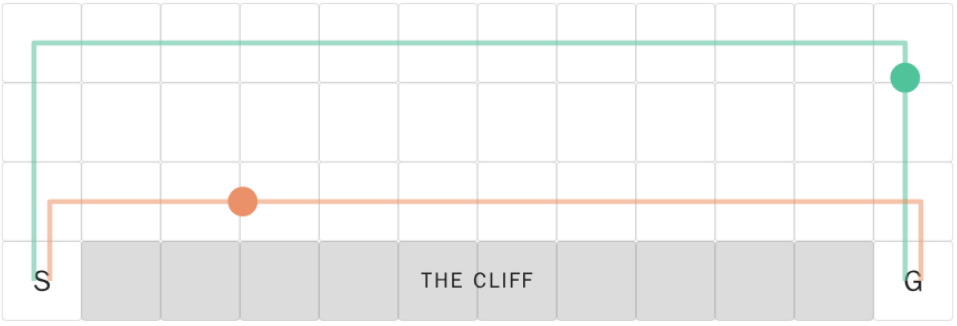
Learning to Act



Reinforcement learning, part 3.2: control without a model, SARSA and Q-learning

AUTHOR

Etienne Chassaing[1, 2]

AFFILIATIONS

1. Hugging Face
2. CentraleSupélec

PUBLISHED

Nov. 2026

Table of Contents

1 Introduction

- 1.1 Where Part 3.1 left us
- 1.2 From evaluating to acting, in a nutshell

2 1. Control, on-policy: Monte Carlo and SARSA

- 2.1 1.1 Learning Q rather than V
- 2.2 1.2 Direct improvement and the exploration problem
- 2.3 1.3 Monte Carlo control
- 2.4 1.4 SARSA: TD for control

3 2. Control, off-policy: Q-learning

- 3.1 2.1 Acting with one policy, learning about another
- 3.2 2.2 Returns from the wrong policy: importance sampling
- 3.3 2.3 Q-learning

4 3. Wrap-up: SARSA or Q-learning?

- 4.1 In a few lines

Introduction

This section continues our journey in learning to make good decisions from the experience we build by interacting with an environment, also named Reinforcement Learning.

Where Part 3.1 left us

Part 3.1, *Learning from Experience* learned the value of a given policy π without a model, from episodes only (as opposed to Part 2, where we computed it exactly). Its two methods (MC and TD) share one update equation, an estimate moved by a step α toward a target:

$$V(S_t) \leftarrow V(S_t) + \alpha \left[\text{target} - V(S_t) \right]$$

where S_t is the state visited at time t.

- Monte Carlo waits for the end of the episode, and its target is the return G_t: unbiased, but noisy.
- Temporal differences (TD) update after every step, and the target is $R_{t+1} + \gamma V(S_{t+1})$, based on the estimate of the next state: biased, but less noisy, and online.

From evaluating to acting, in a nutshell

That was prediction. This page is about control: finding a good policy, still without the model. The method is the same as in Part 2, called generalized policy iteration (GPI), in which we evaluate the policy, make it greedy, and repeat. But, without the model, two things will change from the earlier setup:

- Improving on Q: $\arg\max_a \sum_{s',r} p(s', r \mid s, a)[\dots]$ needs p, so we will learn Q and use $\arg\max_a Q(s, a)$ to get rid of the model.
- Adding a new ingredient, exploration: trying actions that we do not yet believe are best, to compensate for our uncertainty.

We will study:

1. On-policy control: Monte Carlo control and SARSA, and the exploration problem.
2. Off-policy control: importance sampling, then Q-learning.
3. SARSA or Q-learning? The famous cliff example as a wrap-up.

1. Control, on-policy: Monte Carlo and SARSA

As a quick reminder, on-policy means that we improve the very policy we use to explore the environment; we will compare on-policy and off-policy in more details later.

1.1 Learning Q rather than V

The previous part taught us how to evaluate a given policy from experience only. But our recipe for improving a policy still needs the model p. A simple trick allows us to keep our GPI paradigm: learning Q rather than V saves the method.

First, notice that knowing v_* is not enough to act. To be greedy with respect to it, we must look one step ahead (Part 1, section 4.1 shows it), recall that:

$$\pi_*(s) = \arg\max_a \underbrace{\sum_{s',r} p(s', r \mid s, a) \left[r + \gamma\, v_*(s') \right]}_{q_*(s,a)}$$

and that needs the model p.

But, with the action value function Q, the same step is a lookup in the q_* table:

$$\pi_*(s) = \arg\max_a q_*(s, a)$$

The sum under the brace above is exactly $q_*(s, a)$: the averaging over p is already done inside q_*, so improving the policy only compares the m values of a row of the Q-table, with no model.

The Q-table gives one number per state *and* action, and that is enough to make good decisions: this is why, without a model, we estimate Q directly. With this, GPI remains the same recipe, a loop of two steps:

$$\pi \overset{\textbf{evaluation}}{\longrightarrow} Q \approx q_\pi \overset{\textbf{improvement}}{\longrightarrow} \pi'(s) = \arg\max_a Q(s, a) \longrightarrow \cdots$$

From now on, evaluation estimates Q from episodes (Monte Carlo) or transitions (TD), and improvement reads the best action in a row of the Q-table.

1.2 Direct improvement and the exploration problem

But this approach introduces a new challenge, let's introduce it with a small example. Two doors are in front of you, and you always open the one with the highest value so far.

