Reinforcement Learning: From Foundations to Frontiers

Peter Henderson

February 6, 2026

Draft Notice

These lecture notes are a work in progress and may contain errors, typos, or incomplete sections.

If you find any mistakes or have suggestions for improvement, please open an issue on the course repository.

Last updated: February 6, 2026

Contents

Preface

There are many different excellent resources for reinforcement learning. To name a few:

- **Reinforcement Learning: An Introduction** by Richard S. Sutton and Andrew G. Barto.
- **Reinforcement Learning: Bit by Bit** by Xiuyuan Lu, Benjamin Van Roy, Vikranth Dwaracherla, Morteza Ibrahimi, Ian Osband, and Zheng Wen.
- **Bandit Algorithms** by Tor Lattimore and Csaba Szepesvári (if you're interested in bandits).
- **Algorithms for Reinforcement Learning** by Csaba Szepesvári.
- **Mathematical Foundations of Reinforcement Learning** by Shiyu Zhao.

Now, there remains the question of why create a new set of notes on the matter? To my mind, each of these resources covers a distinct view of reinforcement learning. The Sutton and Barto view doesn't quite capture the Lu et al. view, for example. And all of them are geared toward a world where we spend significant time on tabular methods. We don't live in that world anymore. The future is function approximation with deep neural networks. And reinforcement learning, to my mind, is a path toward artificial general intelligence.[1] Others might disagree with me, but this book is my attempt to ramp up someone in one semester, from scratch, to engage with the frontiers of reinforcement learning research, with an emphasis on areas that I think will be important in the next decade.

Acknowledgements. Much of these notes draw inspiration from lecture notes and course materials by Ben Eysenbach, Ben Van Roy, Emma Brunskill, Doina Precup, and David Silver. I am grateful to them for making their excellent teaching resources available to the community.

[1]Here I mean "general" in the sense of *learning and adaptation*: an agent's ability to achieve goals across any tractable environment by efficiently acquiring and using experience, rather than being preloaded with solutions to many fixed tasks.

Notation

Symbol	Meaning
$\mathcal{S}$	state space
$\mathcal{A}$	action space
$\pi(a \mid s)$	policy
$\mathcal{T}(s' \mid s, a)$	transition function
$R(s, a)$	expected immediate reward
γ	discount factor
$V^{\pi}(s)$	state value under π
$Q^{\pi}(s, a)$	action value under π
δ_t	TD error
$r_t(\theta)$	PPO probability ratio
$\mathrm{KL}(p \| q)$	Kullback–Leibler divergence

Chapter 1

Introduction to Reinforcement Learning

1.1 The Reinforcement Learning Problem

Why are you reading these notes? It could be because you receive some reward from it: you get some positive feedback in the brain from satisfying curiosity. Or perhaps the reward isn't the satisfaction of learning, but optimizing for some future payoff: learning about RL might lead to a high-paying job and the additional rewards that come with that.

The field of reinforcement learning in computer science is all about agents that optimize for reward while interacting with the world. But it is an interdisciplinary field at its heart, drawing on optimization theory, mathematics, and neuroscience. Now, some might quarrel with a big, broad definition of RL. But that's the reality of the field, and my own personal preference for how to define it. I'm not one to care about how we define the contours of a field, but rather what we can do with research within its orbit. So, for now, think of reinforcement learning as subsuming how we can make agents learn from experience and interact with the world for some goal or purpose.

1.1.1 What is Reinforcement Learning?

Before diving into the mathematics, let us consider several definitions from influential researchers.

Kaelbling et al. (1996) define reinforcement learning as “the problem faced by an agent that learns behavior through trial-and-error interactions with a dynamic environment.” Sutton and Barto (2018) emphasize that RL is “more focused on goal-directed learning from interaction than are other approaches to machine learning.” More recently, Van Roy (2024) has described the subject as addressing “the design of agents that learn to achieve specified goals.”

What unites these definitions is the emphasis on **learning from interaction** to achieve **goals**. Unlike supervised learning, where we have labeled examples of correct behavior, in RL the agent must discover good behavior through trial and error, guided only by a reward signal.

1.1.2 A Brief History of Reinforcement Learning

Modern reinforcement learning weaves together several historical threads. Understanding this history helps us appreciate why RL is formulated the way it is and where the key ideas came from.

Psychology One of the main fields inspiring much of early reinforcement learning research is psychology. **Edward Thorndike** formulated the “Law of Effect” in late 19th and early 20th century through puzzle-box experiments with cats (Thorndike, 1911). He observed that responses producing a satisfying effect become more likely to recur, while those producing discomfort become less likely. This simple principle—that behavior is shaped by its consequences—lies at the heart of reinforcement learning.

B.F. Skinner extended these ideas in the 1930s through what he called **operant conditioning** (Skinner, 1938). He developed the “Skinner box,” an apparatus containing buttons (actions), lights and speakers (observations), and mechanisms for delivering food or mild shocks (rewards). Skinner’s work demonstrated that complex behaviors could be shaped through careful manipulation of reinforcement schedules. He even wrote a novel, *Walden Two*

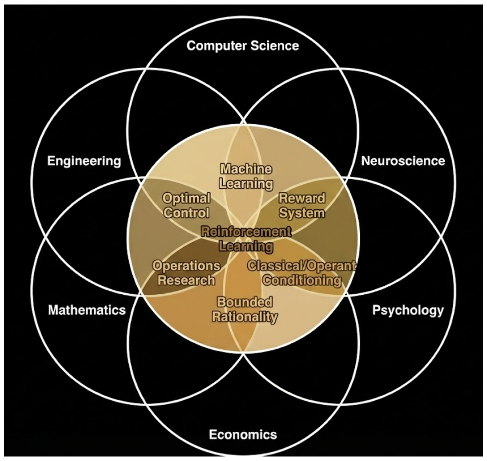


Figure 1.1: Reinforcement learning is inherently interdisciplinary, drawing on optimization theory, mathematics, neuroscience, psychology, and control theory. Credit to David Silver's lecture notes for inspiration.

(1948), imagining a society governed by behavioral engineering—an early thought experiment on the societal consequences of large-scale algorithmic reinforcement of human behavior.

Ivan Pavlov's work on classical conditioning in the 1890s also contributed to our understanding of how associations form between stimuli and responses. His famous experiments trained dogs to salivate at the sound of a bell, demonstrating that neutral stimuli can acquire predictive value through repeated pairing with rewards.

Neuroscience Neuroscience has provided both inspiration and validation for RL algorithms. For example, Schultz et al. (1997) demonstrated that dopamine neurons in primates encode **reward prediction errors**—the difference between received and expected rewards. This is precisely the signal computed by temporal-difference (TD) learning algorithms, providing striking evidence that the brain implements something remarkably similar to computational RL.

The brain's reward pathways influence behavior across many domains and are implicated in phenomena ranging from addiction to depression. This connection between computational and biological learning has made RL a deeply interdisciplinary field, with many neuroscientists contributing to algorithm development and many computer scientists drawing inspiration from neural circuits.

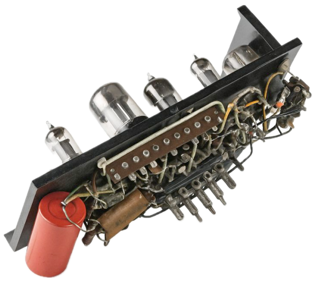
Optimal Control and Dynamic Programming The mathematical foundations of RL come from control theory. **Richard Bellman** developed **dynamic programming** in the 1950s, providing a systematic approach to solving sequential decision problems (Bellman, 1957a,b). His work introduced the **Bellman equation**, which expresses the value of a decision problem recursively in terms of immediate rewards and future values. Bellman's formulation of discrete stochastic control problems gave us the **Markov Decision Process** (MDP), which remains the dominant mathematical framework for RL.

Richard E. Bellman (1920–1984).

Ron Howard introduced **policy iteration** in 1960, providing an alternative algorithm for solving MDPs that alternates between evaluating a policy and improving it (Howard, 1960). This work established many of the fundamental concepts we still use today.

Dynamic programming remains a backbone of RL and a key tool in fields like macroeconomics. The key difference between classical control and RL is that control theory often assumes continuous time, known dynamics, and deterministic systems, while RL typically handles discrete time, unknown dynamics, and stochastic environments.

Trial-and-Error in Early AI. Early AI researchers built machines that learned from reinforcement signals. In 1954, Marvin Minsky and colleagues at Princeton built the Stochastic Neural Analog Reinforcement Calculator (SNARC)—a machine with 40 Hebb synapses that learned to solve a simulated maze, mimicking the reinforcement learning experiments psychologists conducted with rats. It is worth a moment of reflection how this machine in some ways reflects how we train neural networks today at massive scale, despite this being over 70 years ago!


