Reinforcement Learning: Theory and Applications

Based on lectures by David Silver

May 23, 2025

Contents

Chapter 1

Introduction to Reinforcement Learning

1.1 What is Reinforcement Learning?

Reinforcement Learning (RL) represents a fundamental paradigm in machine learning that addresses the problem of learning optimal behavior through interaction with an environment. Unlike supervised learning, where an agent learns from labeled examples, or unsupervised learning, which discovers hidden patterns in data, reinforcement learning focuses on learning through trial and error to maximize cumulative reward.

1.1.1 The Many Faces of Reinforcement Learning

Reinforcement learning draws from multiple disciplines, creating a rich interdisciplinary field that encompasses:

- **Computer Science**: Providing computational frameworks and algorithmic approaches
- **Mathematics**: Offering rigorous theoretical foundations through probability theory and optimization
- **Engineering**: Contributing control theory and optimal control methodologies
- **Neuroscience**: Inspiring algorithms through understanding of brain reward systems
- **Psychology**: Providing insights from classical and operant conditioning
- **Economics**: Contributing game theory and decision-making under uncertainty

This interdisciplinary nature makes reinforcement learning particularly powerful, as it can draw upon insights from each field to develop more effective learning algorithms.

1.1.2 Characteristics of Reinforcement Learning

Reinforcement learning is distinguished from other machine learning paradigms by several key characteristics:

1. **No Supervisor**: There is no direct supervisor providing correct actions. Instead, the agent receives only a reward signal indicating the quality of its actions.

2. **Delayed Feedback**: Unlike supervised learning where feedback is immediate, in RL the consequences of actions may not be apparent until much later in time.
3. **Sequential Decision Making**: Time plays a crucial role, as the agent must make a sequence of decisions where each decision affects future states and opportunities.
4. **Non-i.i.d. Data**: The data encountered by the agent is not independently and identically distributed, as the agent's actions influence the subsequent data it observes.

1.1.3 Examples of Reinforcement Learning Applications

The versatility of reinforcement learning is demonstrated through its successful application to diverse domains:

- **Autonomous Systems**: Helicopter aerobatics, humanoid robot locomotion, and autonomous vehicle navigation
- **Game Playing**: Achieving superhuman performance in games like Backgammon, Go, and various Atari games
- **Finance**: Portfolio management and algorithmic trading strategies
- **Industrial Control**: Power station control and manufacturing optimization

1.2 The Reinforcement Learning Problem

1.2.1 The Reward Hypothesis

Central to reinforcement learning is the *reward hypothesis*, which provides a unified framework for understanding goals and objectives:

Definition 1.1 (Reward Hypothesis). *All goals can be described by the maximization of expected cumulative reward.*

This hypothesis, while seemingly restrictive, has proven remarkably general. It allows us to formalize diverse objectives ranging from game-playing success to robotic control tasks within a unified mathematical framework.

1.2.2 Sequential Decision Making

The essence of many reinforcement learning problems lies in sequential decision making, where:

- Actions may have long-term consequences
- Reward may be delayed
- It may be beneficial to sacrifice immediate reward for greater long-term gain

Consider financial investment as an illustrative example: purchasing a stock may provide no immediate reward but could yield significant returns months or years later. Similarly, in game playing, defensive moves may seem suboptimal in isolation but prove crucial for long-term victory.

1.2.3 The Agent-Environment Interface

The standard reinforcement learning setup involves an agent interacting with an environment through a well-defined interface. At each discrete time step t:

1. The agent observes the current state S_t
2. The agent selects an action A_t based on its current policy
3. The environment responds with a reward R_{t+1} and transitions to a new state S_{t+1}

This cycle continues, generating a trajectory of experience:

$$S_1, A_1, R_2, S_2, A_2, R_3, S_3, A_3, R_4, \ldots$$

1.3 History and State

1.3.1 The Role of History

The *history* H_t represents the sequence of all observations, actions, and rewards up to time t:

$$H_t = O_1, R_1, A_1, O_2, R_2, A_2, \ldots, A_{t-1}, O_t, R_t$$

The history contains all available information about the agent's experience. However, using the complete history for decision-making is typically impractical due to:

- Computational complexity that grows with time
- Storage requirements that become prohibitive
- Difficulty in learning effective policies from such high-dimensional representations

1.3.2 State as a Summary of History

State serves as a more tractable summary of the history that retains essential information for decision-making:

Definition 1.2 (State). *State is the information used to determine what happens next. Formally, state is a function of the history: $S_t = f(H_t)$.*

1.3.3 The Markov Property

A particularly important class of states satisfies the Markov property:

Definition 1.3 (Markov Property). *A state S_t is Markov if and only if:*

$$P[S_{t+1}|S_t] = P[S_{t+1}|S_1, S_2, \ldots, S_t]$$

The Markov property captures the intuitive notion that "the future is independent of the past given the present." When the current state is sufficient to predict the future, we can discard the entire history and focus solely on the current state for decision-making.

1.3.4 Types of Environments

Fully Observable Environments

In fully observable environments, the agent directly observes the environment state:

$$O_t = S_t^a = S_t^e$$

where S_t^a is the agent state, S_t^e is the environment state, and O_t is the observation. This scenario leads to a Markov Decision Process (MDP), which forms the theoretical foundation for much of reinforcement learning.

Partially Observable Environments

In partially observable environments, the agent receives incomplete information about the environment state. Examples include:

- A robot with limited sensor range
- A trading agent observing only current market prices
- A poker player seeing only public cards

These scenarios require the agent to construct its own state representation, leading to Partially Observable Markov Decision Processes (POMDPs). Common approaches include:

- Using complete history: $S_t^a = H_t$
- Maintaining beliefs over environment states: $S_t^a = (P[S_t^e = s^1], \ldots, P[S_t^e = s^n])$
- Using recurrent neural networks: $S_t^a = \sigma(S_{t-1}^a W_s + O_t W_o)$

1.4 Inside a Reinforcement Learning Agent

A reinforcement learning agent may incorporate one or more of the following components, each serving a distinct purpose in the learning and decision-making process.

1.4.1 Policy

The policy represents the agent's behavior function, mapping states to actions:

Definition 1.4 (Policy). *A policy π is a mapping from states to actions. It can be:*

- *Deterministic:* $a = \pi(s)$
- *Stochastic:* $\pi(a|s) = P[A_t = a|S_t = s]$

The policy is the core component that determines how the agent behaves in different situations. It can range from simple rule-based approaches to complex neural networks.

1.4.2 Value Function

The value function provides a mechanism for evaluating the desirability of states or state-action pairs:

Definition 1.5 (State Value Function). *The state value function $v_\pi(s)$ under policy π is the expected return starting from state s and following policy π:*

$$v_\pi(s) = \mathbb{E}_\pi[G_t | S_t = s]$$

where $G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots$ is the return.

The discount factor $\gamma \in [0, 1]$ determines the relative importance of immediate versus future rewards. When γ approaches 0, the agent becomes myopic, focusing only on immediate rewards. When γ approaches 1, the agent becomes far-sighted, giving nearly equal weight to future rewards.

1.4.3 Model

A model represents the agent's understanding of how the environment works:

Definition 1.6 (Model). *A model predicts what the environment will do next, consisting of:*

- *Transition model:* $P^a_{ss'} = P[S_{t+1} = s' | S_t = s, A_t = a]$
- *Reward model:* $R^a_s = \mathbb{E}[R_{t+1} | S_t = s, A_t = a]$

Models enable planning, allowing agents to simulate potential futures and choose actions accordingly. However, learning accurate models can be challenging, and model errors can propagate through planning procedures.

1.4.4 Categorizing Reinforcement Learning Agents

Based on the components they use, RL agents can be categorized as:

- **Value-based**: Use value functions with implicit policies (e.g., ϵ-greedy)
- **Policy-based**: Directly learn policies without value functions
- **Actor-Critic**: Combine both learned policies and value functions
- **Model-free**: Learn policies/values directly from experience
- **Model-based**: Learn environment models to aid planning

1.5 Learning and Planning

Reinforcement learning encompasses two fundamental paradigms for solving sequential decision problems:

1.5.1 Learning

In the learning paradigm:

- The environment is initially unknown
- The agent interacts with the environment
- The agent improves its policy through experience

This represents the classic RL scenario where an agent must discover optimal behavior through trial and error.

1.5.2 Planning

In the planning paradigm:

- A model of the environment is known
- The agent performs computations with its model
- The agent improves its policy through reasoning

Planning is also known as deliberation, reasoning, or search, and forms the basis of traditional AI approaches to sequential decision making.

1.5.3 The Exploration-Exploitation Dilemma

A fundamental challenge in reinforcement learning is balancing exploration and exploitation:

- **Exploration**: Gathering information about the environment to improve future decision-making
- **Exploitation**: Using current knowledge to maximize immediate reward

This dilemma appears in many real-world scenarios:

- Restaurant selection: trying new restaurants versus visiting proven favorites
- Online advertising: showing experimental ads versus proven successful ones
- Medical treatment: testing new treatments versus using established protocols

The optimal balance between exploration and exploitation depends on factors such as the remaining time horizon, the uncertainty in current knowledge, and the potential cost of exploration.

1.6 Prediction and Control

Reinforcement learning problems can be decomposed into two fundamental subproblems:

Definition 1.7 (Prediction Problem). *Given a policy π, evaluate the value function $v_\pi(s)$ for all states s.*

Definition 1.8 (Control Problem). *Find the optimal policy π^* that maximizes the value function.*

The prediction problem asks "how good is this policy?" while the control problem asks "what is the best policy?" Most RL algorithms alternate between these two phases, using policy evaluation to assess current behavior and policy improvement to enhance performance.

Chapter 2

Markov Decision Processes

Markov Decision Processes (MDPs) provide the mathematical foundation for formulating reinforcement learning problems. They offer a framework for modeling decision-making situations where outcomes are partly random and partly under the control of a decision maker.

2.1 Introduction to MDPs

Definition 2.1 (Markov Decision Process). *A Markov Decision Process is a tuple $\langle S, A, P, R, \gamma \rangle$ where:*

- *S is a finite set of states*
- *A is a finite set of actions*
- *P is a state transition probability matrix: $P^a_{ss'} = P[S_{t+1} = s' | S_t = s, A_t = a]$*
- *R is a reward function: $R^a_s = \mathbb{E}[R_{t+1} | S_t = s, A_t = a]$*
- *γ is a discount factor: $\gamma \in [0, 1]$*

MDPs formalize environments where:

- The environment is fully observable
- The current state completely characterizes the process
- Almost all RL problems can be formalized as MDPs

2.2 Markov Processes

Before introducing the full MDP framework, we begin with simpler components.

2.2.1 Markov Property Revisited

The Markov property is fundamental to the mathematical tractability of MDPs:

Theorem 2.2 (Markov Property). *A state S_t is Markov if and only if:*

$$P[S_{t+1} | S_t] = P[S_{t+1} | S_1, S_2, \ldots, S_t]$$

This property ensures that the state captures all relevant information from the history, making the future independent of the past given the present.

2.2.2 State Transition Matrix

For a Markov state s and successor state s', the state transition probability is:

$$P_{ss'} = P[S_{t+1} = s' | S_t = s]$$

The state transition matrix P defines transition probabilities from all states to all successor states:

$$P = \begin{pmatrix} P_{11} & P_{12} & \cdots & P_{1n} \\ P_{21} & P_{22} & \cdots & P_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ P_{n1} & P_{n2} & \cdots & P_{nn} \end{pmatrix}$$

where each row sums to 1: $\sum_{s'} P_{ss'} = 1$.

2.2.3 Markov Chains

Definition 2.3 (Markov Process). *A Markov Process (or Markov Chain) is a tuple $\langle S, P \rangle$ where:*

- *S is a finite set of states*
- *P is a state transition probability matrix*

A Markov process represents a memoryless random process—a sequence of random states with the Markov property.

2.3 Markov Reward Processes

Definition 2.4 (Markov Reward Process). *A Markov Reward Process (MRP) is a tuple $\langle S, P, R, \gamma \rangle$ where:*

- *S is a finite set of states*
- *P is a state transition probability matrix*
- *R is a reward function: $R_s = \mathbb{E}[R_{t+1} | S_t = s]$*
- *γ is a discount factor: $\gamma \in [0, 1]$*

An MRP extends a Markov chain by adding rewards, providing a foundation for evaluating the desirability of different states.

2.3.1 Return and Value Functions

Definition 2.5 (Return). *The return G_t is the total discounted reward from time step t:*

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

The discount factor γ serves several important purposes:

- Ensures mathematical convergence in cyclic processes

- Models uncertainty about the future
- Reflects preference for immediate rewards (financial interpretation)
- Matches animal and human behavior patterns

Definition 2.6 (State Value Function)**.** *The state value function $v(s)$ of an MRP is the expected return starting from state s:*

$$v(s) = \mathbb{E}[G_t | S_t = s]$$

2.3.2 Bellman Equation for MRPs

The value function can be decomposed recursively:

Theorem 2.7 (Bellman Equation for MRPs)**.** *The value function $v(s)$ satisfies:*

$$v(s) = \mathbb{E}[R_{t+1} + \gamma v(S_{t+1}) | S_t = s]$$

Proof.

$$\begin{aligned} v(s) &= \mathbb{E}[G_t | S_t = s] && (2.1) \\ &= \mathbb{E}[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots | S_t = s] && (2.2) \\ &= \mathbb{E}[R_{t+1} + \gamma (R_{t+2} + \gamma R_{t+3} + \cdots) | S_t = s] && (2.3) \\ &= \mathbb{E}[R_{t+1} + \gamma G_{t+1} | S_t = s] && (2.4) \\ &= \mathbb{E}[R_{t+1} + \gamma v(S_{t+1}) | S_t = s] && (2.5) \end{aligned}$$

□

This can be written more explicitly as:

$$v(s) = R_s + \gamma \sum_{s' \in S} P_{ss'} v(s')$$

In matrix form, the Bellman equation becomes:

$$v = R + \gamma P v$$

where v and R are column vectors.

2.3.3 Solving the Bellman Equation

For finite MRPs, the Bellman equation represents a system of linear equations that can be solved directly:

$$v = R + \gamma P v$$
$$(I - \gamma P) v = R$$
$$v = (I - \gamma P)^{-1} R$$

However, direct solution has computational complexity $O(n^3)$ for n states, making it impractical for large state spaces. Alternative iterative methods include:

- Dynamic programming
- Monte Carlo evaluation
- Temporal difference learning

2.4 Markov Decision Processes

2.4.1 MDP Definition and Components

A Markov Decision Process extends an MRP by adding actions, representing environments where all states are Markov and the agent can influence the state transitions.

