UNIT-II

Monte Carlo Methods
Temporal Difference Learning

Dr. Rashmi A. R.

Dept AIML

NMAMIT

Unit II

Monte Carlo Methods: Monte Carlo prediction, Monte Carlo estimation of action values, Monte Carlo control, Monte Carlo Control without exploring starts, Off-policy prediction via importance sampling, incremental implementation, off-policy Monte Carlo Control

Temporal Difference Learning: TD prediction, Advantages of TD Prediction Methods, Optimality of TD(0), Sarsa: On-Policy TD Control, Q-learning: Off-policy TD Control, Expected Sarsa, Maximization Bias and Double Learning.

Tutorials:

Implementing Monte Carlo control algorithm in Python and applying it to a simple environment.

Implementing off-policy prediction using importance sampling in Python and analysing its effectiveness.

Implementing TD prediction algorithm in Python and applying it to a simple environment.

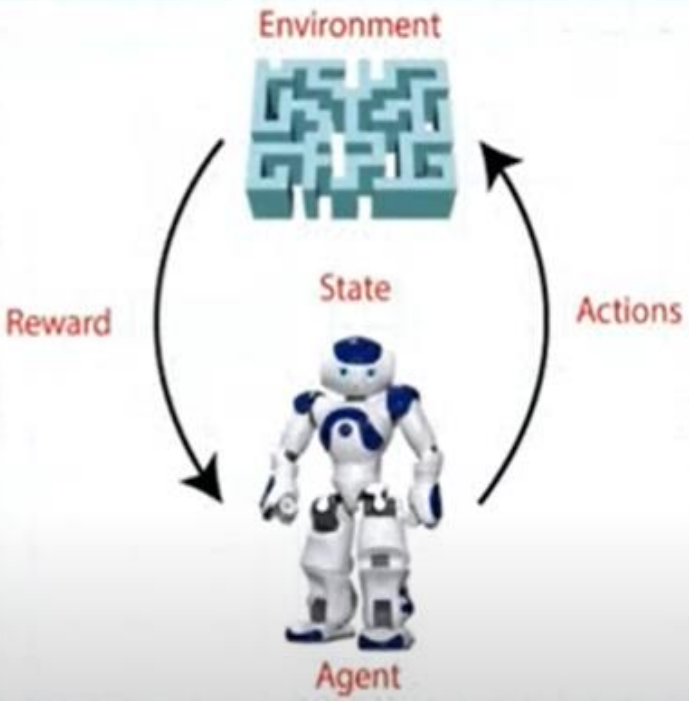
Model Based Reinforcement Learning

In model-based reinforcement learning algorithm, the environment is modelled as a Markov Decision Process (MDS) with following elements:

* A set of states (individual state is denoted by s)
* A set of actions available in each state (individual action is denoted by a)
* Transition probability function from current state (s) to next state (s') under action a

$$T(s, a, s')$$

* Reward function: immediate reward received on transition from current state (s) to next state (s') under action a = $r(s, a, s')$



Model Free Reinforcement Learning: Monte Carlo Method

This is a **Model Free Reinforcement Learning Technique**. It can be used where we are given a simulation model of the environment and the only way to collect information about the environment is by interacting with it.

Monte Carlo Methods

- Monte carlo methods are a broad class of computational algorithms that use repeated random sampling to obtain numerical results like probabilities and expected values. They are often used in physical and mathematical problems where it is difficult or impossible to derive probabilities and expected values using basic principles and concepts

Monte Carlo Method

For symmetric dice (unbiased dice)

Outcome	1	2	3	4	5	6
probability			?			

Monte Carlo Method

For unsymmetric dice (biased dice)

Outcome	1	2	3	4	5	6
probability		?				

Monte Carlo Method

For unsymmetric dice (biased dice)

Outcome	1	2	3	4	5	6
probability		?				

We toss our die (perform the experiment) large number of times, say n, and count n_i = number of times outcome i shows up. Then

$$\text{Probability of outcome } i = \frac{n_i}{n}$$

Monte Carlo Method

For unsymmetric dice (biased dice)

Outcome	1	2	3	4	5	6
probability	$\frac{n_1}{n}$	$\frac{n_2}{n}$	$\frac{n_3}{n}$	$\frac{n_4}{n}$	$\frac{n_5}{n}$	$\frac{n_6}{n}$

n_i = Number of times outcome i shows up

n = Number of times the dice is tossed

Larger the n more accurate the estimate of probabilities will be.

Monte Carlo Methods

- The term “Monte Carlo” is used for any estimation method whose operation involves a significant random component.
- In Reinforcement Learning it is used for methods based on averaging complete returns
  - Does not require complete knowledge of the environment
  - Does not require prior knowledge of the environment’s dynamics
- Although a model is required, the model need only generate sample transitions, not the complete probability distributions of all possible transitions that is required for dynamic programming (DP)
- **Monte Carlo methods require only experience**
  - Sample sequences of states, actions, and rewards from actual or simulated interaction with an environment.
  - Learning from actual experience, yet can still attain optimal behavior.

Monte Carlo Prediction

- Monte Carlo methods can thus be incremental in an episode-by-episode sense, but not in a step-by-step (online) sense
- **Episodic Tasks** :
  - To ensure that well-defined returns are available, we assume experience is divided into episodes
  - all episodes eventually terminate no matter what actions are selected
  - Only on the completion of an episode are value estimates and policies changed

MC Methods

- suppose we wish to estimate $v_{\Pi}(s)$, the value of a state s under policy Π,
- given a set of episodes obtained by following $\Pi$and passing through state s.
- Each occurrence of state s in an episode is called a visit to state s.
- States may be visited multiple times in the same episode
- The first time it is visited in an episode is called the first visit to state s.