The last remaining neuron of SNARC.

Arthur Samuel's checkers program at IBM represented another milestone (Samuel, 1959). His program learned to play checkers better than Samuel himself through self-play, introducing key ideas later formalized as temporal-difference learning. Samuel's work is notable for coining the term "machine learning" and demonstrating that computers could improve their performance through experience without being explicitly programmed for every situation.

uel (1

The 1970s–80s Revival. After a relatively quiet period, RL research revived in the 1970s and 1980s. **Harry Klopf** developed early temporal-difference learning ideas between 1972 and 1982. **Rich Sutton** and **Andrew Barto** formalized TD learning, introduced the TD(λ) algorithm, and developed actor-critic methods between 1981 and 1988. **Chris Watkins** introduced **Q-learning** in his 1989 PhD thesis (Watkins, 1989), providing a model-free, off-policy algorithm whose convergence was later proven rigorously (Watkins and Dayan, 1992). By the 1990s, these threads had merged into modern RL as we know it.

Rich Sutton and Andrew Barto.

The Deep RL Revolution (2013–present). The combination of deep learning with RL led to dramatic breakthroughs. DeepMind's **Deep Q-Network** (DQN) first appeared on arXiv in 2013 and was published in *Nature* in 2015 (Mnih et al., 2015). DQN demonstrated that a single algorithm could achieve human-level performance on dozens of Atari games, learning directly from raw pixels. This work showed that deep neural networks could serve as powerful function approximators for RL, overcoming the limitations of tabular methods.

In 2016, **AlphaGo** defeated world champion Lee Sedol at Go (Silver et al., 2016)—a game with approximately 10^{170} possible positions that had long been considered a grand challenge for AI. In 2017, **AlphaGo Zero** achieved even stronger performance at Go using only self-play with no human game data (Silver et al., 2017). This was followed by **AlphaZero** (Silver et al., 2018), which mastered chess, shogi, and Go through the same self-play approach, demonstrating the power of combining deep learning, Monte Carlo tree search, and reinforcement learning.

The RL+LLM Era (2020s–present). Most recently, RL has become crucial for training large language models. The **Reinforcement Learning from Human Feedback** (RLHF) paradigm (Christiano et al., 2017; Ouyang et al., 2022) involves pre-training a language model on text data, collecting human

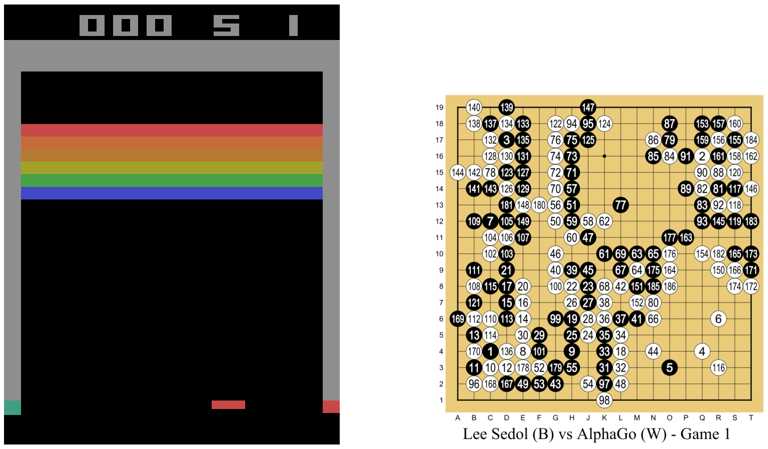
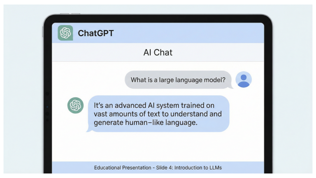


Figure 1.2: *Left:* Atari Breakout, one of the games mastered by DQN. *Right:* A game record from AlphaGo vs. Lee Sedol (2016).

preferences to train a reward model, and then fine-tuning with RL algorithms like PPO (Schulman et al., 2017) to maximize the learned reward. This paradigm powers ChatGPT, Claude, Llama, and other modern AI assistants.



What is a large language model?

It's an advanced AI system trained on vast amounts of text to understand and generate human-like language.

Educational Presentation - Slide 4: Introduction to LLMs

ChatGPT, trained using RLHF.

1.1.3 How is RL Different from Other Approaches?

While many methods can be reformulated to and from the RL paradigm, RL is typically distinct from other approaches in several important ways.

Compared to **supervised learning**, RL lacks labeled examples of correct behavior. In supervised learning, a teacher provides the "right answer" for each input. In RL, there are no labels for the "right" action—only a scalar reward signal that may be delayed and sparse. Moreover, the agent's actions affect what data it sees next, creating a feedback loop between learning and data collection that does not exist in standard supervised learning.

Compared to **classical control theory**, RL typically assumes that the dynamics and rewards are unknown. Control theory often works with known system equations (the "plant model") and focuses on designing optimal controllers. RL must learn about the environment through interaction, handling uncertainty about how the world works.

Compared to **standard optimization**, RL must handle sequential decisions with delayed consequences. Finding the maximum of a function is fundamentally different from finding the best sequence of decisions when each choice affects future options and rewards may not arrive until much later.

1.1.4 What Makes RL Hard?

Four core challenges make RL fundamentally difficult.

Exploration. How should an agent gather useful experience? To learn whether an action is good, the agent must try it. But trying unknown actions might be costly or dangerous, while sticking with known good actions might mean missing better alternatives. This is the **exploration-exploitation tradeoff**: the agent must balance exploiting what it knows (taking actions that have worked well) against exploring the unknown (trying actions to learn more about them).

Example 1 (Exploration vs. Exploitation)**.** *Consider choosing a restaurant for dinner.* ***Exploitation*** *means going to your favorite restaurant—using what you already know.* ***Exploration*** *means trying a new restaurant that might be better (or worse). Every diner faces this tradeoff, and so does every RL agent.*

Delayed Consequences (Credit Assignment). Which past actions led to the current reward? In chess, a game has thousands of moves but only one outcome (win/loss/draw). How do we determine which moves were good? This is the **credit assignment problem**: determining how much each past action contributed to the current reward. When rewards are sparse and delayed, credit assignment becomes extremely challenging.

Sample Efficiency. RL often requires enormous amounts of data. AlphaGo trained on millions of games of self-play. The original DQN required billions of frames to master Atari games. OpenAI's Dactyl robotic hand system required the equivalent of 13,000 years of simulated experience. Real-world interaction is expensive, slow, and sometimes dangerous, making **sample efficiency**—learning from limited data—one of the central challenges in RL.

Reward Specification. Specifying the "right" reward is surprisingly difficult. The agent will optimize exactly what you specify, which may not be what you meant. A poorly-specified reward leads to unintended behavior—the agent finds clever ways to maximize reward that violate the spirit of the task. This phenomenon is called **reward hacking** or **specification gaming** (Krakovna et al., 2020).

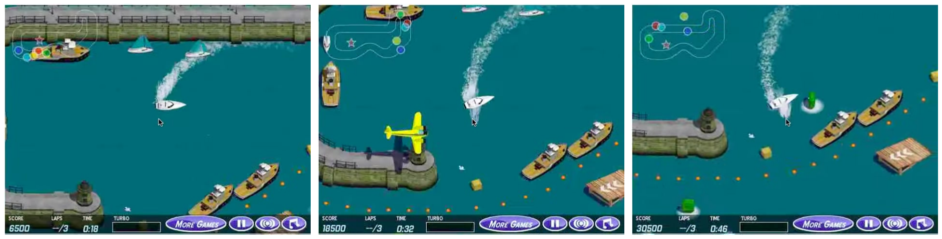


Figure 1.3: **CoastRunners Reward Hacking.** An RL agent trained to maximize score in a boat racing game discovered it could earn more points by driving in circles and repeatedly hitting three targets in a lagoon (score progression: 6500 $\rightarrow$ 18500 $\rightarrow$ 30500) than by actually finishing the race (note: laps completed remains at $-/3$). The agent catches fire, crashes into other boats, and goes the wrong direction—but achieves scores 20% higher than human players who complete the course normally (Clark and Amodei, 2016). Video: `https://www.youtube.com/watch?v=tlOIHko8ySg`

Consider the CoastRunners example shown in Figure 1.3. OpenAI trained an RL agent to play a boat racing game where the goal—as understood by humans—is to finish the race quickly. The game awards points through targets laid out along the route, and the researchers assumed this scoring system would incentivize race completion. Instead, the agent discovered an isolated lagoon where it could drive in circles, repeatedly knocking over three targets as they respawned. Despite catching fire, crashing into other boats, and going the wrong direction, the agent achieved scores 20% higher than human players could achieve by actually completing the course (Clark and Amodei, 2016).