The key addition is the action space A and the dependence of transitions and rewards on actions:

- $P^a_{ss'} = P[S_{t+1} = s' | S_t = s, A_t = a]$
- $R^a_s = \mathbb{E}[R_{t+1} | S_t = s, A_t = a]$

2.4.2 Policies

Definition 2.8 (Policy). *A policy π is a distribution over actions given states:*

$$\pi(a|s) = P[A_t = a | S_t = s]$$

Key properties of policies:

- A policy fully defines the behavior of an agent
- MDP policies depend only on the current state (Markov)
- Policies are stationary (time-independent)

Given an MDP and a policy π, the state sequence becomes a Markov process, and the state-reward sequence becomes an MRP with:

$$P^\pi_{s,s'} = \sum_{a \in A} \pi(a|s) P^a_{ss'} \tag{2.6}$$

$$R^\pi_s = \sum_{a \in A} \pi(a|s) R^a_s \tag{2.7}$$

2.4.3 Value Functions for MDPs

MDPs require two types of value functions:

Definition 2.9 (State Value Function). *The state value function $v_\pi(s)$ is the expected return starting from state s and following policy π:*

$$v_\pi(s) = \mathbb{E}_\pi[G_t | S_t = s]$$

Definition 2.10 (Action Value Function). *The action value function $q_\pi(s,a)$ is the expected return starting from state s, taking action a, and then following policy π:*

$$q_\pi(s,a) = \mathbb{E}_\pi[G_t | S_t = s, A_t = a]$$

2.4.4 Bellman Expectation Equations

The value functions satisfy recursive relationships:

Theorem 2.11 (Bellman Expectation Equations)**.**

$$v_\pi(s) = \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1})|S_t = s] \tag{2.8}$$
$$q_\pi(s,a) = \mathbb{E}_\pi[R_{t+1} + \gamma q_\pi(S_{t+1}, A_{t+1})|S_t = s, A_t = a] \tag{2.9}$$

These equations can be expressed in terms of the policy and transition dynamics:

$$v_\pi(s) = \sum_{a \in A} \pi(a|s) q_\pi(s,a) \tag{2.10}$$
$$q_\pi(s,a) = R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v_\pi(s') \tag{2.11}$$
$$v_\pi(s) = \sum_{a \in A} \pi(a|s) \left(R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v_\pi(s') \right) \tag{2.12}$$

In matrix form:

$$v_\pi = R^\pi + \gamma P^\pi v_\pi$$

with solution:

$$v_\pi = (I - \gamma P^\pi)^{-1} R^\pi$$

2.5 Optimal Value Functions and Policies

2.5.1 Optimal Value Functions

Definition 2.12 (Optimal State Value Function)**.** *The optimal state value function $v^*(s)$ is the maximum value function over all policies:*

$$v^*(s) = \max_\pi v_\pi(s)$$

Definition 2.13 (Optimal Action Value Function)**.** *The optimal action value function $q^*(s,a)$ is the maximum action value function over all policies:*

$$q^*(s,a) = \max_\pi q_\pi(s,a)$$

The optimal value function specifies the best possible performance in the MDP. An MDP is considered "solved" when we know the optimal value function.

2.5.2 Optimal Policies

Definition 2.14 (Policy Ordering)**.** *Define a partial ordering over policies: $\pi \geq \pi'$ if $v_\pi(s) \geq v_{\pi'}(s)$ for all $s \in S$.*

Theorem 2.15 (Existence of Optimal Policy)**.** *For any MDP:*

- *There exists an optimal policy π^* that is better than or equal to all other policies: $\pi^* \geq \pi$ for all π*

- *All optimal policies achieve the optimal value function:* $v_{\pi^*}(s) = v^*(s)$
- *All optimal policies achieve the optimal action value function:* $q_{\pi^*}(s,a) = q^*(s,a)$

Theorem 2.16 (Finding Optimal Policy)**.** *An optimal policy can be found by maximizing over* $q^*(s,a)$*:*

$$\pi^*(a|s) = \begin{cases} 1 & \text{if } a = \operatorname{argmax}_{a \in A} q^*(s,a) \\ 0 & \text{otherwise} \end{cases}$$

This theorem establishes that:

- There always exists a deterministic optimal policy for any MDP
- If we know $q^*(s,a)$, we immediately have the optimal policy

2.5.3 Bellman Optimality Equations

The optimal value functions satisfy the Bellman optimality equations:

Theorem 2.17 (Bellman Optimality Equations)**.**

$$v^*(s) = \max_a q^*(s,a) \tag{2.13}$$

$$q^*(s,a) = R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v^*(s') \tag{2.14}$$

$$v^*(s) = \max_a \left(R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v^*(s') \right) \tag{2.15}$$

$$q^*(s,a) = R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a \max_{a'} q^*(s',a') \tag{2.16}$$

Unlike the Bellman expectation equations, the Bellman optimality equations are nonlinear due to the max operation, and generally do not have closed-form solutions. Iterative solution methods include:

- Value iteration
- Policy iteration
- Q-learning
- Sarsa

2.6 Extensions to MDPs

2.6.1 Infinite and Continuous MDPs

The MDP framework can be extended to handle:

- **Countably infinite state/action spaces**: Straightforward extension
- **Continuous state/action spaces**: Requires measure theory; closed-form solutions exist for Linear Quadratic Regulators (LQR)
- **Continuous time**: Leads to Hamilton-Jacobi-Bellman (HJB) partial differential equations

2.6.2 Partially Observable MDPs

Definition 2.18 (POMDP)**.** *A Partially Observable MDP is a tuple* $\langle S, A, O, P, R, Z, \gamma \rangle$ *where:*

- S, A, P, R, γ *are as in standard MDPs*
- O *is a finite set of observations*
- Z *is an observation function:* $Z^a_{s'o} = P[O_{t+1} = o | S_{t+1} = s', A_t = a]$

In POMDPs, the agent maintains a belief state—a probability distribution over possible true states given the observation history. POMDPs can be reduced to MDPs over belief states, though this often results in infinite-dimensional state spaces.

2.6.3 Average Reward MDPs

For ergodic MDPs (where any state can be reached from any other state), we can consider average reward criteria:

Definition 2.19 (Average Reward)**.** *For any policy* π*, the average reward per time step is:*

$$\rho^\pi = \lim_{T\to\infty} \frac{1}{T} \mathbb{E}\left[\sum_{t=1}^{T} R_t\right]$$

This leads to average reward value functions and corresponding Bellman equations that focus on long-term average performance rather than discounted returns.

Chapter 3

Planning by Dynamic Programming

Dynamic Programming provides a class of algorithms for solving MDPs when the model (transition probabilities and rewards) is known. These methods break down complex problems into simpler subproblems and combine their solutions systematically.

3.1 Introduction to Dynamic Programming

3.1.1 What is Dynamic Programming?

Dynamic Programming (DP) refers to a collection of algorithms for solving complex problems by:

1. Breaking them down into simpler subproblems
2. Solving the subproblems
3. Combining solutions to subproblems

In the context of reinforcement learning, DP algorithms solve MDPs by leveraging the recursive structure of value functions expressed in the Bellman equations.

3.1.2 Requirements for Dynamic Programming

DP can be applied to problems that exhibit two key properties:

Definition 3.1 (Optimal Substructure). *A problem has optimal substructure if an optimal solution can be decomposed into subproblems, such that:*

- *The principle of optimality applies*
- *Optimal solutions to subproblems can be combined to form optimal solutions to the original problem*

Definition 3.2 (Overlapping Subproblems). *A problem has overlapping subproblems if:*

- *Subproblems recur many times*
- *Solutions can be cached and reused*

MDPs satisfy both properties:

- Bellman equations provide recursive decomposition (optimal substructure)
- Value functions store and reuse solutions (overlapping subproblems)

3.1.3 Planning vs. Learning

Dynamic programming assumes full knowledge of the MDP and is used for *planning*:

- **Input**: MDP $\langle S, A, P, R, \gamma \rangle$ and policy π (for prediction) or just MDP (for control)
- **Output**: Value function v_π (prediction) or optimal value function v^* and optimal policy π^* (control)

This contrasts with learning, where the MDP parameters are unknown and must be estimated from experience.

3.2 Policy Evaluation

Policy evaluation solves the prediction problem: given a policy π, compute the state value function v_π.

3.2.1 Iterative Policy Evaluation

The most straightforward approach applies the Bellman expectation equation as an update rule:

Algorithm 1 Iterative Policy Evaluation

```
Initialize $v_0(s)$ arbitrarily for all $s \in S$
for $k = 0, 1, 2, \ldots$ until convergence do
    for all $s \in S$ do
        $v_{k+1}(s) \leftarrow \sum_{a \in A} \pi(a|s) \left(R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v_k(s')\right)$
    end for
end for
```

This algorithm performs *synchronous backups*, updating all states simultaneously using values from the previous iteration.

Theorem 3.3 (Convergence of Iterative Policy Evaluation)**.** *Under the condition that either $\gamma < 1$ or the MDP has terminal states, iterative policy evaluation converges to the unique solution v_π.*

The convergence follows from the contraction mapping properties of the Bellman operator, which we discuss in detail later.

3.2.2 In-Place Policy Evaluation

To improve computational efficiency, we can perform *in-place* updates:

$$v(s) \leftarrow \sum_{a \in A} \pi(a|s) \left(R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v(s') \right)$$

In-place updates typically converge faster than synchronous updates because they immediately use improved estimates.

3.3 Policy Iteration

Policy iteration solves the control problem by alternating between policy evaluation and policy improvement.

3.3.1 Policy Improvement

Given a value function v_π for policy π, we can construct an improved policy π' using the greedy improvement:

$$\pi'(s) = \text{argmax}_{a \in A}\, q_\pi(s,a) = \text{argmax}_{a \in A} \left(R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v_\pi(s') \right)$$

Theorem 3.4 (Policy Improvement Theorem)**.** *Let π and π' be two deterministic policies such that for all $s \in S$:*

$$q_\pi(s, \pi'(s)) \geq v_\pi(s)$$

Then $\pi' \geq \pi$, i.e., $v_{\pi'}(s) \geq v_\pi(s)$ for all $s \in S$.

Proof.

$$\begin{aligned}
v_\pi(s) &\leq q_\pi(s, \pi'(s)) && (3.1)\\
&= \mathbb{E}[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s, A_t = \pi'(s)] && (3.2)\\
&\leq \mathbb{E}[R_{t+1} + \gamma q_\pi(S_{t+1}, \pi'(S_{t+1})) | S_t = s, A_t = \pi'(s)] && (3.3)\\
&\leq \mathbb{E}[R_{t+1} + \gamma R_{t+2} + \gamma^2 v_\pi(S_{t+2}) | S_t = s, A_t = \pi'(s)] && (3.4)\\
&\leq \cdots && (3.5)\\
&\leq v_{\pi'}(s) && (3.6)
\end{aligned}$$

□

3.3.2 Policy Iteration Algorithm

Algorithm 2 Policy Iteration

```
Initialize $\pi_0$ arbitrarily
for $k = 0, 1, 2, \dots$ until convergence do
    Policy Evaluation: Compute $v_{\pi_k}$ using iterative policy evaluation
    Policy Improvement: $\pi_{k+1}(s) \leftarrow \text{argmax}_{a \in A} \left(R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v_{\pi_k}(s')\right)$
end for
```

Theorem 3.5 (Convergence of Policy Iteration)**.** *Policy iteration converges to the optimal policy π^* in a finite number of iterations.*

Proof. Since there are only finitely many deterministic policies and each iteration strictly improves the policy (unless already optimal), convergence is guaranteed in finite time. □

3.3.3 Modified Policy Iteration

The policy evaluation step need not run to completion. We can stop after:

- A fixed number of iterations k
- When the value function change falls below threshold ϵ

When $k = 1$, this becomes equivalent to value iteration.

3.4 Value Iteration

Value iteration combines policy improvement and truncated policy evaluation into a single update.

3.4.1 Principle of Optimality

Theorem 3.6 (Principle of Optimality)**.** *A policy $\pi(a|s)$ achieves the optimal value from state s, $v_\pi(s) = v^*(s)$, if and only if for any state s' reachable from s, π achieves the optimal value from state s'.*

This principle justifies the recursive approach of value iteration: if we know the optimal values of successor states, we can compute the optimal value of the current state.

3.4.2 Value Iteration Algorithm

Algorithm 3 Value Iteration

```
Initialize $v_0(s)$ arbitrarily for all $s \in S$
for $k = 0, 1, 2, \ldots$ until convergence do
    for all $s \in S$ do
        $v_{k+1}(s) \leftarrow \max_{a\in A} \left(R_s^a + \gamma \sum_{s'\in S} P_{ss'}^a v_k(s')\right)$
    end for
end for
Extract policy: $\pi(s) \leftarrow \text{argmax}_{a\in A} \left(R_s^a + \gamma \sum_{s'\in S} P_{ss'}^a v(s')\right)$
```

Value iteration applies the Bellman optimality equation as an update rule. Unlike policy iteration, there is no explicit policy representation during the iterations.

Theorem 3.7 (Convergence of Value Iteration)**.** *Value iteration converges to the optimal value function v^* as $k \to \infty$.*

3.4.3 Comparison of DP Algorithms

Problem	Bellman Equation	Algorithm
Prediction	Bellman Expectation Equation	Iterative Policy Evaluation
Control	Bellman Expectation + Greedy Improvement	Policy Iteration
Control	Bellman Optimality Equation	Value Iteration

All algorithms have computational complexity $O(mn^2)$ per iteration for m actions and n states.

3.5 Asynchronous Dynamic Programming

The DP algorithms presented so far use synchronous backups, updating all states in each iteration. Asynchronous DP backs up states individually in any order, offering several advantages:

- Reduces computation per iteration
- Can focus updates on important states
- Allows online updating during interaction

3.5.1 Asynchronous DP Variants

In-Place Dynamic Programming

Update states using the most current values:

$$v(s) \leftarrow \max_{a \in A} \left(R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v(s') \right)$$

Prioritized Sweeping

Select states for updating based on the magnitude of Bellman error:

$$\left| \max_{a \in A} \left(R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v(s') \right) - v(s) \right|$$

States with larger errors are prioritized for updates.

Real-Time Dynamic Programming

Update only states visited by the agent:

- Agent interacts with environment
- After each transition $S_t \to S_{t+1}$, update $v(S_t)$
- Focuses computation on relevant states

3.6 Contraction Mapping Theory

The convergence properties of DP algorithms can be formally established using contraction mapping theory.

