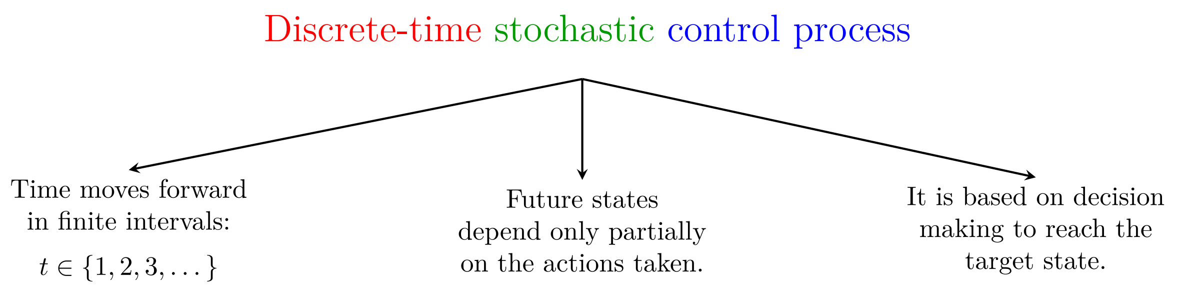
Reinforcement Learning

Reinforcement Learning is the branch of AI focused on solving control tasks, i.e. tasks that we have to perform with right actions in the right situation to get the best possible outcome.

1 Elements of Control Tasks

- State (S_t) : It is the relevant information that describes the situation the task environment is currently in.
- Action (A_t) : They are moves that the player can perform at any moment of time
- Reward (R_t) : It is a numerical value that the agent receives after carrying out an action.
- Agent : It is the entity that will participate in the task by observing its state and carrying out actions at each moment in time
- Environment : It comprises all of the aspects of the task that the agent doesn't control 100%.

2 Markov Decision Process

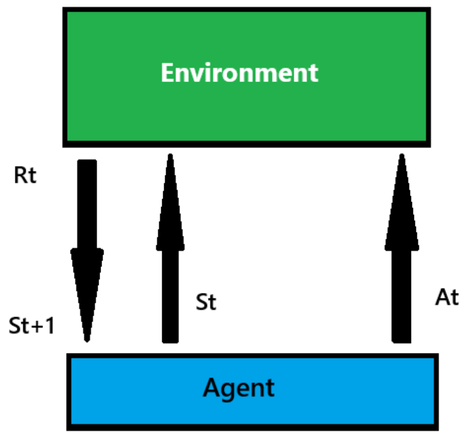


Discrete-time stochastic control process

Time moves forward in finite intervals: $t \in \{1, 2, 3, \dots\}$

Future states depend only partially on the actions taken.

It is based on decision making to reach the target state.



In many sequential decision-making tasks, time progresses in discrete steps — that is, $t = 1, 2, 3, \ldots$. These problems can be effectively modeled using a framework known as a **Markov Decision Process (MDP)**.

An MDP describes the interaction between an **agent** and an **environment** over a sequence of time steps. The process unfolds as follows:

- At time $t = 0$, the agent observes the initial state of the environment.
- Based on this state, it selects an **action**.
- The action is executed, which causes the environment to transition to a new state.
- In response to this transition, the agent:
  - observes the new state,
  - and receives a scalar **reward signal**, indicating the quality or value of the transition.
- This cycle of *observe* $\rightarrow$ *act* $\rightarrow$ *transition* $\rightarrow$ *reward* continues at each time step.

This discrete-time loop forms the basis of reinforcement learning problems, where the goal is to learn a policy that maximizes the cumulative reward over time.

An important advantage of the Markov Decision Process (MDP) framework is that it allows us to describe a control task in a compact and well-defined manner using just four core components:

- **State Space ($\mathcal{S}$):**
  The set of all possible states in which the environment (or task) can exist. Each state represents a snapshot of the environment at a given time.
- **Action Space ($\mathcal{A}$):**
  The set of all possible actions the agent can take while interacting with the environment.
- **Reward Function ($\mathcal{R}$):**
  A mapping that assigns a numerical reward to each state-action pair (or state-action-next-state triplet). It quantifies the immediate benefit of taking a specific action in a given state.
- **Transition Probability Function ($\mathcal{P}$):**
  Defines the probability of transitioning from one state to another, given a specific action. That is,
  $$\mathcal{P}(s' \mid s, a) = \Pr(s_{t+1} = s' \mid s_t = s, a_t = a)$$

Together, these four elements — $(\mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{P})$ — fully specify a Markov Decision Process and allow us to formally describe and solve any control task.

The process has no memory

$$\mathcal{P}(s_{t+1} \mid s_t = s_t)$$

The next state depends only on the current state and not on the previous ones.

Why this is good:

- Simplifies modeling and computation
- Enables efficient algorithms (e.g., Q-learning, Dynamic Programming)
- Requires less memory — only the current state is needed

Why this can be bad:

- Ignores useful historical information
- Fails in partially observable environments

In practice, memory can be added using recurrent networks (e.g., LSTM) or by redefining the state to include sufficient history.

2.0.1 Transition Probability Function

Notation:

$$\mathcal{P}(s' \mid s, a) = \Pr(s_{t+1} = s' \mid s_t = s,\ a_t = a)$$

Meaning:

This represents the probability that the environment will transition to state s' at the next time step $t + 1$, given that it is currently in state s and the agent takes action a.

Intuition:

- Suppose you are controlling a robot.
  - State s = "robot is at location A"
  - Action a = "move forward"
- Possible next states s':
  - With probability 0.9, the robot moves to location B (successful move)
  - With probability 0.1, the robot slips and stays at A

Then we have:

$$\mathcal{P}(\text{B} \mid \text{A},\ \text{move}) = 0.9, \quad \mathcal{P}(\text{A} \mid \text{A},\ \text{move}) = 0.1$$

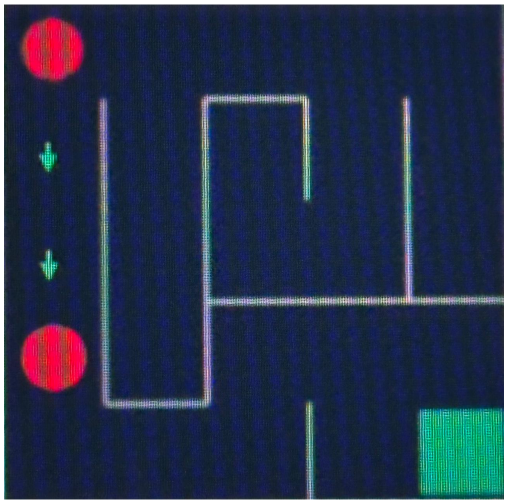
These probabilities capture the uncertainty in the environment's response to the agent's actions.

2.1 Types of Markov Decision Processes

1. **Finite vs. Infinite MDPs**
   - **Finite MDP:** The sets of states, actions, and rewards are all finite.
     Example: A 5×5 maze is a finite MDP:
     - 25 possible positions (states)
     - 4 possible actions per state (e.g., up, down, left, right)
     - Finite set of rewards
   - **Infinite MDP:** At least one of the sets (states, actions, or rewards) is infinite, often due to continuous variables.
     Example: Driving a car:
     - State includes position and speed (both continuous)
     - Infinite possible values (e.g., speed can be 75.0, 75.1, 75.01, etc.)
2. **Episodic vs. Continuing MDPs**
   - **Episodic MDP:** The task terminates after a finite sequence of steps.
     Example: Chess — the episode ends when the game concludes (e.g., checkmate).
   - **Continuing MDP:** The task continues indefinitely without a terminal state.
     Example: Monitoring or controlling an ongoing process (e.g., factory automation).

2.2 Trajectory and Episodes

- **Trajectory**: Elements that are generated when the agent moves from one state to another.



$$\tau = S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3, S_3$$

- **Episode**: Trajectory from the initial state of the task to a terminal state.

$$\tau = S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3, S_3 \rightarrow \text{Terminal}$$

In summary: All episodes are trajectories, but not all trajectories are episodes. An episode must terminate; a trajectory may be partial or infinite.

2.3 Reward vs Return

The goal of the task is represented by the rewards (R_t)
$\rightarrow$ We want to maximize the sum of rewards
$\rightarrow$ A short term reward can worsen long term results

Reward $= R_t$
Return, $G_t = R_{t+1} + R_{t+2} + \ldots\ldots + R_t$
We want to maximize the episode's return

2.4 Discount Factor

The agent has no incentive to go to the goal through the shortest route
$G_t = R_1 + R_2 + \ldots\ldots + R_t$

$\rightarrow$ We have to modify somehow the return in a way that it rewards the most efficient behaviour.
$\therefore$ We will modify future rewards by a discount factor γ
$G_t = R_1 + \gamma R_2 + \gamma^2 R_3 + \ldots\ldots + \gamma^{T-t-1} R_t$

$$\gamma \in [0, 1]$$

We want to maximize the long term sum of discounted rewards
Example:
Suppose an agent is navigating a grid world and receives the following rewards at each time step:

$$R_1 = 5, \quad R_2 = 3, \quad R_3 = 10$$

Let's assume the discount factor $\gamma = 0.9$. The discounted return G_t at time step $t = 1$ would be calculated as:

$$G_1 = R_1 = 5$$

At time step $t = 2$:

$$G_2 = R_1 + \gamma R_2 = 5 + 0.9 \times 3 = 5 + 2.7 = 7.7$$

At time step $t = 3$:

$$G_3 = R_1 + \gamma R_2 + \gamma^2 R_3 = 5 + 0.9 \times 3 + 0.9^2 \times 10 = 5 + 2.7 + 8.1 = 15.8$$

Thus, the agent's total discounted reward at each time step would be:

$$G_1 = 5, \quad G_2 = 7.7, \quad G_3 = 15.8$$

This example shows how future rewards are less valued, making the agent prefer immediate rewards or rewards sooner in time.

