RL4AA

Andrea Santamaria Garcia

RL4AA'25 DESY, Hamburg (02/04/2025)

Introduction to Reinforcement Learning

Disclaimer 🚨

This lecture:

- Is meant for people that are **new to RL.**
- Will introduce you to the **foundational concepts and ideas** used in RL.
- Will show you **the mathematical framework** that RL is based on.
  - it's a bit formula-heavy but bear with me 🧠!
- Will ***briefly*** introduce deep RL (modern RL).
- Will **not** teach you how to be a super deep RL coder (that's at least another lecture 😁).

If you reuse any of the material, please cite it 🙏

DOI:10.5281/zenodo.12649046

Contents

~30 min

1. **High level overview of RL**
2. **Trial-and-error concepts**
   - Agent, reward, and goal
   - Reward is enough
   - Trade-off between exploitation and exploration
   - Sequential decision making
3. **Optimal control concepts**
   - Markov decision processes (MDPs)
   - The Markov property
   - Partially observable Markov decision processes (POMDPs)
   - Reward and return
   - Policy
   - Value function
   - The Bellman equation

~30 min

4. **Gridworld toy problem**
   - Policy evaluation (exact)
   - Policy evaluation (iterative)
   - The reinforcement learning goal
   - How to get the best policy?
   - Bellman optimality equations
   - Policy improvement
   - About greedy actions
4. **Monte Carlo learning**
5. **Temporal difference learning**
6. **Off-policy learning**
7. **Deep reinforcement learning**

~5 min water break

Let's keep the questions for the breaks!

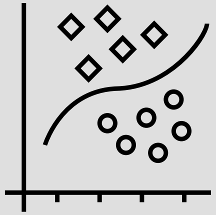
~25 min coffee break

MACHINE LEARNING

SUPERVISED LEARNING

Classification, prediction, forecasting

computer learns by example



Spam detection
Weather forecasting
Housing prices prediction
Stock market prediction

UNSUPERVISED LEARNING

Segmentation of data

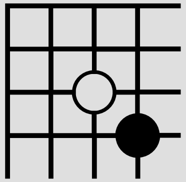
computer learns without prior information about the data

Medical diagnosis
Fraud (anomaly) detection
Market segmentation
Pattern recognition

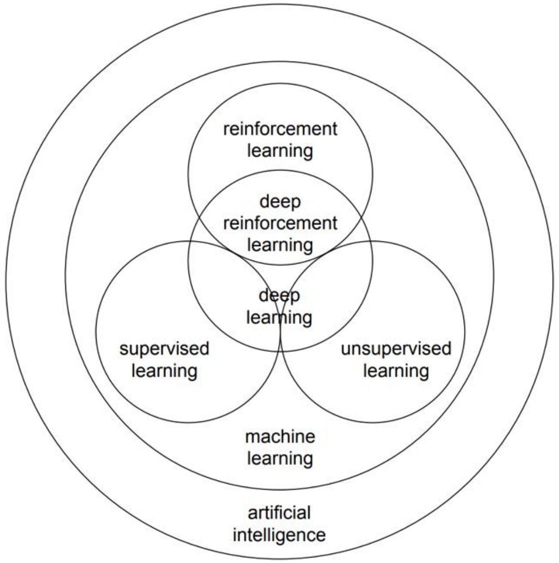
REINFORCEMENT LEARNING

Real-time decisions

computer learns through trial and error



Self-driving cars
Make financial trades
Gaming (AlphaGo)
Robotics manipulation



Reinforcement learning

More than machine learning

BEHAVIOUR LEARNING

Psychology (classical conditioning)
Neuroscience (reward system)
Economics (game theory)
Mathematics (operations research)
Engineering (optimal control, planning)

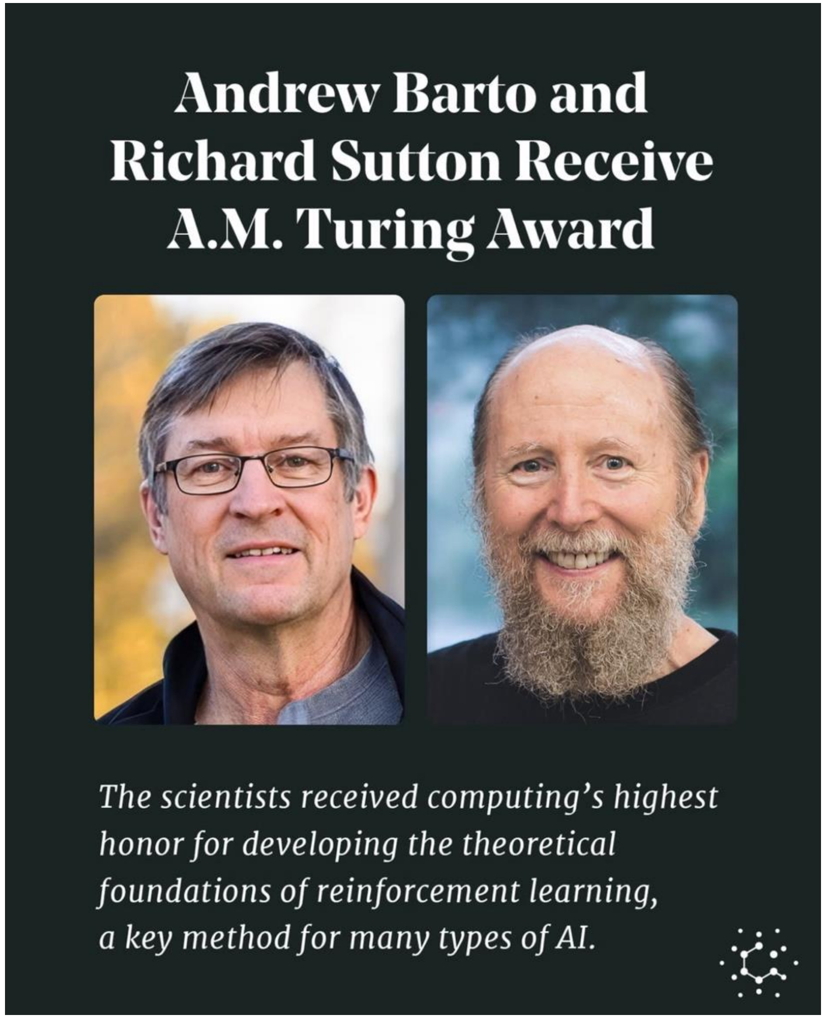
Deep reinforcement learning

Deep reinforcement learning opened the door to high dimensional environments

https://arxiv.org/abs/1707.02286

https://www.deepmind.com/publications/playing-atari-with-deep-reinforcement-learning

Reinforcement learning



"Reinforcement learning is simultaneously a problem, a class of solution methods that work well on the problem, and the field that studies this problem and its solution methods" (Sutton & Barto)

What we understand today as RL (established in the 1980s) inherits concepts from:

- **trial-and-error learning**
- **optimal control**
- **temporal difference learning**

The pillars of reinforcement learning

No deep RL just yet!

Trial-and-error learning

- Inefficient in biological systems! Requires many attempts.
- Pure trial-and-error is just random learning.

Provides the behavioural basis

- Learning emerges through repeated interaction, reward feedback, and adaptation.
- **Exploration vs exploitation dichotomy** inherent in trial and error.

Optimal control

- Computes best strategy and follows it efficiently.
- Relies on model to guide choices instead of random attempts.

Provides the mathematical framework

- Markov decision processes (**MDPs**), **Markov property, Bellman equation**, partially observable MDPs (POMDPs), **value** function, **policy** function, dynamic programming.

Temporal difference

- Efficient sample-based predictions.
- Online learning from experience without a model.

Provides scalability and adaptability for real-world problems

- Enables prediction and learning from **partial experiences**.
- Bootstraps rewards backward through actual experience → provides "foresight" for delayed rewards.

Trial-and-error concepts

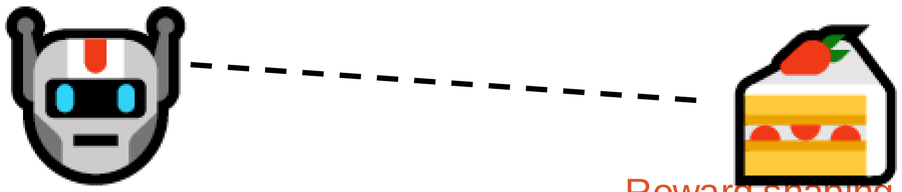
The RL problem: agent, goal, and reward

An agent must learn through trial-and-error interactions with a dynamic environment

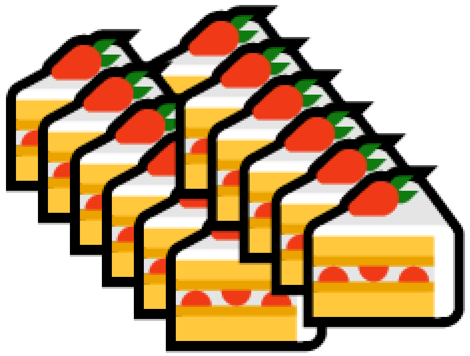
Agent
executes action
→ receives observation
→ receives scalar reward

Reward
scalar feedback signal r_t that indicates how well the agent is doing at step t