Model Free Reinforcement Learning: Monte Carlo Method

This is a **Model Free Reinforcement Learning Technique**. It can be used where we are given a simulation model of the environment and the only way to collect information about the environment is by interacting with it.

This method works along the lines of policy iteration method. There are two steps

1. Policy evaluation: in this method **estimates** of action value function (Q value) for each state action pair (s, a) for a given policy is computed by **averaging the sampled returns that originate from (s, a) over time.** Given sufficient time this procedure can construct precise estimates of Q(s, a) for all state action pairs.

2. Policy improvement: Improved policy is obtained by a **greedy approach** with respect to Q; given a state s new policy returns an action that maximizes Q(s, a)

Model Free Reinfocement Learning - Monte Carlo Method Algorithm

1. Initialize a policy π randomly
2. Policy evaluation step:

    Repeat for large number of times (say n)

        Start an episode and observe initial state s

        Take action a according to policy π

        Complete the game according to policy π and observe Total Reward $R(s, a)$

For each state action pair (s, a) : $Q(s, a)$ = average of $R(s, a)$ originating with (s, a)

3. Policy improvement step:

$$\pi(s) = \underset{a}{arg\,max}\ Q(s, a)$$

4. Repeat step 2 and 3 till policy $\pi(s)$ converges to π^*

Summary of the Algorithm:

The Monte Carlo method for policy evaluation and improvement is an iterative process where the agent starts with a random policy, evaluates it by simulating episodes, and then improves it by making it more greedy based on the estimated value function. This process continues until the policy converges to the optimal policy, ensuring the best possible performance in the environment.

Monte Carlo Method: It relies on averaging returns over many episodes to obtain an accurate estimate of the value function.

Returns: The returns represent the sum of discounted rewards from a particular state onwards.

Types of Monte Carlo (MC) methods are

- **First-visit MC: average returns only for first time S is visited in an episode.**
- First-visit MC method estimates $v_{\Pi}(s)$, as the average of the returns following first visits to S.
- **Every-visit MC: average return for every time S is visited in an episode.**
- Every-visit MC method averages the returns following all visits to S.

Computation of Return

Calculate the value function using the two episodes for both First-Visit and Every Visit Monte Carlo method

$$A+3 \rightarrow A+2 \rightarrow B-4 \rightarrow A+4 \rightarrow B-3 \rightarrow \text{terminate}$$

$$B-2 \rightarrow A+3 \rightarrow B-3 \rightarrow \text{terminate}$$

First Visit MC:

For calculating V(A)

First episode: 3+2-4+4-3=2

Second Episode:3-3=0

V(A)=(2+0)/2=1

For calculating V(B)

First episode: -4+4-3=-3

Second Episode:-2+3-3=-2

V(B)=(-3+-2)/2=-2.5

First visit
V(A) = 1/2(2 + 0) = 1
V(B) = 1/2(-3 + -2) = -5/2

Computation of Return

Calculate the value function using the two episodes for both First-Visit and Every Visit Monte Carlo method

$$A+3 \rightarrow A+2 \rightarrow B-4 \rightarrow A+4 \rightarrow B-3 \rightarrow \text{terminate}$$

$$B-2 \rightarrow A+3 \rightarrow B-3 \rightarrow \text{terminate}$$

Every Visit MC:

For calculating V(A)

First episode: (3+2-4+4-3)+(2-4+4-3)+(4-3)=2+-1+1

Second Episode:(3-3)=0

V(A)=(2+-1+1+0)/4=0.5

For calculating V(B)

First episode: (-4+4-3)+(-3)=-3+-3

Second Episode:(-2+3-3)+(-3)=-2+-3

V(B)=(-3-3-2-3)/4=-2.75

Every visit
V(A) = 1/4(2 + -1 + 1 + 0) = 1/2
V(B) = 1/4(-3 + -3 + -2 + -3) = -11/4

Monte Carlo Prediction-First Visit MC

First-Visit: The update is only performed the first time a state is visited within an episode. This prevents multiple updates in a single episode for the same state, which would otherwise introduce bias.

First-visit MC prediction, for estimating $V \approx v_\pi$

Input: a policy π to be evaluated

Initialize:

$V(s) \in \mathbb{R}$, arbitrarily, for all $s \in \mathcal{S}$

$Returns(s) \leftarrow$ an empty list, for all $s \in \mathcal{S}$

Loop forever (for each episode):

Generate an episode following π: $S_0, A_0, R_1, S_1, A_1, R_2, \ldots, S_{T-1}, A_{T-1}, R_T$

$G \leftarrow 0$

Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:

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

Unless S_t appears in $S_0, S_1, \ldots, S_{t-1}$:

Append G to $Returns(S_t)$

$V(S_t) \leftarrow$ average$(Returns(S_t))$

Monte Carlo Prediction-First Visit MC

Advantages:

Simple and easy to implement.

Does not require knowledge of the transition dynamics of the environment.

Disadvantages:

Requires complete episodes for accurate value estimation.

Can be slow to converge due to the variance in returns across episodes.

Monte Carlo Prediction-Every Visit Monte Carlo

1. Initialize the policy, state-value function
2. Start by generating an episode according to the current policy
    1. Keep track of the states encountered through that episode
3. Select a state in 2.1
    1. Add to a list the return received after every occurrence of this state.
    2. Average over all returns
    3. Set the value of the state as that computed average
4. Repeat step 3
5. Repeat 2-4 until satisfied

Monte Carlo Prediction-Every Visit Monte Carlo

Advantages:

Simplicity: Like First-Visit MC, Every-Visit MC is straightforward and easy to implement.

