Introduction to Reinforcement Learning

Lecture 1: What is RL?

Peter Henderson

COS 435 / ECE 433

Thanks to helpful slides/notes by Ben Eysenbach, Emma Brunskill, Ben Van Roy, and David Silver.

What is Reinforcement Learning?

What is Reinforcement Learning? Definitions.

Kaelbling, Littman & Moore (1996)

"Reinforcement learning is the problem faced by an agent that learns behavior through trial-and-error interactions with a dynamic environment."

Sutton & Barto (2018)

"more focused on goal-directed learning from interaction than are other approaches to machine learning."

Van Roy (2024)

"The subject of reinforcement learning addresses the design of agents that learn to achieve specified goals."

What is Reinforcement Learning? Definitions.

Welcome to the Era of Experience

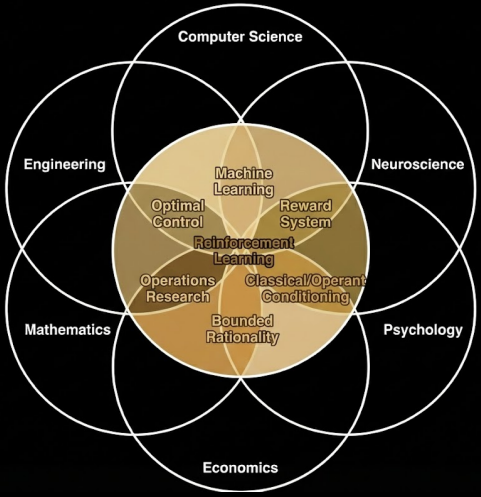
David Silver, Richard S. Sutton*

Abstract

We stand on the threshold of a new era in artificial intelligence that promises to achieve an unprecedented level of ability. A new generation of agents will acquire superhuman capabilities by learning predominantly from experience. This note explores the key characteristics that will define this upcoming era.

A brief history of RL

Many Faces of Reinforcement Learning



History of RL: Many Threads

Modern reinforcement learning weaves together **two threads** (among others):

1. **Optimal control** (1950s–): e.g., dynamic programming. *Largely no learning* — complete model assumed.
2. **Trial-and-error learning** (1890s–): from psychology and neuroscience (Thorndike, Skinner, Pavlov) *Learning from interaction in animals.*

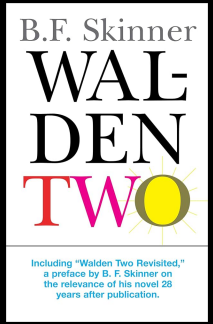
Many threads came together in the late from mid 1950s to late 1980s to form the field as we know it, with still lots of cross-over RL researchers across fields. *Source: Sutton & Barto, Ch. 1.6.*

History of RL: Psychology

Psychology — trial-and-error learning in animals:

- **Edward Thorndike** (1898): “Law of Effect.” Puzzle-box experiments with cats; responses that produce a satisfying effect become more likely, discomforting effect less likely.
- **B. F. Skinner** (1930s): Skinner box — buttons (actions), lights/speakers (observations), food/shocks (rewards). Operant conditioning.
- **Ivan Pavlov** (1890s): demonstrated classical conditioning by training dogs to salivate at the sound of a bell, tying the sound to food.

Skinner also wrote a novel about a society run by calculated reinforcement/conditioning of its citizens



B. F. Skinner, *Walden Two* (1948): a community run by behavioral engineering.

- Positive reinforcement only; behavior shaped by rewards and environment to build a utopian society.
- Controversial: free will, control, scaling behaviorism to society.

Early thought experiment on societal and ethical consequences of large-scale algorithmic reinforcement of human behavior. Something to think about.

History of RL: Neuroscience

Neuroscience — RL as a model of learning in the brain:

- Dopamine as reward prediction error; TD learning in the brain.
- Impacts of reward pathways on behavior, including depression, addiction, etc.
- Many neuroscientists do interdisciplinary work in RL. RL venues often have strong representation from neuroscience, psychology.

History of RL: Optimal Control and Dynamic Programming

Optimal control (late 1950s): design a controller to minimize cost over time.

- **James Clerk Maxwell** (1868): centrifugal governor — early control mechanism in hardware; spinning balls regulate engine speed.
- **Richard Bellman**: Bellman equation, **dynamic programming** (1957). Discrete stochastic $\Rightarrow$ **MDPs**.
- **Ron Howard** (1960): policy iteration for MDPs.
- DP remains a backbone of RL, but also a key tool in other fields like macroeconomics.

