Applied Computational Mathematics Lab

TEDDY LAZEBNIK'S COURSE · LECTURE 1

Reinforcement Learning

Lecture 1 · Introduction: what RL is, the agent–environment loop, and what is inside an agent



Teddy Lazebnik · Applied Computational Mathematics Laboratory

BEFORE WE START

How to read these slides

- They follow the order of a standard first lecture on reinforcement learning (RL): what RL is, the formal problem, what is inside an agent, and examples.
- They are written so you can study **without a lecturer**. Every key idea comes with a short explanation, an example or a picture.
- Each part ends with questions, and the answers are at the end.
- Main reading: Sutton & Barto, *Reinforcement Learning: An Introduction*, 2nd ed., 2018 (free online at incompleteideas.net/book/the-book-2nd.html), chapters 1 and 3.



IN PLAIN WORDS

These blue boxes say the same thing as the slide in everyday words.

CHECK YOURSELF

Yellow boxes are questions to check yourself.

TRY IT

Green boxes are small worked examples.

EXAMPLE

White boxes are examples from everyday life.

BEFORE WE START

Today's map

PART 1

What is RL?

Why we want machines that learn, and how RL differs from other learning.

PART 2

The formal problem

Agent, environment, reward, return and value.

PART 3

Inside the agent

Agent state, policy, value function and model.

PART 4

A maze example

All four components on one small problem.

PART 5

Kinds of agents and problems

Value- or policy-based; prediction or control; learning or planning.

PART 6

Examples

A game screen, and a gridworld solved with numbers.



IN PLAIN WORDS

By the end you should be able to say what an RL agent is trying to do, write down the return and the value, and name the parts an agent can have.

Applied Computational Mathematics Lab

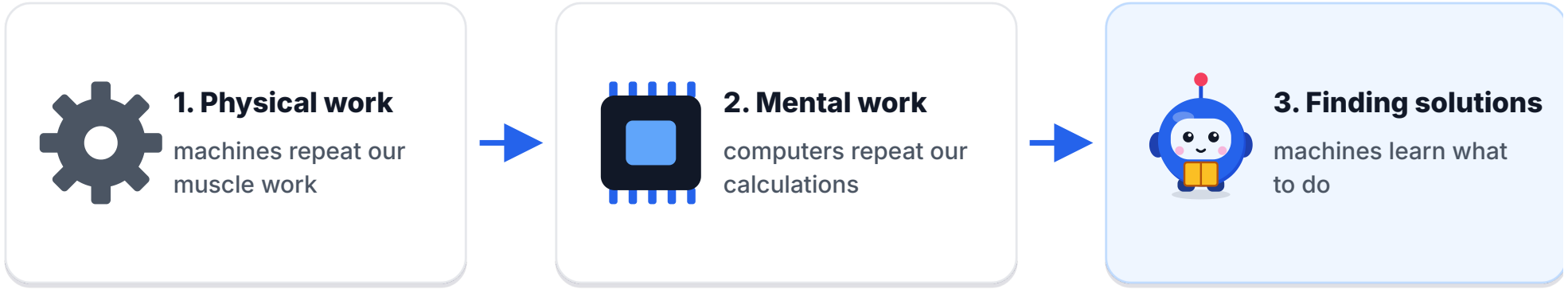
PART 1

What is reinforcement learning?

Why we want machines that learn from experience, and what makes this kind of learning different.



Why we want machines that learn



- **First wave:** machines took over repeated *physical* work (the Industrial Revolution from the mid-1700s, then the Machine Age).
- **Second wave:** computers took over repeated *mental* work (the digital revolution, from the 1950s).
- **Next:** machines that find solutions *themselves*. We only state the problem or the goal; the machine has to learn how to decide.



IN PLAIN WORDS

Before, a person had to know the answer and program it. Now we want to describe **what** we want and let the machine discover **how**.

An old idea: build a mind that learns

- In 1950, Alan Turing asked whether machines can think (*Computing Machinery and Intelligence*).
- He suggested programming something like a **child's** mind, rather than an adult one, and then letting it learn from education and experience.
- He split a mind into three parts: its state at birth, the education it receives, and its other experience.

> **"...why not rather try to produce one which simulates the child's?"**
> Alan Turing, 1950



RL is mostly about the third box: learning from your own experience.

What do we mean by intelligence?

WORKING DEFINITION FOR THIS COURSE

Intelligence is being able to learn to make decisions that achieve goals.

- Three words carry the definition: **learning**, **decisions** and **goals**.
- RL is a mathematical framework built around exactly these three.



IN PLAIN WORDS

Many useful systems have only one or two of the three. An RL agent needs all three together.

SYSTEM	LEARNS FROM EXPERIENCE	ITS ACTIONS CHANGE WHAT COMES NEXT	WORKS TOWARD A GOAL
Thermostat	no	yes	yes
Spam filter	yes	no	yes
RL agent	yes	yes	yes

A spam filter sorts the mail, but its choices don’t change which emails arrive next.

How is RL different from other learning?

- People and animals learn by **interacting** with the world around them.
- Compared with other kinds of machine learning, this learning is
  - **active**: your own actions decide which data you see;
  - **sequential**: what you do now changes what happens next.
- It is **goal-directed**.
- Nobody shows the learner the correct action. It learns from a **reward signal** instead.

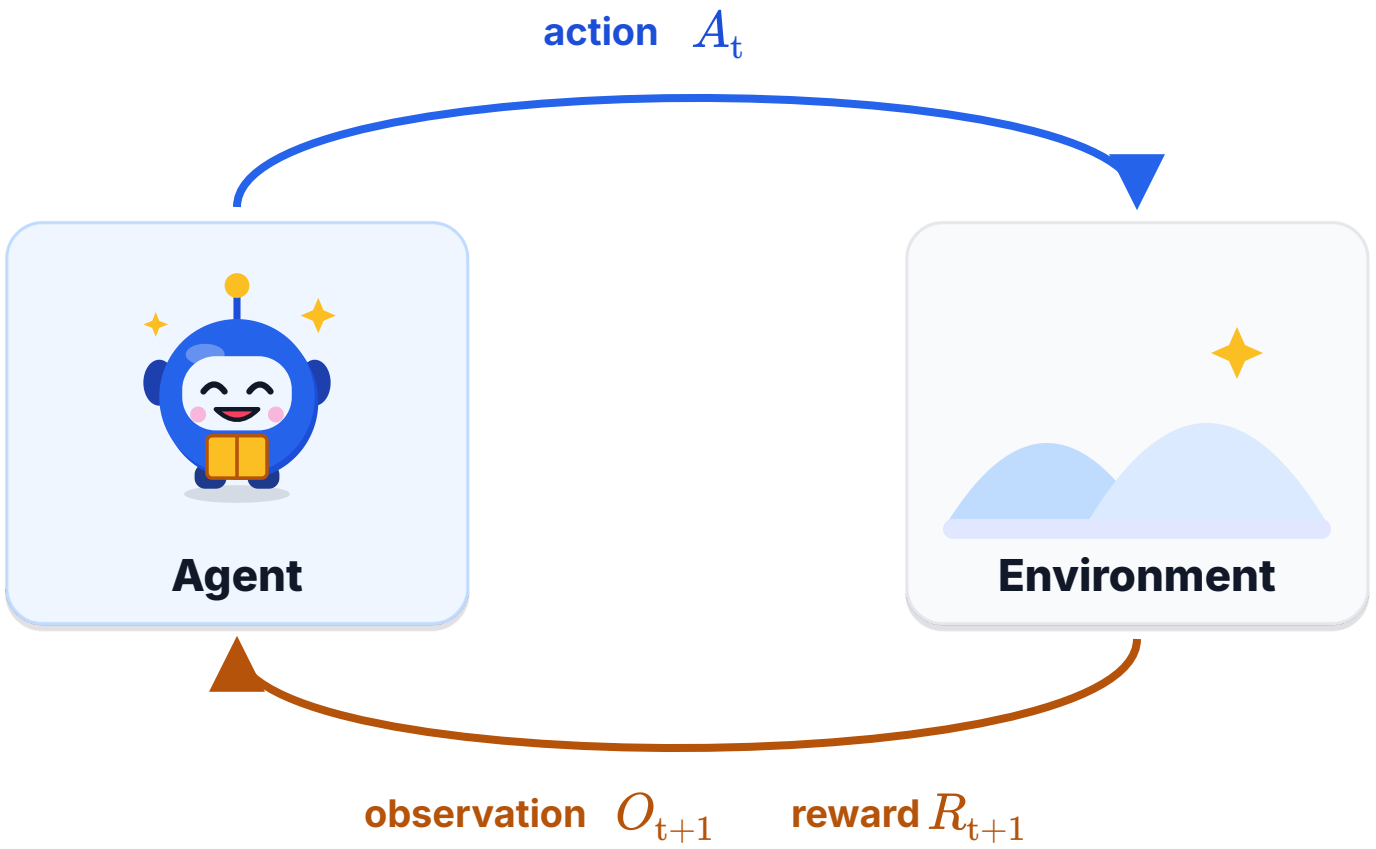
	LEARNS FROM
Supervised	examples with the right answer
Unsupervised	data with no answers (finds structure)
Reinforcement	rewards from its own interaction



IN PLAIN WORDS

A child learning to ride a bike gets no list of correct moves. She tries, wobbles, falls, and slowly does more of what keeps her up.

The interaction loop



- The **agent** acts.
- The **environment** answers with what the agent now sees and a reward.
- Then the loop repeats.

THE GOAL

Maximise the **total reward** collected over many rounds of this loop.

The reward hypothesis

THE ASSUMPTION BEHIND RL

Any goal can be written as maximising the expected total of a single number, the reward, received over time.

See Sutton & Barto (2018), section 3.2

- It is a **hypothesis**, an assumption, not a theorem.
- Choosing the reward is a **design decision**. A badly chosen reward gives strange behaviour.

A GOOD REWARD

Chess. Reward +1 for a win, −1 for a loss, 0 for every other move. Maximising the total means winning.

A RISKY REWARD

A cleaning robot gets +1 for every bit of dust it picks up. It could learn to tip dust out and pick it up again. (A thought example.)

Examples of RL problems

TASK	ACTIONS	OBSERVATIONS	A POSSIBLE REWARD
Fly a helicopter	stick and pedal movements	position, speed, angles	time in the air; staying close to the target
Manage an investment portfolio	how much to buy or sell	prices, holdings, news	gains, or gains minus a penalty for risk
Control a power station	valve and fuel settings	temperatures, pressures, demand	efficiency
Make a robot walk	motor forces	joint angles, touch sensors	distance covered, or speed
Play a video game or board game	moves, or joystick presses	the board or the screen	winning, or the score



IN PLAIN WORDS

What makes these RL problems is the setting, **learning by interacting**, not the method we later use to solve them.