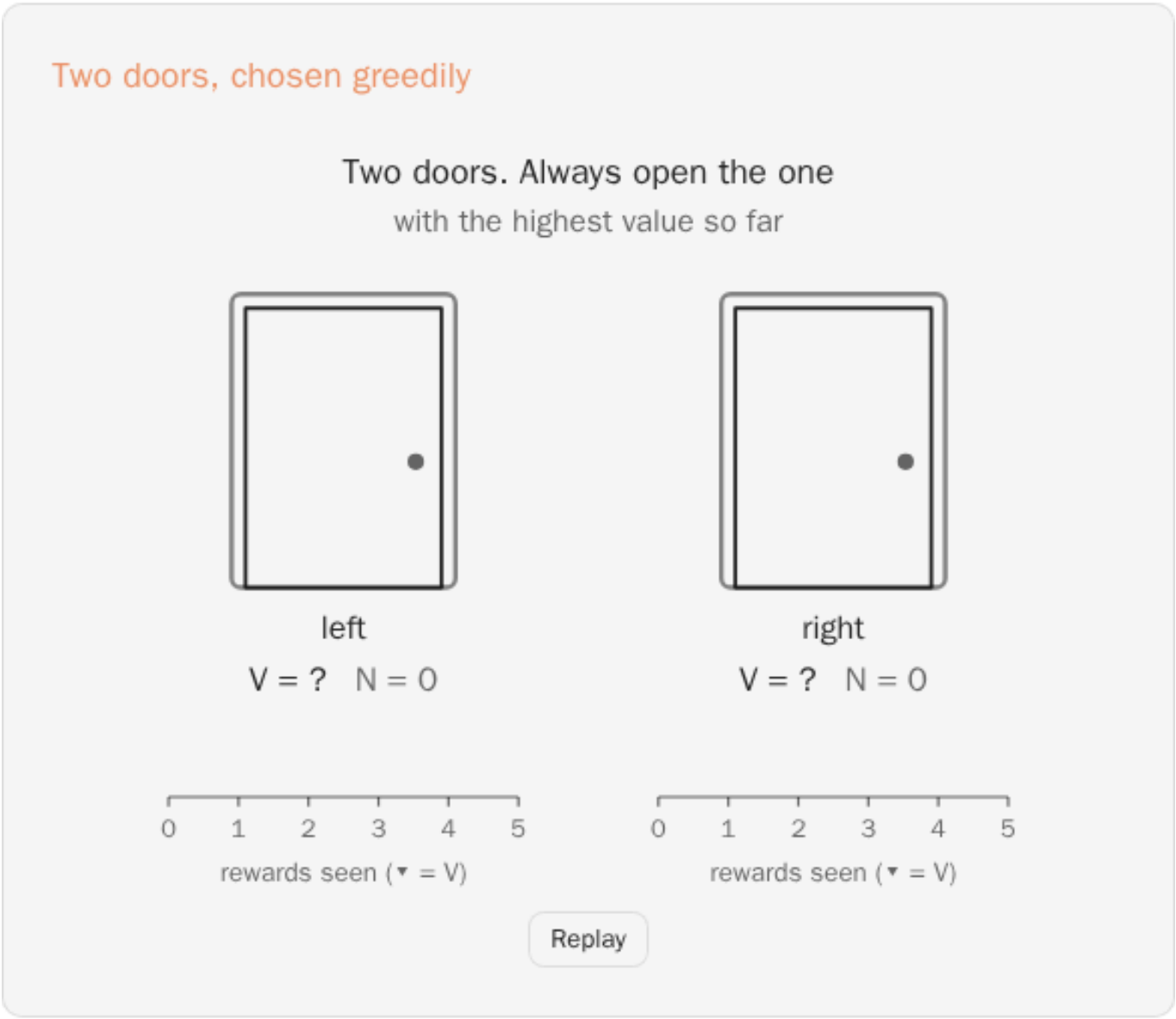


Figure 1 · Always open the door with the highest value so far. Left gives 0 once; right gives +1, +3, +2, so V(right) = +2 and right is opened again and again. One sample says little about the left door, but a greedy agent will never open it again to find out.

Are you sure you chose the best door? The left one was tried once. A purely greedy agent will never open it again, and never find out that it might be better. With a deterministic policy, only one action per state is ever tried: the others never get a return, so they cannot be compared.

Think about how you could solve that; take a few seconds.

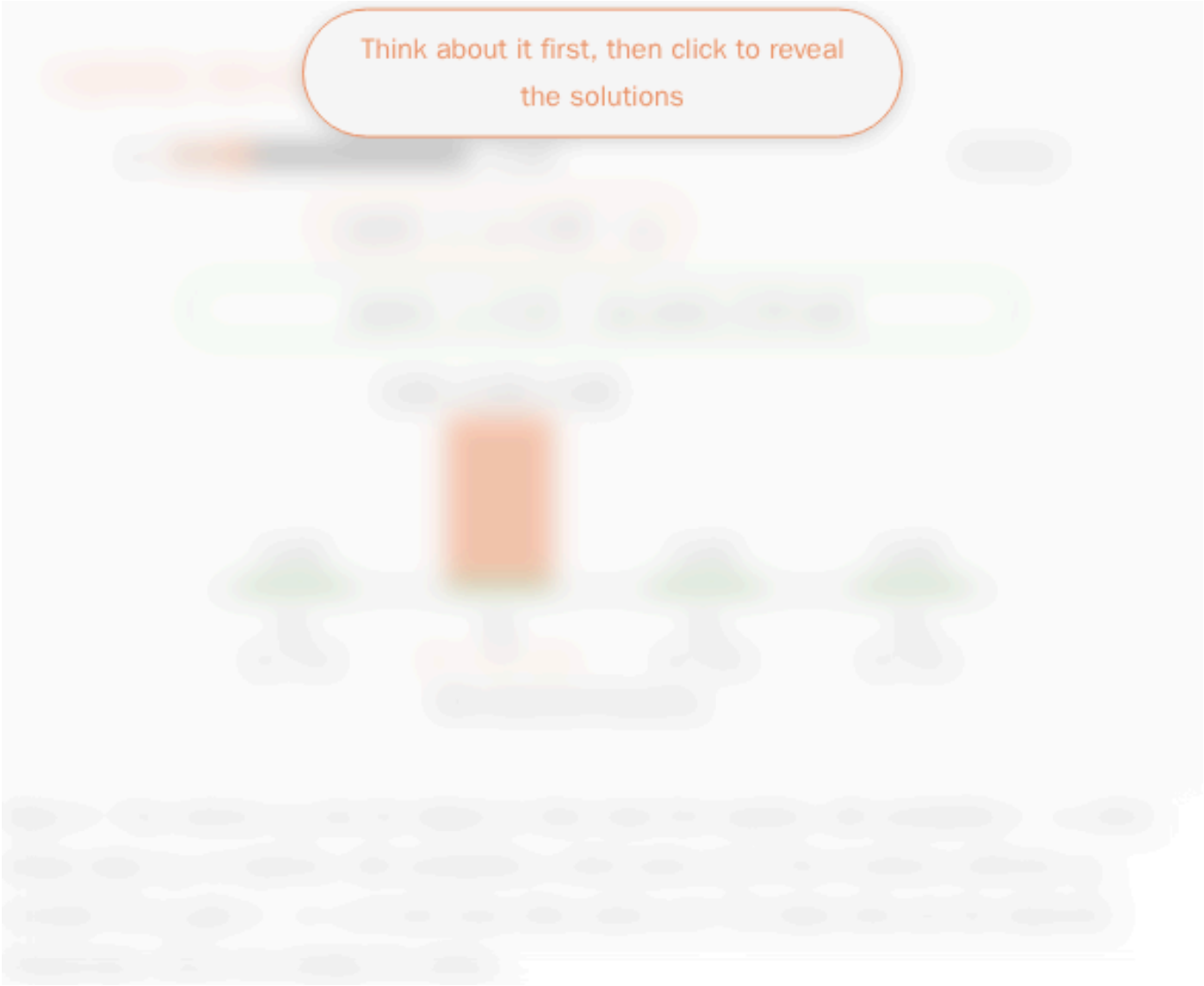
Two ways to solve this problem:

- Exploring starts: start each episode from a random pair (s, a), every pair having a nonzero probability. Every pair is then visited infinitely often in the limit. But this is rarely realistic: we can rarely choose where a real interaction starts.
- Soft policies: keep $\pi(a \mid s) > 0$ for every action, so that exploration is built into the policy itself.

The standard soft policy is ε-greedy.

> E-GREEDY POLICY
> *With probability $1 - \varepsilon$, play the greedy action; with probability ε, play an action drawn uniformly at random. With m actions,*

$$\pi(a \mid s) = \begin{cases} 1 - \varepsilon + \dfrac{\varepsilon}{m} & \text{if } a = \arg\max_{a'} Q(s, a') \\ \dfrac{\varepsilon}{m} & \text{otherwise} \end{cases}$$



Every action keeps a probability of at least $\frac{\varepsilon}{m}$: such a policy is called ε-soft, and its exploration never stops.