R. E. Bellman.

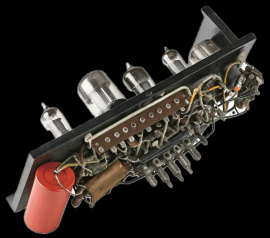
History of RL: Control Theory and RL

Connection: Control theory and RL address the same goal — an agent/controller acting in an environment to optimize long-term outcome — but are formulated differently:

- **Control theory**: often *continuous* time (integrals), *known* and *deterministic* dynamics.
- **RL**: often *discrete* time (summations), *unknown* or *stochastic* dynamics.

History of RL: Trial-and-Error in Early AI

- **Minsky et al.** (1954): Stochastic Neural Analog Reinforcement Calculator (SNARC) built at Princeton!
- 40 Hebb synapses, each holding the probability that signal comes in one input, with a hacked together mechanism for memory, including a surplus Minneapolis-Honeywell C-1 gyroscopic autopilot from a B-24 bomber.
- Provide a reinforcement signal to update the network and use it to solve a simulated maze, like reinforcement learning research with rats.



The last remaining neuron of SNARC.

History of RL: Arthur Samuel's Checkers (1959)

- **Arthur Samuel** at IBM: checkers program that **learned** to beat its creator.
- First program to learn from **self-play**.
- Key ideas later formalized as **temporal-difference learning**.
- Coined the term **"machine learning"**.

"Programming computers to learn from experience should eventually eliminate the need for much of this detailed programming effort." — Arthur L. Samuel, *Some Studies in Machine Learning Using the Game of Checkers*, 3 IBM J. Res. & Dev. 535 (1959).

Arthur Samuel (1901–1990).

History of RL: The 1970s–80s Revival

After a quiet period, RL research revived:

- **Harry Klopf** (1972–82): early temporal-difference learning ideas, learning from trial-and-error.
- **Sutton & Barto** (1981–88): **TD learning**, TD(λ), actor-critic.
- **Chris Watkins** (1989): **Q-learning** — model-free, off-policy.
- More!

By the 1990s the three threads merged into modern RL.

Sutton & Barto

History of RL: Deep RL Revolution (2013–present)

2013: DQN (DeepMind)

- Deep net + Q-learning, **raw pixels**
- Superhuman on many Atari games

Atari Breakout (DQN).

2016: AlphaGo

- Beat Lee Sedol at Go (10^{170} positions)
- **AlphaZero** (2017): zero human data

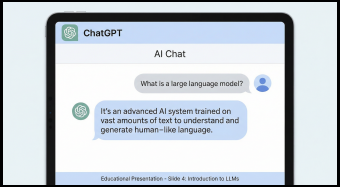
AlphaGo vs Lee Sedol.

RL + Language Models: The RL+LLM Era (2020s–present)

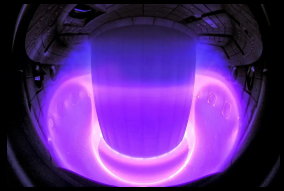
The RL+LLM Pipeline

1. Pre-train LLM on text
2. Collect human preferences or create RL environments
3. Fine-tune with RL to maximize the reward signal using the LLM as the starting point.

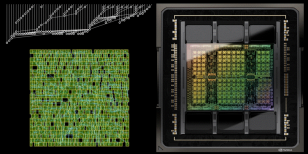
E.g.,: ChatGPT, GPT-4; Claude; Llama 2/3; Gemini; DeepSeek-R1; etc.



Other Real-world RL Uses



RL for fusion control (e.g., Degrave et al., 2022).



RL for chip design.

Believe it or not, bandit algorithms at IRS.

RL for robots.

How is RL different? What makes it hard? Why now?

How is RL Different from Other Approaches?

As you will see, you can reformulate many methods *to* and *from* the RL paradigm — but RL is typically distinct:

- **vs. supervised learning**: No labels for the "right" action; only a reward signal. Your actions affect the data you see next.
- **vs. control theory**: Dynamics and rewards are typically unknown; we learn from interaction, not a given model.
- **vs. plain optimization**: We optimize over *sequences* of decisions with delayed consequences, under uncertainty.

What Makes RL Hard? Why Haven't We Solved It Yet?

Four core challenges (we will revisit these later):

1. **Exploration** — How to gather useful experience?
2. **Delayed consequences** — Which past actions caused the reward? (credit assignment)
3. **Sample efficiency** — How to learn with limited data?
4. **Reward specification** — How to define "good" behavior?