Goal
maximization of cumulative reward through selected actions

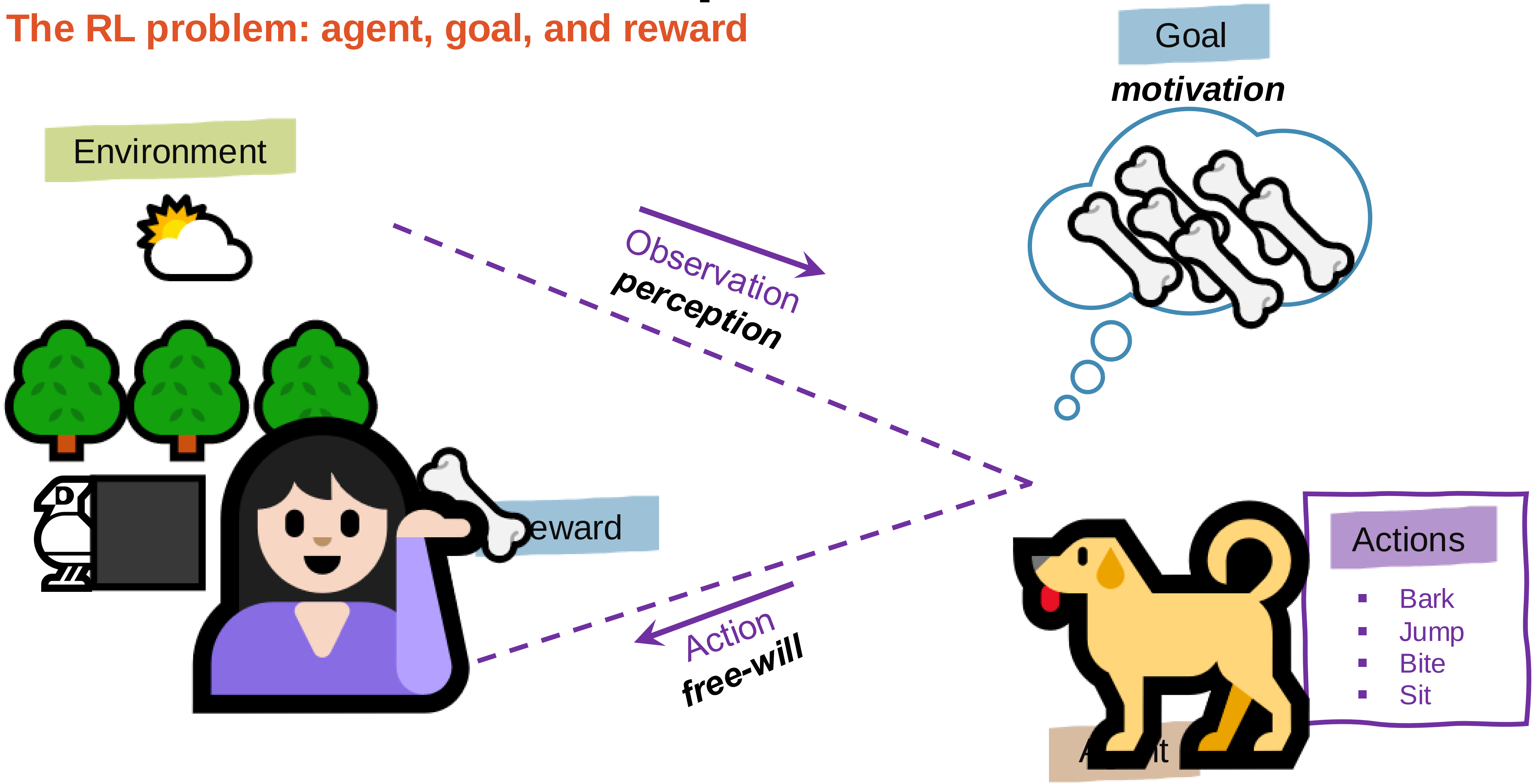


Reward shaping is non-trivial



Trial-and-error concepts

The RL problem: agent, goal, and reward



Trial-and-error concepts

The RL problem: agent, goal, and reward

"Reward is enough" by Silver et al. (2021) 

Proposes that the concept of reward maximization is a sufficient framework to achieve artificial general intelligence (AGI).

The authors argue that **complex intelligent behaviours** (such as perception, language, and social intelligence) **can emerge** from agents solely driven **by the goal of maximizing cumulative reward** in their environments.

- Some people argue that additional mechanisms, such as **intrinsic motivation, curiosity**, or **structured learning paradigms**, might be necessary to replicate the full spectrum of human intelligence.
- Nevertheless, **the single objective of reward maximisation has proven to be extremely powerful.**

"Scalar reward is not enough": a response to Silver et al. (2021)

Trial-and-error concepts

Trade-off between exploitation and exploration

- **Actions** may have **long-term consequences**
- **Reward** might be **delayed** (does not happen immediately)

Should the agent sacrifice immediate reward to gain more long term reward?

Trial-and-error concepts

Trade-off between exploitation and exploration

The agent needs to:

- ✓ <u>**Exploit**</u> what it has already experienced in order to obtain reward now.
- ✓ <u>**Explore**</u> the environment to select better actions in the future by sacrificing known reward now.

…and both cannot be pursued exclusively without failing at the task

<u>**Too much exploitation**</u>
the agent might converge prematurely to a suboptimal strategy

<u>**Too much exploration**</u>
the agent spends too much time testing bad actions, delaying convergence to an optimal strategy

Trial-and-error concepts

Trade-off between exploitation and exploration

- All RL algorithms are designed to deal with this trade-off by **assessing the value of actions** and estimating future reward.
- The **right balance** depends on the **problem, environment, and computational constraints.**

Finite vs. infinite horizons

if the learning time is **limited**, more exploitation is needed. In **long-term settings**, more exploration is feasible.

Deterministic vs. stochastic environments

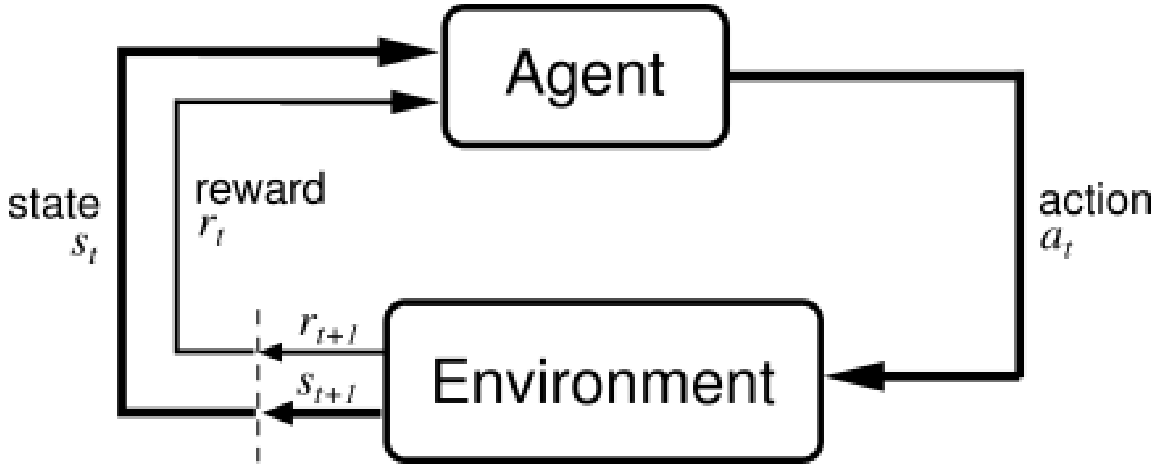
in highly stochastic settings, excessive exploration may be wasteful, while in deterministic ones, exploration can be minimized once a good policy is found.

Examples of different strategies:

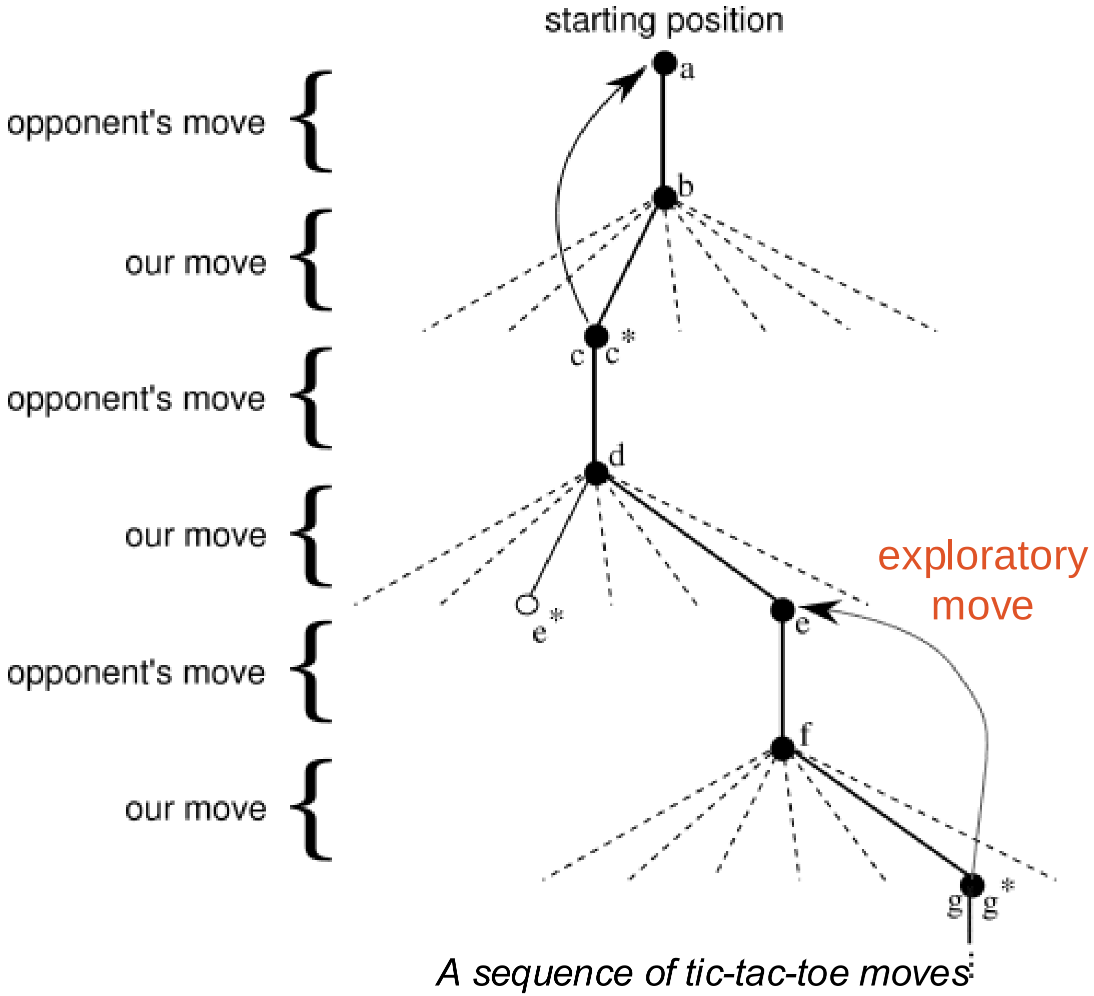
- explore early and exploit later using best-known action as learning progresses.
- better actions (with higher value) have a higher probability, but worse actions can still happen.
- choose actions with high uncertainty (under tested strategies are used until better understood).

Trial-and-error concepts

How to formalise sequential decision making?



