Finite, exact	Asymptotic

Remark 2.7. In practice, policy iteration often converges in very few iterations (5–10), even for large MDPs. Value iteration requires more iterations but each is less expensive since it avoids the full policy evaluation step.

2.7 Generalized Policy Iteration (GPI)

Definition 2.8 (GPI — Generalized Policy Iteration)**. Generalized Policy Iteration** refers to any alternation between (partial or complete) policy evaluation and policy improvement. Nearly all RL algorithms can be viewed as instances of GPI.

Remark 2.9. Value iteration is a limiting case of GPI where evaluation performs only a single Bellman backup before improvement.

2.8 Python Implementation

Policy Evaluation

```
import numpy as np

def policy_evaluation(P, R, pi, gamma=0.99, theta=1e-8):
    """
    Iterative policy evaluation.
    P: transition matrix (nS, nA, nS)
    R: reward matrix (nS, nA, nS)
    pi: deterministic policy (nS,) -> action indices
    """
    nS = P.shape[0]
    V = np.zeros(nS)

    while True:
```

```
        delta = 0.0
        for s in range(nS):
            a = pi[s]
            v = V[s]
            V[s] = np.sum(P[s, a, :] * (R[s, a, :] + gamma * V))
            delta = max(delta, abs(v - V[s]))
        if delta < theta:
            break
    return V
```

Policy Iteration

```
def policy_iteration(P, R, gamma=0.99, theta=1e-8):
    """Complete policy iteration."""
    nS, nA = P.shape[0], P.shape[1]
    pi = np.zeros(nS, dtype=int)

    while True:
        V = policy_evaluation(P, R, pi, gamma, theta)
        stable = True
        for s in range(nS):
            old_action = pi[s]
            Q_s = np.array([
                np.sum(P[s, a, :] * (R[s, a, :] + gamma * V))
                for a in range(nA)
            ])
            pi[s] = np.argmax(Q_s)
            if old_action != pi[s]:
                stable = False
        if stable:
            break
    return V, pi
```

Value Iteration

```
def value_iteration(P, R, gamma=0.99, theta=1e-8):
    """Value iteration."""
    nS, nA = P.shape[0], P.shape[1]
    V = np.zeros(nS)

    while True:
        delta = 0.0
        for s in range(nS):
            v = V[s]
            Q_s = np.array([
                np.sum(P[s, a, :] * (R[s, a, :] + gamma * V))
                for a in range(nA)
            ])
            V[s] = np.max(Q_s)
```

```
            delta = max(delta, abs(v - V[s]))
        if delta < theta:
            break

    pi = np.zeros(nS, dtype=int)
    for s in range(nS):
        Q_s = np.array([
            np.sum(P[s, a, :] * (R[s, a, :] + gamma * V))
            for a in range(nA)
        ])
        pi[s] = np.argmax(Q_s)
    return V, pi
```

Application to FrozenLake

```
import gymnasium as gym

env = gym.make("FrozenLake-v1", is_slippery=True)
nS = env.observation_space.n  # 16
nA = env.action_space.n       # 4
P_mat = np.zeros((nS, nA, nS))
R_mat = np.zeros((nS, nA, nS))

for s in range(nS):
    for a in range(nA):
        for prob, next_s, reward, done in env.unwrapped.P[s][a]:
            P_mat[s, a, next_s] += prob
            R_mat[s, a, next_s] = reward

V_star, pi_star = value_iteration(P_mat, R_mat, gamma=0.99)
print("Optimal policy (4x4 grid):")
actions = ['<', 'v', '>', '^']
for i in range(4):
    print([actions[pi_star[4*i + j]] for j in range(4)])
```

2.9 Exercises

Exercise 2.1 (Value Iteration Convergence)**.** Show that for the 4×4 gridworld from Chapter 1 with $\gamma = 0.9$ and $V_0 = 0$, value iteration achieves precision $\epsilon = 10^{-6}$ in at most k iterations. Compute k.

Exercise 2.2 (Modified Policy Iteration)**.** Implement a version of policy iteration where evaluation performs exactly m sweeps instead of converging. Study the effect of m on overall convergence speed for $m \in \{1, 3, 10, 100\}$.

Exercise 2.3 (Asynchronous Sweeps)**.** Implement an asynchronous version of value iteration where states are updated in random order (one state per iteration). Compare convergence with the standard synchronous version.

Exercise 2.4 (Matrix Solution)**.** For a 3-state MDP with the uniform policy, compute V^π by matrix inversion $(I - \gamma P^\pi)^{-1} r^\pi$ and verify that the result matches iterative evaluation.

Chapter 3

Monte Carlo and TD Methods

Dynamic programming assumes perfect knowledge of the environment: transition probabilities, rewards, everything is given. But in the real world, a robot learning to walk does not know the equations of physics—it only has its *experience*, the trajectories it has actually followed. How can one learn from experience alone?

Two fundamental answers emerge. *Monte Carlo* methods, heirs to the work of Stanislaw Ulam and John von Neumann at the Los Alamos laboratory in the 1940s, estimate values by averaging the returns observed over complete episodes. *Temporal difference* (TD) methods, introduced by Richard Sutton in 1988, do something bolder: they update estimates at every time step, using their own predictions as targets—a mechanism called *bootstrapping*. This intellectually dizzying idea (learning from what you do not yet know) is the keystone of modern reinforcement learning.

Intuition

Monte Carlo and Temporal Difference (TD) methods learn value functions directly from experience without knowing the transition model. Monte Carlo waits until the end of an episode to update; TD updates at every time step using a bootstrap estimate.

3.1 Model-Free Learning

Unlike dynamic programming, which requires complete knowledge of the model (P, R), model-free methods learn from episodes generated by direct interaction with the environment. An episode is a sequence:

$$S_0, A_0, R_1, S_1, A_1, R_2, \ldots, S_{T-1}, A_{T-1}, R_T, S_T$$

where S_T is a terminal state.

3.2 Monte Carlo Methods

3.2.1 MC Value Estimation

Definition 3.1 (First-Visit MC Evaluation)**.** The **first-visit** MC estimate of $V^\pi(s)$ is the average of returns G_t observed at the first occurrence of s in each episode:

$$V^\pi(s) \approx \frac{1}{N(s)} \sum_{i=1}^{N(s)} G_t^{(i)}$$

where $N(s)$ is the number of episodes in which s was visited.

Definition 3.2 (Every-Visit MC Evaluation)**.** The **every-visit** MC estimate uses all occurrences of s in all episodes, not just the first.

Theorem 3.3 (First-Visit MC Convergence)**.** *The first-visit MC estimate converges to $V^\pi(s)$ as $N(s) \to \infty$, by the strong law of large numbers, since the returns $G_t^{(i)}$ are i.i.d. random variables with expectation $V^\pi(s)$.*

First-Visit MC Evaluation

1. Initialize $V(s) \leftarrow 0$, $N(s) \leftarrow 0$ for all s
2. For each episode:
   (a) Generate episode following π: $S_0, A_0, R_1, \ldots, S_T$
   (b) $G \leftarrow 0$
   (c) For $t = T-1, T-2, \ldots, 0$:
      i. $G \leftarrow \gamma G + R_{t+1}$
      ii. If S_t does not appear in $S_0, \ldots, S_{t-1}$:
         A. $N(S_t) \leftarrow N(S_t) + 1$
         B. $V(S_t) \leftarrow V(S_t) + \frac{1}{N(S_t)}(G - V(S_t))$

3.2.2 MC Estimation of the Q-Function

For model-free policy improvement, it is necessary to estimate $Q^\pi(s,a)$ rather than $V^\pi(s)$:

$$Q^\pi(s,a) \approx \frac{1}{N(s,a)} \sum_{i=1}^{N(s,a)} G_t^{(i)}.$$

Exploration Problem

If π is deterministic, some state-action pairs (s,a) will never be visited. Solutions: use **exploring starts** or an ϵ-greedy policy.

3.2.3 MC with ϵ-Greedy Policy

Definition 3.4 (ϵ-Greedy Policy)**.** The ϵ-greedy policy derived from Q is:

$$\pi_\epsilon(a \mid s) = \begin{cases} 1 - \epsilon + \frac{\epsilon}{|\mathcal{A}|} & \text{if } a = \arg\max_{a'} Q(s, a') \\ \frac{\epsilon}{|\mathcal{A}|} & \text{otherwise} \end{cases}$$

Theorem 3.5 (ϵ-Greedy Improvement)**.** *The ϵ-greedy policy π' with respect to Q^{π_ϵ} satisfies $V^{\pi'}(s) \geq V^{\pi_\epsilon}(s)$ for all s.*

3.3 Off-Policy MC with Importance Sampling

Definition 3.6 (Importance Sampling Ratio)**.** Let b be the behavior policy and π the target policy. The **importance sampling ratio** for a trajectory from t to $T-1$ is:

$$\rho_{t:T-1} = \prod_{k=t}^{T-1} \frac{\pi(A_k \mid S_k)}{b(A_k \mid S_k)}.$$

Off-Policy MC Estimator

The off-policy estimate of V^π is:

$$V^\pi(s) \approx \frac{\sum_i \rho_{t_i:T_i-1} G_{t_i}^{(i)}}{\sum_i \rho_{t_i:T_i-1}} \quad \text{(weighted importance sampling)}.$$

Proposition 3.7 (Bias and Variance)**.** The ordinary estimator $\hat{V} = \frac{1}{n}\sum_i \rho_i G_i$ is unbiased but has potentially infinite variance. The weighted estimator $\hat{V}_w = \frac{\sum_i \rho_i G_i}{\sum_i \rho_i}$ is biased but has bounded variance. The bias vanishes as $n \to \infty$.

3.4 Temporal Differences — TD(0)

Definition 3.8 (TD(0) Update)**.** The TD(0) method updates V at every time step using the **TD target**:

$$V(S_t) \leftarrow V(S_t) + \alpha\big[\underbrace{R_{t+1} + \gamma V(S_{t+1})}_{\text{TD target}} - V(S_t)\big].$$

The term $\delta_t = R_{t+1} + \gamma V(S_{t+1}) - V(S_t)$ is called the **TD error**.

Temporal Difference Error

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

The TD error is an unbiased estimator of the advantage $A^\pi(S_t, A_t)$ in expectation.

Theorem 3.9 (Convergence of TD(0))**.** *Under the Robbins-Monro conditions on the learning rate:*

$$\sum_{t=0}^{\infty} \alpha_t = \infty, \qquad \sum_{t=0}^{\infty} \alpha_t^2 < \infty,$$

the TD(0) algorithm converges almost surely to V^π.

TD(0) for Policy Evaluation

1. Initialize $V(s)$ arbitrarily for all s, fix α
2. For each episode:
   (a) Initialize S_0
   (b) For each step $t = 0, 1, 2, \ldots$ until termination:
      i. $A_t \sim \pi(\cdot|S_t)$
      ii. Observe R_{t+1}, S_{t+1}
      iii. $V(S_t) \leftarrow V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)]$

3.5 Comparison: MC vs TD

	Monte Carlo	**TD(0)**
Requires complete episodes	Yes	No
Bootstrapping	No	Yes
Bias	None	Biased (diminishes)
Variance	High	Low
Convergence	$1/\sqrt{N}$	Faster in practice
Sensitivity to α	Low	High

Remark 3.10. The **bias-variance tradeoff** is central: MC has zero bias but high variance (since the return G_t integrates all future rewards). TD has bias due to bootstrapping ($V(S_{t+1})$ is an estimate) but lower variance.

3.6 TD(λ) and Eligibility Traces

Definition 3.11 (λ-Return)**.** The **λ-return** is an exponentially weighted average of n-step returns:

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

where $G_t^{(n)} = \sum_{k=0}^{n-1} \gamma^k R_{t+k+1} + \gamma^n V(S_{t+n})$ is the n-step return.

Definition 3.12 (Eligibility Trace)**.** The **eligibility trace** for state s at time t is:

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

with $e_0(s) = \mathbb{1}_{S_0=s}$.

TD(λ) with Eligibility Traces

1. Initialize $V(s)$ for all s
2. For each episode:
   (a) $e(s) \leftarrow 0$ for all s
   (b) For each step t:
      i. Observe R_{t+1}, S_{t+1}
      ii. $\delta_t \leftarrow R_{t+1} + \gamma V(S_{t+1}) - V(S_t)$
      iii. $e(S_t) \leftarrow e(S_t) + 1$
      iv. For all s: $V(s) \leftarrow V(s) + \alpha \delta_t e(s)$
      v. For all s: $e(s) \leftarrow \gamma \lambda\, e(s)$

Remark 3.13. $\lambda = 0$ yields TD(0); $\lambda = 1$ yields (essentially) MC. Intermediate values $\lambda \in (0, 1)$ offer an optimal bias-variance tradeoff in practice.