Improved Sampling: Every-Visit MC uses **all occurrences** of a state, which can lead to faster convergence since more data is used to update the state value.

Disadvantages:

Variance: Like other Monte Carlo methods, Every-Visit MC can have high variance because it relies on sampling entire episodes.

Requires Episodes: Every-Visit MC can only be applied in episodic environments, meaning environments where there is a clear end to each sequence of states, actions, and rewards.

Monte Carlo Estimation of Action Values

- **Dynamic Programming** (With a model)
  - state values alone are sufficient to determine a policy;
  - looks ahead one step and chooses whichever action leads to the best combination of reward and next state
- **Monte Carlo Methods** (Without a model)
  - however, state values alone are not sufficient
  - One must explicitly estimate the value of each action in order for the values to be useful in suggesting a policy.
- Thus, primary goal for Monte Carlo methods is to estimate q_*
- To achieve this, we first consider the **policy evaluation problem/issues** for action values.

Continued on next page

Policy Evaluation Phase

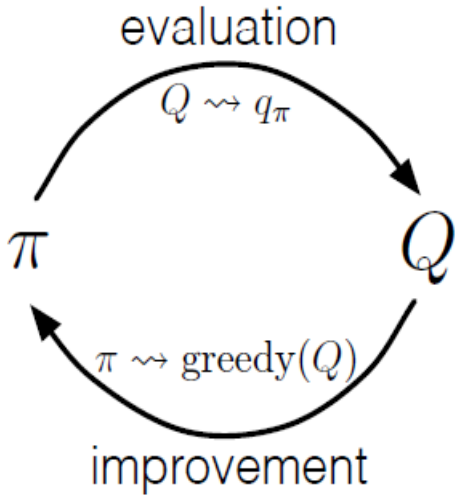
- The policy evaluation problem for action values
  - is to estimate $q_\pi(s, a)$, the expected return when starting in state s, taking action a, following policy π.
- **Issues with policy evaluation for action values include:**
  - Many state–action pairs may never be visited.
  - If π is a deterministic policy, then in following π one will observe returns only for one of the actions from each state.
  - With no returns to average, the Monte Carlo estimates of the other actions will not improve with experience.
- For policy evaluation to work for action values, **continual exploration** is required.
- **Exploring Starts Assumption**
  - Specify that the episodes start in a state–action pair, and that every pair has a nonzero probability of being selected as the start.
  - This guarantees that all state–action pairs will be visited an infinite number of times in the limit of an infinite number of episodes.

Monte Carlo Control

this method, we perform alternating complete steps of policy evaluation and policy improvement, beginning with an arbitrary policy π_0 and ending with the optimal policy and optimal action-value function:

$$\pi_0 \xrightarrow{\text{E}} q_{\pi_0} \xrightarrow{\text{I}} \pi_1 \xrightarrow{\text{E}} q_{\pi_1} \xrightarrow{\text{I}} \pi_2 \xrightarrow{\text{E}} \cdots \xrightarrow{\text{I}} \pi_* \xrightarrow{\text{E}} q_*,$$

where $\xrightarrow{\text{E}}$ denotes a complete policy evaluation and $\xrightarrow{\text{I}}$ denotes a complete policy Improvement.



Generalized Policy Iteration (GPI)

Monte Carlo Control (Cont'd)

- Policy Improvement is a greedy policy with respect to current value function, which is action-value function and therefore no model is required to construct the greedy policy
- For any action-value function q, the corresponding greedy policy is the one that, for each state s, deterministically chooses an action with maximal action value:

$$\pi(s) \doteq \arg\max_a q(s,a).$$

$$\begin{aligned} q_{\pi_k}(s, \pi_{k+1}(s)) &= q_{\pi_k}(s, \arg\max_a q_{\pi_k}(s,a)) \\ &= \max_a q_{\pi_k}(s,a) \\ &\geq q_{\pi_k}(s, \pi_k(s)) \\ &\geq v_{\pi_k}(s). \end{aligned}$$

Monte Carlo Control with Exploring Starts

Monte Carlo ES (Exploring Starts), for estimating $\pi \approx \pi_*$

Initialize:

$\pi(s) \in \mathcal{A}(s)$ (arbitrarily), for all $s \in \mathcal{S}$

$Q(s,a) \in \mathbb{R}$ (arbitrarily), for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$

$Returns(s,a) \leftarrow$ empty list, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$

Loop forever (for each episode):

Choose $S_0 \in \mathcal{S}$, $A_0 \in \mathcal{A}(S_0)$ randomly such that all pairs have probability > 0

Generate an episode from S_0, A_0, following π: $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$

$G \leftarrow 0$

Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:

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

Unless the pair S_t, A_t appears in $S_0, A_0, S_1, A_1 \ldots, S_{t-1}, A_{t-1}$:

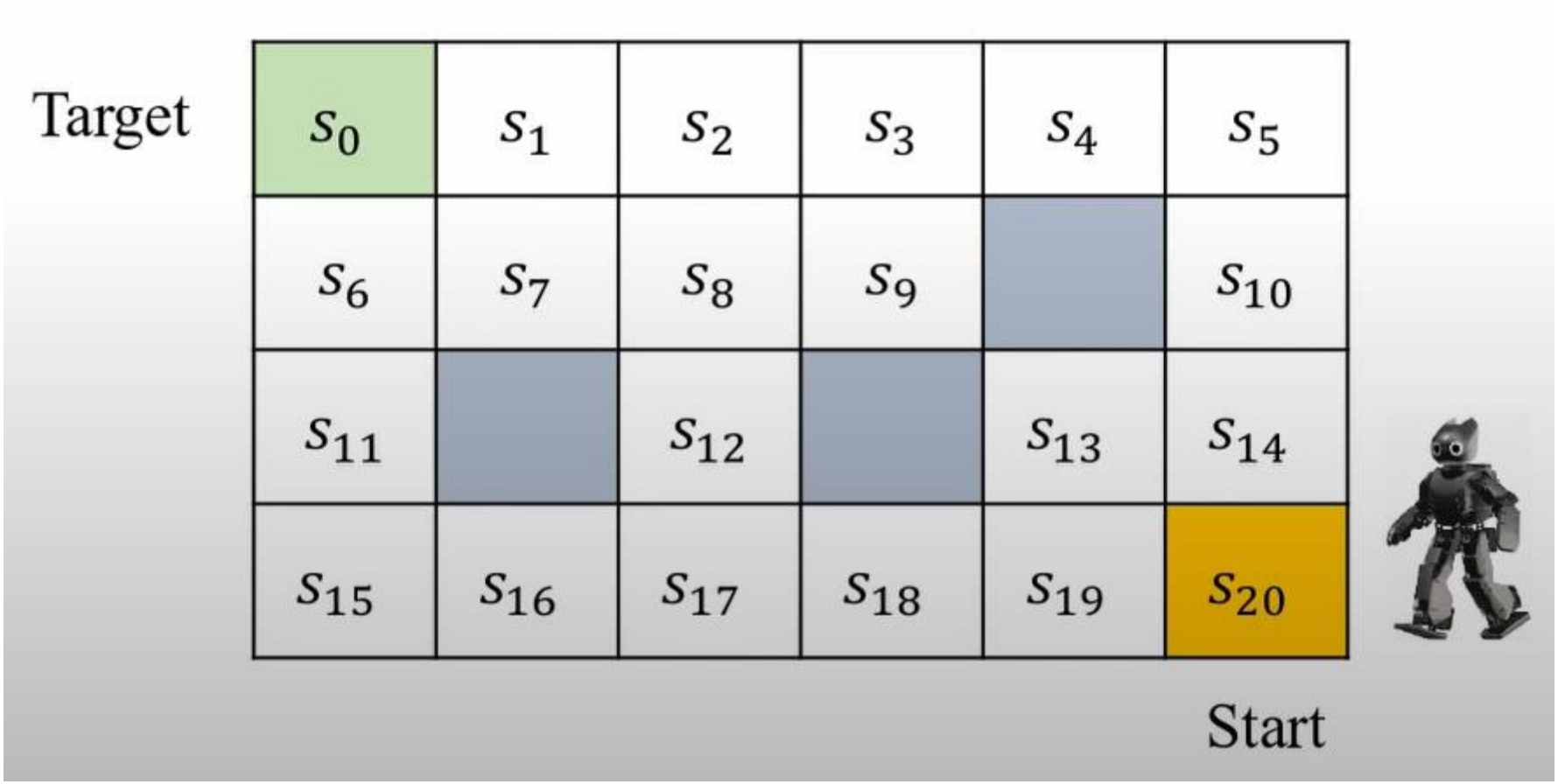
Append G to $Returns(S_t, A_t)$

$Q(S_t, A_t) \leftarrow \text{average}(Returns(S_t, A_t))$

$\pi(S_t) \leftarrow \arg\max_a Q(S_t, a)$

Monte Carlo Control with Exploring Starts

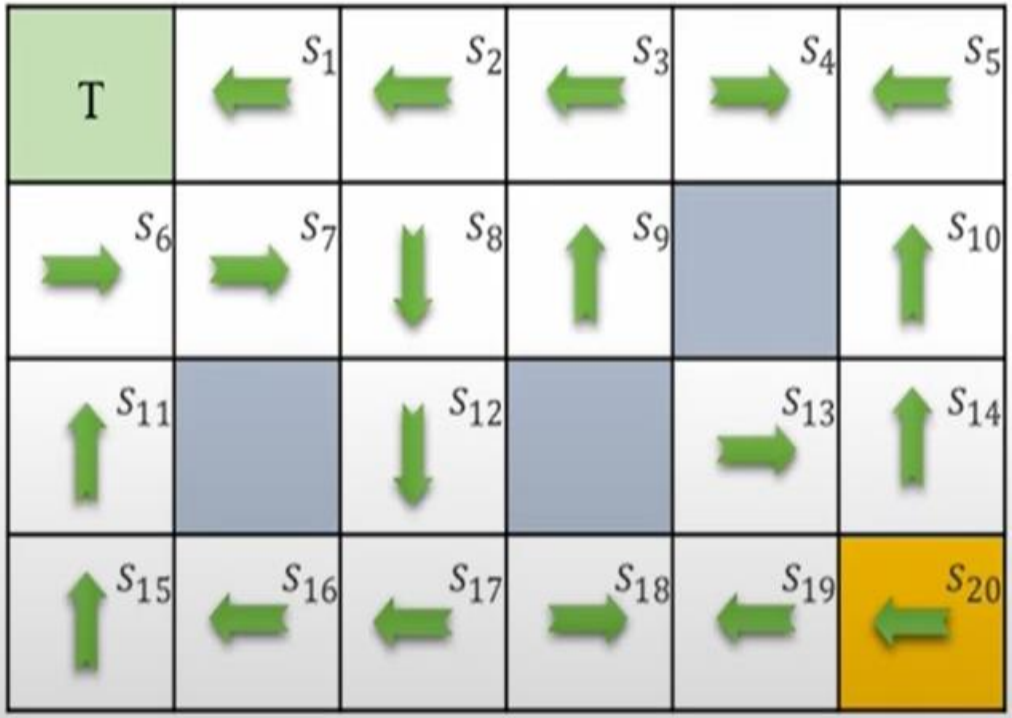
Problem: A robot needs to go from start to target.
Reward: +100 for target, -1 for each other step
Q-value: for simplicity its all zero