Two different reasons to learn

REASON 1

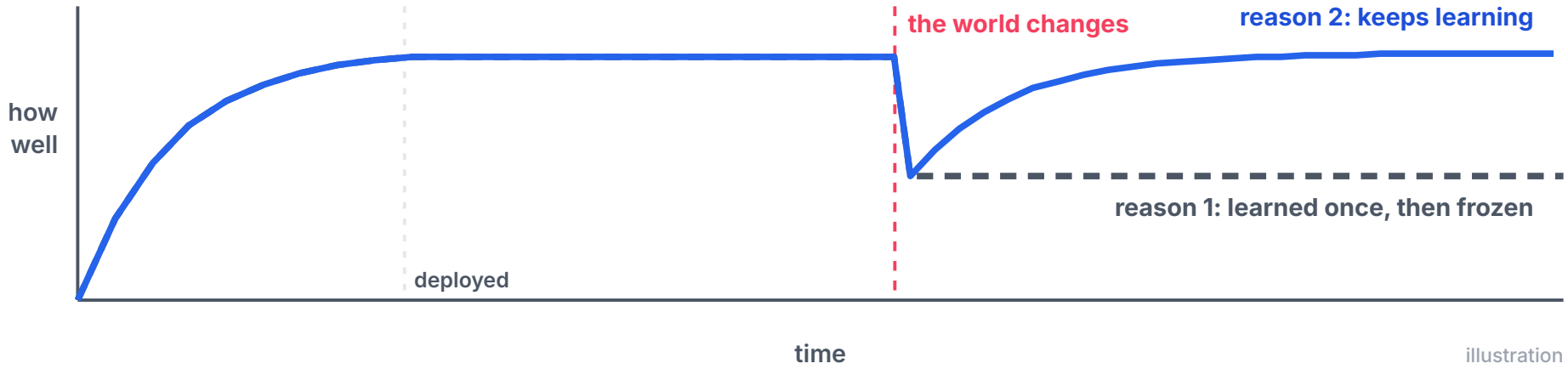
Find a solution

A program that plays chess very well; a factory robot built for one job.

REASON 2

Keep adapting while working

A chess program that adapts to *your* style; a robot that learns to cross ground it has never seen.



illustration

RL offers algorithms for both. Reason 2 is about **continuing to learn efficiently during operation**, not only about generalising.

RL in one sentence

REINFORCEMENT LEARNING

The science and framework of learning to make decisions from interaction.

To do this well, we have to think about:

Time

Decisions come one after another, not all at once.

Long-term consequences

A move now can win or lose the game many moves later.

Gathering experience

If you never try the new café, you never learn that it is good.

Predicting the future

Will this route be slow at 5 pm today?

Uncertainty

The same action can lead to different results.

- The scope is huge: games, robots, recommendations, control, and more.
- Many people see RL as a way of writing down the whole AI problem.

Check yourself: Part 1

CHECK YOURSELF

1. Name two ways in which RL differs from supervised learning.

CHECK YOURSELF

2. State the reward hypothesis in your own words.

CHECK YOURSELF

3. A navigation app learns your favourite routes while you drive. Which of the two reasons to learn is this?

CHECK YOURSELF

4. Pick a task from your own life. What would the actions, the observations and the reward be?





HINT

Answers are at the end of the deck.

Applied Computational Mathematics Lab

PART 2

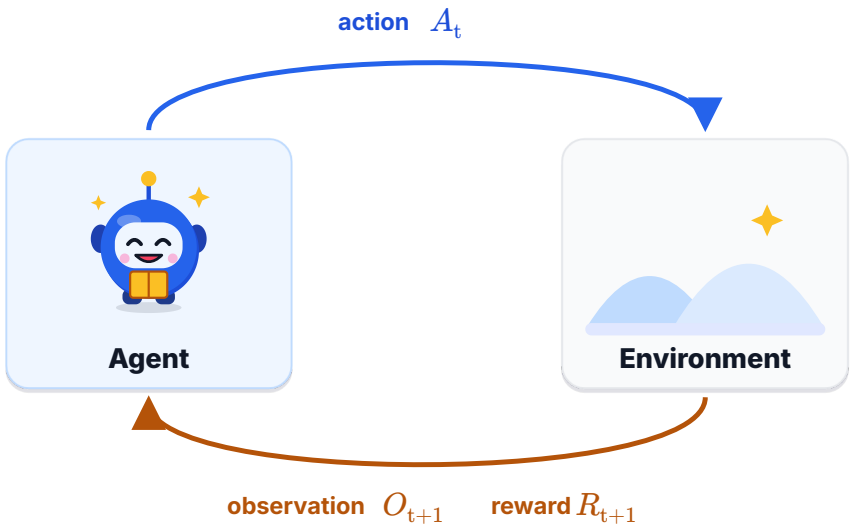
Formalising the RL problem

Agent and environment, reward and return, value and action value.



Agent and environment, step by step

- At each time step t, the **agent**
  - receives an observation O_t and a reward R_t,
  - and takes an action A_t.
- The **environment**
  - receives the action A_t,
  - and emits the next observation O_{t+1} and reward R_{t+1}.



STEP IN THE MAZE (PART 4)	$t = 0$	$t = 1$	$t = 2$
Observation O_t	walls north, south, west	walls north, south	wall north
Reward R_t	none yet	−1	−1
Action A_t	east	east	south

The reward for an action arrives one step later, together with the next observation. That is why it is written R_{t+1}.

Rewards and the return

- A reward R_t is a single number, a **scalar** feedback signal.
- It says how well the agent is doing at step t. It **defines the goal**.
- The agent's job is to maximise the **cumulative** reward, which we call the **return**:

$$G_t = R_{t+1} + R_{t+2} + R_{t+3} + \dots$$

TRY IT

A short game gives rewards 0, 0, 0, +1 and then ends.
From the start, $G_0 = 0 + 0 + 0 + 1 = 1$.
After the second step, $G_2 = 0 + 1 = 1$.



IN PLAIN WORDS

The return is everything still to come from now on, added up. Past rewards don't count.

EXAMPLE: THE MAZE

The shortest route from Start takes 13 steps, each with reward −1, so $G_0 = -13$. After 5 steps only 8 remain, so $G_5 = -8$.

−1	−1	−1	−1	−1	−1	−1	−1	−1	−1	−1	−1	−1

Values

- The **value** of a state s is the **expected** return from that state:

$$\begin{aligned} v(s) &= \mathbb{E}\left[G_t \mid S_t = s\right] \\ &= \mathbb{E}\left[R_{t+1} + R_{t+2} + R_{t+3} + \cdots \mid S_t = s\right] \end{aligned}$$

- The value depends on the actions the agent takes.
- The goal is to **maximise value** by choosing good actions.
- Rewards and values say how good states and actions are. No one supplies labelled answers.

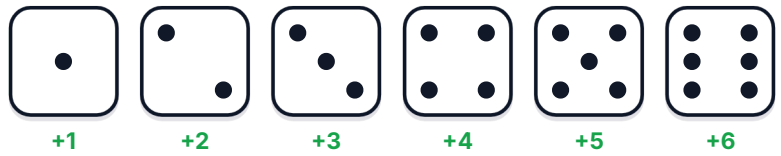


IN PLAIN WORDS

"Expected" means the average over everything that might happen. Value is the return you get **on average** from here.

TRY IT

In state s you roll a fair die, get the number as your reward, and the game ends. The return is 1, 2, …, 6, each equally likely, so $v(s) = (1+2+3+4+5+6)/6 = 3.5$.



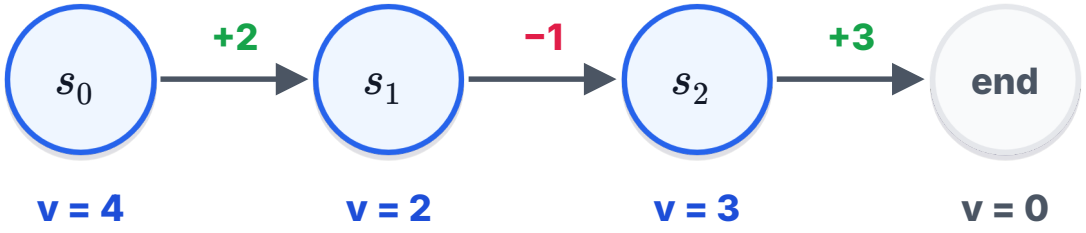
Returns and values are recursive

- The return from now is the next reward plus the return from the next step:

$$G_t = R_{t+1} + G_{t+1}$$

- So the value has the same shape:

$$v(s) = \mathbb{E}\left[R_{t+1} + v(S_{t+1}) \mid S_t = s\right]$$



work backwards: each value = next reward + next value



IN PLAIN WORDS

How good is it to be here? It is the reward for the next step plus how good the next place is. This one idea drives most RL algorithms.

TRY IT

Start from the end: $v(s_2) = 3$. Then $v(s_1) = -1 + 3 = 2$, and $v(s_0) = 2 + 2 = 4$.

Maximising value by choosing actions

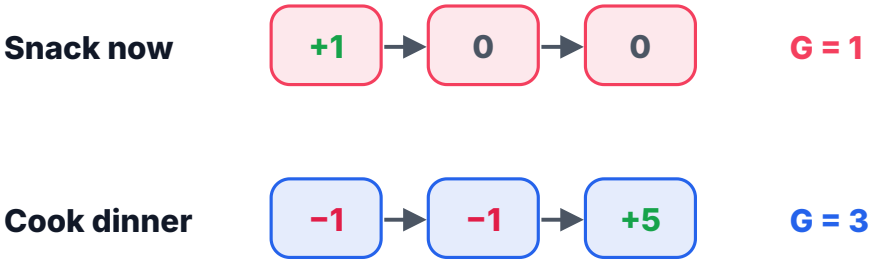
- Goal: choose actions that maximise value.
- Actions can have **long-term consequences**, and reward may be **delayed**.
- It can be better to give up some reward now to get more later.
- Examples:
  - refuelling a helicopter (it may prevent a crash hours later);
  - a defensive move in a game (it may help you win later);
  - learning a new skill (costly and slow at first).

POLICY

A mapping from states to actions is called a **policy**.

EXAMPLE: DINNER

A snack now gives +1 straight away. Cooking costs effort first (−1, −1), then a good meal (+5).





IN PLAIN WORDS

Looking only at the next step picks the snack. Looking at the whole return picks dinner: 3 is more than 1.

