UNIVERSITY OF LIVERPOOL

Department of Physics

INTRODUCTION TO REINFORCEMENT LEARNING

Dr. Andrea Santamaria Garcia
Lecturer

Reinforcement Learning for Autonomous Accelerators workshop 2026 (Liverpool)

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 🧠!
- This topic is usually learned in **one semester or more**

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

DOI:10.5281/zenodo.12649046

OUTLINE

- **Theoretical foundations of reinforcement learning** (26 slides)
  - ***Trial-and-error concepts*** *(agent, reward, goal, reward is enough, trade-off between exploitation and exploration, sequential decision making)*
  - ***Optimal control concepts and dynamic programming*** *(Markov decision processes (MDPs), Markov property, partially observable Markov decision processes (POMDPs), reward and return, policy, value function, the Bellman equation)*

- **An illustrative toy problem: gridworld** (7 slides)
  - *Policy evaluation (exact), policy evaluation (iterative)*

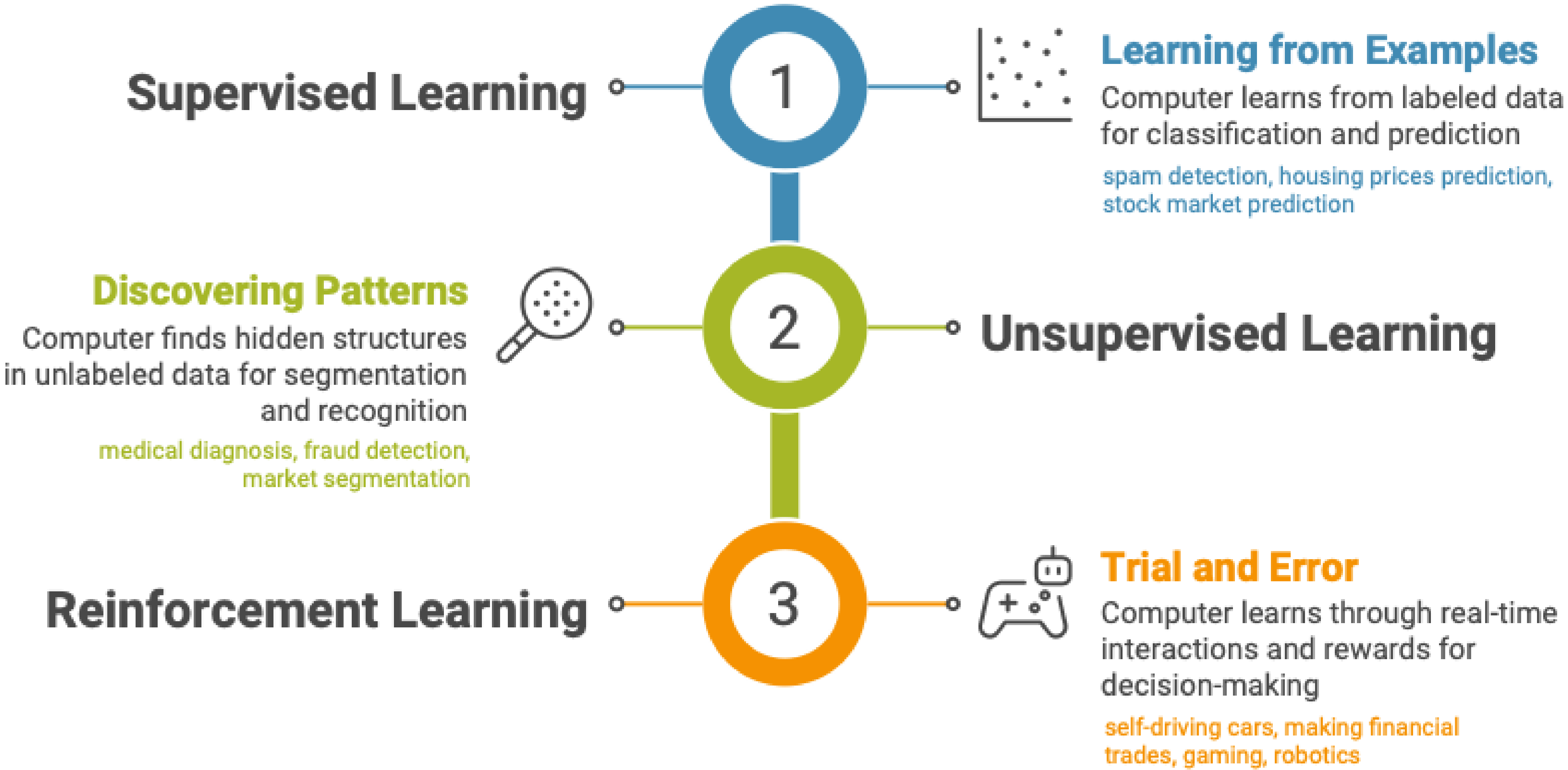
- **How does an agent actually learn?** (9 slides)
  - *The RL goal, optimal policies, Bellman optimality equations, policy improvement, application to gridworld problem, greedy actions*

- **Sampling and approximation methods** (8 slides)
  - *Monte Carlo learning, temporal difference learning*

THEORETICAL FOUNDATIONS OF REINFORCEMENT LEARNING

Learning paradigms (ML)

Ways of learning from data



Supervised Learning

Learning from Examples

Computer learns from labeled data for classification and prediction

spam detection, housing prices prediction, stock market prediction

Unsupervised Learning

Discovering Patterns

Computer finds hidden structures in unlabeled data for segmentation and recognition

medical diagnosis, fraud detection, market segmentation

Reinforcement Learning

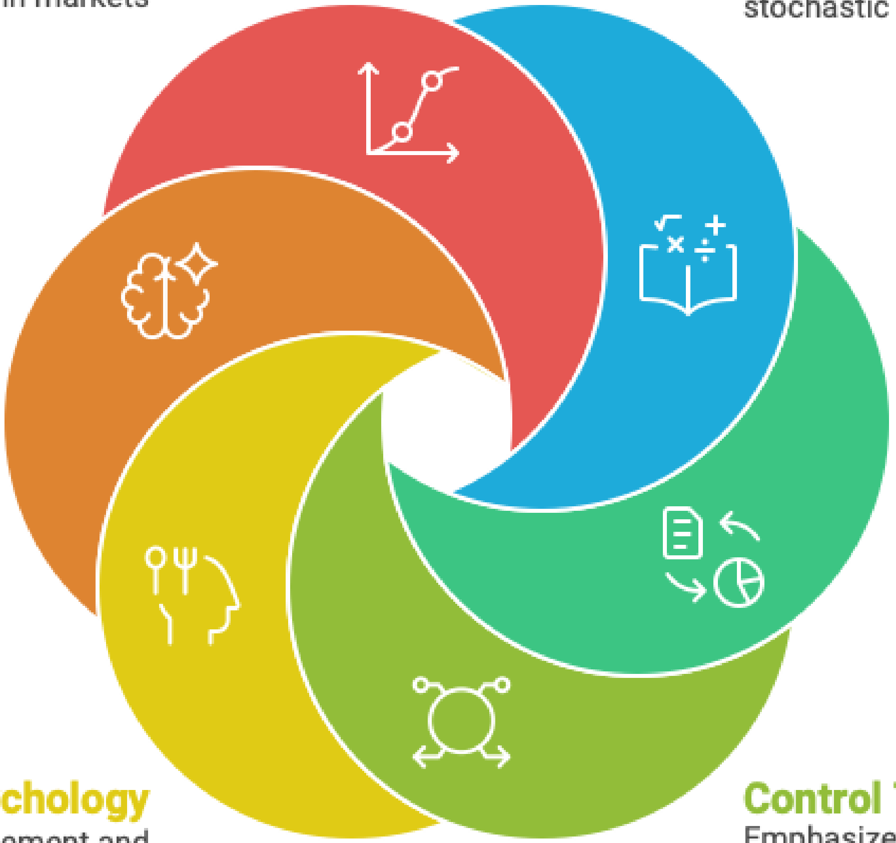
Trial and Error

Computer learns through real-time interactions and rewards for decision-making

self-driving cars, making financial trades, gaming, robotics

Reinforcement learning

More than machine learning: intellectual inheritance



Economics
Analyzes strategic learning and equilibrium in markets

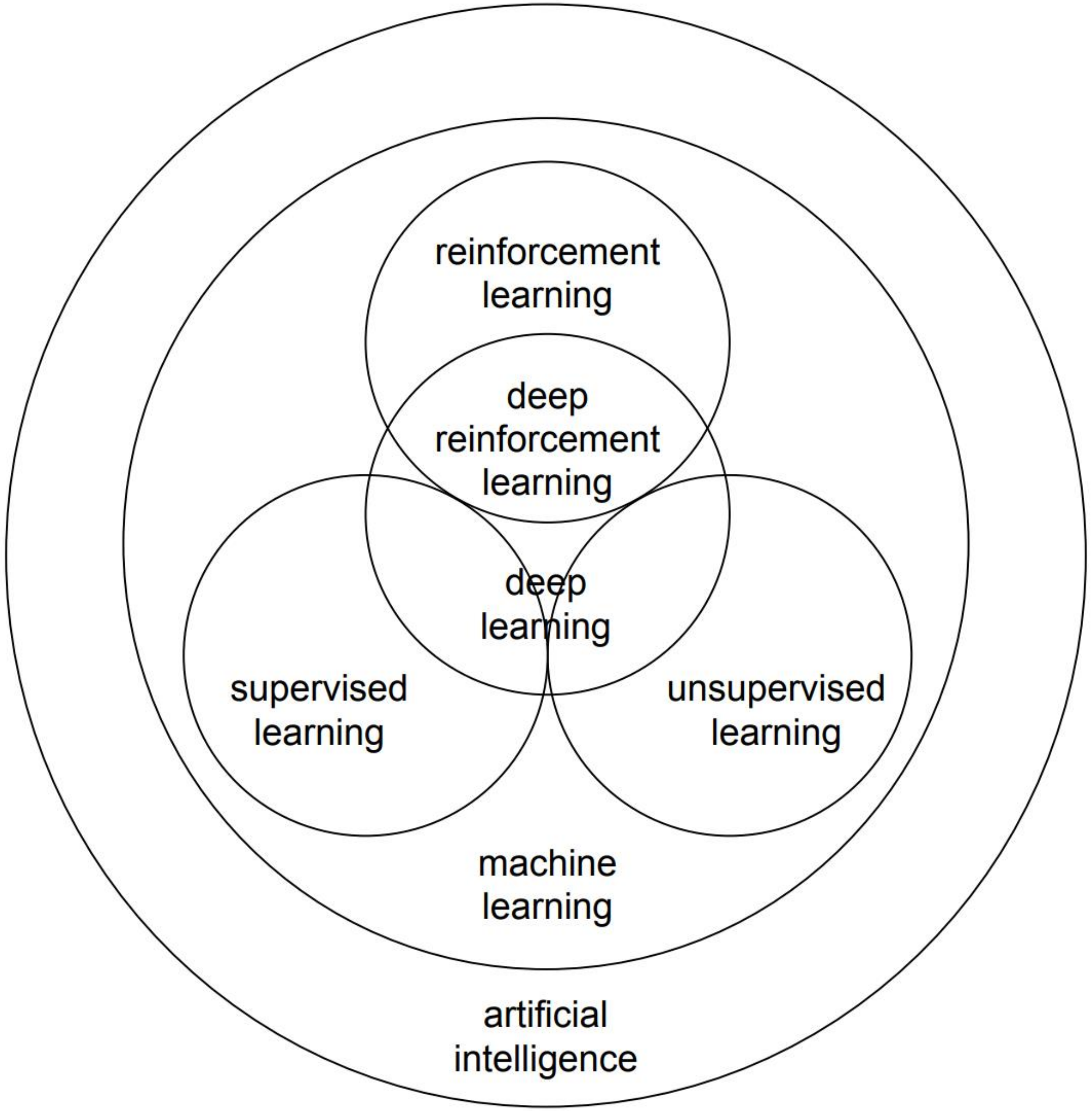
Mathematics
Provides foundational tools like Bellman equations and stochastic processes