3.7 Python Implementation

First-Visit Monte Carlo Evaluation

```python
import numpy as np
import gymnasium as gym

def mc_first_visit(env, pi, n_episodes=10000, gamma=0.99):
    """First-visit MC evaluation of V^pi."""
    nS = env.observation_space.n
    V = np.zeros(nS)
    N = np.zeros(nS)

    for _ in range(n_episodes):
        episode = []
        s, _ = env.reset()
        done = False
        while not done:
            a = pi(s)
            s_next, r, terminated, truncated, _ = env.step(a)
            episode.append((s, a, r))
            s = s_next
            done = terminated or truncated

        G = 0.0
        visited = set()
        for s, a, r in reversed(episode):
            G = gamma * G + r
            if s not in visited:
                visited.add(s)
                N[s] += 1
```

```
                V[s] += (G - V[s]) / N[s]
    return V
```

TD(0) for Policy Evaluation

```
def td0_evaluation(env, pi, n_episodes=10000, alpha=0.01, gamma=0.99):
    """TD(0) evaluation of V^pi."""
    nS = env.observation_space.n
    V = np.zeros(nS)

    for _ in range(n_episodes):
        s, _ = env.reset()
        done = False
        while not done:
            a = pi(s)
            s_next, r, terminated, truncated, _ = env.step(a)
            done = terminated or truncated
            V_next = 0.0 if terminated else V[s_next]
            V[s] += alpha * (r + gamma * V_next - V[s])
            s = s_next
    return V
```

TD(λ) with Eligibility Traces

```
def td_lambda(env, pi, n_episodes=10000, alpha=0.01,
              gamma=0.99, lam=0.8):
    """TD(lambda) with eligibility traces."""
    nS = env.observation_space.n
    V = np.zeros(nS)

    for _ in range(n_episodes):
        e = np.zeros(nS)
        s, _ = env.reset()
        done = False
        while not done:
            a = pi(s)
            s_next, r, terminated, truncated, _ = env.step(a)
            done = terminated or truncated
            V_next = 0.0 if terminated else V[s_next]
            delta = r + gamma * V_next - V[s]
            e[s] += 1.0  # accumulating trace
            V += alpha * delta * e
            e *= gamma * lam
            s = s_next
    return V
```

3.8 Exercises

Exercise 3.1 (MC vs TD Variance)**.** Generate 1000 episodes in a 5-state MDP with a fixed policy. Estimate V^π by first-visit MC and TD(0). Plot the error curves $\|V_n - V^\pi\|_2$ as a function of episode count n.

Exercise 3.2 (n-Step Return)**.** Implement the n-step return $G_t^{(n)}$ and compare performance for $n \in \{1, 2, 4, 8, \infty\}$ on the `FrozenLake-v1` environment.

Exercise 3.3 (TD Convergence Proof)**.** Prove that the TD error δ_t satisfies $\mathbb{E}_\pi[\delta_t | S_t = s] = 0$ when $V = V^\pi$. What does this imply about the fixed point of TD(0)?

Exercise 3.4 (Importance Sampling)**.** Implement off-policy MC estimation with weighted importance sampling to estimate V^π from trajectories generated by a uniform policy b. Compare with the ordinary estimator.

Chapter 4

Q-Learning and Temporal Difference Methods

In 1989, Christopher Watkins, in his doctoral thesis at Cambridge, proposed a deceptively simple algorithm: at each transition, update the value of a state-action pair using the maximum over the next actions. This is *Q-learning*, the first reinforcement learning algorithm proven to converge to the optimal policy *without* knowing the environment model. Almost simultaneously, Rummery and Niranjan proposed *SARSA*, an on-policy variant. The distinction between these two approaches—*off-policy* and *on-policy*—is one of the fundamental dividing lines in reinforcement learning.

Intuition

Q-Learning and SARSA are the foundational model-free control algorithms. They learn the action-value function $Q(s, a)$ directly from experience. SARSA is *on-policy*: it evaluates the policy it follows. Q-Learning is *off-policy*: it learns the optimal policy regardless of the exploration strategy. This chapter covers their derivation, convergence guarantees, and exploration mechanisms.

4.1 TD Control Framework

The **control** problem is to find an optimal policy π^* without access to the transition model. The idea is to combine TD estimation of Q^π with ϵ-greedy policy improvement, in the spirit of Generalized Policy Iteration (GPI).

Definition 4.1 (TD Target for Action-Values)**.** The one-step TD target for $Q(S_t, A_t)$ using a bootstrap estimate is:

$$y_t = R_{t+1} + \gamma Q(S_{t+1}, A_{t+1})$$

where A_{t+1} is the action taken in the next state. The TD error is $\delta_t = y_t - Q(S_t, A_t)$.

4.2 SARSA — On-Policy Control

Definition 4.2 (SARSA)**.** The SARSA algorithm updates $Q(S_t, A_t)$ using the quintuplet $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\big[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)\big].$$

SARSA Update

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\big[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)\big]$$

where $A_{t+1} \sim \pi_\epsilon(\cdot|S_{t+1})$ is the action actually chosen by the ϵ-greedy policy.

SARSA

1. Initialize $Q(s, a)$ arbitrarily, $Q(\text{terminal}, \cdot) = 0$
2. For each episode:
   (a) Initialize S_0; choose A_0 from π_ϵ derived from Q
   (b) For each step $t = 0, 1, \ldots$ until termination:
      i. Execute A_t, observe R_{t+1}, S_{t+1}
      ii. Choose A_{t+1} from π_ϵ derived from Q
      iii. $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)]$

Theorem 4.3 (SARSA Convergence)**.** *Under the Robbins-Monro conditions and if every state-action pair (s, a) is visited infinitely often (GLIE condition: Greedy in the Limit with Infinite Exploration), SARSA converges to Q^* almost surely.*

4.3 Q-Learning — Off-Policy Control

Definition 4.4 (Q-Learning)**.** The Q-Learning algorithm (Watkins, 1989) updates:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\big[R_{t+1} + \gamma \max_{a'} Q(S_{t+1}, a') - Q(S_t, A_t)\big].$$

Q-Learning Update

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\Big[R_{t+1} + \gamma \max_{a' \in \mathcal{A}} Q(S_{t+1}, a') - Q(S_t, A_t)\Big]$$

The key difference with SARSA is the use of max instead of $Q(S_{t+1}, A_{t+1})$. Q-Learning learns Q^* directly, regardless of the behavior policy (as long as all pairs (s, a) are visited).

Q-Learning

1. Initialize $Q(s, a)$ arbitrarily, $Q(\text{terminal}, \cdot) = 0$
2. For each episode:
   (a) Initialize S_0
   (b) For each step $t = 0, 1, \ldots$ until termination:
      i. Choose A_t from π_ϵ derived from Q
      ii. Execute A_t, observe R_{t+1}, S_{t+1}
      iii. $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_{a'} Q(S_{t+1}, a') - Q(S_t, A_t)]$

Theorem 4.5 (Q-Learning Convergence)**.** *If every pair (s, a) is visited infinitely often and the learning rates $\alpha_t(s, a)$ satisfy the Robbins-Monro conditions:*

$$\sum_t \alpha_t(s,a) = \infty, \qquad \sum_t \alpha_t^2(s,a) < \infty,$$

then $Q(s, a) \to Q^*(s, a)$ almost surely for all (s, a).

Proof sketch. The update is a stochastic approximation of the Bellman optimality operator $\mathcal{T}^*$. The contraction property of $\mathcal{T}^*$ in the ℓ_∞ norm, combined with the Robbins-Monro conditions, guarantees convergence (Jaakkola et al., 1994). □

4.4 SARSA vs Q-Learning: The Cliff Walking Example

Example 4.6 (Cliff Walking)**.** The `CliffWalking-v0` environment illustrates the difference:

- **Q-Learning** learns the optimal policy (along the cliff edge), but the ϵ-greedy exploration may cause the agent to fall.
- **SARSA** learns a safer policy (away from the cliff) because it accounts for the ϵ-greedy behavior in its updates.

Q-Learning optimizes the return of the *target* (greedy) policy, while SARSA optimizes the return of the *actual* (ϵ-greedy) policy.

4.5 Expected SARSA

Definition 4.7 (Expected SARSA)**. Expected SARSA** uses the expectation over next actions instead of a sampled action:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\Big[R_{t+1} + \gamma \sum_{a'} \pi(a'|S_{t+1}) Q(S_{t+1}, a') - Q(S_t, A_t)\Big].$$

Remark 4.8. Expected SARSA generalizes both SARSA and Q-Learning:

- If π is the current ϵ-greedy policy, it is an improved version of SARSA with lower variance.
- If π is the greedy policy, it reduces to Q-Learning.

4.6 n-Step Methods

Definition 4.9 (n-Step Return)**.** The n-step return combines n immediate rewards with a bootstrap estimate:

$$G_t^{(n)} = \sum_{k=0}^{n-1} \gamma^k R_{t+k+1} + \gamma^n Q(S_{t+n}, A_{t+n}).$$

Proposition 4.10 (Bias-Variance of n-Step Returns).

- $n = 1$: TD(0) — low variance, high bias.
- $n = \infty$: Monte Carlo — zero bias, high variance.
- Intermediate n: trade-off between bias and variance.

In practice, $n \in \{4, 8, 16\}$ often works well.

4.7 Exploration Strategies

Definition 4.11 (ϵ-Greedy Exploration). The ϵ-greedy policy derived from Q is:

$$\pi_\epsilon(a \mid s) = \begin{cases} 1 - \epsilon + \frac{\epsilon}{|\mathcal{A}|} & \text{if } a = \arg\max_{a'} Q(s, a') \\ \frac{\epsilon}{|\mathcal{A}|} & \text{otherwise} \end{cases}$$

Typically ϵ is decayed over time: $\epsilon_t = \max(\epsilon_{\min}, \epsilon_0 \cdot \beta^t)$.

Definition 4.12 (Boltzmann (Softmax) Exploration). Boltzmann exploration selects actions with probability proportional to exponentiated Q-values:

$$\pi_\tau(a \mid s) = \frac{\exp(Q(s, a)/\tau)}{\sum_{a'} \exp(Q(s, a')/\tau)}$$

where $\tau > 0$ is the temperature. As $\tau \to 0$, this becomes greedy; as $\tau \to \infty$, it becomes uniform.

Definition 4.13 (Upper Confidence Bound (UCB)). UCB-based exploration selects:

$$A_t = \arg\max_a \left[Q(s, a) + c\sqrt{\frac{\ln N(s)}{N(s, a)}} \right]$$

where $N(s)$ and $N(s, a)$ are visit counts and $c > 0$ controls the exploration bonus.

Exploration-Exploitation Dilemma

Insufficient exploration leads to sub-optimal convergence (the agent may miss high-reward regions). Excessive exploration wastes samples on uninformative actions. The optimal balance depends on the environment structure and the learning horizon.

4.8 Maximization Bias and Double Q-Learning

Definition 4.14 (Maximization Bias). Q-Learning uses $\max_{a'} Q(S_{t+1}, a')$ as the target. Because Q is a noisy estimate, the max introduces a positive bias: $\mathbb{E}[\max_a Q(s, a)] \geq \max_a \mathbb{E}[Q(s, a)]$.

Definition 4.15 (Double Q-Learning). Double Q-Learning (Hasselt, 2010) maintains two independent estimators Q_1, Q_2. At each step, one is randomly chosen for the update:

$$Q_1(S_t, A_t) \leftarrow Q_1(S_t, A_t) + \alpha \Big[R_{t+1} + \gamma Q_2\big(S_{t+1}, \arg\max_{a'} Q_1(S_{t+1}, a')\big) - Q_1(S_t, A_t) \Big].$$

The roles of Q_1 and Q_2 are swapped with probability 0.5.

Proposition 4.16 (Double Q-Learning removes maximization bias)**.** By decoupling action selection ($\arg\max$ on Q_1) from evaluation (using Q_2), Double Q-Learning produces unbiased estimates in the limit, eliminating the systematic overestimation of standard Q-Learning.