Action values

- We can also condition the value on the **first action**:

$$q(s,a) = \mathbb{E}\left[G_t \mid S_t = s, A_t = a\right]$$
$$= \mathbb{E}\left[R_{t+1} + R_{t+2} + \cdots \mid S_t = s, A_t = a\right]$$

- $v(s)$ asks how good it is to **be** in state s.
- $q(s,a)$ asks how good it is to **take action** a in state s.



IN PLAIN WORDS

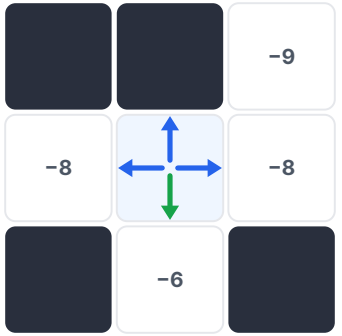
With q the choice is easy: in state s, compare $q(s,a)$ for every action and pick the largest.

TRY IT

In the example on the right, the largest q is -7, for south. It equals the value of the square: $v(s) = \max_a q(s,a)$ when the agent acts at its best.

EXAMPLE: ONE SQUARE OF THE MAZE

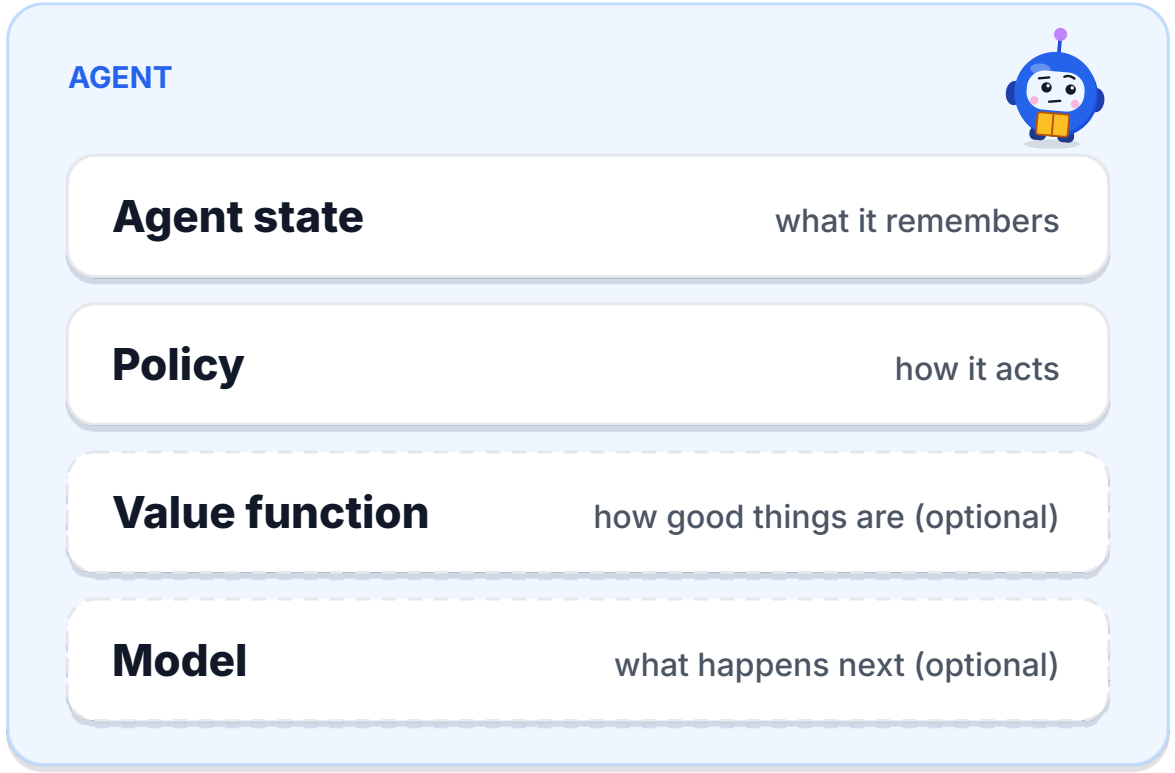
The middle square has value −7. An action costs −1, then you continue from where you land; a wall leaves you in place.



ACTION	$q(s,a)$
south	**−1 + (−6) = −7**
north	−1 + (−7) = −8
east, west	−1 + (−8) = −9

Core concepts

- **Environment**: the dynamics of the problem.
- **Reward signal**: what the goal is.
- **Agent**, which contains
  - an **agent state**,
  - a **policy**,
  - maybe a **value function** estimate,
  - maybe a **model**.
- Next, we look inside the agent.

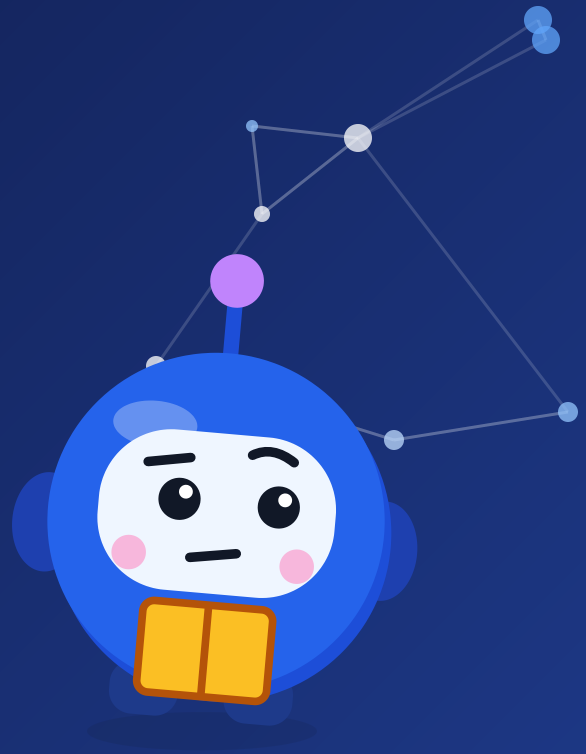


Applied Computational Mathematics Lab

PART 3

Inside the agent

Agent state, policy, value function and model, one at a time.



Component 1: the agent state

AGENT



Agent state — what it remembers

Policy — how it acts

Value function — how good things are (optional)

Model — what happens next (optional)



IN PLAIN WORDS

Before deciding anything, the agent needs some summary of its situation. That summary is its **state**.

EXAMPLES OF AN AGENT STATE

- **Chess program:** the board, and whose turn it is.
- **Self-driving car:** where nearby cars are, and how fast they moved over the last second.
- **Chatbot:** the conversation so far.
- **Bit in the maze:** the walls around it, or more, as we will see.

The environment state

- The **environment state** is the environment's own internal state.
- It is usually **invisible** to the agent.
- Even when it is visible, most of it may be irrelevant to the decision.

EXAMPLE

A robot vacuum: the environment state is the whole flat, with furniture, dust and the people in it. The robot's sensors see only a small circle around it.



IN PLAIN WORDS

The environment state is everything that is true about the world right now. The agent only ever gets a glimpse of it.

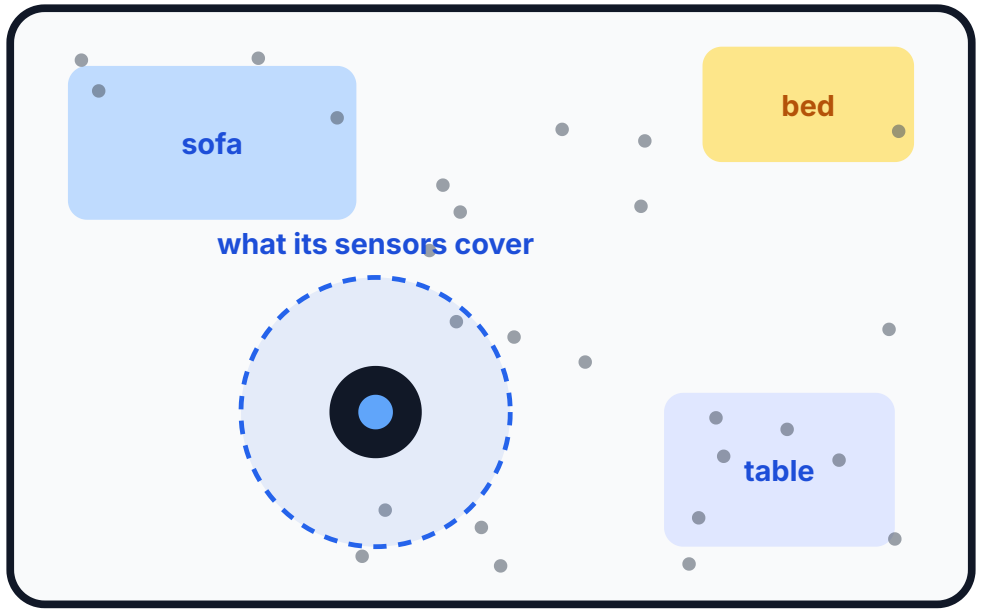
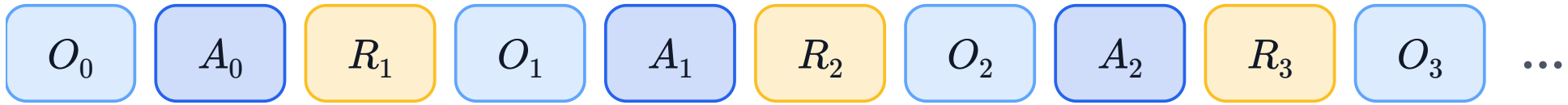


Illustration: the full flat (environment state) and what the sensors cover.

History and agent state

- The **history** is the full sequence of observations, actions and rewards so far:

$$H_t = O_0, A_0, R_1, O_1, \ldots, O_{t-1}, A_{t-1}, R_t, O_t$$



- For example, a robot's full stream of sensor readings and motor commands.
- The agent uses the history to build its **agent state** S_t.

TRY IT

In the maze, after two steps: H_2 = (walls N, S, W), east, −1, (walls N, S), east, −1, (wall N).



IN PLAIN WORDS

The history is the diary of everything that happened. The agent state is the agent's working summary of that diary.

Fully observable environments

FULL OBSERVABILITY

The agent sees the full environment state: the observation **is** the environment state.

