Lab 12

Reinforcement Learning

DS-GA 1003 | Machine Learning | Spring 2026

2026.04.30 Presenter by Yihuai Hong

Today's plan

~60 minutes, with breathing room for questions

01

Concepts

≈20 min

Agent · environment · reward · policy — built up from scratch with a familiar analogy

02

Live-code

≈25 min

Two functions written together: the Q update, then the training loop

03

Results

≈15 min

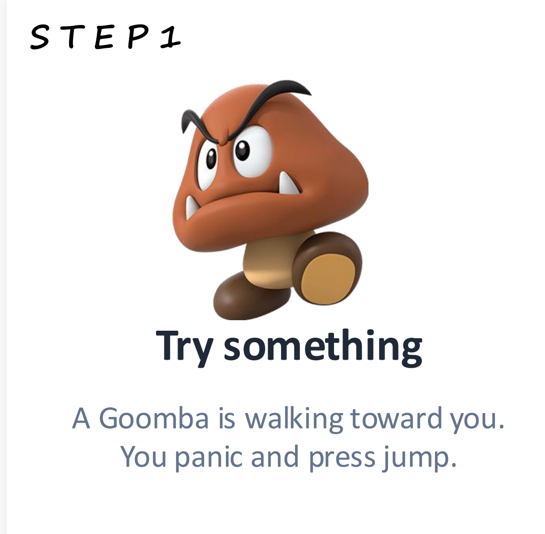
Train, watch the agent learn, discuss what changes when we tweak the knobs

SCORE
0
COINS
0
WORLD
1-1
TIME
386
LIVES
3
SUPER
MARIO BROS.
UNPAUSE

Before any math: a familiar situation

Imagine the first time you played Super Mario — no tutorial, no manual.

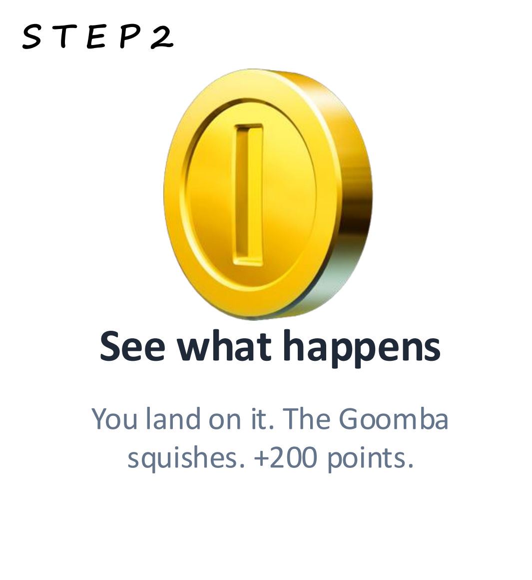
STEP 1



Try something

A Goomba is walking toward you. You panic and press jump.

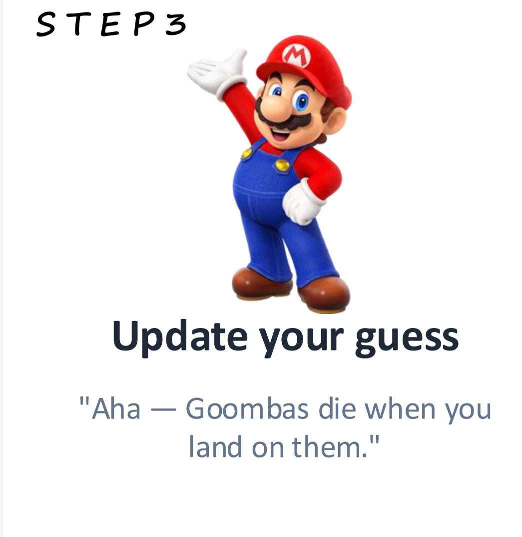
STEP 2



See what happens

You land on it. The Goomba squishes. +200 points.