4.9 Python Implementation

Q-Learning Implementation

```python
import numpy as np
import gymnasium as gym

def q_learning(env, n_episodes=10000, alpha=0.1,
               gamma=0.99, epsilon=1.0, eps_decay=0.999,
               eps_min=0.01):
    """Tabular Q-learning with epsilon-greedy exploration."""
    nS = env.observation_space.n
    nA = env.action_space.n
    Q = np.zeros((nS, nA))

    for ep in range(n_episodes):
        s, _ = env.reset()
        done = False
        while not done:
            # epsilon-greedy action selection
            if np.random.random() < epsilon:
                a = env.action_space.sample()
            else:
                a = np.argmax(Q[s])
            s_next, r, terminated, truncated, _ = env.step(a)
            done = terminated or truncated
            Q_next = 0.0 if terminated else np.max(Q[s_next])
            Q[s, a] += alpha * (r + gamma * Q_next - Q[s, a])
            s = s_next
        epsilon = max(eps_min, epsilon * eps_decay)
    return Q
```

SARSA Implementation

```python
def sarsa(env, n_episodes=10000, alpha=0.1,
          gamma=0.99, epsilon=0.1):
    """Tabular SARSA with epsilon-greedy policy."""
    nS = env.observation_space.n
    nA = env.action_space.n
    Q = np.zeros((nS, nA))

    def eps_greedy(s):
        if np.random.random() < epsilon:
            return env.action_space.sample()
```

```
        return np.argmax(Q[s])

    for ep in range(n_episodes):
        s, _ = env.reset()
        a = eps_greedy(s)
        done = False
        while not done:
            s_next, r, terminated, truncated, _ = env.step(a)
            done = terminated or truncated
            a_next = eps_greedy(s_next) if not done else 0
            Q_next = 0.0 if terminated else Q[s_next, a_next]
            Q[s, a] += alpha * (r + gamma * Q_next - Q[s, a])
            s, a = s_next, a_next
    return Q
```

4.10 Exercises

Exercise 4.1 (Cliff Walking Comparison)**.** Implement Q-Learning and SARSA on `CliffWalking-v0`. Plot the per-episode reward for both algorithms. Explain the difference in learned policies.

Exercise 4.2 (Exploration Strategies)**.** Compare ϵ-greedy, Boltzmann, and UCB exploration on `FrozenLake-v1`. Measure convergence speed (episodes to reach > 0.7 average reward) for each strategy.

Exercise 4.3 (Double Q-Learning)**.** Implement Double Q-Learning and compare with standard Q-Learning on an MDP where maximization bias is harmful (e.g., the "Maximization Bias Example" from Sutton & Barto, Chapter 6). Plot the percentage of times the correct action is chosen at the start state.

Exercise 4.4 (Expected SARSA Derivation)**.** Show that Expected SARSA with a greedy target policy is equivalent to Q-Learning. What happens when $\epsilon = 0$ in the ϵ-greedy case?

Chapter 5

Deep Q-Networks and Variants

In February 2015, a team from DeepMind published in *Nature* a result that made headlines: an algorithm, the *Deep Q-Network* (DQN), learned to play 49 Atari games directly from screen pixels, achieving superhuman performance in several of them. The key: combining the tabular Q-learning of the previous chapter with a deep neural network that approximates the Q function. Two technical innovations—*experience replay* (storing transitions and resampling them randomly) and the *target network* (updating targets more slowly)—stabilised a learning process that, without them, diverges. DQN opened the era of *deep reinforcement learning*.

Intuition

The Deep Q-Network (DQN) by Mnih et al. (2013, 2015) revolutionized RL by combining Q-Learning with deep neural networks. Two key innovations — experience replay and a target network — stabilize training, enabling agents to learn directly from raw pixels and achieve human-level performance on Atari games.

5.1 From Tabular Q to Function Approximation

Tabular Q-Learning cannot handle high-dimensional or continuous state spaces. The idea is to approximate $Q^*(s, a)$ with a neural network parameterized by θ:

$$Q(s, a; \theta) \approx Q^*(s, a).$$

Naive Instability

Directly applying Q-Learning with a neural network diverges due to three issues (the **deadly triad**):

1. **Temporal correlation**: consecutive samples $(s_t, a_t, r_{t+1}, s_{t+1})$ are strongly correlated.
2. **Non-stationary target**: the target $r + \gamma \max_{a'} Q(s', a'; \theta)$ depends on θ, which changes at every update.
3. **Function approximation**: Q-Learning convergence is no longer guaranteed with nonlinear approximators.

5.2 DQN Architecture

Definition 5.1 (Deep Q-Network)**.** The **DQN** uses two stabilization mechanisms:

1. **Experience replay**: transitions (s, a, r, s') are stored in a replay buffer $\mathcal{D}$ of capacity N and uniformly sampled in mini-batches for updates.
2. **Target network**: a separate network $Q(s, a; \theta^-)$ with frozen parameters θ^- computes the target. Parameters θ^- are copied from θ every C steps.

DQN Loss Function

The loss is the mean squared error over a mini-batch sampled from $\mathcal{D}$:

$$\mathcal{L}(\theta) = \mathbb{E}_{(s,a,r,s')\sim\mathcal{D}}\left[\left(r + \gamma \max_{a'} Q(s', a'; \theta^-) - Q(s, a; \theta)\right)^2\right].$$

DQN Algorithm

1. Initialize replay buffer $\mathcal{D}$ with capacity N
2. Initialize $Q(s, a; \theta)$ with random weights θ
3. Set target parameters $\theta^- \leftarrow \theta$
4. For each episode:
   (a) Initialize state s_0 (preprocess frame stack)
   (b) For each step t:
      i. Select $a_t = \arg\max_a Q(s_t, a; \theta)$ with probability $1 - \epsilon$, else random action
      ii. Execute a_t, observe r_{t+1}, s_{t+1}
      iii. Store $(s_t, a_t, r_{t+1}, s_{t+1})$ in $\mathcal{D}$
      iv. Sample mini-batch from $\mathcal{D}$
      v. Compute targets: $y_j = r_j + \gamma \max_{a'} Q(s'_j, a'; \theta^-)$
      vi. Update θ by gradient descent on $\frac{1}{B}\sum_j (y_j - Q(s_j, a_j; \theta))^2$
      vii. Every C steps: $\theta^- \leftarrow \theta$

Theorem 5.2 (Experience Replay Decorrelation)**.** *Uniform sampling from a sufficiently large replay buffer produces approximately i.i.d. samples, satisfying the stochastic gradient descent assumption. Each transition can be reused multiple times, improving sample efficiency.*

5.3 DQN for Atari

Example 5.3 (Atari 2600 Results)**.** The DQN architecture for Atari processes a stack of 4 grayscale 84×84 frames through three convolutional layers followed by two fully connected layers. Key results from Mnih et al. (2015):

- Superhuman performance on 29 out of 49 Atari games.

- Single architecture and hyperparameters across all games.
- Learned from raw pixels with reward clipping to $[-1, +1]$.

5.4 Double DQN

Definition 5.4 (Double DQN)**.** Double DQN (van Hasselt et al., 2016) addresses the maximization bias by decoupling action selection from evaluation:

$$y_t = R_{t+1} + \gamma Q\Big(S_{t+1}, \arg\max_{a'} Q(S_{t+1}, a'; \theta)\,; \theta^-\Big).$$

The online network θ selects the best action; the target network θ^- evaluates it.

Proposition 5.5 (Overestimation Reduction)**.** Standard DQN overestimates Q-values because $\mathbb{E}[\max_a Q] \geq \max_a \mathbb{E}[Q]$. Double DQN reduces this bias significantly, leading to more stable training and improved final performance on most Atari games.

5.5 Dueling DQN

Definition 5.6 (Dueling Architecture)**.** The dueling architecture (Wang et al., 2016) decomposes the Q-function:

$$Q(s, a; \theta, \alpha, \beta) = V(s; \theta, \beta) + \left(A(s, a; \theta, \alpha) - \frac{1}{|\mathcal{A}|} \sum_{a'} A(s, a'; \theta, \alpha) \right).$$

Intuition

The value stream $V(s)$ captures state quality independently of action choice. The advantage stream $A(s, a)$ captures the *relative* benefit of each action. This decomposition accelerates learning in states where action choice has little impact.

5.6 Prioritized Experience Replay

Definition 5.7 (Prioritized Replay)**.** Prioritized experience replay (Schaul et al., 2016) samples transitions with probability proportional to their TD error magnitude:

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha}, \qquad p_i = |\delta_i| + \epsilon_p$$

where δ_i is the last observed TD error, $\alpha \in [0, 1]$ controls prioritization, and $\epsilon_p > 0$ ensures nonzero probability.

Definition 5.8 (Importance Sampling Correction)**.** To compensate for the non-uniform sampling bias, importance sampling weights are applied:

$$w_i = \left(\frac{1}{N \cdot P(i)} \right)^\beta$$

where β is annealed from β_0 to 1 during training. The weighted loss becomes $\mathcal{L} = \frac{1}{B} \sum_j w_j (y_j - Q(s_j, a_j; \theta))^2$.

Remark 5.9. Prioritized replay significantly improves sample efficiency by focusing updates on "surprising" transitions. The sum-tree data structure enables efficient $O(\log N)$ priority-based sampling.

5.7 Rainbow DQN

Definition 5.10 (Rainbow)**.** Rainbow (Hessel et al., 2018) combines six DQN extensions:

1. Double Q-Learning
2. Prioritized experience replay
3. Dueling architecture
4. Multi-step returns (n-step TD targets)
5. Distributional RL (C51)
6. Noisy networks (learned exploration)

n-Step DQN Target

$$y_t^{(n)} = \sum_{k=0}^{n-1} \gamma^k R_{t+k+1} + \gamma^n \max_{a'} Q(S_{t+n}, a'; \theta^-)$$

Proposition 5.11 (Rainbow Ablation)**.** Ablation studies show that prioritized replay and n-step returns provide the largest individual contributions. Removing distributional RL or dueling networks also hurts performance, but less dramatically.

5.8 Soft Target Updates

Definition 5.12 (Polyak Averaging)**.** Instead of periodically copying $\theta \to \theta^-$, soft updates blend the parameters at every step:

$$\theta^- \leftarrow \tau\theta + (1-\tau)\theta^-$$

with $\tau \ll 1$ (typically $\tau = 0.005$). This provides smoother target evolution and is widely used in actor-critic methods.

5.9 Python Implementation

Minimal DQN with PyTorch

```
import torch
import torch.nn as nn
import numpy as np
from collections import deque
import random
```

```python
class QNetwork(nn.Module):
    def __init__(self, state_dim, action_dim, hidden=128):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(state_dim, hidden), nn.ReLU(),
            nn.Linear(hidden, hidden), nn.ReLU(),
            nn.Linear(hidden, action_dim)
        )

    def forward(self, x):
        return self.net(x)

class ReplayBuffer:
    def __init__(self, capacity=100000):
        self.buffer = deque(maxlen=capacity)

    def push(self, s, a, r, s_next, done):
        self.buffer.append((s, a, r, s_next, done))

    def sample(self, batch_size):
        batch = random.sample(self.buffer, batch_size)
        s, a, r, s2, d = zip(*batch)
        return (np.array(s), np.array(a), np.array(r),
                np.array(s2), np.array(d, dtype=np.float32))

def dqn_train(env, n_episodes=500, gamma=0.99, lr=1e-3,
              batch_size=64, target_update=100):
    sd = env.observation_space.shape[0]
    ad = env.action_space.n
    q_net = QNetwork(sd, ad)
    q_target = QNetwork(sd, ad)
    q_target.load_state_dict(q_net.state_dict())
    optimizer = torch.optim.Adam(q_net.parameters(), lr=lr)
    buf = ReplayBuffer()
    step = 0
    for ep in range(n_episodes):
        s, _ = env.reset()
        done = False
        while not done:
            eps = max(0.01, 1.0 - step / 5000)
            if random.random() < eps:
                a = env.action_space.sample()
            else:
                with torch.no_grad():
                    a = q_net(torch.FloatTensor(s)).argmax().item()
            s2, r, term, trunc, _ = env.step(a)
            buf.push(s, a, r, s2, term)
            s = s2
            done = term or trunc
            step += 1
```

```
            if len(buf.buffer) >= batch_size:
                sb, ab, rb, s2b, db = buf.sample(batch_size)
                q_vals = q_net(torch.FloatTensor(sb)).gather(
                    1, torch.LongTensor(ab).unsqueeze(1)).squeeze()
                with torch.no_grad():
                    q_next = q_target(
                        torch.FloatTensor(s2b)).max(1)[0]
                    targets = (torch.FloatTensor(rb)
                        + gamma * q_next
                        * (1 - torch.FloatTensor(db)))
                loss = nn.MSELoss()(q_vals, targets)
                optimizer.zero_grad()
                loss.backward()
                optimizer.step()
            if step % target_update == 0:
                q_target.load_state_dict(q_net.state_dict())
    return q_net
```

5.10 Exercises

Exercise 5.1 (DQN on CartPole)**.** Implement DQN for `CartPole-v1`. Train with and without a target network. Plot the learning curves and explain the effect of the target network on stability.

Exercise 5.2 (Double DQN)**.** Modify the DQN implementation to use Double DQN. Compare Q-value estimates with standard DQN on `LunarLander-v2` and verify that Double DQN reduces overestimation.