Exploration vs Exploitation

The fundamental tradeoff in RL:

Exploitation

- Use what you know
- Take the best known action
- Greedy, safe

Exploration

- Try new things
- Gather information
- Risky, but might find better

Example

Restaurant choice: go to your favorite, or try something new that may or may not be better?

Credit Assignment

Problem: Which actions led to the reward?

- Rewards are often delayed
- A chess game has thousands of moves but one outcome
- How do we know which moves were good?

The Credit Assignment Problem

Determining how much each past action contributed to the current reward.

Sample Efficiency

Problem: RL often requires lots of data

- AlphaGo: millions of games of self-play
- Atari: billions of frames
- OpenAI Dactyl: 13,000 years of simulated experience

Challenge

Real-world interaction is expensive, slow, and sometimes dangerous. How can we learn efficiently?

Reward Specification

Problem: Specifying the “right” reward is hard, will optimize and find weird solutions.

Examples:

- [Video] Hit the target with the baseball. You assume, throwing the ball...
- [Video] Win at this racing game... By finishing the race?
- Win a capture the flag cybersecurity challenge, but successfully hacking... the evaluation docker instance?

The reward defines the problem. A poorly-specified reward leads to unintended behavior — the agent optimizes what you asked for, not what you meant.

Why is Now an Exciting Time to Work on RL?

- **RL + large models**: Large pre-trained models provide a useful starting point, enabling RL to work much more efficiently for open-ended domains.
- **Real-world impact**: Fusion control, chip design, data center cooling, robotics, healthcare, recommendation systems. RL is moving from games and sims into deployed systems.
- **Open problems**: Sample efficiency, safe exploration, reward design, and scaling RL to complex, long-horizon tasks are unsolved; there is lots of room to contribute.
- **Understanding self-driven intelligence**: Importantly, RL is also about a fundamental science of learning from experience, and general artificial intelligence, which still cannot compete with the sample efficiency and generalizabilty of human learning.

Course Goals

This course will give you the foundations to understand, implement, and extend modern RL algorithms and to engage with these challenges. As well as begin to engage you in thinking about the latest frontier research problems in RL.

Discussion - What are some areas/applications of RL that you are most excited about?

Course Logistics

- **Participation** — 15%
  Starting next week: Google form with in-class polling questions; breakout discussions on assigned papers; should submit reading reflection on the assigned papers with the marked up pdf of the paper.
- **Problem sets** — 15%
  3 assignments, due every other week starting in two weeks; small theory problems.
- **Programming assignments** — 20%
  3 assignments, starting in two weeks; small programming tasks.
- **Final project** — 50%
  Biggest one! Research project on a topic in RL; aim for academic workshop-level quality.

Getting in the Course

Fill out this Google Form if you're waiting, can't make any promises, but raised the cap: https://forms.gle/5siGARuazffRtFqu5

Please drop ASAP if you're not likely to take it so that we can let others in.

No formal auditing, but can sit in on lectures if there are seats.

Summary

Course Roadmap

This course will try to get very quickly (after policy-based RL) into advanced topics, often touching on RL with large langauge models. We will have a classic paper and a newer paper for dicussion each week.

1. **RL Basics**: bandits, policy and value iteration
2. **Value-Based RL**: Q-learning, DQN, and extensions
3. **Policy-Based RL**: REINFORCE, PPO, stability and convergence
4. **Model-Based vs Model-Free RL**: when to learn a model
5. **Advanced Topics**: actor-critic methods (SAC, TD3), reward specification
6. **Frontiers**: RLHF, offline RL, multi-agent RL

Philosophy

Ramp up from scratch to engaging with the frontiers of RL research in one semester, with emphasis on function approximation and deep RL.

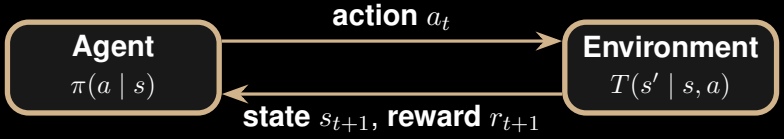
Resources

Resources will be posted after the class for the next week.