Neuroscience
Investigates reward prediction error in the brain

Operations Research
Focuses on MDPs and policy optimisation for decision-making

Psychology
Explores reinforcement and behavioural adaptation

Control Theory
Emphasizes feedback and stability in systems

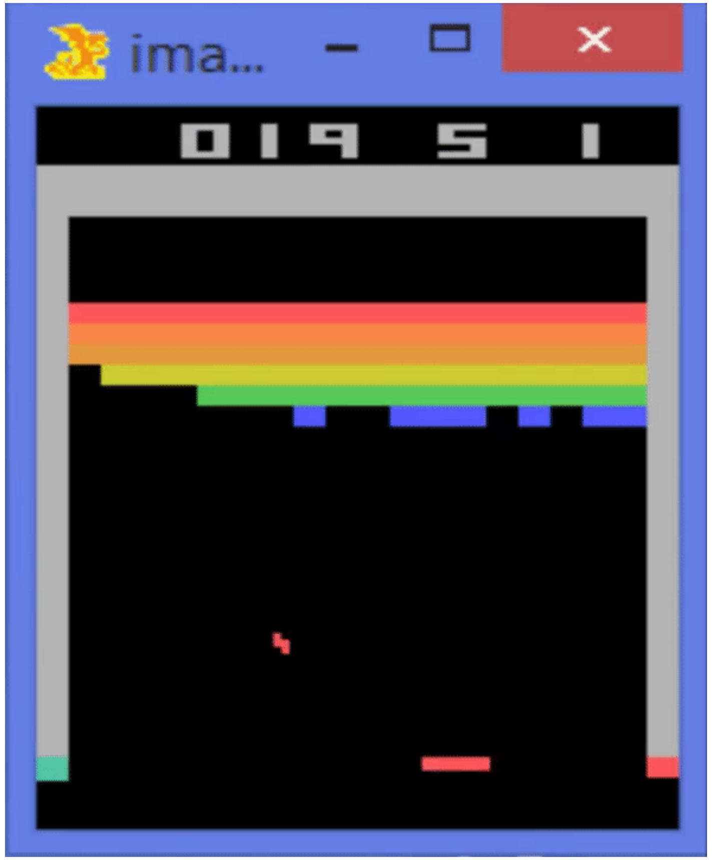


- All learning paradigms have **deep learning variants** (i.e., uses neural networks)
- Today **modern RL** refers to **deep RL**

Deep reinforcement learning

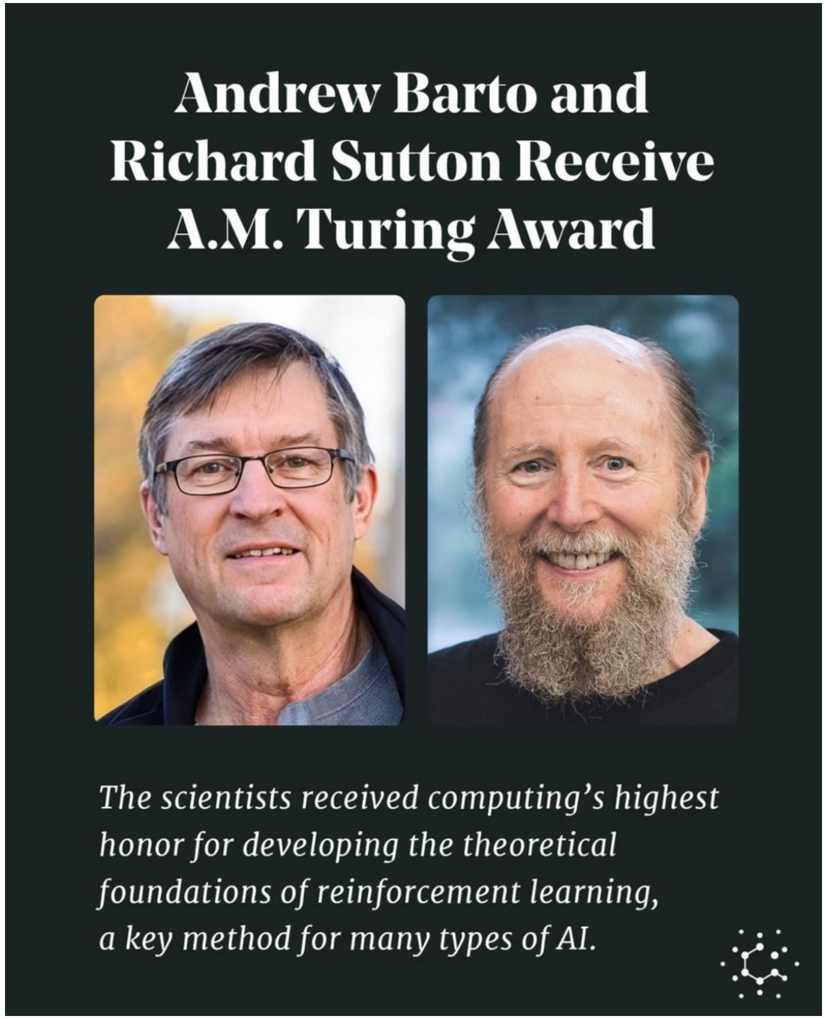
Opened the door to high dimensional environments

https://arxiv.org/abs/1707.02286



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

Reinforcement learning: recent recognition



Andrew Barto and Richard Sutton Receive A.M. Turing Award

The scientists received computing's highest honor for developing the theoretical foundations of reinforcement learning, a key method for many types of AI.

2024 Turing Award

> "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!

Behavioural foundation

Trial-and-error learning

- Behaviour adapts through reward feedback
- Actions that lead to favourable outcomes are reinforced
- Exploration vs exploitation emerges naturally

Provides the behavioural basis:

→ Learning through interaction and delayed consequences

Mathematical decision framework

Optimal control and dynamic programming

- Sequential decision making under uncertainty
- Markov decision processes (MDPs)
- Bellman equation and value functions
- Policy optimisation

Provides the formal structure:

→ Defines what "optimal behaviour" means

Computational learning mechanism

Temporal difference learning

- Learns from partial experience
- Bootstraps future value estimates
- Online, sample-based updating
- Model-free learning

Provides scalability:

→ Makes optimal control feasible without knowing the model

Trial-and-error concepts

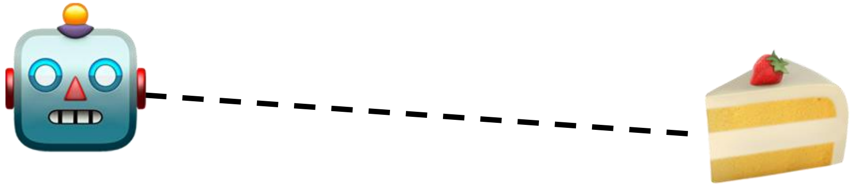
The RL problem: agent, goal, and reward

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

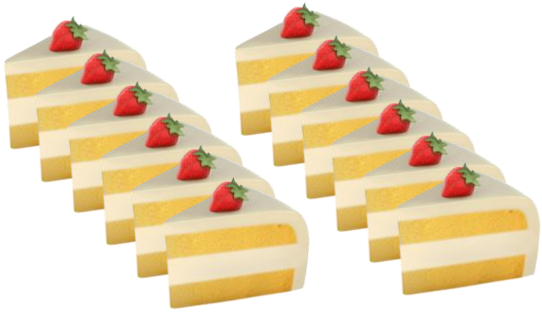
Agent
executes action
→ receives observation
→ receives scalar reward

