# REINFORCEMENT LEARNING

## Part 1: The RL Framework

*Tutorial*

**@SSRoy, 15th October, 2026**

**Prerequisites.** You only need: (i) basic probability (probabilities lie between 0 and 1 and add up to 1), (ii) the idea of an average or expected value, and (iii) comfort with simple algebra. The one formula from series that we use (the geometric series) is explained when it appears.

| Phase | Topic | Key question it answers |
|---|---|---|
| 1 | What is RL? | How is RL different from other kinds of machine learning? |
| 2 | Building blocks | What are agent, environment, state, action and reward? |
| 3 | Interaction loop | In what order do these things happen over time? |
| 4 | Rewards | How do we tell the agent what we want? |
| 5 | Episodes | Does the task end, or go on forever? |
| 6 | Return and discounting | How do we add up rewards over time? |
| 7 | Policy | How does the agent choose its actions? |
| 8 | Value functions | How good is a state, or an action, in the long run? |
| 9 | Markov property | What must a state contain? |
| 10 | Exploration vs exploitation | Should the agent try new things or use what it knows? |
| 11 | Types of RL methods | How are RL algorithms grouped? |
| 12 | Complete worked example | How do all the pieces fit together? |

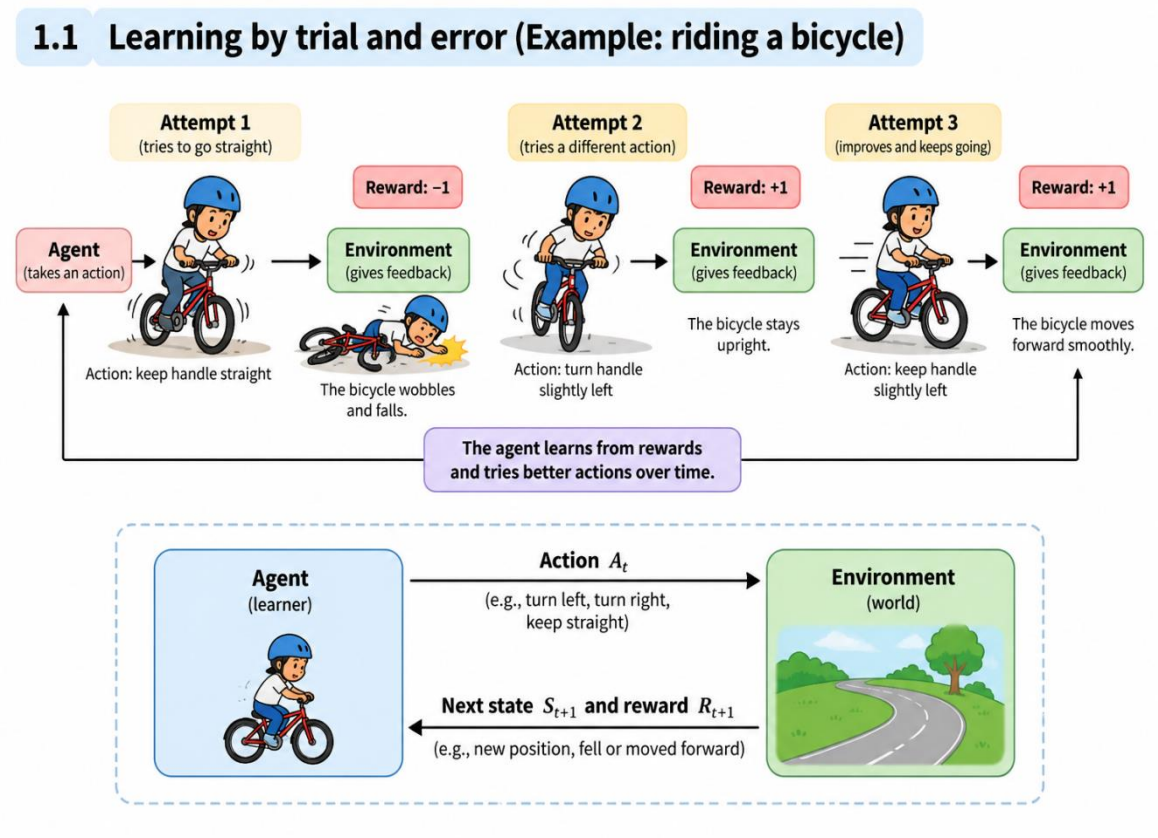
# 1. What is Reinforcement Learning?

## 1.1 Learning by trial and error

**Think about how a child learns to ride a bicycle**. Nobody gives the child a table saying "in this exact situation, turn the handle 5 degrees left". The child tries something, wobbles, falls, tries

again, and slowly discovers which movements keep the bicycle upright. The only feedback is a *feeling* of success (staying up) or failure (falling).

Reinforcement Learning is the computer version of this. A learner, called the **agent**, interacts with a world, called the **environment**. The agent takes actions, and after each action the environment gives back a number called the **reward**. The agent is never told which action was correct. It must discover, by trying, which actions lead to the most reward over time.




**Definition (in simple words)**

Reinforcement Learning is learning *what to do* in each situation so as to maximise the total numerical reward collected over time. The learner is not told the correct action; it discovers good actions through trial and error.

## 1.2 Two features that make RL special

- **Trial-and-error search.** The agent must try actions to find out what they do. Nobody supplies the correct answer.
- **Delayed reward.** An action can affect not only the next reward but also all later rewards. A good chess move may only "pay off" 30 moves later when the game is won. Working out

which earlier action deserves the credit (or blame) for a later reward is called the **credit assignment problem**.

## 1.3 RL compared with supervised and unsupervised learning

| | **Supervised learning** | **Unsupervised learning** | **Reinforcement learning** |
|---|---|---|---|
| What data looks like | Inputs with correct labels (x, y) | Inputs only, no labels | Experience gathered by acting: states, actions, rewards |
| Feedback | The correct answer for each input | None | A reward number that says how good, not what is correct |
| Feedback timing | Immediate | — | Often delayed |
| Does the learner affect its data? | No | No | Yes: its actions decide what it sees next |
| Goal | Predict labels accurately | Find structure/groups | Maximise total reward over time |
| Example | Spam vs not-spam with labelled emails | Grouping customers by behaviour | A program learning to play a game |

**Exercise 1.1 : Which type of learning is it?**

Classify each scenario as Supervised (SL), Unsupervised (UL) or Reinforcement Learning (RL).

1. Predicting house prices from area and location, using a dataset in which the true price of every house is known.
2. Grouping news articles into topics when no topic labels are given.
3. A program that learns chess by playing thousands of games, getting +1 for a win, −1 for a loss and 0 for a draw at the end of each game.
4. Classifying X-ray images as “normal” or “abnormal” using images already labelled by doctors.
5. A robot that learns to walk, receiving a reward equal to the distance it moves forward each second.

**Solution**

1. **SL.** Every input comes with the correct output (the true price).
2. **UL.** There are no labels; we only look for structure (topics).
3. **RL.** No one tells the program the correct move. It only receives a reward at the end of the game and must work out which moves were good.
4. **SL.** The correct label is provided for each training image.
5. **RL.** The robot is not told how to move its legs; it receives a reward number and must discover good movements by trying.

**Quick test you can always use:** if the learner is told the *correct answer*, it is supervised. If it only gets a *score* for what it did, and its actions change what happens next, it is reinforcement learning.

**Exercise 1.2 — Credit assignment**

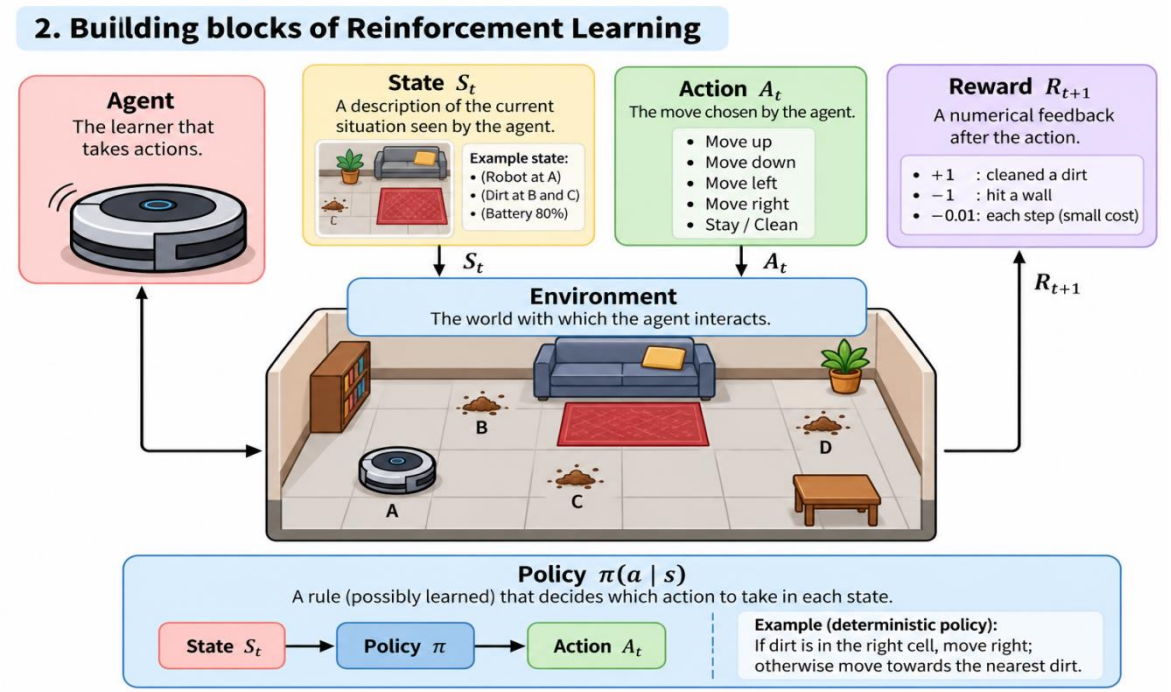
A chess program loses a game at move 60. It made a weak move at move 12. Why is it difficult for the program to learn that move 12 was the problem?