Break - 10 minutes

The Agent-Environment Interface

The Agent-Environment Interface

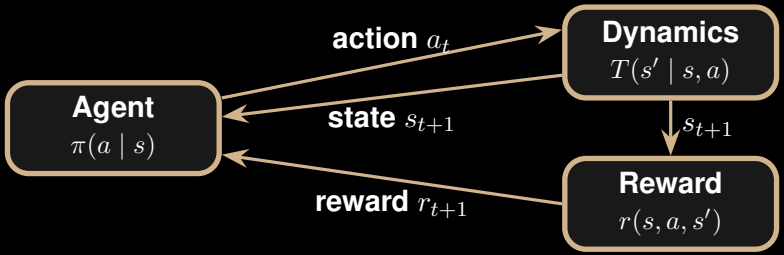


At each discrete time step t:

1. Agent observes state s_t and selects action a_t via policy π
2. Environment transitions to s_{t+1} via $T(s_{t+1} \mid s_t, a_t)$
3. Reward function emits r_{t+1}; agent uses $(s_t, a_t, r_{t+1}, s_{t+1})$ to update

Convention: r_{t+1} is the reward received *after* taking action a_t (Sutton & Barto).

Reward as a Separate Function



Conceptually can think of reward as separate from the environment and its dynamics since we might add things like curiousity bonuses, etc.

Key Components

- **State** $s \in \mathcal{S}$: the current situation
- **Action** $a \in \mathcal{A}$: what the agent can do
- **Reward** $r(s, a) \in \mathbb{R}$: scalar feedback signal
- **Transition dynamics** $T(s' \mid s, a)$: how the environment evolves
- **Policy** $\pi(a \mid s)$: the agent's strategy

Key Assumption

Both the reward function r and dynamics T are unknown to the agent. Experience is organized into **episodes** (trajectories): $\tau = (s_0, a_0, r_1, s_1, a_1, r_2, \ldots)$

The Objective

Goal: Maximize expected cumulative reward

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

Rewards

"All goals can be described by the maximisation of expected cumulative reward."

— David Silver

Examples of reward signals:

- **Helicopter**: $+$reward for desired trajectory, $-$reward for crashing
- **Chess**: $+1$ win, -1 loss, 0 otherwise
- **Robot walking**: $+$forward progress, $-$falling
- **Portfolio management**: profit at each step

Is this always true?

Specifying the "right" reward is one of the hardest problems in RL.

The Discount Factor γ

How much weight do we put on rewards at different time steps?

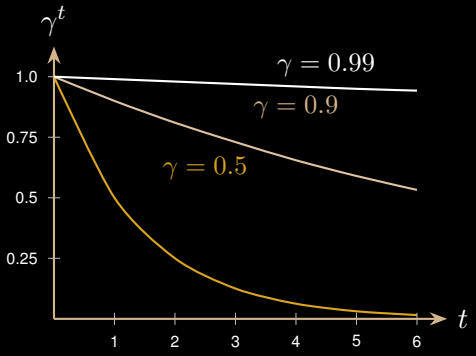
Do you care more about getting high rewards now or in the future?

Can also look at shorter temporal distances, discounting the future rewards:

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

where $\gamma \in [0, 1)$.

Visualizing the Discount Factor



Key observations:

- Higher weights on near-term rewards
- Lower weights on long-term rewards

Interpreting the Discount Factor

Rule of thumb: γ corresponds to reasoning $\frac{1}{1-\gamma}$ steps ahead

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

Why does this work?

The weights γ^t resemble a geometric distribution with parameter γ.
Such a distribution has expected value $\frac{1}{1-\gamma}$.

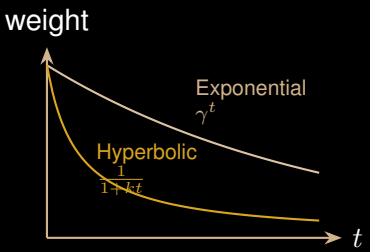
Why Discount? Reasons for $\gamma < 1$

1. **Mathematical convenience**
   - Ensures the sum $\sum_{t=0}^{\infty} \gamma^t r_t$ is finite
   - Required for infinite horizon problems
2. **Uncertainty about the future**
   - Model might be wrong far into future
   - Episode might terminate unexpectedly
3. **Preference for sooner rewards**
   - "A bird in the hand is worth two in the bush"
   - Models economic time preference

Note

Because $\gamma^t \to 0$ for large t, truncating the sum has little effect in practice.