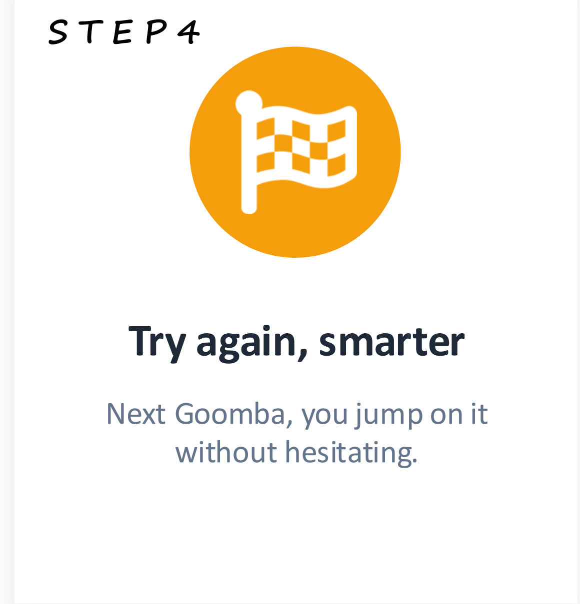
STEP 3



Update your guess

"Aha — Goombas die when you land on them."

STEP 4



Try again, smarter

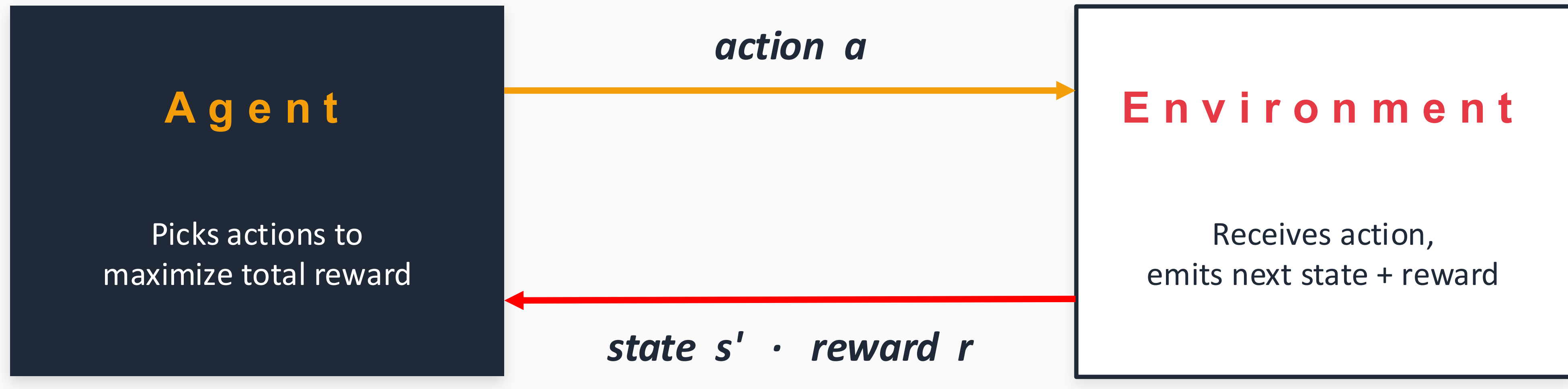
Next Goomba, you jump on it without hesitating.

That loop is reinforcement learning.

The player is the **agent**. Mario's world is the **environment**. Points are the **reward**. "Jump on Goombas, grab coins" is the **policy**.

What is Reinforcement Learning?

An agent learns by interacting with an environment and observing rewards.



Key idea

No labels, no supervisor. The agent must figure out which actions are good by trying them and observing what happens.

The same framework, very different problems

Domain	State	Action	Reward
Chess	Board position	A legal move	+1 win, −1 lose
Robot walking	Joint angles, velocities	Torques on motors	Forward velocity
Recommender	User history	Item to show next	Did the user click?
Today's lab	Position in the maze	Up · Right · Down · Left	+10 at goal, −0.1 per step

→ *We'll work with the bottom row today, but the algorithm we write today applies to all of these.*

The vocabulary

Six terms — and that's everything we need to start coding.

State s

What the agent observes — e.g. its (row, col) in the maze.

Action a

What the agent does — Up / Right / Down / Left.

Reward r

Scalar feedback after each action. +10 at goal, −0.1 per step.

Episode

One run from start until the task ends (reach the goal).