2.5 Policy

Policy is a function that decides what action to take in a particular state.

$$\pi : \mathcal{S} \to \mathcal{A}$$

Probability of taking action a in state s:

$$\pi(\mathcal{A}|\mathcal{S})$$

Action a taken in state s:

$$\pi(\mathcal{S})$$

Deterministic Policy:

$$\pi(s) = a$$

In a deterministic policy, the agent always chooses the same action for a given state.

Example: If the agent is in state $s =$ "intersection", then:

$$\pi(s) = \text{“turn left”}$$

This means the agent will always turn left when it reaches that intersection.

Stochastic Policy:

$$\pi(s) = [p(a_1),\ p(a_2),\ \ldots,\ p(a_n)]$$

In a stochastic policy, the agent chooses an action based on a probability distribution over actions.

Example: In state $s =$ "intersection", the policy might be:

$$\pi(s) = [\text{turn left} : 0.6,\ \text{turn right} : 0.3,\ \text{go straight} : 0.1]$$

Here, the agent mostly prefers to turn left, but occasionally chooses other options based on the probabilities.

Summary:

- Deterministic policies are simpler and more predictable.
- Stochastic policies are more flexible and useful in uncertain or dynamic environments.

2.6 State Value Function

The **state value function**, denoted as $V^\pi(s)$, represents the expected return (cumulative reward) that an agent can obtain starting from state s and following policy π thereafter.

Definition:

$$V^\pi(s) = \mathbb{E}_\pi \left[G_t \mid S_t = s \right]$$

where:

- $V^\pi(s)$: value of state s under policy π
- G_t: return from time t onward
- $\mathbb{E}_\pi$: expectation over the trajectory induced by policy π

Expanded Equation:

$$V^\pi(s) = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s \right]$$

where $\gamma \in [0, 1]$ is the discount factor.

Example:
Consider a simple grid world where:

- The agent receives a reward of $+1$ for reaching the goal.
- From state $s = A$, it takes 3 steps on average to reach the goal.
- The policy π always chooses the shortest path.
- The discount factor $\gamma = 0.9$

Then the expected return from state A is:

$$V^\pi(A) = \gamma^0 \cdot 0 + \gamma^1 \cdot 0 + \gamma^2 \cdot 0 + \gamma^3 \cdot 1 = 0.9^3 = 0.729$$

So, under this policy, the value of state A is 0.729.

The value function helps the agent understand which states are more desirable in the long run.

Difference Between Return and State Value
Consider a simple environment in which an agent starts at state A and navigates step by step through different states to reach a goal state, where it receives a reward of $+10$.

Return: What Actually Happened in One Episode
Suppose the agent starts in state A at time $t = 0$ and reaches the goal in 3 steps. The rewards received during this episode are:

$$R_1 = 0, \quad R_2 = 0, \quad R_3 = 10$$

The return from time step 0 is then:

$$G_0 = R_1 + \gamma R_2 + \gamma^2 R_3 = 0 + 0 + \gamma^2 \cdot 10 = \gamma^2 \cdot 10$$

This value G_0 represents the total reward collected in this single episode.

State Value: Expected Return from a State

Now suppose the agent plays 100 episodes, each starting from state A, following the same policy π. Due to randomness in the environment, the number of steps to reach the goal varies across episodes — some reach the goal in 3 steps, others in 5 steps, and some may not reach it at all.
The return G_0 will vary in each episode. The **state value function** $V^\pi(A)$ is defined as the expected return starting from state A and following policy π:

$$V^\pi(A) = \mathbb{E}_\pi[G_0 \mid S_0 = A]$$

In this case, $V^\pi(A)$ would be the average of the 100 returns observed, and it quantifies how valuable state A is under policy π in terms of long-term expected reward.

Conclusion:
The *return* is the actual reward accumulated in one specific run, while the *state value* is the expected return across many possible runs starting from the same state under a given policy.

2.7 Action-Value Function (Q-value)

The **action-value function**, also known as the **Q-value**, is denoted as $Q^\pi(s,a)$. It represents the expected return (cumulative discounted reward) starting from state s, taking action a, and then following policy π thereafter.

Definition:

$$Q^\pi(s,a) = \mathbb{E}_\pi\left[G_t \mid S_t = s,\ A_t = a\right]$$

where:

- $Q^\pi(s,a)$ is the value of taking action a in state s under policy π
- G_t is the return from time step t onward
- The expectation is over all possible future state-action-reward sequences

Expanded Equation:

$$Q^\pi(s,a) = \mathbb{E}_\pi\left[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots \mid S_t = s,\ A_t = a\right]$$

$$Q^\pi(s,a) = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s,\ A_t = a\right]$$

Interpretation:
While the state-value function $V^\pi(s)$ tells us how good it is to be in a state s, the Q-value $Q^\pi(s,a)$ tells us how good it is to *take action* a in state s, and then follow policy π.

Relation to Value Function:

$$V^\pi(s) = \sum_{a \in \mathcal{A}} \pi(a \mid s) \cdot Q^\pi(s,a)$$

This shows that the state-value is the expected Q-value, averaged over the policy's action probabilities.

Example:
Assume an agent in state $s = A$ has two actions:

- $Q^\pi(A, \text{“left”}) = 3.2$
- $Q^\pi(A, \text{“right”}) = 5.0$

Then, under policy π, the agent is more likely to choose the "right" action, as it has a higher expected return.

subsectionBellman Equations

The Bellman equations describe the recursive relationships for the value functions under a given policy π. They break down the expected return into immediate reward plus the discounted value of the next state.

1. Bellman Equation for State-Value Function $V^\pi(s)$:

$$V^\pi(s) = \sum_{a \in \mathcal{A}} \pi(a \mid s) \sum_{s' \in \mathcal{S}} \mathcal{P}(s' \mid s, a) \left[\mathcal{R}(s, a, s') + \gamma V^\pi(s')\right]$$

- $\pi(a \mid s)$: probability of taking action a in state s under policy π
- $\mathcal{P}(s' \mid s, a)$: probability of transitioning to state s' given (s, a)
- $\mathcal{R}(s, a, s')$: expected immediate reward for that transition
- γ: discount factor

2. Bellman Equation for Action-Value Function $Q^\pi(s, a)$:

$$Q^\pi(s, a) = \sum_{s' \in \mathcal{S}} \mathcal{P}(s' \mid s, a) \left[\mathcal{R}(s, a, s') + \gamma \sum_{a' \in \mathcal{A}} \pi(a' \mid s') Q^\pi(s', a')\right]$$

- $Q^\pi(s, a)$: expected return of taking action a in state s and following π afterward
- The inner sum over a' reflects the expected value over the policy's actions at the next state

These recursive definitions are fundamental to many reinforcement learning algorithms, including policy evaluation, Q-learning, and value iteration.

2.8 Solving a Markov Decision Process (MDP)

The goal of solving an MDP is to find an **optimal policy** π^* that maximizes the expected return from each state.

2.8.1 Steps to Solve an MDP

1. **Define the MDP:**
    - State space $\mathcal{S}$
    - Action space $\mathcal{A}$
    - Transition probability $\mathcal{P}(s' \mid s, a)$
    - Reward function $\mathcal{R}(s, a, s')$
    - Discount factor $\gamma \in [0, 1]$
2. **Initialize the value function:**
$$V_0(s) = 0 \quad \forall s \in \mathcal{S}$$
3. **Iteratively apply the Bellman optimality equation:**
$$V_{k+1}(s) = \max_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} \mathcal{P}(s' \mid s, a) \left[\mathcal{R}(s, a, s') + \gamma V_k(s')\right]$$

4. **Extract the optimal policy:**

$$\pi^*(s) = \arg\max_{a \in \mathcal{A}} \sum_{s'} \mathcal{P}(s' \mid s, a) \left[\mathcal{R}(s, a, s') + \gamma V^*(s')\right]$$

Example:

State B has two possible paths to reach terminal state D:

- **Path 1 (Longer path)**: $B \xrightarrow{+1} s_1 \xrightarrow{+2} s_2 \xrightarrow{+10} D$
- **Path 2 (Shortcut)**: $B \xrightarrow{+5} D$

Assume $\gamma = 0.9$. Calculate the value $V(B)$ using value iteration, and determine which path (i.e., which action) the optimal policy $\pi^*(B)$ will choose.