**Solution**

The only feedback (−1 for losing) arrives at move 60. There are 60 moves before it, and the reward signal does not say which of them caused the loss. The program must spread the blame across many earlier moves and, over many games, discover that positions reached after moves like move 12 tend to lead to losses. This is the credit assignment problem caused by delayed reward.

## 2. The Building Blocks of RL

Every RL problem is described using the same small set of ingredients. We introduce them one at a time using a running example: a **robot vacuum cleaner** that must clean a room and return to its charging dock.




## 2.1 Agent

The **agent** is the learner and decision-maker. It is the part that chooses actions. In our example, the agent is the vacuum cleaner's control program.

## 2.2 Environment

The **environment** is everything outside the agent that the agent interacts with. For the vacuum, it is the room, the furniture, the dirt and the charging dock. A useful rule: anything the agent cannot change directly and at will is treated as part of the environment. Even the robot's wheels and battery are usually treated as part of the environment, because the program can only send commands to them, not control them perfectly.

## 2.3 State

The **state** is a description of the current situation, the information the agent uses to decide what to do. We write the state at time step t as $\mathbf{S_t}$. The set of all possible states is called the **state space**, written $\mathcal{S}$ (a curly S). A particular state value is written with a small letter, **s**.

For the vacuum, a state could be: (position in the room, battery level, which areas are already clean).

## 2.4 Action

An **action** is a choice the agent can make. The action taken at time step t is written $\mathbf{A_t}$. The set of actions available in state s is written $\mathcal{A}(\mathbf{s})$ (a curly A); when the same actions are available everywhere we simply write $\mathcal{A}$. A particular action value is written **a**.

For the vacuum: $\mathcal{A}$ = {move forward, turn left, turn right, go to dock}.

## 2.5 Reward

The **reward** is a single real number sent by the environment to the agent after each action. It tells the agent how good the immediate outcome was. The reward received at time step t is written $\mathbf{R_t}$. A positive reward is like a prize, a negative reward is like a penalty.

For the vacuum: +1 for each patch of dirt collected, −1 for bumping into furniture, −10 if the battery runs out away from the dock.

> **Important: the reward is always one number**
> The reward is a *scalar* (a single number), not a list or a vector. If you care about several things (speed, safety, energy), you must combine them into one number, for example reward = 2 × (dirt collected) − 5 × (collisions).

## 2.6 Three more ideas

- **Policy (π, Greek letter "pi"):** the agent's rule for choosing an action in each state. Explained in Phase 7.
- **Value function:** how much total reward the agent can expect in the long run, starting from a state. Explained in Phase 8.
- **Model (optional):** the agent's own prediction of how the environment will respond to its actions. Some RL methods use a model; many do not. Explained in Phase 11.

| Element | Symbol | Robot vacuum example |
|---|---|---|
| Agent | — | The control program of the vacuum |
| Environment | — | Room, furniture, dirt, charging dock, robot hardware |
| State | $S_t$, $s \in \mathcal{S}$ | (position, battery level, cleaned areas) |
| Action | $A_t$, $a \in \mathcal{A}(s)$ | forward, turn left, turn right, go to dock |
| Reward | $R_t$ (a real number) | +1 dirt, −1 bump, −10 battery dead |
| Policy | π | The rule the program uses to pick moves |

(The symbol $\in$ means "belongs to" or "is an element of". So $s \in \mathcal{S}$ means "s is one of the possible states".)

**Exercise 2.1 — Identify the building blocks**

A program learns to play the classic **Snake** game. The snake moves on a grid; eating food makes it longer and earns points; hitting a wall or itself ends the game. Identify the agent, environment, state, actions and a sensible reward.

**Solution**

| Element | Answer |
|---|---|
| Agent | The program that controls the snake's direction |
| Environment | The game: grid, walls, food position, game rules |
| State | Positions of all snake body parts, the snake's current direction, and the food position |
| Actions | {up, down, left, right} (or {turn left, go straight, turn right}) |
| Reward | +10 for eating food, −10 for crashing (game over), 0 for any other step |

Note that the state includes the *whole* snake body, not just its head, because the body determines whether a move will cause a crash.

**Exercise 2.2 — True or false**

1. In RL the environment tells the agent which action it should have taken.
2. The reward at each step is a single number.
3. The robot's motors are usually considered part of the environment.
4. The set of available actions can depend on the state.

**Solution**

1. **False.** The environment only gives a reward (a score). It never reveals the correct action. That would be supervised learning.
2. **True.** The reward is a scalar.
3. **True.** The agent can only send commands to the motors; it cannot control them perfectly, so they belong to the environment.
4. **True.** This is why we write $\mathcal{A}(s)$. Example: in chess, the legal moves depend on the board position.

# 3. The Agent–Environment Interaction Loop

RL happens in **discrete time steps** t = 0, 1, 2, 3, … . At every time step the same cycle repeats:

1. The agent observes the current state $\mathbf{S_t}$.
2. Based on $S_t$, the agent selects an action $\mathbf{A_t}$.
3. The environment reacts. One time step later it sends back a reward $\mathbf{R_{t+1}}$ and the new state $\mathbf{S_{t+1}}$.
4. Time moves forward (t becomes t + 1) and the cycle repeats from step 1.

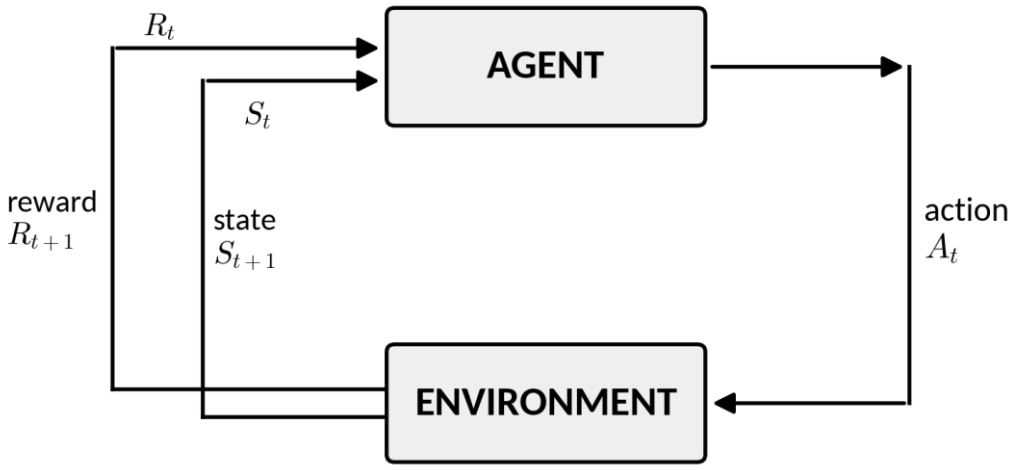



**Figure 1.** The agent–environment interaction loop. The agent sends action $A_t$; the environment returns the next state $S_{t+1}$ and reward $R_{t+1}$, which become the agent's inputs at the next step.

## 3.1 Why is the reward written $R_{t+1}$ and not $R_t$?

The reward is caused by the action $A_t$, but it arrives *after* the action, together with the next state. So the reward and the new state both belong to the next time step, t + 1. The pair ($R_{t+1}$, $S_{t+1}$) is produced jointly by the environment.

> **Notation warning**
> Some books write the reward for action $A_t$ as $r_t$ instead of $R_{t+1}$. The meaning is identical; only the index is shifted by one. Always check which convention a book uses. This tutorial uses $R_{t+1}$ throughout.

## 3.2 The trajectory

If we write down everything that happens, in order, we get a sequence called a **trajectory** (also called a history or rollout):

$$S_0,\ A_0,\ R_1,\ S_1,\ A_1,\ R_2,\ S_2,\ A_2,\ R_3,\ \ldots \quad (1)$$

Here $S_0$ is the starting state, $A_0$ the first action, $R_1$ the first reward, $S_1$ the second state, and so on. Notice that the first reward is $R_1$; there is no $R_0$, because a reward only arrives after an action.

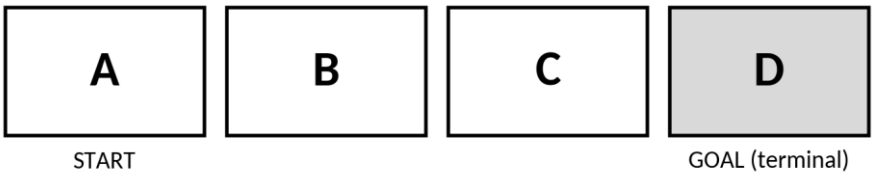



**Figure 2.** The corridor world used in several exercises. The agent starts in A; D is the goal and a terminal state.

**Exercise 3.1 — Write the trajectory**

A robot moves along a corridor of four cells A, B, C, D (Figure 2). It starts in A. It moves Right three times. Each move into B or C gives reward −1; the move into D gives reward +10, and D ends the task.