3.6.1 Value Function Space

Consider the vector space $\mathbb{R}^n$ of all possible value functions over n states. We measure the distance between value functions using the infinity norm:

$$\|u - v\|_\infty = \max_{s \in S} |u(s) - v(s)|$$

3.6.2 Bellman Operators

Definition 3.8 (Bellman Expectation Operator)**.** *The Bellman expectation operator T^π for policy π is defined as:*

$$(T^\pi v)(s) = \sum_{a \in A} \pi(a|s) \left(R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v(s') \right)$$

Definition 3.9 (Bellman Optimality Operator)**.** *The Bellman optimality operator T^* is defined as:*

$$(T^* v)(s) = \max_{a \in A} \left(R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v(s') \right)$$

3.6.3 Contraction Property

Theorem 3.10 (Contraction Property). *Both T^π and T^* are γ-contractions in the infinity norm:*

$$\|T^\pi u - T^\pi v\|_\infty \leq \gamma \|u - v\|_\infty \tag{3.7}$$
$$\|T^* u - T^* v\|_\infty \leq \gamma \|u - v\|_\infty \tag{3.8}$$

Proof sketch for T^π.

$$\begin{aligned}
|(T^\pi u)(s) - (T^\pi v)(s)| &= \left| \gamma \sum_{a,s'} \pi(a|s) P^a_{ss'} (u(s') - v(s')) \right| && (3.9)\\
&\leq \gamma \sum_{a,s'} \pi(a|s) P^a_{ss'} |u(s') - v(s')| && (3.10)\\
&\leq \gamma \|u - v\|_\infty \sum_{a,s'} \pi(a|s) P^a_{ss'} && (3.11)\\
&= \gamma \|u - v\|_\infty && (3.12)
\end{aligned}$$

□

3.6.4 Banach Fixed Point Theorem

Theorem 3.11 (Banach Fixed Point Theorem). *For any contraction mapping T on a complete metric space:*

- *T has a unique fixed point v^**
- *The sequence $v_0, Tv_0, T^2 v_0, \ldots$ converges to v^* for any starting point v_0*
- *The convergence is linear with rate γ*

3.6.5 Convergence of DP Algorithms

Applying the Banach fixed point theorem:

Corollary 3.12 (Convergence of Policy Evaluation). *Iterative policy evaluation converges linearly to the unique fixed point v_π of T^π.*

Corollary 3.13 (Convergence of Value Iteration). *Value iteration converges linearly to the unique fixed point v^* of T^*.*

The linear convergence rate γ means that the error decreases by at least a factor of γ in each iteration.

Chapter 4

Model-Free Prediction

While dynamic programming provides optimal solutions for known MDPs, many real-world problems involve unknown environments. Model-free methods learn value functions directly from experience without requiring knowledge of the MDP's transition probabilities or reward function.

4.1 Introduction to Model-Free Learning

Model-free reinforcement learning addresses the fundamental challenge of learning optimal behavior when the environment is unknown. Unlike planning methods that assume complete knowledge of the MDP, model-free approaches learn directly from sequences of experience.

4.1.1 The Model-Free Paradigm

Model-free methods share several key characteristics:

- **No model required**: Methods work without knowing $P^a_{ss'}$ or R^a_s
- **Learning from experience**: Algorithms improve estimates using observed transitions
- **Online capability**: Updates can be made incrementally during interaction

The trade-off is that model-free methods typically require more experience to achieve good performance compared to model-based approaches with perfect models.

4.2 Monte Carlo Methods

Monte Carlo (MC) methods represent the most straightforward approach to model-free value function estimation, using the simple principle that value equals mean return.

4.2.1 Monte Carlo Policy Evaluation

The goal is to learn v_π from episodes of experience under policy π:

$$S_1, A_1, R_2, S_2, A_2, R_3, \ldots, S_k \sim \pi$$

Since $v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s]$, we can estimate this expectation using sample means of observed returns.

Algorithm 4 First-Visit Monte Carlo Policy Evaluation

```
Initialize: $N(s) \leftarrow 0$, $S(s) \leftarrow 0$ for all $s \in \mathcal{S}$
repeat
  Generate episode following $\pi$: $S_1, A_1, R_2, \ldots, S_T$
  for each state $s$ appearing in the episode do
    $t \leftarrow$ first time $s$ is visited in the episode
    $N(s) \leftarrow N(s) + 1$
    $S(s) \leftarrow S(s) + G_t$
    $V(s) \leftarrow S(s)/N(s)$
  end for
until convergence
```

First-Visit Monte Carlo

Every-Visit Monte Carlo

Every-visit MC updates the value estimate for every occurrence of a state within an episode, not just the first occurrence.

Theorem 4.1 (Convergence of Monte Carlo)**.** *Both first-visit and every-visit Monte Carlo converge to $v_\pi(s)$ as the number of visits to state s approaches infinity, by the Law of Large Numbers.*

4.2.2 Incremental Monte Carlo

For computational efficiency and online learning, we can update value estimates incrementally:

Theorem 4.2 (Incremental Mean)**.** *The sample mean μ_k of a sequence $x_1, x_2, \ldots, x_k$ can be computed incrementally:*

$$\mu_k = \mu_{k-1} + \frac{1}{k}(x_k - \mu_{k-1})$$

This leads to the incremental Monte Carlo update:

$$V(S_t) \leftarrow V(S_t) + \alpha(G_t - V(S_t))$$

where $\alpha = 1/N(S_t)$ for the exact sample mean, or α can be a constant step size for non-stationary environments.

4.2.3 Advantages and Limitations of Monte Carlo

Advantages:

- Simple and intuitive
- Unbiased estimates (returns are unbiased samples of true values)
- No Markov assumption required
- Learns from complete episodes

Limitations:

- High variance (returns depend on many random factors)
- Requires episodic tasks (must wait for episode completion)
- Can be sample inefficient
- No learning during episodes

4.3 Temporal Difference Learning

Temporal Difference (TD) learning addresses many limitations of Monte Carlo methods by learning from incomplete episodes through bootstrapping.

4.3.1 The TD(0) Algorithm

The simplest TD algorithm, TD(0), makes the key insight that we can update value estimates using other value estimates:

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

The TD target $R_{t+1} + \gamma V(S_{t+1})$ serves as an estimate of G_t, and the TD error $\delta_t = R_{t+1} + \gamma V(S_{t+1}) - V(S_t)$ drives the update.

4.3.2 Comparison: MC vs TD

Monte Carlo	**Temporal Difference**
$V(S_t) \leftarrow V(S_t) + \alpha[G_t - V(S_t)]$	$V(S_t) \leftarrow V(S_t)+\alpha[R_{t+1}+\gamma V(S_{t+1})-V(S_t)]$
Target: G_t (actual return)	Target: $R_{t+1} + \gamma V(S_{t+1})$ (estimated return)
Unbiased estimate	Biased estimate (due to bootstrapping)
High variance	Lower variance
Requires complete episodes	Works with incomplete episodes
No bootstrapping	Bootstraps from current estimates

4.3.3 Bias-Variance Trade-off

- **MC Target** G_t: Unbiased estimate of $v_\pi(S_t)$ but high variance (depends on entire future trajectory)
- **TD Target** $R_{t+1} + \gamma V(S_{t+1})$: Biased estimate (since $V(S_{t+1}) \neq v_\pi(S_{t+1})$ initially) but lower variance (depends only on one step)

4.3.4 Advantages of TD Learning

1. **Online learning**: Updates after each step, not episode
2. **Continuing tasks**: Works in non-episodic environments
3. **Lower variance**: More stable learning
4. **Bootstrapping benefits**: Can learn faster by leveraging current estimates

4.3.5 Batch Learning Perspective

Consider learning from a finite batch of experience. What do MC and TD converge to?

Theorem 4.3 (MC Convergence in Batch Setting)**.** *MC converges to the solution that minimizes mean-squared error on the training set:*

$$\sum_{k=1}^{K}\sum_{t=1}^{T_k}(G_t^{(k)} - V(S_t^{(k)}))^2$$

Theorem 4.4 (TD Convergence in Batch Setting)**.** *TD(0) converges to the solution of the maximum likelihood MDP that best fits the data, i.e., the MDP $\langle \mathcal{S}, \mathcal{A}, \hat{P}, \hat{R}, \gamma \rangle$ where:*

$$\hat{P}_{ss'}^{a} = \frac{\sum_{k,t} \mathbf{1}(S_t^{(k)} = s, A_t^{(k)} = a, S_{t+1}^{(k)} = s')}{\sum_{k,t} \mathbf{1}(S_t^{(k)} = s, A_t^{(k)} = a)} \tag{4.1}$$

$$\hat{R}_{s}^{a} = \frac{\sum_{k,t} \mathbf{1}(S_t^{(k)} = s, A_t^{(k)} = a) R_{t+1}^{(k)}}{\sum_{k,t} \mathbf{1}(S_t^{(k)} = s, A_t^{(k)} = a)} \tag{4.2}$$

This difference explains why TD often performs better in Markov environments (it exploits the Markov structure) while MC may be preferred in non-Markov settings.

4.4 n-Step Methods and TD(λ)

n-Step methods bridge Monte Carlo and temporal difference learning by looking n steps ahead rather than just one step.

4.4.1 n-Step Returns

Define the n-step return as:

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

Special cases:

$$G_t^{(1)} = R_{t+1} + \gamma V(S_{t+1}) \quad \text{(TD target)} \tag{4.3}$$

$$G_t^{(\infty)} = G_t \quad \text{(MC return)} \tag{4.4}$$

The n-step TD update becomes:

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

4.4.2 Forward View of TD(λ)

Rather than using a single n-step return, TD(λ) combines all n-step returns using exponentially decaying weights:

Definition 4.5 (λ-return)**.**

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

The λ-return represents a weighted average of all n-step returns, where:

- $\lambda = 0$: Only 1-step return (TD(0))
- $\lambda = 1$: Only complete return (MC)
- $0 < \lambda < 1$: Mixture of n-step returns

The forward-view TD(λ) update is:

$$V(S_t) \leftarrow V(S_t) + \alpha[G_t^\lambda - V(S_t)]$$

4.4.3 Backward View of TD(λ): Eligibility Traces

The forward view requires looking into the future, making it unsuitable for online learning. The backward view provides an online algorithm using eligibility traces.

Eligibility Traces

Eligibility traces solve the credit assignment problem by combining frequency and recency heuristics:

Definition 4.6 (Eligibility Trace)**.** *For each state s, maintain an eligibility trace $E_t(s)$:*

$$E_0(s) = 0 \tag{4.5}$$
$$E_t(s) = \gamma\lambda E_{t-1}(s) + \mathbf{1}(S_t = s) \tag{4.6}$$

The eligibility trace tracks:

- **Recency**: How recently was state s visited?
- **Frequency**: How frequently has state s been visited?

Backward View TD(λ)

4.4.4 Equivalence of Forward and Backward Views

Theorem 4.7 (TD(λ) Equivalence)**.** *For offline learning (updates applied at episode end), the total update from forward-view TD(λ) equals the total update from backward-view TD(λ).*

This equivalence holds exactly for offline updates but only approximately for online updates. Recent work has developed exact online equivalents.

Algorithm 5 TD(λ) with Eligibility Traces

```
Initialize $V(s) = 0$ and $E(s) = 0$ for all $s$
repeat
  Take action $A_t$, observe $R_{t+1}, S_{t+1}$
  $\delta_t \leftarrow R_{t+1} + \gamma V(S_{t+1}) - V(S_t)$
  $E(S_t) \leftarrow E(S_t) + 1$
  for all $s \in \mathcal{S}$ do
    $V(s) \leftarrow V(s) + \alpha \delta_t E(s)$
    $E(s) \leftarrow \gamma \lambda E(s)$
  end for
until termination
```

4.4.5 TD(λ) Properties

- $\lambda = 0$: TD(0), pure bootstrapping
- $\lambda = 1$: Equivalent to Monte Carlo for episodic tasks
- $0 < \lambda < 1$: Interpolation between TD and MC
- Higher λ values increase variance but reduce bias
- Often $\lambda \in [0.8, 0.95]$ works well in practice

4.5 Convergence and Performance

4.5.1 Convergence Guarantees

Theorem 4.8 (Convergence of TD(0))**.** *Under the following conditions, TD(0) converges to v_π with probability 1:*

1. *The policy π is fixed*
2. *Step sizes satisfy: $\sum_{t=1}^{\infty} \alpha_t = \infty$ and $\sum_{t=1}^{\infty} \alpha_t^2 < \infty$*
3. *The states are visited infinitely often*

Similar convergence results hold for TD(λ) and Monte Carlo methods under appropriate conditions.

4.5.2 Performance Comparison

Empirical studies generally show:

- TD methods often learn faster than MC methods
- TD(λ) with intermediate λ values often outperforms both TD(0) and MC
- The optimal λ depends on the domain, function approximation, and step size
- TD methods are more robust to violations of the Markov property

The choice between MC and TD methods involves trade-offs between bias, variance, computational complexity, and domain characteristics.

Chapter 5

Model-Free Control

Building on model-free prediction methods, we now address the control problem: finding optimal policies without knowledge of the environment dynamics. Model-free control methods must solve the challenging problem of balancing exploration and exploitation while improving their policies.

5.1 Introduction to Model-Free Control

Model-free control extends the prediction methods from the previous chapter to solve the optimization problem of finding the best policy. The key challenge is that we must learn about the environment while simultaneously trying to optimize our behavior.

5.1.1 The Challenge of Model-Free Control

Unlike model-based control where we can compute optimal policies through planning, model-free control must:

- Learn value functions from experience
- Improve policies based on current estimates
- Balance exploration (learning about the environment) with exploitation (maximizing reward)
- Handle the fact that changing policies changes the distribution of experience

5.2 Monte Carlo Control

5.2.1 Generalized Policy Iteration for Control

Model-free control typically follows the generalized policy iteration (GPI) framework:

1. **Policy Evaluation**: Estimate value function for current policy
2. **Policy Improvement**: Update policy to be greedy with respect to value function

For model-free control, we must use action-value functions $Q(s, a)$ rather than state-value functions $V(s)$ because we cannot perform the greedy policy improvement without a model:

$$\text{Model-based: } \pi'(s) = \text{argmax}_a \left(R_s^a + \gamma \sum_{s'} P_{ss'}^a V(s') \right) \tag{5.1}$$

$$\text{Model-free: } \pi'(s) = \text{argmax}_a Q(s,a) \tag{5.2}$$

5.2.2 Monte Carlo Control Algorithm

Algorithm 6 Monte Carlo Control

```
Initialize: $Q(s,a) = 0$ for all $s, a$, $\pi$ arbitrary
repeat
    Policy Evaluation:
    Generate episode using $\pi$: $S_1, A_1, R_2, \ldots, S_T$
    for each pair $(s,a)$ appearing in the episode do
        $Q(s,a) \leftarrow$ average return following first occurrence of $(s,a)$
    end for
    Policy Improvement:
    for each state $s$ in the episode do
        $\pi(s) \leftarrow \text{argmax}_a Q(s,a)$
    end for
until convergence
```

5.2.3 The Exploration Problem

A critical issue with greedy policy improvement is that it may never visit state-action pairs that could be optimal. This leads to the *exploration problem*:

- **Greedy policy**: May miss optimal actions due to insufficient exploration
- **Need for exploration**: Must try actions to learn their values
- **Exploration-exploitation dilemma**: Balance learning with performance

5.2.4 ε-Greedy Exploration

The simplest approach to ensuring exploration is ε-greedy action selection:

Definition 5.1 (ε-Greedy Policy)**.**

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

This ensures that:

- All actions have non-zero probability
- The greedy action is selected with highest probability
- Exploration probability ϵ can be decayed over time

Theorem 5.2 (ε-Greedy Policy Improvement)**.** *For any ε-greedy policy π, the ε-greedy policy π' with respect to q_π is an improvement: $v_{\pi'}(s) \geq v_\pi(s)$ for all s.*

5.2.5 GLIE Monte Carlo Control

For theoretical convergence guarantees, we need the Greedy in the Limit with Infinite Exploration (GLIE) property:

Definition 5.3 (GLIE)**.** *A sequence of policies $\{\pi_k\}$ is GLIE if:*

1. *All state-action pairs are explored infinitely often:* $\lim_{k\to\infty} N_k(s,a) = \infty$
2. *The policy converges to a greedy policy:* $\lim_{k\to\infty} \pi_k(a|s) = \mathbf{1}(a = \operatorname{argmax}_{a'} Q_k(s,a'))$

ε-greedy policies are GLIE if ϵ_k decays to zero, e.g., $\epsilon_k = 1/k$.

