Q-LEARNING IN A SIMPLE GRIDWORLD

Authors: Andela Cvetinovic
Student ID: SM3800009
Course: MSc Introduction to AI & ML
Institution: University of Trieste, DSAI
Contact: s239780@ds.units.it

Project Motivation

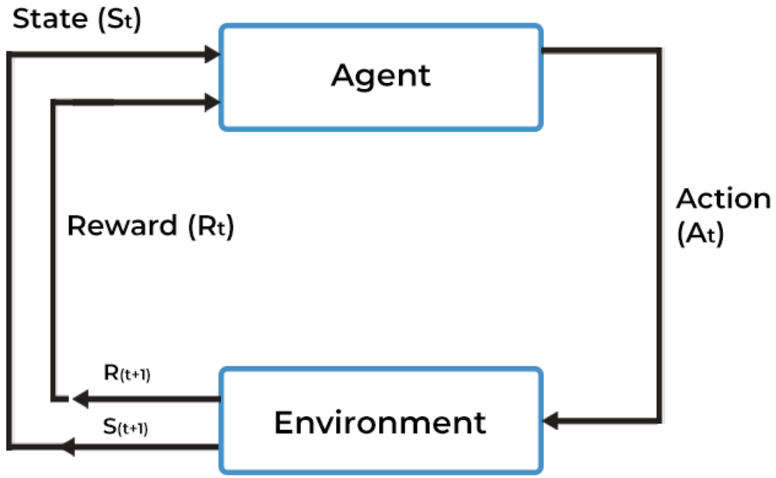
Q-learning enables agents to learn optimal decision-making policies through trial and error in discrete state environments like gridworlds. This project demonstrates hands-on RL mastery by implementing Q-learning in a controlled gridworld scenario.

Problem Definition

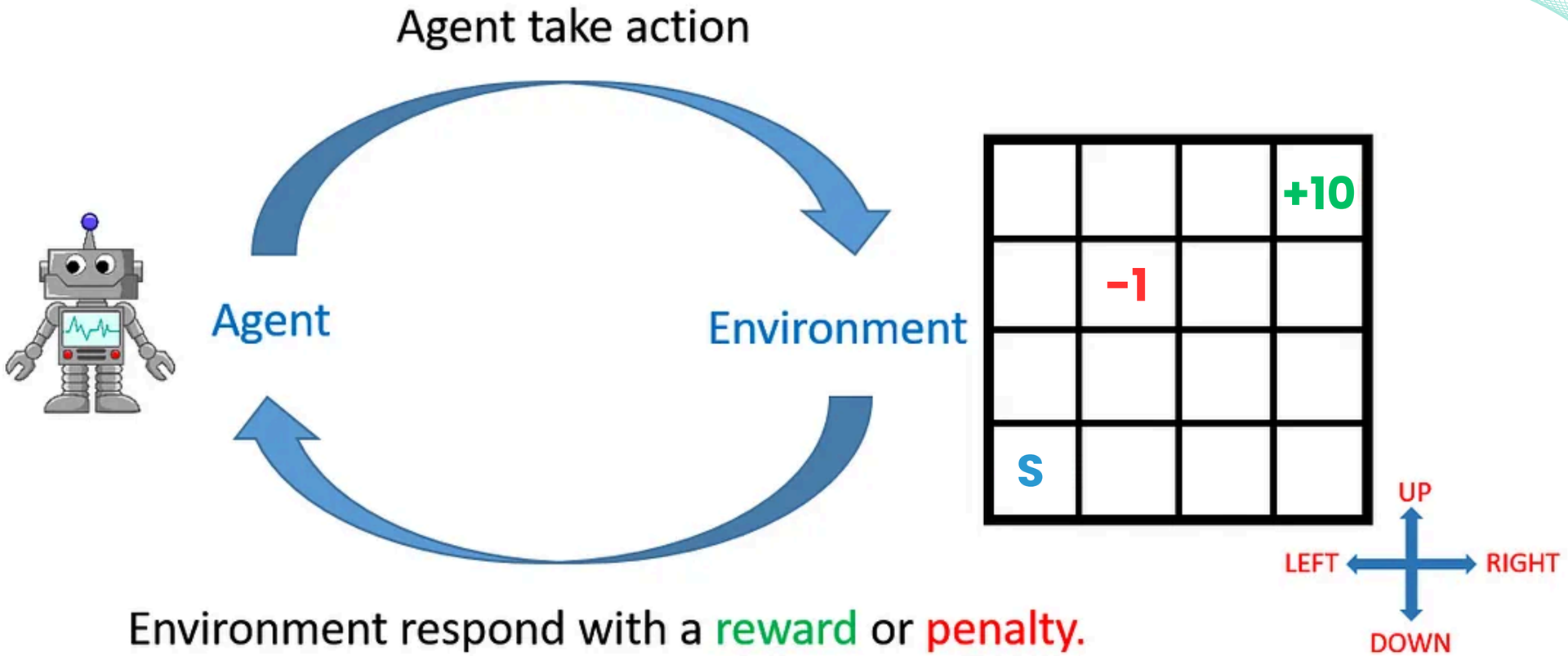
Navigate a 4x4 grid from start to goal, maximizing cumulative reward while avoiding obstacles.

RL approach

- Learn the optimal policy π* directly from experience with minimal environmental assumptions
- Exploration vs. exploitation as a challenge → balance discovering new paths (exploration) with using known rewarding actions (exploitation)
- Formalize as MDP



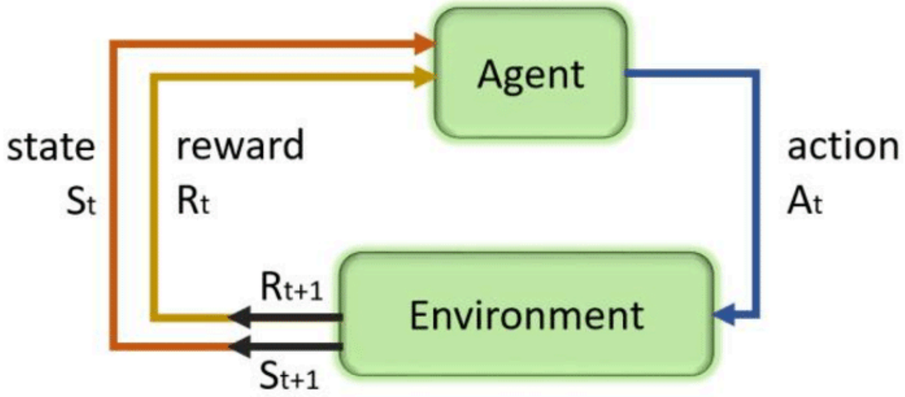
Project illustration



Environment respond with a reward or penalty.

Agent may change state due to chosen action.