Step 0: Initialization

$$V_0(B) = V_0(s_1) = V_0(s_2) = V_0(D) = 0$$

Step 1: Evaluate terminal transitions

$$\begin{aligned}
V_1(s_2) &= 10 + 0.9 \cdot 0 = 10 \\
V_1(s_1) &= 2 + 0.9 \cdot 10 = 11 \\
V_1(B_{\text{long}}) &= 1 + 0.9 \cdot 11 = 10.9 \\
V_1(B_{\text{short}}) &= 5 + 0.9 \cdot 0 = 5 \\
V_1(B) &= \max\{10.9, 5\} = 10.9
\end{aligned}$$

Step 2: Check for convergence

$$\begin{aligned}
V_2(s_2) &= 10 \quad \text{(no change)} \\
V_2(s_1) &= 2 + 0.9 \cdot 10 = 11 \quad \text{(no change)} \\
V_2(B_{\text{long}}) &= 1 + 0.9 \cdot 11 = 10.9 \\
V_2(B_{\text{short}}) &= 5 \\
V_2(B) &= \max\{10.9, 5\} = 10.9
\end{aligned}$$

Final Answer:

$$V(B) = 10.9$$

Optimal Policy:
To find the best action from B, compare all available paths using:

$$\pi^*(B) = \arg\max_a \left[R(s, a, s') + \gamma V(s')\right]$$

This equation means: choose the action a that gives the highest total value.
In other words, take the action that gives the best immediate reward $R(s, a, s')$, plus the future value of the next state $V(s')$ (discounted by γ).

Since:

- Long path gives 10.9
- Shortcut gives 5

Therefore:

$$\pi^*(B) = \text{take the long path via } s_1 \to s_2 \to D$$

This path has a higher total discounted return, hence it is optimal.

3 Value Iteration

Value Iteration is a dynamic programming algorithm used to compute the optimal value function $V^*(s)$ in a Markov Decision Process (MDP). It iteratively updates the value of each state using the Bellman optimality equation:

$$V_{k+1}(s) = \max_a \sum_{s'} P(s'|s,a) \left[R(s,a,s') + \gamma V_k(s')\right]$$

The process continues until the values converge, i.e., the change in value becomes smaller than a threshold ϵ.

Example (2x2 Grid):

$$\begin{bmatrix} A & B \\ C & \text{Terminal } (+1) \end{bmatrix}$$

Assume:

- $\gamma = 1$
- Terminal state at [1][1] with reward +1 and value fixed at $V = 1$
- All other rewards = 0
- Transitions are deterministic (e.g., up/down/left/right always succeed)

Initialization:

$$V_0(A) = V_0(B) = V_0(C) = 0, \quad V_0(\text{Terminal}) = 1$$

Iteration 1:

- $V_1(C) = \max\{0 + \gamma \cdot V_0(\text{Terminal})\} = 1$
- $V_1(B)$ and $V_1(A)$ can only reach states with value 0 $\rightarrow$ remain 0

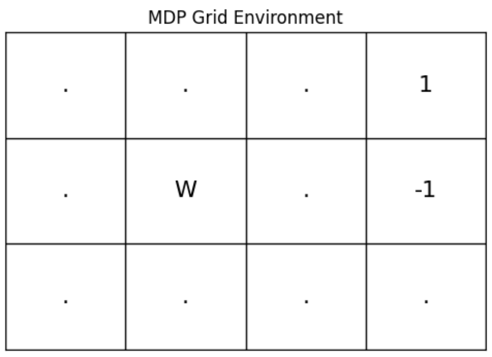
Iteration 2:

- $V_2(B) = \max\{0 + \gamma \cdot V_1(\text{Terminal}), 0 + \gamma \cdot V_1(A)\} = 1$
- $V_2(A) = \max\{0 + \gamma \cdot V_1(B), 0 + \gamma \cdot V_1(C)\} = 1$

Final Value Table (after convergence):

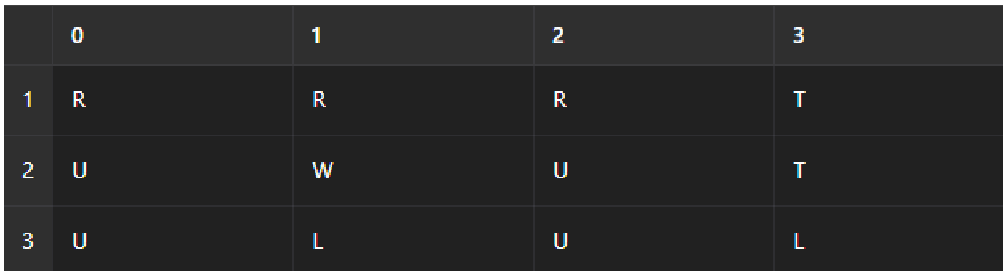
$$V^*(A) = 1, \quad V^*(B) = 1, \quad V^*(C) = 1, \quad V^*(\text{Terminal}) = 1$$

All states learn to reach the terminal state to get the highest reward.



Listing 1: Value Iteration in a Grid World

```python
import numpy as np
import matplotlib.pyplot as plt

# Environment parameters
gamma = 0.9
noise = 0.2
living_reward = 0

# Grid dimensions
rows, cols = 3, 4

# Define the rewards grid
# None = wall, other values = rewards
rewards = np.zeros((rows, cols))
rewards[0, 3] = +1  # Terminal state with +1
rewards[1, 3] = -1  # Terminal state with -1
rewards[1, 1] = None  # Wall

# Terminal and wall states
terminal_states = [(0, 3), (1, 3)]
wall_states = [(1, 1)]

rewards, terminal_states, wall_states

# Display the matrix
display_matrix = np.full((rows, cols), '', dtype=object)
for r in range(rows):
    for c in range(cols):
        if (r, c) in terminal_states:
            display_matrix[r, c] = f"{rewards[r, c]:.0f}"
        elif (r, c) in wall_states:
            display_matrix[r, c] = "W"
        else:
            display_matrix[r, c] = "."

fig, ax = plt.subplots(figsize=(6, 4))
ax.set_xticks(np.arange(cols + 1) - 0.5, minor=True)
ax.set_yticks(np.arange(rows + 1) - 0.5, minor=True)
ax.grid(which="minor", color="black", linestyle='-', linewidth=1)
ax.tick_params(which="minor", bottom=False, left=False)

for r in range(rows):
    for c in range(cols):
        text = display_matrix[r, c]
        ax.text(c, r, text, va='center', ha='center', fontsize=16)

ax.set_xlim(-0.5, cols - 0.5)
ax.set_ylim(rows - 0.5, -0.5)
ax.set_xticks([])
ax.set_yticks([])

plt.title("MDP Grid Environment")
plt.show()

# Actions and their directions
actions = ['U', 'D', 'L', 'R']
action_vectors = {
    'U': (-1, 0),
    'D': (1, 0),
    'L': (0, -1),
    'R': (0, 1)
}

# Initialize value function
V = np.zeros((rows, cols))
policy = np.full((rows, cols), '', dtype=object)

def in_bounds(state):
    r, c = state
    return 0 <= r < rows and 0 <= c < cols and (r, c) not in wall_states

def compute_action_value(state, action, V):
    r, c = state
    primary_move = action_vectors[action]

    if action in ['U', 'D']:
        sideways = ['L', 'R']
    else:
        sideways = ['U', 'D']

    moves = [(primary_move, 1 - noise)] + [(action_vectors[a], noise / 2) for a in
        ↪  sideways]

    value = 0
    for move, prob in moves:
        new_r, new_c = r + move[0], c + move[1]
        if not in_bounds((new_r, new_c)):
            new_r, new_c = r, c
        reward = living_reward if (new_r, new_c) not in terminal_states else
            ↪ rewards[new_r, new_c]
        value += prob * (reward + gamma * V[new_r, new_c])
    return value

theta = 1e-4
iteration = 0
while True:
    delta = 0
    new_V = np.copy(V)
    for r in range(rows):
        for c in range(cols):
            if (r, c) in terminal_states or (r, c) in wall_states:
                continue
            values = [compute_action_value((r, c), a, V) for a in actions]
            best_value = max(values)
            new_V[r, c] = best_value
            delta = max(delta, abs(best_value - V[r, c]))
    V = new_V
    iteration += 1
    if delta < theta:
        break

for r in range(rows):
    for c in range(cols):
        if (r, c) in terminal_states:
            policy[r, c] = 'T'
        elif (r, c) in wall_states:
                policy[r, c] = 'W'
            else:
                values = [compute_action_value((r, c), a, V) for a in actions]
                best_action = actions[np.argmax(values)]
                policy[r, c] = best_action

import pandas as pd
import ace_tools as tools; tools.display_dataframe_to_user(name="Value Function",
    ↪ dataframe=pd.DataFrame(V))

policy_display = pd.DataFrame(policy)
tools.display_dataframe_to_user(name="Policy", dataframe=policy_display)
```



	0	1	2	3
1	R	R	R	T
2	U	W	U	T
3	U	L	U	L

4 Policy Iteration

Policy Iteration is a dynamic programming algorithm used to compute the optimal policy π^* in a Markov Decision Process (MDP). It consists of two main steps that are repeated until convergence:

1. **Policy Evaluation:** For a fixed policy π, compute the value function $V^\pi(s)$ by solving:

$$V^\pi(s) = \sum_{s'} P(s'|s, \pi(s)) \left[R(s, \pi(s), s') + \gamma V^\pi(s')\right]$$

2. **Policy Improvement:** Update the policy by choosing the action that yields the highest expected return:

$$\pi_{\text{new}}(s) = \arg\max_a \sum_{s'} P(s'|s, a) \left[R(s, a, s') + \gamma V^\pi(s')\right]$$

Repeat these two steps until the policy no longer changes. This results in the optimal value function V^* and the optimal policy π^*.