Exercise 5.3 (Dueling Architecture)**.** Implement the dueling architecture. Train on `CartPole-v1` and compare with the standard architecture. In which states does the advantage stream have the largest variance?

Exercise 5.4 (Prioritized Replay)**.** Implement prioritized experience replay with importance sampling correction. Compare sample efficiency with uniform replay on `LunarLander-v2`.

Chapter 6

Policy Gradient Methods

Until now, we have learned *value functions* and derived the policy indirectly. Policy gradient methods invert this logic: they directly parameterise the policy π_θ and optimise its parameters by gradient ascent. The REINFORCE algorithm, proposed by Ronald Williams in 1992, uses the *policy gradient theorem*—an elegant result that expresses the gradient of expected performance as an expectation under the policy, without needing to differentiate through the environment dynamics. This approach opens the door to continuous action spaces and expressive stochastic policies.

Intuition

Policy gradient methods optimize the parameters θ of a policy π_θ by ascending the gradient of the expected return $J(\theta) = \mathbb{E}_{\pi_\theta}[G_0]$. Unlike value-based methods, they naturally handle continuous action spaces and stochastic policies, and avoid the instabilities of deriving a policy from a Q-function.

6.1 Motivation

DQN-style methods have fundamental limitations:

- They require **discrete** and small action spaces (the max over actions is explicit).
- The policy is implicit (greedy w.r.t. Q) and can only represent deterministic policies.
- Small changes in Q can cause large, discontinuous changes in the policy, destabilizing training.

The policy gradient idea is to parameterize π_θ directly and optimize θ by gradient ascent.

6.2 Performance Objective

Definition 6.1 (Performance Objective)**.** The performance objective is the expected return under π_θ:

$$J(\theta) = \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T-1}\gamma^t R_{t+1}\right] = \mathbb{E}_{s_0\sim\rho_0}\left[V^{\pi_\theta}(s_0)\right]$$

where ρ_0 is the initial state distribution and $\tau = (s_0, a_0, r_1, s_1, \ldots)$ is a trajectory.

6.3 The Policy Gradient Theorem

Theorem 6.2 (Policy Gradient Theorem)**.** *The gradient of $J(\theta)$ is:*

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(A_t \mid S_t)\, Q^{\pi_\theta}(S_t, A_t)\right].$$

Proof. Starting from $J(\theta) = \sum_s d^{\pi_\theta}(s) \sum_a \pi_\theta(a|s) Q^{\pi_\theta}(s,a)$ where $d^{\pi_\theta}(s)$ is the discounted state visitation frequency, we compute:

$$\begin{aligned}
\nabla_\theta J(\theta) &= \sum_s d^{\pi_\theta}(s) \sum_a \nabla_\theta \pi_\theta(a|s) Q^{\pi_\theta}(s,a) \\
&= \sum_s d^{\pi_\theta}(s) \sum_a \pi_\theta(a|s) \nabla_\theta \log \pi_\theta(a|s)\, Q^{\pi_\theta}(s,a) \\
&= \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(A_t|S_t)\, Q^{\pi_\theta}(S_t, A_t)\right].
\end{aligned}$$

The key identity used is $\nabla_\theta \pi_\theta = \pi_\theta \nabla_\theta \log \pi_\theta$ (the log-derivative trick). □

Policy Gradient Estimator

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \sum_{t=0}^{T_i - 1} \nabla_\theta \log \pi_\theta(a_t^{(i)} \mid s_t^{(i)})\, \hat{Q}_t^{(i)}$$

where $\hat{Q}_t^{(i)}$ is an estimate of $Q^{\pi_\theta}(s_t, a_t)$ from the i-th trajectory.

6.4 REINFORCE Algorithm

Definition 6.3 (REINFORCE)**.** The REINFORCE algorithm (Williams, 1992) uses the Monte Carlo return $G_t = \sum_{k=0}^{T-t-1} \gamma^k R_{t+k+1}$ as an unbiased estimate of $Q^{\pi_\theta}(S_t, A_t)$:

$$\theta \leftarrow \theta + \alpha \sum_{t=0}^{T-1} \gamma^t \nabla_\theta \log \pi_\theta(A_t \mid S_t)\, G_t.$$

REINFORCE

1. Initialize policy parameters θ randomly
2. For each episode:
   (a) Generate trajectory $\tau = (s_0, a_0, r_1, \ldots, s_T)$ following π_θ
   (b) For $t = T-1, T-2, \ldots, 0$: compute $G_t = \sum_{k=0}^{T-t-1} \gamma^k r_{t+k+1}$
   (c) $\theta \leftarrow \theta + \alpha \sum_{t=0}^{T-1} \gamma^t \nabla_\theta \log \pi_\theta(a_t|s_t)\, G_t$

High Variance of REINFORCE

REINFORCE uses full Monte Carlo returns, which have high variance. This leads to slow convergence and noisy gradient estimates. Variance reduction techniques are essential in practice.

6.5 Baselines for Variance Reduction

Theorem 6.4 (Baseline Invariance)**.** *For any function $b(s)$ that does not depend on a:*

$$\mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(a|s)\, b(s)\right] = 0.$$

Therefore, subtracting $b(s)$ from $Q^{\pi_\theta}(s,a)$ does not change the gradient expectation but can reduce variance.

Proof. $\mathbb{E}_{a\sim\pi_\theta(\cdot|s)}[\nabla_\theta \log \pi_\theta(a|s)\, b(s)] = b(s)\sum_a \nabla_\theta \pi_\theta(a|s) = b(s)\nabla_\theta \underbrace{\sum_a \pi_\theta(a|s)}_{=1} = 0.$ □

Definition 6.5 (REINFORCE with Baseline)**.** Using $b(s) = V^{\pi_\theta}(s)$ as baseline, the update becomes:

$$\theta \leftarrow \theta + \alpha \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(A_t|S_t)\, \underbrace{(G_t - V_\phi(S_t))}_{\hat{A}_t}$$

where $\hat{A}_t$ approximates the advantage $A^{\pi_\theta}(S_t, A_t)$ and V_ϕ is a learned value function.

Proposition 6.6 (Optimal Baseline)**.** The variance-minimizing baseline is:

$$b^*(s) = \frac{\mathbb{E}\left[\|\nabla_\theta \log \pi_\theta(a|s)\|^2 Q^\pi(s,a)\right]}{\mathbb{E}\left[\|\nabla_\theta \log \pi_\theta(a|s)\|^2\right]}.$$

In practice, $V^\pi(s)$ is a good approximation and much simpler.

6.6 Generalized Advantage Estimation (GAE)

Definition 6.7 (Generalized Advantage Estimation)**.** GAE (Schulman et al., 2016) is an exponentially weighted average of n-step advantage estimates:

$$\hat{A}_t^{\mathrm{GAE}(\gamma,\lambda)} = \sum_{l=0}^{\infty} (\gamma\lambda)^l \delta_{t+l}$$

where $\delta_t = R_{t+1} + \gamma V_\phi(S_{t+1}) - V_\phi(S_t)$ is the TD error.

Proposition 6.8 (Limiting Cases of GAE)**.**

- $\lambda = 0$: $\hat{A}_t = \delta_t$ (one-step TD error; low variance, high bias).
- $\lambda = 1$: $\hat{A}_t = G_t - V_\phi(S_t)$ (MC return minus baseline; high variance, no bias).

In practice, $\lambda \in [0.9, 0.99]$ achieves a good trade-off.

GAE Recursive Computation

$$\hat{A}_t^{\mathrm{GAE}} = \delta_t + \gamma\lambda\, \hat{A}_{t+1}^{\mathrm{GAE}}$$

with $\hat{A}_T^{\mathrm{GAE}} = 0$. This enables efficient backward computation in $O(T)$ time.

6.7 Natural Policy Gradient

Definition 6.9 (Fisher Information Matrix)**.** The Fisher information matrix of π_θ is:

$$F(\theta) = \mathbb{E}_{s\sim d^{\pi_\theta}, a\sim\pi_\theta}\left[\nabla_\theta \log \pi_\theta(a|s)\, \nabla_\theta \log \pi_\theta(a|s)^\top\right].$$

Definition 6.10 (Natural Policy Gradient)**.** The natural policy gradient (Kakade, 2001) update uses the inverse Fisher matrix to account for the geometry of the policy space:

$$\theta \leftarrow \theta + \alpha F(\theta)^{-1} \nabla_\theta J(\theta).$$

This is equivalent to steepest ascent in the KL-divergence metric.

Theorem 6.11 (Equivalence to KL-Constrained Update)**.** *The natural gradient step is the solution to:*

$$\max_{\Delta\theta} \nabla_\theta J(\theta)^\top \Delta\theta \quad s.t. \quad \frac{1}{2}\Delta\theta^\top F(\theta)\Delta\theta \le \epsilon.$$

This ensures that each update does not change the policy too much in a distribution-theoretic sense.

6.8 Trust Region Concepts

Definition 6.12 (Trust Region Policy Optimization)**.** TRPO (Schulman et al., 2015) maximizes a surrogate objective subject to a KL-divergence constraint:

$$\max_\theta \mathbb{E}_{s,a\sim\pi_{\theta_{\text{old}}}}\left[\frac{\pi_\theta(a|s)}{\pi_{\theta_{\text{old}}}(a|s)}\hat{A}_t\right] \quad \text{s.t.} \quad \mathbb{E}_s\left[D_{\text{KL}}(\pi_{\theta_{\text{old}}}(\cdot|s)\|\pi_\theta(\cdot|s))\right] \le \delta.$$

Remark 6.13. TRPO guarantees monotonic policy improvement under certain conditions. However, it requires computing second-order information (Fisher-vector products via conjugate gradient), making it computationally expensive. PPO (covered in the next chapter) provides a simpler first-order approximation.

6.9 Python Implementation

REINFORCE with Baseline

```python
import torch
import torch.nn as nn
import numpy as np

class PolicyNet(nn.Module):
    def __init__(self, state_dim, action_dim, hidden=64):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(state_dim, hidden), nn.ReLU(),
            nn.Linear(hidden, action_dim), nn.Softmax(dim=-1)
        )

    def forward(self, x):
```

```
        return self.net(x)

class ValueNet(nn.Module):
    def __init__(self, state_dim, hidden=64):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(state_dim, hidden), nn.ReLU(),
            nn.Linear(hidden, 1)
        )

    def forward(self, x):
        return self.net(x).squeeze(-1)

def reinforce_baseline(env, n_episodes=1000, gamma=0.99,
                       lr_pi=1e-3, lr_v=1e-3):
    sd = env.observation_space.shape[0]
    ad = env.action_space.n
    policy = PolicyNet(sd, ad)
    value = ValueNet(sd)
    opt_pi = torch.optim.Adam(policy.parameters(), lr=lr_pi)
    opt_v = torch.optim.Adam(value.parameters(), lr=lr_v)

    for ep in range(n_episodes):
        states, actions, rewards = [], [], []
        s, _ = env.reset()
        done = False
        while not done:
            probs = policy(torch.FloatTensor(s))
            dist = torch.distributions.Categorical(probs)
            a = dist.sample()
            s2, r, term, trunc, _ = env.step(a.item())
            states.append(s); actions.append(a); rewards.append(r)
            s = s2; done = term or trunc

        # Compute returns
        G, returns = 0.0, []
        for r in reversed(rewards):
            G = r + gamma * G
            returns.insert(0, G)
        returns = torch.FloatTensor(returns)
        states_t = torch.FloatTensor(np.array(states))

        # Update value function
        V = value(states_t)
        value_loss = nn.MSELoss()(V, returns)
        opt_v.zero_grad(); value_loss.backward(); opt_v.step()

        # Update policy with advantage
        with torch.no_grad():
            advantages = returns - value(states_t)
        log_probs = torch.stack([
```

```
            torch.distributions.Categorical(
                policy(torch.FloatTensor(s))).log_prob(a)
            for s, a in zip(states, actions)])
        policy_loss = -(log_probs * advantages).mean()
        opt_pi.zero_grad(); policy_loss.backward(); opt_pi.step()
    return policy
```

6.10 Exercises

Exercise 6.1 (REINFORCE Variance)**.** Run REINFORCE on `CartPole-v1` with and without a baseline. Plot the variance of the gradient estimate (computed over 10 runs) as a function of episodes. Quantify the variance reduction.

Exercise 6.2 (GAE Implementation)**.** Implement GAE with different values of $\lambda \in \{0, 0.5, 0.9, 1.0\}$. Compare learning curves on `LunarLander-v2`.