The famous RL loop



A sequence of tic-tac-toe moves

Images from Sutton & Barto

Optimal control concepts 💡

Markov Decision Processes (MDPs)

A mathematical framework for modelling stochastic decision making

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

- Discrete or continuous
  Countable or real-valued $\mathcal{S}, \mathcal{A}$
- Finite or infinite
  Bounded or unbounded $\mathcal{S}, \mathcal{A}$
- Deterministic or stochastic $\mathcal{S}$, $\mathcal{R}$
- Episodic or continuing

$\mathcal{S}$ state space (all valid states)

$\mathcal{A}$ action space (all valid actions)

$\mathcal{R}$ reward function

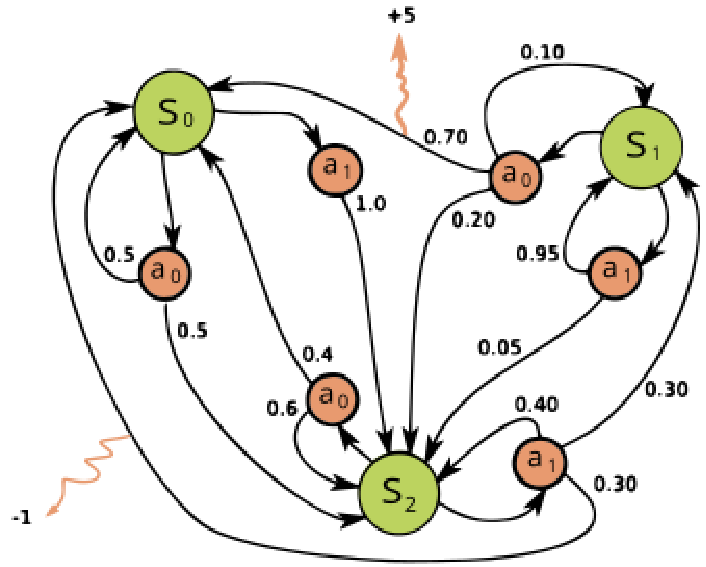
$r = \mathcal{R}(s, a, s') = \mathcal{R}_{ss'}^{a}$ Immediate reward

$\mathcal{P}$ transition probability function

$$\mathcal{P}_{ss'}^{a} = \mathbb{P}[s'|s, a]$$

Probability of transitioning to state s' after taking action a while being in state s

γ discount factor



MDP example from Wikipedia

$\mathcal{A} = \{a_0, a_1\}$; $\mathcal{S} = \{s_0, s_1, s_2\}$; $\mathcal{R}_{s_1 s_0}^{a_0} = +5$; $\mathcal{R}_{s_2 s_0}^{a_1} = -1$

$$\mathcal{P}_{ss'}^{a_0} = \begin{pmatrix} \mathcal{P}_{00} & \mathcal{P}_{01} & \mathcal{P}_{02} \\ \mathcal{P}_{10} & \mathcal{P}_{11} & \mathcal{P}_{12} \\ \mathcal{P}_{20} & \mathcal{P}_{21} & \mathcal{P}_{22} \end{pmatrix} = \begin{pmatrix} 0.5 & 0 & 0.5 \\ 0.7 & 0.1 & 0.2 \\ 0.4 & 0 & 0.6 \end{pmatrix}$$

Optimal control concepts 💡

The Markov property

What makes MDPs computationally tractable is the assumption of the **Markov property**

→ offers simplifications that considerably alleviates computational demands

- The Markov property states that **the system's next state is conditionally independent of all previous states given the current state**, or in other words, that **the future is independent of the past, given the present**.
- This property allows to discard the history of the process, making it **memoryless**.
- We can specify a set of conditional probabilities $\mathcal{P}_{ss'}^{a}$ of ending in state s' after taking action a while being in state s:

$$\mathcal{P} = \mathbb{P}[\, s_{t+1}, r_t | s_t, a_t, s_{t-1}, a_{t-1}, \ldots, a_0\,, s_0] \ = \mathbb{P}[s_{t+1}, r_t | s_t, a_t]$$

which are the entries $\mathcal{P}_{ss'}^{a}$ of the state transition probability function $\mathcal{P}$

Optimal control concepts 💡

The Markov property

Is the **Markov property** a reasonable assumption?

If we can observe the full state, **yes.**

Fully observable environments

The agent directly observes the true state of the environment, which includes everything relevant

state of the agent (belief)

$$\mathcal{O}_t = \mathcal{S}_t^a = \mathcal{S}_t^e$$

observation

true state of the environment

In real-world environments the agent receives partial observations

Partially observable environments

The agent receives partial observations and has to create its own state representation

$$\mathcal{O}_t \neq \mathcal{S}_t^a \neq \mathcal{S}_t^e$$

partial, noisy, filtered

Example: autonomous driving

$\mathcal{S}_t^e$: we know all cars exact positions, road friction, weather conditions, etc.

$\mathcal{O}_t$: pixels from cameras, GPS signal, lidar?

what the agent can "sense"

$\mathcal{S}_t^a$: estimated positions and speeds based on past observations

what the agent "believes" the environment is

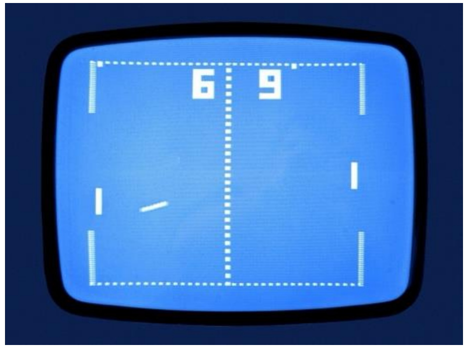
Optimal control concepts 💡

Partially observable Markov decision processes (POMDP)

A POMDP is a 7-tuple: $(\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, \Omega, \mathcal{O}, \gamma)$

$\mathcal{S}$	true state space (all valid states)
$\mathcal{A}$	action space (all valid actions)
$\mathcal{T}(s'\|s,a)$	transition probability function
$\mathcal{R}(s,a)$	reward function
Ω	observation space (all valid observations)
$\mathcal{O}(o\|s')$	observation probability function
γ	discount factor

Example: Atari pong



$\mathcal{S}_t^e$: we know ball and paddle positions and velocities

$\mathcal{O}_t$: one image frame

can't infer velocity

$\mathcal{S}_t^a$: estimated positions and speeds based on few last frames (frame stacking)

velocity inferred from pixel change

In a POMDP the observation o may not uniquely identify the true state s, so the agent must maintain a belief over possible states and update it over time (Bayes' rule)

Optimal control concepts 💡

Partially observable Markov decision processes (POMDP)

What is the consequence of maintaining a belief?

POMDPs are not memoryless like MDPS

The uncertainty introduced by partial observability is dealt with by keeping some past information

Stacking recent observations to approximate motion	Recurrent neural networks	Memory augmented (transformers)	Probabilistic reasoning

- All **real-world problems** are POMDPs
- They don't fulfill the Markov property, which means they are $\underbrace{\textbf{computationally intractable}}_{\text{no exact solution}}$

Optimal control concepts 💡

Reward distribution

In the previous slide we talked about the reward as deterministic but it is generally **stochastic** in **real-world environments**

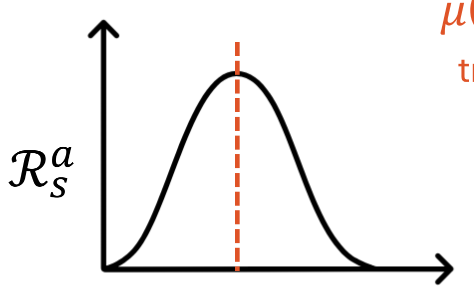
- The same action in the same state can lead to different rewards due to hidden variables
- The received reward is not fixed but rather sampled from a distribution

$$\mathcal{R}_s^a = \mathbb{P}[r|s,a]$$

Probability of receiving a reward r given s and a
Reward distribution or model

The reward distribution is often unknown, so we can:

- assume a distribution shape, collect samples, and **estimate distribution parameters** or
- **model the reward distribution explicitly** (model-based RL, Bayesian RL)



$\mu(s,a) = \mathbb{E}[r|s,a]$

true expected reward

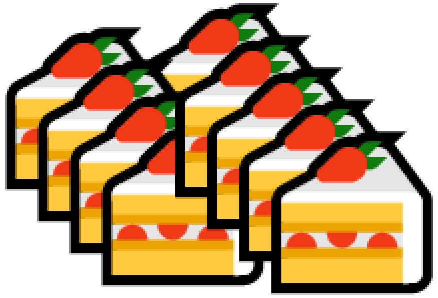
$\mu(s,a) + \varepsilon$

noisy sample

$$\mathcal{R} = \text{cost metric} + \underbrace{\text{environmental noise}}_{\text{what we model}}$$

Optimal control concepts

Return



The return is the total cumulated reward from a given step onward

Finite-horizon return

$$\mathcal{G}_t(\tau) = \sum_{k=0}^{T-t} r_{t+k}$$

- for a finite number of steps T
- for a given trajectory $\tau = (s_0, a_0, s_1, a_1, \ldots)$
- from timestep t

Infinite-horizon discounted return

$$\mathcal{G}_t(\tau) = \sum_{k=0}^{\infty} \gamma^k \, r_{t+k}$$

To ensure convergence when $T \rightarrow \infty$ the discount factor is introduced $\gamma \in [0,1)$

Intuition:  now is better than  later

Optimal control concepts 💡

Policy

The **policy function** is:

- a map from state to action
- completely defines how the agent will behave
- a distribution over actions given a certain state

$$\pi : \mathcal{S} \rightarrow \mathcal{A}$$

Deterministic: $\pi(s) = a$

Stochastic: $\pi(a|s) = \mathbb{P}[a|s]$

Probability of taking a specific action by being in a specific state

At every time step t:

→ The agent is in state s_t

→ The agent samples an action $a_t \sim \pi(a|s)$

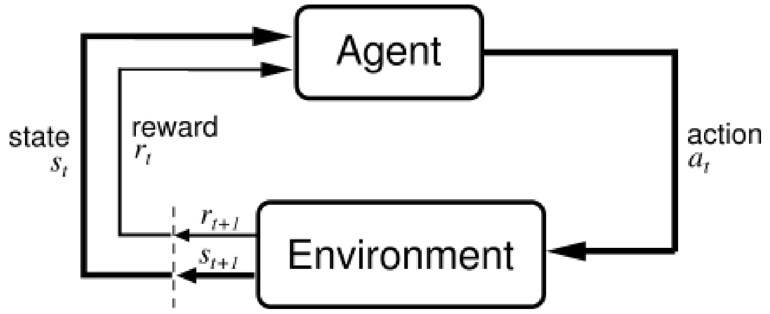
Sample randomly from a Gaussian dist. or from model

→ The environment samples:

→ Next state s_{t+1}

→ Reward r_t

Given by your simulation, experiment, or model



Optimal control concepts 💡

Value function

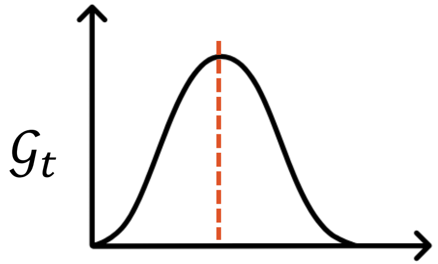
The **value function** is:

- an estimation of expected future reward, gives “value” to an action.
- used to choose between states depending on how much reward we expect to get.
- depends on the agent’s behaviour (policy → action).
- a way to compare policies.