- Then the agent state can simply be the latest observation:

$$S_t = O_t = \text{environment state}$$

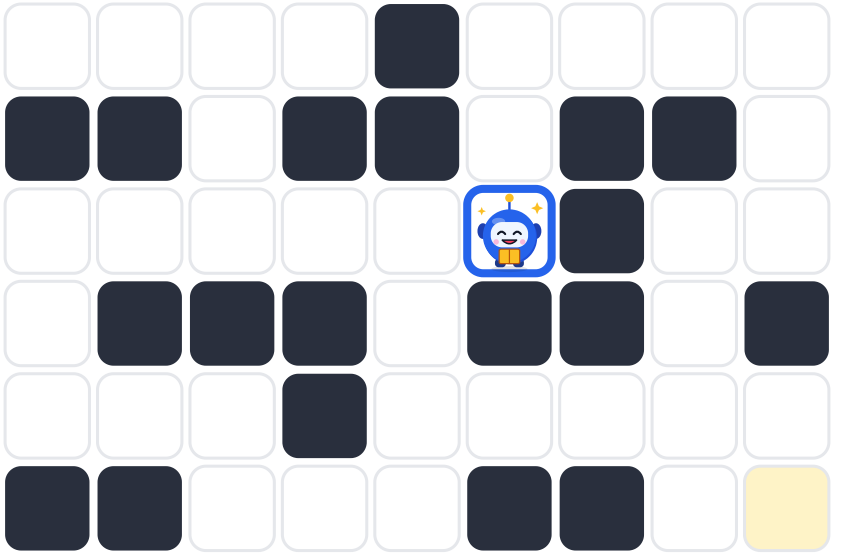


IN PLAIN WORDS

Chess is close to this: the board shows almost everything there is to know about the position.

EXAMPLE: THE MAZE WITH A GPS

Bit is told its square, say row 2, column 5. The walls never move, so knowing the square is knowing everything.



O_t = (row 2, column 5) is a full description of the situation.

Close to fully observable: CHESS · GO · TIC-TAC-TOE · THE MAZE WITH A GPS

Markov decision processes

DEFINITION

A decision process is **Markov** if the next reward and state depend only on the current state and action, not on the rest of the history:

$$p(r, s \mid S_t, A_t) = p(r, s \mid H_t, A_t)$$

- The state then holds everything from the history that matters for prediction. It need not hold everything; adding more history just doesn't help.
- So once the state is known, the history can be thrown away.

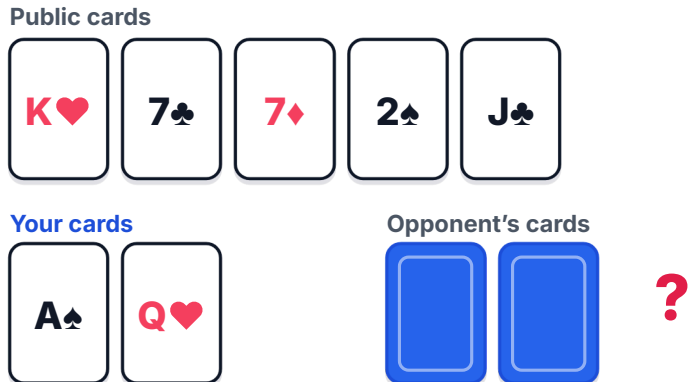
- The full environment state plus agent state is Markov, but large.
- The full history H_t is Markov, but keeps growing.
- Usually the agent state S_t is a **compression** of H_t. From now on, S_t means the **agent** state.



A flying ball: its position alone is not Markov, because you can't tell where it goes next. Position and velocity together are (ignoring wind).

Partially observable environments

- **Partial observability:** the observations are not Markov.
  - A robot with a camera isn't told where it is.
  - A poker player sees only the public cards and its own.
- Using the observation as the state is then not Markov.
- This setting is called a **partially observable** Markov decision process (**POMDP**).
- The environment state can still be Markov; the agent just can't see it.
- We may still be able to **build** a Markov agent state.



The same view can hide very different situations: the opponent could hold almost anything. (Illustration.)



IN PLAIN WORDS

If two different situations look the same to you, the current view alone can't tell you what to do. You need memory.

Updating the agent state

- The agent's actions depend on its state.
- The agent state is a function of the history, for instance $S_t = O_t$.
- More generally, a **state update function** u folds each new step into the state:

$$S_{t+1} = u\left(S_t, A_t, R_{t+1}, O_{t+1}\right)$$

- The agent state is often much smaller than the environment state.

UPDATE RULE u	WHAT THE STATE KEEPS
$S_{t+1} = O_{t+1}$	only the latest observation
$S_{t+1} = (O_t,\ O_{t+1})$	the last two observations
$S_{t+1} = S_t + \text{move}$	your position (example 1)
$S_{t+1} = 0.8\, S_t + 0.2\, O_{t+1}$	a smoothed sensor reading (example 2)



IN PLAIN WORDS

Like a running summary: take the old summary, add what just happened, and you get the new summary. You never need to reread the whole diary.

Example 1: keeping track of where you are



Bit's first steps from Start (numbered).



IN PLAIN WORDS

Two numbers are enough here. In this maze the position describes the whole situation, so this agent state is **Markov**.

THE UPDATE RULE

State = (row, column). Bit knows where it started. Each step, if the side it moves toward was open, it adds the move to its position.

t	S_t	A_t	THEN SEES	S_{t+1}
0	(0, 0)	east	walls N, S	(0, 1)
1	(0, 1)	east	wall N	(0, 2)
2	(0, 2)	south	walls E, W	(1, 2)
3	(1, 2)	south	wall S	(2, 2)
4	(2, 2)	east	walls N, S	(2, 3)

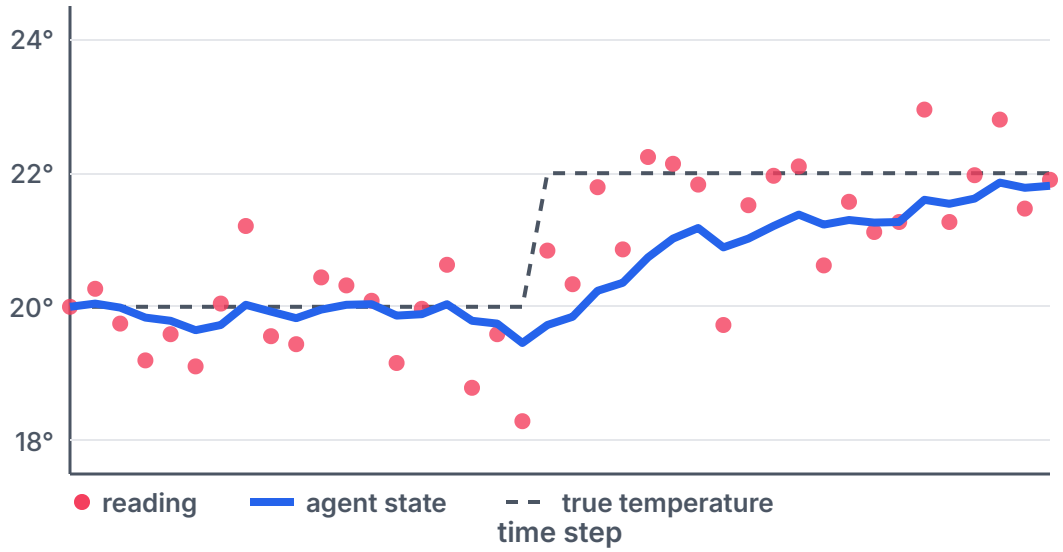
Example 2: smoothing a noisy sensor

- A thermometer's readings jump around: every reading has some noise.
- Instead of trusting the latest reading, the agent keeps a running average as its state:

$$S_{t+1} = 0.8\, S_t + 0.2\, O_{t+1}$$

TRY IT

$S_t = 20.0$ and the new reading is 21.0:
$S_{t+1} = 0.8 \times 20.0 + 0.2 \times 21.0 = 20.2.$



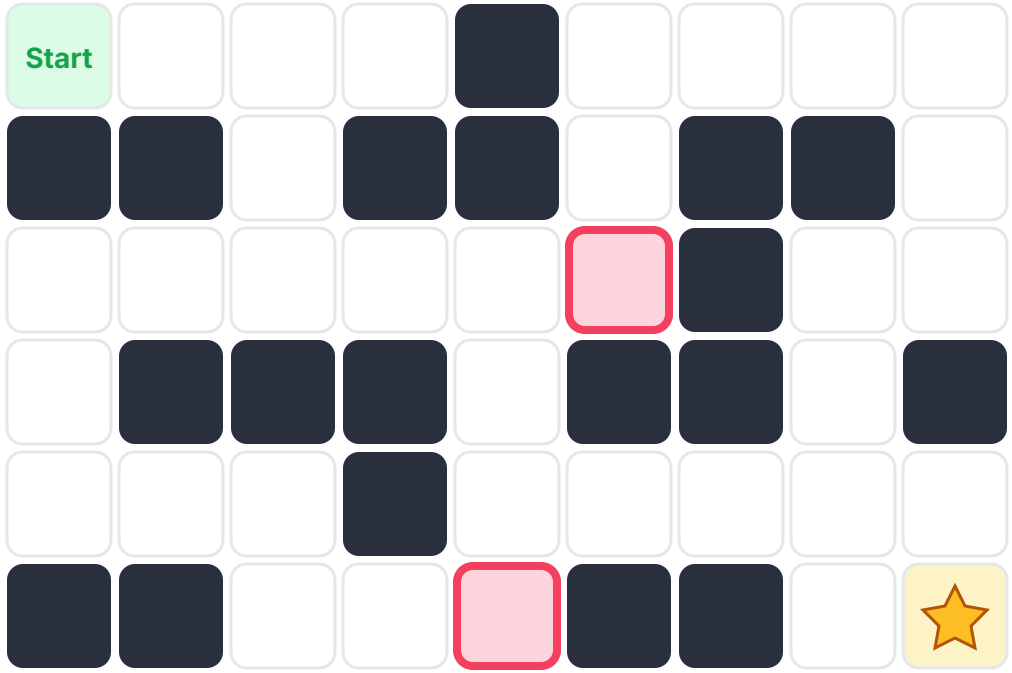
Simulated readings. A heater is switched on at step 19.



IN PLAIN WORDS

The state ignores most of the noise, but it reacts a little late when the heater comes on. Choosing u is a trade-off.

When two places look the same



The full maze is the environment state. Suppose the agent sees only its own square.



What the agent sees in **both** pink squares: walls to the east and south, open to the north and west.