Exercise 6.3 (Continuous Actions)**.** Implement REINFORCE for a continuous action space using a Gaussian policy $\pi_\theta(a|s) = \mathcal{N}(\mu_\theta(s), \sigma^2)$. Test on `MountainCarContinuous-v0`.

Exercise 6.4 (Policy Gradient Theorem Proof)**.** Extend the proof of the policy gradient theorem to the infinite-horizon discounted setting. Show that the state visitation distribution $d^{\pi_\theta}(s) = (1-\gamma)\sum_{t=0}^{\infty} \gamma^t P(S_t = s|\pi_\theta)$ makes the theorem hold.

Chapter 7

Actor-Critic Methods

Policy gradient methods (REINFORCE) suffer from high variance; value-based methods (DQN) struggle with continuous action spaces. In the early 2000s, Richard Sutton, David McAllester, Satinder Singh, and Yishay Mansour formalized an idea that reconciles these two families: the *policy gradient theorem* shows that the gradient can be estimated using a separately learned value function. Thus was born the *actor-critic* architecture: an **actor** that proposes actions and a **critic** that evaluates their quality. The critic provides a baseline that drastically reduces gradient estimate variance, while the actor retains the ability to represent stochastic policies over continuous spaces. From A2C to PPO to SAC, actor-critic methods dominate contemporary deep reinforcement learning.

Intuition

Actor-critic methods combine the best of both worlds: an **actor** (the policy π_θ) and a **critic** (the value function V_ϕ or Q_ϕ). The critic provides a low-variance signal to guide the actor's updates, eliminating the need to wait for complete episodes. Modern variants (PPO, SAC, TD3) form the backbone of contemporary deep RL.

7.1 Actor-Critic Framework

Definition 7.1 (Actor-Critic)**.** An **actor-critic** algorithm maintains two networks:

- **Actor**: $\pi_\theta(a|s)$, parameterized by θ.
- **Critic**: $V_\phi(s)$ or $Q_\phi(s,a)$, parameterized by ϕ.

The critic is updated via TD methods and the actor via policy gradient using the critic as a baseline.

Basic Actor-Critic Update

Critic:

$$\phi \leftarrow \phi - \alpha_\phi \nabla_\phi \left(R_{t+1} + \gamma V_\phi(S_{t+1}) - V_\phi(S_t)\right)^2$$

Actor:

$$\theta \leftarrow \theta + \alpha_\theta \nabla_\theta \log \pi_\theta(A_t|S_t) \underbrace{\left(R_{t+1} + \gamma V_\phi(S_{t+1}) - V_\phi(S_t)\right)}_{\hat{A}_t \text{ (estimated advantage)}}$$

7.2 Advantage Actor-Critic (A2C)

Definition 7.2 (A2C)**.** Advantage Actor-Critic (A2C) is the synchronous version of A3C (Mnih et al., 2016). Multiple workers collect trajectories in parallel. The gradients are averaged and applied synchronously:

$$\nabla_\theta J \approx \frac{1}{K} \sum_{k=1}^{K} \sum_{t} \nabla_\theta \log \pi_\theta(a_t^{(k)} | s_t^{(k)}) \, \hat{A}_t^{(k)}$$

where K is the number of parallel workers.

Definition 7.3 (A3C)**.** Asynchronous Advantage Actor-Critic (A3C) uses multiple workers updating a shared parameter server *asynchronously*. Each worker computes gradients on its own trajectory and applies them to the global parameters without synchronization.

Remark 7.4. A2C typically matches or outperforms A3C in practice because synchronous updates produce less noisy gradients. A3C's main advantage was computational (avoiding GPU memory bottlenecks), which is less relevant with modern hardware.

7.3 Proximal Policy Optimization (PPO)

Definition 7.5 (PPO-Clip)**.** Proximal Policy Optimization (Schulman et al., 2017) uses a clipped surrogate objective to prevent excessively large policy updates:

$$L^{\text{CLIP}}(\theta) = \mathbb{E}_t \left[\min \left(r_t(\theta) \hat{A}_t, \; \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon) \hat{A}_t \right) \right]$$

where $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$ is the probability ratio and $\epsilon \approx 0.2$.

PPO Full Objective

$$L(\theta) = L^{\text{CLIP}}(\theta) - c_1 \, L^{\text{VF}}(\phi) + c_2 \, H[\pi_\theta]$$

where $L^{\text{VF}} = (V_\phi(s_t) - G_t)^2$ is the value loss, $H[\pi_\theta]$ is an entropy bonus for exploration, and c_1, c_2 are coefficients.

PPO Algorithm

1. For each iteration:
   (a) Collect T timesteps of data using $\pi_{\theta_{\text{old}}}$ with K parallel actors
   (b) Compute advantages $\hat{A}_t$ using GAE
   (c) For E epochs over the collected data:
      i. Sample mini-batch from the collected data
      ii. Compute $r_t(\theta)$ and the clipped objective
      iii. Update θ by gradient ascent on $L(\theta)$
   (d) $\theta_{\text{old}} \leftarrow \theta$

Intuition

The clipping mechanism in PPO acts as a "trust region" in probability ratio space. When the advantage is positive, the objective is clipped if $r_t > 1 + \epsilon$ (preventing too large a ratio). When the advantage is negative, it is clipped if $r_t < 1 - \epsilon$. This ensures the new policy stays close to the old one without requiring second-order optimization.

7.4 Continuous Action Spaces: DDPG

Definition 7.6 (DDPG)**.** Deep Deterministic Policy Gradient (Lillicrap et al., 2016) extends DQN to continuous actions using a deterministic policy $\mu_\theta(s)$:

- **Critic**: $Q_\phi(s, a)$ trained with Bellman error: $y = r + \gamma Q_{\phi^-}(s', \mu_{\theta^-}(s'))$
- **Actor**: θ updated by the deterministic policy gradient: $\nabla_\theta J = \mathbb{E}_s[\nabla_a Q_\phi(s, a)|_{a=\mu_\theta(s)} \nabla_\theta \mu_\theta(s)]$

Both use target networks with Polyak averaging: $\phi^- \leftarrow \tau\phi + (1 - \tau)\phi^-$.

Theorem 7.7 (Deterministic Policy Gradient)**.** *For a deterministic policy μ_θ, the policy gradient simplifies to:*

$$\nabla_\theta J(\theta) = \mathbb{E}_{s\sim d^{\mu_\theta}} \left[\nabla_\theta \mu_\theta(s) \nabla_a Q^{\mu_\theta}(s, a)\big|_{a=\mu_\theta(s)} \right].$$

This avoids integration over the action space, making it efficient for high-dimensional continuous actions.

7.5 Twin Delayed DDPG (TD3)

Definition 7.8 (TD3)**.** TD3 (Fujimoto et al., 2018) addresses DDPG's overestimation with three techniques:

1. **Twin critics**: two independent Q-networks Q_{ϕ_1}, Q_{ϕ_2}; the target uses the minimum: $y = r + \gamma \min_{i=1,2} Q_{\phi_i^-}(s', \tilde{a}')$
2. **Delayed updates**: the actor and target networks are updated every d steps (typically $d = 2$), while the critics update every step.
3. **Target policy smoothing**: noise is added to the target action: $\tilde{a}' = \mu_{\theta^-}(s') + \text{clip}(\epsilon, -c, c)$, $\epsilon \sim \mathcal{N}(0, \sigma^2)$.

7.6 Soft Actor-Critic (SAC)

Definition 7.9 (Maximum Entropy RL)**.** The maximum entropy objective augments the standard return with an entropy bonus:

$$J(\theta) = \mathbb{E}_{\pi_\theta} \left[\sum_{t=0}^{\infty} \gamma^t \left(R_{t+1} + \alpha H[\pi_\theta(\cdot|S_t)] \right) \right]$$

where $\alpha > 0$ is the temperature parameter controlling the exploration-exploitation trade-off.

Definition 7.10 (Soft Actor-Critic)**.** SAC (Haarnoja et al., 2018) learns a stochastic policy by optimizing the maximum entropy objective:

- **Soft Q-function**: $Q(s,a) = r + \gamma \mathbb{E}_{s'}[V(s')]$ where $V(s) = \mathbb{E}_{a\sim\pi}[Q(s,a) - \alpha \log \pi(a|s)]$
- **Policy**: π_θ is updated to minimize: $\mathbb{E}_{a\sim\pi_\theta}[\alpha \log \pi_\theta(a|s) - Q_\phi(s,a)]$
- **Temperature**: α is automatically tuned to maintain a target entropy $\bar{H}$.

SAC Soft Bellman Equation

$$Q(s,a) = R(s,a) + \gamma \mathbb{E}_{s'\sim P}\left[\mathbb{E}_{a'\sim\pi}\left[Q(s',a') - \alpha \log \pi(a'|s')\right]\right]$$

Proposition 7.11 (SAC Convergence)**.** In the tabular setting, repeated soft policy evaluation and soft policy improvement converge to the optimal policy of the maximum entropy objective. The entropy regularization ensures sufficient exploration and makes the optimization landscape smoother.

7.7 Entropy Regularization

Definition 7.12 (Entropy Bonus)**.** Adding an entropy term $H[\pi_\theta(\cdot|s)] = -\sum_a \pi_\theta(a|s) \log \pi_\theta(a|s)$ to the objective encourages exploration by penalizing peaked distributions. The coefficient α balances reward maximization with policy stochasticity.

Remark 7.13. Entropy regularization has several benefits:

- Prevents premature convergence to deterministic policies.
- Improves robustness to perturbations.
- Enables multi-modal policies that capture multiple near-optimal behaviors.
- In SAC, automatic temperature tuning adapts α throughout training.

7.8 Python Implementation

PPO (Simplified)

```
import torch
import torch.nn as nn
import numpy as np

def ppo_update(policy, value_fn, optimizer,
               states, actions, old_log_probs,
               returns, advantages,
               clip_eps=0.2, epochs=4, batch_size=64):
    states = torch.FloatTensor(np.array(states))
    actions = torch.LongTensor(actions)
    old_lp = torch.FloatTensor(old_log_probs)
    ret = torch.FloatTensor(returns)
```

```
    adv = torch.FloatTensor(advantages)
    adv = (adv - adv.mean()) / (adv.std() + 1e-8)

    for _ in range(epochs):
        idx = np.random.permutation(len(states))
        for start in range(0, len(states), batch_size):
            mb = idx[start:start+batch_size]
            probs = policy(states[mb])
            dist = torch.distributions.Categorical(probs)
            new_lp = dist.log_prob(actions[mb])
            ratio = torch.exp(new_lp - old_lp[mb])

            surr1 = ratio * adv[mb]
            surr2 = torch.clamp(ratio, 1-clip_eps,
                                1+clip_eps) * adv[mb]
            policy_loss = -torch.min(surr1, surr2).mean()
            value_loss = nn.MSELoss()(value_fn(states[mb]),
                                      ret[mb])
            entropy = dist.entropy().mean()
            loss = policy_loss + 0.5*value_loss - 0.01*entropy

            optimizer.zero_grad()
            loss.backward()
            nn.utils.clip_grad_norm_(
                list(policy.parameters())
                + list(value_fn.parameters()), 0.5)
            optimizer.step()
```

7.9 Exercises

Exercise 7.1 (A2C Implementation)**.** Implement A2C with GAE on `CartPole-v1`. Compare different numbers of parallel workers $K \in \{1, 4, 8, 16\}$ and measure wall-clock time to convergence.

Exercise 7.2 (PPO Clipping Analysis)**.** Visualize the PPO clipped objective L^{CLIP} as a function of r_t for $\hat{A}_t > 0$ and $\hat{A}_t < 0$. Explain why the clipping prevents destructively large updates.

Exercise 7.3 (SAC on Continuous Control)**.** Implement SAC for `Pendulum-v1`. Compare with DDPG and TD3. Plot Q-value estimates and verify that SAC avoids overestimation.

Exercise 7.4 (Entropy Temperature Tuning)**.** Implement automatic entropy temperature tuning for SAC. Compare with fixed $\alpha \in \{0.01, 0.1, 0.5\}$ and explain the effect on exploration behavior.

Chapter 8

State-of-the-Art Algorithms

The landscape of reinforcement learning has expanded dramatically beyond the foundational methods of earlier chapters. Model-based RL, pioneered by approaches like Dyna (Sutton, 1991) and refined by MuZero (Schrittwieser et al., 2020), learns environment dynamics to plan ahead. Distributional RL, introduced by Bellemare, Dabney, and Munos (2017), models full return distributions rather than just expectations. Offline RL, formalized by Levine et al., learns policies from fixed datasets without further environment interaction. And decision transformers (Chen et al., 2021) recast RL as sequence modelling, using the power of transformer architectures. Each approach opens new possibilities and raises new challenges. This chapter surveys these frontier directions.