Formalizing as MDP



State

agent's positions
on 4x4 grid

Actions

Q-Table (4,4,4)

Reward function

+10 reward for goal state (3,0)
-1 reward for wall state (1,1)
0 otheriwise, all other moves

Transition function

deterministic movement

No randomness or uncertainty in transitions: the same action from the same state always leads to the same (next) state.

grid bounds

The agent cannot "fall off" the grid - attempts just leave it at the edge position.

ENVIRONMENT DEFINITION

Gridworld

State space: 4x4 grid

Grid has 4 rows and 4 columns, resulting in 16 different states, represented as a tuple of coordinates (row-column)

- positions: goal at (0,3), wall at (1,1), and start at (3,0)
- 2 terminal states: goal & wall
- boundaries, agent can’t fall off the grid

Rewards

+10 (goal), -1 (wall), 0 (otherwise)

```python
class GridWorld:
    def __init__(self):
        self.grid = np.array([
            [0, 0, 0, 1],       # Goal at (0, 3) with value 1
            [0, -1, 0, 0],      # Wall at (1, 1) with value -1
            [0, 0, 0, 0],
            [0, 0, 0, 0]
        ])
        self.start_state = (3, 0)
        self.goal_state = (0, 3)
        self.wall_state = (1, 1)
        self.state = self.start_state
        self.action_names = ['Up', 'Right', 'Down', 'Left']
```

```python
def step(self, action):
    next_state = self.get_next_state(self.state, action)
    if next_state == self.goal_state:
        reward = 10
    elif next_state == self.wall_state:
        reward = -1
    else:
        reward = 0

    self.state = next_state
    done = self.is_terminal(next_state)
    return next_state, reward, done
```

AGENT DEFINITION

Q-learning agent

Q-table: 4x4x4

Stores Q-values for each state-action pair
(that define *how good* each possible move is in every cell of the grid).

Parameters

- Learning rate **α** - controls how much new information overrides old (0-1)
- Discount factor **γ** - importance of future rewards (0-1)
- Exploration rate **ε** - probability of random action for exploration (0-1)
- Epsilon decay - gradually reduces exploration over time

Learning function

Learning function is here *Q-learning update rule*. It allows the agent to iteratively improve its estimates of the optimal Q-values through experience interacting with the environment, learning to choose actions that *maximize long-term reward*.

```python
class QLearningAgent:
    def __init__(self, learning_rate=0.1, discount_factor=0.9, exploration_rate=1.0):
        self.q_table = np.zeros((4, 4, 4))
        self.learning_rate = learning_rate
        self.discount_factor = discount_factor
        self.exploration_rate = exploration_rate
        self.min_epsilon = 0.01
        self.decay_factor = 0.995
        self.action_names = ['Up', 'Right', 'Down', 'Left']
```

```python
# Extract policy (action indexes 0-3)
# "What action should I take?
    def get_policy(self):
        policy = np.zeros((4, 4), dtype=int)
        for i in range(4):
            for j in range(4):
                policy[i, j] = np.argmax(self.q_table[i, j])
        return policy

# Extract max Q-value for each grid position:
# "How good is this state/how much reward can I expect?"
    def get_value_function(self):
        values = np.zeros((4, 4))
        for i in range(4):
            for j in range(4):
                values[i, j] = np.max(self.q_table[i, j])
        return values
```

Learning function: **Q-function**

Each **Q-table score** is expected future reward that the agent receives if it takes that *action* at that *state*.

To learn each value of Q-table, we use Q-learning algorithm.

Q-function uses the *Bellman equation* and takes 2 inputs: **state (s) & action (a)**

Using **Q-function**, we **get the values of Q** for the cells in the table.
When we start, all the values in the Q-table are zeros. This is an *iterative* process of updating the values of the Q-table.

As we start to explore the environment, the Q-function gives us *better & better approximations* by continuously updating the Q-values in the table.
It tells how good it is to pick this action from this state if we continue to act according to π.

Q-function

for calculating each Q-value

$$Q^{\pi}(s_t, a_t) = E[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots | s_t, a_t]$$

- $Q^{\pi}(s_t, a_t)$: Q-Values for the state given a particular state
- $E[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots]$: Expected discounted cumulative reward
- s_t, a_t: Given the state and action

Algorithm: **Q-learning**

INTUITION

Updating knowledge (Q-table; state-action values) by experience: allows the agent to learn how to act in a given environment by trying actions, receiving rewards & adjusting behaviour to maximize future rewards.

Q-LEARNING STEP-BY-STEP

1. Initialize

- Set all Q-values to zero

2. For each episode:

a) Start at initial state

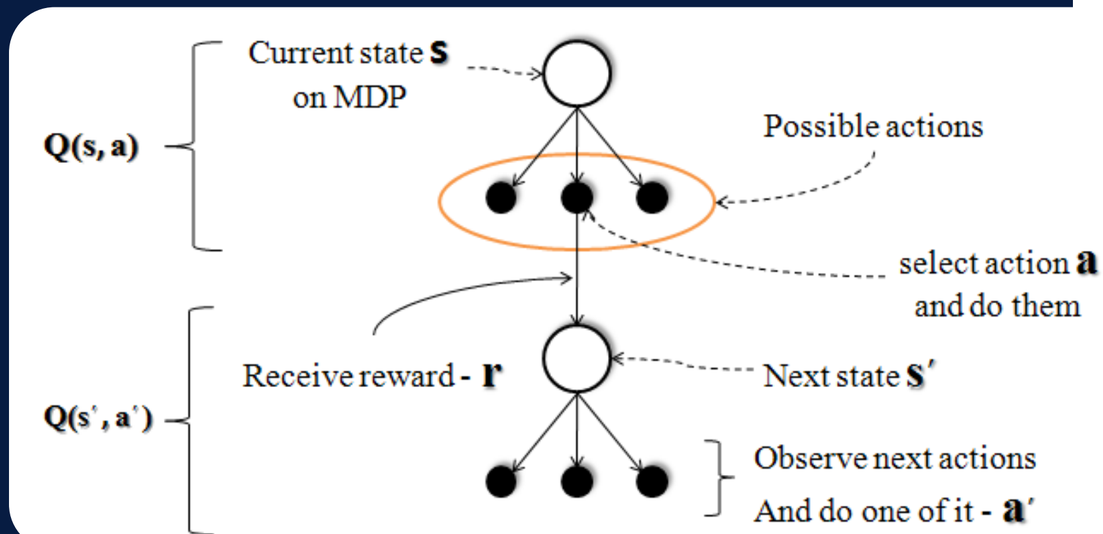
b) While not at goal:

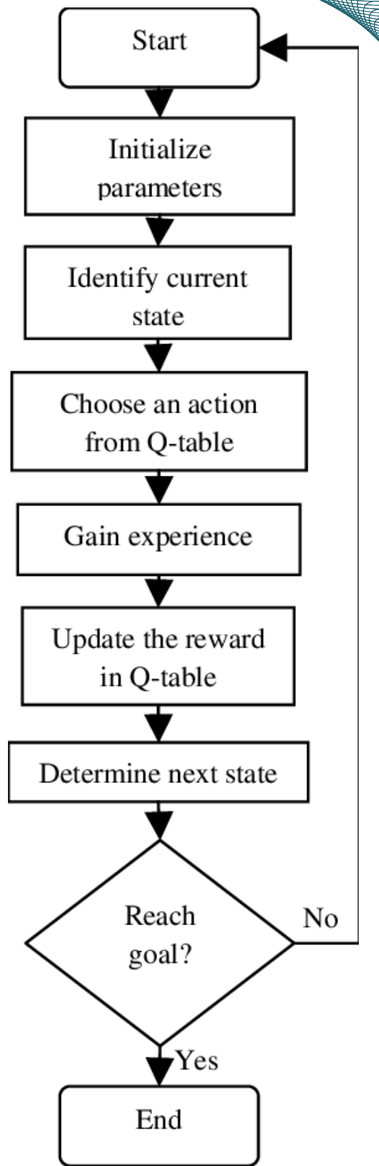
- Choose action (epsilon-greedy)
- Take action, observe reward & next state
- Update Q-table

c) Track episode metrics

3. Repeat for 1000 episodes

4. Result converged Q-table with optimal policy





Algorithm: **Q-learning**

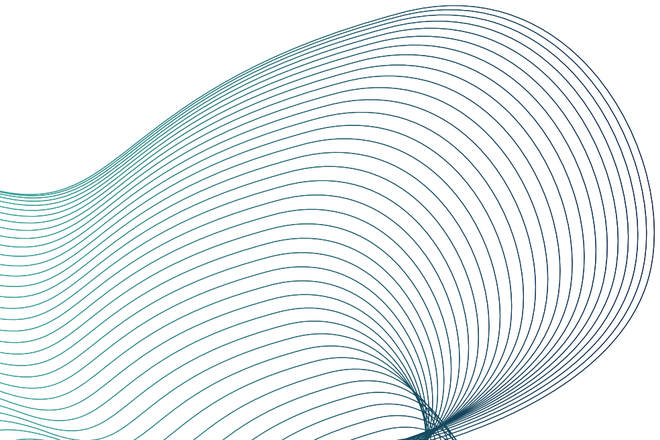
Bootstrapping

Here, Q(s,a) is *bootstrapped* → it uses the **current estimate** max_aQ(s',a') (maximum Q-value of the next state s' over all possible actions a') rather than waiting for the episode to finish and using the total reward, allowing Q-learning to learn from *incomplete* episodes.

Off-policy

It aims for the optimal path, regardless of the actual actions taken along the way. SARSA, by contrast, is on-policy: it learns values following the current exploration pattern.

UPDATE RULE



$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma max_a Q(S_{t+1}, a) - Q(S_t, A_t)]$$

New Q-value estimation

Former Q-value estimation

Learning Rate

Immediate Reward

Discounted Estimate optimal Q-value of next state

Former Q-value estimation

TD Target

TD Error

Training function: **Q-learning training loop**

Main purpose:
Train an agent through repeated episodes in an environment until it *learns optimal behavior*

The training process:

- Agent starts at initial state each episode
- Chooses actions using epsilon-greedy strategy
- Receives rewards and transitions to new states
- Updates Q-table after each step
- Repeats until reaching goal or wall

Tracking progress:

- Total rewards per episode
- Number of steps taken
- TD errors (prediction accuracy)
- Convergence detection

```
def train_agent(env, agent, episodes=1000, verbose=False, log_interval=100):
```

Training function: **Parameters**

Environment setup:

- **GridWorld**: 4×4 grid environment
- **Agent**: Q-Learning algorithm
- **Episodes**: 1000 training iterations

```
rewards, steps, epsilons, td_errors, metrics = train_agent(env, agent, episodes=1000, verbose=True, log_interval=100)
```

Hyperparameters:

- **α = 0.1** (lower, more stable learning)
- **γ = 0.9** (future rewards are valued at 90% of immediate rewards)
- **ε = 1.0 → 0.1** (exploration rate - probability of taking a random action)
- Decays **from 100%** exploration **to 10%** over training
- Decay factor: **0.995 per episode**

```
env = GridWorld()
agent = QLearningAgent(
    learning_rate=0.1,
    discount_factor=0.9,
    exploration_rate=1.0)
```

Action selection strategy: **Epsilon-Greedy**

How agent chooses actions?

EXPLOIT - most of the time

- Pick the **best known action** from Q-table
- Uses learned knowledge

EXPLORE - ε% of the time

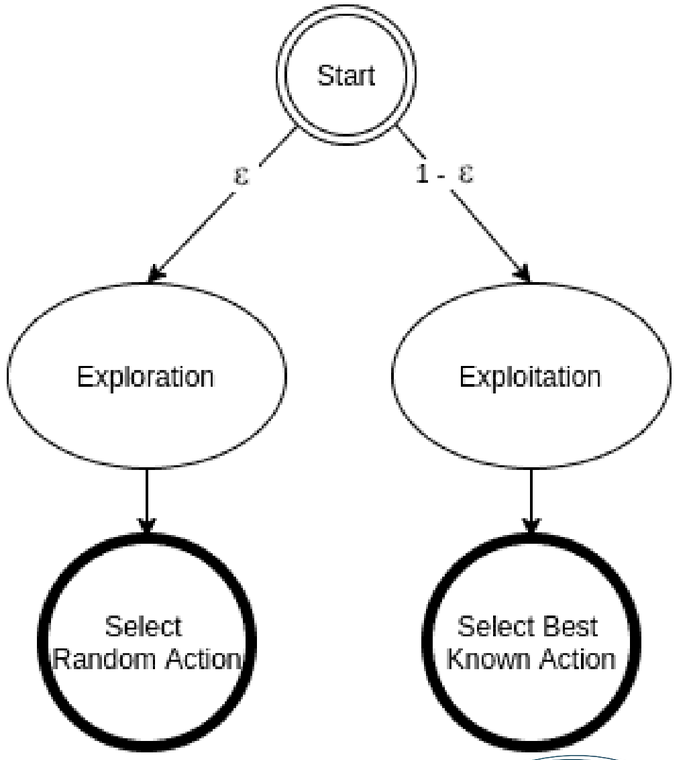
- Pick a **random action**
- Discovers new possibilities

Exploration decay:

- **exploration_rate = max(min_ε, current_ε × decay_factor)**

Why this matters?