Return

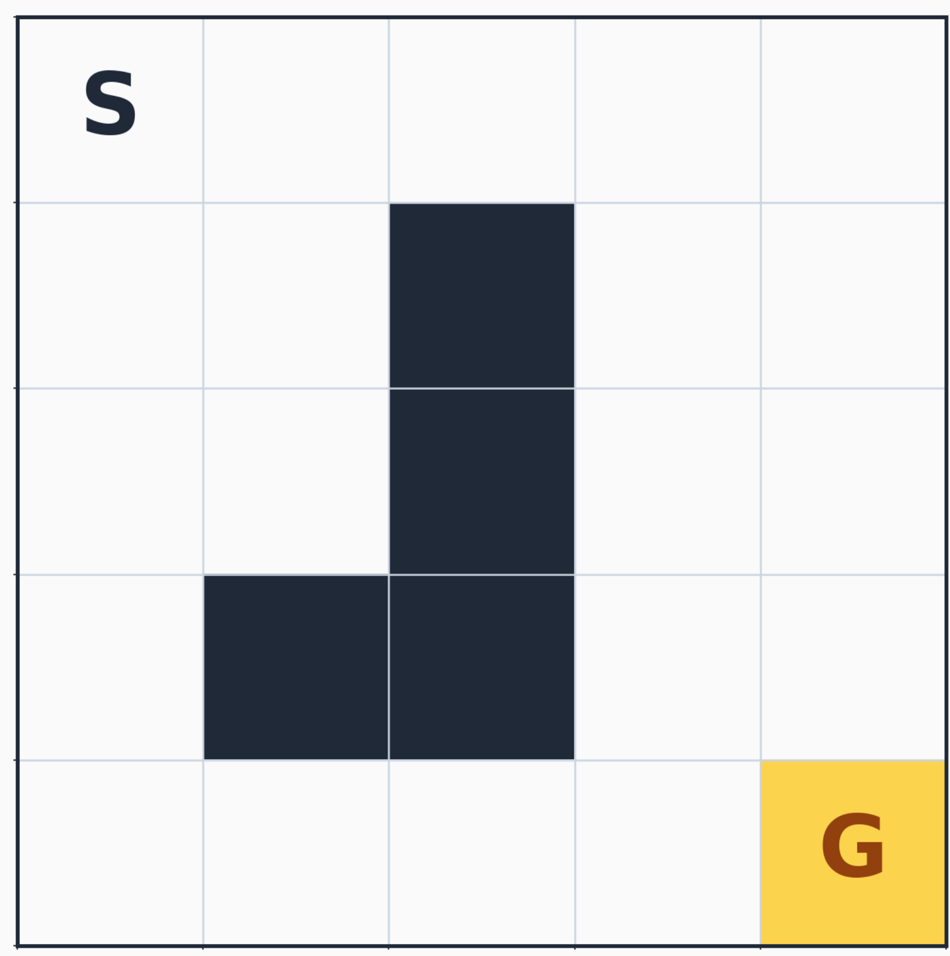
Total reward in one episode. The thing we want to maximize.

Policy $\pi(s)$

The agent's strategy — a rule that maps states to actions.

Our maze

5 × 5 grid · 4 actions · deterministic transitions



- **Start (S)**

  top-left, position (0, 0)
- **Goal (G)**

  bottom-right, reward +10, ends episode
- **Walls**

  dark cells — agent stays put if it tries to enter
- **Empty cell**

  step penalty −0.1 (encourages short paths)
- **Actions**

  0 = Up · 1 = Right · 2 = Down · 3 = Left

PART 2

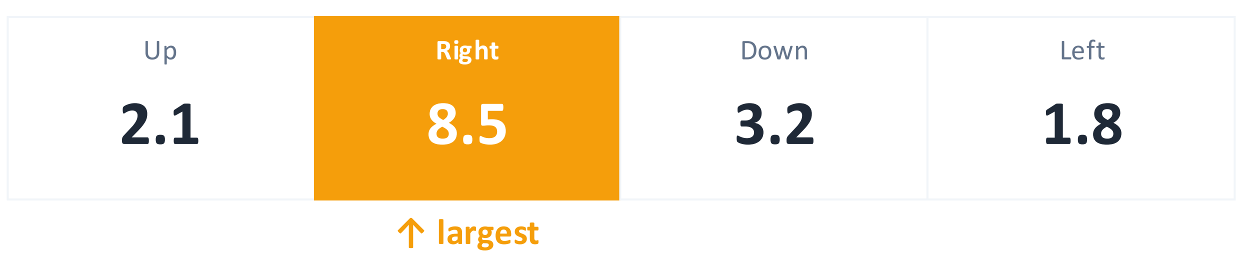
Q-learning, from scratch

How the agent assigns numbers to (state, action) pairs and updates them from experience.

The Q-table

For every **(state, action)** pair, store a number **$Q(s, a)$** that estimates how much total reward the agent will collect from then on.

Once we know Q, the optimal policy is simple — for each state, pick the action with the largest Q-value.



