

Q-Learning

Aayush Bajaj

July 10, 2026

Contents

1	the mdp	1
2	bellman optimality	2
2.1	value iteration	2
3	temporal-difference learning	2
4	the q-learning update	3
5	off-policy, and sarsa	3
6	exploration	4
7	convergence conditions	4
8	function approximation and dqn	5
9	a gridworld, from scratch	5
10	see also	6

q-learning is the algorithm that made reinforcement learning feel inevitable: interact with an unknown world, nudge a table of numbers after every step, and the table converges to the value of *optimal* behaviour — even while you behave suboptimally the entire time. everything runs on one line of arithmetic, and the rest of this page is the machinery needed to say precisely why that line works. the canonical reference for all of it is sutton & barto’s *reinforcement learning: an introduction*, free at <http://incompleteideas.net/book/the-book-2nd.html>.

watkins introduced it in his 1989 phd thesis; watkins and dayan supplied the convergence proof in 1992

1 the mdp

the world is a **markov decision process** $(S, \mathcal{A}, P, R, \gamma)$:

- S : states; $\mathcal{A}$: actions.
- $P(s' \mid s, a)$: transition kernel — markov, because the next state depends only on the current state and action.
- $R(s, a, s')$: expected immediate reward on that transition (realisations r_{t+1} may be random; marginalising s' gives the $R(s, a)$ some texts use).
- $\gamma \in [0, 1)$: discount factor, taming the infinite-horizon **return** $G_t = \sum_{l=0}^{\infty} \gamma^l r_{t+l+1}$.

a **policy** $\pi(a \mid s)$ chooses actions. the **action-value function** of π is what q-learning estimates the optimal version of:

$$q_\pi(s, a) = \mathbb{E}_\pi[G_t \mid s_t = s, a_t = a], \quad q_*(s, a) = \max_\pi q_\pi(s, a). \quad (1)$$

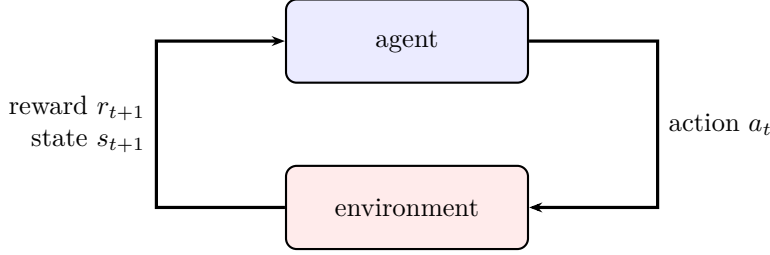


Figure 1: the agent–environment loop: the agent emits a_t , the environment replies with r_{t+1} and s_{t+1} , and the cycle repeats. rl in one picture.

2 bellman optimality

the return telescopes — $G_t = r_{t+1} + \gamma G_{t+1}$ — and taking expectations under optimal play turns that recursion into the **bellman optimality equation**:

$$q_*(s, a) = \mathbb{E} \left[r_{t+1} + \gamma \max_{a'} q_*(s_{t+1}, a') \mid s_t = s, a_t = a \right], \quad (2)$$

with $v_*(s) = \max_a q_*(s, a)$. the max inside the expectation is the whole story: act greedily *once*, then keep acting greedily, and no policy does better. crucially, the right-hand side defines an operator $\mathcal{T}$ on functions q , and for $\gamma < 1$ this operator is a γ -contraction in the sup norm:

$$\|\mathcal{T}q - \mathcal{T}q'\|_\infty \leq \gamma \|q - q'\|_\infty, \quad (3)$$

so by banach’s fixed-point theorem q_* exists, is unique, and repeated application of $\mathcal{T}$ from *any* starting table converges to it geometrically.

2.1 value iteration

if you know P and R , that observation is already an algorithm — iterate the operator:

$$q_{k+1}(s, a) \leftarrow \sum_{s'} P(s' \mid s, a) \left[R(s, a, s') + \gamma \max_{a'} q_k(s', a') \right] \quad \text{for all } (s, a), \quad (4)$$

at cost $O(|\mathcal{S}|^2|\mathcal{A}|)$ per sweep. the catch is the premise: in most interesting problems nobody hands you P . q-learning is what value iteration becomes when the expectation must be *sampled* instead of computed.