- Early training: more exploration (ε = 1.0) to discover environment
- Late training: more exploitation (ε = 0.1) to use learned policy
- Balances learning new strategies vs. using known good ones



Q-Learning training

```
Episode 100/1000 | Avg Reward: 3.51 | Avg Steps: 13.78 | Avg TD Error: 0.7098
Episode 200/1000 | Avg Reward: 7.91 | Avg Steps: 10.02 | Avg TD Error: 1.0321
Episode 300/1000 | Avg Reward: 9.12 | Avg Steps: 8.05 | Avg TD Error: 0.2870
Episode 400/1000 | Avg Reward: 9.23 | Avg Steps: 7.10 | Avg TD Error: 0.1148
Episode 500/1000 | Avg Reward: 9.23 | Avg Steps: 6.35 | Avg TD Error: 0.0368
Episode 600/1000 | Avg Reward: 10.00 | Avg Steps: 6.19 | Avg TD Error: 0.0211
Episode 700/1000 | Avg Reward: 9.89 | Avg Steps: 6.20 | Avg TD Error: 0.0042
Episode 800/1000 | Avg Reward: 10.00 | Avg Steps: 6.09 | Avg TD Error: 0.0011
Episode 900/1000 | Avg Reward: 10.00 | Avg Steps: 6.05 | Avg TD Error: 0.0021
Episode 1000/1000 | Avg Reward: 10.00 | Avg Steps: 6.03 | Avg TD Error: 0.0050
```

- Early in training, the agent performs poorly (lower rewards, more steps, higher TD error), as expected with little prior knowledge
- As learning progresses, the agent rapidly improves, converging to optimal performance after about 600 episodes
- The final output (constant average reward, minimum steps, low TD error) demonstrates that Q-Learning has successfully converged and the agent is now following an optimal policy
- The stable metrics at the end confirm that learning has plateaued, and further episodes provide little change

Results

```
TRAINING SUMMARY
============================================================
  Total episodes: 1000
  Optimal policy found: Episode 905
  Q-Table converged: Episode 818
  Final epsilon: 0.0100
  Final avg reward (last 50 ep): 10.00
  Final avg steps (last 50 ep): 6.06
  Final avg TD error (last 50 ep): 0.001807
```

Testing learned policy (5)

```
Test episode 1:
Start: (3, 0)
Path: (3, 0) → (3, 1) → (3, 2) → (2, 2) → (1, 2) → (1, 3) → (0, 3)
Steps: 6, Total reward: 10

Test episode 2:
Start: (3, 0)
Path: (3, 0) → (3, 1) → (3, 2) → (2, 2) → (1, 2) → (1, 3) → (0, 3)
Steps: 6, Total reward: 10
```

Test result confirms Q-learning agent has learned to navigate the gridworld perfectly:

- **Same path -** every test run follows the same path (repeatable due to deterministic environment)
- **Shortest path** - always 6 steps
- **Max reward** - always 10 (agent reaches the goal without getting penalties)

Results: **Best Value & Best Action at each state**

- **V(s) = max_a Q(s,a) State Value Function** - estimate of expected reward starting from each cell, based on having learned the optimal way to get to the goal; computed by ***selecting the largest Q-value*** at every position after the agent finishes learning.

argmaxQ(s,a): BEST ACTION AT EACH STATE →

```
State Values (Max Q-value per state):
========================================
  1.68    5.25    9.88    0.00
  0.39    0.00    9.00   10.00
  6.03    7.28    8.10    9.00
  5.90    6.56    7.29    7.92
```

```python
def get_value_function(self):
    values = np.zeros((4, 4))
    for i in range(4):
        for j in range(4):
            values[i, j] = np.max(self.q_table[i, j])
    return values
```

The greedy policy grid shows clear optimal arrows toward the goal. Avoiding the wall (X) is evident - careful policy learning.

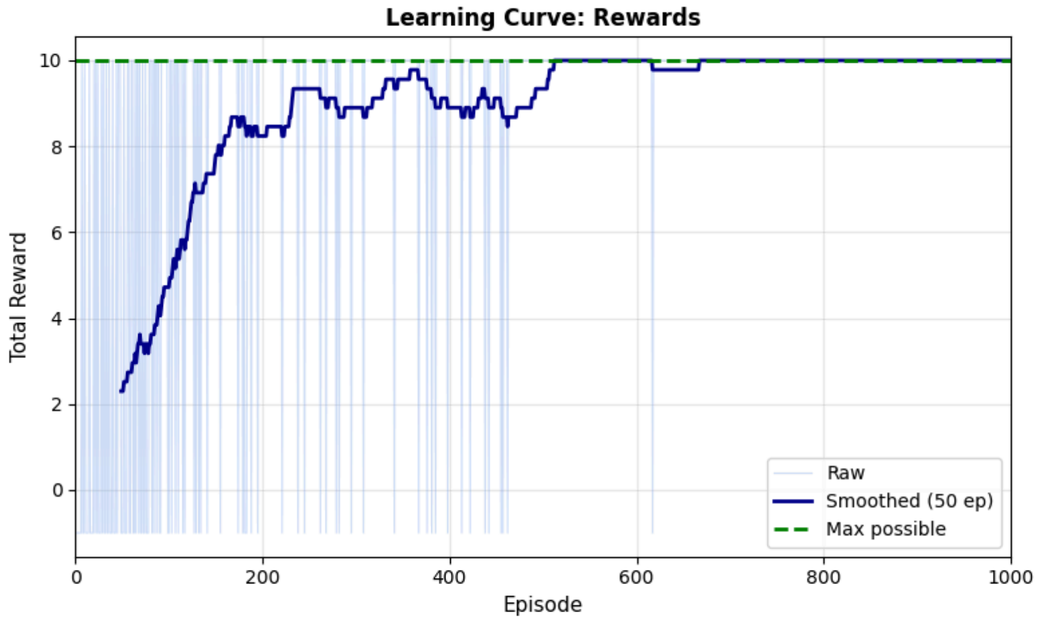
```
Learned policy:
========================================
+-------+-------+-------+-------+
|   →   |   →   |   →   |   G   |
+-------+-------+-------+-------+
|   ↑   |   X   |   →   |   ↑   |
+-------+-------+-------+-------+
|   →   |   →   |   ↑   |   ↑   |
+-------+-------+-------+-------+
|   →   |   →   |   ↑   |   ↑   |
+-------+-------+-------+-------+
```

```python
def get_policy(self):
    policy = np.zeros((4, 4), dtype=int)
    for i in range(4):
        for j in range(4):
            policy[i, j] = np.argmax(self.q_table[i, j])
    return policy
```