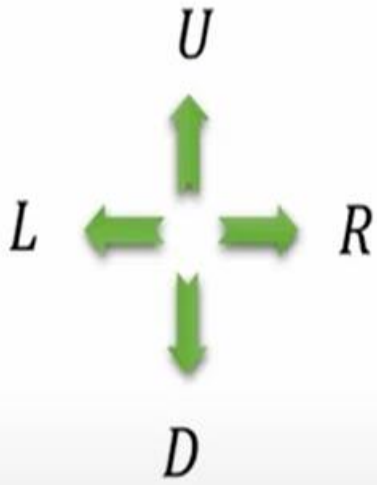
Monte Carlo Control with Exploring Starts

a) We assign arbitrary policy (up, down, left, right) and value for Q-value first .
b) Select random state ‘s’ and random action ‘a’ pair in the environment.
c) Start from selected random state s and run selected random action a first.
d) For each state-action pair compute the discounted returns and create list.
e) Update the policy

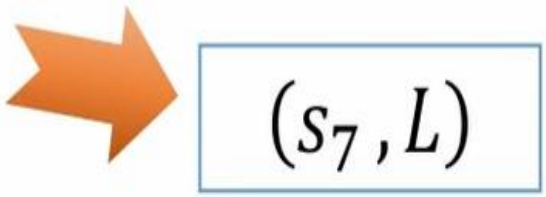
a) We assign arbitrary policy (up, down, left, right) and value for Q-value first



Arbitrary Policy

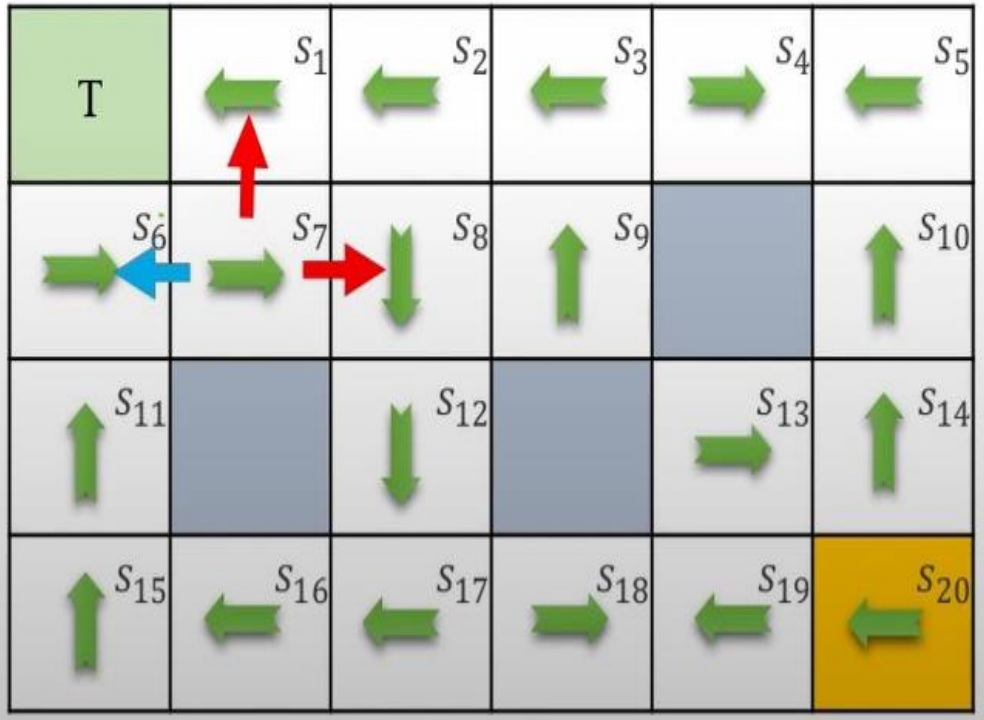


b) Select random state ‘s’ and random action ‘a’ pair in the environment.



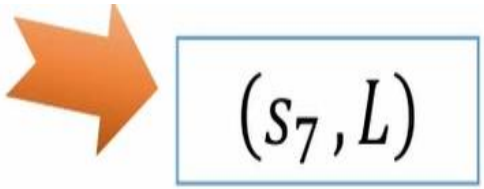
(s_7, L)