Intuition

This chapter covers four directions beyond model-free methods: model-based RL (Dyna, Dreamer, MuZero), distributional RL (C51, QR-DQN), offline RL (CQL), and decision transformers. Each addresses a different limitation of the algorithms seen so far.

8.1 Model-Based Reinforcement Learning

Definition 8.1 (Model-Based RL)**.** A **model-based** RL agent learns an explicit dynamics model $\hat{P}(s'|s,a)$ and reward model $\hat{R}(s,a)$, then uses them for planning or generating synthetic experience.

Proposition 8.2 (Advantages of Model-Based RL)**.**

- **Sample efficiency**: synthetic rollouts in the learned model reduce the number of real environment interactions.
- **Planning**: the model enables look-ahead search (e.g., Monte Carlo Tree Search).
- **Transfer**: the model captures environment structure that transfers across tasks.

Model Bias

Model errors compound over long rollouts, leading to exploitative policies that perform well in the model but poorly in reality. This is known as **model exploitation** and is the main challenge of model-based RL.

8.1.1 Dyna Architecture

Definition 8.3 (Dyna-Q)**.** Dyna-Q (Sutton, 1991) alternates between real experience, model learning, and simulated planning:

1. Interact with the real environment and update Q (direct RL).
2. Update the model $\hat{P}, \hat{R}$ from the transition.
3. Perform k simulated updates using the model (planning).

8.1.2 Dreamer

Definition 8.4 (Dreamer)**.** Dreamer (Hafner et al., 2020) learns a world model in a compact latent space and trains the policy entirely from imagined trajectories:

- **Representation model**: encodes observations into latent states z_t using a recurrent state-space model (RSSM).
- **Dynamics model**: predicts z_{t+1} from z_t and a_t.
- **Reward model**: predicts r_t from z_t.
- **Actor-critic**: trained on imagined rollouts in latent space.

DreamerV3 (2023) achieves human-level performance on Atari and continuous control with a single set of hyperparameters.

8.1.3 MuZero

Definition 8.5 (MuZero)**.** MuZero (Schrittwieser et al., 2020) learns a model that predicts only the quantities needed for planning — rewards, values, and policies — without reconstructing observations:

- **Representation**: $h(o_t) \to s_t$ (encodes observation)
- **Dynamics**: $g(s_t, a_t) \to (r_t, s_{t+1})$
- **Prediction**: $f(s_t) \to (p_t, v_t)$ (policy and value)

Planning is done via Monte Carlo Tree Search (MCTS) in the learned latent space. MuZero matches AlphaZero's performance in Go, chess, and shogi, while also achieving superhuman Atari play.

MuZero Training Loss

$$\mathcal{L}(\theta) = \sum_{k=0}^{K} \left[\ell^r(r_t^k, \hat{r}_t^k) + \ell^v(z_t^k, \hat{v}_t^k) + \ell^p(\pi_t^k, \hat{p}_t^k) + c \left\|\theta\right\|^2 \right]$$

where K is the unroll length, ℓ^r and ℓ^v are scalar losses, and ℓ^p is the cross-entropy between MCTS policy and predicted policy.

8.2 Distributional Reinforcement Learning

Definition 8.6 (Distributional RL)**.** Instead of estimating the expected return $Q(s,a) = \mathbb{E}[Z(s,a)]$, distributional RL models the full distribution of the random return $Z(s,a)$:

$$Z(s,a) \overset{d}{=} R(s,a) + \gamma Z(S', A')$$

where $\overset{d}{=}$ denotes equality in distribution.

Theorem 8.7 (Distributional Bellman Operator)**.** *The distributional Bellman operator $\mathcal{T}^\pi$ is a contraction in the Wasserstein metric $\bar{d}_p$:*

$$\bar{d}_p(\mathcal{T}^\pi Z_1, \mathcal{T}^\pi Z_2) \leq \gamma\, \bar{d}_p(Z_1, Z_2).$$

This guarantees convergence to a unique fixed-point distribution.

8.2.1 C51

Definition 8.8 (C51)**.** C51 (Bellemare et al., 2017) represents the return distribution as a categorical distribution over $N = 51$ fixed atoms $\{z_1, \ldots, z_N\}$ uniformly spaced in $[V_{\min}, V_{\max}]$:

$$Z_\theta(s,a) = \sum_{i=1}^{N} p_i(s,a;\theta)\, \delta_{z_i}$$

The distribution is updated by projecting the Bellman target $r + \gamma z_j$ onto the support atoms.

8.2.2 QR-DQN

Definition 8.9 (Quantile Regression DQN)**.** QR-DQN (Dabney et al., 2018) represents the return distribution via N quantile values $\{\theta_1, \ldots, \theta_N\}$ at fixed quantile fractions $\hat{\tau}_i = (2i-1)/(2N)$:

$$Z(s,a) \approx \frac{1}{N} \sum_{i=1}^{N} \delta_{\theta_i(s,a)}$$

The quantile values are trained using the quantile Huber loss:

$$\rho_\tau^\kappa(\delta) = \left|\tau - \mathbb{1}_{\{\delta<0\}}\right| \frac{L_\kappa(\delta)}{\kappa}$$

where L_κ is the Huber loss with threshold κ.

Remark 8.10. Distributional RL provides richer learning signals than scalar Q-values. Empirically, it leads to better feature representations and more stable optimization, even when the final policy only uses the mean.

8.3 Offline Reinforcement Learning

Definition 8.11 (Offline RL)**. Offline RL** (also called **batch RL**) learns a policy from a fixed dataset $\mathcal{D} = \{(s_i, a_i, r_i, s_i')\}_{i=1}^N$ collected by some behavior policy π_β, without any further environment interaction.

Distribution Shift

Offline RL suffers from **distribution shift**: the learned policy π may select actions in states rarely seen under π_β, leading to unreliable Q-value estimates for out-of-distribution state-action pairs.

Definition 8.12 (Conservative Q-Learning (CQL))**.** CQL (Kumar et al., 2020) penalizes Q-values for actions not in the dataset by adding a regularizer:

$$\mathcal{L}_{\text{CQL}} = \alpha \left(\mathbb{E}_{s \sim \mathcal{D}, a \sim \mu}[Q(s,a)] - \mathbb{E}_{s,a \sim \mathcal{D}}[Q(s,a)] \right) + \mathcal{L}_{\text{TD}}$$

where μ is a broad distribution (e.g., uniform). This pushes Q-values down for unseen actions and up for dataset actions.

Proposition 8.13 (CQL Lower Bound)**.** CQL produces Q-values that are a lower bound of the true Q-function for actions in the dataset (with high probability). This conservatism prevents the policy from selecting overestimated out-of-distribution actions.

8.4 Decision Transformers

Definition 8.14 (Decision Transformer)**.** The Decision Transformer (Chen et al., 2021) recasts RL as a sequence modeling problem. A causal Transformer processes the sequence:

$$(\hat{R}_1, s_1, a_1, \hat{R}_2, s_2, a_2, \ldots, \hat{R}_t, s_t) \to a_t$$

where $\hat{R}_t$ is the desired return-to-go. At test time, setting a high $\hat{R}_t$ conditions the model to generate high-performing trajectories.

Intuition

The Decision Transformer does not use Bellman equations, TD learning, or explicit value functions. Instead, it uses the pattern-recognition capabilities of Transformers to extract good behaviors from datasets. It works particularly well when the dataset contains diverse quality trajectories.

Example 8.15 (Decision Transformer Results)**.** On the D4RL benchmark, the Decision Transformer matches or exceeds CQL and other offline RL methods on locomotion and Atari tasks, demonstrating that sequence modeling provides a viable alternative to traditional RL algorithms.

8.5 Comparison of Approaches

Method	Model	Online	Key Strength
DQN / Rainbow	Free	Yes	Simplicity
PPO / SAC	Free	Yes	Stability / exploration
Dreamer / MuZero	Learned	Yes	Sample efficiency
CQL	Free	No	Fixed datasets
Decision Transformer	Free	No	Sequence modeling

8.6 Python Example: Offline RL with CQL

CQL Regularization (Sketch)

```python
import torch
import torch.nn.functional as F

def cql_loss(q_net, states, actions, rewards, next_states,
             dones, gamma=0.99, alpha=1.0, n_samples=10):
    """CQL loss: standard TD + conservative regularizer."""
    # Standard TD loss
    with torch.no_grad():
        q_next = q_net(next_states).max(1)[0]
        targets = rewards + gamma * q_next * (1 - dones)
    q_vals = q_net(states).gather(
        1, actions.unsqueeze(1)).squeeze()
    td_loss = F.mse_loss(q_vals, targets)

    # CQL regularizer: push down Q for random actions
    batch_size = states.shape[0]
    random_actions = torch.randint(
        0, q_net.net[-1].out_features,
        (batch_size, n_samples))
    q_all = q_net(states)  # (B, A)
    logsumexp = torch.logsumexp(q_all, dim=1).mean()
    q_data = q_vals.mean()
    cql_reg = alpha * (logsumexp - q_data)

    return td_loss + cql_reg
```

8.7 Exercises

Exercise 8.1 (Dyna-Q)**.** Implement Dyna-Q with a tabular model on `FrozenLake-v1`. Vary the number of planning steps $k \in \{0, 5, 50\}$ and compare convergence speed.

Exercise 8.2 (Distributional RL)**.** Implement C51 on `CartPole-v1`. Visualize the learned return distributions for different states and compare with standard DQN.

Exercise 8.3 (Offline RL Challenge)**.** Collect a dataset of 10000 transitions using a partially trained DQN agent on `LunarLander-v2`. Train a policy using only this dataset (no further interaction). Compare naive DQN training on the dataset with CQL.

Exercise 8.4 (Decision Transformer)**.** Explain why conditioning on return-to-go $\hat{R}_t$ enables the Decision Transformer to produce different quality behaviors. What happens if $\hat{R}_t$ is set higher than any return observed in the training data?

Chapter 9

Multi-Agent Reinforcement Learning

What happens when multiple agents learn simultaneously in the same environment? The question is fundamental: the real world is populated by interacting agents — chess players, robots in a warehouse, autonomous vehicles on a road, trading algorithms on a market. The transition from the MDP (single agent) to the *stochastic game* (multiple agents) introduces formidable challenges: the environment becomes *non-stationary* from each agent's perspective (since the other agents are also learning), and the joint action space explodes combinatorially. The work of Littman (1994) on Markov games, followed by Lowe et al. (MADDPG, 2017) and Rashid et al. (QMIX, 2018), paved the way for methods that balance centralization and decentralization.

Intuition

Multi-Agent Reinforcement Learning (MARL) extends RL to settings with multiple interacting agents. Each agent's optimal behavior depends on the other agents' policies, creating non-stationary environments from each agent's perspective. MARL encompasses cooperative, competitive, and mixed settings, with applications ranging from traffic control to multi-player games.

9.1 Problem Formulation

Definition 9.1 (Stochastic Game (Markov Game))**.** A **stochastic game** for n agents is defined by $(\mathcal{S}, \{\mathcal{A}_i\}_{i=1}^n, P, \{R_i\}_{i=1}^n, \gamma)$ where:

- $\mathcal{S}$ is the shared state space
- $\mathcal{A}_i$ is the action space of agent i
- $P(s'|s, a_1, \ldots, a_n)$ is the joint transition function
- $R_i(s, a_1, \ldots, a_n)$ is the reward function for agent i
- $\gamma \in [0, 1)$ is the discount factor

Definition 9.2 (Nash Equilibrium)**.** A joint policy $(\pi_1^*, \ldots, \pi_n^*)$ is a **Nash equilibrium** if no agent can improve its expected return by unilaterally changing its policy:

$$\forall i, \forall \pi_i : \quad J_i(\pi_i^*, \pi_{-i}^*) \geq J_i(\pi_i, \pi_{-i}^*)$$

where π_{-i}^* denotes the policies of all agents except i.

Proposition 9.3 (Existence of Nash Equilibrium)**.** Every finite stochastic game has at least one Nash equilibrium (possibly in mixed strategies). However, finding a Nash equilibrium is PPAD-complete in general, even for two-player games.

9.2 Cooperative vs Competitive Settings

Definition 9.4 (Cooperative MARL)**.** In **fully cooperative** settings, all agents share a common reward: $R_1 = R_2 = \cdots = R_n = R$. The goal is to find a joint policy maximizing the shared return.

Definition 9.5 (Zero-Sum Games)**.** In a **two-player zero-sum** game: $R_1 = -R_2$. One agent's gain is the other's loss. The solution concept is the **minimax** policy:

$$\pi_1^* = \arg\max_{\pi_1} \min_{\pi_2} J_1(\pi_1, \pi_2).$$

Example 9.6 (Mixed Settings)**.** Many real-world problems involve mixed incentives. In autonomous driving, cars cooperate to avoid collisions but compete for lane space. In team sports, players cooperate within teams but compete between teams.