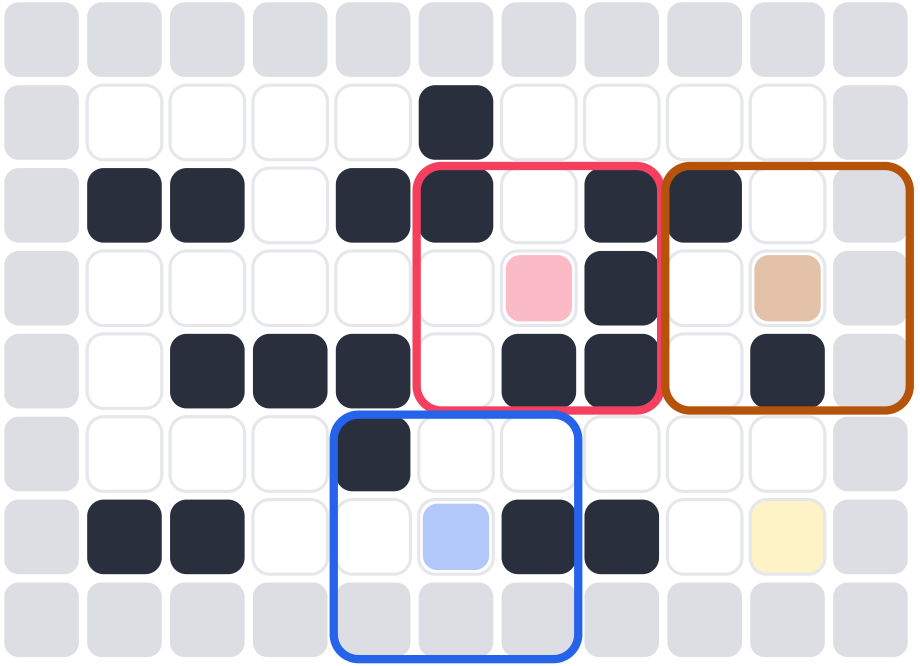
- The two observations are **identical**.
- But the best move differs: **west** in the upper square, **north** in the lower one.
- So “last observation” is **not** a Markov state here.



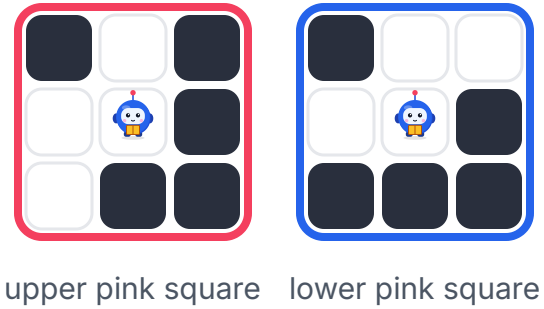
IN PLAIN WORDS

With only this view, the agent must guess. It needs memory or a wider view. Next: what a 3×3 view changes.

A wider view: the agent sees 3×3 squares

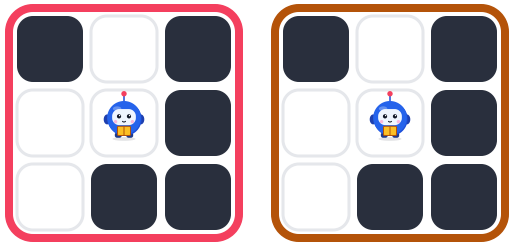


Each frame is what Bit sees from the square in its centre.
Outside the maze counts as wall.



upper pink square lower pink square

Different views. With a 3×3 view the agent can tell these squares apart and act correctly in both.



value −8 value −5

Still the same view. These two squares look identical in 3×3, yet one is 8 steps from the goal and the other 5. The best move is west in both.



IN PLAIN WORDS

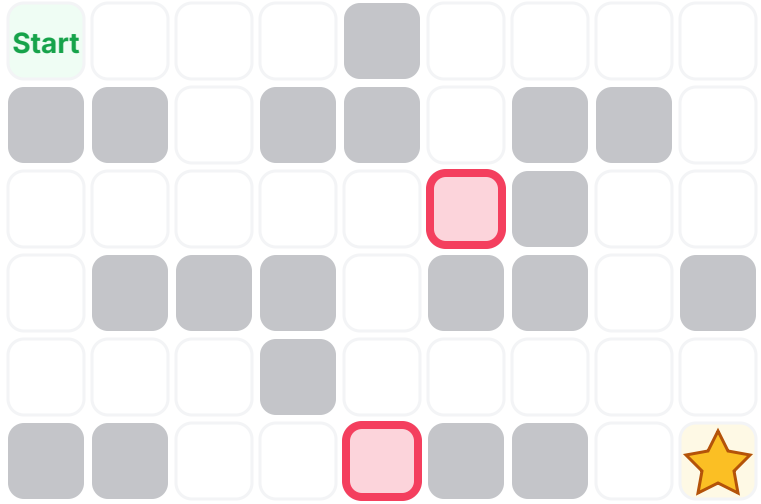
A wider view helps, but it is still not Markov for predicting values. Here it is enough for **acting**: squares with the same 3×3 view always share a best move.

How could we make the state Markov?

CHECK YOURSELF

In this maze, how could you build an agent state that tells the two pink squares apart, whatever the reward?

- Some ideas:
  - remember the **last few observations and actions**;
  - use a **wider view** (the 3×3 view helps, but is not fully Markov);
  - **track your position** from the start (example 1);
  - keep the **whole history** (it works, but grows forever).



The two squares that look the same with a one-square view.



IN PLAIN WORDS

The position tracker solves it with just two numbers: the pink squares are (2, 5) and (5, 4), never the same. This only works if the agent knows where it started and its moves are reliable.

Choices for the agent state

AGENT STATE	TRADE-OFF	IN OUR MAZE
$S_t = O_t$ **last observation**	simple, but may not be enough	confuses the two pink squares
$S_t = H_t$ **complete history**	Markov, but may be too large	works, but grows by three items every step
$S_t = u(S_{t-1}, A_{t-1}, R_t, O_t)$ **general update**	flexible, but how do we pick or learn u?	the position tracker: two numbers, and Markov here

- A fully Markov agent state is often **not feasible**.
- What matters more: the state should support **good policies** and **good value predictions**.



IN PLAIN WORDS

The 3×3 view is a good example: not Markov, yet good enough to act well in this maze.

Component 2: the policy

AGENT



- Agent state — what it remembers
- **Policy** — how it acts
- Value function — how good things are (optional)
- Model — what happens next (optional)



IN PLAIN WORDS

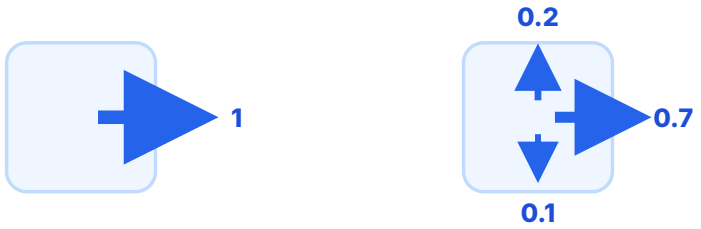
A policy is the agent's rulebook: for every state, what to do.

- A **policy** defines the agent's behaviour.
- It maps an agent state to an action.

TWO KINDS

Deterministic: $A = \pi(S)$

Stochastic: $\pi(A \mid S) = p(A \mid S)$



TRY IT

Stochastic: in this state Bit moves east with probability 0.7, north 0.2 and south 0.1, like rolling a weighted die.

Component 3: the value function

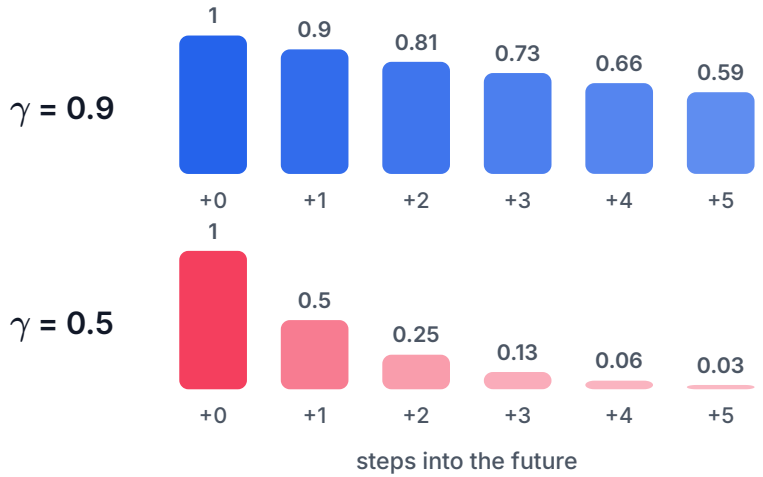
- The true value function is the expected return when following a policy π:

$$
\begin{aligned}
v_\pi(s) &= \mathbb{E}\left[G_t \mid S_t = s, \pi\right] \\
&= \mathbb{E}\left[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots \mid S_t = s, \pi\right]
\end{aligned}
$$

- We add a **discount factor** $\gamma \in [0, 1]$ that trades off immediate and long-term rewards.
- The value depends on the policy. We use it to judge states and to choose between actions.

TRY IT

With $\gamma = 0.9$, a reward $R_{t+4} = 10$ counts as $\gamma^3 \times 10 = 7.29$ in G_t.



How much a reward counts, γ^k, when it arrives k steps after the next one.



IN PLAIN WORDS

Why discount? The future is uncertain, sooner is often better, and the sum stays finite in tasks that never end.

Bellman equations

- With discounting the return is still recursive, $G_t = R_{t+1} + \gamma G_{t+1}$, and so is the value:

$$v_\pi(s) = \mathbb{E}\left[R_{t+1} + \gamma\, v_\pi(S_{t+1}) \mid S_t = s,\ A_t \sim \pi(s)\right]$$

- This is a **Bellman equation** (Bellman, 1957). Here $a \sim \pi(s)$ means a is chosen by π in state s.
- A similar equation holds for the **optimal** (highest possible) value. It does not depend on any policy:

$$v_*(s) = \max_a \mathbb{E}\left[R_{t+1} + \gamma\, v_*(S_{t+1}) \mid S_t = s,\ A_t = a\right]$$



IN PLAIN WORDS

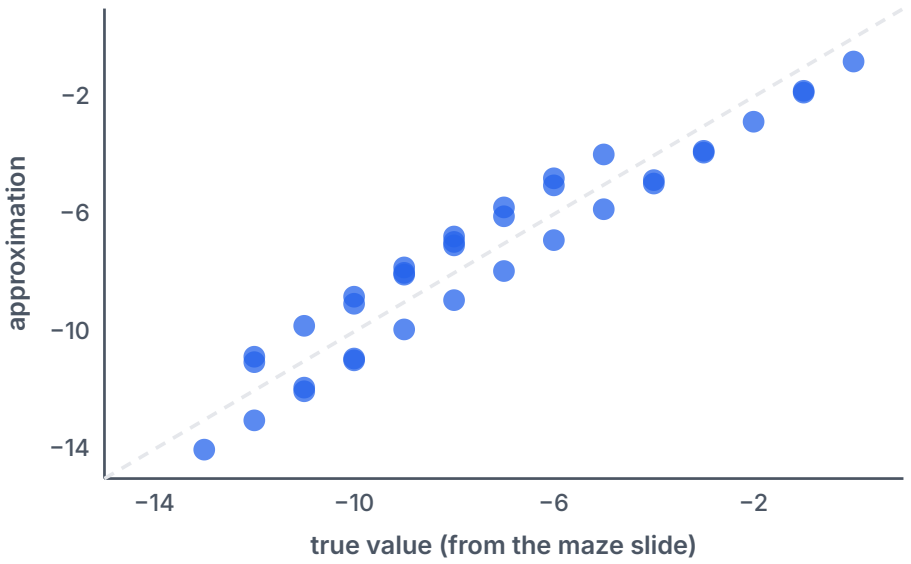
Bellman equations turn "how good is the future?" into a local check between neighbouring states. Most algorithms in this course are built on them.