Note: Consider only possible actions from each state

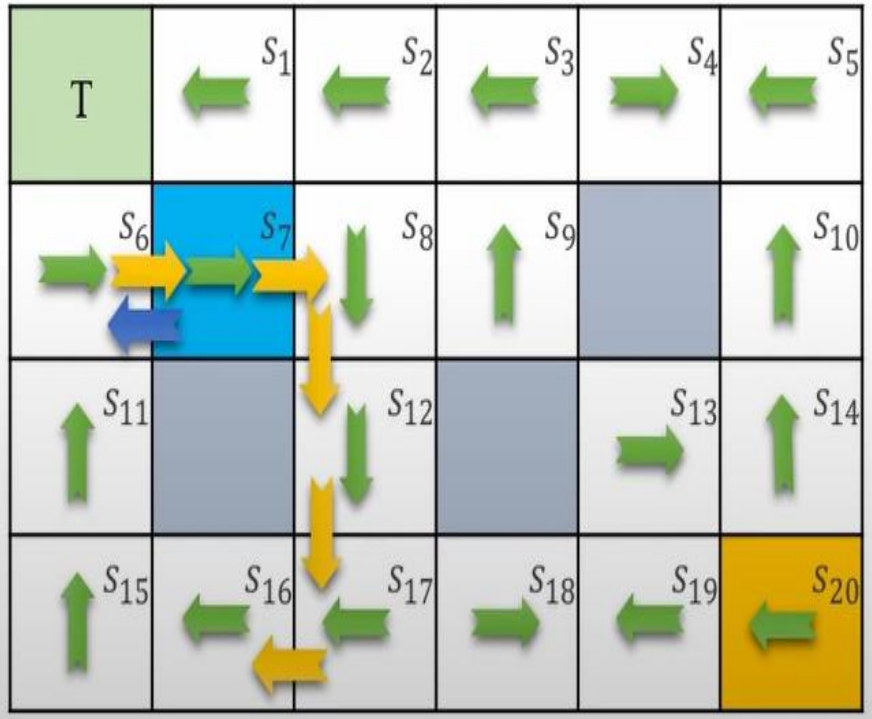


Generate an episode by policy π

c) Start from selected random state s and run selected random action a first.



E1 **Generate episode (1):**



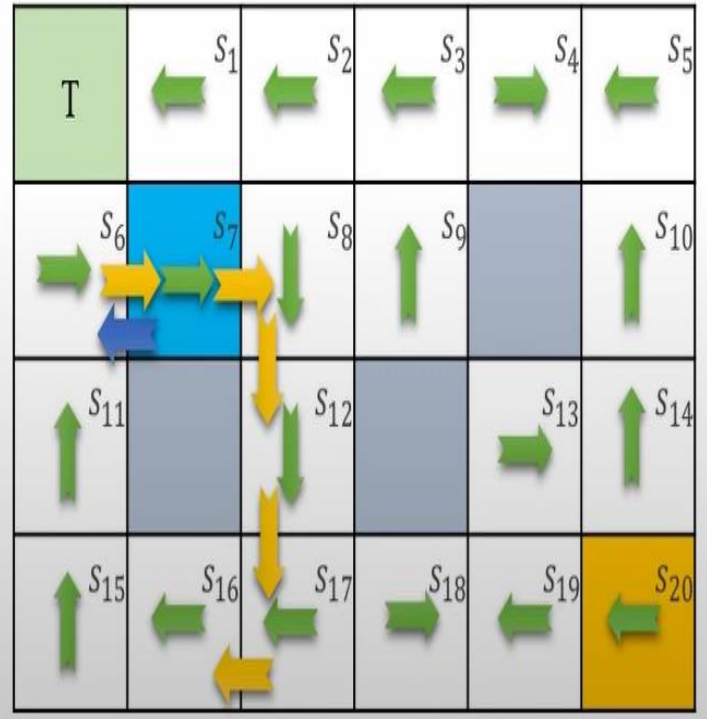
$$(s_7, L) \rightarrow (s_6, R) \rightarrow (s_7, R) \rightarrow (s_8, D) \rightarrow (s_{12}, D) \rightarrow (s_{17}, L) \rightarrow (s_{16}, L)$$

d) For each state-action pair compute the discounted returns and create list.

- **Return is the calculation of only all next states of each state in episode**

Discounted returns: $G = \gamma G + R_{t+1}$ $\gamma = 0.9$

$$(s_7, L) \rightarrow (s_6, R) \rightarrow (s_7, R) \rightarrow (s_8, D) \rightarrow (s_{12}, D) \rightarrow (s_{17}, L) \rightarrow (s_{16}, L)$$