↑ largest

→ *optimal action in state s is Right*

IN ONE LINE

$$\pi^*(s) = \arg\max_a Q(s, a)$$

Q is just a 5 × 5 × 4 array of numbers — one entry per (state, action) pair. We initialize it to all zeros.

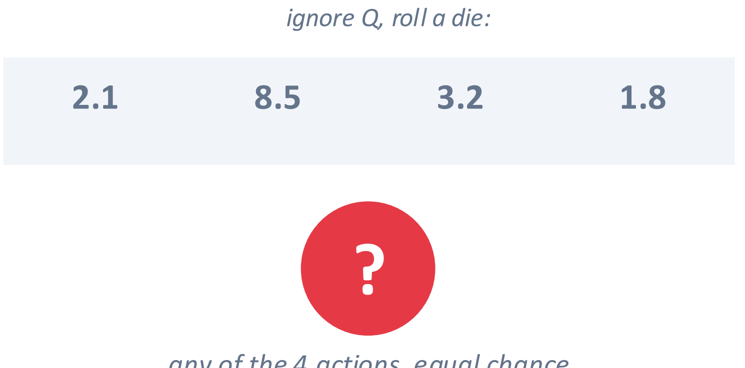
Exploration vs. exploitation

If we always pick the largest Q-value, we'll never try anything new. We need to mix in random actions.

EXPLORE

with probability ε

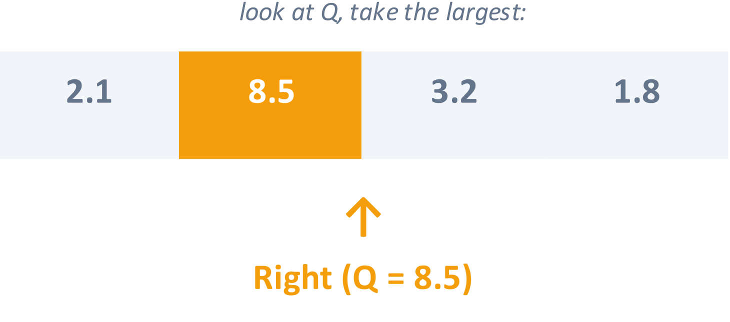
Pick a random action, regardless of Q. This lets the agent discover paths it has never tried.



EXPLOIT

with probability 1 − ε

Pick the action with the largest Q-value. This uses what the agent already knows.



PRACTICAL RECIPE

Start ε ≈ 1.0 (mostly random), decay toward ε ≈ 0.05 (mostly greedy) over training.

How does Q learn?

After every step, the agent has new information. Use it to update one number in the table.

THE IDEA

Take action ***a*** in state ***s***, land in ***s'*** with reward ***r***. Then nudge ***Q(s, a)*** toward what we just observed plus our best guess of the future.

THE UPDATE RULE

$$Q(s,a) \leftarrow Q(s,a) + \alpha \cdot \left[r + \gamma \cdot \max_{a'} Q(s', a') - Q(s,a) \right]$$

`Q(s, a)`

our current estimate

what we wrote in the table last time

`r`

the reward we just got

fresh information from this step

`γ · max Q(s', a')`

best future from s'

look at next state, take the max

α

step size

learning rate — how big a step we take

If the episode just ended, there's no s' — just use target = r.

A toy jupyter notebook for Q-Learning on Colab

PART 3

Let's write the code

Switch to the notebook. Three short functions, then we run training.

Two functions to write

A small helper is provided in the notebook. We'll write the two interesting parts together.

Provided for you: `epsilon_greedy(Q, state, ε)` — flips a coin to either pick a random action or argmax over Q.

1

`q_update`

~15 min

`(Q, s, a, r, s', done, α, γ)  →  updates Q in place`

The heart of the algorithm. Compute the target, the error, and nudge Q. We'll spend most of our time here.

2

`train`

~10 min

`(env, n_episodes)  →  trained Q, list of returns`

Loop over episodes: reset → step → update → repeat. The skeleton is filled in — you write the inner few lines.

Each TODO has a hidden solution cell in the notebook — keep collapsed during live-coding.