1.3 Monte Carlo control

On-policy methods use one policy for everything: the agent acts with π, and evaluates and improves that same π (next section separates the two roles). On-policy Monte Carlo control is GPI with Q instead of V, and averages of returns instead of expectations. It is the first-visit Monte Carlo prediction of Part 3.1, section 1.2, run on Q, with one extra line: the ε-greedy improvement.

Algorithm On-policy first-visit Monte Carlo control

Input: a small $\varepsilon > 0$

$Q(s,a) \leftarrow 0$, $N(s,a) \leftarrow 0$, and π ε-greedy with respect to Q

repeat for each episode

  generate an episode with π: $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$

  $G \leftarrow 0$

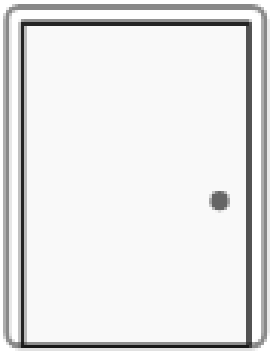
  for $t = T-1, T-2, \ldots, 0$

    $G \leftarrow \gamma G + R_{t+1}$

    if the pair (S_t, A_t) does not appear earlier in the episode

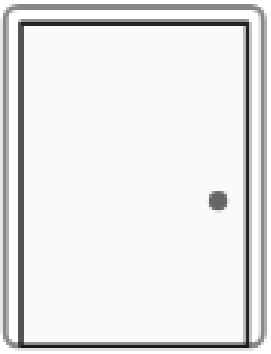
      evaluation: $N(S_t, A_t) \leftarrow N(S_t, A_t) + 1$, $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{1}{N(S_t, A_t)}\left[G - Q(S_t, A_t)\right]$

      improvement: make $\pi(\cdot \mid S_t)$ ε-greedy with respect to $Q(S_t, \cdot)$



left true q = 2.5

Q = 0.00 N = 0



right true q = 2

Q = 0.00 N = 0



Q = 0 and N = 0 for both doors. A tie: π gives each door 0.5

Pause Step Restart

Monte Carlo control on the two doors of section 1.2, seen as a one-state problem: every episode is a single opening, so T = 1 and G = R_1. ϵ = 0.2: the greedy door gets 0.9, the other 0.1. Each door shows its true value q, which the agent does not know: Q should approach it.

Evaluation (orange) averages the returns into Q; improvement (green) makes π ϵ-greedy with respect to the new Q. After an exploratory opening of the left door returns +5, Q(left) passes Q(right) and the policy switches doors: the exploration the greedy agent lacked.

The version with exploring starts (*MC ES*) is the same loop, with each episode started from a random pair (S_0, A_0) and a plain greedy improvement.

Two remarks make this approach work:

First, the improvement is still an improvement: for any ε-soft policy π, the ε-greedy policy π' with respect to q_π is at least as good, $v_{\pi'}(s) \geq v_\pi(s)$ in every state (the proof is in the fold below, if you like maths).

OPTIONAL Why ϵ-greedy improves any ϵ-soft policy: the proof ●●● Prove ›

Second, the evaluation does not need to be complete. Policy iteration in Part 2 evaluated each policy fully before improving it; here each episode only moves Q towards q_π, and the next improvement already uses it. GPI only needs each evaluation step to go in the right direction: more, shorter zigzags, same destination. Switch between the two in the figure below.

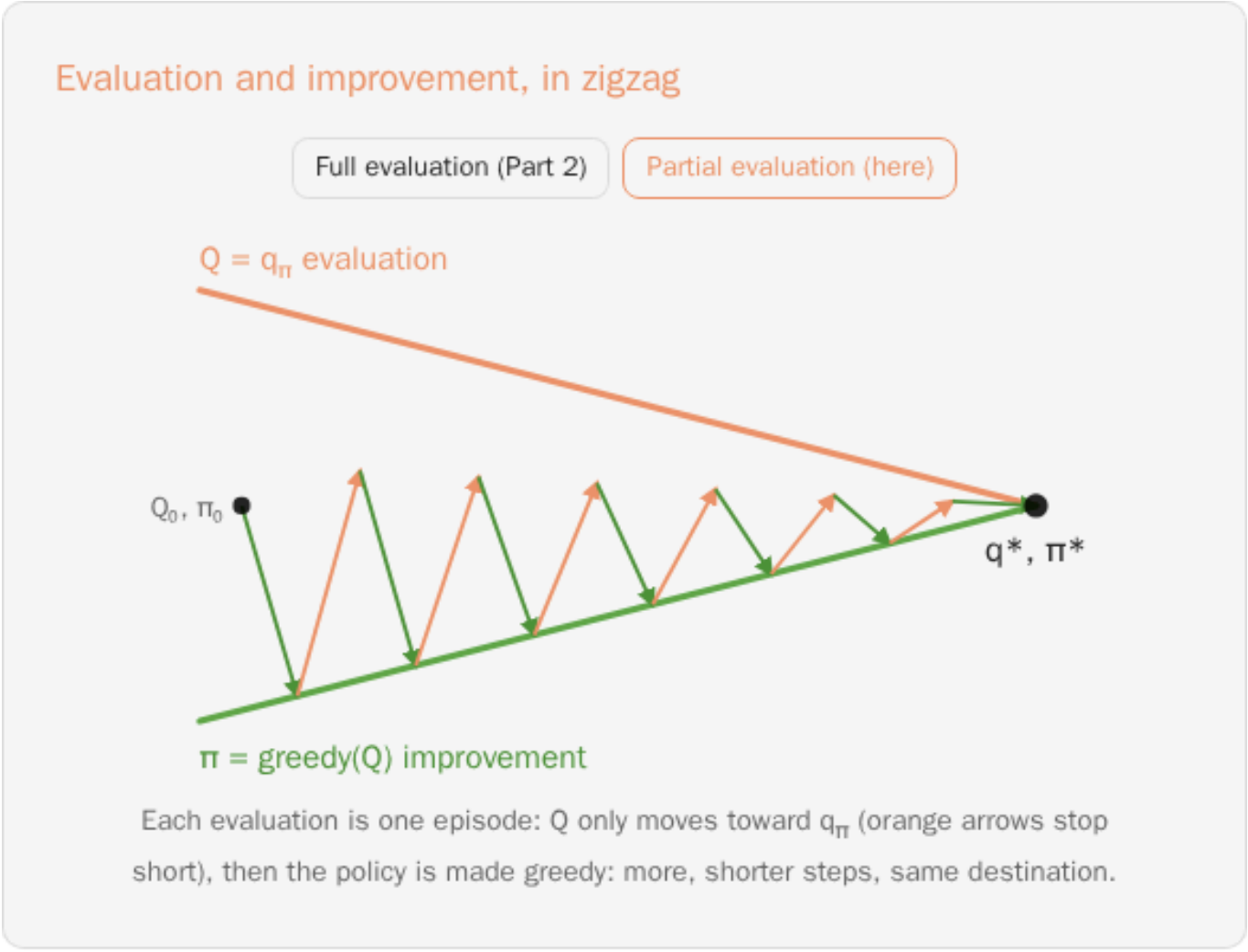


Figure 4 · As in Sutton and Barto (section 4.6): two lines meet at q_*, π_*. The upper line is evaluation, $Q = q_\pi$; the lower one is improvement, π = greedy(Q). Full evaluation (policy iteration, Part 2): every orange arrow reaches the upper line. Partial evaluation (Monte Carlo control, one episode per evaluation): the orange arrows stop short of it, the green ones still reach the greedy line, and there are more, shorter zigzags to the same point.