State-value function Expected return starting from state s and following policy π (evaluates the policy)

$$\mathcal{V}^{\pi}(s) = \mathbb{E}_{\pi}[\mathcal{G}_t \mid \mathcal{S}_t = s]$$

given policy



Action-value function Expected return starting from state s, taking action a, and following policy π

”Q function”

$$\underbrace{\mathcal{Q}^{\pi}(s,a)}_{\text{where the return distribution is centered}} = \mathbb{E}_{\pi}[\mathcal{G}_t \mid \mathcal{S}_t = s, \mathcal{A}_t = a]$$

Optimal control concepts 💡

The Bellman equation

Decomposition of expected return into **immediate reward + expected future return**

$$\mathcal{G}_t = r_t + \gamma \mathcal{G}_{t+1}$$

Recursive structure where we can define the value of a state in terms of its successor states

$$\begin{aligned}\mathcal{V}^\pi(s) &= \mathbb{E}[\mathcal{G}_t \mid \mathcal{S}_t = s] \\ &= \mathbb{E}[r_t + \gamma\, r_{t+1} + \gamma^2\, r_{t+2} \ldots \mid \mathcal{S}_t = s] \\ &= \mathbb{E}[r_t + \gamma\, (r_{t+1} + \gamma\, r_{t+2} \ldots) \mid \mathcal{S}_t = s] \\ &= \mathbb{E}[r_t + \gamma\, \mathcal{G}_{t+1} \mid \mathcal{S}_t = s]\end{aligned}$$

$\mathbb{E}(\mathrm{f}) = \mathbb{E}(\mathbb{E}(\mathrm{f}))$

$$\boxed{\mathcal{V}^\pi(s) = \mathbb{E}[r + \gamma \mathcal{V}^\pi(s')]}$$

Optimal control concepts 💡

The expanded Bellman equation

In **stochastic environments** we need to take the expected value over all possibilities (actions, states):

$$\mathbb{E}_{a\sim\pi,\, s'\sim\mathcal{P}}[r + \gamma\, \mathcal{V}(s')]$$

We can expand the Bellman equation to explicitly account for it through the law of total expectation:

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

discrete case

Optimal control concepts 💡

The expanded Bellman equation

$$\boxed{\mathcal{V}^{\pi}(s) = \sum_{a \in \mathcal{A}} \pi\,(a|s) \sum_{s\prime \in \mathcal{S}} \mathcal{P}_{s,s\prime}^{a} \left(\mathcal{R}_{s}^{a} + \gamma \mathcal{V}^{\pi}(s')\right)}$$

How to solve it?

Approximation
Temporal-difference learning

Sampling
Monte Carlo methods

Iteration
Dynamic programming

Directly
System of $\mathcal{S}$ simultaneous linear equations with $\mathcal{S}$ unknowns

Computational complexity

Small 5 min break!

Richard Bellman

Gridworld toy problem

Let's use all the optimal control concepts we have learned and **solve the Bellman equation directly and exactly**

We will need:

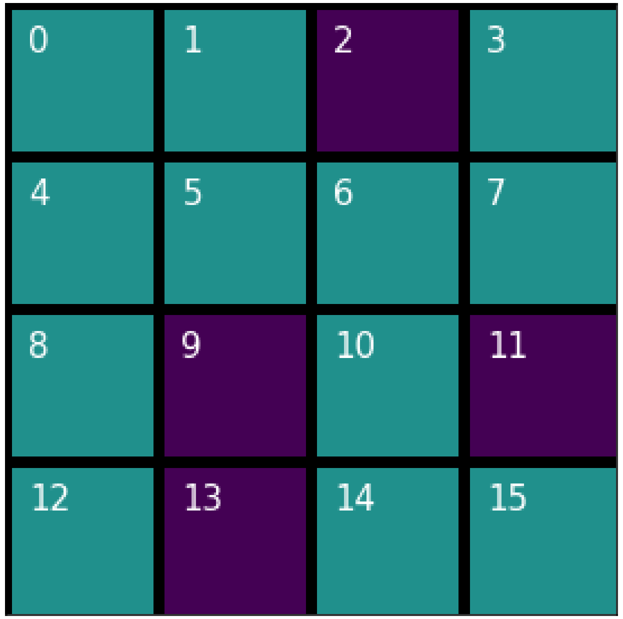
- A fully observable environment (MDP) → Markovian
- A small state space and action spaces $\mathcal{S}, \mathcal{A}$
- Know all transition probabilities $\mathcal{P}$

Welcome to gridworld!

$\mathcal{S} = (0, 1, 3, 4, 5, 6, 7, 8, 10, 12, 14, 15)$

$\mathcal{A} = (\uparrow, \downarrow, \leftarrow, \rightarrow)$

$\mathcal{P}^a_{s,s'} = 1$ Deterministic environment



Gridworld toy problem

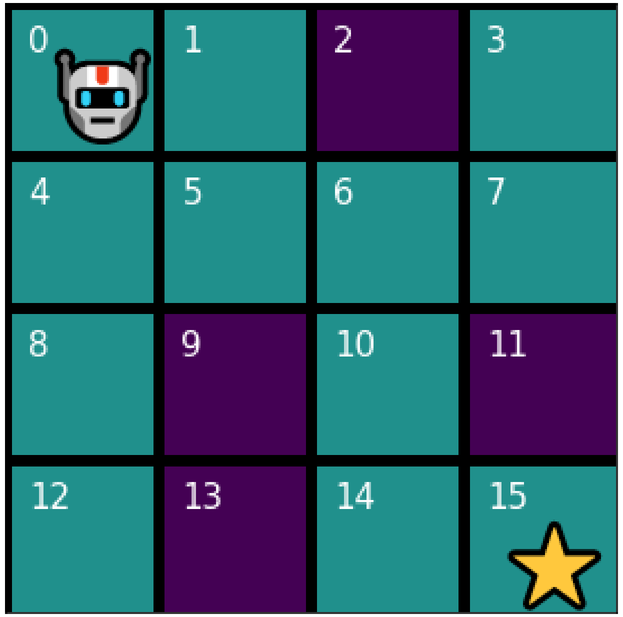
$\mathcal{S} = (0, 1, 3, 4, 5, 6, 7, 8, 10, 12, 14, 15)$

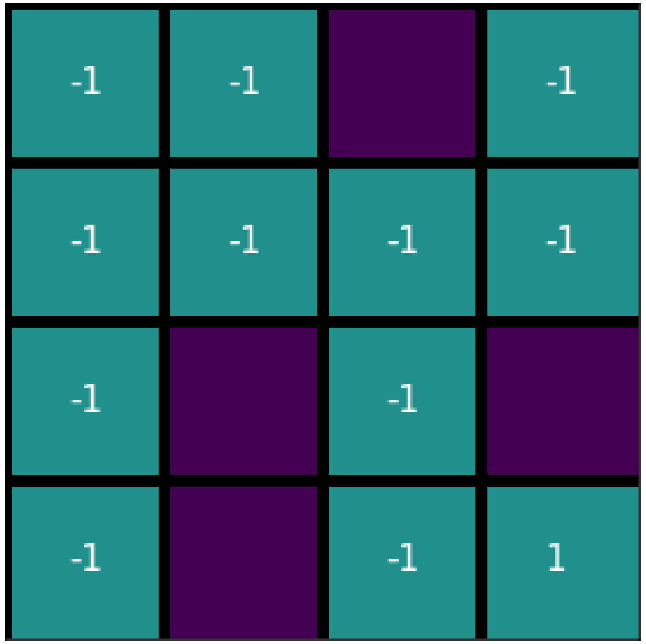
$\mathcal{A} = (\uparrow, \downarrow, \leftarrow, \rightarrow)$

$\mathcal{P}^{a}_{s,s'} = 1$ Deterministic environment

Our goal: get to state 15 (out of the maze)
Agent's goal: cumulate reward

Reward design: why negative?





$\mathcal{R}$

Gridworld toy problem

$$\mathcal{S} = (0, 1, 3, 4, 5, 6, 7, 8, 10, 12, 14, 15)$$

$$\mathcal{A} = (\uparrow, \downarrow, \leftarrow, \rightarrow)$$

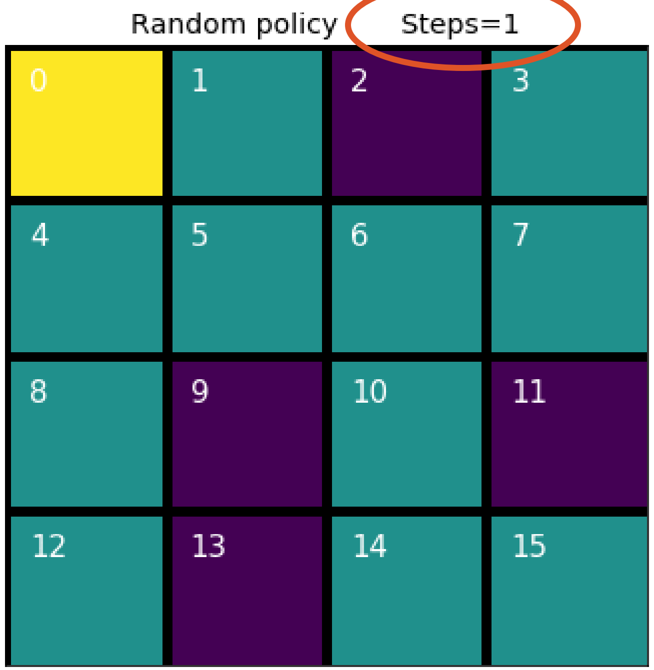
$\mathcal{P}^{a}_{s,s'} = 1$ Deterministic environment

$$\mathcal{R} = \begin{cases} -1 \; \forall s, s \neq 15 \\ 1 \; s = 15 \end{cases}$$

We need a policy: what is the simplest?

$$\pi(a|s) = \mathbb{P}[\uparrow, \downarrow, \leftarrow, \rightarrow \mid \mathcal{S}_t] = 0.25$$

Let’s see the **random policy** in action



Gridworld toy problem

$$\mathcal{S} = (0, 1, 3, 4, 5, 6, 7, 8, 10, 12, 14, 15)$$

$$\mathcal{A} = (\uparrow, \downarrow, \leftarrow, \rightarrow)$$

$\mathcal{P}^{a}_{s,s'} = 1$ Deterministic environment

$$\mathcal{R} = \begin{cases} -1 \; \forall s, s \neq 15 \\ 1 \; s = 15 \end{cases}$$

$$\pi(a|s) = \mathbb{P}[\uparrow, \downarrow, \leftarrow, \rightarrow \mid \mathcal{S}_t] = 0.25$$

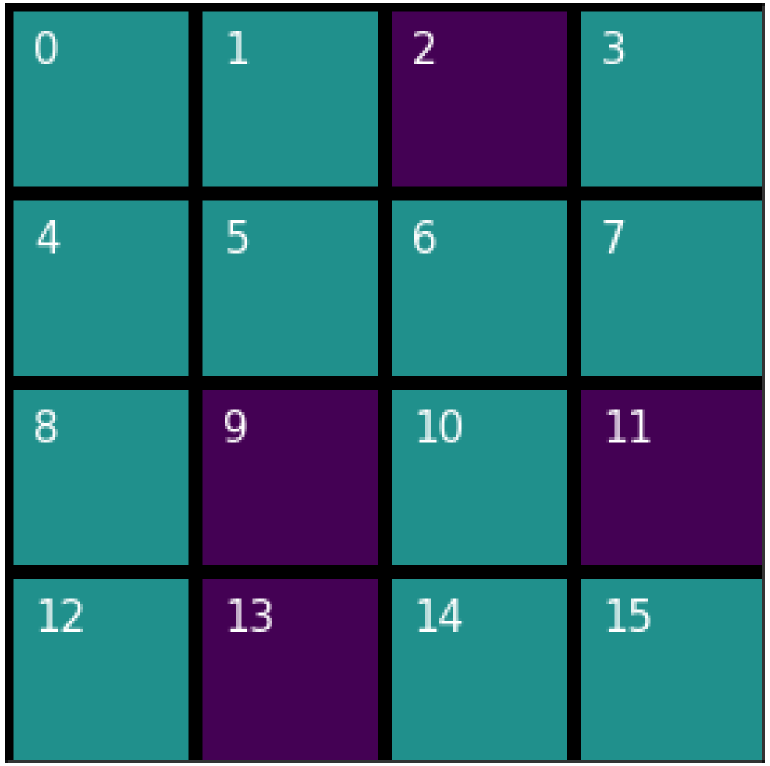
Let's solve our set of simultaneous equations