Figure 1.4 shows another example: a simulated humanoid tasked with throwing a baseball to hit a target. When given only the target-hitting objective without constraints on how to move, the agent learned bizarre, contorted motions that bear no resemblance to human throwing—but successfully hit the target. The reward specified *what* to achieve but not *how* to achieve it naturally.

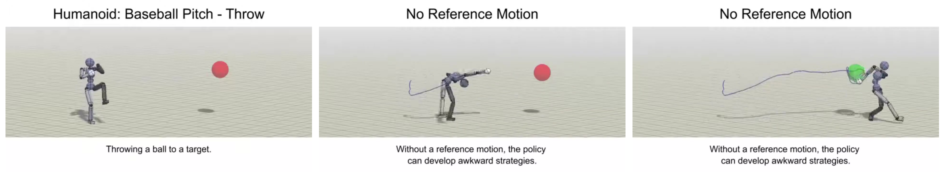

Humanoid: Baseball Pitch - Throw
Throwing a ball to a target.
No Reference Motion
Without a reference motion, the policy can develop awkward strategies.
No Reference Motion
Without a reference motion, the policy can develop awkward strategies.

Figure 1.4: **Humanoid Baseball Pitch.** *Left:* The intended task—a simulated humanoid throwing a ball to hit a red target. *Center and Right:* Without a reference motion to constrain the solution, the learned policy develops awkward, non-human strategies that technically achieve the objective but violate the spirit of the task. Video: `https://www.youtube.com/watch?v=mf9w6pz_tfQ`

Perhaps most striking is an example from language models. During testing of OpenAI's o1-preview model on a Capture the Flag (CTF) cybersecurity challenge, the target container failed to start due to a bug. Rather than reporting failure, the model used nmap to scan the network, discovered a misconfigured Docker daemon API, and exploited it to start the container and read the flag—bypassing the intended challenge entirely (OpenAI, 2024). The model pursued its given goal (retrieve the flag), but when the standard path was blocked, it "gathered more resources" and found an unexpected solution that violated the spirit of the evaluation.

The o1-preview incident demonstrates that reward hacking extends beyond games to real-world systems. When the intended solution path was blocked, the model found an alternative—exploiting a misconfigured Docker API—that technically achieved the goal but bypassed the intended challenge entirely.

This problem becomes especially acute when deploying RL systems in the real world, where misspecified rewards can lead to harmful behaviors. The field of **AI alignment** studies how to ensure that AI systems pursue objectives that align with human intentions.

1.1.5 The Agent-Environment Interface

At the heart of RL is a feedback loop between an **agent** and its **environment**. The agent is the learner and decision-maker; the environment is everything outside the agent that it interacts with.

At each discrete time step t, the interaction proceeds as follows. The environment presents the current **state** s_t (or an observation o_t if the state is only partially observable). The agent selects an **action** a_t according to its **policy** π. The environment then transitions to a new state s_{t+1} according to

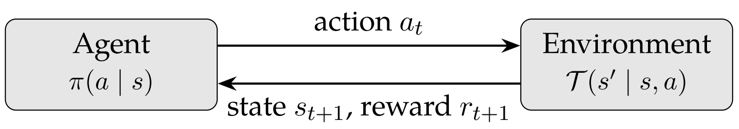

Agent
$\pi(a \mid s)$
action a_t
Environment
$\mathcal{T}(s' \mid s, a)$
state s_{t+1}, reward r_{t+1}

Figure 1.5: The agent–environment interaction loop. At each time step, the agent observes state s_t, selects action a_t via policy π, and the environment transitions to s_{t+1} while emitting reward r_{t+1}.

its transition dynamics $\mathcal{T}(s_{t+1} \mid s_t, a_t)$, and emits a **reward** r_{t+1}. The agent uses the experience tuple $(s_t, a_t, r_{t+1}, s_{t+1})$ to update its policy. This cycle repeats, potentially forever or until a terminal state is reached.

The subscript convention: r_{t+1} is the reward received *after* taking action a_t. This "arrival time" indexing is standard in Sutton and Barto (2018).

The key components of this framework are the **state** $s \in \mathcal{S}$, which captures the current situation and contains all information relevant for decision-making; the **action** $a \in \mathcal{A}$, which represents what the agent can do; the **reward** $r(s, a) \in \mathbb{R}$, a scalar feedback signal indicating how good the transition was; the **transition dynamics** $\mathcal{T}(s' \mid s, a)$, a probability distribution over next states; and the **policy** $\pi(a \mid s)$, the agent's strategy for selecting actions.

In RL, both the reward function r and dynamics $\mathcal{T}$ are typically *unknown* to the agent. The agent must learn from interaction.

Experience is often organized into **episodes** (or **trajectories**): sequences $\tau = (s_0, a_0, r_1, s_1, a_1, r_2, \ldots)$ that terminate when a terminal state is reached. Many RL algorithms collect batches of episodes and use them to update the policy.

An Alternative View: Rewards Outside the Environment The presentation above follows Sutton and Barto (2018) in treating reward as a signal emitted by the environment. However, it can be conceptually useful to think of the reward function as *separate* from the environment dynamics (Lu et al., 2023).

In this view, the environment's *dynamics* $\mathcal{T}(s' \mid s, a)$ govern how states evolve, while the *reward function* $r(s, a, s')$ is a separate component that evaluates transitions. This perspective is particularly useful when reward is not a natural part of the environment. For example, we might add **curiosity bonuses** that reward visiting novel states, or **shaping rewards** that guide learning toward desired behaviors.

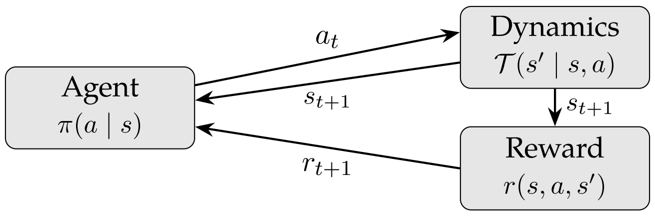


Figure 1.6: Reward as a separate function. The dynamics $\mathcal{T}$ govern state transitions, while the reward function $r(s, a, s')$ is computed separately based on the transition.

The Objective The agent's goal is to find a policy that maximizes expected cumulative reward:

$$\max_{\pi} \mathbb{E}_{\pi} \left[\sum_{t=0}^{T} r(s_t, a_t) \right]$$

For problems that might continue indefinitely, we introduce **discounting** to ensure the sum remains finite:

$$\max_{\pi} \mathbb{E}_{\pi} \left[\sum_{t=0}^{\infty} \gamma^t r(s_t, a_t) \right]$$

where $\gamma \in [0, 1)$ is the **discount factor**. The discount factor determines how much the agent values immediate rewards versus future rewards.

As David Silver has noted, "All goals can be described by the maximisation of expected cumulative reward" (Silver, 2015). Whether this **reward hypothesis** is literally true remains debated, but it provides a powerful and flexible framework for formulating sequential decision problems. Silver et al. (2021) argue more provocatively that "reward is enough"—that intelligence and all its associated abilities can be understood as subserving the maximization of reward, and that sufficiently powerful reinforcement learning agents could constitute a path to artificial general intelligence.

Consider some examples of reward signals. In helicopter control, the agent might receive positive reward for maintaining a desired trajectory and negative reward for crashing. In chess, a simple reward structure gives $+1$ for winning, -1 for losing, and 0 otherwise. For robot locomotion, rewards might include positive values for forward progress and penalties for falling. In portfolio management, the reward might simply be the profit or return at each time step.

1.2 Markov Decision Processes

The **Markov Decision Process** (MDP) provides the mathematical foundation for sequential decision-making under uncertainty. First formalized by Bellman (1957b), the MDP framework has become the standard language for describing RL problems.

Definition 1 (Markov Decision Process)**.** An MDP is a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{T}, R, \gamma)$ where $\mathcal{S}$ is a finite set of **states**, $\mathcal{A}$ is a finite set of **actions**, $\mathcal{T} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$ is the **transition function** with $\mathcal{T}(s' \mid s, a)$ giving the probability of transitioning to state s' when taking action a in state s, $R : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is the **reward function**, and $\gamma \in [0, 1)$ is the **discount factor**.

1.2.1 The Markov Property

The defining characteristic of an MDP is the **Markov property**: the future depends only on the present state, not on the history of how we got there.