9.3 Independent Learners

Definition 9.7 (Independent Q-Learning (IQL))**.** Each agent i independently runs Q-Learning, treating other agents as part of the environment:

$$Q_i(s, a_i) \leftarrow Q_i(s, a_i) + \alpha \left[r_i + \gamma \max_{a_i'} Q_i(s', a_i') - Q_i(s, a_i) \right]$$

Non-Stationarity Problem

Independent learners face a non-stationary environment because other agents' policies change during training. This violates the Markov property from each agent's perspective and can prevent convergence. In practice, IQL often works surprisingly well despite lacking theoretical guarantees.

9.4 Centralized Training, Decentralized Execution (CTDE)

Definition 9.8 (CTDE Paradigm)**.** The **Centralized Training, Decentralized Execution** paradigm allows agents to share information during training (centralized critic, global state) but requires each agent to act using only local observations at execution time:

- **Training**: access to global state s, all actions $\mathbf{a}$, joint reward r
- **Execution**: agent i observes only o_i and selects $a_i = \pi_i(o_i)$

Intuition

CTDE resolves the tension between the benefits of shared information (stability, coordination) and the practical need for decentralized policies (communication constraints, partial observability). The centralized critic guides training but is discarded at deployment.

9.5 QMIX

Definition 9.9 (QMIX)**.** QMIX (Rashid et al., 2018) learns a joint action-value function Q_{tot} that factorizes as a monotonic function of individual Q-values:

$$Q_{\text{tot}}(s, \mathbf{a}) = f_{\text{mix}}\left(Q_1(o_1, a_1), \ldots, Q_n(o_n, a_n); s\right)$$

where f_{mix} is a mixing network with non-negative weights (ensured by hypernetworks conditioned on s).

Proposition 9.10 (IGM Condition)**.** The monotonicity constraint ensures **Individual-Global-Max** (IGM):

$$\arg\max_{\mathbf{a}} Q_{\text{tot}}(s, \mathbf{a}) = \left(\arg\max_{a_1} Q_1(o_1, a_1), \ldots, \arg\max_{a_n} Q_n(o_n, a_n)\right)$$

This allows decentralized greedy action selection.

Remark 9.11. QMIX can only represent monotonic value decompositions, which limits its expressiveness. Extensions like QPLEX and WQMIX address this by allowing more general decompositions while preserving the IGM property.

9.6 MAPPO

Definition 9.12 (MAPPO)**.** Multi-Agent PPO (Yu et al., 2022) applies PPO independently to each agent with a shared centralized value function:

- **Actor**: $\pi_{\theta_i}(a_i|o_i)$ for each agent i (decentralized)
- **Critic**: $V_\phi(s)$ uses the global state (centralized)
- **Training**: PPO clipped objective with shared advantages

Proposition 9.13 (MAPPO Effectiveness)**.** Despite its simplicity, MAPPO achieves competitive or superior performance to more complex MARL algorithms on cooperative benchmarks (SMAC, Hanabi, MPE). Key factors include parameter sharing, value normalization, and large batch sizes.

9.7 Communication Protocols

Definition 9.14 (Learned Communication)**.** In **differentiable communication**, agents learn to send messages m_i through a differentiable channel:

1. Agent i produces message $m_i = g_\psi(o_i)$
2. Messages are aggregated: $\bar{m}_i = \text{Aggregate}(\{m_j\}_{j \neq i})$
3. Agent i acts: $a_i = \pi_\theta(o_i, \bar{m}_i)$

Examples include CommNet, TarMAC, and IC3Net.

Example 9.15 (Emergent Communication)**.** In referential games, agents develop emergent communication protocols: a "speaker" agent describes an image using discrete symbols, and a "listener" agent identifies the image. The resulting languages exhibit properties like compositionality, reminiscent of natural language.

9.8 Emergent Behaviors

Example 9.16 (Hide and Seek)**.** In the OpenAI hide-and-seek environment, teams of hiders and seekers develop increasingly sophisticated strategies through self-play:

1. Seekers chase hiders directly.
2. Hiders learn to build shelters using movable boxes.
3. Seekers learn to use ramps to jump over walls.
4. Hiders learn to lock ramps before seekers can use them.

This illustrates the emergence of tool use and strategic reasoning purely from multi-agent competition.

Remark 9.17. Self-play in competitive settings can produce open-ended learning dynamics where agents continuously develop new strategies in an arms-race fashion. This has been central to breakthroughs in Go (AlphaGo/AlphaZero) and StarCraft (AlphaStar).

9.9 Python Example

Independent Q-Learning for Two Agents

```
import numpy as np

def independent_q_learning(env, n_episodes=5000,
                           alpha=0.1, gamma=0.99,
                           epsilon=0.1):
    """IQL for a two-agent grid world."""
    n_agents = 2
    nS = env.observation_space.n
    nA = env.action_space.n
    Q = [np.zeros((nS, nA)) for _ in range(n_agents)]

    for ep in range(n_episodes):
        obs = env.reset()  # obs[i] for agent i
        done = False
```

```
        while not done:
            actions = []
            for i in range(n_agents):
                if np.random.random() < epsilon:
                    a = np.random.randint(nA)
                else:
                    a = np.argmax(Q[i][obs[i]])
                actions.append(a)
            next_obs, rewards, done, info = env.step(actions)
            for i in range(n_agents):
                s, a, r, s2 = obs[i], actions[i], rewards[i], next_obs[i]
                Q[i][s, a] += alpha * (
                    r + gamma * np.max(Q[i][s2]) - Q[i][s, a])
            obs = next_obs
    return Q
```

9.10 Exercises

Exercise 9.1 (Matrix Games)**.** Implement independent Q-Learning for the Prisoner's Dilemma repeated game. Does the algorithm converge to the Nash equilibrium? How does the result change with different exploration rates?

Exercise 9.2 (QMIX Mixing Network)**.** Design and implement the QMIX mixing network with hypernetworks. Verify that the monotonicity constraint holds by checking that all mixing weights are non-negative.

Exercise 9.3 (CTDE Analysis)**.** Compare IQL, VDN (additive decomposition), and QMIX on the `spread` task from the Multi-Agent Particle Environment. Which method benefits most from centralized training?

Exercise 9.4 (Self-Play)**.** Implement self-play for Tic-Tac-Toe using tabular Q-Learning. Track the win rate of the latest policy against a random opponent over training. Does the agent converge to optimal play?

Chapter 10

Constrained and Safe RL

Intuition

Standard RL maximizes cumulative reward, but real-world applications require satisfying safety constraints: a robot must avoid collisions, an autonomous car must obey traffic laws, and an LLM must not generate harmful content. Constrained and safe RL formalize these requirements as constraints on cost functions, ensuring that the learned policy is both performant and safe.

10.1 Constrained Markov Decision Processes

Definition 10.1 (Constrained MDP)**.** A **Constrained MDP** (CMDP) extends the standard MDP with K cost functions $c_k : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ and corresponding thresholds d_k:

$$\max_{\pi} J(\pi) = \mathbb{E}_{\pi}\left[\sum_{t=0}^{\infty} \gamma^t R_t\right] \quad \text{s.t.} \quad J_{c_k}(\pi) = \mathbb{E}_{\pi}\left[\sum_{t=0}^{\infty} \gamma^t c_k(S_t, A_t)\right] \leq d_k, \quad k = 1, \ldots, K.$$

Theorem 10.2 (CMDP Optimality)**.** *Under standard regularity conditions, the optimal policy of a CMDP is a mixture of at most $K+1$ deterministic policies. If the feasible set is non-empty and the constraint functions are continuous, an optimal policy exists.*

Example 10.3 (Robot Navigation)**.** A mobile robot must reach a goal (reward) while limiting the probability of collision (cost). The CMDP formulation:

- Reward: $+1$ for reaching the goal, -0.01 per step
- Cost: $c(s, a) = \mathbb{1}_{\{s \in \mathcal{S}_{\text{unsafe}}\}}$
- Constraint: $J_c(\pi) \leq 0.05$ (collision rate below 5%)

10.2 Lagrangian Methods

Definition 10.4 (Lagrangian Relaxation)**.** The CMDP is reformulated as an unconstrained saddle-point problem using Lagrange multipliers $\lambda_k \geq 0$:

$$\min_{\lambda \geq 0} \max_{\pi} \; J(\pi) - \sum_{k=1}^{K} \lambda_k \left(J_{c_k}(\pi) - d_k\right) = \min_{\lambda \geq 0} \max_{\pi} \; L(\pi, \lambda).$$

Lagrangian Actor-Critic Update

Policy (primal):

$$\theta \leftarrow \theta + \alpha_\theta \nabla_\theta \left[J(\pi_\theta) - \sum_k \lambda_k J_{c_k}(\pi_\theta) \right]$$

Multipliers (dual):

$$\lambda_k \leftarrow \max\left(0,\ \lambda_k + \alpha_\lambda (J_{c_k}(\pi_\theta) - d_k)\right)$$

Lagrangian PPO for CMDPs

1. Initialize policy θ, value networks V_ϕ, $V^c_{\phi_k}$ for each cost, multipliers $\lambda_k = 0$
2. For each iteration:
   (a) Collect trajectories using π_θ
   (b) Estimate reward advantages $\hat{A}^r_t$ and cost advantages $\hat{A}^{c_k}_t$ using GAE
   (c) Update θ using PPO with modified advantage: $\hat{A}_t = \hat{A}^r_t - \sum_k \lambda_k \hat{A}^{c_k}_t$
   (d) Update multipliers: $\lambda_k \leftarrow \max(0, \lambda_k + \alpha_\lambda(\hat{J}_{c_k} - d_k))$

Theorem 10.5 (Lagrangian Duality for CMDPs)**.** *Under Slater's condition (there exists a strictly feasible policy), strong duality holds for CMDPs:*

$$\max_{\pi \in \Pi_C} J(\pi) = \min_{\lambda \geq 0} \max_{\pi} L(\pi, \lambda)$$

where Π_C is the set of feasible policies. The Lagrangian approach therefore converges to the optimal constrained policy.

10.3 Safe Exploration

Definition 10.6 (Safe Exploration)**. Safe exploration** requires that the agent satisfies safety constraints *during training*, not just at convergence. This is critical for physical systems where unsafe exploration causes irreversible damage.

Definition 10.7 (Safety Layer)**.** A **safety layer** projects the agent's action onto the safe set at each step:

$$a_{\text{safe}} = \arg\min_{a \in \mathcal{A}_{\text{safe}}(s)} \|a - a_{\text{proposed}}\|^2$$

where $\mathcal{A}_{\text{safe}}(s) = \{a : c(s,a) \leq 0\}$ is the set of safe actions in state s.

Proposition 10.8 (Lyapunov-Based Safety)**.** A Lyapunov function $L(s) \geq 0$ with $L(s) = 0$ only for safe states can certify safety. If the policy ensures:

$$\mathbb{E}[L(S_{t+1}) | S_t = s, A_t = a] \leq L(s) - \alpha L(s) + \epsilon$$

for some $\alpha > 0$ and small $\epsilon > 0$, then the system remains within a safe region with high probability.

10.4 Reward Shaping

Definition 10.9 (Potential-Based Reward Shaping)**.** Given a potential function $\Phi : \mathcal{S} \to \mathbb{R}$, the shaped reward is:

$$R'(s, a, s') = R(s, a, s') + \gamma\Phi(s') - \Phi(s).$$

Theorem 10.10 (Ng et al., 1999)**.** *Potential-based reward shaping preserves the optimal policy: the set of optimal policies under R' is identical to that under R. Any non-potential shaping function can change the optimal policy.*

Remark 10.11. Reward shaping is widely used to incorporate domain knowledge and accelerate learning. For safety, negative shaping near dangerous states (e.g., $\Phi(s) = -d(s, \mathcal{S}_{\text{unsafe}})^{-1}$) guides the agent away from unsafe regions without altering optimality.

10.5 RLHF: Reinforcement Learning from Human Feedback

Definition 10.12 (RLHF)**. Reinforcement Learning from Human Feedback** (Christiano et al., 2017; Ouyang et al., 2022) trains a policy to align with human preferences in three stages:

1. **Supervised fine-tuning** (SFT): fine-tune a pretrained LLM on demonstration data.
2. **Reward modeling**: train a reward model $r_\psi(x, y)$ from human preference comparisons $(y_w \succ y_l|x)$ using the Bradley-Terry model:
$$P(y_w \succ y_l|x) = \sigma\left(r_\psi(x, y_w) - r_\psi(x, y_l)\right)$$
3. **RL optimization**: optimize the policy π_θ using PPO with the learned reward, subject to a KL constraint from the reference policy π_{ref}:
$$\max_\theta \ \mathbb{E}_{x,y\sim\pi_\theta}\left[r_\psi(x, y)\right] - \beta\, D_{\text{KL}}\left(\pi_\theta \| \pi_{\text{ref}}\right)$$

RLHF Objective

$$J(\theta) = \mathbb{E}_{x\sim\mathcal{D}, y\sim\pi_\theta(\cdot|x)}\left[r_\psi(x, y) - \beta \log \frac{\pi_\theta(y|x)}{\pi_{\text{ref}}(y|x)}\right]$$

The KL penalty prevents the policy from diverging too far from the pretrained model, maintaining coherent language generation.