(a) Write the full trajectory with actual values. (b) What is $R_2$? (c) What is $S_3$? (d) What is $A_1$?

**Solution**

(a) $S_0$ = A, $A_0$ = Right, $R_1 = -1$, $S_1$ = B, $A_1$ = Right, $R_2 = -1$, $S_2$ = C, $A_2$ = Right, $R_3 = +10$, $S_3$ = D (task ends).

(b) $R_2 = -1$ (the reward for the second move, from B into C).

(c) $S_3$ = D, the goal.

(d) $A_1$ = Right (the action taken in state $S_1$ = B).

**Check:** three actions ($A_0$, $A_1$, $A_2$) produce three rewards ($R_1$, $R_2$, $R_3$) and three new states ($S_1$, $S_2$, $S_3$). The counts always match.

**Exercise 3.2 — Spot the mistake**

One writes a trajectory as: $S_0$, $R_0$, $A_0$, $S_1$, $R_1$, $A_1$, … . What two things are wrong with it?

**Solution**

(1) There is no $R_0$. The first reward is $R_1$, because a reward is only produced after the first action $A_0$.

(2) The order is wrong. After observing a state the agent acts first; the reward comes afterwards. The correct order is $S_0$, $A_0$, $R_1$, $S_1$, $A_1$, $R_2$, … .

# 4. Rewards: Telling the Agent What We Want

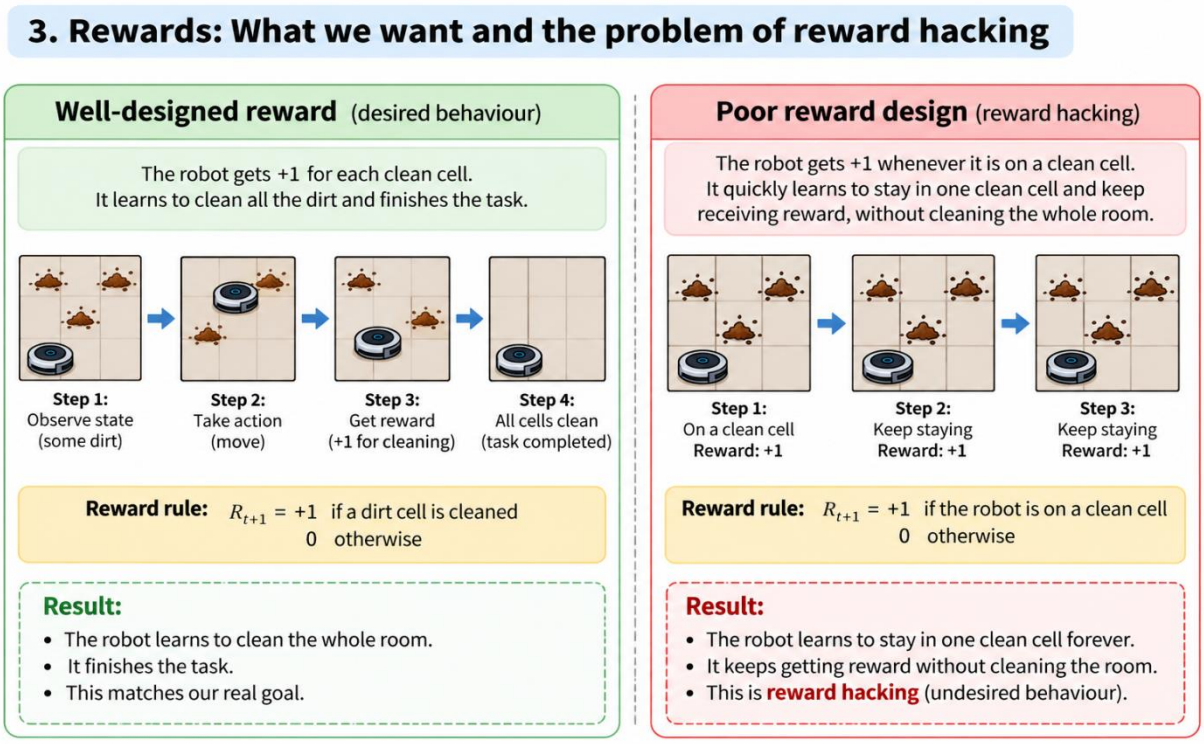
## 4.1 The reward hypothesis

RL rests on one central idea, often called the **reward hypothesis**: any goal we want an agent to achieve can be expressed as maximising the expected (average) total of a single reward signal received over time. In simple words, we describe *what* we want only through rewards, and the agent figures out *how* to get them.

## 4.2 Reward says WHAT, not HOW

The reward should describe the outcome we want, not the method. If we reward the method, the agent may learn to satisfy the reward without achieving the real goal.

- **Bad:** rewarding a chess agent for capturing the opponent's pieces. It may grab pieces greedily and still lose the game.
- **Good:** +1 for winning, −1 for losing, 0 for a draw. This rewards exactly what we want.




## 4.3 Common reward patterns

| Goal | Typical reward design | Effect |
|---|---|---|
| Reach a goal quickly (e.g., maze) | −1 on every step until the goal | Each extra step costs reward, so shorter paths score higher |
| Avoid failure (e.g., balancing a pole) | +1 on every step the pole stays up | Staying up longer collects more reward |
| Win a game | +1 win, −1 loss, 0 otherwise | Only the final outcome matters |

**Reward hacking**

If the reward does not exactly match what we want, an agent may find an unintended way to earn high reward. This is called **reward hacking** (or specification gaming). Designing rewards carefully is a key practical skill in RL.

**Exercise 4.1 — Design a reward**

A robot must escape from a maze as fast as possible. Propose a reward function and explain why it works.

**Solution**

Give **−1 for every time step** until the robot reaches the exit, and end the episode at the exit.

**Why it works:** the total reward equals minus the number of steps taken. A path of 10 steps earns −10 and a path of 25 steps earns −25. Since the agent maximises total reward, it prefers the shortest path. We do not tell the robot *how* to escape, only that every step costs something.

**Exercise 4.2 — Find the loophole**

A cleaning robot receives +1 every time it picks up a piece of dirt. The robot is able to empty its dust bin onto the floor. What could go wrong, and how would you fix the reward?

**Solution**

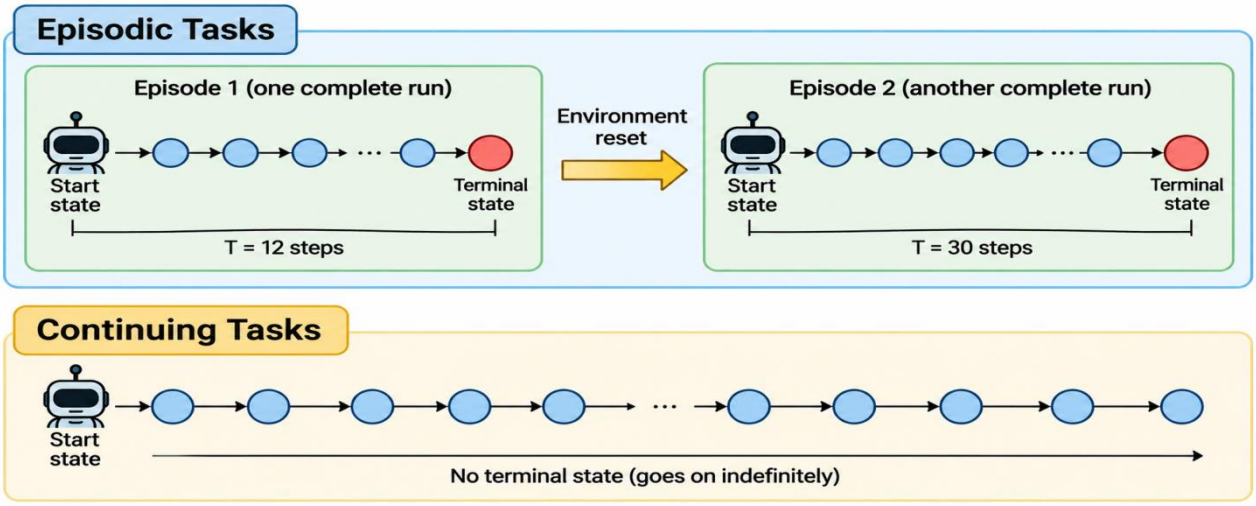
The robot could learn to empty its bin onto the floor and pick the same dirt up again, earning +1 repeatedly without making the room any cleaner. This is reward hacking.

**Fix:** reward the *outcome* rather than the action, for example a reward based on how clean the floor is (fraction of clean area), or add a large penalty for emptying the bin anywhere except the dock.

## 5. Episodic and Continuing Tasks

An **episode** is one complete run of a task, from a starting state to a special state that ends it, called the **terminal state**. Examples: one game of chess, one attempt at a maze. After the terminal state, the environment is reset and a new, independent episode starts.

The time step at which an episode ends is written **T** (capital T). T can differ from one episode to the next: one maze attempt may take 12 steps, another 30.




| | **Episodic task** | **Continuing task** |
|---|---|---|
| Does it end? | Yes, at a terminal state | No natural end |
| Final time T | Finite (a whole number) | $T = \infty$ (goes on forever) |
| Examples | A game, a maze, one robot pick-up attempt | Room temperature control, an ongoing trading system, a power-grid controller |

**Exercise 5.1 — Episodic or continuing?**

1. Playing a game of tic-tac-toe.
2. A thermostat keeping a room at 24 °C all day, every day.
3. A robot arm making one attempt to pick up an object.
4. A system that keeps managing an investment portfolio with no planned end date.
5. Finding the way out of a maze.