Results: **Learning curves**

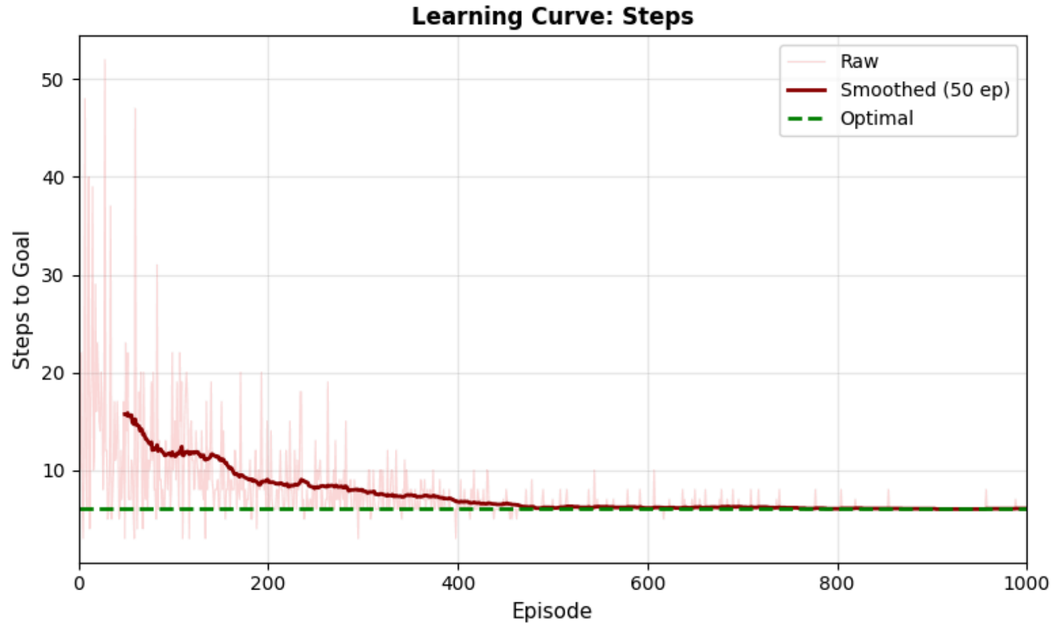
Learning Curve: REWARDS

- The smoothed reward line climbs steadily and reaches the max possible value by around episode 600, then remains stable.
- The total reward per episode increases rapidly in the early episodes, indicating the agent is exploring, learning the environment, and discovering paths to the goal.
- After about 200 episodes, rewards become consistently high and eventually plateau at the maximum possible value (10), demonstrating that the agent has learned the optimal policy and reliably reaches the goal without unnecessary penalties.
- The straight lines in the raw rewards show that, following convergence, the agent's behavior is highly repeatable and optimal.



Learning Curve: STEPS

- Initially, the agent uses many steps per episode → over 10 on average, with high variability.
- As learning progresses, the number of steps needed to reach the goal quickly drops toward the optimal value (6 steps)
- This decrease in steps reflects the agent's improved efficiency, as it discovers and exploits the shortest path.