Key Advantage: Policy Iteration often converges faster than Value Iteration in practice, since it uses full value evaluation before each improvement step.

Listing 2: Policy Iteration in Grid World

```
import numpy as np

# Environment parameters
gamma = 0.9
noise = 0.2
living_reward = 0

# Grid dimensions
rows, cols = 3, 4

# Define rewards grid
```

```
rewards = np.zeros((rows, cols))
rewards[0, 3] = +1
rewards[1, 3] = -1
rewards[1, 1] = None  # Wall

# Terminal and wall states
terminal_states = [(0, 3), (1, 3)]
wall_states = [(1, 1)]

# Actions and their vectors
actions = ['U', 'D', 'L', 'R']
action_vectors = {
    'U': (-1, 0),
    'D': (1, 0),
    'L': (0, -1),
    'R': (0, 1)
}

# Utility function to check bounds
def in_bounds(state):
    r, c = state
    return 0 <= r < rows and 0 <= c < cols and (r, c) not in wall_states

# Compute expected value of taking an action at a state
def compute_action_value(state, action, V):
    r, c = state
    primary_move = action_vectors[action]

    if action in ['U', 'D']:
        sideways = ['L', 'R']
    else:
        sideways = ['U', 'D']

    moves = [(primary_move, 1 - noise)] + [(action_vectors[a], noise / 2) for a in
        ↪ sideways]

    value = 0
    for move, prob in moves:
        new_r, new_c = r + move[0], c + move[1]
        if not in_bounds((new_r, new_c)):
            new_r, new_c = r, c
        reward = living_reward if (new_r, new_c) not in terminal_states else
            ↪ rewards[new_r, new_c]
        value += prob * (reward + gamma * V[new_r, new_c])
    return value

# Initialize policy randomly
policy = np.full((rows, cols), 'U', dtype=object)
for r, c in terminal_states + wall_states:
    policy[r, c] = None

# Initialize value function
V = np.zeros((rows, cols))

# Policy Iteration
is_policy_stable = False
iteration = 0

while not is_policy_stable:
    # Policy Evaluation
    while True:
        delta = 0
        new_V = np.copy(V)
        for r in range(rows):
            for c in range(cols):
                if (r, c) in terminal_states or (r, c) in wall_states:
                    continue
                v = compute_action_value((r, c), policy[r, c], V)
                new_V[r, c] = v
                delta = max(delta, abs(v - V[r, c]))
        V = new_V
        if delta < 1e-4:
            break

    # Policy Improvement
    is_policy_stable = True
    for r in range(rows):
        for c in range(cols):
            if (r, c) in terminal_states or (r, c) in wall_states:
                continue
            old_action = policy[r, c]
            action_values = [compute_action_value((r, c), a, V) for a in actions]
            best_action = actions[np.argmax(action_values)]
            policy[r, c] = best_action
            if best_action != old_action:
                is_policy_stable = False

import pandas as pd
import ace_tools as tools; tools.display_dataframe_to_user(name="Policy Iteration
    ↪ - Value Function", dataframe=pd.DataFrame(V))

policy_display = pd.DataFrame(policy)
tools.display_dataframe_to_user(name="Policy Iteration - Policy", dataframe=
    ↪ policy_display)
```

4.1 Q. If We Learn $V(s)$, How Can We Learn $Q(s, a)$ From That?

Let's walk through a concrete example to show how you can estimate $Q(s, a)$ from $V(s)$.

4.1.1 Example: Small 3-State World

Setup:

- States: S_1, S_2, S_3 (where S_3 is terminal)
- Actions: Left ($\leftarrow$), Right ($\rightarrow$)
- Discount factor: $\gamma = 1.0$

Transitions:

- From S_1:
  - Right $\rightarrow S_2$ (reward $= 0$)
  - Left $\rightarrow S_1$ (stay in place, reward $= 0$)
- From S_2:

  - Right $\rightarrow S_3$ (reward = 1)
  - Left $\rightarrow S_1$ (reward = 0)

- S_3 is terminal (no actions)

4.1.2 Suppose We Have Learned $V(s)$

State	$V(s)$
S_1	0.9
S_2	1.0
S_3	0

4.1.3 Compute $Q(s,a)$ Using:

$$Q(s,a) = r + \gamma V(s')$$

Where:

- r is the reward for taking action a
- s' is the next state
- $\gamma = 1.0$

4.1.4 From S_1:

$$Q(S_1, \text{Right}) = 0 + 1.0 \cdot V(S_2) = 1.0$$
$$Q(S_1, \text{Left}) = 0 + 1.0 \cdot V(S_1) = 0.9$$

State	Action	$Q(s,a)$
S_1	Right	1.0
S_1	Left	0.9

4.1.5 From S_2:

$$Q(S_2, \text{Right}) = 1 + 1.0 \cdot V(S_3) = 1.0$$
$$Q(S_2, \text{Left}) = 0 + 1.0 \cdot V(S_1) = 0.9$$

State	Action	$Q(s,a)$
S_2	Right	1.0
S_2	Left	0.9

4.1.6 Final Result

State	Action	$Q(s,a)$
S_1	Right	1.0
S_1	Left	0.9
S_2	Right	1.0
S_2	Left	0.9

4.1.7 What Did We Do?

We used the formula:

$$Q(s,a) = r + \gamma V(s')$$

to estimate the value of each action in a state, given the estimated value of the next state $V(s')$.

This approach is useful when:

- You've already learned $V(s)$, and
- You either know or can sample what happens when you take each action.

5 Monte Carlo Methods

Monte Carlo methods are a class of algorithms used in Reinforcement Learning (RL) to estimate value functions and learn optimal policies by averaging returns from sampled episodes. **Unlike dynamic programming methods, Monte Carlo methods do not require a complete model of the environment (i.e., transition probabilities or reward functions). Instead, they learn directly from experience.**

Why Monte Carlo?

- Can be used in model-free settings, where the environment's dynamics are unknown.
- Suitable for episodic tasks where learning happens at the end of each episode.
- Converges to accurate value estimates given enough samples (by the Law of Large Numbers).

How it Works:

1. Generate episodes by interacting with the environment using a policy π.
2. For each state (or state-action pair) visited, observe the return G_t after that visit.
3. Use the average of these returns to estimate the value function $V(s)$ or $Q(s, a)$.
4. Optionally, improve the policy using the estimated values (e.g., ϵ-greedy).

Two Variants:

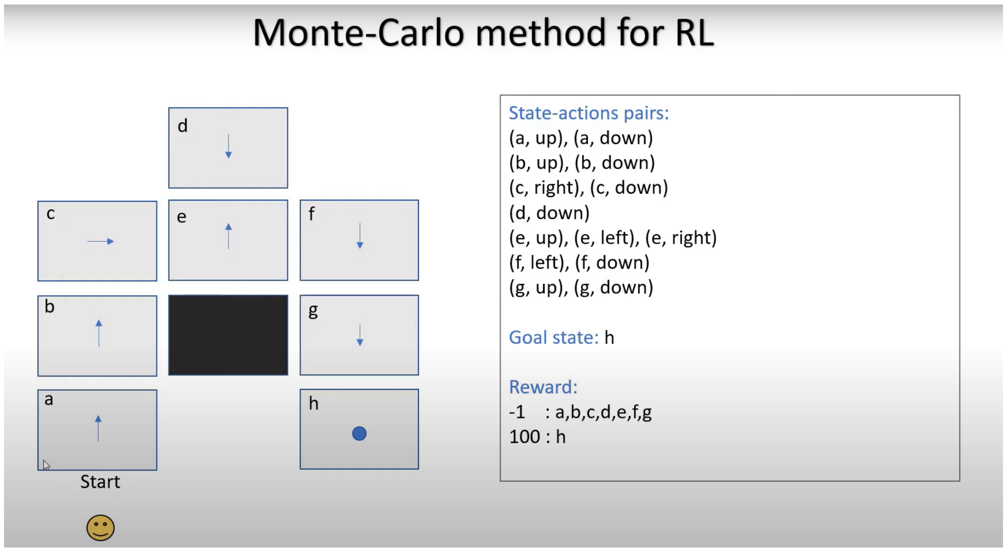
- **First-visit MC:** Only the first occurrence of a state in each episode is used for updating the value.
- **Every-visit MC:** Every occurrence of the state in an episode is used for updating.