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

Goal
maximisation of cumulative reward through selected actions

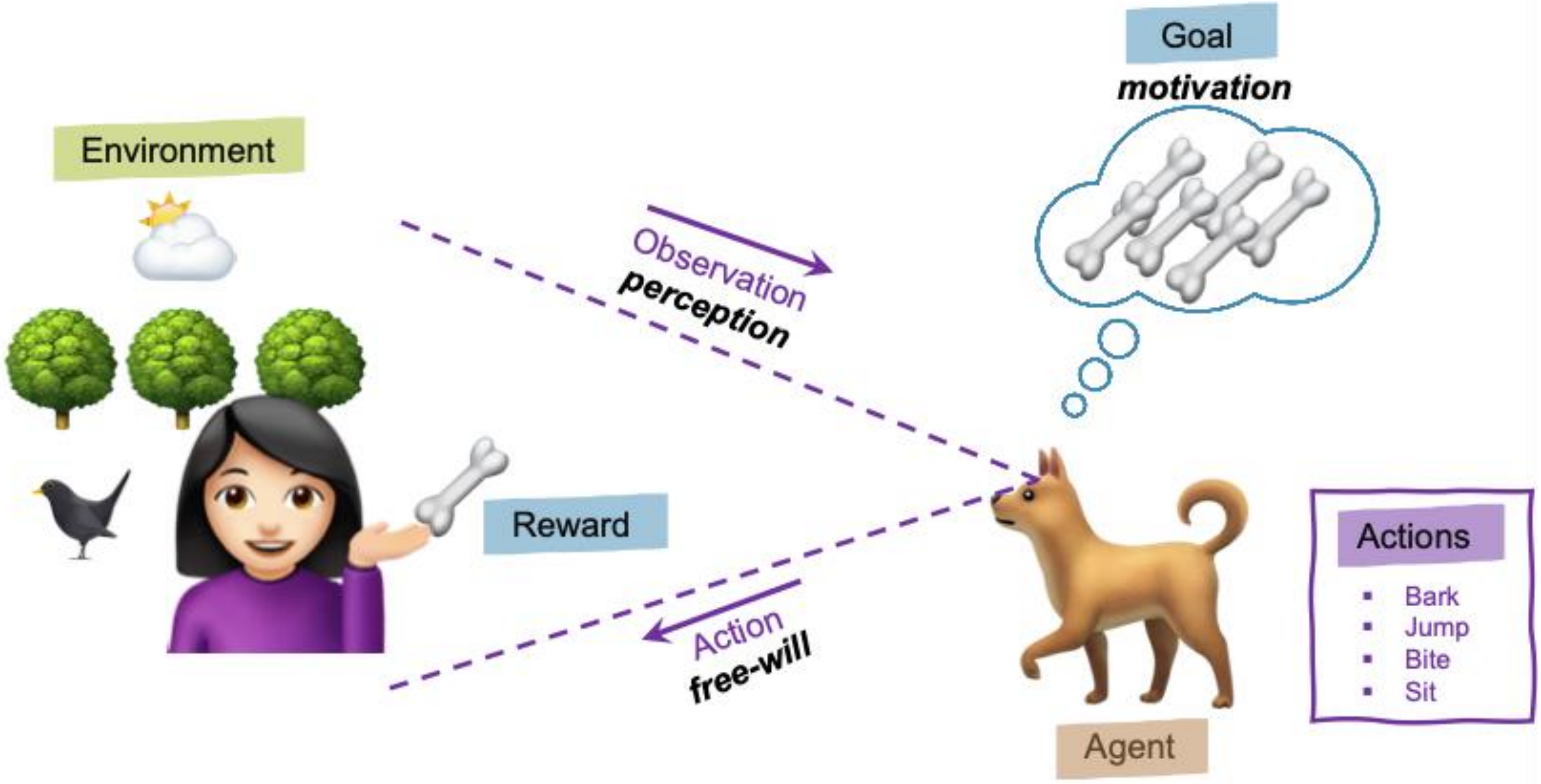


Reward shaping is non-trivial



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.**

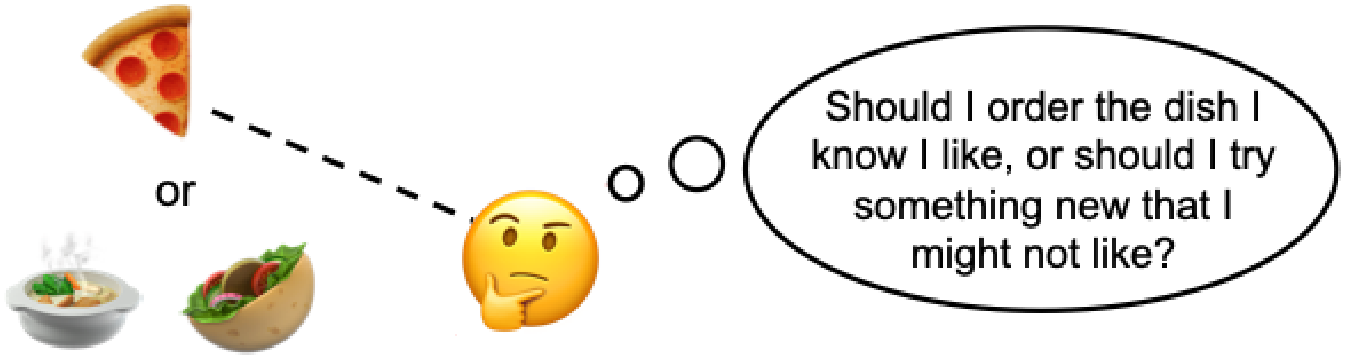
"Scalar reward is not enough": a response to Silver et al. (2021)

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?



Should I order the dish I know I like, or should I try something new that I might not like?

Trial-and-error concepts

Trade-off between exploitation and exploration

The agent needs to:

- ✓ **Exploit** what it has already experienced in order to obtain reward now.
- ✓ **Explore** 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

Too much exploitation
the agent might converge prematurely
to a suboptimal strategy

Too much exploration
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 must implicitly or explicitly address this trade-off (e.g., **assessing the value of actions** and estimating future reward).
- ➢ The **right balance** depends on the **problem, environment, and computational constraints.**

Finite vs. infinite horizons

In **finite horizon settings** exploitation pressure increases near the end. In **long-term settings**, exploration can be amortised over time.

Deterministic vs. stochastic environments

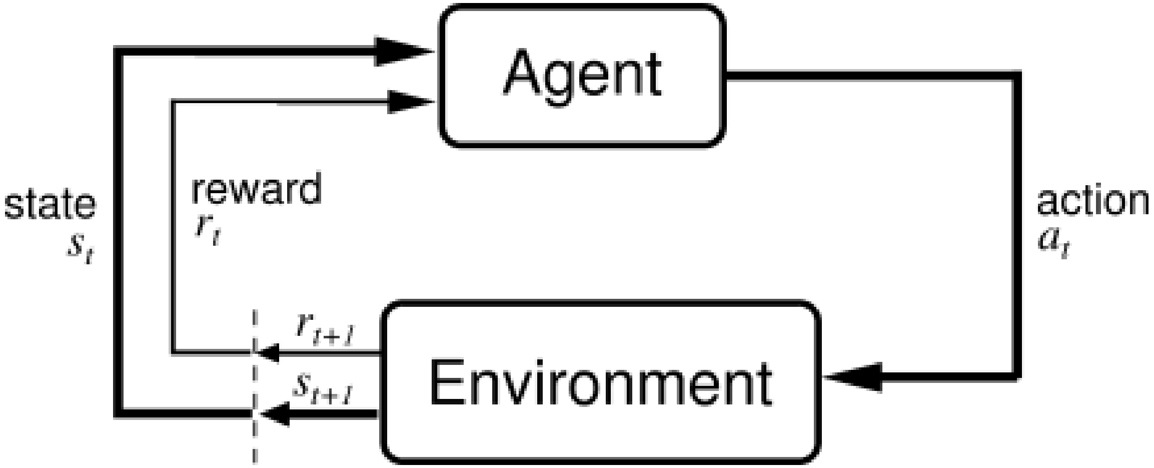
In **deterministic settings**, exploration can decay once the optimal policy is identified. In **stochastic settings**, uncertainty may persist, requiring continued sampling.

Examples of different strategies:

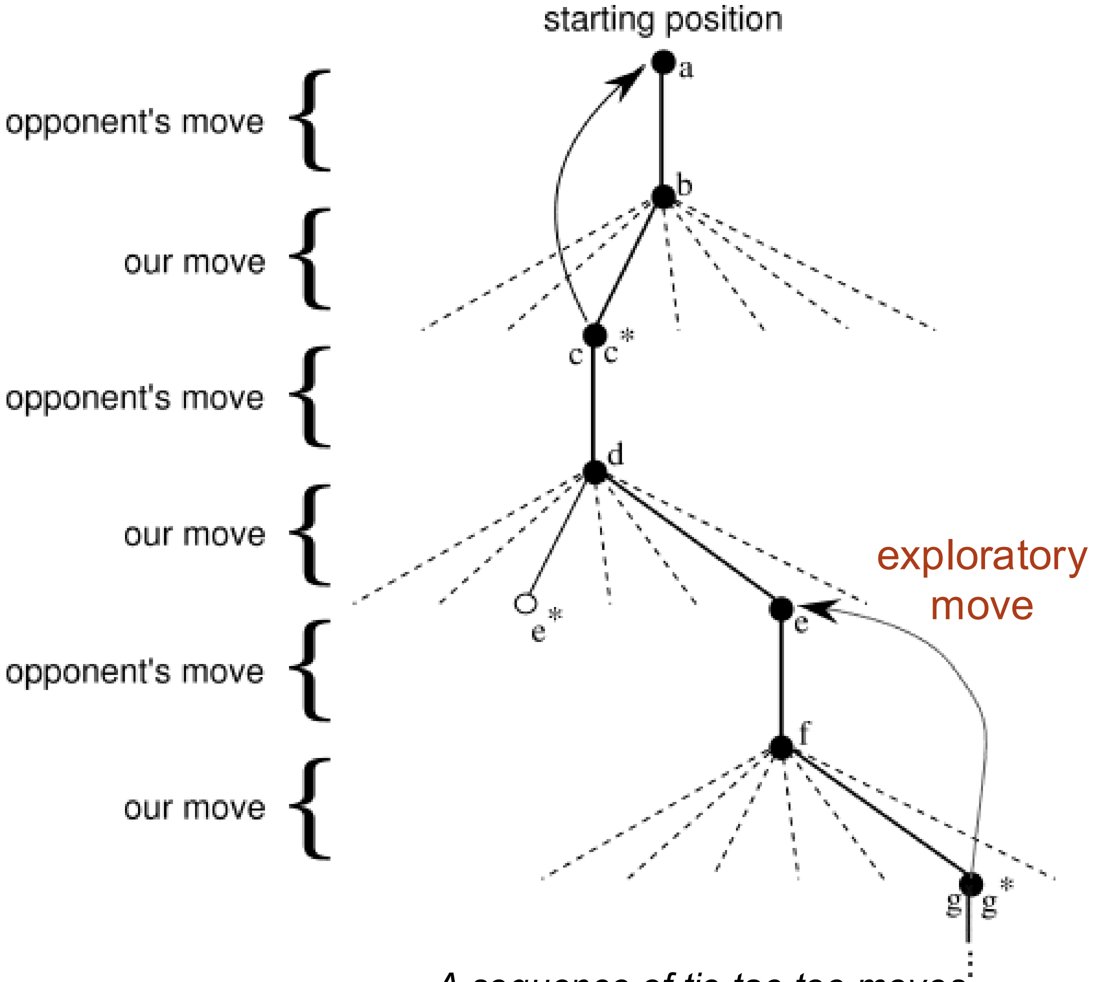
- ➢ **ε-greedy schedule**: with small probability ε select a random action, otherwise choose the current best, typically decreasing ε over time to shift from exploration to exploitation.
- ➢ **Softmax / Boltzmann:** sample actions probabilistically according to their estimated value, so higher-value actions are favoured but all remain possible depending on a temperature parameter.
- ➢ **Optimism / UCB-style**: select actions by balancing estimated value with an uncertainty bonus, favouring options that are either promising or under-explored.