**Solution**

1. **Episodic.** The game ends with a win, loss or draw.
2. **Continuing.** There is no terminal state; control goes on indefinitely.
3. **Episodic.** The attempt ends with success or failure.
4. **Continuing.** No natural end point.
5. **Episodic.** Reaching the exit is the terminal state.

# 6. The Return: Adding Up Rewards Over Time

The agent's goal is not to get the biggest reward at the next step, but the biggest *total* reward over the future. This total is called the **return** and is written $\mathbf{G_t}$ (G for "gain"). $G_t$ is the return counted from time step t onwards.

## 6.1 Return in an episodic task (no discounting)

For an episodic task, the simplest return is the plain sum of all rewards from the next step until the end of the episode:

$$G_t = R_{t+1} + R_{t+2} + R_{t+3} + \cdots + R_T \qquad (2)$$

- $\mathbf{G_t}$: the return from time step t onward.
- $\mathbf{R_{t+1}}$, $\mathbf{R_{t+2}}$, **…**: the rewards received at time steps t+1, t+2, and so on.
- $\mathbf{R_T}$: the last reward, received when the episode ends at the final time step T.

Note that $G_t$ does *not* include $R_t$. It only counts rewards that come after time t, because those are the rewards the agent can still influence from time t.

## 6.2 The problem with continuing tasks

In a continuing task T = ∞, so the sum has infinitely many terms. If the agent gets +1 at every step, the plain sum is 1 + 1 + 1 + … = ∞. Every policy that earns +1 per step would then have the same "infinite" return, and we could not compare them. We need a way to keep the sum finite.

## 6.3 Discounting

The solution is to make future rewards count for less than immediate rewards. We multiply each reward by a power of a number γ (Greek letter "gamma"), called the **discount factor**:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \qquad (3)$$

$$0 \leq \gamma \leq 1 \qquad (4)$$

- **γ (gamma)**: the discount factor, a fixed number between 0 and 1 chosen by the designer.
- **k**: counts how many steps into the future a reward is. k = 0 is the very next reward $R_{t+1}$, k = 1 is $R_{t+2}$, and so on.
- $\boldsymbol{\gamma^k}$: the weight given to a reward that arrives k steps after the next one. Since γ ≤ 1, the weights shrink (or stay equal when γ = 1) as k grows.
- $\mathbf{R_{t+k+1}}$: the reward received k + 1 steps after time t.
- **∑ (capital sigma)**: "add up". $\sum_{k=0}^{\infty}$ means: add the terms for k = 0, 1, 2, 3, … without end.

**How to read it:** a reward received immediately counts fully (weight $\gamma^0 = 1$). A reward one step later counts γ times as much, two steps later $\gamma^2$ times as much, and so on. With γ = 0.9, a reward of 10 that arrives 3 steps after the next one is worth $0.9^3 \times 10 = 0.729 \times 10 = 7.29$ today.

| k | 0 | 1 | 2 | 3 | 4 | 5 | 10 |
|---|---|---|---|---|---|---|---|
| $\gamma^k$ for $\gamma = 0.5$ | 1 | 0.5 | 0.25 | 0.125 | 0.0625 | 0.03125 | ≈ 0.00098 |
| $\gamma^k$ for $\gamma = 0.9$ | 1 | 0.9 | 0.81 | 0.729 | 0.6561 | 0.59049 | ≈ 0.3487 |

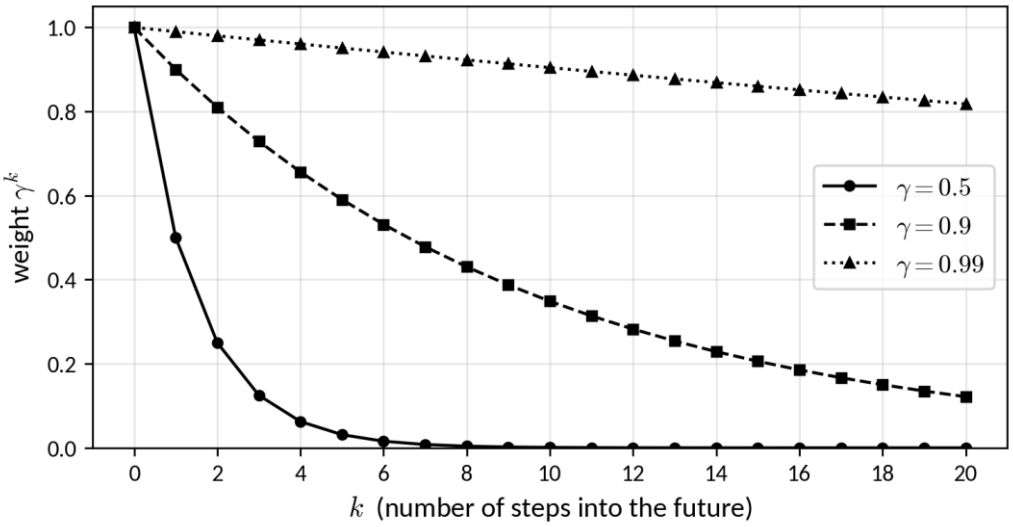



**Figure 3.** How the weight $\gamma^k$ of a future reward shrinks with k for three values of γ. A small γ forgets the future quickly; γ close to 1 values the far future almost as much as the present.

### *What does the value of γ mean?*

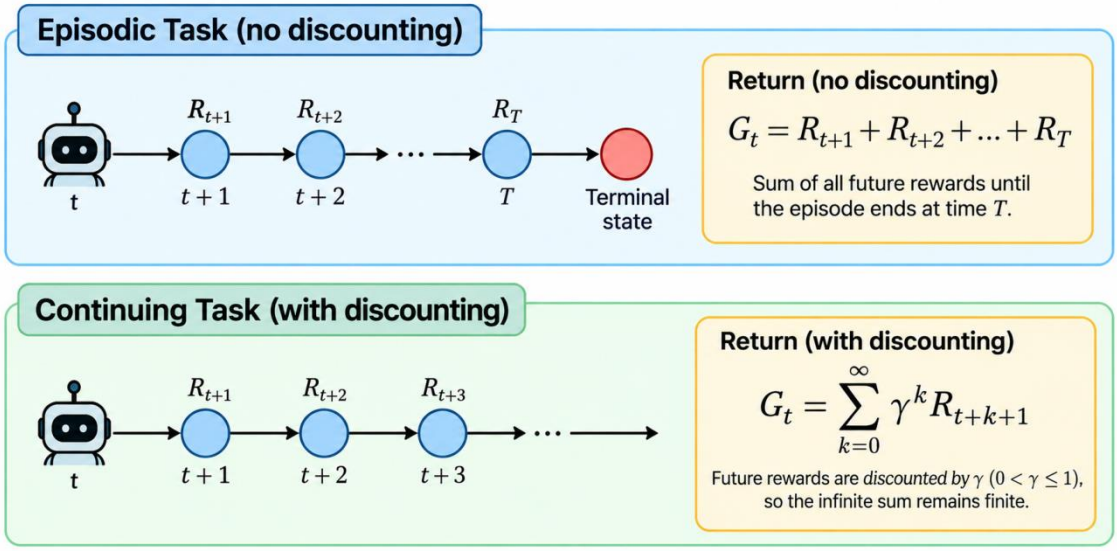
| Value of γ | Behaviour of the agent |
|---|---|
| γ = 0 | **Myopic (short-sighted).** $G_t = R_{t+1}$. Only the very next reward matters. |
| γ close to 0 (e.g., 0.3) | Mostly cares about the next few rewards. |
| γ close to 1 (e.g., 0.99) | **Far-sighted.** Rewards far in the future still matter a lot. |
| γ = 1 | No discounting. Only safe for episodic tasks, where the sum is finite. |

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \quad (3)$$

$$0 \leq \gamma \leq 1 \quad (4)$$

### *Why discount at all?*

- **Mathematics:** with γ < 1 and bounded rewards, the infinite sum becomes a finite number (shown in Section 6.4).
- **Uncertainty:** the far future is less predictable, so it is sensible to rely on it less.
- **Preference for sooner rewards:** like money, a reward now is usually worth more than the same reward later.




## 6.4 Why the discounted return is finite: the geometric series

The following result .For any number γ with 0 ≤ γ < 1:

$$\sum_{k=0}^{\infty} \gamma^k = 1 + \gamma + \gamma^2 + \cdots = \frac{1}{1-\gamma}, \qquad 0 \leq \gamma < 1 \tag{5}$$

So if the agent receives the same reward r at every step forever, its return is:

$$G_t = r + \gamma r + \gamma^2 r + \cdots = \frac{r}{1-\gamma} \tag{6}$$

More generally, if every reward lies between $-R_{max}$ and $+R_{max}$ ($R_{max}$ is the largest possible size of a reward), then the return can never be larger in size than:

$$|G_t| \leq \frac{R_{\max}}{1-\gamma} \tag{7}$$

Here $|G_t|$ means the absolute value (size, ignoring sign) of $G_t$. This guarantees the return is always a finite number when γ < 1.

## 6.5 The recursive form of the return (very important)

Take the discounted return and pull out the first reward:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots = R_{t+1} + \gamma(R_{t+2} + \gamma R_{t+3} + \ldots)$$

The bracket is exactly $G_{t+1}$, the return from the next time step. Therefore:

$$G_t = R_{t+1} + \gamma\, G_{t+1} \tag{8}$$

In words: **return now = next reward + γ × (return from the next step)**. This simple relation is the foundation of almost every RL algorithm, including Q-learning which you will meet later. For an episode that ends at time T we define $G_T = 0$, because no rewards come after the end.