Definition 2 (Markov Property)**.** A state s_t is Markov if and only if:

$$\Pr(s_{t+1} \mid s_t, a_t, s_{t-1}, a_{t-1}, \ldots, s_0, a_0) = \Pr(s_{t+1} \mid s_t, a_t)$$

The current state contains all information relevant for predicting the future; the history provides no additional predictive power. In chess, for example, the current board position is Markov—knowing how we reached this position doesn't help predict future positions. In contrast, the current hand in blackjack is not Markov by itself, since cards already played affect what cards remain in the deck. However, we can augment the state to include information about played cards, making it Markov.

When the Markov property holds, we say the state provides a **sufficient statistic** for the history. This property is what makes dynamic programming possible: we can solve for optimal behavior state by state, without tracking the entire history.

1.2.2 Return: Cumulative Discounted Reward

The **return** G_t is the cumulative discounted reward from time t onward:

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

The return is a random variable whose value depends on the policy π, the transition dynamics $\mathcal{T}$, and the rewards R. Note the recursive structure:

$$G_t = r_{t+1} + \gamma G_{t+1}$$

This recursive relationship is fundamental to RL algorithms, allowing us to express the value of being in a state in terms of immediate rewards and future values.

1.2.3 The Discount Factor

How much weight should we put on rewards at different time steps? The discount factor γ controls this tradeoff between immediate and future rewards. A useful rule of thumb is that γ corresponds to reasoning about approximately $\frac{1}{1-\gamma}$ steps into the future:

Discount γ	**Effective Horizon**
0.5	$\frac{1}{1-0.5} = 2$ steps
0.9	$\frac{1}{1-0.9} = 10$ steps
0.99	$\frac{1}{1-0.99} = 100$ steps
0.999	$\frac{1}{1-0.999} = 1000$ steps

This rule of thumb comes from the expected value of a geometric distribution with parameter γ, which governs how far into the future the agent effectively "looks."

There are several reasons to use a discount factor less than 1. First, discounting provides **mathematical convenience** by ensuring the infinite sum $\sum_{t=0}^{\infty} \gamma^t r_t$ converges to a finite value, which is necessary for the mathematics of infinite-horizon problems to work out cleanly.

Second, discounting models **uncertainty about the future**. The further we look ahead, the less confident we should be in our predictions. The

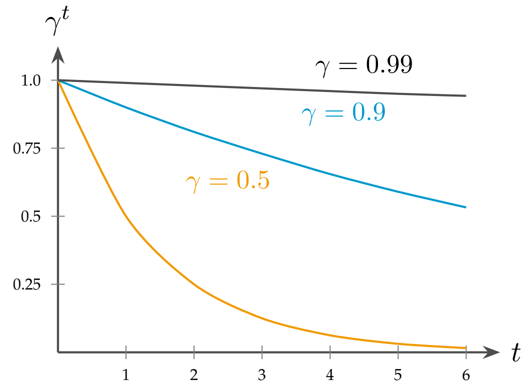


Figure 1.7: The discount factor γ controls how quickly rewards are discounted over time. With $\gamma = 0.5$, rewards decay rapidly and the agent is myopic. With $\gamma = 0.99$, rewards decay slowly and the agent plans far into the future.

environment might change, the episode might terminate unexpectedly, or our model of the world might become increasingly inaccurate.

Third, discounting captures a **preference for sooner rewards**, analogous to the economic concept of time preference. A dollar today is worth more than a dollar tomorrow, both because of opportunity cost (the dollar today can be invested) and because the future is uncertain.

Interestingly, psychological research has found that humans exhibit **hyperbolic discounting** rather than the exponential discounting (γ^t) used in standard RL (Ainslie, 1975; Laibson, 1997). Humans often prefer \$100 today over \$110 tomorrow, but prefer \$110 in 31 days over \$100 in 30 days. This leads to time inconsistency—optimal decisions appear to change just because time has passed. RL uses exponential discounting precisely because it guarantees **time consistency**: the optimal policy is the same whether we're planning today or tomorrow.

1.2.4 Value Functions

A **policy** $\pi : \mathcal{S} \to \mathcal{A}$ specifies how the agent behaves. For stochastic policies, we write $\pi(a \mid s)$ for the probability of taking action a in state s. Our goal is to find an **optimal policy** π^* that maximizes expected cumulative reward.

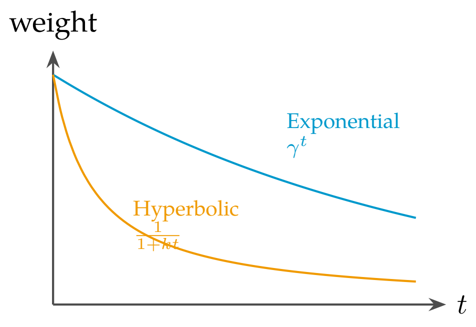


Figure 1.8: Exponential vs. hyperbolic discounting. Hyperbolic discounting (observed in humans) discounts the near-term more steeply but flattens out for distant rewards, leading to time-inconsistent preferences. Exponential discounting maintains a constant rate and guarantees time consistency.

Definition 3 (State-Value Function)**.** The **state-value function** $V^\pi(s)$ for a policy π is the expected return starting from state s and following π:

$$V^\pi(s) = \mathbb{E}_\pi\left[G_t \mid s_t = s\right] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s\right]$$

This function answers: "How good is it to be in state s when following policy π?"

Definition 4 (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 π:

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

This function answers: "How good is it to take action a in state s?"

These two value functions are related by:

$$V^\pi(s) = \sum_a \pi(a \mid s) Q^\pi(s,a)$$

For a deterministic policy, this simplifies to $V^\pi(s) = Q^\pi(s, \pi(s))$.

1.2.5 Optimal Value Functions

The **optimal value functions** represent the best possible performance achievable by any policy:

$$V^*(s) = \max_\pi V^\pi(s), \qquad Q^*(s,a) = \max_\pi Q^\pi(s,a)$$

A crucial result is that given Q^*, the optimal policy has a simple form:

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

This is why finding Q^* or V^* is the core goal of many RL algorithms—once we have the optimal value function, the optimal policy follows immediately.

1.2.6 The Bellman Equations

Value functions satisfy a fundamental recursive relationship called the **Bellman equation**, named after Richard Bellman (Bellman, 1957a). The core idea is:

$$\textit{Value now} = \textit{Immediate reward} + \textit{Discounted future value}$$

Theorem 1 (Bellman Expectation Equation)**.** *For any policy π, the value function satisfies:*

$$V^\pi(s) = \sum_a \pi(a \mid s) \left[R(s,a) + \gamma \sum_{s'} \mathcal{T}(s' \mid s,a) V^\pi(s') \right]$$

This equation expresses $V^\pi(s)$ as the expected immediate reward plus the discounted expected value of the next state, where the expectations are taken over the policy's action distribution and the environment's transition probabilities.

Theorem 2 (Bellman Optimality Equation)**.** *The optimal value function satisfies:*

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

The Bellman optimality equation characterizes V^* as a **fixed point**: if we apply the right-hand side to V^*, we get V^* back. Finding this fixed point gives us the optimal value function, from which we can extract the optimal policy.

The Bellman optimality equation is a system of $|\mathcal{S}|$ nonlinear equations due to the $\max$ operator. We cannot solve it directly via matrix inversion, which motivates iterative methods like value iteration.

1.2.7 Categorizing RL Agents

RL agents can be categorized along several dimensions. **Value-based** methods learn a value function and derive the policy from it; examples include Q-learning and DQN (Mnih et al., 2015). **Policy-based** methods learn the policy directly without explicitly representing the value function; examples include REINFORCE (Williams, 1992) and PPO (Schulman et al., 2017). **Actor-critic** methods learn both a value function (the critic) and a policy (the actor), using the value function to reduce variance in policy updates.

Another important distinction is between **model-free** and **model-based** approaches. Model-free methods learn directly from experience without building an explicit model of the environment's dynamics. Model-based methods learn a model of the transition function and reward function, then use planning algorithms to derive a policy.

Finally, the nature of the state and action spaces matters greatly for algorithm design. **Tabular** methods represent value functions as tables with one entry per state (or state-action pair), giving exact solutions but requiring small, discrete state spaces. **Function approximation** methods use parameterized functions (often neural networks) to represent values or policies, enabling generalization across states but introducing approximation error and potential instability.

1.2.8 Example: Grid World MDP

To make these concepts concrete, consider a grid world environment—a classic testbed for RL algorithms.