Theorem 5.4 (GLIE MC Control Convergence)**.** *GLIE Monte Carlo control converges to the optimal action-value function: $Q(s,a) \to q^*(s,a)$.*

5.3 Temporal Difference Control

5.3.1 Sarsa: On-Policy TD Control

Sarsa (State-Action-Reward-State-Action) extends TD(0) to action-value functions:

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

Algorithm 7 Sarsa

```
Initialize $Q(s,a)$ arbitrarily, $s \in \mathcal{S}, a \in \mathcal{A}$
repeat
  Initialize $S$
  Choose $A$ from $S$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
  repeat
    Take action $A$, observe $R, S'$
    Choose $A'$ from $S'$ using policy derived from $Q$
    $Q(S,A) \leftarrow Q(S,A) + \alpha[R + \gamma Q(S',A') - Q(S,A)]$
    $S \leftarrow S', A \leftarrow A'$
  until $S$ is terminal
until convergence
```

Sarsa is called *on-policy* because it learns the value of the policy it is following (including exploration).

Theorem 5.5 (Sarsa Convergence)**.** *Sarsa converges to the optimal action-value function $q^*(s,a)$ under the following conditions:*

1. *GLIE sequence of policies*
2. *Robbins-Monro sequence of step sizes: $\sum_t \alpha_t = \infty$, $\sum_t \alpha_t^2 < \infty$*

5.3.2 Sarsa(λ): Eligibility Traces for Control

Sarsa can be extended to use eligibility traces:

Algorithm 8 Sarsa(λ)

```
Initialize $Q(s,a)$ and $E(s,a) = 0$ for all $s, a$
repeat
    Initialize $S$, choose $A$ using policy derived from $Q$
    repeat
        Take action $A$, observe $R, S'$
        Choose $A'$ from $S'$ using policy derived from $Q$
        $\delta \leftarrow R + \gamma Q(S', A') - Q(S, A)$
        $E(S, A) \leftarrow E(S, A) + 1$
        for all $s, a$ do
            $Q(s, a) \leftarrow Q(s, a) + \alpha \delta E(s, a)$
            $E(s, a) \leftarrow \gamma \lambda E(s, a)$
        end for
        $S \leftarrow S', A \leftarrow A'$
    until $S$ is terminal
until convergence
```

5.4 Off-Policy Learning

Off-policy methods learn about one policy (the target policy) while following another policy (the behavior policy). This separation provides several advantages:

- Learn optimal policy while following exploratory policy
- Reuse experience from old policies
- Learn from watching other agents
- Enable parallel learning of multiple policies

5.4.1 Importance Sampling

The key technique enabling off-policy learning is importance sampling:

Theorem 5.6 (Importance Sampling)**.** *To estimate $\mathbb{E}_{X\sim P}[f(X)]$ using samples from distribution Q:*

$$\mathbb{E}_{X\sim P}[f(X)] = \mathbb{E}_{X\sim Q}\left[\frac{P(X)}{Q(X)} f(X)\right]$$

For off-policy Monte Carlo, we weight returns by the importance sampling ratio:

$$G_t^{\pi/\mu} = \frac{\pi(A_t|S_t)}{\mu(A_t|S_t)} \frac{\pi(A_{t+1}|S_{t+1})}{\mu(A_{t+1}|S_{t+1})} \cdots \frac{\pi(A_T|S_T)}{\mu(A_T|S_T)} G_t$$

The off-policy MC update becomes:

$$V(S_t) \leftarrow V(S_t) + \alpha[G_t^{\pi/\mu} - V(S_t)]$$

Limitations of importance sampling:

- Can have infinite variance if behavior and target policies differ significantly
- Requires that $\mu(a|s) > 0$ whenever $\pi(a|s) > 0$
- Importance sampling ratios can become very large

5.4.2 Q-Learning: Off-Policy TD Control

Q-learning circumvents the importance sampling problems by using a simple insight:

Algorithm 9 Q-Learning

```
Initialize $Q(s, a)$ arbitrarily
repeat
  Initialize $S$
  repeat
    Choose $A$ from $S$ using behavior policy (e.g., $\varepsilon$-greedy)
    Take action $A$, observe $R, S'$
    $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma \max_{a'} Q(S', a') - Q(S, A)]$
    $S \leftarrow S'$
  until $S$ is terminal
until convergence
```

Key insight: The target policy is implicitly greedy with respect to Q, so we use $\max_{a'} Q(S', a')$ rather than $Q(S', A')$.

Theorem 5.7 (Q-Learning Convergence)**.** *Q-learning converges to the optimal action-value function $q^*(s, a)$ under mild conditions, regardless of the behavior policy (as long as all actions continue to be tried).*

5.4.3 Expected Sarsa

Expected Sarsa combines ideas from Sarsa and Q-learning:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \mathbb{E}_\pi[Q(S_{t+1}, A_{t+1})|S_{t+1}] - Q(S_t, A_t)\right]$$

$$= Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \sum_a \pi(a|S_{t+1}) Q(S_{t+1}, a) - Q(S_t, A_t)\right]$$

Expected Sarsa:

- Reduces variance compared to Sarsa
- Can be used on-policy or off-policy
- Often performs better than both Sarsa and Q-learning

5.5 Maximization Bias and Double Learning

5.5.1 The Maximization Bias Problem

All the control algorithms we've discussed suffer from maximization bias:

Definition 5.8 (Maximization Bias)**.** *If we have noisy estimates $Q(s, a)$ of true values $q(s, a)$, then:*

$$\mathbb{E}[\max_a Q(s, a)] \geq \max_a \mathbb{E}[Q(s, a)] = \max_a q(s, a)$$

This bias can lead to suboptimal policies because the algorithm overestimates the value of suboptimal actions due to noise.

5.5.2 Double Q-Learning

Double Q-learning addresses maximization bias by maintaining two value functions:

Algorithm 10 Double Q-Learning

```
Initialize $Q_1(s, a)$ and $Q_2(s, a)$ arbitrarily
repeat
  Initialize $S$
  repeat
    Choose $A$ from $S$ using $\epsilon$-greedy policy w.r.t. $Q_1 + Q_2$
    Take action $A$, observe $R, S'$
    With probability 0.5:
      $Q_1(S, A) \leftarrow Q_1(S, A) + \alpha[R + \gamma Q_2(S', \text{argmax}_{a'} Q_1(S', a')) - Q_1(S, A)]$
    Else:
      $Q_2(S, A) \leftarrow Q_2(S, A) + \alpha[R + \gamma Q_1(S', \text{argmax}_{a'} Q_2(S', a')) - Q_2(S, A)]$
    $S \leftarrow S'$
  until $S$ is terminal
until convergence
```

The key insight is to use one function to select actions and the other to evaluate them, breaking the maximization bias.

5.6 Summary of Control Algorithms

Algorithm	Type	Backup	Target
MC Control	On-policy	Sample	G_t
Sarsa	On-policy	Sample	$R_{t+1} + \gamma Q(S_{t+1}, A_{t+1})$
Q-learning	Off-policy	Sample	$R_{t+1} + \gamma \max_{a'} Q(S_{t+1}, a')$
Expected Sarsa	On/Off-policy	Sample	$R_{t+1} + \gamma \sum_a \pi(a \vert S_{t+1}) Q(S_{t+1}, a)$

The relationship between these algorithms mirrors the relationship between DP algorithms:

- MC Control $\leftrightarrow$ Policy Iteration (complete backup, policy evaluation to completion)
- Sarsa $\leftrightarrow$ Policy Iteration (sample backup, policy evaluation)
- Q-learning $\leftrightarrow$ Value Iteration (sample backup, policy improvement)

Each algorithm represents different trade-offs between computational complexity, sample efficiency, convergence guarantees, and practical performance.

Chapter 6

Value Function Approximation

The tabular methods discussed in previous chapters are limited to problems with small state and action spaces. For problems with large or continuous state spaces, we must approximate value functions using parameterized function approximators. This chapter addresses the fundamental challenge of scaling reinforcement learning to real-world problems.

6.1 The Need for Function Approximation

6.1.1 Limitations of Tabular Methods

Tabular reinforcement learning methods store a separate value for each state (and action). This approach faces severe limitations:

- **Memory requirements**: Storing $|\mathcal{S}|$ values becomes prohibitive for large state spaces
- **Learning speed**: Learning the value of each state individually is slow
- **Generalization**: No mechanism to share information between similar states
- **Continuous spaces**: Impossible to represent infinite state spaces

Real-world examples demonstrate the scale of the problem:

- Backgammon: $\approx 10^{20}$ states
- Computer Go: $\approx 10^{170}$ states
- Autonomous helicopter: Continuous state space

6.1.2 Function Approximation Solution

Function approximation addresses these limitations by:

- **Compact representation**: Using parameters $\mathbf{w} \in \mathbb{R}^d$ where $d \ll |\mathcal{S}|$
- **Generalization**: Sharing information between similar states
- **Continuous spaces**: Naturally handling infinite state spaces

We approximate value functions as:

$$\hat{v}(s, \mathbf{w}) \approx v_\pi(s) \tag{6.1}$$

$$\hat{q}(s, a, \mathbf{w}) \approx q_\pi(s, a) \tag{6.2}$$

6.2 Types of Function Approximators

6.2.1 Linear Function Approximation

The simplest and most well-studied approach uses linear combinations of features:

Definition 6.1 (Linear Value Function Approximation).

$$\hat{v}(s, \mathbf{w}) = \mathbf{x}(s)^\top \mathbf{w} = \sum_{j=1}^{d} x_j(s) w_j$$

where $\mathbf{x}(s) \in \mathbb{R}^d$ *is a feature vector representing state* s.

Advantages of linear approximation:

- Simple and efficient to compute
- Well-understood theoretical properties
- Guaranteed convergence for many algorithms
- Easy to interpret

Feature engineering challenges:

- Requires domain knowledge to design good features
- Limited representational power
- May need many features for complex functions

6.2.2 Nonlinear Function Approximation

For more complex problems, nonlinear approximators offer greater representational power:

- **Neural networks**: Universal function approximators
- **Decision trees**: Handle discrete and categorical features well
- **Kernel methods**: Flexible basis function approaches
- **Fourier/wavelet bases**: Useful for specific function classes

The trade-off is typically between representational power and theoretical guarantees.

6.3 Prediction with Function Approximation

6.3.1 The Objective Function

We frame value function approximation as a supervised learning problem. The natural objective is the mean squared value error:

$$J(\mathbf{w}) = \mathbb{E}_\pi[(v_\pi(S) - \hat{v}(S, \mathbf{w}))^2]$$

However, we don't know the true values $v_\pi(S)$, so we use targets from our RL algorithms.

6.3.2 Stochastic Gradient Descent

To minimize $J(\mathbf{w})$, we use stochastic gradient descent:

Theorem 6.2 (SGD for Value Function Approximation)**.** *The stochastic gradient descent update is:*

$$\begin{aligned}\mathbf{w}_{t+1} &= \mathbf{w}_t - \frac{\alpha}{2}\nabla_{\mathbf{w}}(v_\pi(S_t) - \hat{v}(S_t, \mathbf{w}_t))^2 \\ &= \mathbf{w}_t + \alpha(v_\pi(S_t) - \hat{v}(S_t, \mathbf{w}_t))\nabla_{\mathbf{w}}\hat{v}(S_t, \mathbf{w}_t)\end{aligned}$$

The gradient $\nabla_{\mathbf{w}}\hat{v}(S_t, \mathbf{w}_t)$ depends on the choice of function approximator:

- **Linear**: $\nabla_{\mathbf{w}}\hat{v}(s, \mathbf{w}) = \mathbf{x}(s)$
- **Neural network**: Computed via backpropagation

6.3.3 Monte Carlo with Function Approximation

MC methods have the advantage that the target G_t is an unbiased estimate of $v_\pi(S_t)$:

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha(G_t - \hat{v}(S_t, \mathbf{w}_t))\nabla_{\mathbf{w}}\hat{v}(S_t, \mathbf{w}_t)$$

Theorem 6.3 (MC Function Approximation Convergence)**.** *For linear function approximation, MC methods converge to the global optimum of the mean squared error. For nonlinear approximators, convergence is to a local optimum.*

6.3.4 Temporal Difference with Function Approximation

TD methods use the biased but lower-variance target $R_{t+1} + \gamma\hat{v}(S_{t+1}, \mathbf{w}_t)$:

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha(R_{t+1} + \gamma\hat{v}(S_{t+1}, \mathbf{w}_t) - \hat{v}(S_t, \mathbf{w}_t))\nabla_{\mathbf{w}}\hat{v}(S_t, \mathbf{w}_t)$$

The TD error becomes:

$$\delta_t = R_{t+1} + \gamma\hat{v}(S_{t+1}, \mathbf{w}_t) - \hat{v}(S_t, \mathbf{w}_t)$$

For linear function approximation:

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha\delta_t\mathbf{x}(S_t)$$

6.3.5 TD(λ) with Function Approximation

Eligibility traces extend naturally to function approximation:

6.4 Control with Function Approximation

6.4.1 Action-Value Function Approximation

For control, we approximate the action-value function:

$$\hat{q}(s, a, \mathbf{w}) \approx q_\pi(s, a)$$

The policy is then derived as:

$$\pi(s) = \text{argmax}_a\, \hat{q}(s, a, \mathbf{w})$$

Algorithm 11 TD(λ) with Function Approximation

```
Initialize $\mathbf{w}$ and $\mathbf{e} = \mathbf{0}$
repeat
  Take action, observe $R, S'$
  $\delta \leftarrow R + \gamma \hat{v}(S', \mathbf{w}) - \hat{v}(S, \mathbf{w})$
  $\mathbf{e} \leftarrow \gamma\lambda\mathbf{e} + \nabla_{\mathbf{w}} \hat{v}(S, \mathbf{w})$
  $\mathbf{w} \leftarrow \mathbf{w} + \alpha\delta\mathbf{e}$
  $S \leftarrow S'$
until termination
```

6.4.2 Linear Action-Value Functions

For linear approximation:

$$\hat{q}(s, a, \mathbf{w}) = \mathbf{x}(s, a)^\top \mathbf{w}$$

where $\mathbf{x}(s, a)$ is a feature vector for state-action pairs.