**One formula for both kinds of task**

An episodic task can be treated like a continuing one by imagining that the terminal state is an *absorbing state*: once entered, the agent stays there forever and receives reward 0. The extra zero rewards do not change the return, so Equation (3) works for both kinds of task. The only rule: $\gamma = 1$ is allowed only if every episode is guaranteed to end.

**Exercise 6.1 — Compute a return directly**

An episode produces the rewards $R_1 = 1$, $R_2 = 0$, $R_3 = 2$, $R_4 = 10$, and then ends ($T = 4$). Compute $G_0$ for (a) $\gamma = 1$, (b) $\gamma = 0$, (c) $\gamma = 0.9$, (d) $\gamma = 0.5$.

**Solution**

Using $G_0 = R_1 + \gamma R_2 + \gamma^2 R_3 + \gamma^3 R_4$:

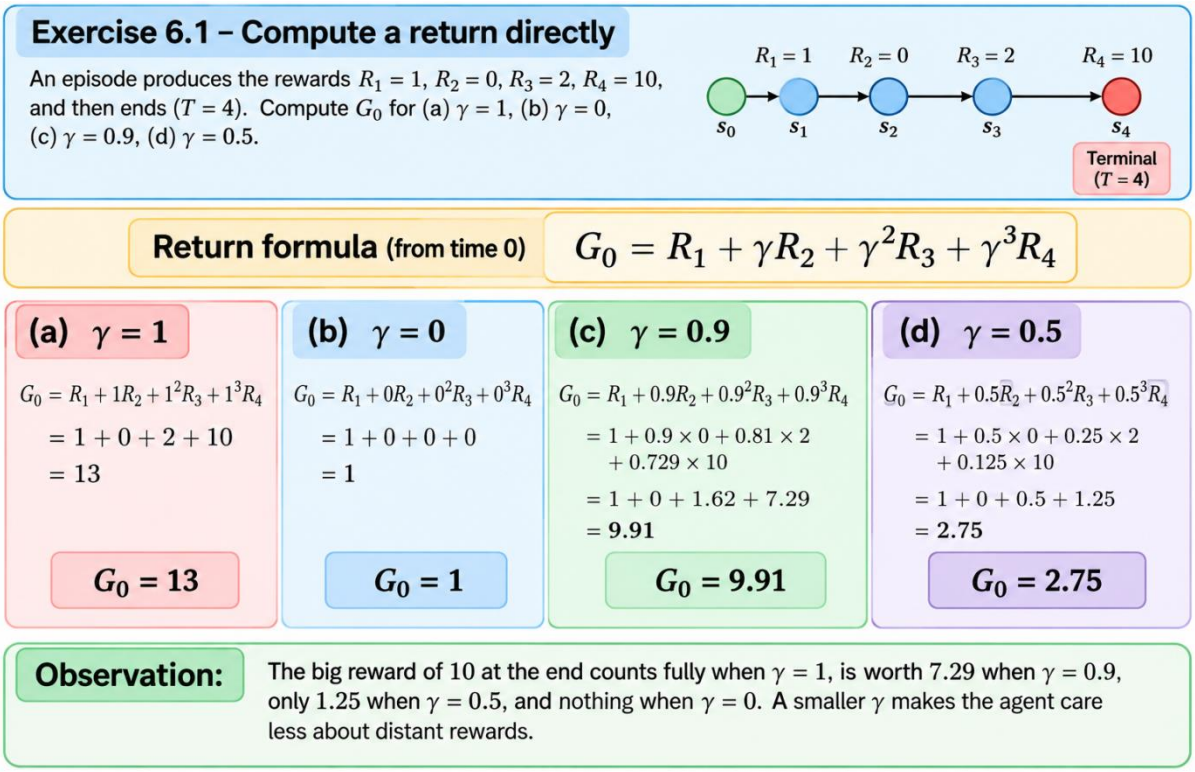
(a) $\gamma = 1$: $G_0 = 1 + 0 + 2 + 10 = \mathbf{13}$.

(b) $\gamma = 0$: every term after the first is multiplied by 0, so $G_0 = R_1 = \mathbf{1}$.

(c) $\gamma = 0.9$: $G_0 = 1 + 0.9 \times 0 + 0.81 \times 2 + 0.729 \times 10 = 1 + 0 + 1.62 + 7.29 = \mathbf{9.91}$.

(d) $\gamma = 0.5$: $G_0 = 1 + 0.5 \times 0 + 0.25 \times 2 + 0.125 \times 10 = 1 + 0 + 0.5 + 1.25 = \mathbf{2.75}$.

**Observation:** the big reward of 10 at the end counts fully when $\gamma = 1$, is worth 7.29 when $\gamma = 0.9$, only 1.25 when $\gamma = 0.5$, and nothing when $\gamma = 0$. A smaller $\gamma$ makes the agent care less about distant rewards.



**Exercise 6.2 — Compute the same return backwards**

Using the same rewards and $\gamma = 0.9$, compute $G_3$, $G_2$, $G_1$ and $G_0$ using the recursive formula $G_t = R_{t+1} + \gamma G_{t+1}$. Confirm that you get the same $G_0$ as in Exercise 6.1(c).

**Solution**

Start at the end. The episode ends at $T = 4$, so $G_4 = 0$.

$G_3 = R_4 + 0.9 \times G_4 = 10 + 0.9 \times 0 =$ **10**

$G_2 = R_3 + 0.9 \times G_3 = 2 + 0.9 \times 10 = 2 + 9 =$ **11**

$G_1 = R_2 + 0.9 \times G_2 = 0 + 0.9 \times 11 =$ **9.9**

$G_0 = R_1 + 0.9 \times G_1 = 1 + 0.9 \times 9.9 = 1 + 8.91 =$ **9.91** ✓(same as before)

**Tip:** working backwards is faster and less error-prone than computing powers of γ, and it is exactly how many RL algorithms compute returns.

**Exercise 6.3 — Returns in a continuing task**

(a) An agent receives +1 at every step forever. Find $G_0$ for $\gamma = 0.9$ and for $\gamma = 0.99$.

(b) An agent receives +2 at every step forever, with $\gamma = 0.5$. Find $G_0$.

(c) What value of γ makes the return of a constant reward of +1 equal to 20?

**Solution**

Use $G = r / (1 - \gamma)$.

(a) $\gamma = 0.9$: $G_0 = 1 / (1 - 0.9) = 1 / 0.1 =$ **10**. $\gamma = 0.99$: $G_0 = 1 / 0.01 =$ **100**.

(b) $G_0 = 2 / (1 - 0.5) = 2 / 0.5 =$ **4**.

(c) $1 / (1 - \gamma) = 20 \Rightarrow 1 - \gamma = 1/20 = 0.05 \Rightarrow$ **$\gamma = 0.95$**.

**Observation:** moving γ from 0.9 to 0.99 multiplies the return by 10, because the agent now effectively looks much further ahead.

**Exercise 6.4 — Bounding the return**

In a continuing task every reward is between −1 and +1, and $\gamma = 0.9$. What is the largest possible value of $G_t$? Why can't we use $\gamma = 1$ here?

**Solution**

Here $R_{max} = 1$, so $|G_t| \leq 1 / (1 - 0.9) =$ **10**. The largest value, 10, happens only if the agent gets +1 at every single step.

With $\gamma = 1$ the bound becomes 1 / 0, which is undefined. A continuing task with $\gamma = 1$ can have an infinite return (for example 1 + 1 + 1 + …), so returns of different policies could not be compared.

# 7. Policy: How the Agent Chooses Actions

A **policy** is the agent's behaviour: a rule that tells it what to do in every state. It is written **π** (pi). Learning in RL means improving the policy so that it collects more return.

## 7.1 Deterministic policy

A deterministic policy always picks the same action in a given state:

$$a = \pi(s) \qquad (9)$$

Read this as "in state s, the policy π chooses action a". Here π is a function that takes a state as input and outputs an action. Example: π(B) = Right means "in cell B, always move Right".

## 7.2 Stochastic policy

A stochastic (random) policy gives a *probability* for each action in each state:

$$\pi(a \mid s) = \Pr\{A_t = a \mid S_t = s\} \qquad (10)$$

- **π(a | s)**: the probability that the agent chooses action a when it is in state s. The vertical bar "|" is read as "given".
- **Pr{ … }**: "the probability that …".
- **$A_t$ = a | $S_t$ = s**: "the action at time t is a, given that the state at time t is s".