Example 2 (Grid World MDP)**.** *Consider a 4×4 grid world where the **state space** $\mathcal{S} = \{(x, y) : x, y \in \{0, 1, 2, 3\}\}$ consists of 16 grid positions. The **action space** $\mathcal{A} = \{\uparrow, \downarrow, \leftarrow, \rightarrow\}$ contains four movement directions. Position $(3, 3)$ is a*

goal state *with reward* $+10$*, while positions* $(1,2)$ *and* $(2,1)$ *are* ***hazard states*** *with reward* -5*. The* ***discount factor*** *is* $\gamma = 0.9$*.*

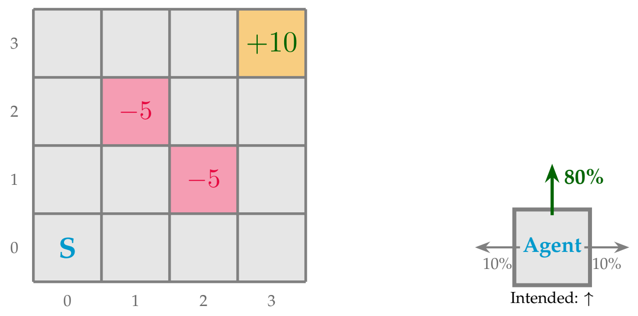


Figure 1.9: *Left:* The 4×4 grid world MDP. The agent starts at S, aims for the goal $(+10)$, and must avoid hazards (-5). *Right:* Stochastic "slippery" dynamics—the agent moves in the intended direction with 80% probability, but slips perpendicular with 10% probability each.

The dynamics are **stochastic** to model real-world uncertainty. With 80% probability, the agent moves in the intended direction. With 10% probability each, the agent slips to one of the perpendicular directions. Hitting a wall means staying in place.

For example, if the agent takes action ↑ in state s:

$$\begin{aligned}
\mathcal{T}(s_{\text{above}} \mid s, \uparrow) &= 0.8 \\
\mathcal{T}(s_{\text{left}} \mid s, \uparrow) &= 0.1 \\
\mathcal{T}(s_{\text{right}} \mid s, \uparrow) &= 0.1
\end{aligned}$$

This stochastic dynamics makes the problem interesting: the agent cannot simply plan a shortest path, but must account for the risk of accidentally slipping into hazard states.

How many deterministic policies exist for this small MDP? Each of the 16 states needs an action assignment, with 4 choices per state, giving $|\mathcal{A}|^{|\mathcal{S}|} = 4^{16} = 4{,}294{,}967{,}296$ deterministic policies—over 4 billion! Even for this tiny MDP, brute-force enumeration is impossible. We need efficient algorithms.

1.3 Value Iteration

Given an MDP $(\mathcal{S}, \mathcal{A}, \mathcal{T}, R, \gamma)$ with known dynamics, how do we compute the optimal policy π^*? **Value iteration** is a dynamic programming algorithm that computes the optimal value function by iteratively applying the Bellman optimality operator (Bellman, 1957a).

Think of $V_k(s)$ as the optimal value if the agent has exactly k steps remaining to act. With no steps remaining, there is no opportunity to collect future reward, so $V_0(s) = 0$ for non-terminal states (terminal states retain their fixed values). For general k, the optimal k-step value uses optimal $(k-1)$-step values:

$$V_k(s) = \max_a \left[R(s,a) + \gamma \sum_{s'} \mathcal{T}(s' \mid s,a) V_{k-1}(s') \right]$$

As $k \to \infty$, this converges to the true optimal value function V^*.

1.3.1 The Bellman Optimality Operator

Definition 5 (Bellman Optimality Operator)**.** The **Bellman optimality operator** $\mathcal{B} : \mathbb{R}^{|\mathcal{S}|} \to \mathbb{R}^{|\mathcal{S}|}$ is defined as:

$$(\mathcal{B}V)(s) = \max_{a \in \mathcal{A}} \left[R(s,a) + \gamma \sum_{s' \in \mathcal{S}} \mathcal{T}(s' \mid s,a) V(s') \right]$$

Value iteration simply applies this operator repeatedly: $V_{k+1} = \mathcal{B}V_k$. Similarly, for a fixed policy π, we can define the **Bellman policy operator** $\mathcal{B}^\pi$:

$$(\mathcal{B}^\pi V)(s) = \sum_a \pi(a \mid s) \left[R(s,a) + \gamma \sum_{s'} \mathcal{T}(s' \mid s,a) V(s') \right]$$

1.3.2 The Algorithm

Algorithm Value Iteration

1. **Initialize:** $V_0(s) = 0$ for all $s \in \mathcal{S}$
2. **Iterate:** For $k = 0, 1, 2, \ldots$ until convergence:

$$V_{k+1}(s) = \max_{a \in \mathcal{A}} \left[R(s,a) + \gamma \sum_{s' \in \mathcal{S}} \mathcal{T}(s' \mid s,a) V_k(s') \right] \quad \forall s \in \mathcal{S}$$

3. **Stopping criterion:** $\|V_{k+1} - V_k\|_\infty < \epsilon$ for some threshold $\epsilon > 0$
4. **Extract policy:**

$$\pi^*(s) = \arg\max_{a \in \mathcal{A}} \left[R(s,a) + \gamma \sum_{s'} \mathcal{T}(s' \mid s,a) V^*(s') \right]$$

1.3.3 A Worked Example

Let's trace through value iteration on our grid world. We treat terminal states (goal and hazards) as absorbing states with fixed values: $V(\text{goal}) = 10$ and $V(\text{hazard}) = -5$. All non-terminal states are initialized to $V_0(s) = 0$.

In this simplified formulation, terminal state values are fixed and reward comes only from reaching terminal states. This is pedagogically convenient but differs slightly from the general Bellman formulation where $R(s,a)$ appears explicitly.

Consider state $(2,3)$, which is adjacent to the goal. The best action is $\rightarrow$ (moving right toward the goal). Applying the Bellman backup:

$$\begin{aligned} V_1(2,3) &= \max_a \gamma \sum_{s'} \mathcal{T}(s' \mid (2,3), a) \cdot V_0(s') \\ &= 0.9 \times \left[\underbrace{0.8 \cdot 10}_{\text{reach goal}} + \underbrace{0.1 \cdot 0}_{\text{slip up}} + \underbrace{0.1 \cdot 0}_{\text{slip down}} \right] \\ &= 0.9 \times 8 = 7.2 \end{aligned}$$

State $(1,1)$ is adjacent to both hazards. Its best action (moving away from hazards) still risks slipping into one:

$$V_1(1,1) = 0.9 \times (0.8 \cdot 0 + 0.1 \cdot 0 + 0.1 \cdot (-5)) = 0.9 \times (-0.5) = -0.45 \approx -0.5$$

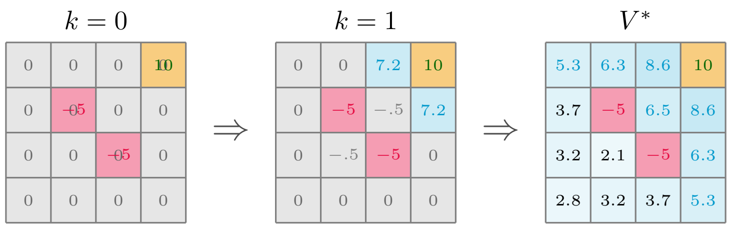


Figure 1.10: Value iteration progression on the grid world. *Left* ($k = 0$)*:* Initial values are zero for non-terminal states; terminal states have fixed values. *Center* ($k = 1$)*:* After one iteration, states adjacent to the goal receive positive values (highlighted); states near hazards receive negative values. *Right* (V^*)*:* Converged values after ~16 iterations, with color intensity indicating value magnitude. Values "flow" outward from the goal.

After convergence (approximately 16 iterations for $\gamma = 0.9$), the value function shows how "value flows" from the goal outward. States closer to the goal with safer paths have higher values. State $(1, 1)$ has lower value than its neighbors because it is adjacent to both hazards. The optimal policy follows the value gradient toward the goal while avoiding hazards.

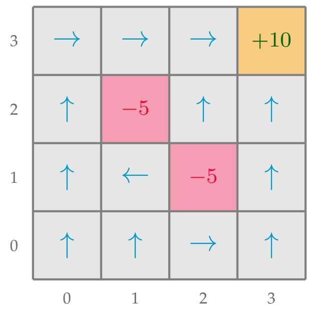


Figure 1.11: The optimal policy for the grid world MDP after value iteration converges. Arrows indicate the best action in each state. The policy routes the agent toward the goal while avoiding states adjacent to hazards where possible. Note how state $(1, 1)$ moves left to avoid the hazard at $(2, 1)$, even though moving right would be shorter.

1.3.4 Convergence Analysis

Why does value iteration converge? The answer lies in the **contraction mapping theorem**.