Human Discounting: Not Quite Exponential



Humans exhibit hyperbolic discounting:

- Steeper drop for near-term
- Flatter for distant future
- Leads to time inconsistency

Example:

- Prefer \$100 today over \$110 tomorrow
- But prefer \$110 in 31 days over \$100 in 30 days

RL uses exponential discounting

Makes sure there is **time consistency**: optimal policy doesn't change as time passes.

The MDP Formalism

The Markov Decision Process (MDP)

MDP: The formal mathematical framework for RL

Markov Decision Process

An MDP is a tuple $(\mathcal{S}, \mathcal{A}, T, R, \gamma)$:

- $\mathcal{S}$: State space (all possible situations)
- $\mathcal{A}$: Action space (all possible actions)
- $T(s'|s, a)$: Transition dynamics (how environment evolves)
- $R(s, a)$: Reward function (scalar feedback)
- $\gamma \in [0, 1)$: Discount factor

Key Assumption in RL

Both T and R are unknown — the agent must learn from interaction.

The Markov Property

Markov Property

The future depends only on the present, not the past.

$$T(s_{t+1}|s_t, s_{t-1}, \ldots, s_0) = T(s_{t+1}|s_t)$$

Why: The current state contains all relevant information for predicting the future.

Examples

- Chess: board position is Markov
- Blackjack: need to track cards played (not Markov with just current hand, but can reformulate to be Markov)

Return: Cumulative Discounted Reward

The **return** G_t is the cumulative discounted reward from time 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 return is a random variable — depends on policy π, dynamics T, and rewards R.

Value Functions

State-Value Function $V^\pi(s)$: Expected return starting from s, following π
$V^\pi(s) = \mathbb{E}_\pi[G_t | s_t = s]$ *"How good is it to be in state s?"*

Action-Value Function $Q^\pi(s, a)$: Expected return starting from s, taking a, then following π
$Q^\pi(s, a) = \mathbb{E}_\pi[G_t | s_t = s, a_t = a]$ *"How good is it to take action a in state s?"*

Relationship

$V^\pi(s) = \sum_a \pi(a|s) Q^\pi(s, a)$ (For deterministic π: $V^\pi(s) = Q^\pi(s, \pi(s))$)

Optimal Value Functions and Policy

The **optimal value functions** are the best achievable:

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

Key Result

Given Q^*, the optimal policy is simple: $\pi^*(s) = \arg\max_a Q^*(s,a)$

Finding Q^* or V^* is the core of many RL algorithms!

Bellman Equations

Value functions satisfy a recursive relationship:

Value now = Immediate reward + Discounted future value

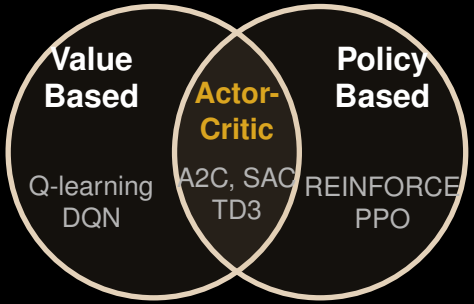
Bellman Expectation Equation (for policy π)

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

Bellman Optimality Equation

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

Categorizing RL Agents



Model-Free
Learn directly from experience
(most of this course)

Model-Based
Learn a model, then plan
(DreamervX, many robotics settings)

RL Terminology: State and Action Spaces

State Spaces

- **Discrete/Finite**: Countable states (e.g., board positions in chess)
- **Continuous**: $\mathcal{S} \subseteq \mathbb{R}^n$ (e.g., robot joint angles)
- **Tabular**: Small discrete $\mathcal{S}$ — can store $V(s)$ for every s in a table

Action Spaces

- **Discrete**: Finite choices (e.g., left/right/jump)
- **Continuous**: $\mathcal{A} \subseteq \mathbb{R}^m$ (e.g., torques, forces)
- **Control**: Often implies continuous actions/states (from control theory)

Why This Matters

- **Tabular methods**: Exact solutions, but don't scale to large/continuous spaces
- **Function approximation**: Use neural nets to generalize across states — required for most real problems

Computing Optimal Policies: Dynamic Programming

How Do We Find the Optimal Policy in Tabular MDPs?

Given an MDP $(\mathcal{S}, \mathcal{A}, T, R, \gamma)$, how do we compute π^*?

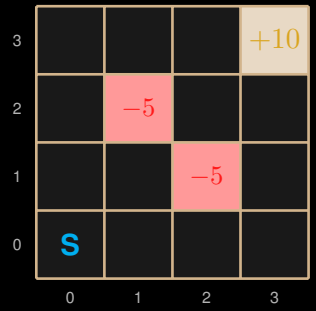
Two classic dynamic programming algorithms:

1. **Policy Iteration**: Evaluate a policy, then improve it. Repeat.
2. **Value Iteration**: Iteratively compute optimal values directly.