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

Brute force

```
0.5*v0 - 0.25*v1 - 0.25*v4 + 1.0 = 0
-0.25*v0 + 0.5*v1 - 0.25*v5 + 1.0 = 0
0.25*v3 - 0.25*v7 + 1.0 = 0
-0.25*v0 + 0.75*v4 - 0.25*v5 - 0.25*v8 + 1.0 = 0
-0.25*v1 - 0.25*v4 + 0.75*v5 - 0.25*v6 + 1.0 = 0
-0.25*v10 - 0.25*v5 + 0.75*v6 - 0.25*v7 + 1.0 = 0
-0.25*v3 - 0.25*v6 + 0.5*v7 + 1.0 = 0
-0.25*v12 - 0.25*v4 + 0.5*v8 + 1.0 = 0
0.5*v10 - 0.25*v14 - 0.25*v6 + 1.0 = 0
0.25*v12 - 0.25*v8 + 1.0 = 0
-0.25*v10 + 0.5*v14 + 0.5 = 0

11 variables,  11 equations
```



We can see this way of solving it won't scale with the number of states

Gridworld toy problem

The Bellman equation becomes an update rule:

$$\mathcal{V}^{\pi}(s) \leftarrow \sum_{a \in \mathcal{A}} \pi\,(a|s) \sum_{s' \in \mathcal{S}} \mathcal{P}^{a}_{s,s'} \left(\mathcal{R}^{a}_{s} + \mathcal{V}^{\pi}(s')\right)$$

Dynamic programming

- Initialise the value of all states to 0
- **For each state:**
  - Use $\mathcal{P}^{a}_{s,s'}$ to figure out the next possible states and the associated reward.
  - Calculate your value estimate for that state with the Bellman update rule:
    Average of those rewards from possible future states weighted by how likely each action is.
- Repeat loop for each state until values stop changing.

Computationally less expensive, but also won't scale

$$\mathcal{S} = (0, 1, 3, 4, 5, 6, 7, 8, 10, 12, 14, 15)$$

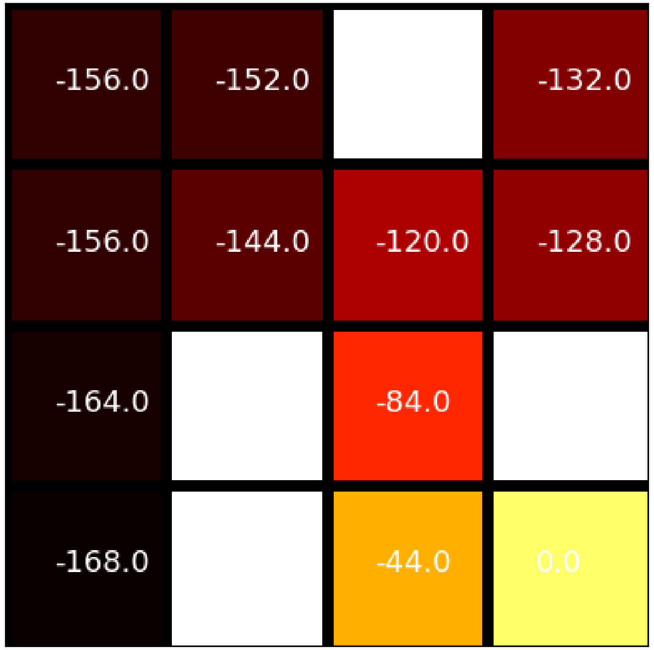
$$\mathcal{A} = (\uparrow, \downarrow, \leftarrow, \rightarrow)$$

$\mathcal{P}^{a}_{s,s'} = 1$ Deterministic environment

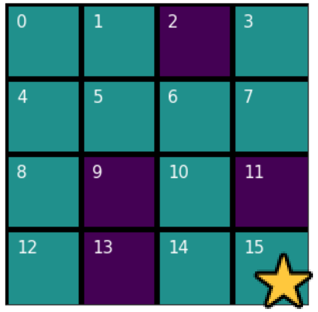
$$\mathcal{R} = \begin{cases} -1 \ \forall s, s \neq 15 \\ 1 \ s = 15 \end{cases}$$

$$\pi(a|s) = \mathbb{P}[\uparrow, \downarrow, \leftarrow, \rightarrow \mid \mathcal{S}_t] = 0.25$$

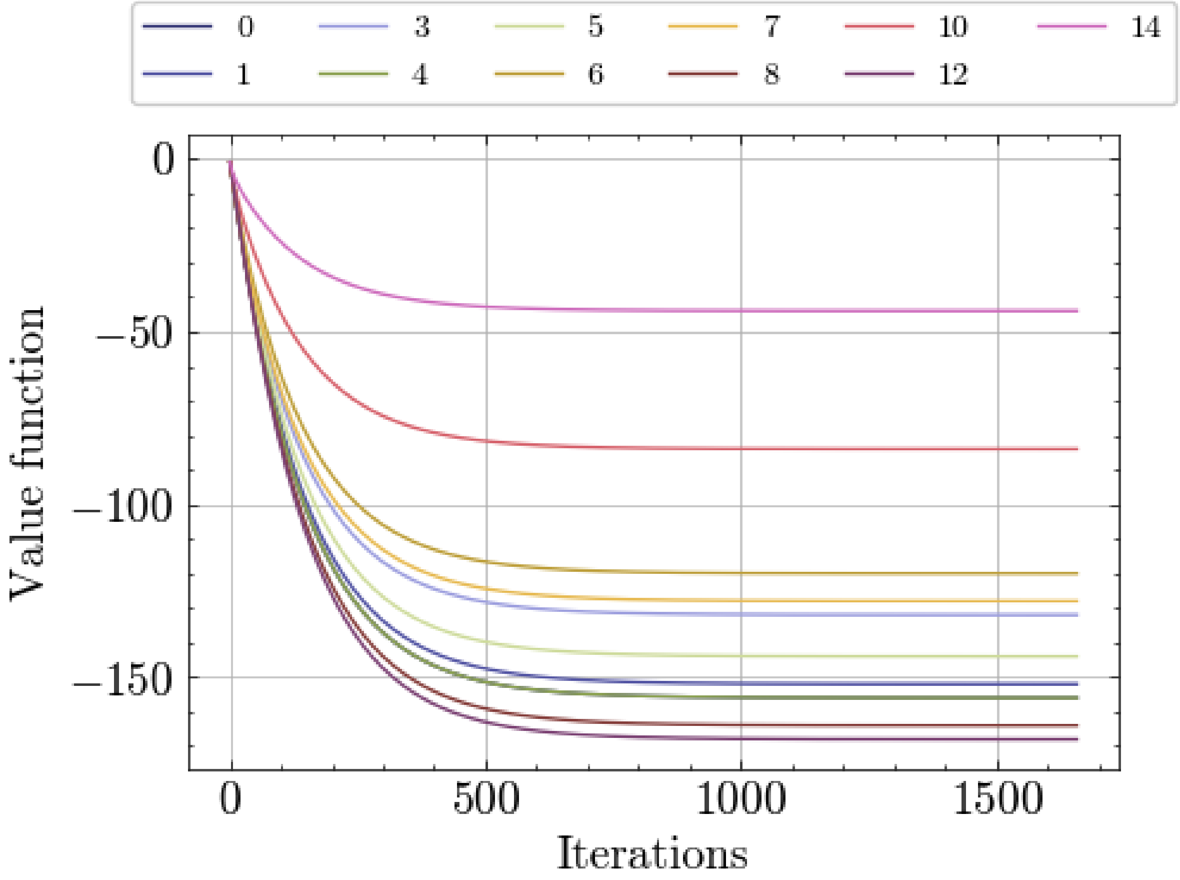
Value of random policy

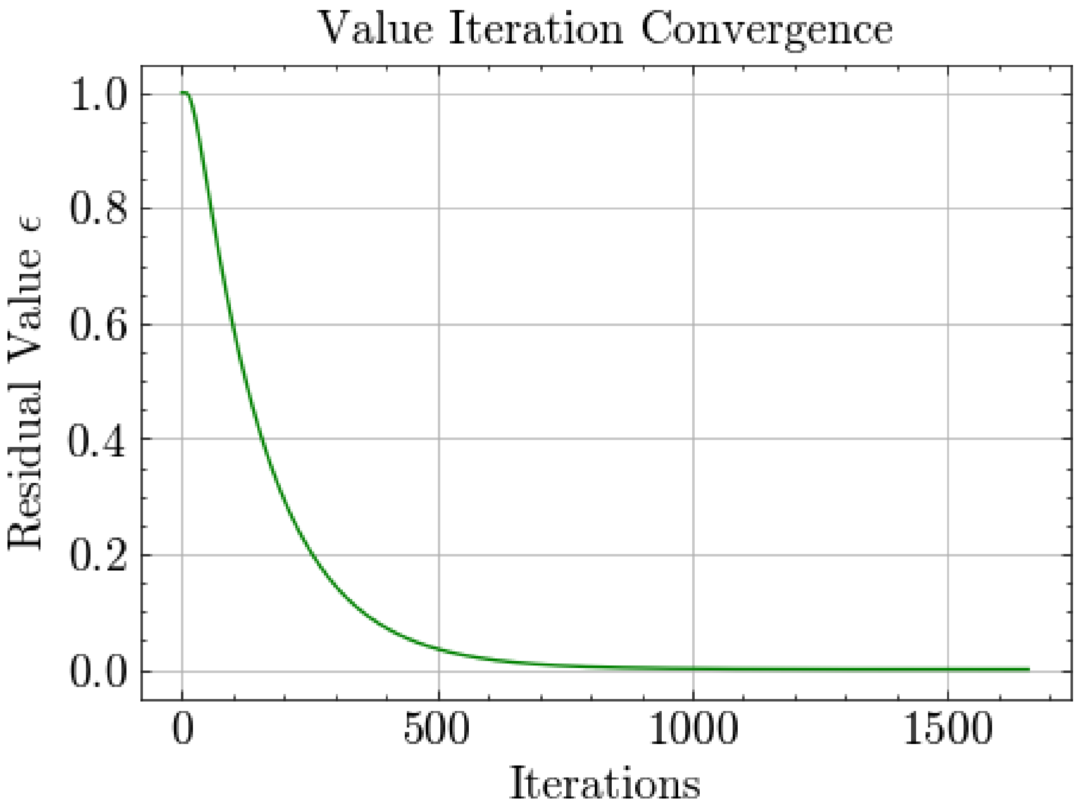


Gridworld toy problem



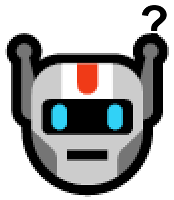
Policy evaluation with value iteration (dynamic programming)





What have we learned?