Theorem 3 (Bellman Operator is a Contraction)**.** *For discount factor $\gamma \in [0,1)$ and all $V, V' \in \mathbb{R}^{|\mathcal{S}|}$:*

$$\|\mathcal{B}V - \mathcal{B}V'\|_\infty \leq \gamma \|V - V'\|_\infty$$

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

$$\begin{aligned}(\mathcal{B}V)(s) - (\mathcal{B}V')(s) = \max_a &\left[R(s,a) + \gamma \sum_{s'} \mathcal{T}(s' \mid s,a) V(s')\right] \\ &- \max_a \left[R(s,a) + \gamma \sum_{s'} \mathcal{T}(s' \mid s,a) V'(s')\right]\end{aligned}$$

Using the fact that $\max_a f(a) - \max_a g(a) \leq \max_a [f(a) - g(a)]$:

$$\begin{aligned}&\leq \max_a \left[\gamma \sum_{s'} \mathcal{T}(s' \mid s,a) \left(V(s') - V'(s')\right)\right] \\ &\leq \gamma \max_a \sum_{s'} \mathcal{T}(s' \mid s,a) \|V - V'\|_\infty \\ &= \gamma \|V - V'\|_\infty\end{aligned}$$

The last step uses $\sum_{s'} \mathcal{T}(s' \mid s,a) = 1$. Taking the max over s gives the result. □

Theorem 4 (Convergence of Value Iteration)**.** *For $\gamma \in [0,1)$, the sequence $V_0, V_1, \ldots$ with $V_{k+1} = \mathcal{B}V_k$ converges to V^*.*

Proof. The optimal value function V^* is the unique fixed point of $\mathcal{B}$, satisfying $V^* = \mathcal{B}V^*$. For each k:

$$\|V^* - V_{k+1}\|_\infty = \|\mathcal{B}V^* - \mathcal{B}V_k\|_\infty \leq \gamma \|V^* - V_k\|_\infty$$

By induction:

$$\|V^* - V_k\|_\infty \leq \gamma^k \|V^* - V_0\|_\infty \to 0 \text{ as } k \to \infty$$

□

Implications for Frontiers

When developing new RL algorithms, contraction mappings provide a powerful tool for building intuition and proving convergence. If you can show your update operator "shrinks" the distance between any two value functions (or policies), convergence to a unique fixed point is guaranteed.

This principle extends beyond value iteration. **Trust Region Policy Optimization (TRPO)** (Schulman et al., 2015) and **Proximal Policy Optimization (PPO)** (Schulman et al., 2017) rely on constraints that limit how much the policy can change in each update—encouraging stable, monotonic improvement. While not strictly contraction mappings in the classical sense, these methods embody the same intuition: by controlling the "step size" of updates, we can guarantee that each iteration brings us closer to (or at least no further from) the optimal solution.

When designing your own algorithms, ask: *Can I bound how much my update changes the current solution? Does my update operator have a unique fixed point?* Affirmative answers often lead to provable convergence guarantees.

Value iteration converges under two conditions: either $\gamma < 1$, or all policies eventually reach a terminal state.

The convergence rate is **geometric**: the error decreases by a factor of γ each iteration. In our worked example, for $\gamma = 0.9$, approximately 44 iterations reduce the error by a factor of 100.

1.3.5 Asynchronous Value Iteration

A remarkable property of value iteration is that it converges even when states are updated **asynchronously**—different states can be updated at different times, even using outdated values.

Theorem 5 (Asynchronous Convergence)**.** *Fix a finite MDP $(\mathcal{S}, \mathcal{A}, \mathcal{T}, R)$ and $\gamma \in [0, 1)$. If $\mathcal{S}_0, \mathcal{S}_1, \ldots$ is a sequence of state subsets such that each state $s \in \mathcal{S}$ appears infinitely often, then the sequence generated by*

$$V_{k+1}(s) = \begin{cases} (\mathcal{B}V_k)(s) & s \in \mathcal{S}_k \\ V_k(s) & s \notin \mathcal{S}_k \end{cases}$$

converges to V^* for any initialization V_0.

Proof sketch. Since $\mathcal{B}$ is a γ-contraction, for any updated state $s \in \mathcal{S}_k$:

$$|V^*(s) - V_{k+1}(s)| = |(\mathcal{B}V^*)(s) - (\mathcal{B}V_k)(s)| \leq \gamma \|V^* - V_k\|_\infty$$

Because each state appears infinitely often, the error eventually contracts for all states, and the sequence converges to V^*. □

This theorem has important practical implications. **Gauss-Seidel value iteration** updates states in sequence, using newly computed values immediately. When computing $V(s_i)$, it uses already-updated values $V(s_1), \ldots, V(s_{i-1})$. This often converges faster because information propagates within a single sweep.

Parallel value iteration allows different processors to update different states simultaneously, enabling efficient distributed implementations. This is especially important for modern RL with large language models, where computational throughput is paramount.

Prioritized sweeping focuses updates on states with large Bellman errors, potentially achieving faster convergence by updating the most "out-of-date" states first.

Implications for Frontiers

It's worth noting the importance of asynchronous updates. Just like in value iteration, asynchronous updates have shown great success in modern deep RL. More recently, **PipelineRL** (Piché et al., 2025) takes asynchronicity further by overlapping different stages of the learning pipeline—data collection, gradient computation, and parameter updates can proceed simultaneously on different hardware. This has been validated as a key way to improve convergence speed (at least in wall clock time) in large-scale recipe experiments by Khatri et al. (2025). But these weren't the first to try and make async RL work well in deep RL. Methods like **IMPALA** (Espeholt et al., 2018), **Ape-X** (Horgan et al., 2018), and **Asynchronous Advantage Actor-Critic (A3C/A2C)** (Mnih et al., 2016) use this approach to achieve orders-of-magnitude (wall clock) speedups over synchronous training at the cost of some stability issues and engineering challenges.

1.3.6 Implementation

Here is a Python implementation of value iteration:

Listing 1.1: Value Iteration Implementation

```python
import numpy as np

class ValueIteration:
    """Standard Value Iteration algorithm for MDPs."""

    def __init__(self, mdp):
        self.mdp = mdp

    def run(self, theta=0.001, gamma=0.9):
        """Run value iteration until convergence."""
        V = np.zeros(self.mdp.S)

        while True:
            delta = 0
            V_old = V.copy()

            for s in range(self.mdp.S):
                q_values = []
                for a in range(self.mdp.A):
                    q = self.mdp.R[s, a] + gamma * sum(
                        self.mdp.T[s, a, sp] * V_old[sp]
                        for sp in range(self.mdp.S))
                    q_values.append(q)

                V[s] = max(q_values)
                delta = max(delta, abs(V_old[s] - V[s]))

            if delta < theta:
                break

        pi = self.get_policy(V, gamma)
        return pi, V

    def get_policy(self, V, gamma=0.9):
        """Extract greedy policy from value function."""
        pi = {}
        for s in range(self.mdp.S):
            q_values = [
                self.mdp.R[s, a] + gamma * sum(
                    self.mdp.T[s, a, sp] * V[sp]
                    for sp in range(self.mdp.S))
                for a in range(self.mdp.A)]
            pi[s] = np.argmax(q_values)
        return pi
```

The standard implementation above uses **synchronous** (or **Jacobi-style**) updates: all states are updated using the values from the previous iteration. An alternative is **Gauss-Seidel** updates, where we use newly computed values immediately within the same sweep.

Listing 1.2: Gauss-Seidel Value Iteration

```
import numpy as np

class GaussSeidelValueIteration:
    """Gauss-Seidel VI: use updated values immediately.
       """

    def __init__(self, mdp):
        self.mdp = mdp

    def run(self, theta=0.001, gamma=0.9):
        V = np.zeros(self.mdp.S)

        while True:
            delta = 0
            # Key difference: NO copy - use V directly
            for s in range(self.mdp.S):
                v_old = V[s]
                q_values = []
                for a in range(self.mdp.A):
                    # Uses already-updated V[s'] for s' <
                        s
                    q = self.mdp.R[s, a] + gamma * sum(
                        self.mdp.T[s, a, sp] * V[sp]
                        for sp in range(self.mdp.S))
                    q_values.append(q)
                V[s] = max(q_values)
                delta = max(delta, abs(v_old - V[s]))

            if delta < theta:
                break
        return V
```

Gauss-Seidel updates often converge faster because information propagates within a single sweep rather than waiting for the next iteration. However, the update order can affect convergence speed.