$(s_{17}, L) = (0.9 \times 0) - 1 = -1$

$(s_{12}, D) = (0.9 \times -1) - 1 = -1.9$

$(s_8, D) = (0.9 \times -1.9) - 1 = -2.71$

$(s_7, R) = (0.9 \times -2.71) - 1 = -3.439$

$(s_6, R) = (0.9 \times -3.349) - 1 = -4.095$

$(s_7, L) = (0.9 \times -4.095) - 1 = -4.685$

$\text{Returns\_list}(s_{17}, L) = [-1]$

$\text{Returns\_list}(s_{12}, D) = [-1.9]$

$\text{Returns\_list}(s_8, D) = [-2.71]$

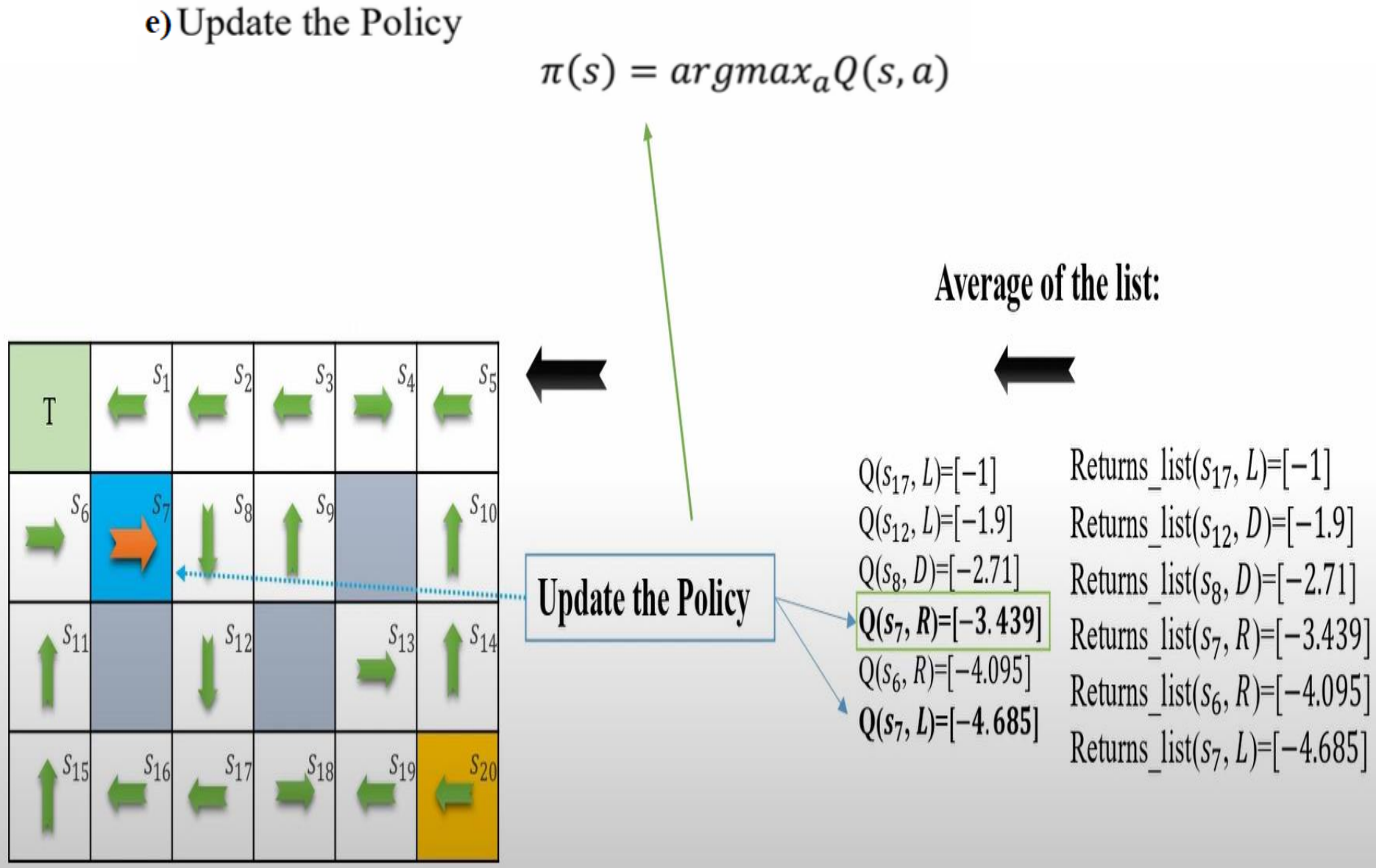
$\text{Returns\_list}(s_7, R) = [-3.439]$

$\text{Returns\_list}(s_6, R) = [-4.095]$

$\text{Returns\_list}(s_7, L) = [-4.685]$

Note: since we do did not run action for the last state, do not need to include.

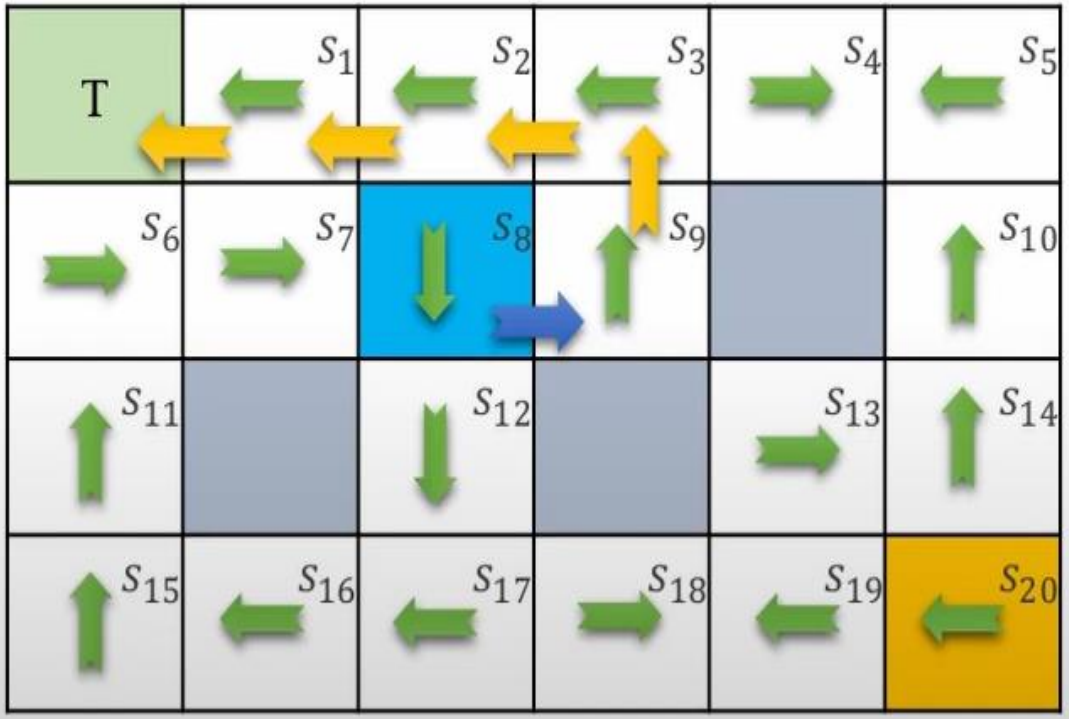
e) Update the policy



Episode 2

Random → (s_8, R)