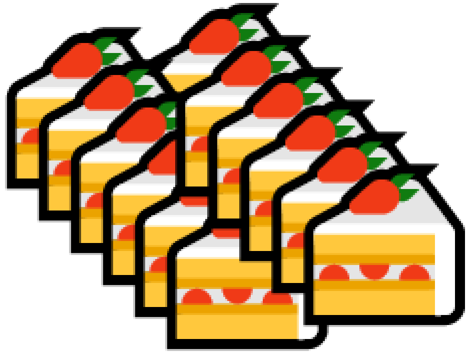
- **MDPs** formalise control problems by capturing the dynamics (transitions) and objectives (rewards).
- The **value function** tells us how good it is to be in each state and evaluate a policy.
- The **policy** represents the control strategy.
- The **Bellman equation** breaks down the global optimisation problem into local, recursive subproblems.
  - Turns a long-term planning problem into a set of local updates.
  - Enables both exact and approximate solutions.
  - Enables the computation of value functions and provides mathematical foundation to find the optimal policy.



But the **agent has not learned so far**! we have only evaluated the policy
Learning means updating your **policy**, your control strategy

The reinforcement learning goal

Goal
maximization of cumulative reward through selected actions



The expected return is:

$$J(\pi) = \mathbb{E}_{\pi}[\mathcal{G}_t]$$

Starting from time step t averaged over all possible trajectories induced by policy π

The optimisation problem can be expressed as:

$$\pi^* = \arg\max_{\pi} J(\pi)$$

where π^* is the **optimal policy**

The optimal policy will tell you the optimal action to take in each state

→ **the control problem is completely solved**

The reinforcement learning goal

Ideal setting

State fully observable

- MDP
- Model known
- Value function exact
- Optimal policy computable

We can completely solve the control problem and find the **optimal policy** $\boldsymbol{\pi}^*$

vs

Real world

State partially observable

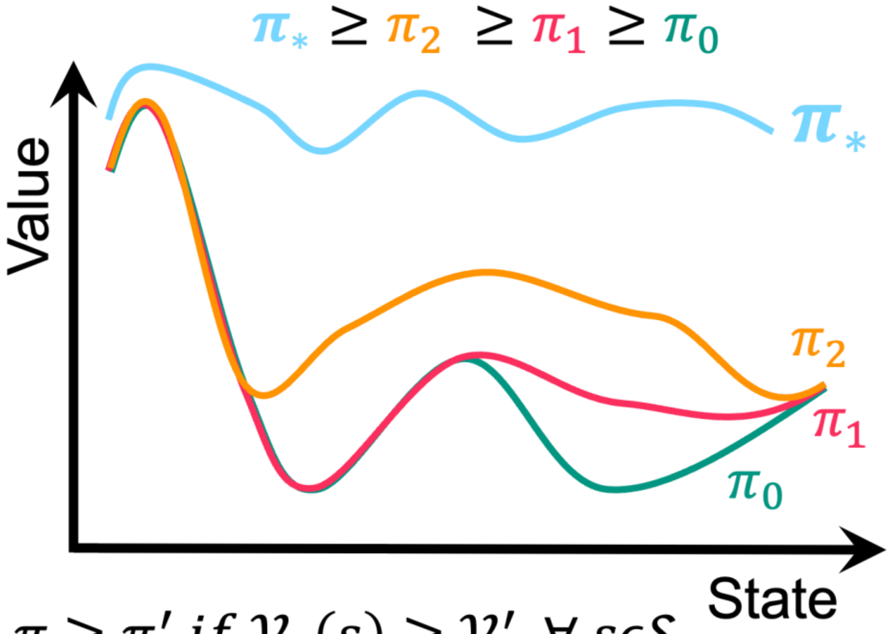
- POMDP
- Model unknown or learned
- Value function approximated
- Policy approximated

We just want **good-enough policies** that are robust, generalizable, sample-efficient, and safe

But how can we get the best policy?

For any MDP:

- There exists an optimal policy $\boldsymbol{\pi}^*$ that is better or equal to all other policies $\pi^* \geq \pi \ \forall \pi$
- All optimal policies achieve the optimal value function $\mathcal{V}^*$ and $\mathcal{Q}^*$



$\pi \geq \pi' \ if \ \mathcal{V}_\pi(s) \geq \mathcal{V}'_\pi, \forall \ s \epsilon \mathcal{S}$

So…do I have to calculate the value of every policy and compare them?

$|\mathcal{A}|^{|\mathcal{S}|}$ deterministic policies in an MDP

$4^{11} \approx 4$ million policies for simple gridworld example

Bellman optimality equations

All optimal policies achieve the optimal value function:

$$\mathcal{V}_{\pi}^{*}(s) = \max \mathcal{V}_{\pi}(s) \qquad \forall\ s \epsilon \mathcal{S}$$

$$\mathcal{Q}_{\pi}^{*}(s) = \max \mathcal{Q}_{\pi}(s) \qquad \forall\ s \epsilon \mathcal{S}, a \epsilon \mathcal{A}$$

- These equations define the value of a state under the optimal policy $\boldsymbol{\pi}^*$ the one that gives most total reward starting from any state.
- They tell you **how to act** if you want to get the **best possible future.**

$$\mathcal{V}^*(s) = \max_a \sum_{s\prime \in \mathcal{S}} \mathcal{P}_{s,s\prime}^{a} [\mathcal{R}_s^a + \gamma \mathcal{V}^*(s')]$$

- Policy is fixed
- Continuous action spaces
- How good is it to be in a state

$$\mathcal{Q}^*(s,a) = \sum_{s\prime \in \mathcal{S}} \mathcal{P}_{s,s\prime}^{a} [\mathcal{R}_s^a + \gamma \max_a \mathcal{Q}^*(s',a')]$$

- Want to know learn a policy
- Discrete action spaces (can enumerate actions
- How good is it to take an action from that state

Maximum value over every next possible state and action

We can use this!

Policy improvement

- Let's consider a non-optimal policy $\boldsymbol{\pi}$ and its value function $\boldsymbol{\mathcal{V}^{\pi}}$
- We can select an action that is greedy with respect to it to improve the policy

$$\pi'(s) = \arg\max_{a} Q^{\pi}(s,a)$$

Greedy action

next policy

$$= \arg\max_{a} \left(\mathcal{R}_s + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{s,s'} \mathcal{V}_{\pi}(s') \right)$$

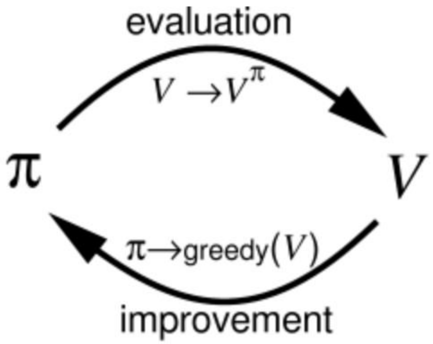
We have it from our policy evaluation

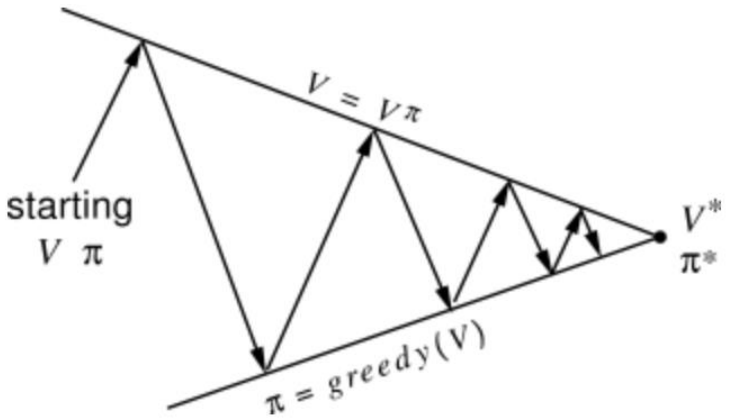
- **If the action has a higher value, the policy is better**
- $\mathcal{V}^*$ is the unique solution to the Bellman optimality eq.
- If this greedy operation does not change $\mathcal{V}$, then it converged to the optimal policy because it satisfies the Bellman optimality eq.

improvement

$$\pi_1 \rightarrow \mathcal{V}^{\pi 1} \rightarrow \pi_2 \rightarrow \cdots \rightarrow \pi_*$$

evaluation





Images from http://incompleteideas.net/book/ebook/node46.html

Gridworld toy problem

Dynamic programming

Policy improvement

$$\boldsymbol{\pi}^*(\boldsymbol{s}) = \arg\max_a \left(\mathcal{R}_s + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{s,s'} \mathcal{V}^*(s') \right) = \arg\max_a \mathcal{Q}^*$$

- Calculate the value for your current policy with value iteration (what we did before).
- **For each state:**
  - Look at the next possible states and their value.
  - Choose the action that will give you the maximum value and save it in an array.
- Repeat loop for each state until actions stop changing.

{0: 'right', 1: 'down', 3: 'down', 4: 'right', 5: 'right', 6: 'down', 7: 'left', 8: 'up', 10: 'down', 12: 'up', 14: 'right', 15: 0.0}

Takes one iteration in this case

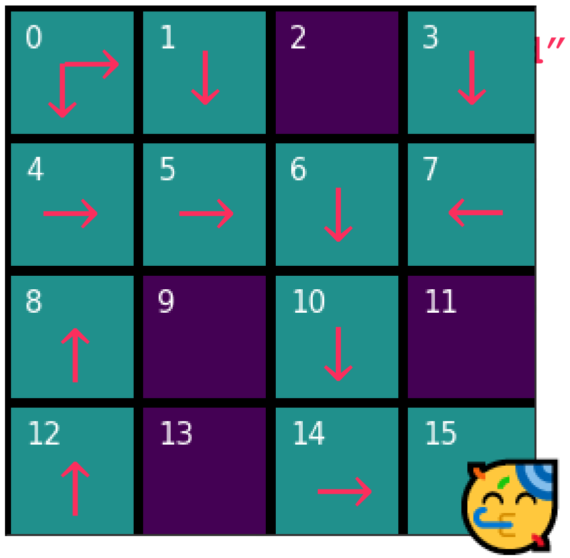
$\mathcal{S} = (0, 1, 3, 4, 5, 6, 7, 8, 10, 12, 14, 15)$

$\mathcal{A} = (\uparrow, \downarrow, \leftarrow, \rightarrow)$

$\mathcal{P}^a_{s,s'} = 1$ Deterministic environment

$$\mathcal{R} = \begin{cases} -1 \; \forall s, s \neq 15 \\ 1 \; s = 15 \end{cases}$$

$\pi(a|s) = \mathbb{P}[\uparrow, \downarrow, \leftarrow, \rightarrow \mid \mathcal{S}_t] = 0.25$