Common feature construction approaches:

- **Separate features**: Different features for each action
- **Shared features**: Same state features combined with action indicators
- **Action-dependent features**: Features that depend on both state and action

6.4.3 Sarsa with Function Approximation

Algorithm 12 Sarsa with Function Approximation

```
Initialize $\mathbf{w}$ arbitrarily
repeat
  Initialize $S$, choose $A$ using $\epsilon$-greedy w.r.t. $\hat{q}$
  repeat
    Take action $A$, observe $R, S'$
    Choose $A'$ using $\epsilon$-greedy w.r.t. $\hat{q}$
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha[R + \gamma \hat{q}(S', A', \mathbf{w}) - \hat{q}(S, A, \mathbf{w})]\nabla_{\mathbf{w}} \hat{q}(S, A, \mathbf{w})$
    $S \leftarrow S', A \leftarrow A'$
  until $S$ is terminal
until convergence
```

6.4.4 Q-learning with Function Approximation

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha[R_{t+1} + \gamma \max_{a'} \hat{q}(S_{t+1}, a', \mathbf{w}_t) - \hat{q}(S_t, A_t, \mathbf{w}_t)]\nabla_{\mathbf{w}} \hat{q}(S_t, A_t, \mathbf{w}_t)$$

6.5 Convergence of Function Approximation Methods

6.5.1 The Challenge of Convergence

Function approximation introduces new convergence challenges:

- Limited representational capacity
- Approximation errors that can accumulate
- Bootstrap bias interacting with function approximation
- Off-policy learning complications

6.5.2 Convergence Results

Algorithm	On-Policy	Linear FA	Nonlinear FA
MC	✓	✓	✓ (local)
TD(0)	✓	✓	$\times$
TD(λ)	✓	✓	$\times$
Sarsa	✓	✓	$\times$
Q-learning	$\times$	$\times$	$\times$

Theorem 6.4 (Linear TD Convergence)**.** *For on-policy learning with linear function approximation, TD(0) converges to:*

$$\mathbf{w}^* = \mathbf{A}^{-1}\mathbf{b}$$

where:

$$\mathbf{A} = \mathbb{E}[\mathbf{x}(S_t)(\mathbf{x}(S_t) - \gamma\mathbf{x}(S_{t+1}))^\top] \tag{6.3}$$

$$\mathbf{b} = \mathbb{E}[\mathbf{x}(S_t)R_{t+1}] \tag{6.4}$$

6.5.3 The Deadly Triad

The combination of three factors can cause instability:

1. **Function approximation**: Using limited representational capacity
2. **Bootstrapping**: Using estimates to update estimates (as in TD)
3. **Off-policy learning**: Training on data from a different policy

When all three are present, algorithms may diverge.

6.6 Deep Reinforcement Learning

6.6.1 Deep Q-Networks (DQN)

DQN revolutionized RL by successfully combining Q-learning with deep neural networks:

Key innovations:

- **Experience replay**: Breaks correlation in data
- **Target networks**: Stabilizes learning
- **Convolutional networks**: Processes raw visual input

Algorithm 13 DQN

```
Initialize replay memory $\mathcal{D}$ and Q-network $Q(s, a; \theta)$
Initialize target network $Q(s, a; \theta^-)$
for episode = 1 to M do
  for t = 1 to T do
    Select action $a_t = \epsilon$-greedy($Q(s_t, \cdot; \theta)$)
    Execute $a_t$, observe $r_t, s_{t+1}$
    Store $(s_t, a_t, r_t, s_{t+1})$ in $\mathcal{D}$
    Sample mini-batch from $\mathcal{D}$
    Set $y_j = r_j + \gamma \max_{a'} Q(s_{j+1}, a'; \theta^-)$
    Perform gradient step on $(y_j - Q(s_j, a_j; \theta))^2$
    Every $C$ steps: $\theta^- \leftarrow \theta$
  end for
end for
```

6.6.2 Policy Gradient Methods

Deep networks are also used in policy gradient methods, which we cover in detail in the next chapter. These methods directly parameterize the policy:

$$\pi(a|s, \boldsymbol{\theta}) = \text{neural network}(s; \boldsymbol{\theta})$$

6.7 Batch Methods

6.7.1 Experience Replay

Rather than updating immediately after each experience, batch methods collect experience and learn from it repeatedly:

Algorithm 14 Experience Replay

```
Initialize dataset $\mathcal{D} = \{(s_i, v_i^\pi)\}$
repeat
  Sample $(s, v^\pi) \sim \mathcal{D}$
  $\mathbf{w} \leftarrow \mathbf{w} + \alpha(v^\pi - \hat{v}(s, \mathbf{w}))\nabla_{\mathbf{w}}\hat{v}(s, \mathbf{w})$
until convergence
```

Advantages:

- Better sample efficiency
- Breaks temporal correlations
- Enables offline learning
- Can reuse experience multiple times

6.7.2 Least Squares Methods

For linear function approximation, we can solve the least squares problem directly:

Theorem 6.5 (Linear Least Squares Solution)**.** *The least squares solution for linear function approximation is:*

$$\mathbf{w}^* = (\mathbf{X}^\top \mathbf{X})^{-1}\mathbf{X}^\top \mathbf{v}$$

where $\mathbf{X}$ *is the feature matrix and* $\mathbf{v}$ *is the vector of targets.*

This leads to algorithms like:

- **LSMC**: Least Squares Monte Carlo
- **LSTD**: Least Squares Temporal Difference
- **LSTDQ**: Least Squares SARSA

These methods often have better sample efficiency than incremental methods but higher computational cost per update.

6.8 Summary

Function approximation is essential for scaling RL to real-world problems. Key insights include:

- Linear methods provide strong theoretical guarantees but limited representational power
- Nonlinear methods offer greater flexibility but weaker convergence guarantees
- The deadly triad (function approximation + bootstrapping + off-policy) can cause instability
- Experience replay and target networks help stabilize deep RL
- Batch methods can improve sample efficiency at the cost of computational complexity

The choice of function approximation method involves trade-offs between representational power, theoretical guarantees, computational efficiency, and practical performance.

Chapter 7

Policy Gradient Methods

While value-based methods learn value functions and derive policies implicitly, policy gradient methods directly parameterize and optimize the policy. This approach offers several advantages, particularly for problems with continuous action spaces or when stochastic policies are desired.

7.1 Introduction to Policy-Based Reinforcement Learning

7.1.1 Value-Based vs. Policy-Based Methods

The reinforcement learning approaches we have studied so far follow a value-based paradigm:

1. Learn a value function $V^\pi(s)$ or $Q^\pi(s,a)$
2. Derive policy implicitly (e.g., ϵ-greedy, greedy)

Policy-based methods take a different approach:

1. Directly parameterize the policy: $\pi_\theta(a|s) = P[a|s,\theta]$
2. Optimize the policy parameters θ directly

7.1.2 Advantages of Policy-Based Methods

Policy gradient methods offer several compelling advantages:

- **Better convergence properties**: Policy gradient methods have stronger convergence guarantees, often converging to local optima rather than oscillating
- **Effective in high-dimensional action spaces**: Natural for continuous action spaces where value-based methods struggle
- **Can learn stochastic policies**: Essential for environments where randomization is optimal (e.g., rock-paper-scissors, poker)
- **Smoother policy changes**: Small changes in parameters lead to small changes in policy, providing stability

7.1.3 Disadvantages of Policy-Based Methods

- **Local optima**: Typically converge to local rather than global optima
- **High variance**: Policy gradient estimates often have high variance, leading to slow learning
- **Sample inefficiency**: Often require more samples than value-based methods

7.1.4 Actor-Critic Methods

Actor-critic methods combine the advantages of both approaches:

- **Actor**: The policy (policy-based component)
- **Critic**: The value function (value-based component)

7.2 Policy Objective Functions

To optimize a policy, we need to define what makes one policy better than another.

7.2.1 Episodic Environments

For episodic environments with a clear starting state, we can use the start value: $J_1(\theta) = V^{\pi_\theta}(s_1) = \mathbb{E}_{\pi_\theta}[G_1]$

where s_1 is the initial state and G_1 is the return from the first time step.

7.2.2 Continuing Environments

For continuing environments, we have two main options:

Average Value

$J_{avV}(\theta) = \sum_s d^{\pi_\theta}(s) V^{\pi_\theta}(s)$

where $d^{\pi_\theta}(s)$ is the stationary distribution of states under policy π_θ.

Average Reward Rate

$J_{avR}(\theta) = \sum_s d^{\pi_\theta}(s) \sum_a \pi_\theta(a|s) R_s^a$

This represents the average reward per time step under the stationary distribution.

7.3 Policy Gradient Theorem

The fundamental challenge in policy optimization is computing gradients of the policy objective. The policy gradient theorem provides the foundation for this computation.

7.3.1 The Challenge of Computing Policy Gradients

Computing $\nabla_\theta J(\theta)$ is challenging because:

- The objective depends on the state distribution, which depends on the policy
- Changing the policy changes the states we visit
- The reward depends on the entire trajectory, not just immediate actions

7.3.2 Policy Gradient Theorem

Theorem 7.1 (Policy Gradient Theorem)**.** *For any differentiable policy $\pi_\theta(a|s)$ and for any policy objective function $J(\theta)$ (start value, average reward, or average value), the policy gradient is:*
$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(a|s) Q^{\pi_\theta}(s,a)]$

This remarkable result shows that the gradient depends only on:

- The current policy $\pi_\theta(a|s)$
- The action-value function $Q^{\pi_\theta}(s,a)$
- The score function $\nabla_\theta \log \pi_\theta(a|s)$

Importantly, it does *not* depend on the gradient of the state distribution.

7.3.3 Proof Sketch for Episodic Case

Proof sketch. For the episodic case with start value objective:

$$\begin{aligned}
\nabla_\theta J(\theta) &= \nabla_\theta V^{\pi_\theta}(s_0) && (7.1)\\
&= \nabla_\theta \left[\sum_a \pi_\theta(a|s_0) Q^{\pi_\theta}(s_0,a)\right] && (7.2)\\
&= \sum_a \left[\nabla_\theta \pi_\theta(a|s_0) Q^{\pi_\theta}(s_0,a) + \pi_\theta(a|s_0) \nabla_\theta Q^{\pi_\theta}(s_0,a)\right] && (7.3)
\end{aligned}$$

The key insight is that the $\nabla_\theta Q^{\pi_\theta}(s_0,a)$ terms, when expanded recursively, lead to a telescoping series where the state distribution dependencies cancel out, leaving only the score function terms. □

7.3.4 Score Function

The score function $\nabla_\theta \log \pi_\theta(a|s)$ is central to policy gradient methods. For common policy parameterizations:

Softmax Policy (Discrete Actions)

$\pi_\theta(a|s) = \frac{\exp(\phi(s,a)^\top \theta)}{\sum_{a'} \exp(\phi(s,a')^\top \theta)}$
The score function is: $\nabla_\theta \log \pi_\theta(a|s) = \phi(s,a) - \mathbb{E}_{\pi_\theta}[\phi(s,\cdot)]$

Gaussian Policy (Continuous Actions)

$\pi_\theta(a|s) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(a-\mu(s))^2}{2\sigma^2}\right)$
where $\mu(s) = \phi(s)^\top \theta$. The score function is: $\nabla_\theta \log \pi_\theta(a|s) = \frac{(a-\mu(s))\phi(s)}{\sigma^2}$

7.4 REINFORCE Algorithm

REINFORCE (REward Increment = Nonnegative Factor × Offset Reinforcement × Characteristic Eligibility) is the simplest policy gradient algorithm.

7.4.1 Monte Carlo Policy Gradient

Using the policy gradient theorem with Monte Carlo estimation: $\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(a|s) Q^{\pi_\theta}(s,a)]$

We can estimate $Q^{\pi_\theta}(s,a)$ using the return G_t from episodes: $\nabla_\theta J(\theta) \approx \nabla_\theta \log \pi_\theta(A_t|S_t) G_t$

7.4.2 REINFORCE Algorithm

Algorithm 15 REINFORCE

```
Initialize policy parameters $\theta$ arbitrarily
for each episode $\{S_1, A_1, R_2, \ldots, S_T, A_T, R_{T+1}\} \sim \pi_\theta$ do
  for $t = 1$ to $T$ do
    $\theta \leftarrow \theta + \alpha\gamma^t G_t \nabla_\theta \log \pi_\theta(A_t|S_t)$
  end for
end for
```

7.4.3 Properties of REINFORCE

Theorem 7.2 (REINFORCE Convergence). *REINFORCE converges to a local optimum of $J(\theta)$ under standard stochastic approximation conditions.*

Advantages:

- Unbiased gradient estimates
- Simple to implement
- Guaranteed convergence to local optimum

Disadvantages:

- High variance (entire episode return used)
- Slow convergence
- Only learns from complete episodes

7.5 Reducing Variance with Baselines

7.5.1 Baseline Functions

A key insight for reducing variance is that we can subtract any function $b(s)$ that depends only on the state:

Theorem 7.3 (Baseline Theorem). *For any function $b(s)$:* $\mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(a|s) b(s)] = 0$

Proof.

$$\begin{aligned}
&\mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(a|s) b(s)] && (7.4)\\
&= \sum_s d^{\pi_\theta}(s) \sum_a \pi_\theta(a|s) \nabla_\theta \log \pi_\theta(a|s) b(s) && (7.5)\\
&= \sum_s d^{\pi_\theta}(s) b(s) \sum_a \nabla_\theta \pi_\theta(a|s) && (7.6)\\
&= \sum_s d^{\pi_\theta}(s) b(s) \nabla_\theta \sum_a \pi_\theta(a|s) && (7.7)\\
&= \sum_s d^{\pi_\theta}(s) b(s) \nabla_\theta 1 = 0 && (7.8)
\end{aligned}$$

□

Therefore, we can use: $\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(a|s)(Q^{\pi_\theta}(s,a) - b(s))]$

7.5.2 Optimal Baseline

The variance-minimizing baseline is: $b^*(s) = \frac{\mathbb{E}[(\nabla_\theta \log \pi_\theta(a|s))^2 Q^{\pi_\theta}(s,a)]}{\mathbb{E}[(\nabla_\theta \log \pi_\theta(a|s))^2]}$

In practice, the state value function $V^{\pi_\theta}(s)$ is often used as a baseline, leading to the advantage function: $A^{\pi_\theta}(s,a) = Q^{\pi_\theta}(s,a) - V^{\pi_\theta}(s)$

7.5.3 REINFORCE with Baseline

Algorithm 16 REINFORCE with Baseline

```
Initialize policy parameters $\theta$ and baseline parameters $\mathbf{w}$
for each episode do
  Generate episode $\{S_1, A_1, R_2, \ldots, S_T, A_T, R_{T+1}\} \sim \pi_\theta$
  for $t = 1$ to $T$ do
    $G_t \leftarrow$ return from time $t$
    $\delta \leftarrow G_t - V_{\mathbf{w}}(S_t)$
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha_w \delta \nabla_{\mathbf{w}} V_{\mathbf{w}}(S_t)$
    $\theta \leftarrow \theta + \alpha_\theta \gamma^t \delta \nabla_\theta \log \pi_\theta(A_t|S_t)$
  end for
end for
```

7.6 Actor-Critic Methods

Actor-critic methods address the high variance of REINFORCE by using a learned value function to estimate returns rather than Monte Carlo samples.