TRY IT

In the maze ($\gamma = 1$): the square with value −7 has a south neighbour with value −6, and $-7 = -1 + (-6)$. The equation holds.

Approximating values

- Agents often **approximate** value functions rather than store them exactly.
- Much of this course is about algorithms that learn these approximations efficiently.
- With an accurate value function we can behave **optimally**.
- With good approximations we can behave **well**, even in huge problems we could never solve exactly.





IN PLAIN WORDS

Nobody can list the value of every chess position. A good guess, learned from experience, is often enough to play strongly.

EXAMPLE: OUR MAZE

Its 35 values, approximated by just 3 numbers:
$v(\text{row}, \text{col}) \approx -14.0 + 1.05 \cdot \text{row} + 0.99 \cdot \text{col}$.
Average error: about 1 step.

Component 4: the model

- A **model** predicts what the environment will do next.
- For example, $\mathcal{P}$ predicts the next state:

$$\mathcal{P}(s, a, s') \approx p\left(S_{t+1} = s' \mid S_t = s, A_t = a\right)$$

- and $\mathcal{R}$ predicts the next reward:

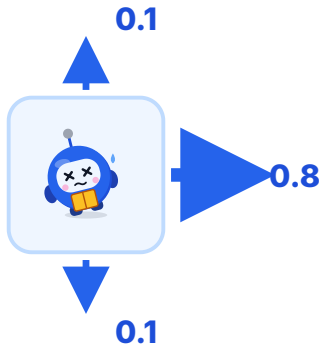
$$\mathcal{R}(s, a) \approx \mathbb{E}\left[R_{t+1} \mid S_t = s, A_t = a\right]$$



IN PLAIN WORDS

A model is the agent's own simulator: "if I do this here, I expect that to happen."

- A model does not hand us a good policy; we would still need to **plan** with it.
- Models can also be stochastic (generative).



action: east, on a slippery floor

A stochastic model on a slippery floor: $\mathcal{P}$ gives 0.8 for the square to the east and 0.1 each for north and south. In our maze the model is exact: each move lands where it points, and $\mathcal{R}(s, a) = -1$.

Check yourself: Parts 2 and 3

CHECK YOURSELF

5. With $\gamma = 0.5$, rewards 1, 1, 1 and then nothing: what is G_0?

CHECK YOURSELF

6. Why is the last observation not always a good agent state?

CHECK YOURSELF

7. What is the difference between $v_\pi(s)$ and $v_*(s)$?

CHECK YOURSELF

8. Does a model, on its own, tell the agent what to do?

CHECK YOURSELF

9. In the maze, a square's south neighbour has value −6. What is $q(s, \text{south})$?



Applied Computational Mathematics Lab

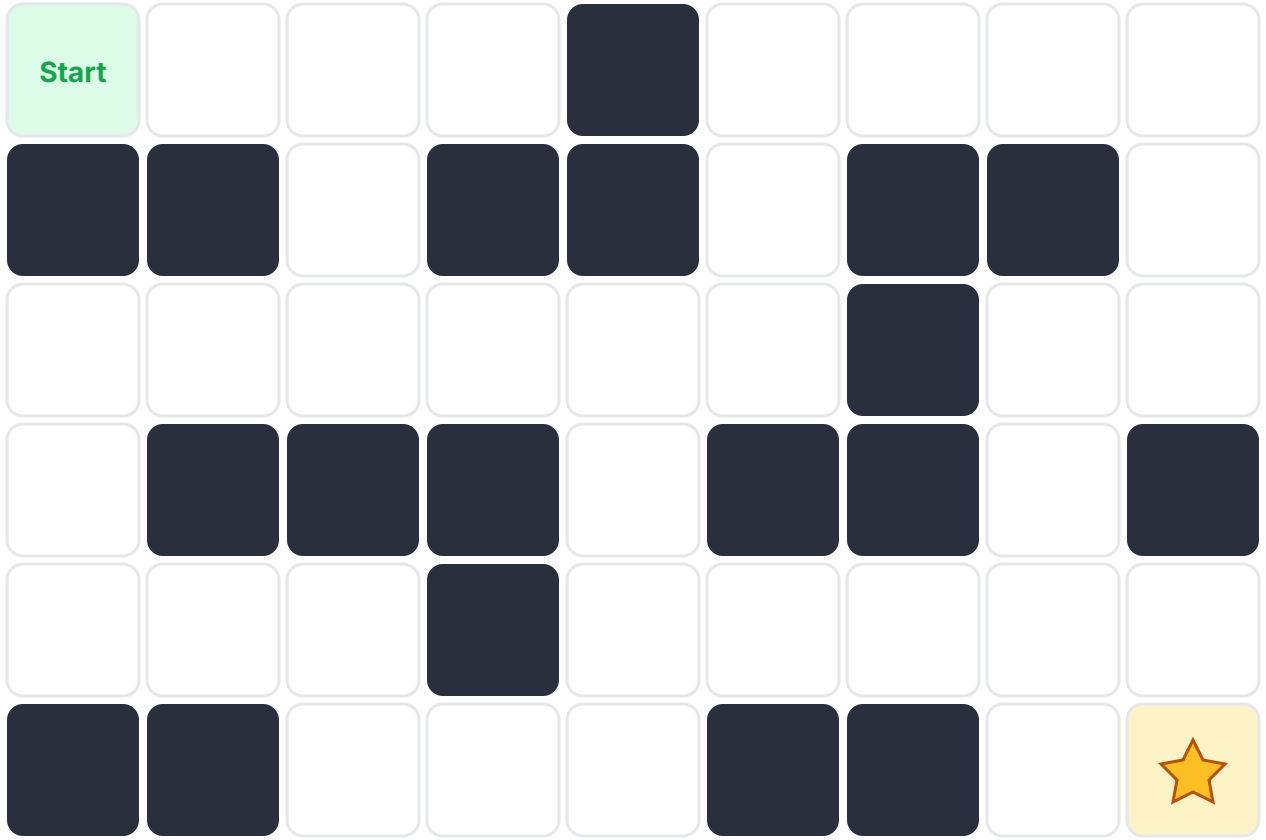
PART 4

An example: the maze

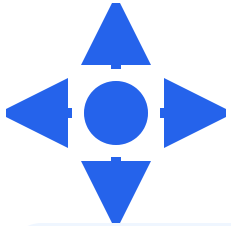
One small problem, seen through the policy, the value function and the model.



The maze



- **Rewards:** −1 for every time step
- **Actions:** north, east, south, west
- **Walls:** moving into a wall leaves you in place (still −1)
- **States:** the agent's square
- The episode ends at the star.

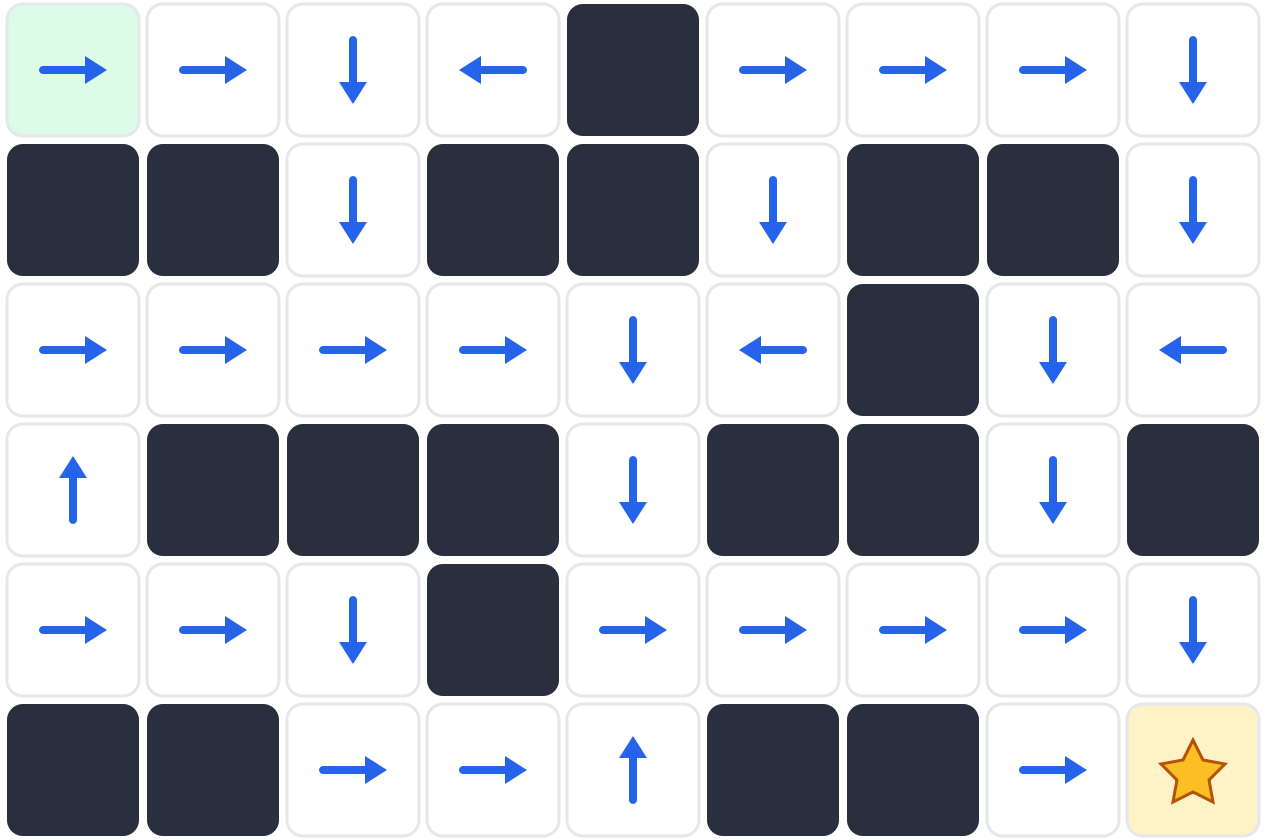




IN PLAIN WORDS

A reward of −1 per step means "be quick": the fewer steps to the goal, the better.