Value Estimation Equation:

$$V(s) \approx \frac{1}{N(s)} \sum_{i=1}^{N(s)} G^{(i)} \quad \text{where } G^{(i)} \text{ is the return after } i^{\text{th}} \text{ visit to } s$$

Monte Carlo methods are powerful when dealing with environments where learning is based purely on sampling, such as in games, simulations, or real-world robotics.

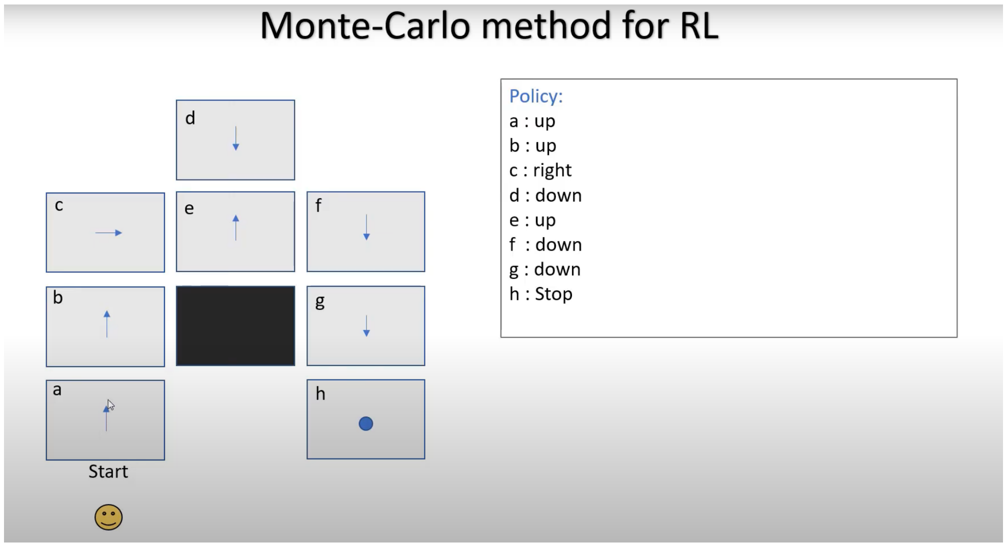
5.1 A Classic Example



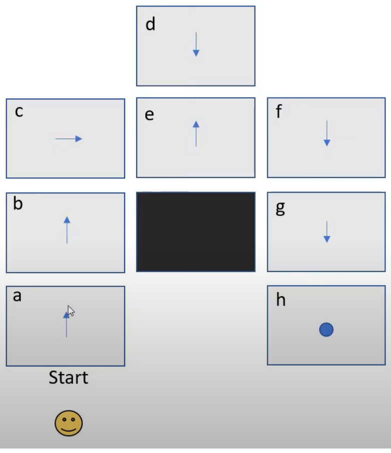
State-actions pairs:
(a, up), (a, down)
(b, up), (b, down)
(c, right), (c, down)
(d, down)
(e, up), (e, left), (e, right)
(f, left), (f, down)
(g, up), (g, down)

Goal state: h

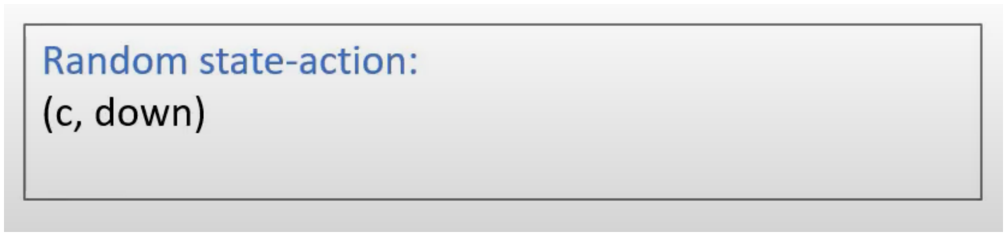
Reward:
-1 : a,b,c,d,e,f,g
100 : h



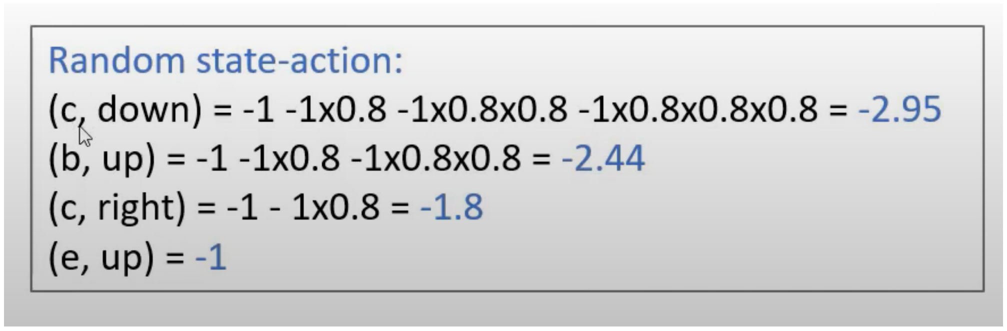
Monte-Carlo method for RL



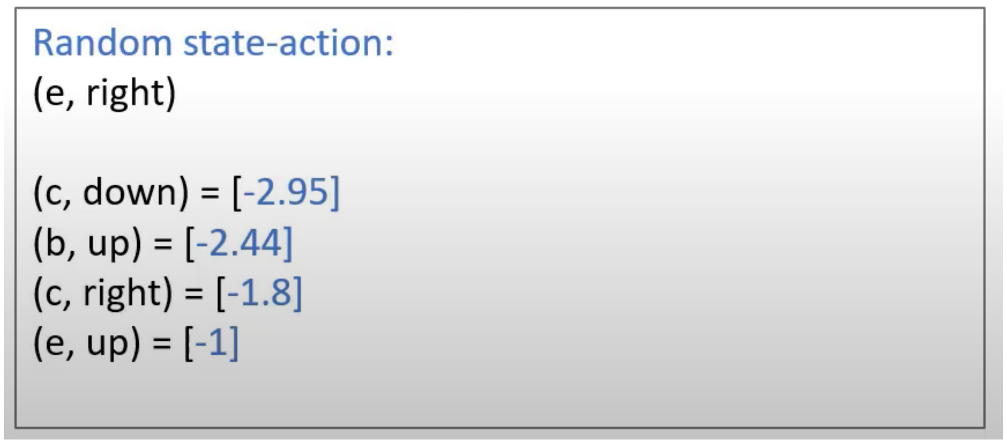
Policy:
a : up
b : up
c : right
d : down
e : up
f : down
g : down
h : Stop



Random state-action:
(c, down)

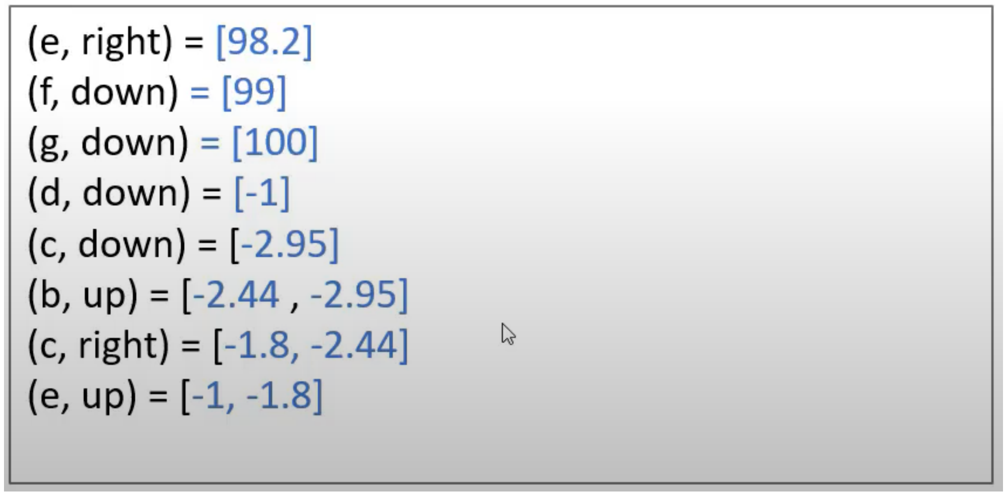


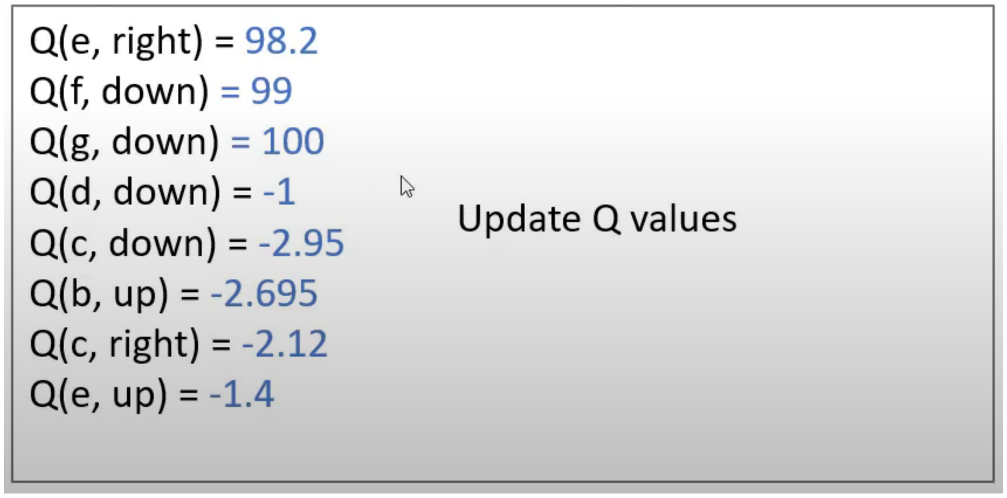
Random state-action:
(c, down) = -1 -1x0.8 -1x0.8x0.8 -1x0.8x0.8x0.8 = -2.95
(b, up) = -1 -1x0.8 -1x0.8x0.8 = -2.44
(c, right) = -1 - 1x0.8 = -1.8
(e, up) = -1