$\boldsymbol{\pi}^*$

Gridworld toy problem

Policy improvement

$$\boldsymbol{\pi}^*(\boldsymbol{s}) = \underset{a}{\operatorname{argmax}} \left(\mathcal{R}_s + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{s,s'} \mathcal{V}^*(s') \right) = \underset{a}{\arg\max}\, Q^*$$

$$\mathcal{S} = (0, 1, 3, 4, 5, 6, 7, 8, 10, 12, 14, 15)$$

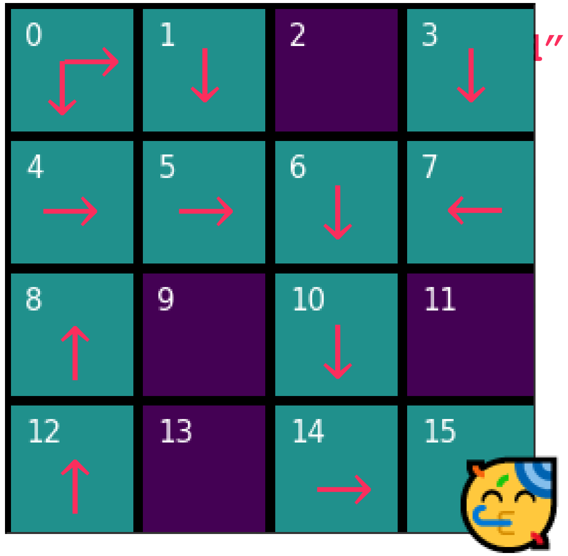
$$\mathcal{A} = (\uparrow, \downarrow, \leftarrow, \rightarrow)$$

$\mathcal{P}^a_{s,s'} = 1$ Deterministic environment

$$\mathcal{R} = \begin{cases} -1\ \forall s, s \neq 15 \\ 1\ s = 15 \end{cases}$$

$$\pi(a|s) = \mathbb{P}[\uparrow, \downarrow, \leftarrow, \rightarrow \mid \mathcal{S}_t] = 0.25$$

$\mathcal{Q}$	$\uparrow$	$\downarrow$	$\leftarrow$	$\rightarrow$
s_0	$\mathcal{Q}(s_0 \uparrow)$	$\mathcal{Q}(s_0, \downarrow)$	$\mathcal{Q}(s_0, \leftarrow)$	$\mathcal{Q}(s_0, \rightarrow)$
s_1	$\mathcal{Q}(s_1, \uparrow)$	$\mathcal{Q}(s_1, \downarrow)$	$\mathcal{Q}(s_1, \leftarrow)$	$\mathcal{Q}(s_1, \rightarrow)$
$\vdots$				
s_{14}	$\mathcal{Q}(s_{14}, \uparrow)$	$\mathcal{Q}(s_{14}, \downarrow)$	$\mathcal{Q}(s_{14}, \leftarrow)$	$\mathcal{Q}(s_{14}, \rightarrow)$



$\boldsymbol{\pi}^*$

{0: 'right', 1: 'down', 3: 'down', 4: 'right', 5: 'right', 6: 'down', 7: 'left', 8: 'up', 10: 'down', 12: 'up', 14: 'right', 15: 0.0}

Takes one iteration in this case

About greedy actions

: *Cool. So, if the <u>value function</u> gives "value" to an action…we just keeping choosing the action with more value every time! problem solved.*

: *Well, this only works if the environment is fully observable, and we know the model.*

In partially observable environments we have <u>**estimations**</u> of the values of the actions:

- $Q_t(s,a)$ → estimation
- $q_t^*(s,a)$ → exact

We want $|\boldsymbol{Q}(\boldsymbol{a}) - \boldsymbol{q}^*(\boldsymbol{a})|$ to be minimal

Example of value estimation: sample-average method

$$Q_t(a) \doteq \frac{\text{sum of rewards when } a \text{ taken prior to } t}{\text{number of times } a \text{ taken prior to } t}$$

$$\lim_{t \to \infty} Q_t(a) = q^*(a)$$

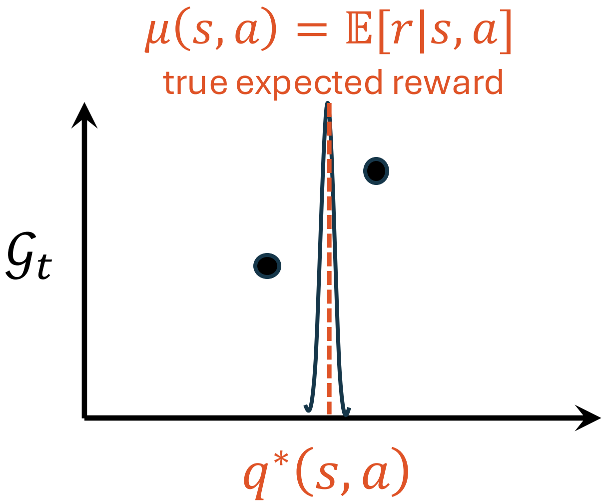
About greedy actions

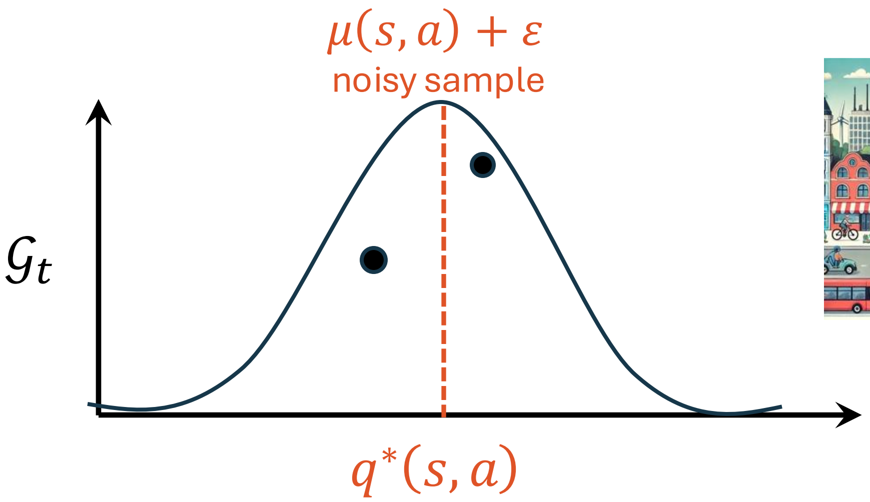
$$\mathcal{Q}_\pi(s,a) = \mathbb{E}_\pi[\mathcal{G}_t \,|s,a]$$

<u>Greedy action:</u> $a_t \doteq \arg\max_a \mathcal{Q}_t\,(s,a)$ select action with most value → pure exploitation

<u>Near-greedy action:</u> small probability ε to select randomly from all actions → ensures convergence

Does **<u>greedy action</u>** work? → it will depend on the uncertainties (noise)





$\mathcal{R}$ = cost metric + environmental noise

If $\sigma = 0$ you will know the value of each action after trying it once

If σ is large (noisy reward) you will need more **<u>exploration</u>**

Transitioning to modern RL

Ideal setting

State fully observable

- MDP (finite, discrete)
- Model known
- Value function exact
- Optimal policy computable

vs

Real world

State partially observable

- POMDP
- Model unknown or learned
- Value function approximated
- Policy approximated $\pi \approx \pi^*$

Classical dynamic programming

- Bellman equations + greedy action.
- Policy evaluation, policy improvement, value iteration.
- Non-tractable for large state and action spaces.

Modern RL (model free!)

- One sample does not return the true expected value (noisy reward).
- The same action does not always lead to the same next state.
- We don't know the true state (only observed).

Monte Carlo learning

- We have access to a black box model that we query (simulation or real-world).
- We get samples of trajectories.
- We don't know $\mathcal{P}$.

The experience is organised in episodes:

$s_0, a_0, r_1, s_1, a_1, r_2, \ldots, s_{T-1}, a_{T-1}, r_T$

Episode 1

$\vdots$

$s_0, a_0, r_1, s_1, a_1, r_2, \ldots, s_{T-1}, a_{T-1}, r_T$

Under policy π

Episode M

Value estimation $\mathcal{V}^{\pi}(s)$

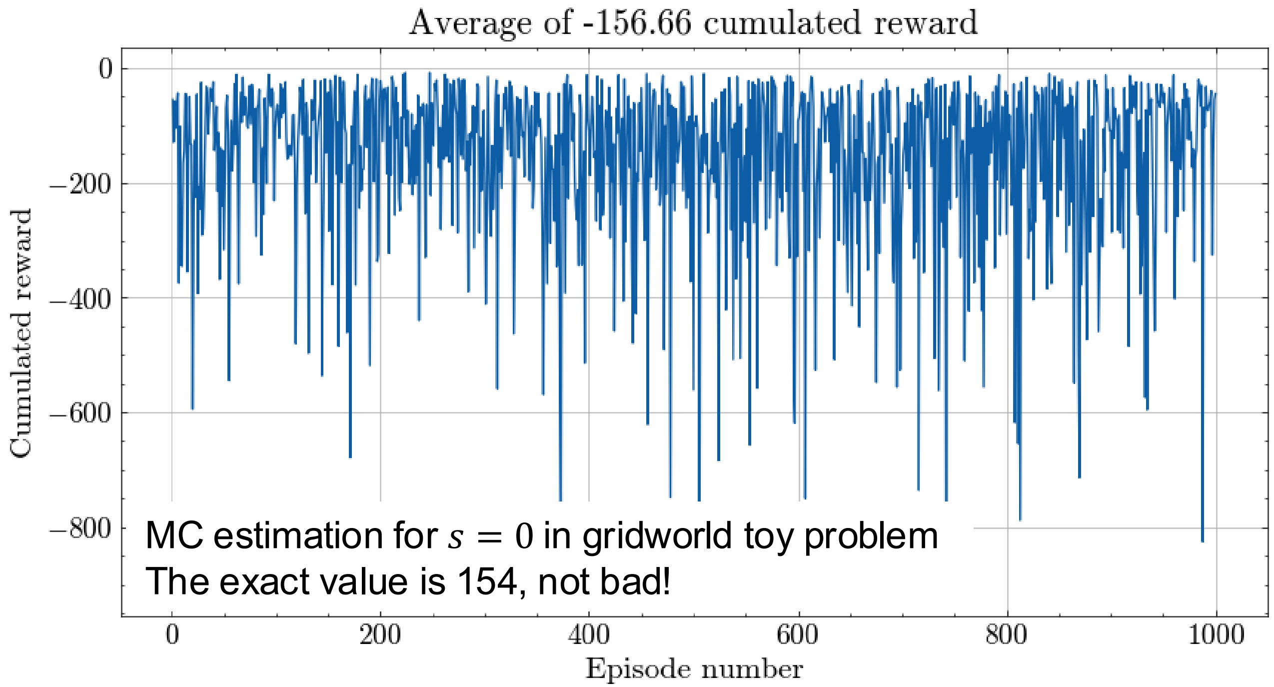
- Loop through each episode to see when the state s was visited.
- Compute the return starting from s each time you encounter s (or only the first time).
- Average the returns to estimate $\mathcal{V}^{\pi}(s)$.

$$\boxed{\mathcal{V}(s) \approx \frac{1}{N(s)} \sum_{i=1}^{N(s)} \mathcal{G}_t^{(i)}}$$

N = # of times s was visited across episodes

$$\lim_{t \to \infty} \mathcal{V}_t(s) = v^*(s)$$

Monte Carlo learning



Use in value estimation and policy improvement (“learning”):

- In **dynamic programming** we use the Bellman equation as an update rule to estimate the value function → needs $\mathcal{P}$
- With **Monte Carlo** we can estimate the value function with full episodes → no need for $\mathcal{P}$

- Very simple and intuitive
- You only need experience, not the environment dynamics $\mathcal{P}$
- Key role in modern RL

- Requires full episodes (slow learning, expensive simulation or experiment)
- High variance (noisy, uncorrelated future)
- Sample inefficient (**some states never get updated, depends on exploration** 🚨)

Temporal difference learning

How to compute the averages of action-value methods with **constant memory** and **constant computation step**, i.e., without storing and averaging a lot of data in tables?