Trial-and-error concepts

How to formalise sequential decision making?



The famous RL loop



Images from Sutton & Barto

A sequence of tic-tac-toe moves

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

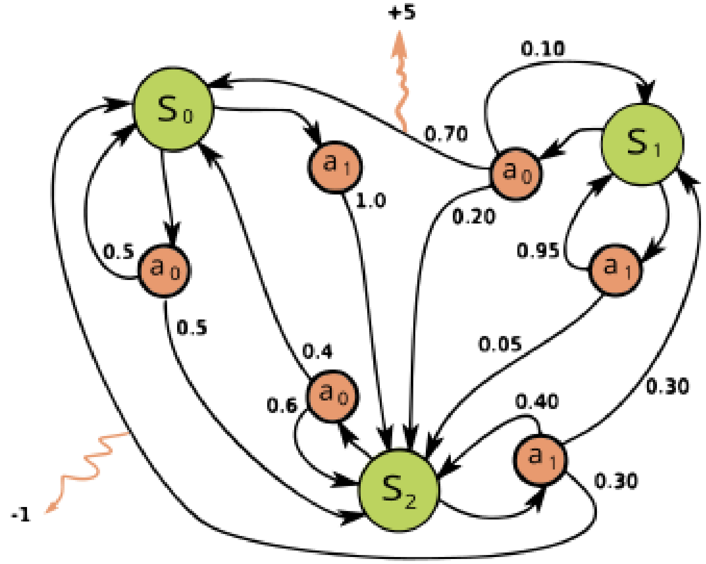
$r = \mathcal{R}(s, a, s') = \mathcal{R}^{a}_{ss'}$ Immediate reward

$\mathcal{P}$ transition probability function

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

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

γ discount factor



MDP example from Wikipedia

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

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

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 $\mathcal{P}^a_{ss'}$ of ending in state s' after taking action a while being in state s:

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

which are the entries $\mathcal{P}^a_{ss'}$ 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 must 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) - hidden
$\mathcal{A}$	action space (all valid actions)
$\mathcal{T}(s' \vert s, a)$	transition probability function
$\mathcal{R}(s, a)$	reward function
➡ Ω	observation space (all valid observations)
➡ $\mathcal{O}(o \vert s')$	observation probability function
γ	discount factor

🚨

- In a POMDP the agent does not observe s_t directly (latent)
- It observes o_t which is noisy, partial → does not uniquely identify s_t
- The agent must maintain a **belief state** over possible states and update it over time

$$b_t(s) = P(s_t = s | o_{1:t,} a_{1:t-1})$$

Most real-world problems are partially observable

Optimal control concepts 💡

Partially observable Markov decision processes (POMDP)

A **belief** is:

- a **probability distribution** over latent states (explicit uncertainty)
- a minimal **sufficient statistic** of the entire history (posterior dist.)
- a **compressed representation of uncertainty** (all earlier history is encoded in the belief)

Maintaining a **belief**:

- restores the **Markov property**
- transforms the problem into an MDP over a **continuous belief space**
- increases **computational complexity** dramatically (state space → space of probability measures). Exact solutions intractable.

🚨

If you choose a latent state that omits relevant variables, belief will not be sufficient

Belief assumes the state definition is complete

Key takeaway: *in a POMDP, the latent state evolves according to a Markov process, but because the state is not directly observable, the observation history is required for optimal decisions. However, this history can be compressed into a belief state, which restores the Markov property in belief space.*

Optimal control concepts

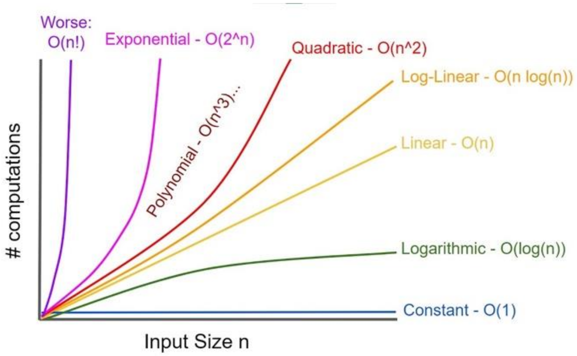
Partially observable Markov decision processes (POMDP)

Finite-state **MDP** is solvable in polynomial time

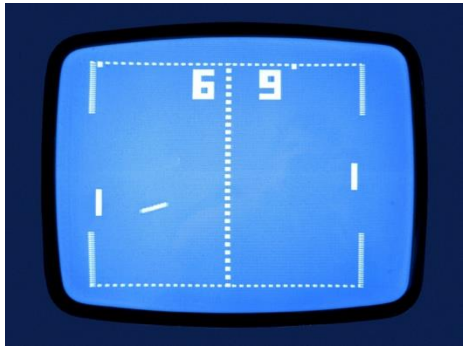
Finite-state **POMDP**:

- PSPACE-hard (infinite horizon)
- EXPTIME-hard (finite horizon)

because belief space is continuous and planning branches over observation histories



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

Memory-based approximations

Most **deep RL** methods **do not maintain an explicit belief distribution**, but use **approximations** and learn a hidden state representation:

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

→ ***no guarantees of optimality under partial observability***

Optimal control concepts 💡

Reward distribution

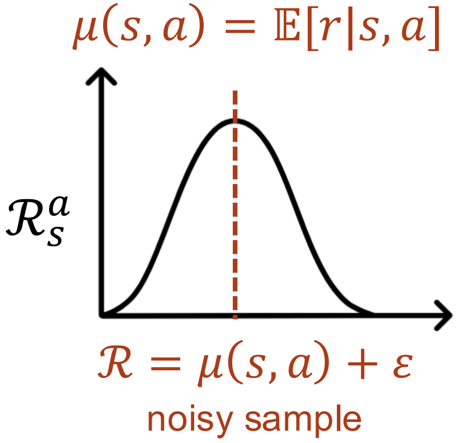
In **real-world environments**, **reward** is typically **stochastic**

- The same state-action pair can yield different rewards due to stochastic dynamics or unmodelled factors
- 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

- Most classical algorithms optimise for $\mu(s,a) = \mathbb{E}[r|s,a]$
- Only the expected reward is used for decision-making
- This collapses the distribution to a single scaler, ignoring variance, tail behaviour, risk sensitivity



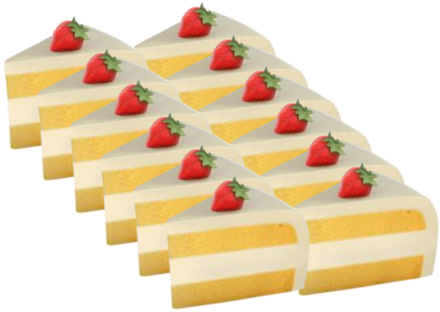
$\mu(s,a) = \mathbb{E}[r|s,a]$
$\mathcal{R}_s^a$
$\mathcal{R} = \mu(s,a) + \varepsilon$
noisy sample

Alternatives

- Estimate distribution parameters from samples
- Model the full reward distribution explicitly (model-based RL, Bayesian RL, distributional RL)

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 γ** 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 $a_t \sim \pi(a|s)$

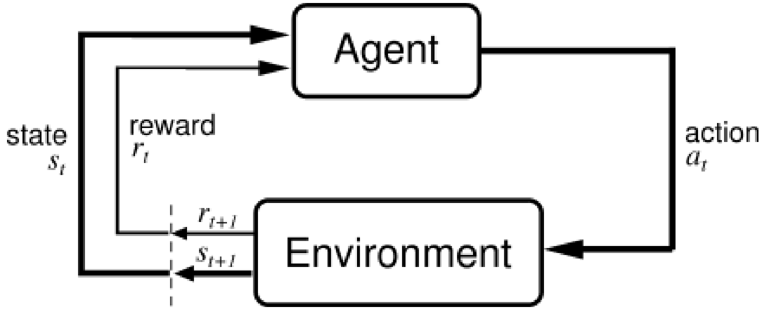
Sample randomly from a Gaussian dist. or from model

→ 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:

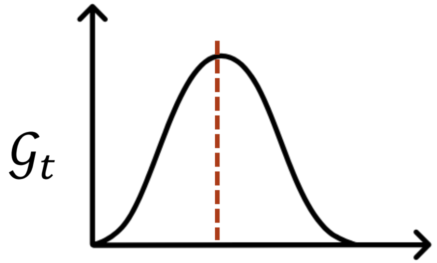
- an **estimation of expected future return**, gives “value” to an action.
- used to choose between states depending on how much reward we expect to get.
- depends on the agent’s behaviour (policy → action).
- a way to compare policies.

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”

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

Dynamic programming 💡

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

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

$$\xrightarrow[\mathbb{E}(\mathrm{f}) = \mathbb{E}(\mathbb{E}(\mathrm{f}))]{}$$

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

Dynamic programming

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:

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

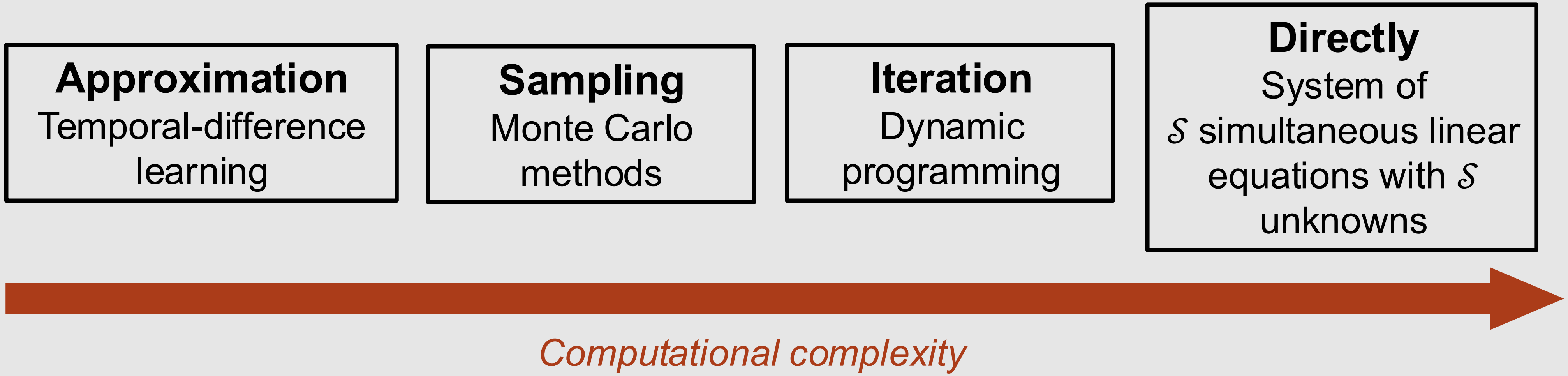
discrete case

Dynamic programming

The expanded Bellman equation

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

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

AN ILLUSTRATIVE TOY PROBLEM: GRIDWORLD

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:

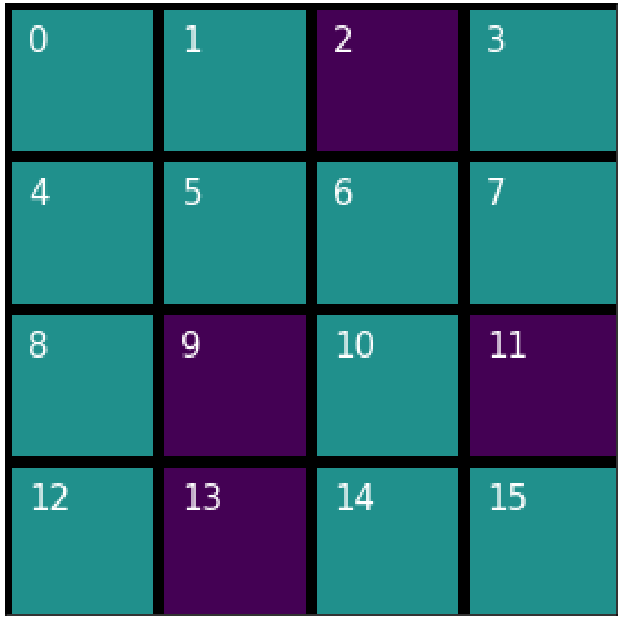
- A fully observable environment (MDP) → Markovian
- A small state space and action spaces $\mathcal{S}, \mathcal{A}$
- Know all transition probabilities $\mathcal{P}$

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



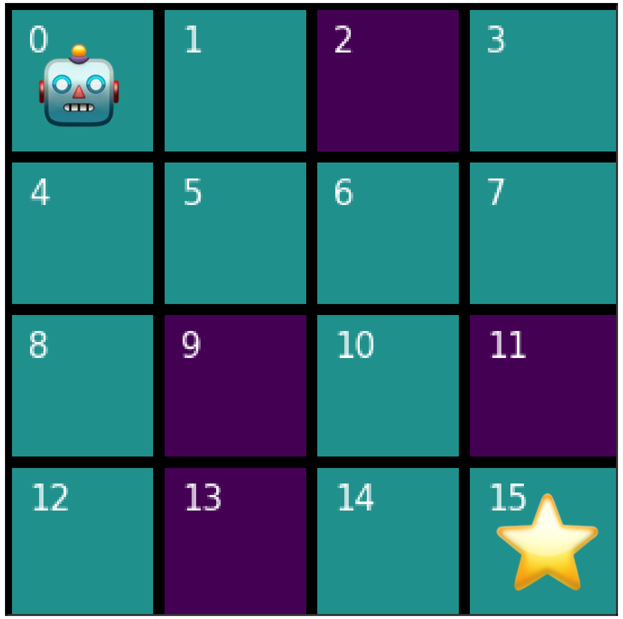
Gridworld toy problem

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

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

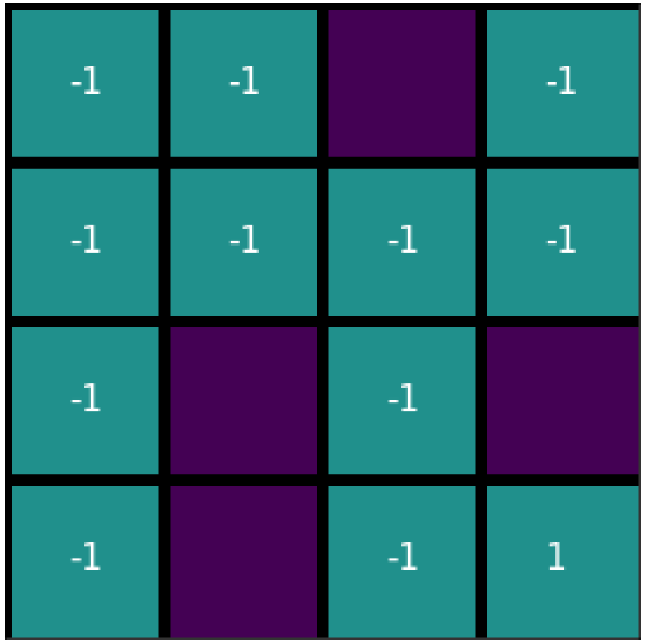
$\mathcal{P}^{a}_{s,s'} = 1$ Deterministic environment

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



0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

Reward design: why negative?



-1	-1		-1
-1	-1	-1	-1
-1		-1	
-1		-1	1

$\mathcal{R}$

Gridworld toy problem

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

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

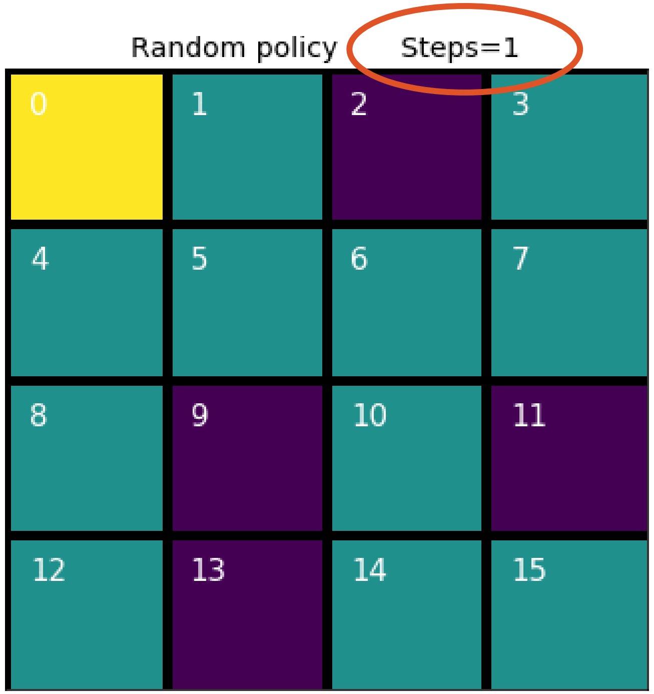
$\mathcal{P}^{a}_{s,s'} = 1$ Deterministic environment

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

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

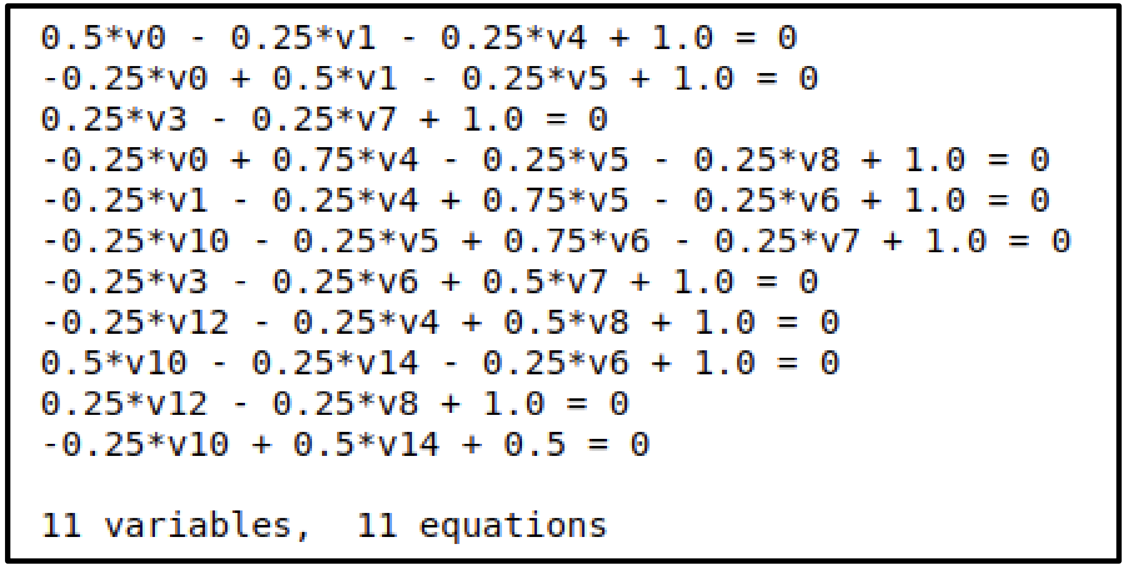
$$\mathcal{R} = \begin{cases} -1 \ \forall s, s \neq 15 \\ 1 \ s = 15 \end{cases}$$

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

Let's solve our set of simultaneous equations

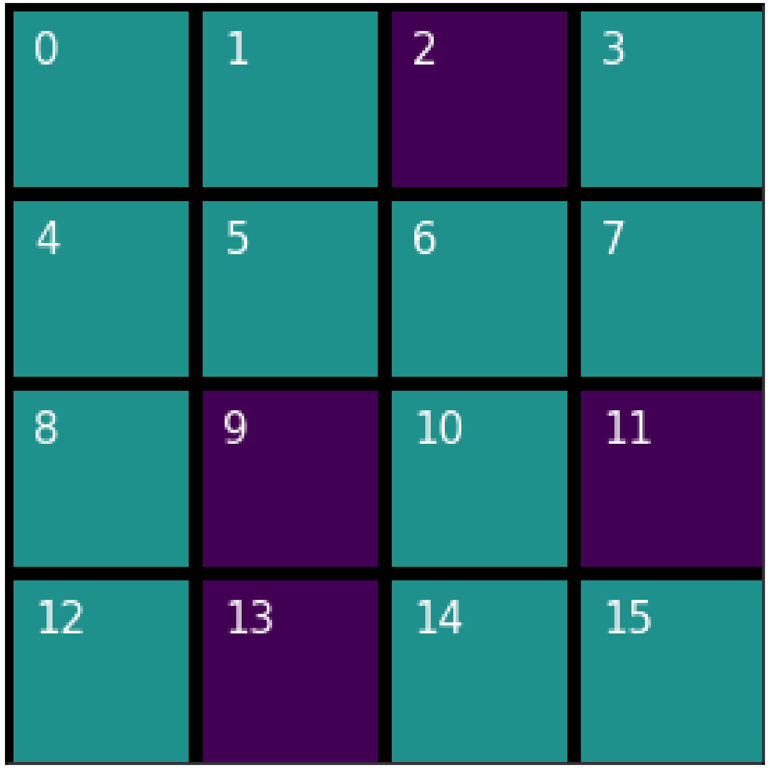
$$\mathcal{V}^{\pi}(s) = \sum_{a\in\mathcal{A}} \pi(a|s) \sum_{s'\in\mathcal{S}} \mathcal{P}^a_{s,s'}\left(\mathcal{R}^a_s + \mathcal{V}^{\pi}(s')\right)$$