⚠️ Caution: where does it land?

On the best ε-soft policy, not on π_*: the policy must keep exploring, so it keeps making random moves. To reach π_*, ε must go to 0, slowly enough to keep exploring (GLIE, in the fold at the end of section 1.4).

1.4 SARSA: TD for control

Monte Carlo control inherits the weaknesses of Monte Carlo: high variance, and nothing learned before the end of the episode. The TD update fixes both, so the natural idea is to use TD in the control loop. To do so, we apply TD(0) to $Q(S, A)$, keep the ε-greedy improvement, and update after every step. This is SARSA:

$$\underbrace{Q(S_t, A_t)}_{\text{new estimate}} \leftarrow \underbrace{Q(S_t, A_t)}_{\text{old estimate}} + \alpha \underbrace{\left[\underbrace{R_{t+1} + \gamma\, Q(S_{t+1}, A_{t+1})}_{\text{target}} - Q(S_t, A_t)\right]}_{\text{TD error}}$$

The name is the tuple that each update uses: $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$.

The important detail is A_{t+1}: it is the action really taken next, exploration included.

See what it looks like in the algorithm, on a small example:

ENVIRONMENT A 2 × 3 grid with a pit

Grid	start bottom-left, pit bottom-middle, goal bottom-right; both end the episode
Rewards	-1 per step, -10 for the pit; a wall leaves the agent in place, for -1
Settings	$\gamma = 1$, $\alpha = 0.5$, ε-greedy
Reading	each cell shows its four values $Q(s, \cdot)$, one per side, and its greedy action

Algorithm SARSA

Input: a step size $\alpha \in (0, 1]$, a small $\varepsilon > 0$

$Q(s, a)$ arbitrary, with $Q(\text{terminal}, \cdot) = 0$

repeat for each episode

  initialize S; choose A in S, ε-greedily with respect to Q

  repeat until S is terminal

    take the action A; observe R and S'

    choose A' in S', ε-greedily with respect to Q

    $Q(S, A) \leftarrow Q(S, A) + \alpha \left[R + \gamma Q(S', A') - Q(S, A) \right]$

    $S \leftarrow S', \;\; A \leftarrow A'$

Q(s, a) arbitrary: here, rough guesses



Q(start, →): the value of moving right

−1 per step, −10 for the pit · α = 0.5, γ = 1, ϵ-greedy

S A R S′ A′

Q(pit, ·) = Q(goal, ·) = 0, terminal

Pause Step Restart

SARSA step by step on a 2 × 3 grid (−1 per step, −10 for the pit, α = 0.5, γ = 1), with the line being run highlighted. Each cell shows its four values Q(s, ·) and its greedy action. In the first episode, the agent walks into the pit: Q(start, →) drops and the greedy action of the start flips from → to ↑, which is policy improvement. In the second, the next action A′ is a random move toward the pit, and SARSA's target reads its value, Q(S′, A′) = −6: exploration is part of what SARSA learns.

A' is chosen *before* the update, and it is the action played at the next step: this is what makes SARSA on-policy. Its target includes the cost of its own exploration, which makes it conservative, as section 3 will show on a cliff.

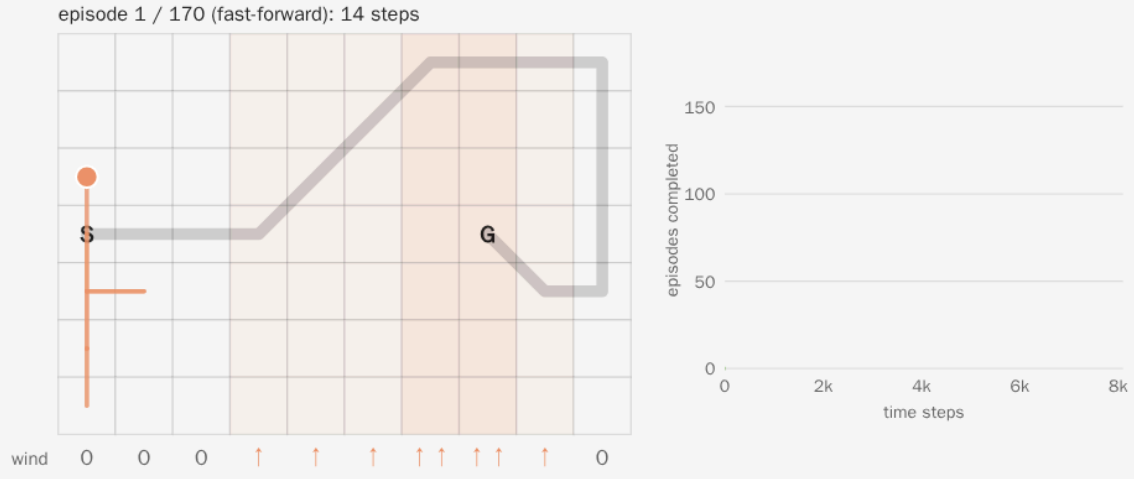
Another example, the Windy Gridworld shows why learning during the episode matters:

ENVIRONMENT Windy Gridworld — Sutton & Barto, Example 6.5 ↗

Grid	7×10, a start and a goal, four moves
Wind	in the middle columns, an upward push of 1 or 2 cells; the straight line to the goal does not work
Rewards	-1 per step, $\gamma = 1$
Best path	15 steps (right 9, down 4, left 2)

curve of the episodes

SARSA in the wind



Fast ×4 | Restart | ▶ Play with it yourself | Open in a new tab ↗

Figure 6 · SARSA on the Windy Gridworld with the book's settings, ϵ = 0.1 and α = 0.5, computed in your browser and replayed. Left: the wind under each column, the shortest path (light gray, the goal), the current episode (orange), and the greedy path, redrawn every 5 episodes (green). Right: the episodes completed against the time steps spent. The first episode takes 468 steps; the curve starts flat, then gets steeper as the episodes get shorter: SARSA learns during the episodes themselves. After 170 episodes, the greedy path is the shortest one, 15 steps. "Play with it yourself" opens the course Space below the figure, on the same environment.

With SARSA, the first episodes are long, then they get shorter and shorter: SARSA learns during the episode. Monte Carlo would be stuck: a policy that never reaches the goal never ends its episode, so it never gets a return to learn from.

WHAT TO LOOK AT in the Space ("Play with it yourself") ›

SARSA converges to the best ε-soft policy, like Monte Carlo control, with much smaller and more frequent steps: each transition does one evaluation update and one improvement step, since the next action is chosen ε-greedily from the updated Q.

If settling for the best ε-soft policy rather than the optimal one annoys you, the fold below shows how these methods can still reach π_*.

OPTIONAL GLIE: explore forever, but less and less Reason

2. Control, off-policy: Q-learning

Q-learning is one of the most famous and influential algorithms in RL. This section gives a formal introduction to it; if you are not math-oriented, you might want to skip section 2.2.