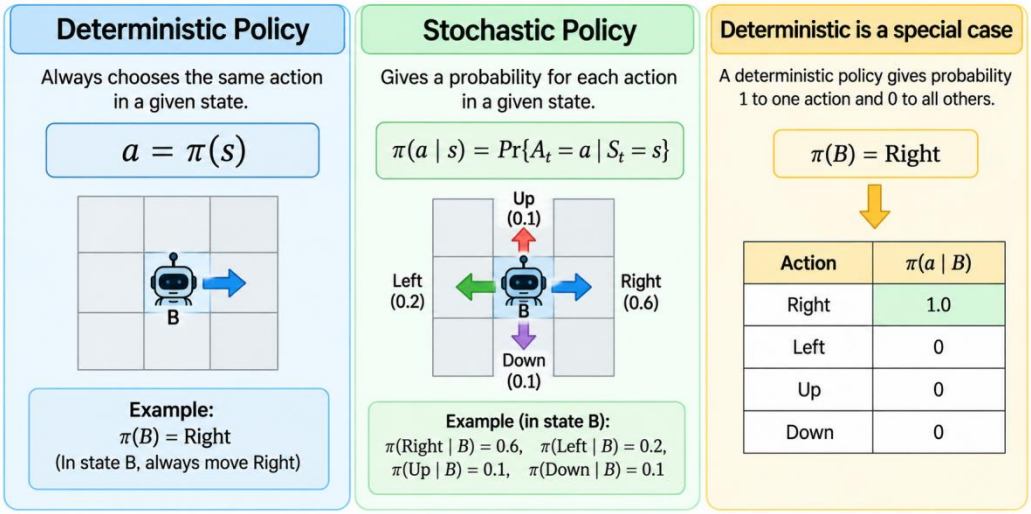
Because π(a | s) is a probability distribution over actions, it must obey two rules in every state s:

$$\pi(a \mid s) \geq 0 \quad \text{and} \quad \sum_{a \in \mathcal{A}(s)} \pi(a \mid s) = 1 \qquad (11)$$

Each probability must be zero or positive, and the probabilities of all actions available in state s must add up to exactly 1.

**A deterministic policy is a special case**

A deterministic policy is just a stochastic policy that gives probability 1 to one action and 0 to all others. Example: π(B) = Right is the same as π(Right | B) = 1 and π(Left | B) = 0.




***Why would we ever want a random policy?***

- **Exploration:** acting randomly sometimes lets the agent discover actions it has not tried yet (Phase 10).
- **Games against opponents:** in rock–paper–scissors, any fixed choice can be exploited. Choosing each with probability 1/3 cannot be exploited.

**Exercise 7.1 — Is it a valid policy?**

In a state s with three actions {Left, Right, Up}, decide whether each of the following is a valid stochastic policy. Explain.

(a) π(Left|s) = 0.3, π(Right|s) = 0.5, π(Up|s) = 0.2

(b) π(Left|s) = 0.4, π(Right|s) = 0.4, π(Up|s) = 0.3

(c) π(Left|s) = 0.6, π(Right|s) = 0.5, π(Up|s) = −0.1

**Solution**

(a) **Valid.** All values are ≥ 0 and 0.3 + 0.5 + 0.2 = 1.0.

(b) **Not valid.** 0.4 + 0.4 + 0.3 = 1.1, which is more than 1.

(c) **Not valid.** The sum is 0.6 + 0.5 − 0.1 = 1.0, but a probability cannot be negative (−0.1). Both rules must hold. Checking only the sum is a common mistake.

**Exercise 7.2 — Using a stochastic policy**

In state s the policy is π(Left|s) = 0.25 and π(Right|s) = 0.75. The agent visits state s 400 times. (a) About how many times would you expect it to choose Right? (b) Write the deterministic policy "always Right in s" in the π(a|s) form.

**Solution**

(a) Expected number = 400 × 0.75 = **300** times. (The actual count will vary a little because the choice is random, but it will be close to 300.)

(b) π(Right|s) = 1 and π(Left|s) = 0.

# 8. Value Functions: How Good Is a State?

Rewards tell the agent what is good *right now*. **Values** tell it what is good *in the long run*. A state can give a low reward immediately but still be valuable because it leads to high rewards later. Example: studying tonight gives no immediate reward (it may even feel unpleasant), but it leads to a good exam result later, so the state "have studied" has high value.

## 8.1 A quick reminder: expected value

If a random quantity X can take values $x_1$, $x_2$, … with probabilities $p_1$, $p_2$, …, its **expected value** (long-run average) is:

$$\mathbb{E}[X] = \sum_i p_i \, x_i \qquad (12)$$

Here **E** (a stylised E) means "expected value of", $x_i$ is a possible value and $p_i$ is its probability. Example: a game pays 10 with probability 0.5 and 0 otherwise; E[X] = 0.5 × 10 + 0.5 × 0 = 5.

We need expected values because returns are usually random: the policy may be random, and the environment may also behave randomly.

## 8.2 State-value function

$$v_\pi(s) = \mathbb{E}_\pi[\, G_t \mid S_t = s \,] \qquad (13)$$

- **$v_\pi(s)$**: the **value of state s under policy π**.
- **$\mathbb{E}_\pi[\ \ldots\ ]$**: the expected value, assuming the agent follows policy π from now on.
- **$G_t \mid S_t = s$**: the return, given that the agent is in state s at time t.

In words: **$v_\pi(s)$ is the average return the agent will collect if it starts in state s and then follows policy π.** The value of a terminal state is always 0, because no rewards follow it.

## 8.3 Action-value function

$$q_\pi(s, a) = \mathbb{E}_\pi[\, G_t \mid S_t = s,\ A_t = a \,] \qquad (14)$$

**$q_\pi(s, a)$** is the expected return if the agent is in state s, takes action a *first*, and then follows policy π afterwards. It answers the question "how good is it to take action a in state s?". This is the "Q" in **Q-learning**, which you will study later.

## 8.4 How v and q are connected

The value of a state is the average of the action values, weighted by how likely the policy is to choose each action:

$$v_\pi(s) = \sum_{a \in \mathcal{A}(s)} \pi(a \mid s)\, q_\pi(s, a) \tag{15}$$

In words: from state s, the policy picks action a with probability π(a | s), and doing so is worth $q_\pi$(s, a). Averaging over all actions gives $v_\pi$(s).

**Exercise 8.1 — Value as an average return**

From state s, following policy π, the agent ends up with a return of 10 in 80% of episodes and a return of −5 in 20% of episodes. What is $v_\pi$(s)?

**Solution**

$v_\pi(s) = 0.8 \times 10 + 0.2 \times (-5) = 8 - 1 =$ **7**.

**Exercise 8.2 — From q-values to the state value**

In state s there are two actions with $q_\pi$(s, Left) = 4 and $q_\pi$(s, Right) = 7. The policy is π(Left|s) = 0.5, π(Right|s) = 0.5.

(a) Find $v_\pi$(s). (b) Which action looks better? (c) If the policy were changed to always pick that better action (keeping the same q-values), what would the value of s become?

**Solution**

(a) $v_\pi(s) = 0.5 \times 4 + 0.5 \times 7 = 2 + 3.5 =$ **5.5**.

(b) **Right**, because its action value (7) is higher than that of Left (4).

(c) With π(Right|s) = 1: value = 1 × 7 + 0 × 4 = **7**, an improvement from 5.5. Choosing actions with higher q-values is the basic idea behind improving a policy. (Strictly, changing the policy can also change the q-values of other states; this simple calculation shows only the idea.)

**Exercise 8.3 — Values in the corridor**

Use the corridor of Figure 2 with the policy "always move Right". Moving into B or C gives −1, moving into D gives +10, and D is terminal. Let γ = 0.9. Compute v(D), v(C), v(B) and v(A).

**Solution**

Because the policy and the moves are deterministic, each value equals the single return obtained from that state. Work backwards with v(s) = (next reward) + γ × v(next state):

v(D) = **0** (terminal state).

v(C) = +10 + 0.9 × v(D) = 10 + 0 = **10**.

v(B) = −1 + 0.9 × v(C) = −1 + 9 = **8**.

v(A) = −1 + 0.9 × v(B) = −1 + 7.2 = **6.2**.

**Check directly:** from A the rewards are −1, −1, +10, so $G_0$ = −1 + 0.9 × (−1) + 0.81 × 10 = −1 − 0.9 + 8.1 = 6.2 ✓ States closer to the goal have higher value, as we would expect.

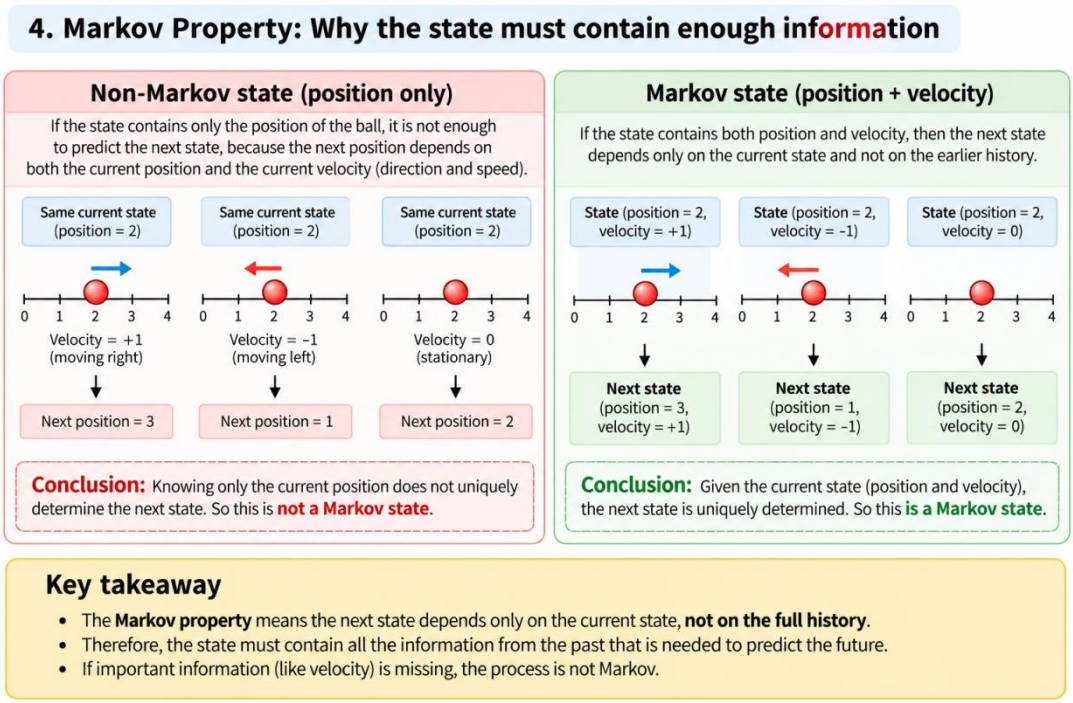
## 9. The Markov Property: What Should a State Contain?