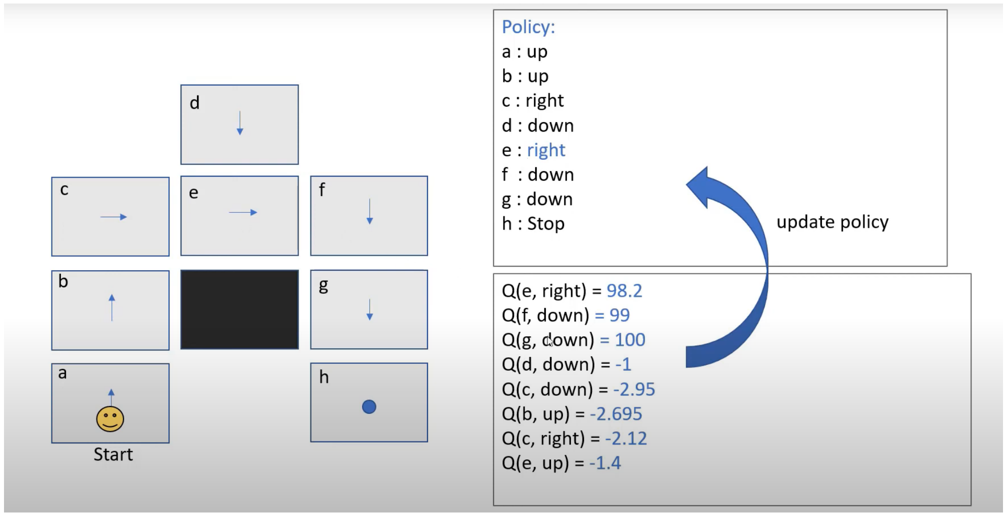
Random state-action:
(e, right)

(c, down) = [-2.95]
(b, up) = [-2.44]
(c, right) = [-1.8]
(e, up) = [-1]


(e, right) = [98.2]
(f, down) = [99]
(g, down) = [100]
(d, down) = [-1]
(c, down) = [-2.95]
(b, up) = [-2.44 , -2.95]
(c, right) = [-1.8, -2.44]
(e, up) = [-1, -1.8]


Q(e, right) = 98.2
Q(f, down) = 99
Q(g, down) = 100
Q(d, down) = -1
Q(c, down) = -2.95
Q(b, up) = -2.695
Q(c, right) = -2.12
Q(e, up) = -1.4

Update Q values



Policy:
a : up
b : up
c : right
d : down
e : right
f : down
g : down
h : Stop

Q(e, right) = 98.2
Q(f, down) = 99
Q(g, down) = 100
Q(d, down) = -1
Q(c, down) = -2.95
Q(b, up) = -2.695
Q(c, right) = -2.12
Q(e, up) = -1.4

5.2 On-Policy Monte Carlo

On-policy Monte Carlo methods learn the value of the policy that is used to generate the data. That is, the same policy π is used both for interacting with the environment (exploration) and for evaluating or improving the policy.

A commonly used approach is ϵ-greedy:

- With probability $1 - \epsilon$, choose the best action.
- With probability ϵ, choose a random action.

This helps in balancing *exploration* and *exploitation*.
The ϵ-greedy policy is a simple yet effective method for balancing exploration and exploitation.

- With probability ϵ, select a random action.
- With probability $1 - \epsilon$, select the action with the highest $Q(s, a)$.

The formal definition of the policy is:

$$\pi(a \mid s) = \begin{cases} 1 - \epsilon + \epsilon_r, & \text{if } a = a^* \\ \epsilon_r, & \text{if } a \neq a^* \end{cases}$$

where:

$$\epsilon_r = \frac{\epsilon}{|\mathcal{A}|}$$

and a^* is the action with the highest value:

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

Here, $|\mathcal{A}|$ is the total number of actions available in state s.

Scenario:
An agent navigates a 3×3 grid world. The goal is in the bottom-right cell with a reward of $+10$. All other steps have a reward of -1. The agent uses an ϵ-greedy policy to explore and improve.

Start $(0, 0)$	$(0, 1)$	$(0, 2)$
$(1, 0)$	$(1, 1)$	$(1, 2)$
$(2, 0)$	$(2, 1)$	**Goal** $(2, 2)$**: +10**

Episode:
The agent starts at state $(0, 0)$ and takes the following actions:

$$(0,0) \xrightarrow{\text{Right}} (0,1) \xrightarrow{\text{Down}} (1,1) \xrightarrow{\text{Right}} (1,2) \xrightarrow{\text{Down}} (2,2)$$

Each move gives a reward of -1, except reaching $(2, 2)$, which gives $+10$. So the rewards are:

$$R_1 = -1, \quad R_2 = -1, \quad R_3 = -1, \quad R_4 = +10$$

The return from time $t = 0$ is:

$$G_0 = -1 + (-1) + (-1) + 10 = 7$$

Update Rule:

$$Q(s, a) \leftarrow Q(s, a) + \alpha \left[G_t - Q(s, a)\right]$$

Apply this for each (s, a) pair the agent visited.

Policy Update:
After updating $Q(s, a)$ values, the agent updates its policy to become more greedy:

$$\pi(a|s) = \begin{cases} 1 - \epsilon + \frac{\epsilon}{|A|}, & \text{if } a = \arg\max Q(s, a) \\ \frac{\epsilon}{|A|}, & \text{otherwise} \end{cases}$$

Key Point:
The same ϵ-greedy policy is used to **generate episodes** and to **improve itself**, which makes it an *on-policy* method.

After updating the action-value estimates $Q(s,a)$ from an episode, the agent updates its policy to favor better actions. This is done using the **ϵ-greedy strategy**, which balances *exploration* and *exploitation.*

Recall the formula:

$$\pi(a|s) = \begin{cases} 1-\epsilon+\frac{\epsilon}{|A|}, & \text{if } a = \arg\max Q(s,a) \\ \frac{\epsilon}{|A|}, & \text{otherwise} \end{cases}$$

- $|A|$ is the number of possible actions (e.g., Up, Down, Left, Right $\Rightarrow |A| = 4$).
- ϵ is the probability of *not* selecting the greedy action (used for exploration).
- The greedy action is the one with the highest $Q(s,a)$ value.

Numerical Illustration:
Suppose in state $(0,0)$ the updated Q-values after the episode are:

$$Q((0,0),\text{Right}) = 6, \quad Q((0,0),\text{Down}) = 3, \quad Q((0,0),\text{Left}) = 1, \quad Q((0,0),\text{Up}) = 2$$

Let $\epsilon = 0.2$ and $|A| = 4$.

Step 1: Identify best action:
$\arg\max Q((0,0),a) = \text{Right}$

Step 2: Compute probabilities:

- For action **Right (greedy)**:

$$\pi(\text{Right}|(0,0)) = 1-0.2+\frac{0.2}{4} = 0.8+0.05 = 0.85$$

- For other actions (Down, Left, Up):

$$\pi(a|(0,0)) = \frac{0.2}{4} = 0.05$$

Resulting Policy at $(0,0)$:

$$\pi((0,0)) = \{\text{Right: } 0.85, \text{Down: } 0.05, \text{Left: } 0.05, \text{Up: } 0.05\}$$

Interpretation:
The agent will most likely choose the greedy action (Right) 85% of the time. However, 15% of the time, it will explore by randomly picking any of the other three actions. This small probability of exploration helps the agent discover potentially better strategies over time.

This is a core idea behind **On-policy Monte Carlo control with ϵ-greedy policies**, where the same policy is used to both generate and evaluate experience.

5.3 Off-Policy Monte Carlo

Off-policy Monte Carlo methods allow an agent to learn the value of a target policy π using data generated from a different behavior policy μ. This is particularly useful when the optimal policy is not yet fully known, or when exploration must happen under a safer or different policy.

Analogy:
Suppose you want to evaluate the best way to commute to work (your *target policy* π), but instead of driving yourself, you ride with a friend who takes random routes each day (the *behavior policy* μ). After several

trips, you use what you observed to infer how your preferred route might perform. Since your friend's routes differ from yours, you must apply a correction factor — this is where *importance sampling* comes in.

Analogy:
Imagine you want to evaluate a diet plan (target policy π), but instead of following it yourself, you observe your friend who eats randomly (behavior policy μ). After watching many of their meals and health results, you try to estimate how *your* planned diet would have worked by reweighting your friend's outcomes based on how likely their actions align with your diet. This reweighting is done using **importance sampling**.