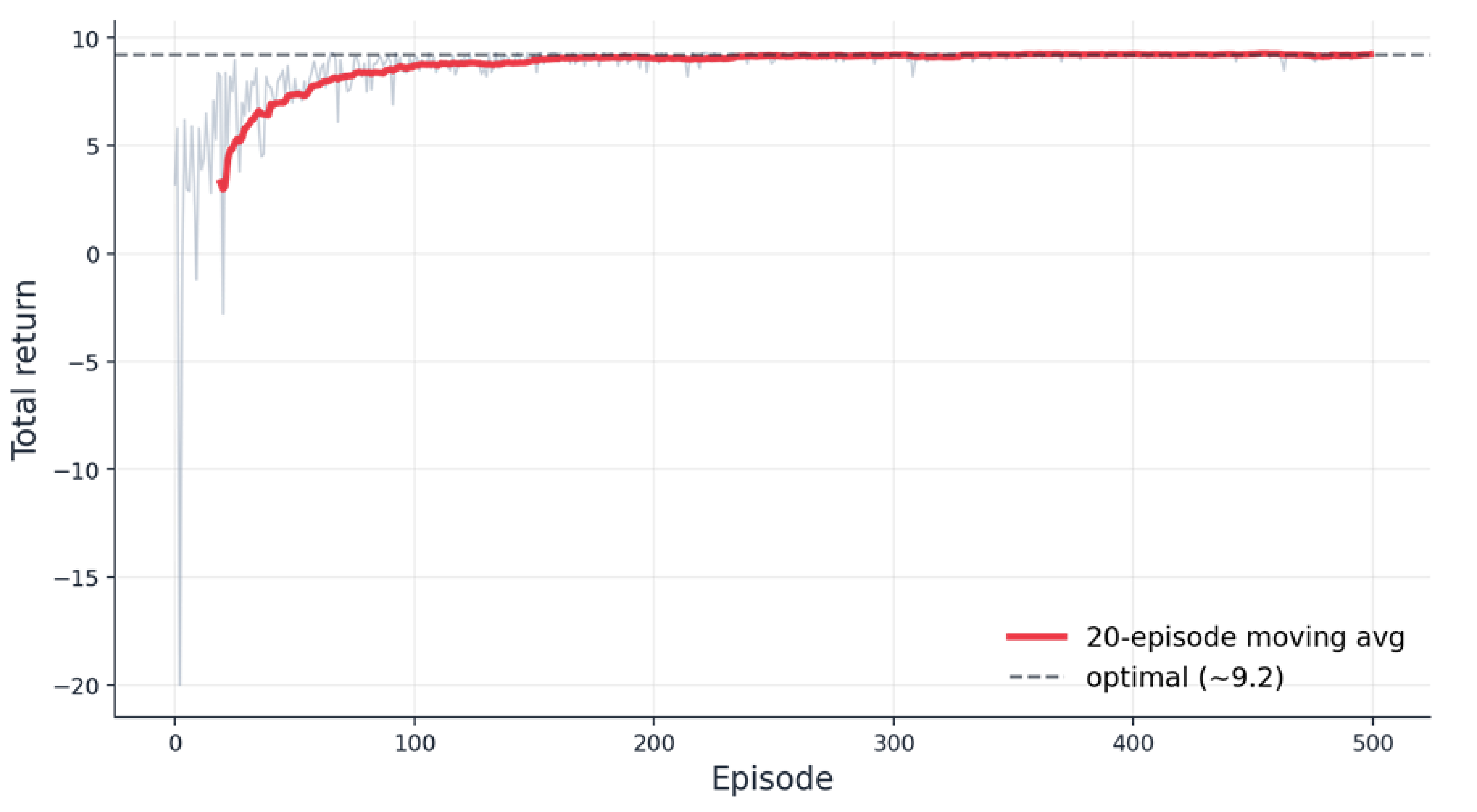
PART 4

Did it work?

Run training and see what the agent learned.

It learns!

500 episodes. Return climbs from −20-ish to ~9.2 (the optimal 8-step return).



Early episodes

Agent bumbles randomly. Many timeouts.

Sharp climb (~ep 100)