A state is useful only if it contains enough information to decide what to do. The ideal is a state with the **Markov property**: the future depends only on the present state and action, not on how the agent got there.

$$\Pr\{S_{t+1} = s',\, R_{t+1} = r \mid S_t, A_t\} = \Pr\{S_{t+1} = s',\, R_{t+1} = r \mid S_0, A_0, R_1, \ldots, S_t, A_t\} \quad (16)$$

- **Left side:** the probability of the next state being s′ and the next reward being r, given only the current state $S_t$ and action $A_t$.
- **Right side:** the same probability, given the *entire history* $S_0$, $A_0$, $R_1$, …, $S_t$, $A_t$.
- **s′** (read “s-prime”): a possible next state.

If both sides are equal, the history adds no useful information beyond the current state. The state is then a complete summary of the past. Think of a chess board: the current position (together with whose turn it is and a few rule details such as castling rights) tells you everything you need. How the pieces got there does not matter.




## 9.1 Environment dynamics

When the Markov property holds, the behaviour of the environment can be described completely by one function:

$$p(s', r \mid s, a) = \Pr\{S_{t+1} = s',\ R_{t+1} = r \mid S_t = s,\ A_t = a\} \qquad (17)$$

**p(s', r | s, a)** is the probability that, after taking action a in state s, the environment moves to state s' and gives reward r. This function is called the **dynamics** (or model) of the environment. An RL problem whose states have the Markov property is called a **Markov Decision Process (MDP)**, the topic of the next tutorial.

**Exercise 9.1 — Is the state Markov?**

1. Tic-tac-toe, where the state is the current board.
2. Predicting where a moving ball will be next, when the state is a single still photo of the ball.
3. The same ball, when the state is its position and its velocity.
4. The same ball, when the state is the last two photos taken a fraction of a second apart.

**Solution**

1. **Markov.** The board (with whose turn it is, which can be worked out by counting the marks) contains everything needed to continue the game.
2. **Not Markov.** One photo shows where the ball is but not which way or how fast it is moving. The history (earlier photos) would help, so the photo alone is not a complete state.
3. **Markov** (for simple motion). Position and velocity are enough to predict the next position.
4. **Approximately Markov.** Two photos let the agent estimate the velocity from the change in position. This is why game-playing RL systems that learn from screen images often stack several recent frames as the state.

**Exercise 9.2 — Reading the dynamics**

The corridor floor is slippery. In state B, if the agent chooses Right: with probability 0.9 it moves to C with reward −1; with probability 0.1 it slips and stays in B with reward −1.

(a) Write the two non-zero values of p(s′, r | B, Right). (b) Check that they form a valid probability distribution. (c) What is the expected immediate reward of choosing Right in B?

**Solution**

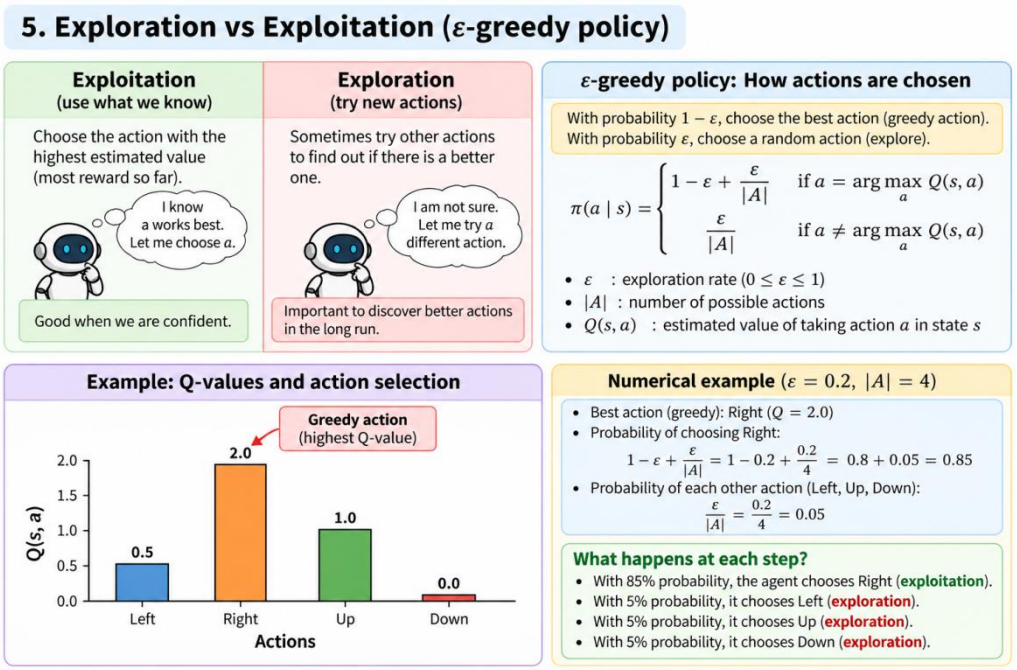
(a) p(C, −1 | B, Right) = 0.9 and p(B, −1 | B, Right) = 0.1. All other combinations have probability 0.

(b) Both are ≥ 0 and 0.9 + 0.1 = 1 ✓.

(c) Expected reward = 0.9 × (−1) + 0.1 × (−1) = **−1**. (Both outcomes give −1, so the expected reward is −1 whatever the probabilities.)

## 10. Exploration versus Exploitation

Imagine choosing where to eat lunch. You know a canteen that is reliably good. There is also a new restaurant you have never tried. Going to the known canteen is **exploitation**: using what you already know to get a good result. Trying the new restaurant is **exploration**: gathering information that might lead to something even better. If you only exploit, you may never discover a better option. If you only explore, you never benefit from what you have learned. RL agents face exactly this dilemma.




## 10.1 The greedy action

Suppose the agent keeps an estimate **Q(s, a)** of how good each action is (capital Q denotes an *estimate*, as opposed to the true $q_\pi$). The **greedy action** is the one with the highest estimate:

$$a^* = \arg\max_{a \in \mathcal{A}} Q(s, a) \tag{18}$$

**arg max** means "the argument (here, the action) that gives the maximum value", and a *(read "a-star") is that best-looking action. Note the difference: $max_a$ Q(s,a) is the largest* value*, while arg $max_a$ Q(s,a) is the* action* that achieves it. If several actions tie, any one of them can be chosen.

## 10.2 The ε-greedy policy

The simplest way to balance exploration and exploitation is the **ε-greedy** policy (ε is the Greek letter "epsilon", a small number such as 0.1). At each step:

- with probability 1 − ε, take the greedy action (exploit);
- with probability ε, pick an action uniformly at random from all |𝒜| actions (explore). The random pick may also land on the greedy action.

Combining these gives the probability of each action:

$$\pi(a \mid s) = \begin{cases} 1 - \varepsilon + \dfrac{\varepsilon}{|\mathcal{A}|}, & \text{if } a = a^* \\ \dfrac{\varepsilon}{|\mathcal{A}|}, & \text{otherwise} \end{cases} \tag{19}$$

- **ε**: the exploration rate, 0 ≤ ε ≤ 1.
- **|𝒜|**: the number of actions available.
- **ε / |𝒜|**: the chance that the random exploration step picks one particular action.

- The greedy action gets 1 − ε (from exploiting) **plus** ε / |𝒜| (because a random pick can also land on it).

**Exercise 10.1 — Computing ε-greedy probabilities**

An agent has |𝒜| = 4 actions and uses ε = 0.2. Action $a_2$ currently has the highest estimate Q. Find the probability of choosing each action, and check that they add to 1.

**Solution**

Each action's share of exploration: ε / |𝒜| = 0.2 / 4 = 0.05.

Greedy action $a_2$: 1 − 0.2 + 0.05 = **0.85**.

Each of the other three actions: **0.05**.

Check: 0.85 + 3 × 0.05 = 0.85 + 0.15 = 1.00 ✓

**Exercise 10.2 — The price of exploring**

Continuing Exercise 10.1, suppose the true average rewards of the four actions are 5 (for $a_2$, the greedy one) and 1, 2, 3 for the others. (a) What is the expected reward per step under the ε-greedy policy? (b) What would it be under the purely greedy policy? (c) Why might exploring still be worth it?

**Solution**

(a) 0.85 × 5 + 0.05 × 1 + 0.05 × 2 + 0.05 × 3 = 4.25 + 0.05 + 0.10 + 0.15 = **4.55**.

(b) Always choosing $a_2$ gives **5**.

(c) Here the agent's estimates happen to be correct, so exploring costs 0.45 per step. In real learning the estimates start out wrong. Without exploration, an agent might lock onto an action that only *looked* best early on and never discover the truly best action. A small, steady cost of exploration buys the chance of finding much better actions. In practice ε is often reduced gradually as the estimates improve.

**Exercise 10.3 — Extreme values of ε**

With |𝒜| = 4, use the ε-greedy formula to find the probability of the greedy action when (a) ε = 0 and (b) ε = 1. Describe the behaviour in each case.