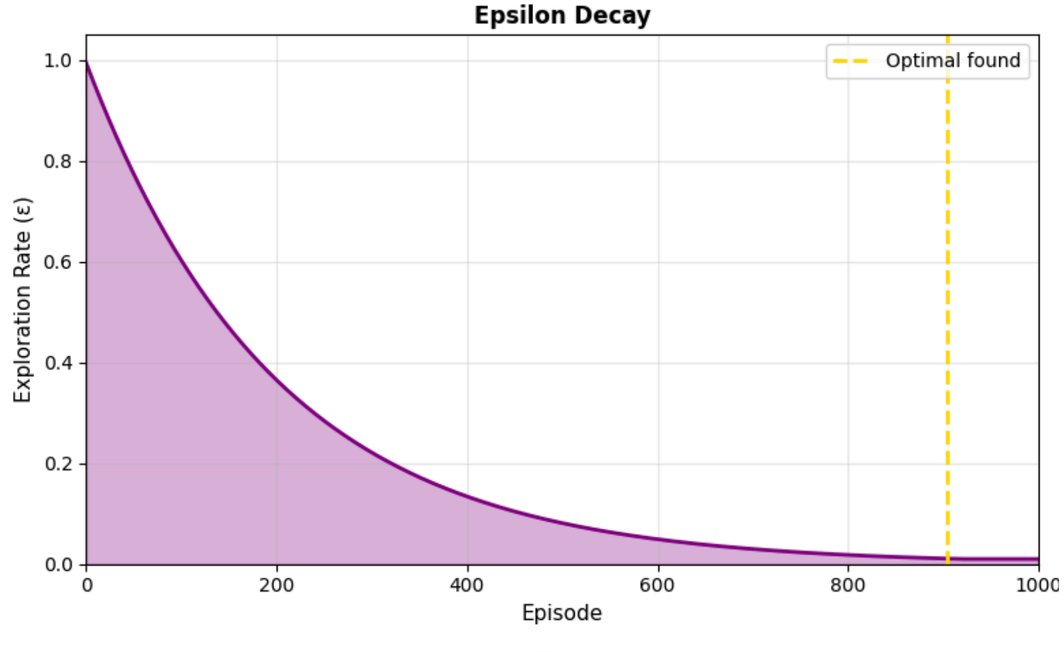
Results: **Decay & TD**

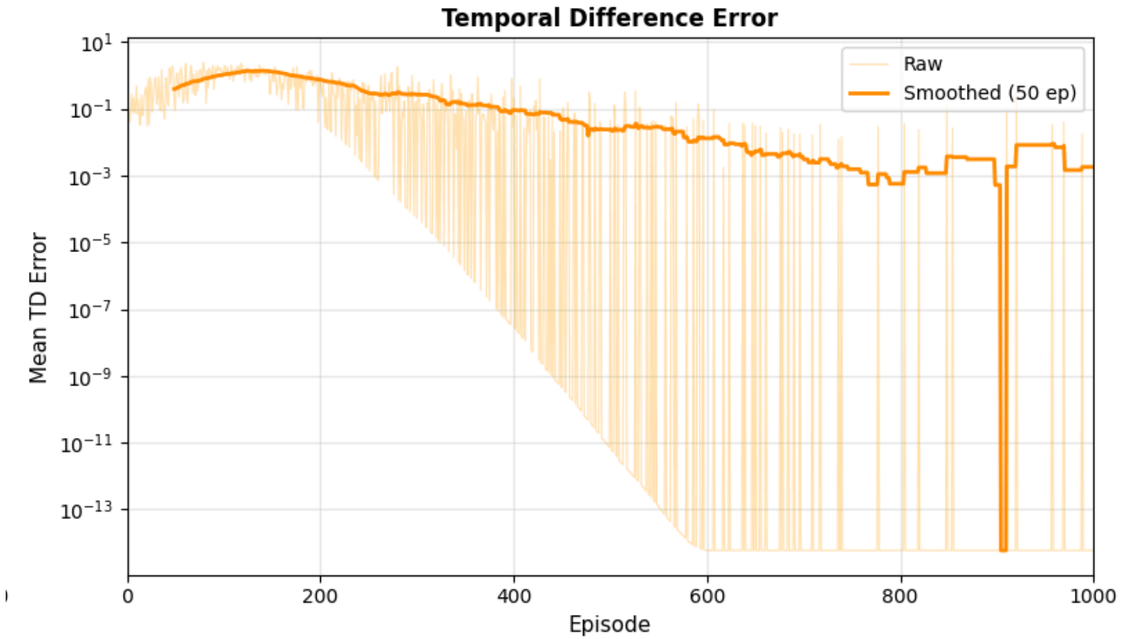
Epsilon decay

- The exploration rate (ε) starts at 1.0, ensuring maximum exploration initially, and decays smoothly toward 0.01
- By the time the optimal policy is found (around episode 600), ε is already quite low, meaning the agent is primarily exploiting what it has learned rather than exploring.
- This progression is ideal and underlines good exploration-exploitation balance.

Temporal difference (TD) error

- The mean TD error starts higher (around 0.1) while the agent is still learning and its predictions are inaccurate.
- Over time, the TD error rapidly decreases, reaching almost zero as the Q-table values converge and the agent's predictions match actual experience.
- This is a clear indicator of convergence and stable learning.

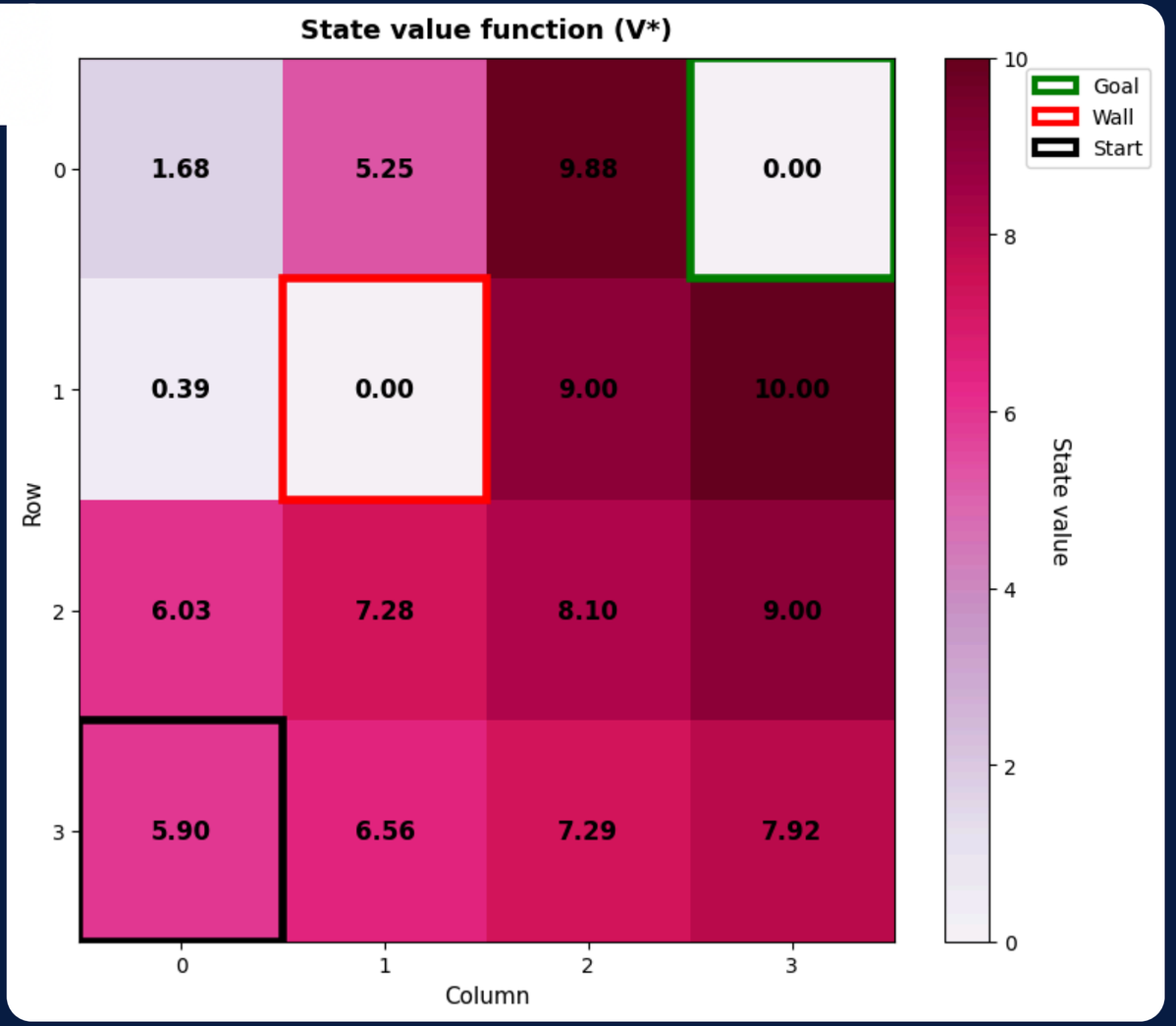




Results: **Best value at each state** *(heatmap)*

$$V_*(s) = \max_a Q_*(s,a)$$

- Heatmap shows how good is to be (state) in each cell in terms of expected future reward if following optimal learned policy
- Highest values near the goal; lowest furthest away
- Goal state has value of 0 → because, once there, no more reward can be accumulated
- Wall value is 0 as well, since it cannot be entered



Results: **Flatten Q-table**

- **Q-table: all 64 state-action pair values Q(s,a)**
- Originally shaped **(4, 4, 4): [rows, columns, actions]**
- Reshaped to 2D array with 16 rows (one for each grid cell: 4×44×4) and 4 columns (1 Q-value per possible action)

```
# Flatten q_table from 3D shape (4,4,4) to 2D array(16,4) in row-major order
q_flat = agent.q_table.reshape(-1, 4)
```