7.6.1 The Actor-Critic Architecture

- **Actor**: The policy $\pi_\theta(a|s)$, updated using policy gradients
- **Critic**: The value function $V_{\mathbf{w}}(s)$ or $Q_{\mathbf{w}}(s,a)$, learned using TD methods

The critic provides variance reduction by:

- Bootstrapping from current estimates
- Providing immediate feedback rather than waiting for episode completion
- Reducing the dependency on complete trajectory returns

7.6.2 One-Step Actor-Critic

Using the one-step TD error as an estimate of the advantage: $\delta_t = R_{t+1} + \gamma V_{\mathbf{w}}(S_{t+1}) - V_{\mathbf{w}}(S_t)$

Theorem 7.4 (TD Error as Advantage Estimate)**.** *The TD error is an unbiased estimate of the advantage function:* $\mathbb{E}_{\pi_\theta}[\delta_t | S_t = s, A_t = a] = A^{\pi_\theta}(s, a)$

Algorithm 17 One-Step Actor-Critic

```
Initialize policy parameters $\theta$ and value function parameters $\mathbf{w}$
for each episode do
  Initialize $S$
  repeat
    Sample $A \sim \pi_\theta(\cdot|S)$
    Take action $A$, observe $R, S'$
    $\delta \leftarrow R + \gamma V_{\mathbf{w}}(S') - V_{\mathbf{w}}(S)$
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha_w \delta \nabla_{\mathbf{w}} V_{\mathbf{w}}(S)$
    $\theta \leftarrow \theta + \alpha_\theta \delta \nabla_\theta \log \pi_\theta(A|S)$
    $S \leftarrow S'$
  until $S$ is terminal
end for
```

7.6.3 Advantage Actor-Critic (A2C)

A2C explicitly learns both state and action-value functions to compute advantages: $A(s, a) = Q_{\mathbf{w}}(s, a) - V_{\mathbf{v}}(s)$

Alternatively, we can use n-step returns or TD(λ) to estimate advantages with different bias-variance trade-offs.

7.6.4 Asynchronous Advantage Actor-Critic (A3C)

A3C scales actor-critic methods by running multiple agents in parallel:

- Multiple workers explore different parts of the environment
- Asynchronous updates reduce correlation in training data
- Achieves faster training and better exploration

7.7 Natural Policy Gradients

7.7.1 The Problem with Vanilla Policy Gradients

Standard policy gradients can be sensitive to policy parameterization. The same policy can be represented with different parameters, leading to different gradients for the same policy improvement.

7.7.2 Natural Gradient

The natural policy gradient addresses this issue by using the Fisher information matrix:

Definition 7.5 (Fisher Information Matrix). $G_\theta = \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(a|s) \nabla_\theta \log \pi_\theta(a|s)^\top]$

Definition 7.6 (Natural Policy Gradient). $\nabla_\theta^{nat} J(\theta) = G_\theta^{-1} \nabla_\theta J(\theta)$

The natural gradient finds the steepest ascent direction in the space of policy distributions, making it invariant to policy parameterization.

7.7.3 Natural Actor-Critic

For actor-critic methods with compatible function approximation: $\nabla_{\mathbf{w}} A_{\mathbf{w}}(s,a) = \nabla_\theta \log \pi_\theta(a|s)$

The natural policy gradient simplifies to: $\nabla_\theta^{nat} J(\theta) = \mathbf{w}$

This means we can update the actor parameters directly using the critic parameters: $\theta \leftarrow \theta + \alpha \mathbf{w}$

7.8 Trust Region Methods

7.8.1 Motivation

Large policy updates can be catastrophic, leading to:

- Poor performance due to bad policy updates
- Difficulty recovering from bad updates
- Training instability

Trust region methods constrain policy updates to ensure improvement.

7.8.2 Trust Region Policy Optimization (TRPO)

TRPO solves the constrained optimization problem: $\max_\theta \mathbb{E}_{s,a\sim\pi_{\theta_{old}}}\left[\frac{\pi_\theta(a|s)}{\pi_{\theta_{old}}(a|s)} A^{\pi_{\theta_{old}}}(s,a)\right]$

subject to: $\mathbb{E}_{s\sim\pi_{\theta_{old}}}[D_{KL}(\pi_{\theta_{old}}(\cdot|s) \| \pi_\theta(\cdot|s))] \le \delta$

This ensures that the new policy doesn't deviate too far from the old policy in terms of KL divergence.

7.8.3 Proximal Policy Optimization (PPO)

PPO simplifies TRPO by using a clipped objective: $L^{CLIP}(\theta) = \mathbb{E}_t\left[\min(r_t(\theta)A_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)A_t)\right]$

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

The clipping prevents large policy updates while being simpler to implement than TRPO.

7.9 Continuous Control

7.9.1 Deterministic Policy Gradients

For continuous action spaces, we can consider deterministic policies: $a = \mu_\theta(s)$

Theorem 7.7 (Deterministic Policy Gradient). $\nabla_\theta J(\theta) = \mathbb{E}_{s\sim\rho^\mu}[\nabla_\theta \mu_\theta(s) \nabla_a Q^\mu(s,a)|_{a=\mu_\theta(s)}]$

This requires the gradient of the action-value function with respect to actions, making it suitable for continuous control problems.

7.9.2 Deep Deterministic Policy Gradient (DDPG)

DDPG combines deterministic policy gradients with deep neural networks:

- Actor network: $\mu_\theta(s)$
- Critic network: $Q_\phi(s, a)$
- Target networks for stability
- Experience replay for sample efficiency

7.10 Summary

Policy gradient methods provide a powerful framework for reinforcement learning:

- Direct policy optimization often has better convergence properties
- Natural for continuous action spaces and stochastic policies
- Can be combined with value functions in actor-critic architectures
- Trust region methods ensure stable learning
- Enable end-to-end learning in complex domains

The choice between value-based and policy-based methods depends on the specific characteristics of the problem, including the action space, the need for stochastic policies, and the desired convergence properties.

Chapter 8

Integrating Learning and Planning

Real-world reinforcement learning often benefits from combining learning (acquiring knowledge from experience) with planning (using models to reason about future actions). This chapter explores how to integrate these complementary approaches to create more efficient and effective RL systems.

8.1 Introduction

8.1.1 Learning vs. Planning Revisited

Throughout this book, we have explored two fundamental paradigms:

Model-Free Learning:

- Learn value functions or policies directly from experience
- No explicit model of the environment
- Robust to model errors (no model to be wrong)
- Can be sample inefficient

Model-Based Planning:

- Use a model of the environment to compute optimal policies
- Requires knowledge or learning of transition dynamics and rewards
- Can be very sample efficient when the model is accurate
- Performance limited by model quality

The integration of learning and planning combines the robustness of model-free methods with the efficiency of model-based approaches.

8.2 Model-Based Reinforcement Learning

8.2.1 Learning Environment Models

When the environment is unknown, we must learn a model from experience.

Definition 8.1 (Environment Model)**.** *A model $M = \langle P_\eta, R_\eta \rangle$ represents:*

- *State transitions:* $S_{t+1} \sim P_\eta(S_{t+1}|S_t, A_t)$
- *Rewards:* $R_{t+1} = R_\eta(R_{t+1}|S_t, A_t)$

parameterized by η.

Typically, we assume conditional independence: $P[S_{t+1}, R_{t+1}|S_t, A_t] = P[S_{t+1}|S_t, A_t]P[R_{t+1}|S_t, A_t]$

8.2.2 Model Learning as Supervised Learning

Model learning can be framed as supervised learning:

- **Input**: State-action pairs (s, a)
- **Output**: Next state and reward (s', r)

The training data consists of experience tuples: $\{(S_1, A_1, R_2, S_2), (S_2, A_2, R_3, S_3), \ldots, (S_{T-1}, A_{T-1}, R_T, S_T)\}$
We can use any supervised learning method:

- Linear regression for rewards
- Gaussian processes for continuous states
- Neural networks for complex dynamics
- Table lookup for discrete states

8.2.3 Table Lookup Models

For discrete state and action spaces, the simplest approach is table lookup:
$\hat{P}^a_{ss'} = \frac{1}{N(s,a)} \sum_{t=1}^T \mathbf{1}(S_t = s, A_t = a, S_{t+1} = s')$
$\hat{R}^a_s = \frac{1}{N(s,a)} \sum_{t=1}^T \mathbf{1}(S_t = s, A_t = a)R_{t+1}$
where $N(s, a)$ is the number of times action a was taken in state s.

8.2.4 Planning with Learned Models

Once we have a model, we can use any planning algorithm:

- Value iteration
- Policy iteration
- Tree search algorithms
- Linear programming (for specific problem classes)

The key insight is that we can treat the learned model as if it were the true environment for planning purposes.

8.2.5 Sample-Based Planning

Rather than solving the model analytically, we can use the model to generate synthetic experience:
This approach allows us to apply any model-free algorithm to simulated experience.

Algorithm 18 Sample-Based Planning

```
Learn model $M_\eta$ from real experience
repeat
  Sample starting state $S \sim \rho$
  Sample action $A \sim$ policy
  Sample transition: $S' \sim P_\eta(S'|S, A)$, $R = R_\eta(R|S, A)$
  Apply model-free RL update using $(S, A, R, S')$
until convergence
```

8.3 Integrated Architectures

8.3.1 The Dyna Architecture

Dyna integrates learning and planning by maintaining both real and simulated experience streams.

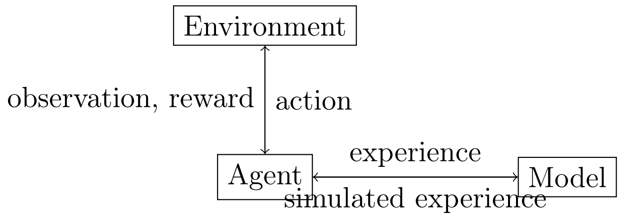

Figure 8.1: Dyna Architecture

Dyna-Q Algorithm

Algorithm 19 Dyna-Q

```
Initialize $Q(s, a)$ and $Model(s, a)$ for all $s, a$
repeat
  (a) Acting and Learning:
  Take action $A$ in state $S$, observe $R, S'$
  $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma \max_{a'} Q(S', a') - Q(S, A)]$
  $Model(S, A) \leftarrow R, S'$ (store experience)

  (b) Planning:
  for $n$ planning steps do
    Sample random previously observed state-action pair $S, A$
    $R, S' \leftarrow Model(S, A)$ (retrieve stored experience)
    $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma \max_{a'} Q(S', a') - Q(S, A)]$
  end for
until convergence
```

8.3.2 Benefits of Dyna

- **Sample efficiency**: Each real experience is used multiple times

- **Faster learning**: Planning accelerates convergence
- **Robustness**: Model-free updates handle model errors
- **Flexibility**: Can adjust the balance between learning and planning

8.3.3 Handling Model Errors

When the environment changes, the model becomes inaccurate. Dyna can handle this through:

- **Continued real experience**: Model-free updates eventually correct errors
- **Exploration bonuses**: Encourage exploration of potentially changed areas
- **Model confidence**: Weight planning updates by model certainty

8.4 Simulation-Based Search

8.4.1 Forward Search

Rather than learning a global value function, forward search focuses computation on the current state by building a search tree rooted at the current state.

- **Selective**: Only explores relevant states
- **Adaptive**: Computation adapts to the current situation
- **Anytime**: Can return the best action found so far at any time

8.4.2 Monte Carlo Tree Search (MCTS)

MCTS combines Monte Carlo evaluation with tree search:

Algorithm 20 MCTS

```
Initialize search tree with root state $s_0$
for simulation = 1 to $N$ do
    (1) Selection: Traverse tree using tree policy to reach leaf
    (2) Expansion: Add new child node(s) to the tree
    (3) Simulation: Simulate from new node using default policy
    (4) Backup: Update value estimates along the path
end for
Return action with highest estimated value
```

Upper Confidence Bound for Trees (UCT)

UCT uses the UCB1 algorithm for action selection in the tree:

$$a^* = \text{argmax}_a \left[Q(s,a) + c\sqrt{\frac{\ln N(s)}{N(s,a)}} \right]$$

where:

- $Q(s,a)$ is the average return from action a in state s

- $N(s, a)$ is the number of times action a was tried from state s
- $N(s)$ is the number of times state s was visited
- c is an exploration constant

8.4.3 Properties of MCTS

Theorem 8.2 (MCTS Convergence)**.** *Under mild conditions, MCTS converges to the optimal policy as the number of simulations approaches infinity.*

Advantages of MCTS:

- Works with any simulator (black box)
- Handles large branching factors
- Anytime algorithm
- Parallelizable
- No need for domain-specific evaluation functions

8.4.4 MCTS in Practice: Computer Go

MCTS revolutionized computer Go:

- Go has branching factor ≈ 250 and game length ≈ 150
- Traditional minimax search was ineffective
- MCTS enabled programs to reach professional level
- Combined with deep learning in AlphaGo for superhuman performance

8.5 Temporal Difference Search

8.5.1 TD Search vs. MC Search

While MCTS uses Monte Carlo evaluation, we can also use temporal difference learning within search trees:

Monte Carlo Search	Temporal Difference Search
Simulate to terminal states	Bootstrap from value estimates
Unbiased but high variance	Biased but lower variance
No bootstrapping	Bootstraps from current estimates

8.5.2 TD Search Algorithm

Algorithm 21 TD Search

```
Initialize value function $V(s)$ for all states in search tree
for simulation = 1 to $N$ do
    Generate trajectory: $s_0, a_0, r_1, s_1, a_1, r_2, \ldots, s_k$
    for each transition $(s_t, a_t, r_{t+1}, s_{t+1})$ in trajectory do
        $V(s_t) \leftarrow V(s_t) + \alpha[r_{t+1} + \gamma V(s_{t+1}) - V(s_t)]$
    end for
end for
Return action $a^* = \text{argmax}_a V(\text{succ}(s_0, a))$
```

8.5.3 Advantages of TD Search

- **Lower variance**: Bootstrapping reduces variance of value estimates
- **Faster convergence**: Often converges with fewer simulations than MC
- **Online updates**: Can update values immediately during search
- **Function approximation**: Natural integration with learned value functions

8.6 Dyna-2: Integrating Short and Long-Term Memory

8.6.1 Motivation

Different types of knowledge require different learning approaches:

- **General knowledge**: Learned slowly from many experiences
- **Local knowledge**: Learned quickly for current situation

8.6.2 The Dyna-2 Architecture

Dyna-2 maintains two separate memory systems:

- **Long-term memory**: General value function $V^{LT}(s)$ learned from real experience
- **Short-term memory**: Local value function $V^{ST}(s)$ learned from simulated experience

The overall value estimate combines both: $V(s) = V^{LT}(s) + V^{ST}(s)$

8.6.3 Dyna-2 Algorithm

8.7 Model Uncertainty

8.7.1 The Challenge of Model Errors

Model-based methods face a fundamental challenge: models are typically imperfect, and planning with wrong models can lead to suboptimal or even catastrophic behavior.