2.1 Acting with one policy, learning about another

On-policy methods learn about the policy they follow, exploration included, but off-policy methods separate the two roles:

- the target policy π is the one we want to evaluate or improve, usually greedy;
- the behavior policy μ is the one that generates the data, usually exploratory.

It might sound crazy to learn from another policy but think of learning chess by watching an amateur play. You never play yourself, yet you want the value of the *best* play, not of the amateur's: μ is the amateur, π is optimal play. Why bother?

Because it lets us reuse old data, or data collected by others, and because it lets the agent keep exploring while learning about the optimal policy.

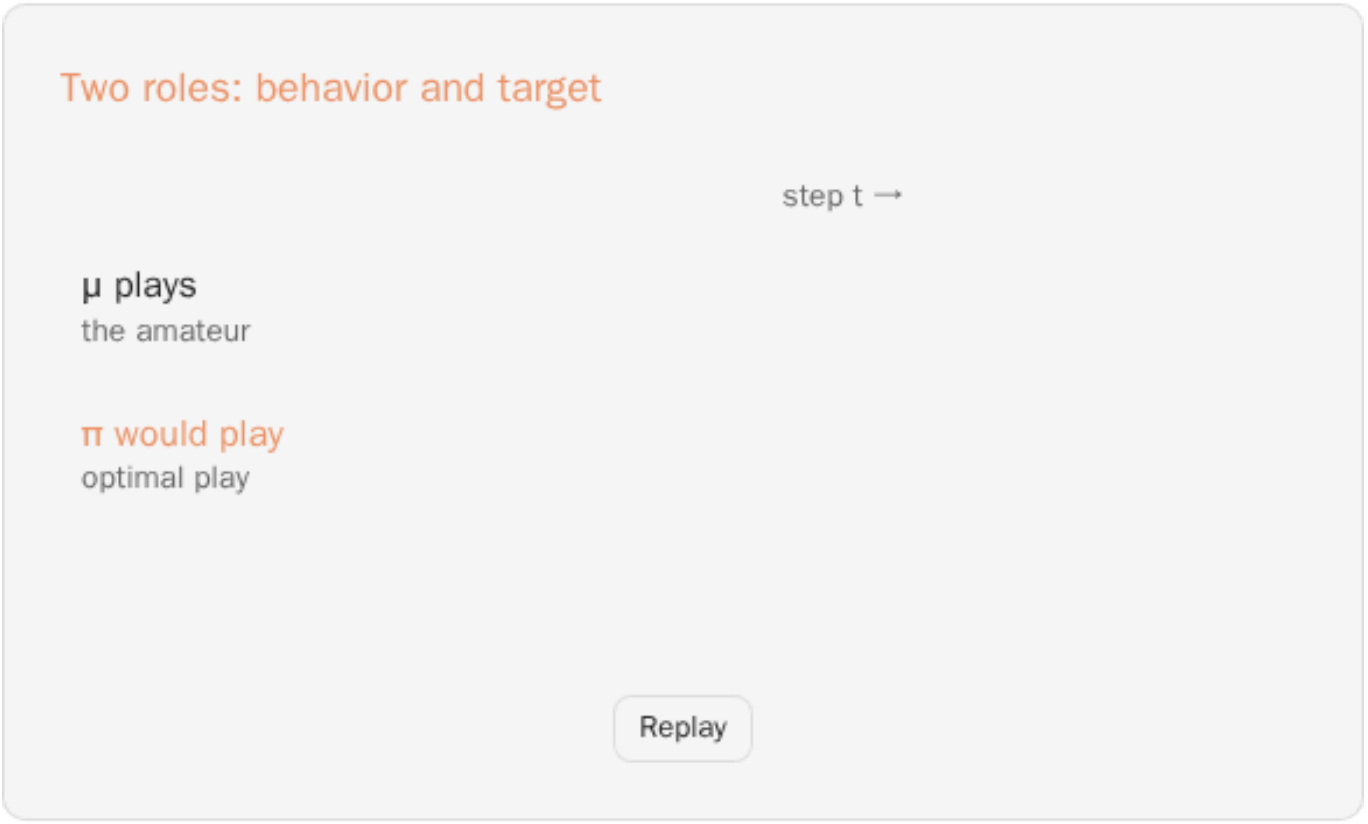


Figure 7 · The behavior policy μ, the amateur, plays one move per step (top row). Below, the move the target policy π, optimal play, would have played in the same position. Where they agree, the step says something directly about π; where they differ, it does not. On-policy the two rows would always agree; off-policy, only a fraction of the steps do.

Off-policy, only a fraction of the transitions are ones that π would have played: the others must be reweighted, or ignored. That is the problem of the next section.

2.2 Returns from the wrong policy: importance sampling

Reason A bit mathematical. If it is not your thing, you can skip to 2.3: Q-learning avoids importance sampling.

Monte Carlo prediction averages the returns it observes. Off-policy, these returns come from μ, so their average converges to the wrong value:

$$\frac{1}{N}\sum_{i=1}^{N} G_i \;\longrightarrow\; \mathbb{E}_{\mu}\big[G_t \mid S_t = s\big] = \boxed{v_{\mu}(s) \;\neq\; v_{\pi}(s)}$$

How can we solve this?

Some trajectories are more likely under π than under μ: they should count more. Others are less likely under π, or impossible: they should count less, or not at all.

The solution is called importance sampling: it keeps the returns collected with μ, and reweights each one by how much more likely its trajectory is under π.

The idea is general. To estimate an expectation under a distribution p with samples drawn from another distribution q:

$$\mathbb{E}_{X\sim p}\big[f(X)\big] = \sum_{x} p(x)\, f(x) = \sum_{x} q(x)\, \frac{p(x)}{q(x)}\, f(x) = \mathbb{E}_{X\sim q}\left[\frac{p(X)}{q(X)}\, f(X)\right]$$

Taking the mean under p is the same as taking the mean of samples drawn from q, after multiplying each sample by the weight $\frac{p(x)}{q(x)}$. Caution: this needs $q(x) > 0$ wherever $p(x) > 0$, which a deterministic q, for instance, does not satisfy.

One example is to get the mean of a fair die while rolling a loaded one:

face	fair die $p(x)$	loaded die $q(x)$	weight $\frac{p(x)}{q(x)}$
1 to 5	$\frac{1}{6}$	$\frac{1}{10}$	$\frac{5}{3}$: too rare, counts more
6	$\frac{1}{6}$	$\frac{1}{2}$	$\frac{1}{3}$: too frequent, counts less

Check: $\frac{1}{2} \times \frac{1}{3} \times 6 + \frac{1}{10} \times \frac{5}{3} \times (1+2+3+4+5) = 1 + 2.5 = 3.5$.