Assumption

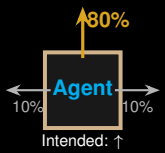
These algorithms assume we know the MDP (dynamics T and rewards R). Later we'll learn methods that don't require this.

Example: Grid World MDP



- **States:**
  $\mathcal{S} = \{(x, y) : x, y \in \{0, 1, 2, 3\}\}$
  16 grid positions
- **Actions:** $\mathcal{A} = \{\uparrow, \downarrow, \leftarrow, \rightarrow\}$
- **Rewards:**
  - Goal $(3, 3)$: $+10$
  - Hazards: -5
  - Step cost: -0.04
- **Discount:** $\gamma = 0.9$

Grid World: Stochastic Dynamics



"Slippery" dynamics:

- 80% move in intended direction
- 10% slip to each perpendicular
- Hitting wall ⇒ stay in place

Why stochastic dynamics?

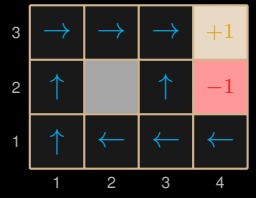
- Models real-world uncertainty (wind, slippery surfaces, motor noise)
- Makes planning non-trivial — can't just find shortest path
- Agent must account for risk of ending up in bad states

Transition Function

If action is ↑ in state s: $T(s_{\text{above}}|s,\uparrow) = 0.8$, $T(s_{\text{left}}|s,\uparrow) = 0.1$, $T(s_{\text{right}}|s,\uparrow) = 0.1$

Solving MDPs: What We Want

- In an MDP, we want an optimal policy $\pi^* : \mathcal{S} \to \mathcal{A}$
  - A policy π gives an action for each state
- An optimal policy maximizes expected sum of rewards
- **Contrast:** In deterministic planning, want an optimal plan (sequence of actions from start to goal)



Example: Optimal policy for a 3×4 grid world

How Many Policies Are There?

Grid World: 16 states, 4 actions

Question: How many deterministic policies exist?

- Each state needs an action assignment
- $|\mathcal{A}|$ choices per state, $|\mathcal{S}|$ states
- Total: $|\mathcal{A}|^{|\mathcal{S}|}$ deterministic policies

Grid World Answer

$4^{16} = 4{,}294{,}967{,}296$ deterministic policies (over 4 billion!)

Scaling Problem

Even small MDPs have exponentially many policies. We need efficient algorithms — not brute-force search!

MDP Control: Finding the Optimal Policy

Goal: Compute the optimal policy

$$\pi^*(s) = \arg\max_\pi V^\pi(s)$$

Naive Approach: Policy Search

Enumerate all $|\mathcal{A}|^{|\mathcal{S}|}$ policies, evaluate each, pick best.
Far too slow! We need dynamic programming algorithms.

Value Iteration: Key Idea

Idea: Iteratively compute optimal values for increasingly long horizons.

Key Idea

Maintain $V_k(s)$ = optimal value if you have k steps left to act.
Iterate to consider longer and longer horizons until convergence.

Intuition:

- $V_0(s) = 0$ (no steps left $\Rightarrow$ no reward)
- $V_1(s) = \max_a R(s, a)$ (one step: just get immediate reward)
- $V_k(s)$ builds on V_{k-1} (optimal k-step value uses optimal $(k-1)$-step values)

Value Iteration: The Algorithm

Value Iteration Algorithm

Initialize: $V_0(s) = 0$ for all s

For $k = 0, 1, 2, \ldots$ until convergence (e.g., $\|V_{k+1} - V_k\|_\infty < \epsilon$):

For each state s:

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

Extract policy (after convergence or at any iteration):

$$\pi(s) = \arg\max_a \left[R(s,a) + \gamma \sum_{s'} T(s'|s,a) V(s') \right]$$

The Bellman Operator

Bellman operators offer concise notation for expressing value iteration as a single operation.