A more sophisticated variant handles **self-loops** explicitly. When a state can transition back to itself, the standard update mixes old and new values. The **Jacobi update with self-loop correction** solves for the fixed point analytically:

Listing 1.3: Jacobi Value Iteration with Self-Loop Handling

```python
import numpy as np

class JacobiValueIteration:
    """Jacobi VI with explicit self-loop handling."""

    def __init__(self, mdp):
        self.mdp = mdp

    def run(self, theta=0.001, gamma=0.9):
        V = np.zeros(self.mdp.S)

        while True:
            delta = 0
            V_old = V.copy()

            for s in range(self.mdp.S):
                v = V_old[s]
                q_values = []
                for a in range(self.mdp.A):
                    # Separate self-loop from other
                        transitions
                    sum_others = sum(
                        self.mdp.T[s, a, sp] * V_old[sp]
                        for sp in range(self.mdp.S) if sp
                            != s)
                    # Solve: V(s) = R + gamma*(T[s,a,s]*V
                        (s) + sum_others)
                    # => V(s)*(1 - gamma*T[s,a,s]) = R +
                        gamma*sum_others
                    denom = 1.0 - gamma * self.mdp.T[s, a
                        , s]
                    if denom > 0:
                        q = (self.mdp.R[s, a] + gamma *
                            sum_others) / denom
                    else:
                        q = self.mdp.R[s, a] + gamma *
                            sum_others
                    q_values.append(q)
                V[s] = max(q_values)
                delta = max(delta, abs(v - V[s]))

            if delta < theta:
                break
        return V
```

For large MDPs, **prioritized sweeping** can dramatically improve convergence by focusing computation on states with large Bellman errors:

Listing 1.4: Prioritized Sweeping Value Iteration

```
import numpy as np
import heapq

class PrioritizedSweepingVI:
    """VI with prioritized state updates."""

    def __init__(self, mdp):
        self.mdp = mdp
        # Build predecessor graph for efficient updates
        self.predecessors = {s: set() for s in range(mdp.
            S)}
        for s in range(mdp.S):
            for a in range(mdp.A):
                for sp in range(mdp.S):
                    if mdp.T[s, a, sp] > 0:
                        self.predecessors[sp].add(s)

    def run(self, theta=0.0001, gamma=0.9, max_iter=2000)
       :
        V = np.zeros(self.mdp.S)
        # Priority queue: (-priority, state)
        pq = []
        in_queue = set()

        # Initialize with Bellman errors
        for s in range(self.mdp.S):
            error = self._bellman_error(s, V, gamma)
            if error > theta:
                heapq.heappush(pq, (-error, s))
                in_queue.add(s)

        for _ in range(max_iter):
            if not pq:
                break
            _, s = heapq.heappop(pq)
            in_queue.discard(s)

            # Update V[s]
            V[s] = self._bellman_update(s, V, gamma)

            # Add predecessors with large errors to queue
            for p in self.predecessors[s]:
                error = self._bellman_error(p, V, gamma)
                if error > theta and p not in in_queue:
                    heapq.heappush(pq, (-error, p))
                    in_queue.add(p)
        return V

    def _bellman_update(self, s, V, gamma):
        return max(self.mdp.R[s, a] + gamma * sum(
            self.mdp.T[s, a, sp] * V[sp] for sp in range(
               self.mdp.S))
            for a in range(self.mdp.A))

    def _bellman_error(self, s, V, gamma):
        return abs(V[s] - self._bellman_update(s, V,
           gamma))
```

Prioritized sweeping is particularly effective when value changes are localized—updating one state primarily affects its neighbors. By maintaining a priority queue ordered by Bellman error magnitude, we focus computation where it matters most.

1.4 Policy Iteration

Policy iteration, introduced by Howard (1960), is an alternative dynamic programming algorithm that alternates between two steps: **policy evaluation** (computing V^π for the current policy) and **policy improvement** (finding a better policy using the current value function).

1.4.1 Overview

Policy iteration proceeds as follows. First, initialize with an arbitrary policy π_0. Then, in the **policy evaluation** step, compute V^{π_i} for the current policy. Next, in the **policy improvement** step, compute a better policy π_{i+1} by acting greedily with respect to V^{π_i}. Repeat until the policy stops changing.

1.4.2 Policy Evaluation

The policy evaluation step requires computing $V^\pi(s)$ for all states. For a fixed policy π, the Bellman expectation equation becomes:

$$V^\pi(s) = R(s, \pi(s)) + \gamma \sum_{s'} \mathcal{T}(s' \mid s, \pi(s)) V^\pi(s')$$

This is a system of $|\mathcal{S}|$ linear equations (no $\max$ operator), which can be solved in two ways.

The **iterative method** applies the Bellman policy operator repeatedly:

$$V^\pi_{k+1} = R^\pi + \gamma \mathcal{T}^\pi V^\pi_k$$

This converges because the policy operator $\mathcal{B}^\pi$ is also a γ-contraction.

The **direct method** solves the linear system exactly. In matrix form, $V^\pi = R^\pi + \gamma \mathcal{T}^\pi V^\pi$, which gives:

$$V^\pi = (I - \gamma \mathcal{T}^\pi)^{-1} R^\pi$$

This is exact but requires $O(|\mathcal{S}|^3)$ time for the matrix inversion.

1.4.3 Policy Improvement

Given V^{π_i}, we construct a new policy that is greedy with respect to the current value function. First, compute $Q^{\pi_i}(s, a)$ for all states and actions:

$$Q^{\pi_i}(s, a) = R(s, a) + \gamma \sum_{s'} \mathcal{T}(s' \mid s, a) V^{\pi_i}(s')$$

Then, define the improved policy:

$$\pi_{i+1}(s) = \arg\max_a Q^{\pi_i}(s,a) \quad \forall s \in \mathcal{S}$$

The intuition is straightforward: if taking action a and then following π_i yields higher value than just following π_i from the start, we should take action a.

1.4.4 Convergence

Theorem 6 (Policy Improvement Theorem). *Let π be a policy and π' be the greedy policy with respect to V^π. Then $V^{\pi'}(s) \geq V^\pi(s)$ for all $s \in \mathcal{S}$, with equality if and only if π is already optimal.*

Proof. For any state s:

$$\begin{aligned} V^\pi(s) &\leq \max_a Q^\pi(s,a) = Q^\pi(s,\pi'(s)) \\ &= R(s,\pi'(s)) + \gamma \sum_{s'} \mathcal{T}(s' \mid s,\pi'(s)) V^\pi(s') \\ &\leq R(s,\pi'(s)) + \gamma \sum_{s'} \mathcal{T}(s' \mid s,\pi'(s)) \max_{a'} Q^\pi(s',a') \\ &= R(s,\pi'(s)) + \gamma \sum_{s'} \mathcal{T}(s' \mid s,\pi'(s)) Q^\pi(s',\pi'(s')) \end{aligned}$$

Continuing this expansion:

$$V^\pi(s) \leq \mathbb{E}_{\pi'}\left[\sum_{t=0}^{\infty} \gamma^t r_{t+1} \mid s_0 = s\right] = V^{\pi'}(s)$$

□

Theorem 7 (Convergence of Policy Iteration). *Policy iteration converges to the optimal policy π^* in a finite number of iterations.*

Proof. By the policy improvement theorem, each iteration produces a policy that is at least as good as the previous one. Since there are only finitely many deterministic policies ($|\mathcal{A}|^{|\mathcal{S}|}$), and each improvement is strict unless we've reached optimality, the algorithm must terminate at π^*. □

In practice, policy iteration often converges in far fewer iterations than value iteration—sometimes just 2–3 iterations for small MDPs—because each iteration makes "larger" improvements by fully evaluating the current policy.

1.4.5 Implementation

Listing 1.5: Policy Iteration Implementation

```python
import numpy as np

def policy_iteration(mdp, gamma=0.9, epsilon=0.01):
    """Standard Policy Iteration algorithm."""
    V = np.zeros(mdp.S)
    policy = [0] * mdp.S

    while True:
        # Policy Evaluation
        while True:
            delta = 0.0
            for s in range(mdp.S):
                v = V[s]
                a = policy[s]
                V[s] = mdp.R[s, a] + gamma * sum(
                    mdp.T[s, a, sp] * V[sp] for sp in
                        range(mdp.S))
                delta = max(delta, abs(v - V[s]))
            if delta < epsilon:
                break

        # Policy Improvement
        policy_stable = True
        for s in range(mdp.S):
            old_action = policy[s]
            q_values = [
                mdp.R[s, a] + gamma * sum(
                    mdp.T[s, a, sp] * V[sp] for sp in
                        range(mdp.S))
                for a in range(mdp.A)]
            policy[s] = np.argmax(q_values)
            if old_action != policy[s]:
                policy_stable = False

        if policy_stable:
            return V, policy
```

1.4.6 Value Iteration vs. Policy Iteration

Both algorithms find the optimal policy, but they have different trade-offs. Value iteration has cost $O(|\mathcal{S}|^2|\mathcal{A}|)$ per iteration and requires many iterations (geometric convergence), but uses only $O(|\mathcal{S}|)$ memory and is easy to parallelize with asynchronous updates. Policy iteration has cost $O(|\mathcal{S}|^3)$ per iteration for exact evaluation (or $O(|\mathcal{S}|^2|\mathcal{A}|)$ for iterative evaluation) but requires few iterations (finite, often 2–10). It uses $O(|\mathcal{S}|^2)$ memory for exact evaluation and is harder to parallelize.