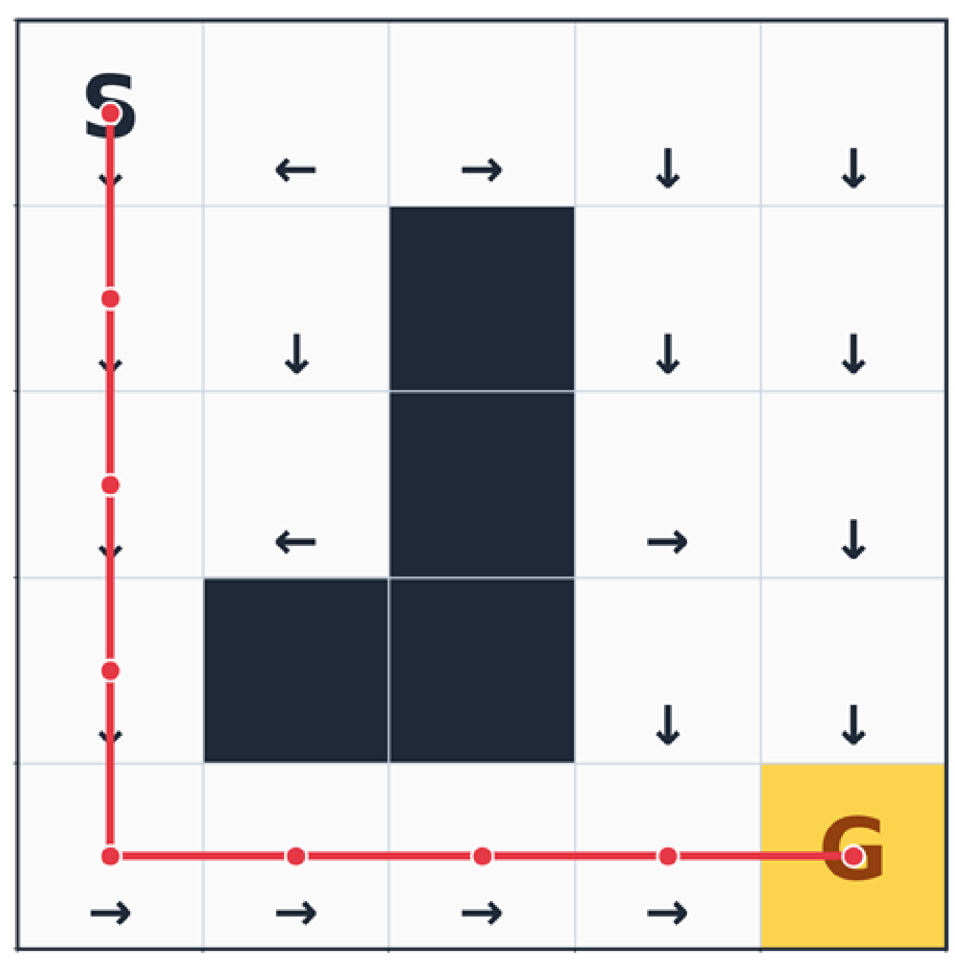
Agent finds the goal more reliably.

Plateau

Stable near optimal. Time to stop.

The learned policy

Arrows show argmax_a Q[s, a]. The red line is the greedy rollout from start to goal.



What to notice

- **Coherent flow**
  Arrows form a path toward the goal — they didn't 5 minutes ago.
- **Walls respected**
  The agent learned which actions don't help, even though we never told it about walls.
- **Optimal path: 8 steps**
  Matches the Manhattan distance lower bound. Return = 10 − 0.1·8 = 9.2.
- **Some arrows are arbitrary**
  Cells the agent rarely visits don't have meaningful Q-values. That's fine.

Three knobs that change everything

Learning rate

TOO SMALL

Updates are tiny. Q changes too slowly.

TOO LARGE

Updates overshoot. Q's learning is not steady

Today: 0.1

Discount factor

TOO SMALL

Agent is myopic. Only cares about next step.

TOO LARGE

$\gamma \geq 1 \rightarrow$ returns can diverge.

Today: 0.99

Exploration rate

TOO SMALL

Stuck in first decent path. Misses shortcuts.

TOO LARGE

Keeps wandering randomly even when trained.

Today: 1.0 → 0.05

Today's update rule is the seed of every modern RL algorithm.

VALUE-BASED

DQN

Replace the Q-table with a neural network $Q_\theta(s, a)$. Train it by minimizing the same TD error we wrote today, just with squared loss and gradients.

Mnih et al., 2015 — Atari from pixels.

POLICY-BASED

PPO · GRPO

Skip Q entirely — parameterize the policy $\pi_\theta(a|s)$ and do gradient ascent on expected return. Same agent–environment loop, different update.

The family used to fine-tune LLMs with RLHF.

Recommended next: Sutton & Barto (chapters 1-6) · OpenAI Spinning Up in Deep RL

Lab 12

Reinforcement Learning

DS-GA 1003 | Machine Learning | Spring 2026

2026.04.30 Presenter by Yihuai Hong