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

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

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

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

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

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

The Bellman equation becomes an update rule:

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

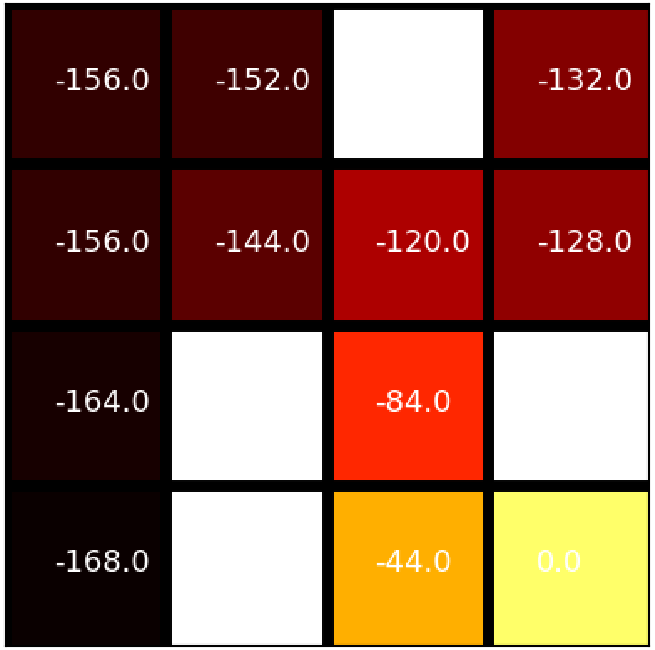
Dynamic programming

- 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.

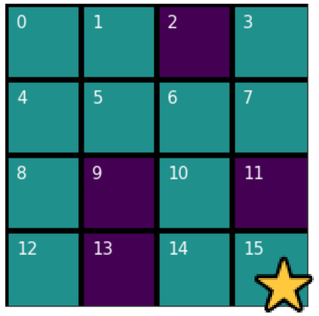
Computationally less expensive, but also won't scale

Value of random policy

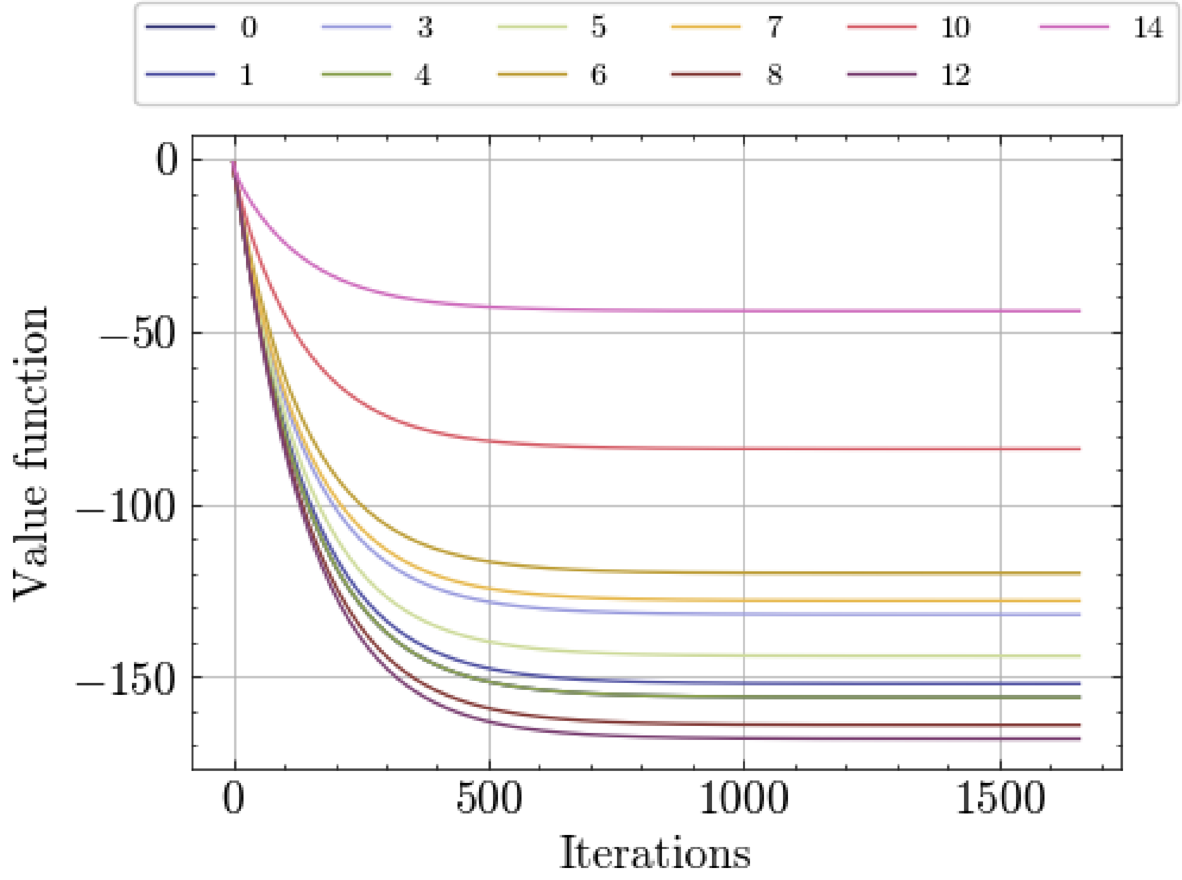


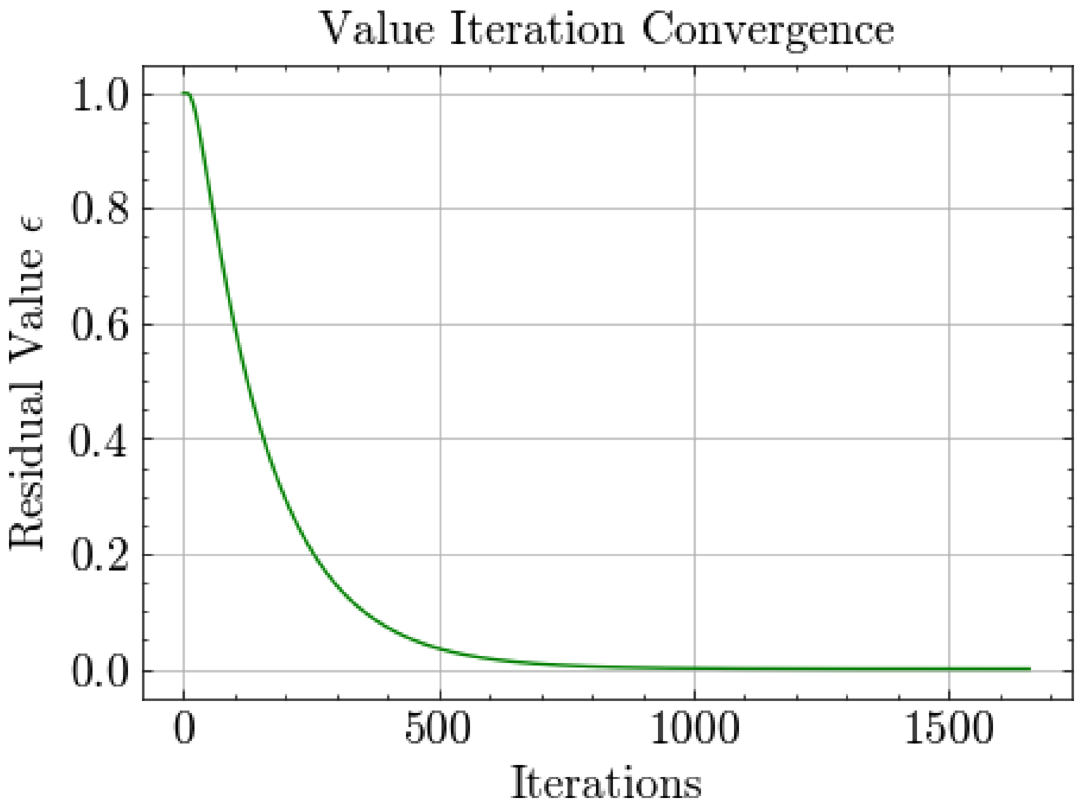
-156.0	-152.0		-132.0
-156.0	-144.0	-120.0	-128.0
-164.0		-84.0	
-168.0		-44.0	0.0

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.

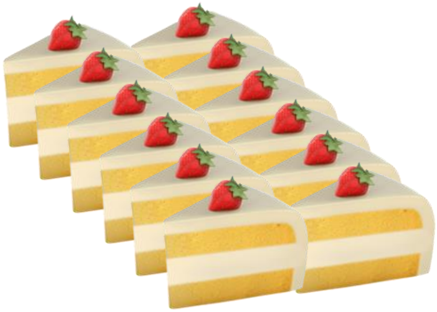
🤖 ?

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

HOW DOES AN AGENT ACTUALLY LEARN?