StateIndex	Row	Col	Q_Up	Q_Right	Q_Down	Q_Left	MaxQ	BestAction
0	0	0	0.08	1.68	0.02	0.10	1.68	Right
1	0	1	0.63	5.25	-0.69	0.19	5.25	Right
2	0	2	4.84	9.88	3.20	1.97	9.88	Right
3	0	3	0.00	0.00	0.00	0.00	0.00	Up
4	1	0	0.39	-0.69	0.36	0.02	0.39	Up
5	1	1	0.00	0.00	0.00	0.00	0.00	Up
6	1	2	8.49	9.00	7.05	-0.99	9.00	Right
7	1	3	10.00	8.85	7.88	7.99	10.00	Up
8	2	0	0.10	6.03	0.77	1.31	6.03	Right
9	2	1	-0.97	7.28	3.57	2.01	7.28	Right
10	2	2	8.10	7.91	6.43	5.87	8.10	Up
11	2	3	9.00	5.85	2.19	3.40	9.00	Up
12	3	0	4.19	5.90	5.24	5.21	5.90	Right
13	3	1	6.35	6.56	5.80	5.13	6.56	Right
14	3	2	7.29	6.60	6.33	5.70	7.29	Up
15	3	3	7.92	2.28	2.32	3.89	7.92	Up

Agent's complete "memory" of learned actions after training

Final word 🙂

- The agent learns efficiently, converging to the optimal policy well before 1000 episodes.
- Performance metrics (reward, steps, TD error) show rapid and stable convergence.
- Exploration is phased out at the right time, supporting robust learning early and reliable exploitation later.
- The results are highly interpretable, and the agent behaves exactly as expected for tabular Q-learning in a deterministic, fully observable gridworld

Theoretical: **Why Q-Learning over SARSA**

My goal was to find an optimal path, not the safest, and Q-learning always updates **OPTIMAL** action (γ**max** Q(s',a'))

- Single clear goal → I want the shortest path, not safest
- Deterministic environment → no need for cautious learning
- Small state space → can fully explore quickly
- Offline learning → not performing while learning

Q-Learning Update:

$$Q(s,a) \leftarrow Q(s,a) + \alpha[R + \gamma \cdot \max Q(s',a') - Q(s,a)]$$

↑ Same Q-table ↑ Uses MAX (best action)

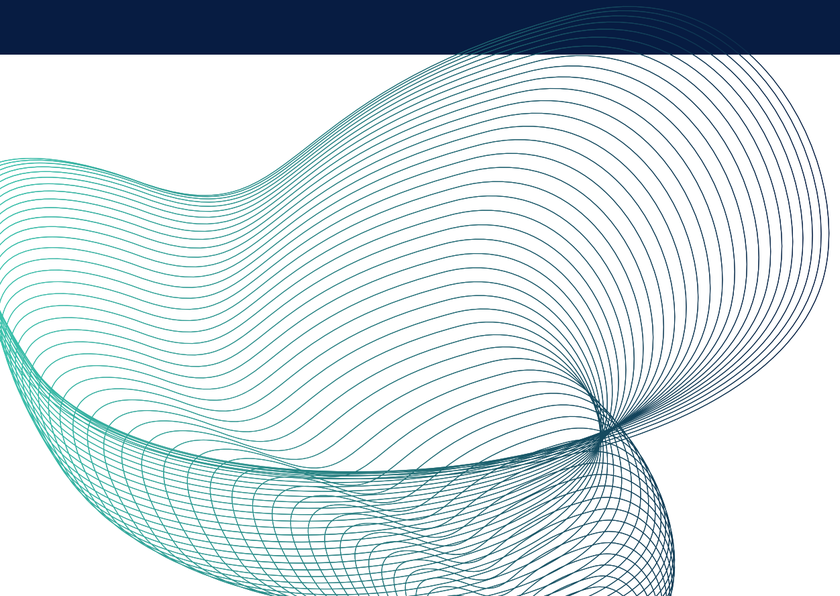
SARSA Update:

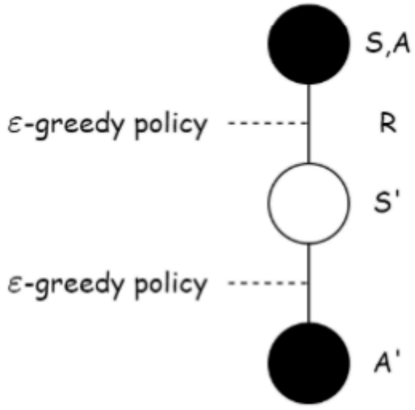
$$Q(s,a) \leftarrow Q(s,a) + \alpha[R + \gamma \cdot Q(s',a') - Q(s,a)]$$

↑ Same Q-table ↑ Uses actual next action

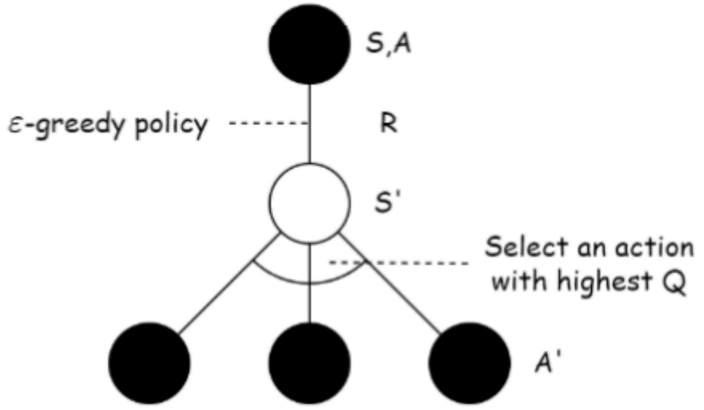
When SARSA might be preferred?

If gridworld included many *risky* paths, and envirnment would be stochastic/noisy.





SARSA



Q-learning

SARSA training & learned policy

```
TRAINING SARSA AGENT
Episode 100/1000 | Avg Reward: 5.49 | Avg Steps: 19.76 | Avg TD Error: 0.5550
Episode 200/1000 | Avg Reward: 8.79 | Avg Steps: 10.35 | Avg TD Error: 1.2102
Episode 300/1000 | Avg Reward: 9.78 | Avg Steps: 8.33 | Avg TD Error: 1.0873
Episode 400/1000 | Avg Reward: 9.89 | Avg Steps: 7.26 | Avg TD Error: 0.7594
Episode 500/1000 | Avg Reward: 10.00 | Avg Steps: 6.74 | Avg TD Error: 0.4978
Episode 600/1000 | Avg Reward: 10.00 | Avg Steps: 6.44 | Avg TD Error: 0.3061
Episode 700/1000 | Avg Reward: 9.89 | Avg Steps: 6.38 | Avg TD Error: 0.2774
Episode 800/1000 | Avg Reward: 10.00 | Avg Steps: 6.11 | Avg TD Error: 0.0952
Episode 900/1000 | Avg Reward: 10.00 | Avg Steps: 6.04 | Avg TD Error: 0.0695
Episode 1000/1000 | Avg Reward: 10.00 | Avg Steps: 6.04 | Avg TD Error: 0.0369
```

```
Learned policy:
============================================
+-------+-------+-------+-------+
|   →   |   →   |   →   |   G   |
+-------+-------+-------+-------+
|   ↑   |   X   |   ↑   |   ↑   |
+-------+-------+-------+-------+
|   ↓   |   →   |   →   |   ↑   |
+-------+-------+-------+-------+
|   →   |   →   |   ↑   |   ↑   |
+-------+-------+-------+-------+
```

```
============================================================
SARSA TRAINING SUMMARY
============================================================
  Total Episodes: 1000
  Optimal Policy Found: Episode 897
  Final Epsilon: 0.0100
  Final Avg Reward (last 50 ep): 10.00
  Final Avg Steps (last 50 ep): 6.06
============================================================
```

Practial:
Q-Learning VS SARSA

SARSA wins slightly

- Faster convergence (866 vs 905)
- Lower variance (0.28 vs 0.31)
- Slightly smoother convergence (less spiky)

But...