See it live:

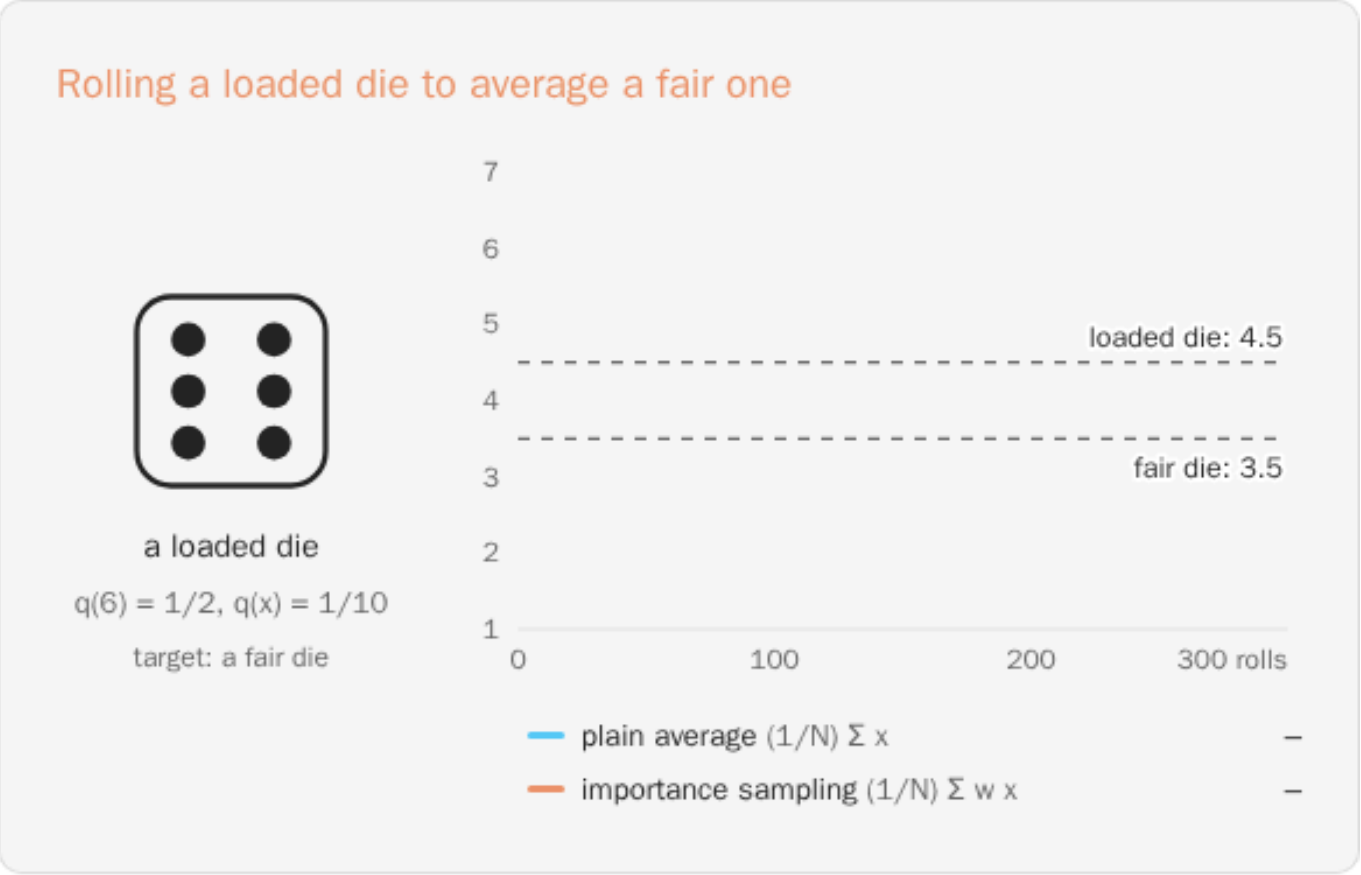


Figure 8 · Rolls of the loaded die, q(6) = 1/2 and q(x) = 1/10 for the other faces. Blue: the plain average of the rolls, which settles at the loaded die's mean, 0.5 × 6 + 0.1 × 15 = 4.5. Orange: each roll multiplied by its weight, 1/3 for a six and 5/3 otherwise, then averaged; it settles at the fair die's 3.5, more noisily.

In reinforcement learning, the sample is the trajectory after S_t, and its weight is the ratio of the trajectory probability under the two policies:

$$\rho_{t:T-1} = \frac{\prod_{k=t}^{T-1} \pi(A_k \mid S_k)\, p(S_{k+1} \mid S_k, A_k)}{\prod_{k=t}^{T-1} \mu(A_k \mid S_k)\, p(S_{k+1} \mid S_k, A_k)} = \boxed{\prod_{k=t}^{T-1} \frac{\pi(A_k \mid S_k)}{\mu(A_k \mid S_k)}}$$

Note that the dynamics cancel: the ratio needs only the two policies, no model.

Then reweighting works:

$$\underbrace{\mathbb{E}_{\mu}}_{\text{average over the episodes of } \mu}\Big[\underbrace{\rho_{t:T-1}}_{\text{weight}}\ \underbrace{G_t}_{\text{return observed}}\ \Big|\ S_t = s\Big] = \underbrace{v_{\pi}(s)}_{\text{value under } \pi}$$

This holds only under the coverage assumption that $\pi(a \mid s) > 0 \Rightarrow \mu(a \mid s) > 0$, or in words, μ must be able to play every action that π may play. Otherwise some trajectories of π never appear in the data, and no weight can bring them back (and we would divide by zero).

Over N visits to s, with ratios ρ_i and returns G_i, the estimate is the average of the reweighted returns, called ordinary importance sampling:

$$V(s) = \frac{1}{N} \sum_{i=1}^{N} \rho_i \, G_i$$

It is unbiased, but its variance can be huge, even infinite, since ρ is a product of ratios. A variant, weighted importance sampling, solves this issue; it is defined in the exercise below, which compares the two. On a blackjack state, the weighted estimate is accurate from the first episodes, while the ordinary one needs many more.

Exercise 2.2 Importance sampling by hand ●○○ Compute Show ›

2.3 Q-learning

Q-learning makes one choice that removes the ratios: its target policy is the greedy policy with respect to Q, $\pi(s) = \arg\max_a Q(s, a)$.