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 and tractable
- Value function computable
- Optimal policy computable

We can completely solve the control problem and find the **optimal policy** π^*

vs

Real world

State partially observable

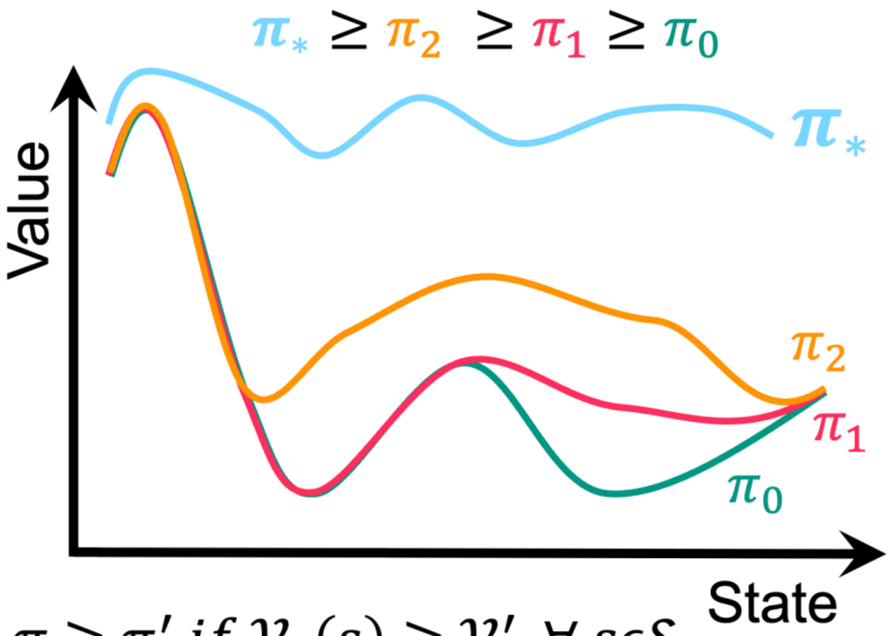
- POMDP
- Model unknown or learned
- Value function approximated
- Policy approximated

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.**

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

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

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

- 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

improvement

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

evaluation

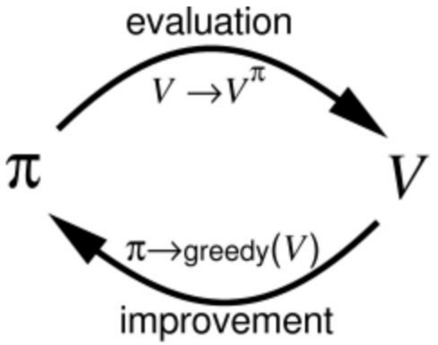
$$\boldsymbol{\pi}'(\boldsymbol{s}) = \arg\max_a Q^{\pi}(s,a)$$

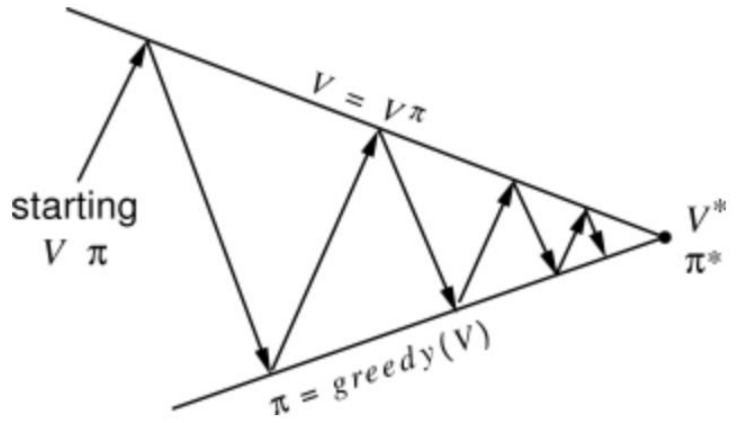
Greedy action

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.

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

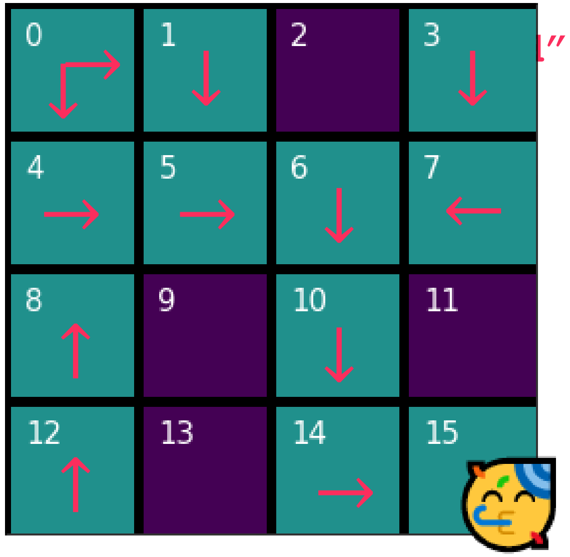
$$\mathcal{S} = (0, 1, 3, 4, 5, 6, 7, 8, 10, 12, 14, 15)$$

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

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

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

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

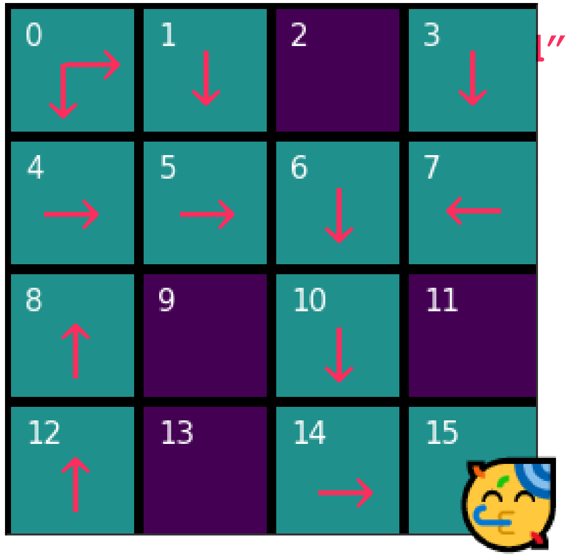
$\mathcal{A} = (\uparrow, \downarrow, \leftarrow, \rightarrow)$

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

About greedy actions 🚨

👧🏻 : *Cool. So, if the **value function** 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 **estimations** 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

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

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

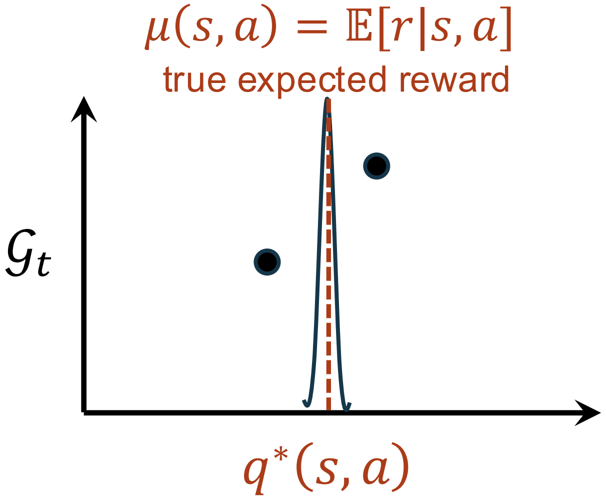
About greedy actions 🚨

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

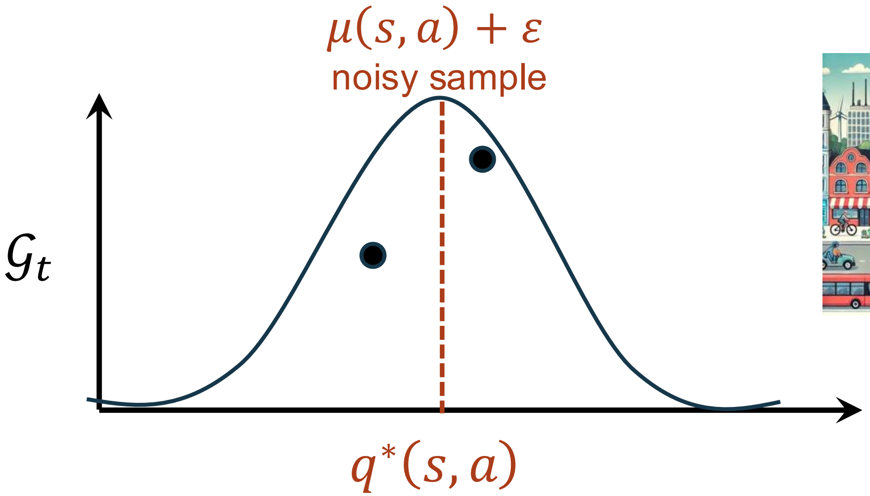
Greedy action: $a_t \doteq \arg\max_a \mathcal{Q}_t\,(s,a)$ select action with most value → pure exploitation

Near-greedy action: small probability ε to select randomly from all actions → ensures sufficient exploration

Does **greedy action** work? → it will depend on the uncertainties



If reward and transitions are deterministic, one sample is sufficient



If value uncertainty is large, more exploration is required to reduce estimation error.

$\mathcal{R}$ = deterministic + stochastic components

SUMMARY

Summary: introduction to RL

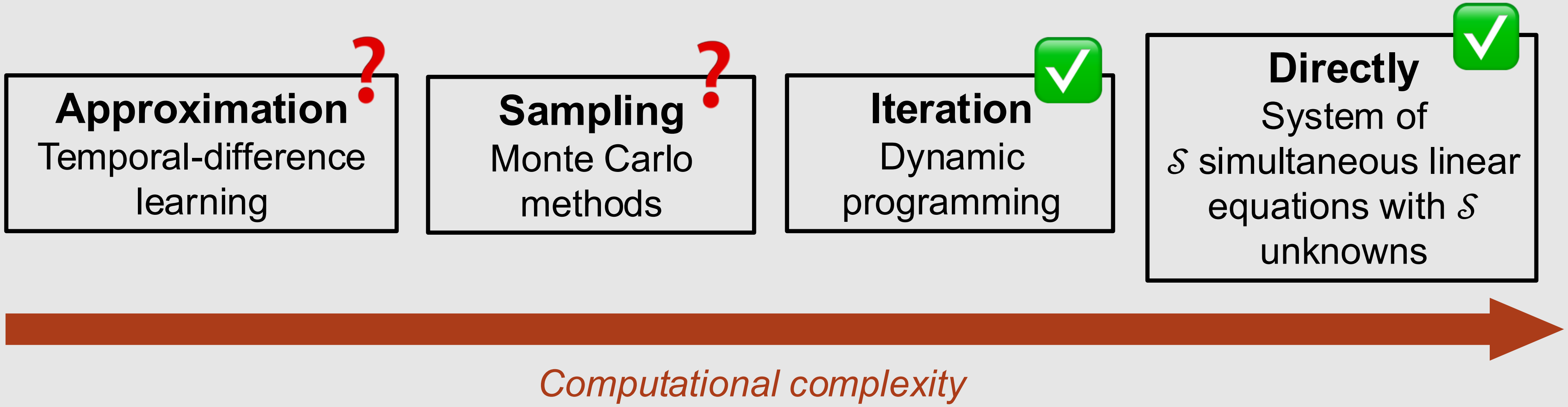
- Reinforcement learning formalises **sequential decision-making under uncertainty**.
- **MDPs** provide the **mathematical structure**: state, transition dynamics, reward, discounting, and the Markov property.
- The **Bellman equation** converts a global optimisation problem into **recursive local updates** (dynamic programming).
- When the model is known and tractable, we can compute the exact value function and optimal policy.
- In realistic settings, environments are stochastic, partially observable, and high-dimensional (**POMDPs**).
- **Exact solutions** become **computationally intractable** → we rely on sampling, approximation, and learning from interaction.
- **Exploration** is required when value estimates are uncertain. Purely greedy control is insufficient.
- The practical objective is not perfect optimality, but **robust, good-enough policies** under uncertainty and computational constraints.