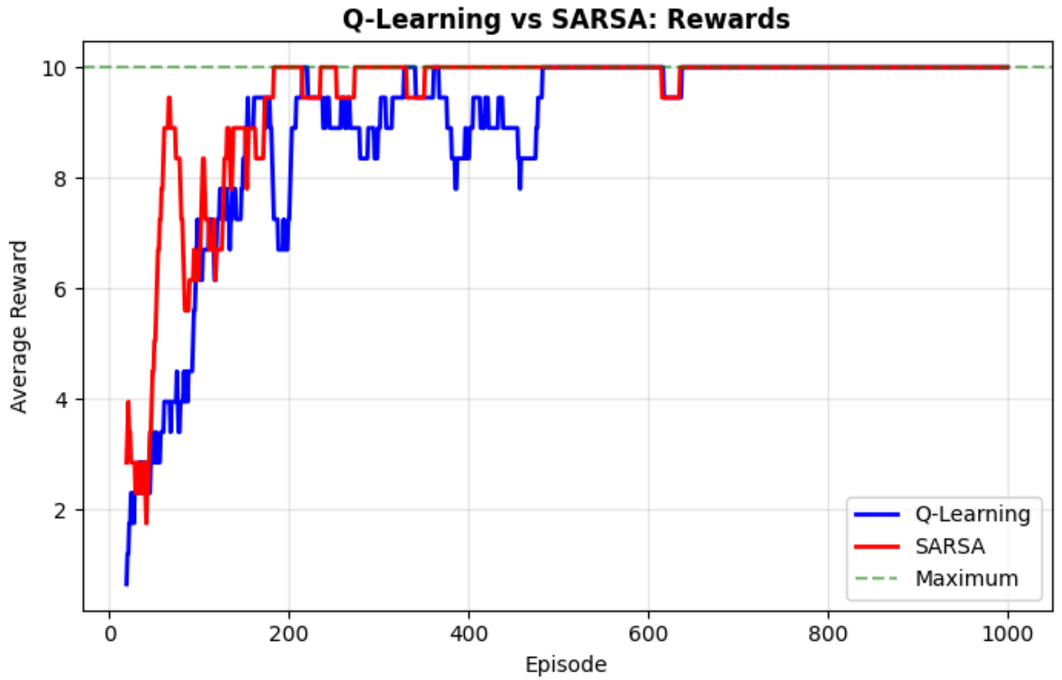
- Both reach maximum reward (10)
- Both achieve optimal path length (~6 steps)

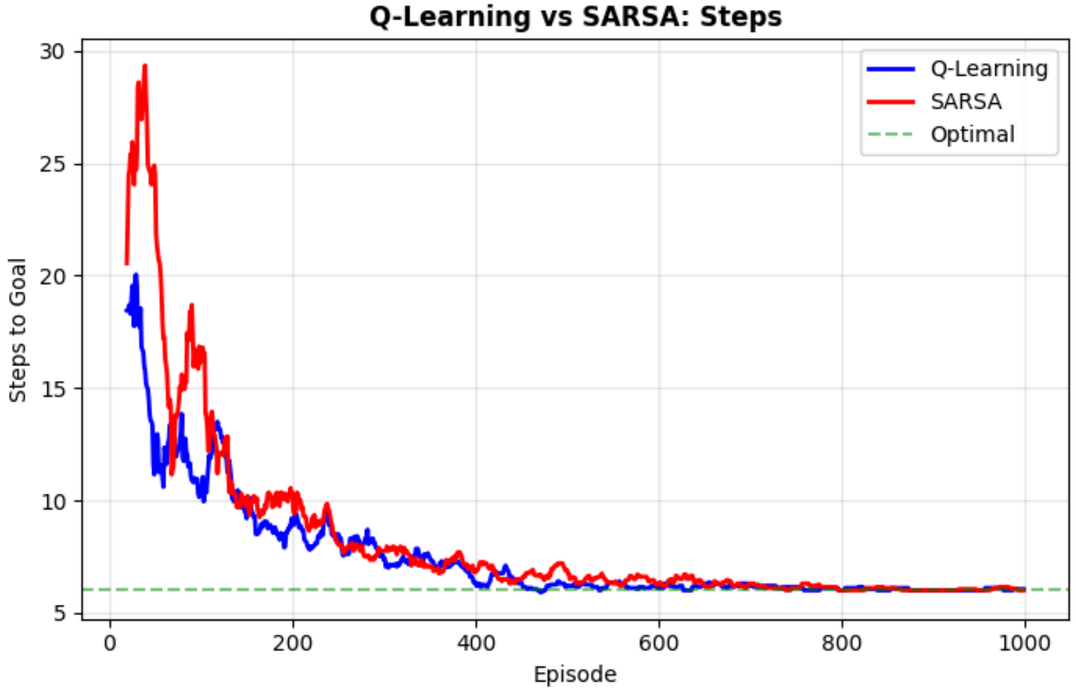
Similar because:

- Safe, deterministic and small environment
- Both use the same greedy policy, ε decays to 0.01

```
Convergence speed:
 Q-Learning: Episode 905
 SARSA: Episode 805

Final Performance (last 50 episodes):
 Q-Learning:
   Reward: 10.00 ± 0.00
   Steps: 6.06 ± 0.31
 SARSA:
   Reward: 10.00 ± 0.00
   Steps: 6.08 ± 0.39
```





Extensions & Limitations

Limitations

- Discrete states - can't handle continuous environments (robot navigation in real space)
- Deterministic transitions -real-world actions often have uncertainty (slippery floors, wind, sensor noise)
- Small state space - tabular methods don't scale to complex environments (millions of states)
- Perfect information - assumes agent knows exact state, which may not be realistic
- Simple reward structure - real-world rewards often sparse, delayed, or multi-objective

Ideas for future work

- Bigger grids
- Stochastic transitions - by adding action noise (e.g., 10% chance of moving in wrong direction)
- Multiple goals - dynamic or multiple objectives requiring hierarchical reinforcement learning
- Continuous state/action spaces