Mathematical Formulation:
The expected return of the target policy π starting from state s is given by:

$$V^\pi(s) = \mathbb{E}_\mu\left[\rho \cdot G_t\right]$$

where ρ is the *importance sampling ratio*:

$$\rho = \frac{\pi(a_1|s_1) \cdot \pi(a_2|s_2) \cdots \pi(a_T|s_T)}{\mu(a_1|s_1) \cdot \mu(a_2|s_2) \cdots \mu(a_T|s_T)}$$

- $\pi(a|s)$: probability of action a under the target policy.
- $\mu(a|s)$: probability of action a under the behavior policy.
- G_t: actual return observed in the episode starting at time t.
- ρ: adjusts the return G_t to make it represent what would have happened under policy π.

Example:
Suppose an episode consists of two state-action pairs:

$$(s_1, a_1), \quad (s_2, a_2)$$

and the return $G_t = 20$. Assume:

$$\pi(a_1|s_1) = 0.8, \quad \mu(a_1|s_1) = 0.4$$
$$\pi(a_2|s_2) = 0.5, \quad \mu(a_2|s_2) = 0.25$$

Then, the importance sampling ratio is:

$$\rho = \frac{0.8 \cdot 0.5}{0.4 \cdot 0.25} = \frac{0.4}{0.1} = 4$$

So, the corrected return is:

$$V^\pi(s_1) = \rho \cdot G_t = 4 \cdot 20 = 80$$

This tells us that the observed return (20) should count as 80 in estimating $V^\pi(s_1)$, because the actions taken in the episode were much more likely under the target policy than the behavior policy.

Caution: If π and μ are very different, ρ can become extremely large or small, causing *high variance* in the estimates. In practice, techniques like *weighted importance sampling* or *truncation* are often used to control this. **Scenario:**
The agent again navigates a 3×3 grid world. The goal is in the bottom-right cell with a reward of +10. All other steps give a reward of -1.

Start $(0,0)$	$(0,1)$	$(0,2)$
$(1,0)$	$(1,1)$	$(1,2)$
$(2,0)$	$(2,1)$	**Goal** $(2,2)$**: +10**

Behavior Policy:
The agent explores using an ϵ-greedy strategy with $\epsilon = 0.3$.
Target Policy:
The agent wants to learn the **greedy policy** with respect to $Q(s, a)$.
Episode (Generated by Behavior Policy):

$$(0,0) \xrightarrow{\text{Down}} (1,0) \xrightarrow{\text{Right}} (1,1) \xrightarrow{\text{Down}} (2,1) \xrightarrow{\text{Right}} (2,2)$$

Rewards:

$$R_1 = -1, \quad R_2 = -1, \quad R_3 = -1, \quad R_4 = +10$$

Return:

$$G_0 = -1 + (-1) + (-1) + 10 = 7$$

Update Rule (Weighted Importance Sampling):

$$Q(s, a) \leftarrow Q(s, a) + \alpha \cdot w \cdot (G_t - Q(s, a))$$

Importance Sampling Ratio:
For each step in the episode, compute the ratio between the target policy and behavior policy:

$$w_t = \prod_{k=t}^{T-1} \frac{\pi(a_k|s_k)}{b(a_k|s_k)}$$

- $\pi(a|s) = 1$ if a is greedy under target policy, 0 otherwise.
- $b(a|s)$ = probability of taking action a under behavior policy.

Example Calculation:
Suppose there are 4 possible actions. The behavior policy is ϵ-greedy with $\epsilon = 0.3$:

$$b(a|s) = \begin{cases} 0.925, & \text{if } a = \arg\max Q(s, a) \\ 0.025, & \text{otherwise} \end{cases}$$

Assume the agent followed the greedy action for the first 2 steps, and a non-greedy action in step 3. Then:

$$w = \frac{1}{0.925} \cdot \frac{1}{0.925} \cdot \frac{1}{0.025} \cdot \dots$$

Since the importance sampling ratio w becomes small when the agent deviates from the target policy, off-policy methods can suffer from high variance — but they are powerful because they allow learning from any behavior.

Summary:

- Off-policy learning uses one policy for acting (behavior policy) and another for learning (target policy).
- It allows the agent to learn optimal behavior while still exploring broadly.
- Requires importance sampling to correct for distribution mismatch.

Comparison:

- **On-policy:** Learns and evaluates the same policy (e.g., ϵ-greedy).
- **Off-policy:** Learns a target policy while following a different one (requires importance sampling).

Q: What can be done if a Monte Carlo (MC) agent gets stuck in a loop (e.g., moving up and down) and fails to reach the goal?
A: Introduce exploration strategies (e.g., ϵ-greedy) to encourage trying new actions.
Alternatively, use TD methods that update before episode termination to learn even from partial progress.

Refer to this video for more.

6 Temporal Difference Methods

Temporal Difference (TD) methods are a class of model-free reinforcement learning (RL) techniques used to estimate the value of a state (or state-action pair) by learning from incomplete episodes—that is, learning from experience before knowing the final outcome.

Basic TD(0) Update Rule (for state-value function $V(s)$):

$$V(s_t) \leftarrow V(s_t) + \alpha\left[r_{t+1} + \gamma V(s_{t+1}) - V(s_t)\right]$$

Where:

- α: learning rate
- γ: discount factor
- r_{t+1}: reward received after taking action at time t
- $V(s_t)$: current estimate of value at time t
- $V(s_{t+1})$: estimate of value at next state

This TD error:

$$\delta_t = r_{t+1} + \gamma V(s_{t+1}) - V(s_t)$$

represents the difference between the expected and actual reward + future estimate.

Advantages of TD Methods: Can learn online after each step (unlike MC, which waits for episodes to finish).
Can learn without a model (unlike DP).
More efficient and incremental for many practical RL tasks.

Environment Recap
States: S1 → S2 → S3 → Goal
Rewards:

- S1, S2: 0
- S3 → Goal: 1

Terminal state: Goal (no update)
Learning rate: $\alpha = 0.5$
Discount factor: $\gamma = 1.0$
Initial values:

$$V(S1) = 0, \quad V(S2) = 0, \quad V(S3) = 0$$

TD(0) Update Formula

$$V(s) \leftarrow V(s) + \alpha\left[r + \gamma V(s') - V(s)\right]$$

Episode 1

$$\begin{aligned} S1 \to S2: &\quad V(S1) = 0 + 0.5(0 + 0 - 0) = 0 \\ S2 \to S3: &\quad V(S2) = 0 + 0.5(0 + 0 - 0) = 0 \\ S3 \to \text{Goal}: &\quad V(S3) = 0 + 0.5(1 + 0 - 0) = 0.5 \end{aligned}$$

After Episode 1:

$$V(S1) = 0, \quad V(S2) = 0, \quad V(S3) = 0.5$$

Episode 2

$$\begin{aligned} S1 \to S2: &\quad V(S1) = 0 + 0.5(0 + 0 - 0) = 0 \\ S2 \to S3: &\quad V(S2) = 0 + 0.5(0 + 0.5 - 0) = 0.25 \\ S3 \to \text{Goal}: &\quad V(S3) = 0.5 + 0.5(1 - 0.5) = 0.75 \end{aligned}$$

After Episode 2:

$$V(S1) = 0, \quad V(S2) = 0.25, \quad V(S3) = 0.75$$

Episode 3

$$\begin{aligned} S1 \to S2: &\quad V(S1) = 0 + 0.5(0 + 0.25 - 0) = 0.125 \\ S2 \to S3: &\quad V(S2) = 0.25 + 0.5(0 + 0.75 - 0.25) = 0.5 \\ S3 \to \text{Goal}: &\quad V(S3) = 0.75 + 0.5(1 - 0.75) = 0.875 \end{aligned}$$

After Episode 3:

$$V(S1) = 0.125, \quad V(S2) = 0.5, \quad V(S3) = 0.875$$

Episode 4

$$\begin{aligned} S1 \to S2: &\quad V(S1) = 0.125 + 0.5(0 + 0.5 - 0.125) = 0.3125 \\ S2 \to S3: &\quad V(S2) = 0.5 + 0.5(0 + 0.875 - 0.5) = 0.6875 \\ S3 \to \text{Goal}: &\quad V(S3) = 0.875 + 0.5(1 - 0.875) = 0.9375 \end{aligned}$$

After Episode 4:

$$V(S1) = 0.3125, \quad V(S2) = 0.6875, \quad V(S3) = 0.9375$$

Episode 5

$$\begin{aligned} S1 \to S2: &\quad V(S1) = 0.3125 + 0.5(0 + 0.6875 - 0.3125) = 0.5 \\ S2 \to S3: &\quad V(S2) = 0.6875 + 0.5(0 + 0.9375 - 0.6875) = 0.8125 \\ S3 \to \text{Goal}: &\quad V(S3) = 0.9375 + 0.5(1 - 0.9375) = 0.96875 \end{aligned}$$

After Episode 5:

$$V(S1) = 0.5, \quad V(S2) = 0.8125, \quad V(S3) = 0.96875$$

Final Values After 5 Episodes

State	**Value**
S1	0.500
S2	0.812
S3	0.969
Goal	0 (terminal)

These values are approaching the expected return from each state.