Bellman Optimality Operator $\mathcal{B} : \mathbb{R}^{|\mathcal{S}|} \mapsto \mathbb{R}^{|\mathcal{S}|}$

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

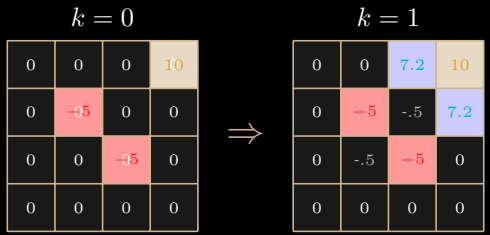
Value iteration is simply: $V_{k+1} = \mathcal{B} V_k$

Bellman Policy Operator $\mathcal{B}^\pi : \mathbb{R}^{|\mathcal{S}|} \mapsto \mathbb{R}^{|\mathcal{S}|}$

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

Policy evaluation: $V^\pi_{k+1} = \mathcal{B}^\pi V^\pi_k$

Value Iteration: First Iteration ($k = 0 \to k = 1$)



Terminal states (goal, hazards) have fixed values:

- $V(\text{goal}) = +10, \quad V(\text{hazard}) = -5$

Example: State $(2, 3)$ taking action $\to$ toward goal:

$$V_1(2,3) = \gamma \sum_{s'} T(s'|s, \to) \cdot V_0(s')$$

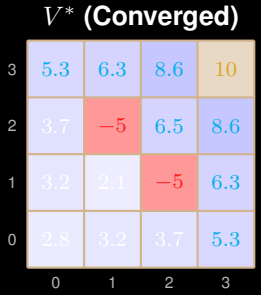
$$= 0.9 \times \big[\underbrace{0.8 \cdot 10}_{\text{reach goal}} + \underbrace{0.1 \cdot 0 + 0.1 \cdot 0}_{\text{slip}}\big]$$

$$= 0.9 \times 8 = \mathbf{7.2}$$

State $(1, 1)$ is adjacent to both hazards:

- Best action avoids hazards but risks slipping
- $V_1(1,1) = 0.9 \times 0.1 \times (-5) = -0.5$

Value Iteration: Converged Values



Key observations:

- Values "flow" outward from the goal
- Higher values $\Rightarrow$ closer/safer path to goal
- **State** $(1, 1)$: value 2.1 (lower than neighbors 3.2)
  - Adjacent to both hazards, risk of slipping

Optimal policy follows the value gradient:

- From $(0, 0)$: go $\uparrow$ (value $3.2 > 2.8$)
- From $(1, 1)$: go $\downarrow$ to avoid hazards
- From $(2, 3)$: go $\rightarrow$ to goal

Convergence

Converges in $\sim$16 sweeps ($\gamma = 0.9$). Code example in lecture notes.

Contraction Mapping Theorem

Theorem (Contraction Mapping)

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 $s \in \mathcal{S}$,

$$
\begin{aligned}
(\mathcal{B}V)(s) - (\mathcal{B}V')(s) &= \max_a \left[r(s,a) + \gamma \textstyle\sum_{s'} T(s'|s,a) V(s') \right] - \max_a \left[r(s,a) + \gamma \textstyle\sum_{s'} T(s'|s,a) V'(s') \right] \\
&\leq \max_a \left[r(s,a) + \gamma \textstyle\sum_{s'} T(s'|s,a) V(s') - r(s,a) - \gamma \textstyle\sum_{s'} T(s'|s,a) V'(s') \right] \\
&= \gamma \max_a \textstyle\sum_{s'} T(s'|s,a) (V(s') - V'(s')) \\
&\leq \gamma \max_{s'} |V(s') - V'(s')| = \gamma \|V - V'\|_\infty \quad \square
\end{aligned}
$$

Convergence of Value Iteration

Theorem (Convergence)

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

Proof: Recall $V^* = \mathcal{B}V^*$ (fixed point). For each k:

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

By induction: $\|V^* - V_k\|_\infty \leq \gamma^k \|V^* - V_0\|_\infty \to 0$ as $k \to \infty$. □

Convergence Conditions

Value iteration converges if: (1) $\gamma < 1$, OR (2) all policies reach a terminal state.

Asynchronous Value Iteration

Value iteration can be applied in a distributed and asynchronous manner — different states can be updated at different times, even with outdated values.

Theorem (Asynchronous Convergence)

Fix a finite MDP $(\mathcal{S}, \mathcal{A}, T)$, reward $r : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$, and $\gamma \in [0, 1)$. If $\mathcal{S}_0, \mathcal{S}_1, \ldots$ is a sequence of subsets of $\mathcal{S}$ such that each state $s \in \mathcal{S}$ appears infinitely often, then for any $\tilde{V}_0$, 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^*.