Maze: a policy



- Each arrow shows the policy $\pi(s)$: the action taken in square s.
- This policy follows a shortest route to the goal from every square.



IN PLAIN WORDS

A policy is a full rulebook: it has an answer for **every** square, not just the ones on your route.

TRY IT

Follow the arrows from Start and count the steps to the star. You should get 13.

Maze: the value function



- Each number is the value $v_\pi(s)$ of the square under the arrow policy.
- Here it is simply minus the number of steps to the goal. From the start it takes 13.

TRY IT

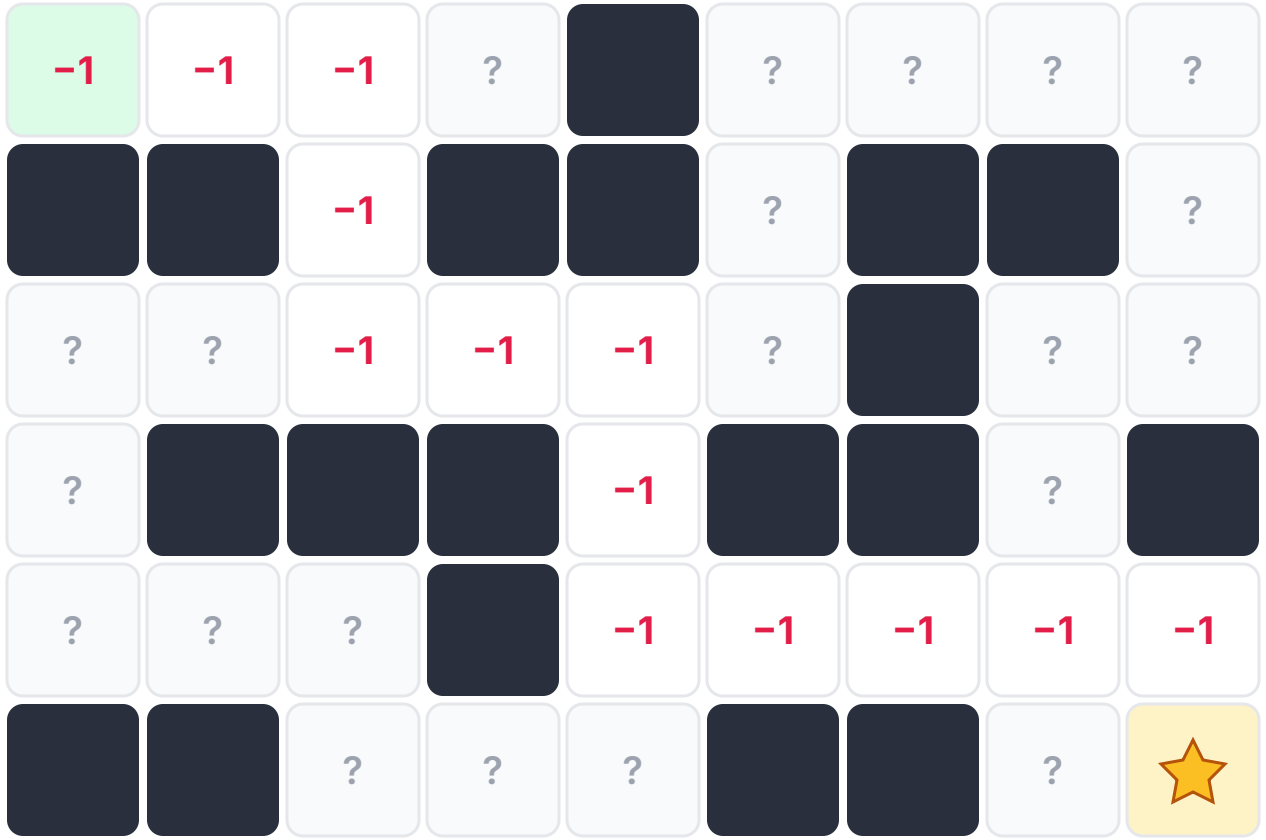
Check the Bellman equation on any square: its value is -1 plus the value of the square its arrow points to.



IN PLAIN WORDS

Values grow (get closer to 0) as you approach the goal. Picking the neighbour with the higher value is a policy too.

Maze: a learned model



- Suppose the agent has walked the route from start to goal once.
- Its model knows what each move does, and the reward (−1), only on the squares it visited.
- Everything else is still a question mark.



IN PLAIN WORDS

A model learned from experience is usually **partial** and can be **wrong**. Planning with it is only as good as the model.

Applied Computational Mathematics Lab

PART 5

Kinds of agents and problems

Which components an agent uses, and which sub-problems RL splits into.



Agent categories

VALUE-BASED

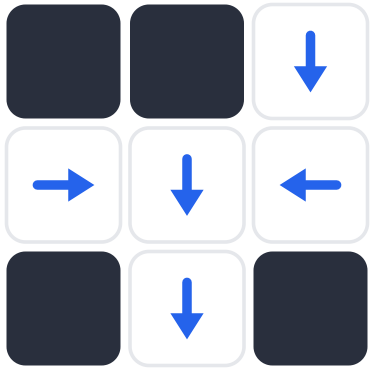
Value function; no explicit policy.



It moves to the neighbour with the highest value.

POLICY-BASED

Policy; no value function.



It just follows its arrows.

ACTOR-CRITIC

Policy (the actor) and **value function** (the critic).



Arrows to act, numbers to judge.



IN PLAIN WORDS

An actor-critic agent has a doer and a judge: the policy acts, and the value function grades how it went.

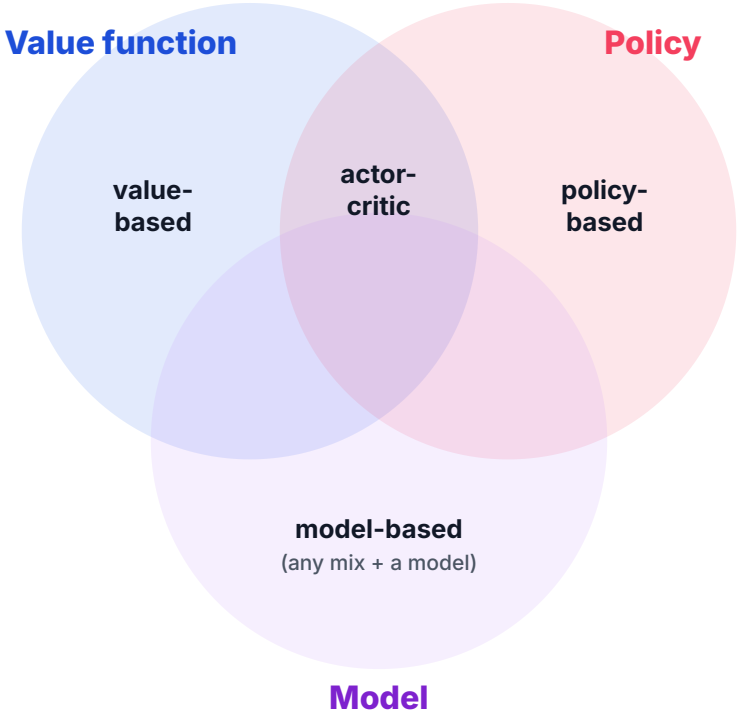
Model-free and model-based

- **Model-free**
  - a policy and/or a value function
  - no model
- **Model-based**
  - a model
  - optionally a policy and/or a value function



IN PLAIN WORDS

Model-free agents learn what to do directly from experience. Model-based agents can also “think ahead” in their own simulator.



Model-free agents live outside the Model circle.

Prediction and control

- **Prediction:** evaluate the future for a given policy.
- **Control:** optimise the future, that is, find the best policy.
- The two are closely linked. The best policy is the one with the highest value:

$$\pi_*(s) = \operatorname*{argmax}_{\pi} v_\pi(s)$$

CHECK YOURSELF

If we could predict everything, would we need anything else?



IN PLAIN WORDS

Prediction asks “how will this go if I keep doing this?”. Control asks “what should I do to make it go best?”.

PREDICTION

“If I always take the bus, how long is my commute on average?”

CONTROL

“Which way should I travel to get there fastest?”

Part 6 shows both on a gridworld.

Learning and planning

LEARNING



- The environment is initially **unknown**.
- The agent **interacts** with it.

EXAMPLE

Learning to ride a bike: you only find out by trying.

PLANNING



- A **model** is given or learnt.
- The agent plans **inside the model**, with no outside interaction.

EXAMPLE

Planning a trip on a map: you try routes on paper before you leave.



IN PLAIN WORDS

Planning is also called reasoning, pondering, thinking or search. An agent can do both: learn a model from experience, then plan inside it. Lecture 9 is about this.

All the components are functions

COMPONENT	MAPS	IN THE MAZE
Policy	$\pi : \mathcal{S} \to \mathcal{A}$ (or to probabilities over $\mathcal{A}$)	$\pi(2,4) = \text{south}$
Value function	$v : \mathcal{S} \to \mathbb{R}$	$v(2,4) = -7$
Model	$m : \mathcal{S} \to \mathcal{S}$ and/or $r : \mathcal{S} \to \mathbb{R}$	$m((2,4), \text{south}) = (3,4),\ r = -1$
State update	$u : \mathcal{S} \times \mathcal{O} \to \mathcal{S}$	the position tracker: old square plus the move gives the new square

- So we can learn them with, for example, neural networks and deep learning.
- Careful: RL often breaks supervised-learning assumptions, such as independent samples (i.i.d.) and a fixed (stationary) data distribution.



IN PLAIN WORDS

In RL the data depends on the agent's own choices, and it changes as the agent learns. That is what makes deep RL tricky, and a rich, active research field.