Intuition

RLHF bridges the gap between language model capabilities and human values. The reward model captures nuanced human preferences that are difficult to specify programmatically. The KL constraint acts as a safety mechanism, preventing reward hacking (exploiting quirks of the learned reward model).

Reward Hacking

Without the KL constraint, the policy may find adversarial inputs that maximize r_ψ without genuinely satisfying human preferences. This is a form of Goodhart's law: "when a measure becomes a target, it ceases to be a good measure."

10.6 Direct Preference Optimization (DPO)

Definition 10.13 (DPO)**.** Direct Preference Optimization (Rafailov et al., 2023) eliminates the explicit reward model by directly optimizing the policy from preferences:

$$\mathcal{L}_{\text{DPO}}(\theta) = -\mathbb{E}_{(x,y_w,y_l)}\left[\log \sigma\left(\beta \log \frac{\pi_\theta(y_w|x)}{\pi_{\text{ref}}(y_w|x)} - \beta \log \frac{\pi_\theta(y_l|x)}{\pi_{\text{ref}}(y_l|x)}\right)\right]$$

Proposition 10.14 (DPO-RLHF Equivalence)**.** The DPO loss is derived from the closed-form solution of the RLHF objective under the KL constraint. The optimal policy satisfies:

$$\pi^*(y|x) = \frac{\pi_{\text{ref}}(y|x)\exp(r(x,y)/\beta)}{Z(x)}$$

Substituting this into the Bradley-Terry model yields the DPO loss, which can be optimized without RL.

10.7 Python Example

Lagrangian PPO (Simplified)

```python
import torch

class LagrangianPPO:
    def __init__(self, n_constraints, init_lambda=0.0,
                 lr_lambda=0.01):
        self.lambdas = [init_lambda] * n_constraints
        self.lr = lr_lambda

    def modified_advantage(self, adv_reward, adv_costs):
        """Compute Lagrangian advantage."""
        adv = adv_reward.clone()
        for k, lam in enumerate(self.lambdas):
            adv -= lam * adv_costs[k]
        return adv

    def update_multipliers(self, cost_values, thresholds):
        """Dual ascent on Lagrange multipliers."""
        for k in range(len(self.lambdas)):
            self.lambdas[k] = max(
                0.0,
                self.lambdas[k]
                + self.lr * (cost_values[k] - thresholds[k])
```

)

10.8 Exercises

Exercise 10.1 (CMDP Formulation)**.** Formulate the autonomous braking problem as a CMDP. Define the state space, action space, reward, cost function, and threshold. What makes this problem challenging compared to standard RL?

Exercise 10.2 (Lagrangian Method)**.** Implement Lagrangian PPO on a grid world where the agent must reach a goal while avoiding certain cells (cost $= 1$ when entering a forbidden cell). Vary the cost threshold d and observe the trade-off between reward and constraint satisfaction.

Exercise 10.3 (Reward Shaping)**.** Design a potential-based shaping function for the CartPole environment that encourages keeping the pole upright without changing the optimal policy. Measure the speedup compared to the unshaped reward.

Exercise 10.4 (RLHF Pipeline)**.** Describe the three stages of RLHF for training a dialogue system. What happens if the reward model is trained on biased preference data? How can this be mitigated?

Chapter 11

Applications

Intuition

Reinforcement learning has moved from theoretical curiosity to transformative technology. From superhuman game play to robotic manipulation, autonomous driving, recommendation systems, and large language model alignment, RL is at the core of some of the most impactful AI systems. This chapter surveys key application domains, highlighting the RL techniques and challenges specific to each.

11.1 Game Playing

11.1.1 Board Games: Go, Chess, and Shogi

Definition 11.1 (AlphaGo and AlphaZero)**.** AlphaGo (Silver et al., 2016) combined supervised learning from expert games with RL self-play and MCTS. AlphaZero (2017) removed the need for human data entirely, learning from self-play alone:

- **Representation**: the board is encoded as a multi-channel image (stone positions, liberties, history).
- **Network**: a ResNet outputs both a policy $\mathbf{p}$ and a value v from the board state.
- **Training**: MCTS generates training targets; the network is trained to predict MCTS policy and game outcome.
- **Result**: superhuman in Go, chess, and shogi with a single algorithm and architecture.

Proposition 11.2 (Self-Play Improvement)**.** In two-player zero-sum games, self-play generates a curriculum of increasing difficulty. Each iteration of training against the current best policy produces a stronger opponent, driving continuous improvement without external data.

11.1.2 Video Games: Atari and StarCraft

Example 11.3 (Atari (DQN))**.** DQN (Mnih et al., 2015) demonstrated that a single deep RL algorithm can learn directly from pixels across 49 Atari games, achieving superhuman

performance on the majority. Rainbow (2018) pushed performance further by combining multiple DQN improvements.

Example 11.4 (StarCraft II (AlphaStar))**.** AlphaStar (Vinyals et al., 2019) achieved Grandmaster level in StarCraft II, a real-time strategy game with:

- Imperfect information (fog of war)
- Enormous action space ($\sim 10^{26}$ possible actions per step)
- Long-horizon planning (games last ~20 minutes)

Key techniques: multi-agent league training, Transformer-based architecture, pointer networks for action selection.

11.2 Robotics

Definition 11.5 (Sim-to-Real Transfer)**. Sim-to-real transfer** trains RL policies in simulation and deploys them on physical robots. The **reality gap** (mismatch between simulation and reality) is addressed via:

- **Domain randomization**: randomize physics parameters, textures, lighting, and noise during training.
- **System identification**: calibrate the simulator to match the real system.
- **Domain adaptation**: fine-tune in the real world with few interactions.

Example 11.6 (Dexterous Manipulation)**.** OpenAI demonstrated solving a Rubik's cube with a robotic hand (2019) using PPO with massive domain randomization (thousands of physics parameters varied during training). The policy transferred to the real robot without any real-world training.

Example 11.7 (Locomotion)**.** Legged locomotion has been a major success of deep RL:

- Anymal (ETH Zurich): PPO-trained quadruped traversing challenging terrain.
- Stanford Pupper / Unitree Go1: sim-to-real locomotion policies using teacher-student distillation.
- Key challenge: robustness to perturbations, terrain variation, and hardware damage.

Safety in Robotics

RL for robotics requires constrained and safe learning (Chapter 10). Physical hardware is expensive and fragile. Safety constraints during training (not just at convergence) are essential to prevent damage.

11.3 Autonomous Driving

Definition 11.8 (RL for Autonomous Driving)**.** RL approaches to autonomous driving typically address:

- **Decision making**: lane changes, merging, intersection navigation (often modeled as MDPs or POMDPs).
- **Motion planning**: generating smooth, collision-free trajectories.
- **End-to-end driving**: mapping sensor inputs directly to steering/acceleration commands.

Remark 11.9. Pure RL for autonomous driving faces challenges: safety-critical requirements, rare but important events (edge cases), and the need for interpretable decisions. In practice, RL is combined with rule-based safety layers, imitation learning, and planning.

Example 11.10 (Wayve and Tesla)**.** Industry applications include:

- **Wayve**: end-to-end RL-based driving in urban environments with sim-to-real transfer.
- **Tesla**: uses imitation learning from human drivers combined with RL for decision optimization in Autopilot/FSD.

11.4 Recommendation Systems

Definition 11.11 (RL for Recommendations)**.** A recommendation system can be modeled as an MDP:

- **State**: user interaction history, context, user profile.
- **Action**: item to recommend (from a large catalog).
- **Reward**: click, purchase, engagement time, long-term user satisfaction.

Proposition 11.12 (Long-Term vs Short-Term Optimization)**.** Greedy recommendation (maximize immediate click probability) often leads to filter bubbles and user fatigue. RL enables optimization of long-term engagement by modeling multi-step interactions and exploring diverse content.

Example 11.13 (Industry Applications)**.**

- **YouTube**: uses RL to optimize watch time while balancing user satisfaction and content diversity (REINFORCE-based).
- **Netflix**: offline RL methods to optimize session-level engagement from logged interaction data.
- **E-commerce**: multi-armed bandits and contextual bandits for product ranking and pricing.

11.5 LLM Alignment: RLHF and Beyond

Definition 11.14 (LLM Alignment)**. Alignment** refers to training large language models to be helpful, harmless, and honest (the 3H criteria). RLHF (Chapter 10) is the dominant approach:

1. Collect human preference data on model outputs.
2. Train a reward model on preferences.
3. Optimize the LLM policy via PPO with KL constraint.

Example 11.15 (ChatGPT and Claude)**.** Both ChatGPT (OpenAI) and Claude (Anthropic) use RLHF as a key training stage. The process transforms a capable but uncontrolled base model into an assistant that follows instructions, refuses harmful requests, and provides calibrated uncertainty.

Definition 11.16 (DPO for Alignment)**.** Direct Preference Optimization eliminates the separate reward model and RL loop, directly optimizing the policy from preference pairs. DPO has become popular for its simplicity and stability:

$$\mathcal{L}_{\text{DPO}} = -\mathbb{E}\left[\log \sigma\left(\beta \log \frac{\pi_\theta(y_w|x)}{\pi_{\text{ref}}(y_w|x)} - \beta \log \frac{\pi_\theta(y_l|x)}{\pi_{\text{ref}}(y_l|x)}\right)\right]$$

Remark 11.17. Recent alternatives to RLHF include:

- **DPO**: direct preference optimization (no RL loop).
- **RLAIF**: RL from AI feedback (using a model as judge).
- **Constitutional AI**: self-improvement via critique and revision guided by principles.
- **KTO**: Kahneman-Tversky Optimization (uses only binary good/bad labels, no pairwise comparisons).

11.6 Other Applications

Example 11.18 (Science and Engineering)**.**

- **Protein folding**: AlphaFold uses RL-like techniques for structure prediction.
- **Chip design**: Google's RL-based chip placement achieves superhuman floorplanning.
- **Nuclear fusion**: DeepMind's RL controller for plasma confinement in tokamaks.
- **Compiler optimization**: MLIR/LLVM use RL for register allocation and instruction scheduling.

Example 11.19 (Healthcare)**.** RL has been applied to:

- Treatment planning (e.g., sepsis management, ventilator control)
- Drug discovery and molecular optimization
- Adaptive clinical trials

These applications require offline RL (no live experimentation on patients) and rigorous safety constraints.

11.7 Challenges and Open Problems

Key Open Challenges

1. **Sample efficiency**: real-world interactions are expensive.
2. **Safety**: ensuring constraints during training and deployment.
3. **Generalization**: policies that transfer across tasks and environments.
4. **Reward specification**: designing rewards that capture true objectives (avoiding reward hacking).
5. **Scalability**: training with billions of parameters and massive action spaces.

11.8 Exercises

Exercise 11.1 (Game Environment)**.** Choose an Atari game from Gymnasium and train a DQN agent from scratch. Report the score after 1M frames and compare with human performance. Identify which game features make RL easy or hard.

Exercise 11.2 (Recommendation MDP)**.** Design an MDP for a music recommendation system. Define states, actions, rewards, and transitions. Discuss the trade-off between exploration (recommending new artists) and exploitation (playing known favorites).

Exercise 11.3 (RLHF Simulation)**.** Simulate a simplified RLHF pipeline: (1) generate text with a small GPT, (2) collect synthetic preferences using a rule-based scorer, (3) train a reward model, (4) fine-tune with PPO. Measure alignment improvement on a held-out test set.

Exercise 11.4 (Sim-to-Real Gap)**.** Train a PPO agent for `CartPole-v1` with randomized physics parameters (pole length, cart mass, gravity). Test on the default parameters. Does domain randomization improve robustness?

Bibliography

[1] Sutton, R.S. and Barto, A.G., *Reinforcement Learning: An Introduction*, 2nd ed., MIT Press, 2018.

[2] Bertsekas, D.P., *Dynamic Programming and Optimal Control*, 4th ed., Athena Scientific, 2017.

[3] Szepesvári, C., *Algorithms for Reinforcement Learning*, Morgan & Claypool, 2010.