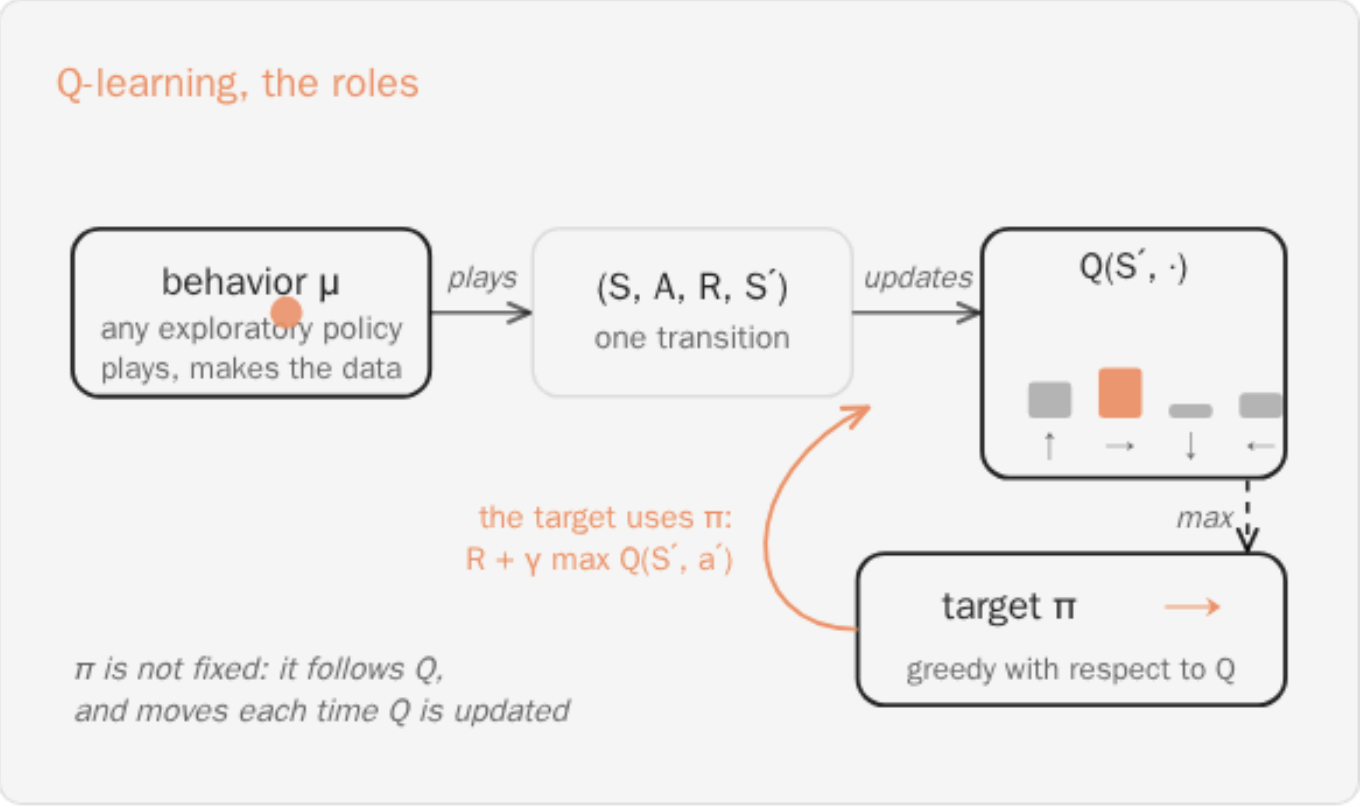


Figure 9 · The behavior policy μ, any exploratory policy, plays and produces the transitions; each one updates Q. The target policy π is greedy with respect to the current Q: watch it flip when an update changes which Q(S′, ·) is largest. The update's target reads π through a max over the next actions.

Here is why that is enough, in three steps.

1. Start from the TD target, SARSA's target, $R_{t+1} + \gamma\, Q(S_{t+1}, A_{t+1})$, is TD(0) applied to Q: it bootstraps on the value of the next action, *sampled* from the policy that acts. To learn about π while μ acts, we want the next action as π would play it, on average: $R_{t+1} + \gamma\, \mathbb{E}_{A' \sim \pi}\big[Q(S_{t+1}, A')\big]$. This π is not a fixed policy: it is greedy with respect to the *current* Q, so it changes each time Q does.

2. A greedy policy has a single action, $\pi(a' \mid s')$ is 1 for the best action and 0 for all the others, so the expectation collapses to one term:

$$\mathbb{E}_{A' \sim \pi}\big[Q(s', A')\big] = \sum_{a'} \pi(a' \mid s')\, Q(s', a') = \max_{a'} Q(s', a')$$

3. Nothing is left to sample, the $\max$ is *computed* from the table, not drawn from a policy: the action that μ plays next is never used. No sample from π means no ratio $\frac{\pi}{\mu}$ to correct it, hence no extra variance.

SARSA could not do the same directly: its A_{t+1} is *sampled* from the policy that acts, so learning about another policy would need the ratio $\frac{\pi(A_{t+1}|S_{t+1})}{\mu(A_{t+1}|S_{t+1})}$.

The update is now:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \Big[R_{t+1} + \gamma \underbrace{\max_{a'} Q(S_{t+1}, a')}_{\text{target}} - Q(S_t, A_t)\Big]$$

The agent acts with any exploratory μ, typically ε-greedy in Q, but its target uses the best next action, not the one played.

Q-learning is to value iteration what TD is to dynamic programming. Value iteration on Q (Part 2) applies the Bellman optimality backup, with the model:

$$q_{k+1}(s, a) = \sum_{s', r} p(s', r \mid s, a)\, \big[r + \gamma \max_{a'} q_k(s', a')\big]$$

Q-learning applies the same backup to one sampled transition $(S_t, A_t, R_{t+1}, S_{t+1})$ instead of the sum over p, and moves Q only a step α toward it.

One update does both jobs of GPI: the $\max$ evaluates and improves at once, so Q is pulled directly towards q_*. Q-learning converges to q_* and π_*, whatever μ is, as long as every pair (s, a) keeps being visited and the step sizes decrease. In doing so, it ignores the cost of exploring, which makes it aggressive, and it can learn from data collected by any policy: that is why it is the standard off-policy control algorithm.

See it in action on the same example as SARSA:

Algorithm Q-learning

Input: a step size $\alpha \in (0, 1]$, a small $\varepsilon > 0$

1 $Q(s, a)$ arbitrary, with $Q(\text{terminal}, \cdot) = 0$

2 repeat for each episode

3 initialize S

4 repeat until S is terminal

5 choose A in S, ε-greedily with respect to Q (the behavior policy μ)

6 take the action A; observe R and S'

7 $Q(S, A) \leftarrow Q(S, A) + \alpha \left[R + \gamma \max_{a'} Q(S', a') - Q(S, A) \right]$

8 $S \leftarrow S'$

Q(s, a) arbitrary: here, rough guesses



Q(start, →): the value of moving right

−1 per step, −10 for the pit · α = 0.5, γ = 1, ε-greedy

S A R S′ $\max_{a'} Q(S', a')$

Q(pit, ·) = Q(goal, ·) = 0, terminal

Pause Step Restart

Q-learning on the same grid, with the same moves as the SARSA figure of section 1.4; only the target changes. The target reads the best entry of the next state, $\max_{a'} Q(S', a')$, whatever is played next. On the step where the next move is a random one toward the pit, SARSA's estimate dropped to −5; Q-learning's rises to −2.5.

Compared with SARSA, one line is gone: the next action is no longer chosen before the update, since the target does not use it.

OPTIONAL SARSA → Q-learning: the two algorithms as a diff, line by line ›

After enough learning, the table becomes the answer: it is one row per state, one column per action, and $Q \approx q_*$. The best entry of a row is then $v_*(s) = \max_a q_*(s, a)$, and its column is the optimal action, $\pi_*(s) = \arg\max_a q_*(s, a)$, which we can read without any model.

Note a small drawback of the Q-table: when two entries of a row are close, say 5.2 and 5.0, a small error in Q is enough to flip the greedy action.

3. Wrap-up: SARSA or Q-learning?

The two control algorithms of this page differ by one term of their target. Let's see what this changes on an example where the agent must walk along a cliff.