3 temporal-difference learning

model-free methods estimate expectations from experienced transitions $(s_t, a_t, r_{t+1}, s_{t+1})$. the monte carlo approach waits for the whole return G_t — unbiased, high variance,

and useless before episode end. **temporal-difference** learning instead bootstraps: replace the tail of the return with the current estimate of it, and move a step toward that target,

$$v(s_t) \leftarrow v(s_t) + \alpha \underbrace{[r_{t+1} + \gamma v(s_{t+1}) - v(s_t)]}_{\text{td error } \delta_t}. \quad (5)$$

biased (the target leans on the current guess) but low-variance and fully online — learning happens at every tick, not every episode.

the bias-variance dial between td and monte carlo is continuous: n-step returns and td-lambda interpolate

4 the q-learning update

apply td to the bellman *optimality* equation rather than to a fixed policy's values and you get q-learning. after each transition $(s_t, a_t, r_{t+1}, s_{t+1})$:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha [r_{t+1} + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t)]. \quad (6)$$

read it as four parts:

- **current estimate** $Q(s_t, a_t)$: the number being improved.
- **target** $r_{t+1} + \gamma \max_{a'} Q(s_{t+1}, a')$: a one-sample, bootstrapped stand-in for the right side of the bellman optimality equation — real reward plus discounted *greedy* value of wherever you landed.
- **td error**: target minus estimate; positive means the transition went better than the table believed.
- **step size** $\alpha \in (0, 1]$: how far to trust one sample.

in expectation the update applies the bellman optimality operator; the contraction does the rest. this is asynchronous stochastic approximation to value iteration, one visited state–action pair at a time.

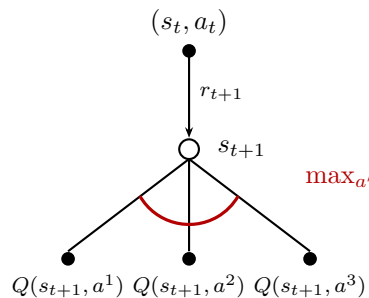


Figure 2: the q-learning backup: from the landed-in state s_{t+1} , back up through the max over next actions (red arc) — not through the action the behaviour policy actually takes next.

5 off-policy, and sarsa

the max makes q-learning **off-policy**: the value being learned is that of the greedy policy, while the data can come from any sufficiently exploratory behaviour — an

ϵ -greedy actor, an old logged dataset, a human. its on-policy sibling sarsa backs up through the action the behaviour policy *actually takes next*:

aspect	q-learning	sarsa
target	$r + \gamma \max_{a'} Q(s', a')$	$r + \gamma Q(s', a')$, a' actually taken
learns value of	greedy policy	the behaviour policy itself
policy class	off-policy	on-policy
exploration risk	ignores it (assumes future greed)	prices it in
cliff-walking	hugs the cliff edge	learns the safe detour
can learn from logs	yes	not directly

neither is “better”: q-learning learns the optimal solution to the mdp; sarsa learns the best policy *among those that keep exploring* — which is why, executed ϵ -greedily on the classic cliff-walk, q-learning’s edge-hugging agent occasionally lemmings off while sarsa’s detour agent collects more reward. deploy accordingly.

6 exploration

behaving greedily against a half-learned table starves the untried actions of data — an action whose Q is pessimistically wrong never gets revisited to be corrected. the standard fix is **ϵ -greedy**: with probability $1 - \epsilon$ take $\arg \max_a Q(s, a)$, otherwise a uniform random action. this supplies the infinite visitation the convergence theorem demands; annealing $\epsilon \rightarrow 0$ at a suitable rate (glie: greedy in the limit with infinite exploration) additionally makes the behaviour policy itself converge to optimal. smarter schemes — optimistic initialisation, ucb-style bonuses, softmax over Q — buy sample efficiency, not correctness.

7 convergence conditions

tabular q-learning converges to q_* with probability 1 under conditions inherited from robbins–monro stochastic approximation:¹

- every state–action pair is visited infinitely often (exploration is a *requirement*, not a nicety);
- per-pair step sizes satisfy $\sum_t \alpha_t = \infty$ and $\sum_t \alpha_t^2 < \infty$ (e.g. $\alpha_t = 1/t$: steps large enough to travel anywhere, small enough that noise averages out);
- rewards bounded, $\gamma < 1$.

practice cheerfully violates the second condition with a small constant α , accepting persistent jitter around q_* in exchange for the ability to track non-stationary worlds. the same robbins–monro conditions moonlight in sgd — see [optimisers](#).

¹watkins & dayan (1992), *q-learning*, machine learning 8(3-4): the original convergence proof for tabular q-learning. robbins & monro (1951) is the general stochastic-approximation scaffold the step-size conditions come from.

references

Goodfellow, Ian (2016). *Deep Learning*, MIT Press.