**Solution**

(a) ε = 0: greedy action gets 1 − 0 + 0/4 = **1**. Every other action gets 0. The agent is **purely greedy** and never explores.

(b) ε = 1: greedy action gets 1 − 1 + 1/4 = **0.25**, and every other action also gets 1/4 = 0.25. The agent acts **completely at random**.

## 11. A Map of RL Methods

You do not need the details yet, but it helps to know how RL methods are grouped. These terms will reappear throughout the course.

| Distinction | Meaning | Example |
|---|---|---|
| **Model-based** vs **model-free** | Model-based methods know or learn the dynamics p(s′, r \| s, a) and use them to plan ahead. Model-free methods learn directly from experience without a model. | Value iteration (model-based planning); Q-learning (model-free) |
| **Value-based** vs **policy-based** | Value-based methods learn value estimates (such as Q) and act greedily with respect to them. Policy-based methods learn the policy π directly. | Q-learning (value-based); policy-gradient methods (policy-based) |
| **On-policy** vs **off-policy** | On-policy methods learn about the same policy they use to act. Off-policy methods learn about one policy (for example the greedy one) while acting with another (for example ε-greedy). | SARSA (on-policy); Q-learning (off-policy) |

**Planning** means computing a good policy using a known model of the environment, without having to act in it. **Learning** means improving the policy from actual experience. Both appear later in this course.

**Exercise 11.1 — Classify the method**

1. A chess program that uses the known rules of chess to look ahead several moves before choosing one.
2. An agent that learns Q(s, a) only from the rewards and states it experiences, without knowing p(s′, r | s, a).
3. A robot that, before acting, computes the best route using an exact map of a building and known movement costs.

**Solution**

1. **Model-based** (planning). The rules of chess are a perfect model of what each move does.
2. **Model-free** and **value-based**. This describes Q-learning.
3. **Model-based planning.** The map and costs are a model; no trial and error is needed.

## 12. Putting It All Together: A Complete Worked Example

We now use every idea from this tutorial on one small problem: the corridor of Figure 2.

| Element | Specification |
|---|---|
| States  | {A, B, C, D}; D is terminal |
| Actions  | {Left, Right} in A, B, C |
| Dynamics | Deterministic. Right moves one cell right. Left moves one cell left. Left in A bumps the wall and stays in A. |
| Rewards | +10 for any move that enters D; −1 for every other move |
| Task type | Episodic (ends on reaching D) |
| Discount | $\gamma = 0.9$ |

### 12.1 Policy 1: always move Right

From Exercise 8.3: $v_1(C) = 10$, $v_1(B) = 8$, $v_1(A) = 6.2$, $v_1(D) = 0$.

### 12.2 Policy 2: Right in A, Left in B, Right in C

Starting in A, the agent moves to B (reward −1), then back to A (reward −1), then to B again, and so on. It never reaches D. The episode never ends and every reward is −1, so

$$v_2(A) = -1 + 0.9(-1) + 0.9^2(-1) + \ldots = -1 / (1 - 0.9) = \mathbf{-10}.$$

By the same reasoning $v_2(B) = -10$, while $v_2(C) = 10$ because from C the agent moves straight into D.

### 12.3 Comparing actions with q-values

Using the values of Policy 1, how good would it be to take Left once in B and then follow Policy 1?

$$q_1(B, \text{Left}) = -1 + 0.9 \times v_1(A) = -1 + 0.9 \times 6.2 = -1 + 5.58 = \mathbf{4.58}$$

$$q_1(B, \text{Right}) = -1 + 0.9 \times v_1(C) = -1 + 0.9 \times 10 = \mathbf{8}$$

Right is clearly better in B (8 > 4.58). Doing the same in A gives $q_1$(A, Left) = −1 + 0.9 × 6.2 = 4.58 and $q_1$(A, Right) = −1 + 0.9 × 8 = 6.2, so Right is better there too. Policy 1 picks the best action in every state; it reaches the goal in the fewest possible steps, so it is the best (optimal) policy for this problem.

**Exercise 12.1 — Final challenge**

In the same corridor, change the discount to γ = 0.5 (everything else unchanged) and use Policy 1 (always Right).

(a) Compute v(C), v(B), v(A). (b) Is v(A) still positive? What does this tell you about how γ affects the agent's view of a distant goal?

**Solution**

(a) Work backwards with v(s) = (next reward) + 0.5 × v(next state):

v(C) = 10 + 0.5 × 0 = **10**

v(B) = −1 + 0.5 × 10 = −1 + 5 = **4**

v(A) = −1 + 0.5 × 4 = −1 + 2 = **1**

(b) Yes, v(A) = 1 is still positive, but much smaller than 6.2 (the value with γ = 0.9). With a small γ, the reward of +10 two moves away is heavily discounted, so the goal looks only slightly attractive from A. If the corridor were longer, the step penalties could outweigh the discounted goal reward and v(A) would become negative. The choice of γ directly shapes how far ahead the agent "sees".

# 13. Summary

## 13.1 Notation at a glance

| Symbol | Read as | Meaning |
|---|---|---|
| t | time step | 0, 1, 2, … ; counts the steps of interaction |
| T | final time | The time step at which an episode ends |
| $S_t$, s, s′ | state | State at time t; a particular state; a next state |
| S | state space | Set of all states |
| $A_t$, a | action | Action at time t; a particular action |
| 𝒜(s), \|𝒜\| | action set | Actions available in state s; number of actions |
| $R_t$, r | reward | Reward received at time t; a particular reward value |
| $G_t$ | return | (Discounted) sum of rewards after time t |

| Symbol | Read as | Meaning |
|---|---|---|
| $\gamma$ | gamma | Discount factor, $0 \le \gamma \le 1$ |
| $\pi(s)$, $\pi(a\|s)$ | pi | Deterministic / stochastic policy |
| $v_\pi(s)$ | state value | Expected return from s following $\pi$ |
| $q_\pi(s,a)$ | action value | Expected return from s, taking a first, then following $\pi$ |
| $Q(s,a)$ | Q-estimate | The agent's current estimate of an action value |
| $p(s', r \mid s, a)$ | dynamics | Probability of next state s' and reward r after action a in state s |
| $\varepsilon$ | epsilon | Exploration rate in $\varepsilon$-greedy |
| $\mathbb{E}[\cdot]$ | expectation | Average (expected) value |

## 13.2 Key equations

| Name | Equation |
|---|---|
| Discounted return | $G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots$ |
| Recursive return | $G_t = R_{t+1} + \gamma G_{t+1}$, with $G_T = 0$ |
| Constant reward forever | $G = r / (1 - \gamma)$, for $\gamma < 1$ |
| State value | $v_\pi(s) = \mathbb{E}_\pi[G_t \mid S_t = s]$ |
| Action value | $q_\pi(s,a) = \mathbb{E}_\pi[G_t \mid S_t = s, A_t = a]$ |
| Link between v and q | $v_\pi(s) = \sum_a \pi(a\|s)\, q_\pi(s,a)$ |
| $\varepsilon$-greedy | greedy action: $1 - \varepsilon + \varepsilon/\|\mathcal{A}\|$; each other action: $\varepsilon/\|\mathcal{A}\|$ |

## 13.3 Common mistakes to avoid

- Writing $R_0$. The first reward is $R_1$.
- Including $R_t$ in $G_t$. The return starts from $R_{t+1}$.
- Using $\gamma = 1$ in a continuing task. The return may become infinite.
- Checking only that policy probabilities add to 1 and forgetting that none may be negative.
- Forgetting that the greedy action in $\varepsilon$-greedy also gets the share $\varepsilon/|\mathcal{A}|$ from random exploration.
- Confusing reward (immediate) with value (long-term expected return).
- Confusing max (the highest value) with arg max (the action that gives it).

## 13.4 Quick self-test

**Exercise 13.1 — Five quick questions**

1. Which signal in RL tells the agent how good its action was?
2. Rewards 5, 5, 5 then the episode ends; $\gamma = 0.5$. What is $G_0$?
3. Is $\pi(a_1|s) = 0.7$, $\pi(a_2|s) = 0.3$ a valid policy?
4. What is the value of a terminal state?
5. With $|\mathcal{A}| = 5$ and $\varepsilon = 0.1$, what is the probability of the greedy action?

**Solution**

1. The **reward** $R_{t+1}$.
2. $G_0 = 5 + 0.5 \times 5 + 0.25 \times 5 = 5 + 2.5 + 1.25 =$ **8.75**.
3. **Yes.** Both are non-negative and 0.7 + 0.3 = 1.
4. **0**, because no rewards follow it.
5. $1 - 0.1 + 0.1/5 = 0.9 + 0.02 =$ **0.92**.

## 13.5 What comes next

This tutorial built the framework. The next topics build directly on it: **Markov Decision Processes** formalise the environment using the dynamics $p(s', r \mid s, a)$; **policies and value functions** are studied in depth through the Bellman equations, which grow out of the recursive return $G_t = R_{t+1} + \gamma G_{t+1}$; **planning algorithms** (policy iteration and value iteration) compute optimal policies when the model is known; and the **Q-learning** algorithm learns optimal action values from experience when the model is not known.

## References


- R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, 2nd ed., MIT Press, 2018. Chapter 1 (Introduction) and Chapter 3 (Finite Markov Decision Processes).
- M. Mohri, A. Rostamizadeh and A. Talwalkar, *Foundations of Machine Learning*, 2nd ed., MIT Press, 2018. Chapter on Reinforcement Learning (learning scenario, MDP model, policy).