Algorithm 22 Dyna-2

```
Initialize long-term and short-term value functions
repeat
    Real Experience:
    Take action, observe transition $(s, a, r, s')$
    Update long-term memory: $V^{LT}$ using TD learning
    Update model

    Simulated Experience:
    Run search from current state using current model
    Update short-term memory: $V^{ST}$ using TD search

    Action Selection:
    Choose action based on $V^{LT} + V^{ST}$
until episode ends
Reset short-term memory
```

8.7.2 Sources of Model Uncertainty

- **Limited data**: Insufficient experience in some state-action pairs
- **Function approximation error**: Model cannot represent true dynamics perfectly
- **Stochastic environments**: Inherent randomness in transitions
- **Non-stationary environments**: Environment changes over time

8.7.3 Robust Planning

Several approaches handle model uncertainty:

Conservative Planning

Use pessimistic estimates when uncertain:

- Lower confidence bounds on rewards
- Upper confidence bounds on costs
- Worst-case scenario planning

Risk-Sensitive Planning

Incorporate variance or risk measures into planning objectives: $\max_\pi \mathbb{E}[G] - \beta \text{Var}[G]$

Distributional Models

Maintain distributions over possible models rather than point estimates, enabling:

- Uncertainty quantification
- Risk-aware planning
- Exploration bonuses for uncertain regions

8.8 Summary

Integrating learning and planning offers the best of both worlds:

- **Sample efficiency**: Models allow reuse of experience through simulation
- **Robustness**: Model-free updates correct for model errors
- **Flexibility**: Can adjust the balance based on model quality and computational budget
- **Scalability**: Search-based methods focus computation where needed

The key insights are:

- Models need not be perfect to be useful
- Combining multiple sources of information improves performance
- Search methods can make effective use of imperfect models
- Different types of knowledge benefit from different learning approaches

Modern deep reinforcement learning systems often integrate these ideas, using learned models for planning while maintaining model-free policy learning for robustness.

Chapter 9

Exploration and Exploitation

The exploration-exploitation dilemma represents one of the most fundamental challenges in reinforcement learning. An agent must balance between exploiting its current knowledge to maximize immediate reward and exploring to gain information that could lead to better long-term performance.

9.1 The Exploration-Exploitation Dilemma

9.1.1 Fundamental Trade-off

Every learning agent faces the choice between:

- **Exploitation**: Using current knowledge to maximize expected reward
- **Exploration**: Gathering new information that might lead to better decisions in the future

This trade-off appears across many domains:

- Restaurant selection: familiar favorites vs. trying new places
- Medical treatment: proven therapies vs. experimental treatments
- Investment: established assets vs. emerging opportunities
- Online advertising: successful ads vs. testing new variants

9.1.2 Why Exploration Matters

Pure exploitation can lead to:

- **Premature convergence**: Settling on suboptimal actions
- **Missing opportunities**: Failing to discover better alternatives
- **Inability to adapt**: Poor response to environmental changes

Pure exploration leads to:

- **Low performance**: Constantly trying poor actions
- **Inefficiency**: Not using acquired knowledge
- **Opportunity cost**: Forgoing known good rewards

9.2 Multi-Armed Bandits

The multi-armed bandit problem provides a clean framework for studying exploration-exploitation trade-offs without the complexity of sequential decision making.

9.2.1 Problem Formulation

Definition 9.1 (Multi-Armed Bandit). *A multi-armed bandit is a tuple $\langle \mathcal{A}, \mathcal{R} \rangle$ where:*

- *$\mathcal{A}$ is a known set of m actions (arms)*
- *$\mathcal{R} = \{R_a(r) = P[r|a] : a \in \mathcal{A}\}$ are unknown reward distributions*

At each time step t:

1. Agent selects action $A_t \in \mathcal{A}$
2. Environment generates reward $R_t \sim \mathcal{R}_{A_t}$
3. Goal: Maximize cumulative reward $\sum_{t=1}^{T} R_t$

9.2.2 Regret

Definition 9.2 (Action Value). *The value of action a is its expected reward: $q(a) = \mathbb{E}[R_t | A_t = a]$*

Definition 9.3 (Optimal Value). $V^* = \max_{a \in \mathcal{A}} q(a)$

Definition 9.4 (Regret). *The regret at time t is the opportunity loss: $\ell_t = V^* - q(A_t)$*
The total regret after T steps is: $L_T = \sum_{t=1}^{T} \ell_t = \sum_{t=1}^{T} (V^* - q(A_t))$

Minimizing regret is equivalent to maximizing cumulative reward.

9.2.3 Gap-Dependent Regret

Define the *gap* $\Delta_a = V^* - q(a)$ for each action a. If $N_T(a)$ is the number of times action a is selected in T steps:
$L_T = \sum_{a \in \mathcal{A}} \mathbb{E}[N_T(a)] \Delta_a$
Good algorithms ensure small counts for actions with large gaps.

9.3 Bandit Algorithms

9.3.1 Greedy Algorithm

The greedy algorithm always selects the action with the highest estimated value:
Problem: Can get stuck on suboptimal actions forever, leading to linear regret.

9.3.2 ϵ-Greedy Algorithm

ϵ-greedy adds random exploration:
Problem: Constant ϵ leads to linear regret because it never stops exploring.

Algorithm 23 Greedy Bandit

```
Initialize $\hat{q}(a) = 0$, $N(a) = 0$ for all $a$
for $t = 1$ to $T$ do
    $A_t = \text{argmax}_a \hat{q}(a)$
    Take action $A_t$, observe reward $R_t$
    $N(A_t) \leftarrow N(A_t) + 1$
    $\hat{q}(A_t) \leftarrow \hat{q}(A_t) + \frac{1}{N(A_t)}(R_t - \hat{q}(A_t))$
end for
```

Algorithm 24 ϵ-Greedy Bandit

```
Initialize $\hat{q}(a) = 0$, $N(a) = 0$ for all $a$
for $t = 1$ to $T$ do
    With probability $1 - \epsilon$: $A_t = \text{argmax}_a \hat{q}(a)$
    With probability $\epsilon$: $A_t =$ uniform random action
    Take action $A_t$, observe reward $R_t$
    Update $\hat{q}(A_t)$ using incremental average
end for
```

9.3.3 Decaying ϵ-Greedy

Use a decreasing exploration rate ϵ_t:

Theorem 9.5 (Decaying ϵ-Greedy Regret). *If $\epsilon_t = \min\{1, \frac{c|\mathcal{A}|}{d^2 t}\}$ where $c > 0$ and $d = \min_{a:\Delta_a > 0} \Delta_a$, then decaying ϵ-greedy achieves logarithmic regret.*

Problem: Requires knowledge of the gaps, which are unknown.

9.4 Upper Confidence Bound Algorithms

9.4.1 Optimism in the Face of Uncertainty

The key insight is to be optimistic about uncertain actions:

- Actions with high uncertainty could be optimal
- Trying them resolves uncertainty
- If they're good, we exploit them; if not, we've learned something valuable

9.4.2 Upper Confidence Bounds

Definition 9.6 (Upper Confidence Bound). *For each action a, compute an upper confidence bound UCB(a) such that: $q(a) \leq \hat{q}(a) + UCB(a)$ with high probability.*

Select the action with the highest upper confidence bound: $A_t = \text{argmax}_a[\hat{q}(a) + \text{UCB}(a)]$

9.4.3 Hoeffding's Inequality

Theorem 9.7 (Hoeffding's Inequality). *Let $X_1, \ldots, X_t$ be i.i.d. random variables in $[0, 1]$ with sample mean $\bar{X}_t$. Then: $P[\mathbb{E}[X] > \bar{X}_t + u] \leq e^{-2tu^2}$*

Applying this to bandit rewards: if we want $P[q(a) > \hat{q}_t(a) + u] \leq p$, solve: $e^{-2N_t(a)u^2} = p$ $u = \sqrt{\frac{-\ln p}{2N_t(a)}}$

9.4.4 UCB1 Algorithm

Setting $p = t^{-4}$ (so confidence increases with time):

Algorithm 25 UCB1

```
Initialize $\hat{q}(a) = 0$, $N(a) = 0$ for all $a$
for $t = 1$ to $T$ do
    $A_t = \text{argmax}_a \left[\hat{q}(a) + \sqrt{\frac{2\ln t}{N(a)}}\right]$
    Take action $A_t$, observe reward $R_t$
    Update $\hat{q}(A_t)$ and $N(A_t)$
end for
```

Theorem 9.8 (UCB1 Regret Bound). *The regret of UCB1 is bounded by:* $L_T \leq 8 \ln T \sum_{a:\Delta_a>0} \frac{1}{\Delta_a}$

This achieves logarithmic regret without knowing the gaps.

9.4.5 Lower Bounds

Theorem 9.9 (Lai-Robbins Lower Bound). *For any consistent algorithm, the regret is lower bounded by:* $\liminf_{T\to\infty} \frac{L_T}{\ln T} \geq \sum_{a:\Delta_a>0} \frac{\Delta_a}{KL(\mathcal{R}_a \| \mathcal{R}^*)}$

UCB1 achieves logarithmic regret but is not optimal in the constant.

9.5 Bayesian Bandits

9.5.1 Prior Knowledge

Bayesian methods incorporate prior beliefs about the reward distributions and update these beliefs as data is observed.

Definition 9.10 (Bayesian Bandit).

- *Prior distribution over reward parameters:* $p(\theta)$
- *Likelihood of observations:* $p(r|\theta, a)$
- *Posterior after observing history* h_t*:* $p(\theta|h_t) \propto p(\theta) \prod_{s=1}^{t} p(r_s|\theta, a_s)$

9.5.2 Thompson Sampling

Thompson sampling implements *probability matching*:

Definition 9.11 (Probability Matching). *Select action* a *with probability equal to the probability that* a *is optimal:* $\pi(a|h_t) = P[q(a) > q(a'), \forall a' \neq a|h_t]$

Theorem 9.12 (Thompson Sampling Optimality). *For many bandit problems, Thompson sampling achieves the Lai-Robbins lower bound.*

Algorithm 26 Thompson Sampling

```
Initialize prior $p(\theta)$
for $t = 1$ to $T$ do
    Sample $\tilde{\theta} \sim p(\theta|h_{t-1})$ from posterior
    $A_t = \text{argmax}_a \mathbb{E}[r|\tilde{\theta}, a]$
    Take action $A_t$, observe reward $R_t$
    Update posterior: $p(\theta|h_t) \propto p(\theta|h_{t-1})p(R_t|\theta, A_t)$
end for
```

9.5.3 Example: Beta-Bernoulli Bandits

For Bernoulli rewards with Beta priors:

- Prior: $\theta_a \sim \text{Beta}(\alpha_a, \beta_a)$
- After observing s successes and f failures: $\theta_a|\text{data} \sim \text{Beta}(\alpha_a + s, \beta_a + f)$

Thompson sampling reduces to:

1. Sample $\tilde{\theta}_a \sim \text{Beta}(\alpha_a + s_a, \beta_a + f_a)$ for each arm a
2. Select $A_t = \text{argmax}_a \tilde{\theta}_a$

9.6 Contextual Bandits

9.6.1 Problem Definition

Contextual bandits extend multi-armed bandits by including context (state) information:

Definition 9.13 (Contextual Bandit)**.** *At each time step:*

1. *Environment reveals context s_t*
2. *Agent selects action a_t*
3. *Agent receives reward $r_t \sim \mathcal{R}(r|s_t, a_t)$*

The goal is to learn a policy $\pi(a|s)$ that maximizes expected reward.

9.6.2 Linear Contextual Bandits

Assume the expected reward is linear in features: $\mathbb{E}[r|s, a] = \phi(s, a)^\top \theta^*$
where $\phi(s, a)$ is a feature vector and θ^* is unknown.

9.6.3 LinUCB Algorithm

Maintain confidence ellipsoids around parameter estimates:
The confidence width $\sqrt{\phi(s, a)^\top A^{-1} \phi(s, a)}$ captures uncertainty due to limited data.

Algorithm 27 LinUCB

```
Initialize $A = I$, $b = 0$
for $t = 1$ to $T$ do
    Observe context $s_t$
    $\hat{\theta}_t = A^{-1}b$
    $a_t = \text{argmax}_a \left[\phi(s_t, a)^\top \hat{\theta}_t + \alpha\sqrt{\phi(s_t, a)^\top A^{-1}\phi(s_t, a)}\right]$
    Take action $a_t$, observe reward $r_t$
    $A \leftarrow A + \phi(s_t, a_t)\phi(s_t, a_t)^\top$
    $b \leftarrow b + \phi(s_t, a_t)r_t$
end for
```

9.7 Exploration in MDPs

9.7.1 Challenges in Sequential Decision Making

MDPs introduce additional complexity:

- Actions affect future states and opportunities
- Need to explore state-action pairs, not just actions
- Exploration in one state affects learning in other states
- Must balance immediate and long-term exploration benefits

9.7.2 Optimistic Initialization

Initialize value functions optimistically:

- $Q(s, a) = \frac{R_{\max}}{1-\gamma}$ (maximum possible value)
- Encourages exploration of unvisited state-action pairs
- Often combined with ϵ-greedy or other exploration strategies

9.7.3 UCB for MDPs

Extend confidence bounds to action-values: $Q(s, a) + C(s, a)$
where $C(s, a)$ is a confidence bonus based on visit counts. Challenges:

- Visit counts for (s, a) pairs can be sparse
- Need to account for uncertainty in value function updates
- Optimal exploration depends on the entire MDP structure

9.7.4 Thompson Sampling for MDPs

Maintain posterior distributions over:

- Transition probabilities: $P(s'|s, a)$
- Reward functions: $R(s, a)$

At each episode:

1. Sample MDP from posterior
2. Solve sampled MDP to get policy
3. Follow policy for the episode
4. Update posterior with observed data

9.7.5 Information-Directed Sampling

Balance expected reward and information gain: $\max_\pi \mathbb{E}[\text{reward}] - \lambda \cdot \text{information gain}$

This approach explicitly trades off exploitation and exploration based on the value of information.

9.8 Count-Based Exploration

9.8.1 Exploration Bonuses

Add bonuses to encourage visiting less-explored states: $Q^+(s,a) = Q(s,a) + \frac{C}{\sqrt{N(s,a)}}$

where $N(s,a)$ is the visit count and C is a exploration constant.

9.8.2 Pseudo-Counts for Complex State Spaces

For large or continuous state spaces, direct counting is impractical. Pseudo-count methods estimate novelty:

- Use density models to estimate state visitation
- Assign higher bonuses to states that appear rare under the model
- Update the model as new states are encountered

9.8.3 Curiosity-Driven Exploration

Reward the agent for encountering surprising or novel experiences:

- **Prediction error**: Reward states where predictions are wrong
- **Information gain**: Reward states that provide most learning
- **Compression progress**: Reward improvements in state representation

9.9 Deep Exploration

9.9.1 The Challenge of Deep Exploration

Simple exploration strategies like ϵ-greedy may be insufficient for environments requiring *deep exploration*—coordinated sequences of actions to reach informative states.

9.9.2 UCB for Deep RL

Extend UCB to neural network function approximation:

- Maintain uncertainty estimates over network parameters
- Use Bayesian neural networks or ensemble methods
- Add exploration bonuses based on parameter uncertainty

9.9.3 Noisy Networks

Add noise directly to network parameters: $y = (W + \sigma \odot \epsilon)x + b$

where ϵ is random noise and σ are learned noise parameters. This provides state-dependent exploration that can lead to consistent strategies.