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

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

Because each state appears infinitely often, the error contracts for all states.

Intuition about Better Algorithms

Note

Understanding contraction mappings and other tricks for building intuition on convergent algorithms helps design better RL optimization methods. We'll see something similar again in policy gradients.

Discussion: Speeding Up Value Iteration

Turn to your neighbor and discuss:

Question

What strategies can we use to speed up convergence of value iteration?

Take 2–3 minutes to brainstorm with your neighbor.

Variants of Value Iteration

Gauss-Seidel VI: Update states in order, using new values immediately.

- When computing $V(s_i)$, use already-updated $V(s_1), \ldots, V(s_{i-1})$
- Often converges faster — new information propagates within an iteration
- Same convergence guarantees as standard VI

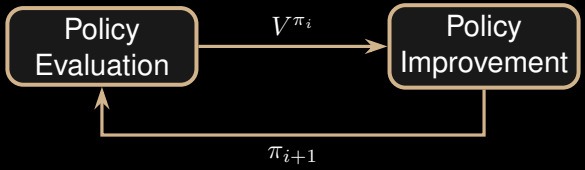
Asynchronous VI: Update states in any order, even in parallel.

- Each processor updates its own subset of states
- Converges as long as every state is updated infinitely often
- Great for distributed/parallel implementations
- Make a note of this! Modern RL for LLMs is all about throughput, async methods help a lot!

Policy Iteration: Overview

Idea: Alternate between evaluating and improving the policy.

1. **Initialize:** Start with arbitrary policy π_0
2. **Policy Evaluation:** Compute V^{π_i} for current policy
3. **Policy Improvement:** Compute better policy π_{i+1}
4. **Repeat** until policy stops changing



Policy Evaluation: Iterative Algorithm

Goal: Compute $V^\pi(s)$ for all states s

Iterative Policy Evaluation

Initialize $V_0(s) = 0$ for all s
For $k = 1, 2, \ldots$ until convergence:

$$V_k^\pi(s) = \sum_a \pi(a|s) \left[R(s,a) + \gamma \sum_{s' \in \mathcal{S}} T(s'|s,a) V_{k-1}^\pi(s') \right]$$

For a deterministic policy $\pi(s)$, this simplifies to:

$$V_k^\pi(s) = R(s, \pi(s)) + \gamma \sum_{s' \in \mathcal{S}} T(s'|s, \pi(s)) V_{k-1}^\pi(s')$$

Policy Improvement

Given V^{π_i}, how do we get a better policy?

Step 1: Compute $Q^{\pi_i}(s, a)$ for all states and actions:

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

Step 2: Act greedily with respect to Q^{π_i}:

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

Intuition

If taking action a then following π_i is better than just following π_i, we should take a!

Why Does Policy Iteration Converge?

Key insight: The greedy action is at least as good as the current policy.

$$\max_a Q^{\pi_i}(s,a) \geq Q^{\pi_i}(s,\pi_i(s)) = V^{\pi_i}(s)$$

Monotonic Improvement Theorem

$V^{\pi_{i+1}}(s) \geq V^{\pi_i}(s)$ for all states s.

Consequences:

- Policy iteration converges to optimal policy π^*
- Maximum $|\mathcal{A}|^{|\mathcal{S}|}$ iterations (number of policies is finite)
- In practice, converges much faster than that.

Policy Iteration: Full Algorithm

Policy Iteration Algorithm

Initialize: $\pi_0(s)$ arbitrarily for all s; set $i = 0$

Repeat:

1. **Policy Evaluation:** Compute V^{π_i} by iterating:
   $V_k^{\pi_i}(s) = R(s, \pi_i(s)) + \gamma \sum_{s'} T(s'|s, \pi_i(s)) V_{k-1}^{\pi_i}(s')$
   until convergence
2. **Policy Improvement:** For all $s \in \mathcal{S}$:
   $Q^{\pi_i}(s, a) = R(s, a) + \gamma \sum_{s'} T(s'|s, a) V^{\pi_i}(s')$
   $\pi_{i+1}(s) = \arg\max_a Q^{\pi_i}(s, a)$
3. $i \leftarrow i + 1$

Until: $\pi_i = \pi_{i-1}$ (policy unchanged)

Questions?

Next week's papers/reading

Next week's papers/reading will be posted after the class with an announcement on submitting the reading reflection. We will be moving beyond tabular methods pretty quickly and going straight into value-based function approximation methods.