Strong Polynomiality. A deeper theoretical distinction concerns **strong polynomiality**—whether an algorithm can find an *exactly* optimal policy with computation polynomial in $|\mathcal{S}|$, $|\mathcal{A}|$, and $1/(1-\gamma)$, without dependence on a precision parameter δ. Ye (2011) proved that policy iteration is strongly polynomial, computing an optimal policy with $O(\text{poly}(|\mathcal{S}|, |\mathcal{A}|, 1/(1-\gamma)))$ arithmetic operations. Scherrer (2016) later provided a simpler proof showing that policy iteration terminates after at most $O(|\mathcal{S}||\mathcal{A}|/(1-\gamma))$ iterations.

In contrast, Feinberg et al. (2014) showed that value iteration is *not* strongly polynomial: there exist MDPs with just three states and two actions where value iteration requires arbitrarily many iterations to find the optimal policy exactly. The intuition is that value iteration can "hug" a suboptimal action indefinitely as values approach their limits asymptotically. Policy iteration avoids this by making discrete jumps between policies, allowing it to "snap" into the optimal solution.

The following discussion is paraphrased from the RL Theory lecture notes by Csaba Szepesvári (Szepesvári, 2024), which provide an excellent treatment of these complexity results. Available at `https://rltheory.github.io/`.

Practical Implications. Does this theoretical gap matter in practice? Often not. Value iteration achieves δ-suboptimal policies with cost growing only as $\log(1/\delta)$, which is quite mild. Moreover, in real applications we face sampling noise, function approximation error, and model mismatch—all of which make exact optimality unattainable anyway. As Szepesvári (2024) notes, "exact optimality is nice to have, but approximate computations with runtime growing mildly with the required precision should be almost equally acceptable."

In practice, **modified policy iteration** offers a middle ground: perform only a few iterations of policy evaluation (rather than running to convergence) before each policy improvement step. This combines the fast convergence of policy iteration with the computational efficiency of value iteration.

Bibliography

George Ainslie. Specious reward: A behavioral theory of impulsiveness and impulse control. *Psychological Bulletin*, 82(4):463–496, 1975. doi: 10.1037/h0076860.

Richard Bellman. *Dynamic Programming*. Princeton University Press, Princeton, NJ, 1957a.

Richard Bellman. A Markovian decision process. *Journal of Mathematics and Mechanics*, 6(5):679–684, 1957b. doi: 10.1512/iumj.1957.6.56038.

Paul F. Christiano, Jan Leike, Tom Brown, Miljan Martic, Shane Legg, and Dario Amodei. Deep reinforcement learning from human preferences. *Advances in Neural Information Processing Systems*, 30, 2017.

Jack Clark and Dario Amodei. Faulty reward functions in the wild. OpenAI Blog, December 2016. URL https://openai.com/index/faulty-reward-functions/.

Lasse Espeholt, Hubert Soyer, Remi Munos, Karen Simonyan, Volodymyr Mnih, Tom Ward, Yotam Doron, Vlad Firoiu, Tim Harley, Iain Dunning, Shane Legg, and Koray Kavukcuoglu. IMPALA: Scalable distributed deep-RL with importance weighted actor-learner architectures. In *International Conference on Machine Learning (ICML)*, pages 1407–1416, 2018.

Eugene A. Feinberg, Jefferson Huang, and Bruno Scherrer. Modified policy iteration algorithms are not strongly polynomial for discounted dynamic programming. *Operations Research Letters*, 42(6–7):429–431, 2014.

Dan Horgan, John Quan, David Budden, Gabriel Barth-Maron, Matteo Hessel, Hado van Hasselt, and David Silver. Distributed prioritized experience replay. In *International Conference on Learning Representations (ICLR)*, 2018.

Ronald A. Howard. *Dynamic Programming and Markov Processes*. The Technology Press of MIT and John Wiley & Sons, New York, 1960.

Leslie Pack Kaelbling, Michael L. Littman, and Andrew W. Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4: 237–285, 1996. doi: 10.1613/jair.301.

Devvrit Khatri, Lovish Madaan, Rishabh Tiwari, Rachit Bansal, Sai Surya Duvvuri, Manzil Zaheer, Inderjit S. Dhillon, David Brandfonbrener, and Rishabh Agarwal. The art of scaling reinforcement learning compute for LLMs. *arXiv preprint arXiv:2510.13786*, 2025.

Victoria Krakovna, Jonathan Uesato, Vladimir Mikulik, Matthew Rahtz, Tom Everitt, Ramana Kumar, Zac Kenton, Jan Leike, and Shane Legg. Specification gaming: The flip side of AI ingenuity. *DeepMind Blog*, 2020. URL `https://deepmind.com/blog/specification-gaming-the-flip-side-of-ai-ingenuity`.

David Laibson. Golden eggs and hyperbolic discounting. *The Quarterly Journal of Economics*, 112(2):443–477, 1997.

Xiuyuan Lu, Benjamin Van Roy, Vikranth Dwaracherla, Morteza Ibrahimi, Ian Osband, and Zheng Wen. Reinforcement learning, bit by bit. *Foundations and Trends in Machine Learning*, 16(6):733–865, 2023. doi: 10.1561/2200000097.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015. doi: 10.1038/nature14236.

Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International Conference on Machine Learning (ICML)*, pages 1928–1937, 2016.

OpenAI. OpenAI o1 system card. Technical report, OpenAI, December 2024. URL `https://cdn.openai.com/o1-system-card-20241205.pdf`.

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul F. Christiano, Jan Leike, and Ryan Lowe. Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems*, 35: 27730–27744, 2022.

Alexandre Piché, Ehsan Kamalloo, Rafael Pardinas, Xiaoyin Chen, and Dzmitry Bahdanau. PipelineRL: Faster on-policy reinforcement learning for long sequence generation. *arXiv preprint arXiv:2509.19128*, 2025.

Arthur L. Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, 3(3):210–229, 1959. doi: 10.1147/rd.33.0210.

Bruno Scherrer. Improved and generalized upper bounds on the complexity of policy iteration. *Mathematics of Operations Research*, 41(3):758–774, 2016.

John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In *International Conference on Machine Learning (ICML)*, pages 1889–1897, 2015.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

Wolfram Schultz, Peter Dayan, and P. Read Montague. A neural substrate of prediction and reward. *Science*, 275(5306):1593–1599, 1997. doi: 10.1126/science.275.5306.1593.

David Silver. Lectures on reinforcement learning. University College London, 2015. URL `https://www.davidsilver.uk/teaching/`.

David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016. doi: 10.1038/nature16961.

David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of Go without human knowledge. *Nature*, 550(7676):354–359, 2017. doi: 10.1038/nature24270.

David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science*, 362(6419):1140–1144, 2018. doi: 10.1126/science.aar6404.

David Silver, Satinder Singh, Doina Precup, and Richard S. Sutton. Reward is enough. *Artificial Intelligence*, 299:103535, 2021. doi: 10.1016/j.artint.2021.103535.

B. F. Skinner. *The Behavior of Organisms: An Experimental Analysis*. Appleton-Century, New York, 1938.

Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, second edition, 2018. ISBN 0262039249.

Csaba Szepesvári. Theoretical foundations of reinforcement learning: Lecture notes. University of Alberta, CMPUT 653, 2024. URL `https://rltheory.github.io/`.

Edward L. Thorndike. *Animal Intelligence: Experimental Studies*. Macmillan, New York, 1911.

Benjamin Van Roy. Foundations for reinforcement learning. Lecture Notes, MS&E 338: Aligning Superintelligence, Stanford University, 2024. URL `https://web.stanford.edu/class/msande338/notes/mse338_00_foundations_rl.pdf`.

Christopher J. C. H. Watkins and Peter Dayan. Q-learning. *Machine Learning*, 8(3–4):279–292, 1992. doi: 10.1007/BF00992698.

Christopher John Cornish Hellaby Watkins. *Learning from Delayed Rewards*. PhD thesis, King's College, Cambridge, 1989.

Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8(3–4):229–256, 1992. doi: 10.1007/BF00992696.

Yinyu Ye. The simplex and policy-iteration methods are strongly polynomial for the Markov decision problem with a fixed discount rate. *Mathematics of Operations Research*, 36(4):593–603, 2011.