8 function approximation and dqn

a table dies with the state space: $|\mathcal{S}|$ for pixels or for go is not a number you allocate. replace the table with a parametric $Q_{\theta}(s, a)$ — historically linear features, now deep networks (Goodfellow, Ian, 2016) — and update θ by semi-gradient descent on the td error. honesty first: with function approximation, the combination of bootstrapping and off-policy learning is the “deadly triad”, and the clean convergence guarantee is *gone* — divergence is not hypothetical (baird’s counterexample is a seven-state mdp that blows up with linear features). dqn (mnih et al. 2015, the atari paper) made the combination work in practice with two stabilisers:

- **replay buffer:** store transitions, train on random minibatches. breaks the temporal correlation of consecutive samples (sgd wants i.i.d.-ish data) and reuses each interaction many times.
- **target network:** compute targets $r + \gamma \max_{a'} Q_{\theta^-}(s', a')$ with a frozen copy θ^- , refreshed every few thousand steps, so the regression target stops chasing its own tail every gradient step.

both are engineering answers to statistical problems, and both matter — ablate either and atari performance craters. the max also inflicts systematic overestimation (a max of noisy estimates is biased upward); double q-learning decouples action *selection* from action *evaluation* to cancel most of it.

9 a gridworld, from scratch

tabular q-learning with ϵ -greedy exploration on a 4×4 gridworld: start at the top-left, goal (+10, terminal) at bottom-right, two pits (−10, terminal), a wall, and −1 per step to make dawdling expensive:

```
1 import numpy as np
2
3 rng = np.random.default_rng(0)
4
5 # 4x4 gridworld. S start, G goal (+10, terminal), X pit (-10, terminal).
6 # step reward -1. actions: 0=up 1=down 2=left 3=right.
7 grid = ["S...",
8         ".#.X",
9         "....",
10        "X..G"]
11 H, W = 4, 4
12 moves = {0: (-1, 0), 1: (1, 0), 2: (0, -1), 3: (0, 1)}
13
14 def step(s, a):
15     r, c = divmod(s, W)
16     dr, dc = moves[a]
17     nr, nc = r + dr, c + dc
18     if not (0 <= nr < H and 0 <= nc < W) or grid[nr][nc] == "#":
19         nr, nc = r, c # bump: stay put
20     ns = nr * W + nc
21     cell = grid[nr][nc]
22     if cell == "G": return ns, 10.0, True
23     if cell == "X": return ns, -10.0, True
24     return ns, -1.0, False
25
26 Q = np.zeros((H * W, 4))
27 alpha, gamma, eps = 0.5, 0.95, 0.2
28
29 for ep in range(500):
30     s, done, ret = 0, False, 0.0
```

```

31 while not done:
32     a = rng.integers(4) if rng.random() < eps else int(np.argmax(Q[s]))
33     ns, rew, done = step(s, a)
34     # q-learning update: bootstrap off the greedy action, not the taken one
35     Q[s, a] += alpha * (rew + gamma * np.max(Q[ns]) * (not done) - Q[s, a])
36     s, ret = ns, ret + rew
37 if ep in (0, 9, 49, 499):
38     print(f"episode {ep:>3}: return {ret:>6.1f}")
39
40 arrows = "^v<>"
41 print("\nlearned greedy policy:          state values max_a Q(s,a):")
42 for r in range(H):
43     pol = "".join(grid[r][c] if grid[r][c] in "#GX"
44                   else arrows[int(np.argmax(Q[r * W + c]))] for c in range(W))
45     val = " ".join(f"{np.max(Q[r * W + c]):>6.2f}" for c in range(W))
46     print(f" {pol} {val}")

```

```

episode 0: return -40.0
episode 9: return -21.0
episode 49: return 5.0
episode 499: return 5.0

```

```

learned greedy policy:          state values max_a Q(s,a):
v<v^          3.21  2.05  4.23  -1.46
v#vX          4.44  0.00  7.07   0.00
>>vv          5.72  7.07  8.50  10.00
X>>G          0.00  8.50 10.00   0.00

```

reading the output:

- returns climb from -40 (episode 0: random flailing, pit visits) to the optimal $+5$ by episode 49 — five -1 steps and then $+10$ on the goal-entering move.
- the greedy policy from the start traces down around the wall and across to the goal, never pointing at a pit.
- the values decode as geometry: 10.00 one step from the goal, then $8.50 = -1 + 0.95 \cdot 10$, $7.07 = -1 + 0.95 \cdot 8.5$, and so on back up the path — discounted distance-to-goal, which is exactly what q_* means in this mdp.
- off-policy in action: the ϵ -greedy behaviour keeps stumbling randomly forever (20% of steps), yet the table converges to the values of the *greedy* policy, exploration noise excluded.

10 see also

- [policy gradients](#) — skip the value table and optimise the policy directly
- [cartpole](#) — the classic control benchmark for exactly these methods
- [optimisers](#) — robbins-monro step sizes in their day job, sgd
- [bias-variance decomposition](#) — the td-vs-monte-carlo trade in its supervised costume