9.10 Summary

Exploration-exploitation trade-offs are fundamental to effective reinforcement learning:

- **Multi-armed bandits** provide a clean framework for studying exploration without sequential complications
- **Upper confidence bound methods** provide principled approaches with strong theoretical guarantees
- **Thompson sampling** offers Bayesian approach that often performs well in practice
- **Contextual bandits** bridge simple bandits and full RL problems
- **MDP exploration** requires handling sequential decision making and sparse rewards
- **Deep exploration** becomes crucial in complex environments requiring coordinated action sequences

The choice of exploration strategy depends on:

- Problem structure (bandits vs. MDPs)
- Available prior knowledge
- Computational constraints
- Required performance guarantees

Effective exploration remains an active area of research, especially for complex environments with large state spaces, sparse rewards, and long horizons.

Chapter 10

Reinforcement Learning in Classic Games

Games have served as important testbeds for artificial intelligence throughout the field's history. They provide well-defined environments with clear objectives, making them ideal for developing and evaluating reinforcement learning algorithms. This chapter examines how RL has been successfully applied to classic games, revealing insights applicable to broader domains.

10.1 Why Study Games?

10.1.1 Games as AI Testbeds

Games offer several advantages as research domains:

- **Clear objectives**: Well-defined winning conditions and scoring systems
- **Controlled environments**: Rules are precise and unchanging
- **Measurable progress**: Easy to evaluate performance against human experts
- **Increasing complexity**: Range from simple to extremely complex scenarios
- **Rich strategy**: Require long-term planning and strategic thinking

10.1.2 Historical Significance

Games have been central to AI development:

- **Turing's chess program (1950)**: One of the first AI programs
- **Arthur Samuel's checkers player (1959)**: Pioneer of machine learning
- **Deep Blue vs. Kasparov (1997)**: Landmark achievement in AI
- **AlphaGo vs. Lee Sedol (2016)**: Breakthrough in deep reinforcement learning

10.2 Game-Theoretic Foundations

10.2.1 Two-Player Zero-Sum Games

Most classic board games are two-player zero-sum games where:

- Two players alternate moves
- One player's gain equals the other's loss: $R_1 + R_2 = 0$
- Perfect information: Complete game state is observable

10.2.2 Nash Equilibrium in Games

Definition 10.1 (Nash Equilibrium). *A strategy profile* (π_1^*, π_2^*) *is a Nash equilibrium if:*

- π_1^* *is a best response to* π_2^*
- π_2^* *is a best response to* π_1^*

Neither player can unilaterally improve by changing strategies.

10.2.3 Minimax Value

For two-player zero-sum games, the minimax value represents the game's outcome under optimal play:

Definition 10.2 (Minimax Value). $v^*(s) = \max_{\pi_1} \min_{\pi_2} v^{\pi_1,\pi_2}(s)$

This value can be computed through minimax search or learned through self-play.

10.3 Classical Game Tree Search

10.3.1 Minimax Algorithm

The minimax algorithm computes optimal play by recursively evaluating positions:

10.3.2 Alpha-Beta Pruning

Alpha-beta pruning reduces the search space by eliminating branches that cannot affect the final result:

- α: Best value maximizing player can guarantee
- β: Best value minimizing player can guarantee
- Prune when $\alpha \geq \beta$

This can reduce search complexity from $O(b^d)$ to $O(b^{d/2})$ in the best case.

Algorithm 28 Minimax

```
function Minimax(state, depth, maximizing_player)
if depth = 0 OR game_over(state) then
  return evaluate(state)
end if
if maximizing_player then
  value ← $-\infty$
  for each child of state do
    value ← max(value, Minimax(child, depth-1, False))
  end for
  return value
else
  value ← $+\infty$
  for each child of state do
    value ← min(value, Minimax(child, depth-1, True))
  end for
  return value
end if
end function
```

10.3.3 Evaluation Functions

For games too complex to search to completion, evaluation functions estimate position values:

- **Material balance**: Piece values in chess
- **Positional factors**: King safety, pawn structure, piece activity
- **Pattern recognition**: Known good/bad configurations
- **Machine learning**: Learned evaluation functions

10.4 Reinforcement Learning Approaches

10.4.1 Self-Play Learning

Self-play enables RL agents to improve without human supervision:

Algorithm 29 Self-Play Learning

```
Initialize policy $\pi_0$
for iteration $k = 1, 2, \ldots$ do
  Generate games by playing $\pi_{k-1}$ against itself
  Collect training data: positions, moves, outcomes
  Update policy: $\pi_k \leftarrow$ improve($\pi_{k-1}$, data)
end for
```

10.4.2 Temporal Difference Learning for Games

TD learning updates value estimates using successive positions:
$V(s_t) \leftarrow V(s_t) + \alpha[V(s_{t+1}) - V(s_t)]$
At game end, use the actual outcome (win/loss/draw) as the final target.

10.4.3 Afterstate Representation

For deterministic games, it's often more efficient to evaluate *afterstates*—positions resulting from actions:

- $q(s, a) = v(\text{afterstate}(s, a))$
- Reduces the action space evaluation problem
- Natural for turn-based games

10.5 Success Stories

10.5.1 TD-Gammon: Backgammon

TD-Gammon demonstrated the power of self-play temporal difference learning:
Architecture:

- Neural network with raw board position input
- Output: Probability of winning from current position
- No game-specific knowledge beyond rules

Training:

- Self-play using TD(0)
- No exploration—purely greedy play
- 1.5 million training games

Results:

- Achieved superhuman level
- Discovered new opening strategies
- Changed human understanding of backgammon strategy

Key insight: The stochastic nature of backgammon (dice rolls) provided natural exploration, enabling successful convergence without explicit exploration.

10.5.2 Deep Blue: Chess

While not primarily an RL system, Deep Blue incorporated learning elements:

Architecture:

- Specialized chess chips for rapid position evaluation
- 480 special-purpose processors
- Searched 200 million positions per second

Evaluation Function:

- 8000+ hand-crafted features
- Weights tuned through analysis of grandmaster games
- Some machine learning for opening book and endgame databases

Achievement:

- Defeated world champion Garry Kasparov (1997)
- Demonstrated the power of computational search

10.5.3 AlphaGo: The Game of Go

AlphaGo combined deep learning with tree search:

Components:

- **Policy network**: Predict human expert moves
- **Value network**: Evaluate board positions
- **Monte Carlo Tree Search**: Guided by neural networks

Training Process:

1. Pre-train policy network on human expert games
2. Improve policy through self-play reinforcement learning
3. Train value network on self-play positions
4. Combine networks with MCTS for play

Breakthrough Results:

- First AI to defeat professional Go player (2015)
- Defeated world champion Lee Sedol 4-1 (2016)
- Retired undefeated in competition play

10.5.4 AlphaZero: Generalized Game Learning

AlphaZero generalized the AlphaGo approach to multiple games:

Key Innovations:

- No human game data—purely self-play learning
- Single neural network for both policy and value estimation
- Generalized to chess, shogi, and Go with identical algorithm

Results:

- Superhuman performance in all three games
- Learned distinctive playing styles for each game
- Rediscovered and transcended centuries of human strategy

10.6 Combining Learning and Search

10.6.1 Learning Value Functions for Search

Several approaches combine learned evaluation with tree search:

TD-Root

Update value function using search results: $V(s_t) \leftarrow V(s_t) + \alpha[V_{\text{search}}(s_{t+1}) - V(s_t)]$

TD-Leaf

Update leaf positions reached by search: $V(\text{leaf}) \leftarrow V(\text{leaf}) + \alpha[V_{\text{search}}(s_{t+1}) - V_{\text{search}}(s_t)]$

TreeStrap

Update all positions in search tree: $V(s) \leftarrow V(s) + \alpha[V_{\text{search}}(s) - V(s)]$

10.6.2 Monte Carlo Tree Search

MCTS provides a unified framework for learning and planning:

Algorithm 30 MCTS Four Phases

```
  Selection: Traverse tree using UCB to balance exploration/exploitation
  Expansion: Add new node(s) to the tree
  Simulation: Estimate value through rollout or neural network
  Backup: Propagate value estimates back through tree
```

Advantages:

- Anytime algorithm—can return best move found so far
- Naturally handles large branching factors

- Requires minimal domain knowledge
- Easily parallelizable
- Combines exploration and exploitation principally

10.7 Dealing with Imperfect Information

10.7.1 Information Sets

In games with hidden information (like poker), players have *information sets*—collections of game states that are indistinguishable to the player.

10.7.2 Counterfactual Regret Minimization

CFR is a powerful algorithm for imperfect information games:

- Iteratively improves strategy by minimizing regret
- Handles information sets naturally
- Converges to Nash equilibrium in zero-sum games
- Successfully applied to poker variants

10.7.3 Neural Fictitious Self-Play

Extends self-play to imperfect information games:

- Maintains average strategy over training
- Uses neural networks for strategy representation
- Achieved superhuman performance in poker

10.8 Key Lessons from Game AI

10.8.1 The Power of Self-Play

Self-play has emerged as a powerful paradigm:

- Removes dependence on human expert data
- Enables discovery of novel strategies
- Scales naturally with computational resources
- Works across diverse game domains

10.8.2 Combining Representation Learning and Planning

The most successful systems combine:

- Deep neural networks for pattern recognition
- Tree search for tactical calculation
- Reinforcement learning for strategy improvement

10.8.3 Domain-Specific vs. General Approaches

Evolution from specialized to general:

- **1990s**: Hand-crafted evaluation functions
- **2000s**: Learning with domain knowledge
- **2010s**: End-to-end learning with minimal domain knowledge
- **Future**: Transfer learning across multiple domains

10.8.4 Computational Scaling

Game AI has benefited enormously from computational advances:

- Faster hardware enables deeper search
- Parallel computation accelerates self-play training
- GPU acceleration makes deep learning practical
- Cloud computing democratizes access to computation

10.9 Beyond Classical Games

10.9.1 Real-Time Strategy Games

Games like StarCraft present additional challenges:

- Partial observability
- Real-time decision making
- Large action spaces
- Multi-agent coordination
- Long-term strategic planning

10.9.2 Video Games

Modern AI systems tackle complex video games:

- Atari games: Raw pixel input, diverse objectives
- MOBA games: Team coordination, complex strategies
- First-person shooters: Spatial reasoning, rapid reactions

10.9.3 Transfer Learning

Current research focuses on:

- Learning general game-playing abilities
- Transferring knowledge between related games
- Meta-learning for rapid adaptation to new games

10.10 Summary

Reinforcement learning in games has yielded several key insights:

- **Self-play** is a powerful paradigm for generating training data without human supervision
- **Combining learning and search** leverages the strengths of both neural networks and tree search
- **Deep reinforcement learning** can discover strategies that surpass human knowledge
- **General algorithms** can achieve superhuman performance across diverse domains
- **Computational scaling** remains crucial for achieving breakthrough performance

The progression from hand-crafted game programs to general learning algorithms represents one of AI's greatest success stories, demonstrating the potential for RL to tackle complex sequential decision-making problems across many domains.

Bibliography

[1] David Silver. Lectures on reinforcement learning. URL: https://www.davidsilver.uk/teaching/, 2015.

Mathematical Background

.1 Probability Theory

This section reviews key concepts from probability theory used throughout the book.

.1.1 Probability Distributions

Definition .3 (Probability Mass Function)**.** *For a discrete random variable X, the probability mass function is: $p(x) = P(X = x)$ with $\sum_x p(x) = 1$.*

Definition .4 (Probability Density Function)**.** *For a continuous random variable X, the probability density function satisfies: $P(a \leq X \leq b) = \int_a^b f(x)dx$ with $\int_{-\infty}^{\infty} f(x)dx = 1$.*

.1.2 Expectation and Variance

Definition .5 (Expectation)**.** *The expectation of a random variable X is: $\mathbb{E}[X] = \begin{cases} \sum_x x \cdot p(x) & \textit{if } X \textit{ is discret} \\ \int_{-\infty}^{\infty} x \cdot f(x)dx & \textit{if } X \textit{ is continu} \end{cases}$*

Definition .6 (Variance)**.** $Var(X) = \mathbb{E}[(X - \mathbb{E}[X])^2] = \mathbb{E}[X^2] - (\mathbb{E}[X])^2$

.2 Linear Algebra

.2.1 Vectors and Matrices

Key operations used in reinforcement learning:

- **Vector addition**: $\mathbf{u} + \mathbf{v} = (u_1 + v_1, \ldots, u_n + v_n)$
- **Scalar multiplication**: $c\mathbf{u} = (cu_1, \ldots, cu_n)$
- **Dot product**: $\mathbf{u} \cdot \mathbf{v} = \sum_{i=1}^n u_i v_i$
- **Matrix multiplication**: $(AB)_{ij} = \sum_{k=1}^n A_{ik} B_{kj}$

.2.2 Norms

Definition .7 (Vector Norms)**.**

- *L_1 norm: $\|\mathbf{x}\|_1 = \sum_{i=1}^n |x_i|$*
- *L_2 norm: $\|\mathbf{x}\|_2 = \sqrt{\sum_{i=1}^n x_i^2}$*
- *L_∞ norm: $\|\mathbf{x}\|_\infty = \max_{i=1,\ldots,n} |x_i|$*

.3 Optimization

.3.1 Gradient Descent

The basic gradient descent update rule: $\mathbf{x}_{t+1} = \mathbf{x}_t - \alpha \nabla f(\mathbf{x}_t)$
where $\alpha > 0$ is the step size and $\nabla f(\mathbf{x})$ is the gradient of f at $\mathbf{x}$.

.3.2 Stochastic Gradient Descent

For objectives of the form $f(\mathbf{x}) = \mathbb{E}[g(\mathbf{x}, \xi)]$: $\mathbf{x}_{t+1} = \mathbf{x}_t - \alpha \nabla g(\mathbf{x}_t, \xi_t)$
where ξ_t is a random sample.

Algorithms Summary

This appendix provides a concise reference for the key algorithms presented in the book.

.4 Dynamic Programming

- **Policy Evaluation**: Iteratively apply Bellman expectation equation
- **Policy Iteration**: Alternate between policy evaluation and improvement
- **Value Iteration**: Apply Bellman optimality equation as update rule

.5 Model-Free Prediction

- **Monte Carlo**: Use sample returns to estimate value functions
- **TD(0)**: Bootstrap using immediate successor values
- **TD(λ)**: Combine multiple n-step returns using eligibility traces

.6 Model-Free Control

- **Monte Carlo Control**: Generalized policy iteration with MC evaluation
- **Sarsa**: On-policy TD control for action-value functions
- **Q-Learning**: Off-policy TD control using max operator

.7 Function Approximation

- **Linear methods**: Use feature vectors with linear combinations
- **Neural networks**: Nonlinear function approximation with backpropagation
- **Experience replay**: Reuse past experience for more stable learning

.8 Policy Gradient

- **REINFORCE**: Basic policy gradient with Monte Carlo returns
- **Actor-Critic**: Combine policy gradients with value function baselines
- **Natural gradients**: Use Fisher information matrix for better optimization