Making long-term predictions is exponentially complex, memory scales with the number of steps of the prediction

Instead of:

- computing expected values over all possible next states, which requires $\mathcal{P}$ (full Bellman backup) or
- waiting for complete episodes to compute the full return $\mathcal{G}$ (MC learning)

we can simply sample the next state s' and reward from the unknown $\mathcal{P}$ (one step lookahead) and already estimate $\mathcal{V}(s)$ by bootstrapping from a guess of the value of the next state $\mathcal{V}(s')$.

→ **We update the value based on a single transition instead of the full distribution (DP) and without waiting (MC).**

→ We do not compute an expectation! But with enough samples it will converge to it.

Temporal difference learning

Target = $r + \gamma\, \mathcal{V}(s')$

Bootstrapped sampled-based estimation of the expected return (one step)

It's the value we want our current $\mathcal{V}$ to move toward

No expectation, one sample only

$$\mathcal{V}(s) \leftarrow \mathcal{V}(s) + \alpha[r + \gamma \mathcal{V}(s') - \mathcal{V}(s)]$$

New estimate ← Old estimate + Step size $\underbrace{[\text{Target - Old estimate}]}_{\text{Temporal difference error}}$

$\mathcal{V}(s)$ and the target are "guesses" → TD learning is a guess from a guess!

- We can update the value function after each step → great for continuing tasks
- Much more sample efficient than MC
- Does not need to know $\mathcal{P}$
- Foundational in modern RL

- Bootstrapping bias
- Can be unstable when paired with function approximation
- Requires access to the environment (might be expensive or unsafe)
- Does not give returns

Off-policy learning

Exploration vs exploitation dilemma appears again:

We want to learn the optimal behaviour and for that we need to behave non-optimally to explore all state-action pairs.

Off-policy learning decouples data collection from policy learning:

Behaviour policy $b(a|s)$
Policy to generate behaviour

Exploration (e.g. epsilon-greedy, soft policy)

Target policy $\pi(a|s)$
Policy being learned $\pi \approx \pi^*$

Exploitation (e.g. greedy)

Example: Q-learning $$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \underbrace{\gamma \max Q(s',a)}_{\text{Target policy } \pi(a|s)} - Q(s,a)]$$

Act under $b(a|s)$, update with $\pi(a|s)$

Summary

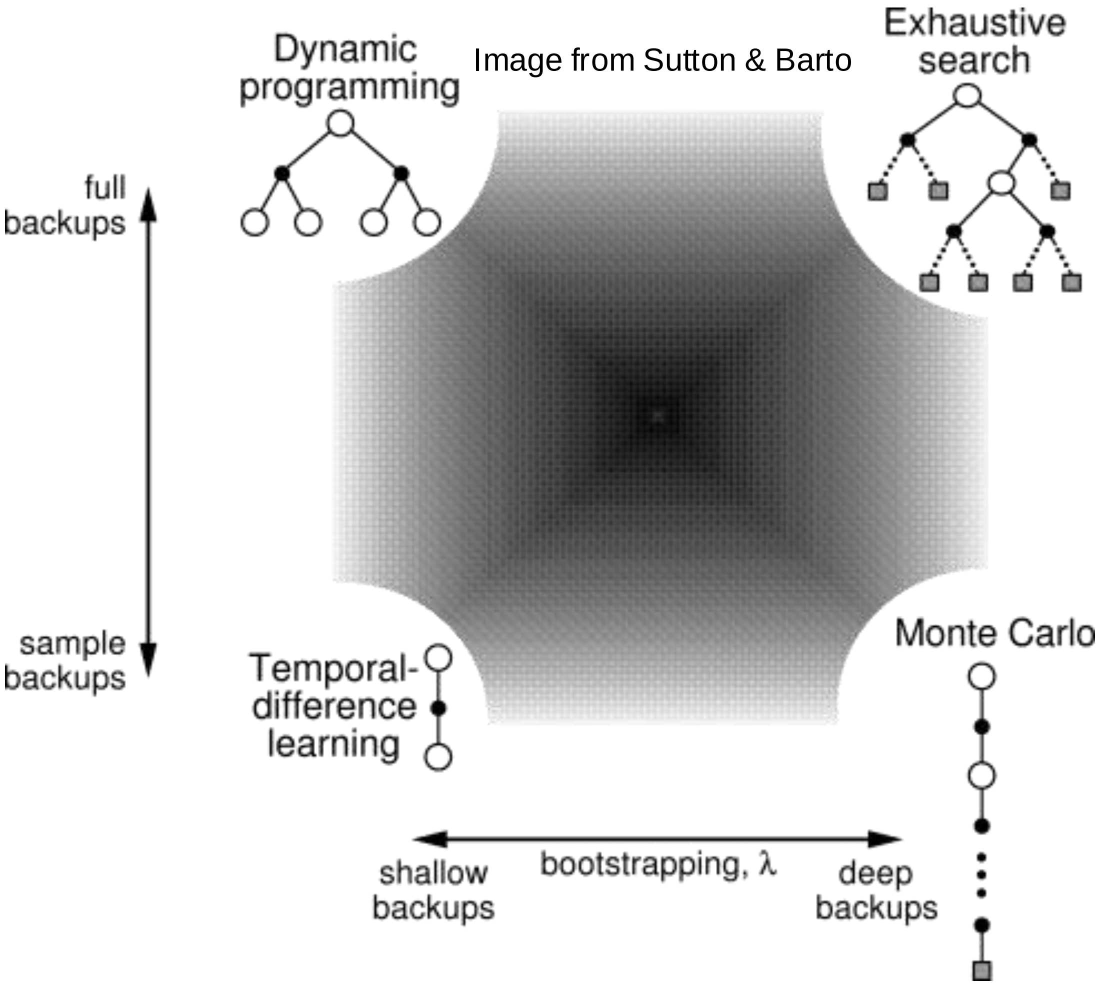
Tabular solution methods for finite MDPs

Methods	Techniques	Model-based	Bootstrapping	Algorithms
Dynamic programming	Iterative	Yes	Yes	Policy evaluation Policy iteration Value iteration
Monte Carlo	Sampling (episode-based estimation)	No	No	First-visit MC Every-visit MC
Temporal difference	Approximation (sampling + approximation)	No	Yes	TD(0) Q-learning SARSA

Model-based = we know the transition dynamics $\mathcal{P}$ of the problem

Summary

Tabular solution methods for finite discrete MDPs



$\mathcal{V}$ / $\mathcal{Q}$ and π are stored as arrays

- What happens to infinite or continuous MDPs?
- Can we identify and enumerate all states? (not in POMDPs)

Model-free deep RL

- **Function approximation of $\mathcal{V}$ / $\mathcal{Q}$ and π**
  → Opens the door to high dimensional continuous problems (tractable).
  → Can learn abstract features.
  → Introduces bias, variance, and stability challenges.
  → Fewer convergence guarantees.
- The function we learn can generalise to states never seen before.
  → Parameters θ are shared over all states.
  → Generalisation only as good as data.

Deep reinforcement learning

Policy gradient

Policies are parametrized with parameters θ and the goal is always to maximise the cumulated expected reward

$$\max_{\theta} J(\pi_\theta) = \mathbb{E}_{\pi_\theta}[\mathcal{G}_t]$$

If the policy is parametrized with a neural network we can optimise the policy with gradient descent:

$$\theta \leftarrow \theta + \alpha \nabla J(\pi_\theta)|_\theta$$

How to calculate $\nabla J(\pi_\theta)$
→ Policy gradient theorem

- Poor sample efficiency (needs many interactions).
- Sensitive to learning rate α and initialization parameters.
- In its basic form has high variance due to MC return estimations.
- Used in REINFORCE, A2C, A3C, TRPO, PPO, SAC.

Deep reinforcement learning

Value-based

- Approximate the value function with neural networks.
- Same concept as before: take the action with the highest Q-value.
- Does not explicitly store the policy.
- The noise in actions is dealt with by averaging over many samples and exploration.

Actor-critic methods

Actor: learns the policy $\pi_\theta(\mathrm{a}|s)$ and improves it with policy gradient

Critic: learns the value function $\mathcal{V}(s)$ or $\mathcal{Q}(s,a)$ or $\mathcal{A}(s,a)$= $\mathcal{Q}(s,a) - \mathcal{V}(s)$

- Actor uses the critic’s value in policy gradient
- Critic updated using TD error (bootstrapping, sample efficient)

Deep reinforcement learning

Common model-free algorithms

	Description	Policy	Action space	State space	Operator
DQN	Deep Q Network	Off-policy	Discrete	Continuous	Q-value
DDPG	Deep Deterministic Policy Gradient	Off-policy	Continuous	Continuous	Q-value
A3C	Asynchronous Advantage Actor-Critic Algorithm	On-policy	Continuous	Continuous	Advantage
TRPO	Trust Region Policy Optimization	On-policy	Continuous	Continuous	Advantage
PPO	Proximal Policy Optimization	On-policy	Continuous	Continuous	Advantage
TD3	Twin Delayed Deep Deterministic Policy Gradient	Off-policy	Continuous	Continuous	Q-value
SAC	Soft Actor Critic	Off-policy	Continuous	Continuous	Advantage

- Model-based RL
- Meta RL
- Multi-agent RL
- Hierarchical RL
- …

Other concepts

Imitation learning

- No trial and error, no solving an MDP, no learning from reward, no explicit reward.
- Learns by mimicking expert behaviour.
  → It's easier to show behaviour than to engineer a reward.

Inverse RL: you can learn a reward function that explains the expert behaviour.

Behaviour cloning: you can learn a policy from expert (s, a) pairs → no need for extensive exploration (warm start to traditional RL, safer).

Distributional RL

- Instead of estimating the expectation of returns (mean) we estimate the whole distribution over returns.
- With full distribution agent knows about uncertainty, risk, and variability in future rewards.
  → Robust policies

Well done!

You made it through the introduction of foundational RL concepts 🤖

Let’s get some questions now and continue the discussion during the coffee break

☕

Resources

- Sutton & Barto book
- https://arxiv.org/pdf/cs/9605103.pdf
- Reinforcement learning lectures by David Silver
- https://spinningup.openai.com/en/latest/
- Coursera RL specialization
- https://arxiv.org/pdf/1810.06339.pdf

Let’s connect

@ansantam (LinkedIn, Instagram)