ENVIRONMENT Cliff Walking — Sutton & Barto, Example 6.6 ↗

Grid	4×12; start bottom-left, goal bottom-right, the cliff along the bottom row between them
Moves	four, deterministic
Rewards	-1 per step; the cliff gives -100 and sends the agent back to the start
Settings	both act ε-greedily with $\varepsilon = 0.1$, $\alpha = 0.5$: only the update differs

See the two algorithms solving this problem below :

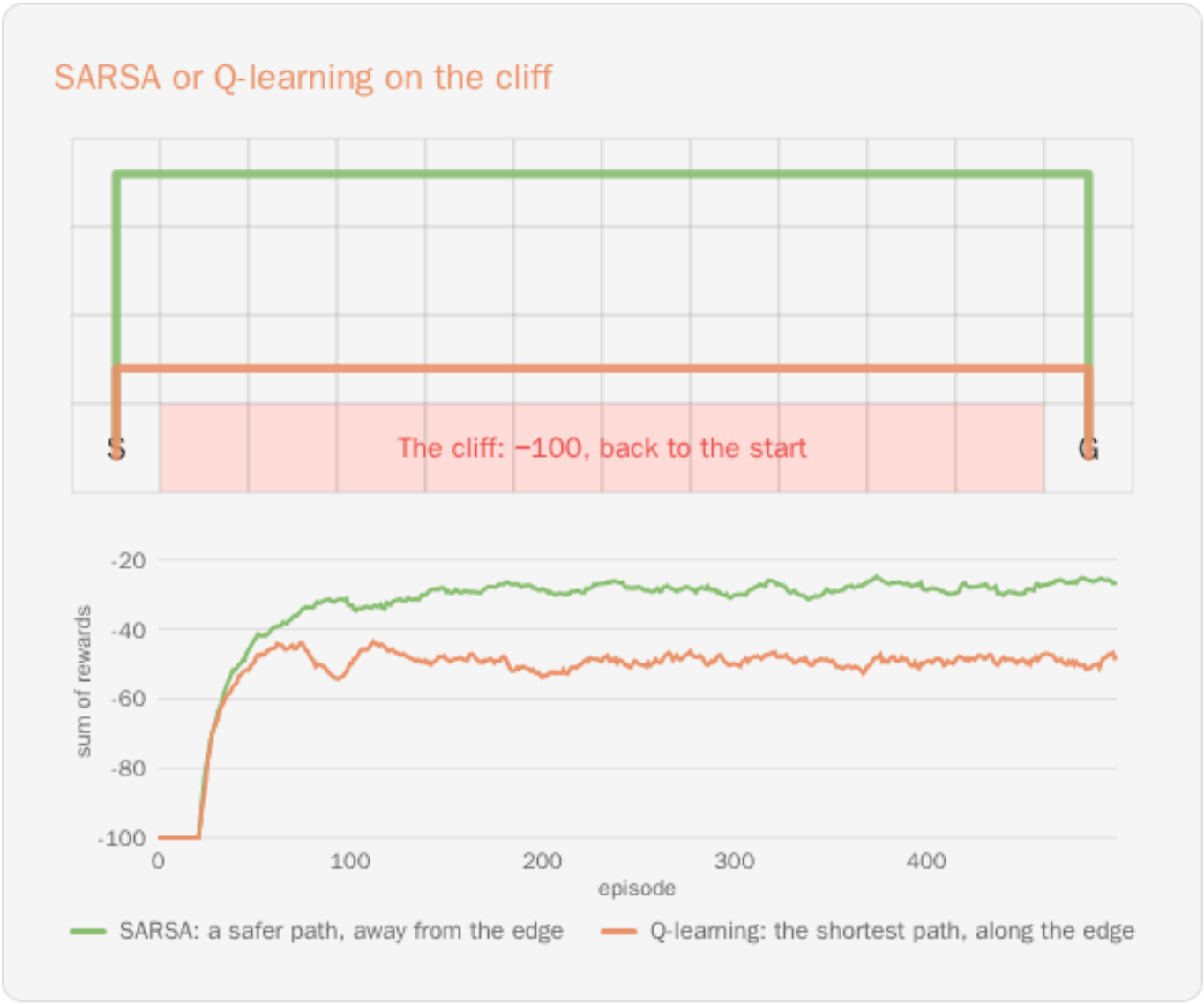


Figure 11 · Both algorithms trained with the book's settings, $\epsilon = 0.1$ and $\alpha = 0.5$, computed in your browser. Top: the greedy path each one has learned. Bottom: the sum of rewards per episode during training, averaged over 50 runs (Sutton and Barto, Figure 6.4). Q-learning's path is the shorter one, but its ϵ-greedy behavior falls off the edge from time to time: during training, SARSA collects more reward.

For once, the algorithms don't converge to the same solution and learn two different paths:

- Q-learning learns q_*, so its greedy path is the shortest one, 13 steps right along the edge of the cliff, even though its ε-greedy behavior still falls in from time to time.
- SARSA learns the value of the policy it actually follows, random moves included: next to the cliff, a random move costs -100, so it learns that the edge is dangerous and keeps a safety margin, one or two rows away.

The paradox: during training, SARSA collects more reward per episode than Q-learning (about -25 against -50 in the book's figure), because Q-learning keeps walking along the edge and keeps falling.

Side by side, everything follows from the difference of target:



Figure 12 · The two control algorithms differ by one term of their target; everything else on these cards follows from it. The GPI pictures follow the course slides: SARSA makes small zigzags between $Q = q_\pi$ and its ϵ-greedy line and, with a fixed ϵ, stops at the best ϵ-soft policy; Q-learning keeps π greedy and follows the greedy line to q_*, π_*. "Compare them live" opens the course Space below the cards.

WHAT TO LOOK AT in the Space ("Compare them live")

Rule of thumb: SARSA when mistakes during learning are costly, Q-learning when only the final policy matters.

Exercise 3.1 SARSA vs Q-learning, one update by hand Compute Show

In a few lines

We finally learned to make good decisions (or almost the best ones) from experience only, congrats 😁

This rests on a few building blocks:

- Control: learn Q, so that improvement needs no model.
- Exploration is essential; ε-greedy is the simple baseline.

- SARSA: on-policy, conservative; Q-learning: off-policy, learns q_*.
- Off-policy Monte Carlo needs importance sampling; Q-learning avoids it.
- Tractable only if the environment is fast to evaluate: learning takes many episodes, so a simulator, not the real system.

So, what's next for solving real-world problems?

Every method of Parts 3.1 and 3.2 stores one number per state (V-table), or per state and action (Q-table), in a table. Part 4: From Discrete to Continuous takes up what happens when that table is too large, or when the states are continuous. To handle this, the table $Q(s, a)$ must be replaced by a function: deep Q-networks, policy gradient and actor-critic are our next targets. And most importantly, we will apply them on a real system 👀🤖

Citation

For attribution in academic contexts, please cite this work as

```
Etienne Chassaing (2026). "Learning to Act".
```

BibTeX citation

```
@misc{chassaing2026_learning_to_act,
  title={Learning to Act},
  author={Etienne Chassaing},
  year={2026},

}
```

Reuse

Text and figures are licensed under CC-BY 4.0. Built with the Research Article Template (Thibaud Frère, CC-BY 4.0) and the exercise components of NLP before LLMs (David Louapre).

Made with ❤️ with research article template