Applied Computational Mathematics Lab

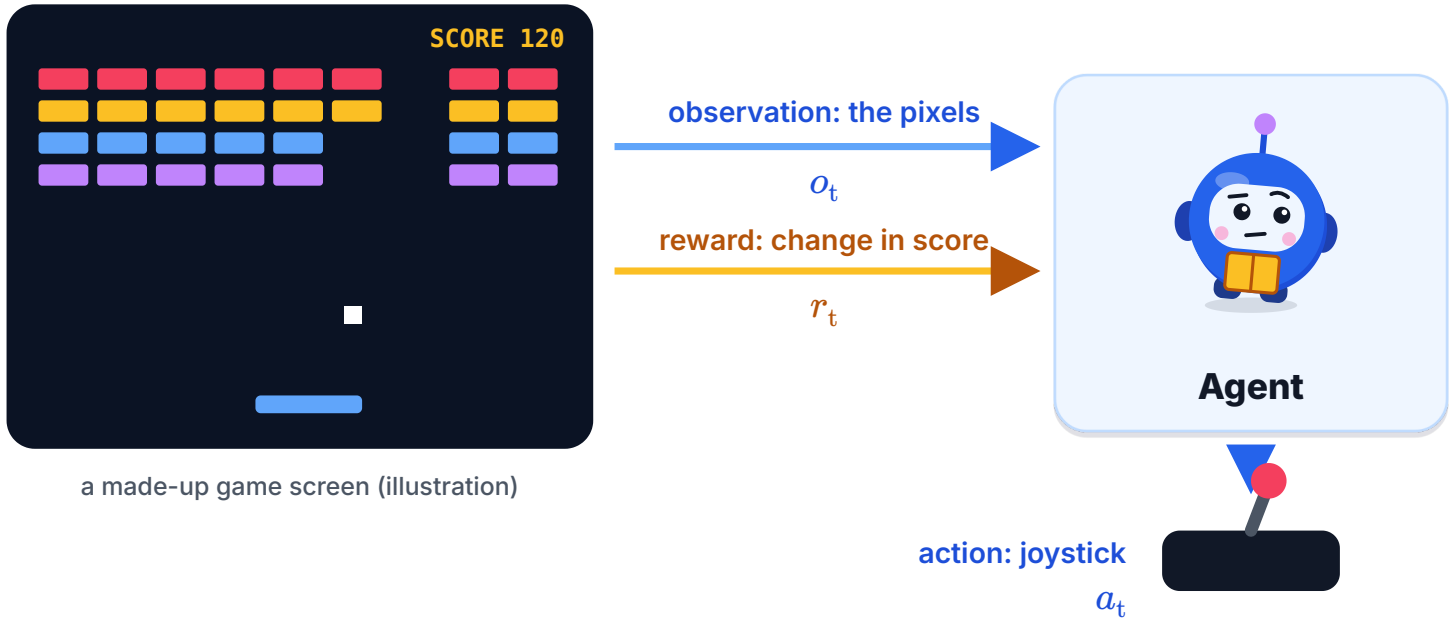
PART 6

Examples

A game screen as an RL problem, and a small gridworld solved with numbers.



A video game as an RL problem



a made-up game screen (illustration)

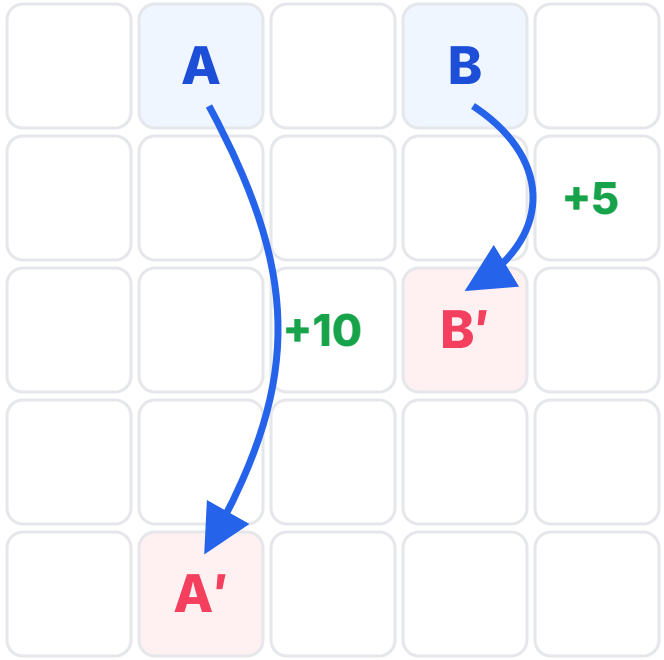
- The rules of the game are **not given** to the agent.
- It learns directly from playing: it picks joystick actions and sees pixels and scores.



IN PLAIN WORDS

Atari games became a classic test bed for exactly this setting.

Gridworld: prediction



From A, every action gives +10 and jumps to A′. From B, every action gives +5 and jumps to B′.

3.3	8.8	4.4	5.3	1.5
1.5	3.0	2.3	1.9	0.5
0.1	0.7	0.7	0.4	−0.4
−1.0	−0.4	−0.4	−0.6	−1.2
−1.9	−1.3	−1.2	−1.4	−2.0

v_π for the uniform random policy, $\gamma = 0.9$

- Moving off the grid gives −1 and the agent stays put. All other moves give 0.
- What is the value of each square if the agent moves at random?



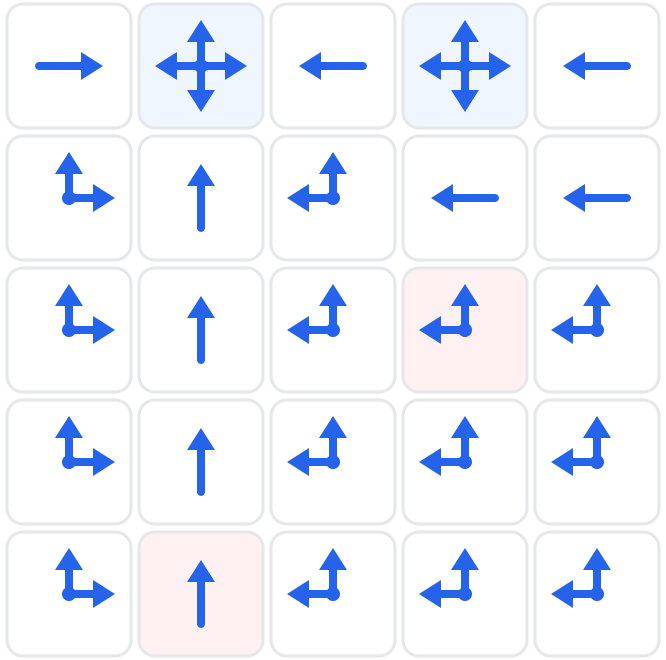
IN PLAIN WORDS

A is worth less than 10: after the jump, A′ sits on the edge, where random moves often hit the wall. B is worth more than 5 because B′ is a safer spot.

Gridworld from Sutton & Barto (2018), chapter 3. Values computed by us.

Gridworld: control

22.0	24.4	22.0	19.4	17.5
19.8	22.0	19.8	17.8	16.0
17.8	19.8	17.8	16.0	14.4
16.0	17.8	16.0	14.4	13.0
14.4	16.0	14.4	13.0	11.7



v_*: the best possible values

π_*: an optimal policy (all best moves)

Gridworld from Sutton & Barto (2018), chapter 3. Values computed by us.

- What is the optimal value function, over all possible policies?
- What is the optimal policy?



IN PLAIN WORDS

Almost every square heads for A, to collect the +10 again and again. From the top-right corner it pays to grab B's +5 first.

WRAP-UP

Summary

- RL is learning to make decisions from **interaction**, guided by a **reward**.
- The agent maximises the **return** G_t, the (discounted) sum of future rewards.
- The **value** $v_\pi(s)$ is the expected return; $q_\pi(s, a)$ also fixes the first action.
- An agent may have an **agent state**, a **policy**, a **value function** and a **model**.

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

$$v_\pi(s) = \mathbb{E}\left[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s\right]$$

$$v_*(s) = \max_a \mathbb{E}\left[R_{t+1} + \gamma v_*(S_{t+1}) \mid S_t = s, A_t = a\right]$$

Agent
the learner and decision maker

Environment
everything the agent interacts with

Reward
one number per step that defines the goal

Return
the (discounted) sum of future rewards

Value
the expected return from a state

Policy
what the agent does in each state

Model
the agent's prediction of what happens next

Markov
the state is enough; the past adds nothing

WRAP-UP

In this course

- We study how to learn by interaction.
- The focus is on **core principles** and **learning algorithms**.



IN PLAIN WORDS

Companion videos: **ACML Explains**, season 1. Each lecture has a few short episodes.

1 · Ep 1–5 · **Introduction**	2 · Ep 6–9 · **Exploration: bandits**	3 · Ep 10–14 · **MDPs and dynamic programming**
4 · Ep 15–16 · **Theory of dynamic programming**	5 · Ep 17–20 · **Model-free prediction**	6 · Ep 21–23 · **Model-free control**
7 · Ep 24–26 · **Function approximation**	8 · Ep 27–30 · **Deep RL**	9 · Ep 31–37 · **Planning, models and search**
10 · Ep 38–42 · **Policy gradients**	11 · Ep 43–46 · **Learning about many things**	12 · Ep 47–51 · **Learning from humans**

ANSWERS

Answers to check yourself

- **1.** RL is active (its actions choose its data), sequential (actions affect what comes next), and learns from rewards rather than labelled answers.
- **2.** Any goal can be expressed as making the expected sum of a reward signal as large as possible.
- **3.** Reason 2: adapting online, during operation.
- **4.** Many answers. Cycling to work: actions are the route and the speed; observations are traffic and weather; the reward could be minus the minutes travelled.
- **5.** $G_0 = 1 + 0.5 + 0.25 = 1.75$.
- **6.** Under partial observability, different situations can look identical (the two pink squares in the maze), so the observation is not Markov.
- **7.** v_π is the expected return when following a given policy π; v_* is the highest value achievable by any policy.
- **8.** No. A model predicts what happens next; the agent still has to plan with it to choose actions.
- **9.** $q(s, \text{south}) = -1 + (-6) = -7$.

CREDITS

Sources and credits

- The order of topics follows *Lecture 1: Introduction* of the DeepMind × UCL Reinforcement Learning Lecture Series (2021), by Hado van Hasselt. This deck is not an official DeepMind or UCL material.
- The text, figures, maze, thermometer data (simulated) and worked examples were written and drawn for ACML. The gridworld example is from Sutton & Barto (2018), chapter 3; its values were recomputed by us.
- Sutton, R. S. & Barto, A. G. (2018). *Reinforcement Learning: An Introduction* (2nd ed.). MIT Press. Read chapters 1 and 3 next.
- Turing, A. M. (1950). Computing Machinery and Intelligence. *Mind*, 59(236), 433–460.
- Bellman, R. (1957). *Dynamic Programming*. Princeton University Press.
- Companion videos: **ACML Explains**, season 1, episodes 1–5. More from the lab: acml.teddylazebnik.com





Applied Computational Mathematics Lab

END OF LECTURE 1

Next: explore or exploit?

Lecture 2 looks at the simplest RL problem, the multi-armed bandit, and how to balance trying new things with using what works.