6.1 Temporal Differences vs Monte Carlo

Episode:

$$S_1 \to S_2 \to S_3 \to \text{Goal} \quad (\text{reward} = 1)$$

Assume discount factor $\gamma = 1$.

Monte Carlo Update:

Monte Carlo waits until the episode ends and then updates each visited state using the total return.

$$\begin{aligned}
G(S_1) &= 1 &\Rightarrow \quad V(S_1) \leftarrow V(S_1) + \alpha[1 - V(S_1)] \\
G(S_2) &= 1 &\Rightarrow \quad V(S_2) \leftarrow V(S_2) + \alpha[1 - V(S_2)] \\
G(S_3) &= 1 &\Rightarrow \quad V(S_3) \leftarrow V(S_3) + \alpha[1 - V(S_3)]
\end{aligned}$$

TD(0) Update:

TD(0) updates estimates during the episode, using the immediate reward and the estimated value of the next state.

$$\begin{aligned}
S_1 \to S_2: &\quad V(S_1) \leftarrow V(S_1) + \alpha[0 + \gamma V(S_2) - V(S_1)] \\
S_2 \to S_3: &\quad V(S_2) \leftarrow V(S_2) + \alpha[0 + \gamma V(S_3) - V(S_2)] \\
S_3 \to \text{Goal}: &\quad V(S_3) \leftarrow V(S_3) + \alpha[1 + \gamma V(\text{Goal}) - V(S_3)]
\end{aligned}$$

Note: TD(0) uses its own current value estimates to bootstrap during learning, updating after each transition.

Feature	**TD(0)**	**Monte Carlo (MC)**
Update Timing	After each step	**After each episode**
Target	Uses bootstrapped estimate	**Uses actual return**
Data Needed	Doesn't need full episode	**Needs entire episode to finish**
Bias vs Variance	More biased, less variance	**Unbiased**, more variance
Speed	Learns online, incrementally	**Learns slower, episodically**

6.2 SARSA

SARSA stands for: State → *Action* → *Reward* → *State* → *Action*
It's and on-policy reinforcement learning algorithm used to learn the action-value function $Q(s, a)$
On-policy means the agent learns the value of the policy it is actually using to act, including any exploration (like ϵ-greedy behavior). So the updates are based on the actions it actually takes, not just the "greedy" ones.

6.2.1 SARSA Update Rule

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + (s_{t+1}, a_{t+1} - Q(s_t, a_t)]$$

Where:

- s_t, a_t: current state and action
- r_{t+1}: reward after taking a_t
- s_{t+1}, a_{t+1}: next state and next action, taken using current policy (eg, ϵ-greedy).
- α: learning rate
- γ: discount factor

6.2.2 SARSA Example: 3x1 Grid Environment

Environment Description

Grid Layout:

$$[\text{S0}] \xrightarrow{\text{Right}} [\text{S1}] \xrightarrow{\text{Right}} [\text{S2}] \to \text{Goal (reward} = +1)$$

States: S0, S1, S2
Actions: Left ($\leftarrow$), Right ($\rightarrow$)
Start: Always starts at S0
Goal: Located at S2. Reaching it gives reward = +1 and ends the episode.
All other transitions: Reward = 0
Setup

- Discount factor: $\gamma = 1.0$
- Learning rate: $\alpha = 0.5$
- Policy: ε-greedy with $\varepsilon = 0.1$
- Initialization: $Q(s, a) = 0$ for all s, a

One Episode (Using SARSA)
Step 0:

- State: S_0
- Action: $\rightarrow$ (chosen greedily)
- Transition: $S_0 \xrightarrow{\text{Right}} S_1$
- Reward: 0

Step 1:

- State: S_1
- Action: $\rightarrow$
- Transition: $S_1 \xrightarrow{\text{Right}} S_2$
- Reward: 0

Step 2:

- State: S_2
- Action: $\rightarrow$
- Transition: $S_2 \rightarrow$ Goal
- Reward: +1

SARSA Update Rule

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)\right]$$

Update 1: From S_0 with action $\rightarrow$

$$Q(S_0, \rightarrow) = 0 + 0.5 \cdot [0 + 1.0 \cdot Q(S_1, \rightarrow) - 0] = 0$$

(No change since $Q(S_1, \rightarrow) = 0$)
Update 2: From S_1 with action $\rightarrow$

$$Q(S_1, \rightarrow) = 0 + 0.5 \cdot [0 + 1.0 \cdot Q(S_2, \rightarrow) - 0] = 0$$

(No change since $Q(S_2, \rightarrow) = 0$)
Update 3: From S_2 with action $\rightarrow$

$$Q(S_2, \rightarrow) = 0 + 0.5 \cdot [1 + 1.0 \cdot 0 - 0] = 0.5$$

Next Episode: Using Updated $Q(S_2, \rightarrow)$
Back at S_1:

$$Q(S_1, \rightarrow) = 0 + 0.5 \cdot [0 + 1.0 \cdot 0.5 - 0] = 0.25$$

Back at S_0:

$$Q(S_0, \rightarrow) = 0 + 0.5 \cdot [0 + 1.0 \cdot 0.25 - 0] = 0.125$$

After 2 Episodes

State	Action	$Q(s, a)$
S0	$\rightarrow$	0.125
S1	$\rightarrow$	0.25
S2	$\rightarrow$	0.5

6.2.3 Why SARSA Is Special

SARSA:

- Uses **actual actions taken** (on-policy, including exploration via ε-greedy)
- Learns **safely**, reflecting the true behavior of the agent
- Suited for environments with **risk or noise**

6.3 Q-Learning

Q-Learning is an off-policy reinforcement learning algorithm that learns the optimal action-value function $Q^*(s, a)$ regardless of the agent's behavior policy.

It learns: $Q^*(s, a)$ = maximum expected return achievable from state s taking action a

6.4 Q-Learning Update Rule

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

Where:

- s_t, a_t: current state and action
- r_{t+1}: reward after taking a_t
- s_{t+1}: next state
- a': any possible next action
- α: learning rate
- γ: discount factor

6.4.1 Example: Q-Learning in a Linear 3-State World

Environment Setup

$$[\text{S1}] \xrightarrow{\text{Right}} [\text{S2}] \xrightarrow{\text{Right}} [\text{Goal}]$$

- **States:** S1, S2, Goal
- **Actions:** $\rightarrow$ (Right), $\leftarrow$ (Left)
- **Start state:** S1
- **Terminal state:** Goal
- **Reward:** +1 when reaching Goal, otherwise 0
- **Discount factor:** $\gamma = 1.0$
- **Learning rate:** $\alpha = 0.5$
- **Initial Q-values:** $Q(s, a) = 0$ for all s, a

Initial Q-Table

State	Action	Q-value
S1	$\rightarrow$	0
S1	$\leftarrow$	0
S2	$\rightarrow$	0
S2	$\leftarrow$	0

Episode 1
At S1:

- Action taken: $\rightarrow$
- Transition: S1 $\rightarrow$ S2
- Reward: 0

Q-learning update:

$$Q(S1, \rightarrow) \leftarrow Q(S1, \rightarrow) + \alpha \left[r + \gamma \max_{a'} Q(S2, a') - Q(S1, \rightarrow) \right]$$

Since $Q(S2, a') = 0$ for all a':

$$Q(S1, \rightarrow) = 0 + 0.5(0 + 1 \cdot 0 - 0) = 0$$

No change yet.
At S2:

- Action taken: $\rightarrow$
- Transition: S2 $\rightarrow$ Goal
- Reward: 1

Goal is terminal, so $\max Q(\text{Goal}, a') = 0$:

$$Q(S2, \rightarrow) = 0 + 0.5(1 + 0 - 0) = 0.5$$

Updated Q-Table After Episode 1

State	Action	Q-value
S1	$\rightarrow$	0
S1	$\leftarrow$	0
S2	$\rightarrow$	0.5
S2	$\leftarrow$	0

Episode 2
At S1:

- Action taken: $\rightarrow$
- Transition: S1 $\rightarrow$ S2
- Reward: 0

Use updated $Q(S2, \rightarrow) = 0.5$:

$$Q(S1, \rightarrow) = 0 + 0.5(0 + 1 \cdot 0.5 - 0) = 0.25$$

At S2:

- Action taken: $\rightarrow$
- Transition: S2 $\rightarrow$ Goal
- Reward: 1

$$Q(S2, \rightarrow) = 0.5 + 0.5(1 + 0 - 0.5) = 0.75$$

Updated Q-Table After Episode 2

State	Action	Q-value
S1	$\rightarrow$	0.25
S1	$\leftarrow$	0
S2	$\rightarrow$	0.75
S2	$\leftarrow$	0

Summary: Q-Learning Behavior
Q-learning always updates toward the **best possible future**, regardless of the action actually taken (i.e., *off-policy*).

- Learns from $\max Q(s', a')$ at the next state
- Can converge faster to optimal policy than on-policy methods like SARSA
- But may behave more aggressively due to not accounting for exploration