SAMPLING AND APPROXIMATION METHODS

The expanded Bellman equation

How to solve it?

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



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

Transitioning to “modern” RL

Ideal setting

State fully observable

- MDP (finite, discrete)
- Model known and tractable
- Value function computable
- 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

- The Bellman operator is evaluated exactly using the known transition model
- Expectations are computed analytically from $\mathcal{P}(s|s,a)$
- Value functions are obtained via deterministic fixed-point iteration
- Convergence is algebraic and guaranteed under contraction of the Bellman operator

Modern RL (model free!)

- The Bellman operator is approximated via stochastic samples
- Expectations are replaced by sample-based estimators
- Value functions are learned through stochastic approximation
- Convergence is statistical and depends on sampling, noise, and step sizes

Monte Carlo learning

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

The experience is organised in episodes:

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

Episode 1

$$\vdots$$

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

Under policy π

Episode M

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

Every visit MC

- Loop through each episode $t = T, T-1, \dots, 0$ to see when each state s was visited
- Each time a particular s is visited update the return

$$\mathcal{G} \leftarrow \gamma\mathcal{G} + \mathcal{R}_{t+1}$$

- 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

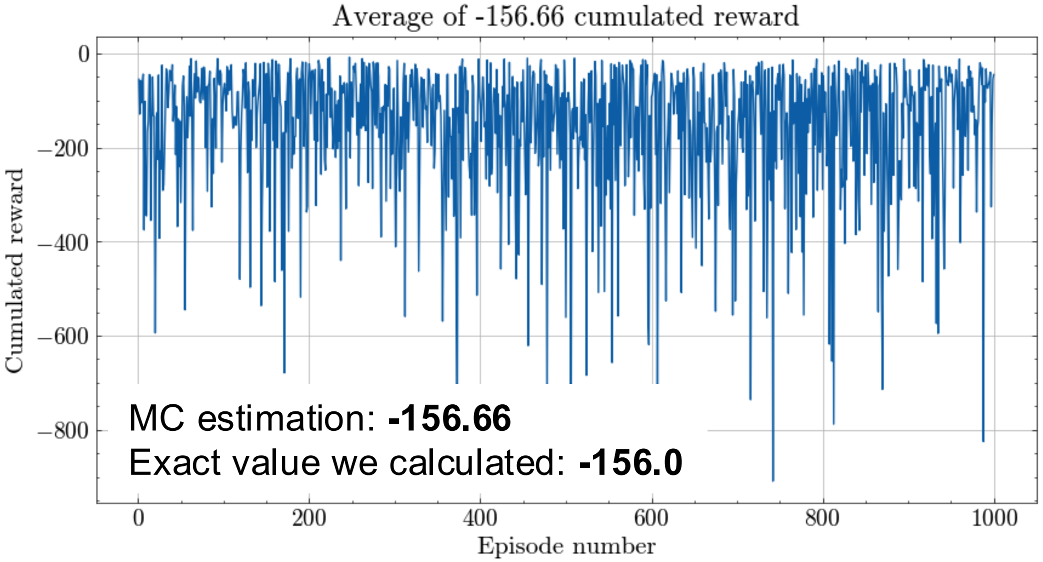
- The first-visit MC variant averages returns only from the first occurrence of each state per episode
- As the number of visits tends to infinity, the **estimator converges** by the law of large numbers:

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

Monte Carlo estimates the expectation in the Bellman equation by empirical averages of full returns

Monte Carlo learning

Let's try this by considering 1000 episodes from our gridworld example for state $s = 0$ (initial state) and a random policy:



MC estimation: **-156.66**
Exact value we calculated: **-156.0**

Can we update before the episode ends?

😏 *well, yes, we can*

- Conceptually simple: **direct empirical estimation of expected return**
- No model $\mathcal{P}$ required: **uses only observed trajectories**
- **Unbiased estimator** of $v^{\pi}(s)$
- Foundation for many **modern policy gradient methods**

- Requires **full episodes before updating** (slow learning, expensive simulation or experiment)
- **High variance**: returns depend on long stochastic trajectories
- **Sample inefficient**: many visits needed for accurate estimates
- **No bootstrapping**: slow propagation of value information

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?

Dynamic programming *exact expectation*

- **Memory**: must store full transition model $\mathcal{P}(s|s,a)$ and value function over all states
- **Computation per update**: requires summing over all possible next states (full expectation)

Monte Carlo *empirical full return*

- **Memory**: must store complete trajectories (or future rewards) until episode termination
- **Computation per update**: computes full returns by summing over the entire remaining trajectory

Temporal difference *stochastic fixed-point bootstrapping*

- **Memory**: stores only current value estimates, no model and no trajectory buffering required
- **Computation per update**: constant per step, uses a single transition (s, r, s')

- TD replaces exact expectation with a **stochastic one-step update** using the current estimate itself (bootstrapping)
- It trades exactness for incremental, constant-memory updates that converge to the Bellman fixed point

💡 *Bootstrapping means updating an estimate using another estimate in place of an exact quantity*

Temporal difference learning

Bellman equation: $\mathcal{V}^{\pi}(s) = \mathbb{E}[r + \gamma \mathcal{V}^{\pi}(s')] = \sum_{a \in \mathcal{A}} \pi\,(a|s) \sum_{s' \in \mathcal{S}} \mathcal{P}^{a}_{s,s'} \left(\mathcal{R}^{a}_{s} + \gamma \mathcal{V}^{\pi}(s')\right)$

Full expectation, sum over all states and actions required

TD update: $\mathcal{V}(s) \leftarrow \mathcal{V}(s) + \alpha\underbrace{[r' + \gamma \mathcal{V}(s')}_{\textbf{Target}} - \mathcal{V}(s)]$

No expectation, one sample only

New estimate ← Old estimate + Step size $\underbrace{\text{(Target - Old estimate)}}_{\textit{Temporal difference error}}$

The target itself depends on the current value estimate (bootstrapping)

- **Online updates:** learns after each transition, no need to wait for episode termination
- **Constant memory and computation per step:** uses only (s, r, s')
- **Lower variance than Monte Carlo:** bootstrapping reduces trajectory-level noise
- **Scales better to continuing tasks:** naturally handles infinite-horizon settings.

- **Bootstrapping introduces bias:** the target depends on current estimates
- **Convergence requires conditions:** appropriate step sizes and sufficient exploration
- **Requires explicit representation of each state,** impractical for very large or continuous state spaces

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

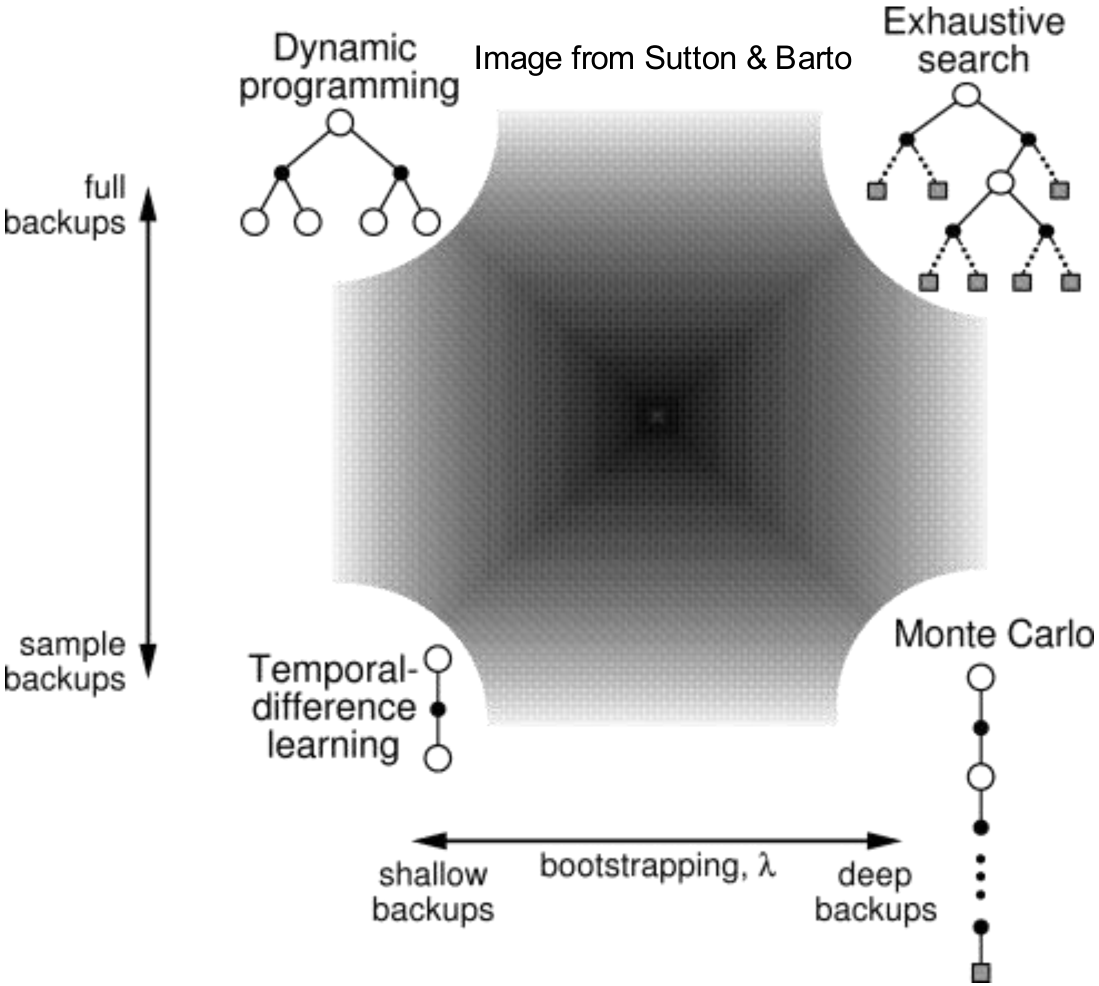


Image from Sutton & Barto

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

- What happens to infinite or continuous MDPs? $\mathcal{V}(s)$?
- Can we identify and enumerate all states and actions?

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

THANK YOU FOR YOUR ATTENTION!

What questions do you have for me?

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

Dr. Andrea Santamaria Garcia
Lecturer at University of Liverpool
Cockcroft Institute

ansantam@liverpol.ac.uk

https://www.linkedin.com/in/ansantam/

https://github.com/ansantam

https://instagram.com/ansantam

https://www.liverpool.ac.uk/people/andrea-santamaria-garcia