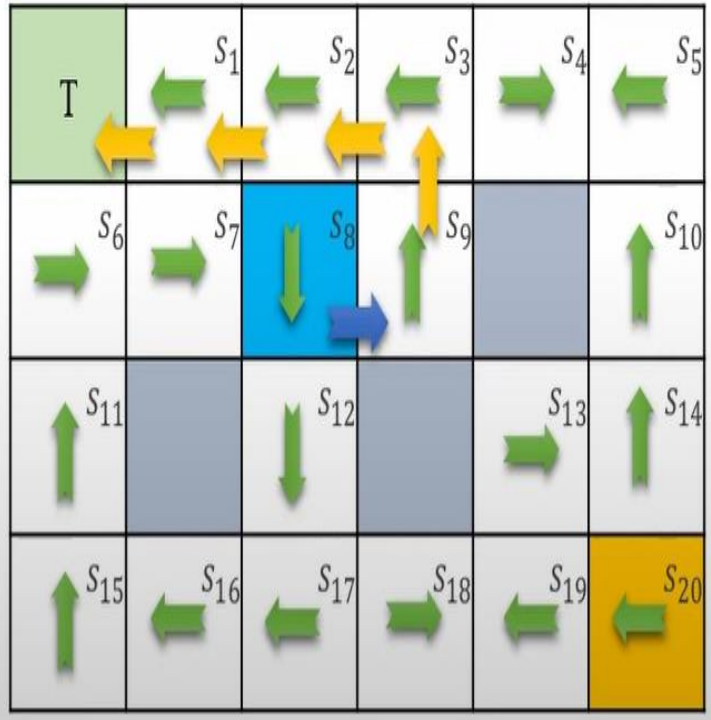
Generate episode (2):



$$(\boldsymbol{s_8}, \boldsymbol{R}) \rightarrow (s_9, U) \rightarrow (s_3, L) \rightarrow (s_2, L) \rightarrow (s_1, L) \rightarrow (s_0, stop)$$

Discounted returns: $G = \gamma G + R_{t+1}$ $\gamma = 0.9$

$$(s_8, R) \rightarrow (s_9, U) \rightarrow (s_3, L) \rightarrow (s_2, L) \rightarrow (s_1, L) \rightarrow (s_0, stop)$$



$(s_1, L) = (0.9 \times 0) + 100 = 100$

$(s_2, L) = (0.9 \times 100) - 1 = 89$

$(s_3, L) = (0.9 \times 89) - 1 = 79.1$

$(s_9, U) = (0.9 \times 79.1) - 1 = 70.19$

$(s_8, R) = (0.9 \times 70.19) - 1 = 62.171$

Returns_list(s_{17}, L)=[−1]

Returns_list(s_{12}, D)=[−1.9]

Returns_list(s_8, D)=[−2.71]

Returns_list(s_7, R)=[−3.439]

Returns_list(s_6, R)=[−4.095]

Returns_list(s_7, L)=[−4.685]

Returns_list(s_1, L)=[100]

Returns_list(s_2, L)=[89]

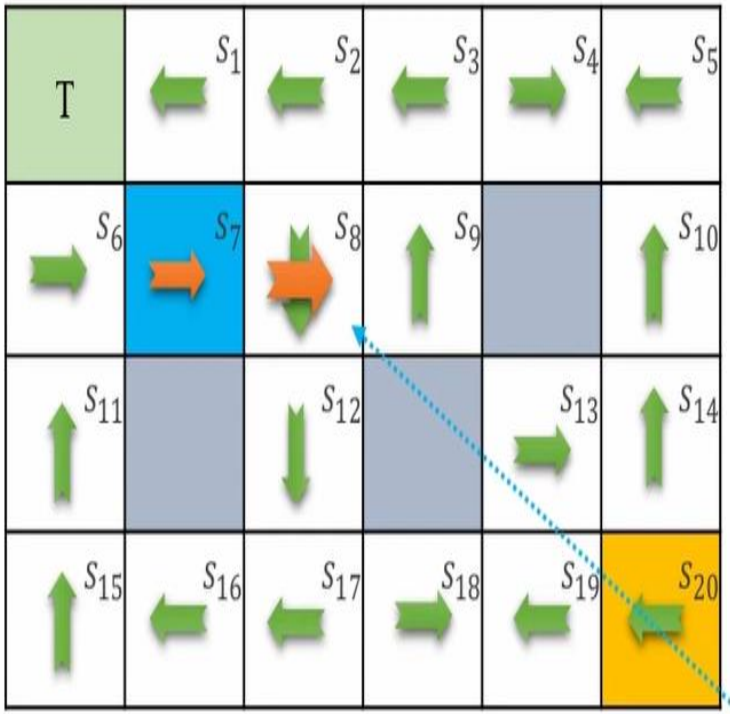
Returns_list(s_3, L)=[79.1]

Returns_list(s_9, U)=[70.19]

Returns_list(s_8, R)=[62.171]

e) Update the Policy

$$\pi(s) = argmax_a Q(s,a)$$



Update the Policy

Average of the list:

$Q(s_{17}, L)$=[−1] — Returns_list(s_{17}, L)=[−1]

$Q(s_{12}, D)$=[−1.9] — Returns_list(s_{12}, D)=[−1.9]

$Q(s_8, D)$=[−2.71] — Returns_list(s_8, D)=[−2.71]

$Q(s_7, R)$=[−3.439] — Returns_list(s_7, R)=[−3.439]

$Q(s_6, R)$=[−4.095] — Returns_list(s_6, R)=[−4.095]

$Q(s_7, L)$=[−4.685] — Returns_list(s_7, L)=[−4.685]

$Q(s_1, L)$=[100] — **Returns_list(s_1, L)=[100]**

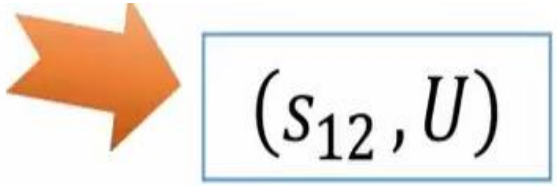
$Q(s_2, L)$=[89] — **Returns_list(s_2, L)=[89]**

$Q(s_3, L)$=[79.1] — **Returns_list(s_3, L)=[79.1]**

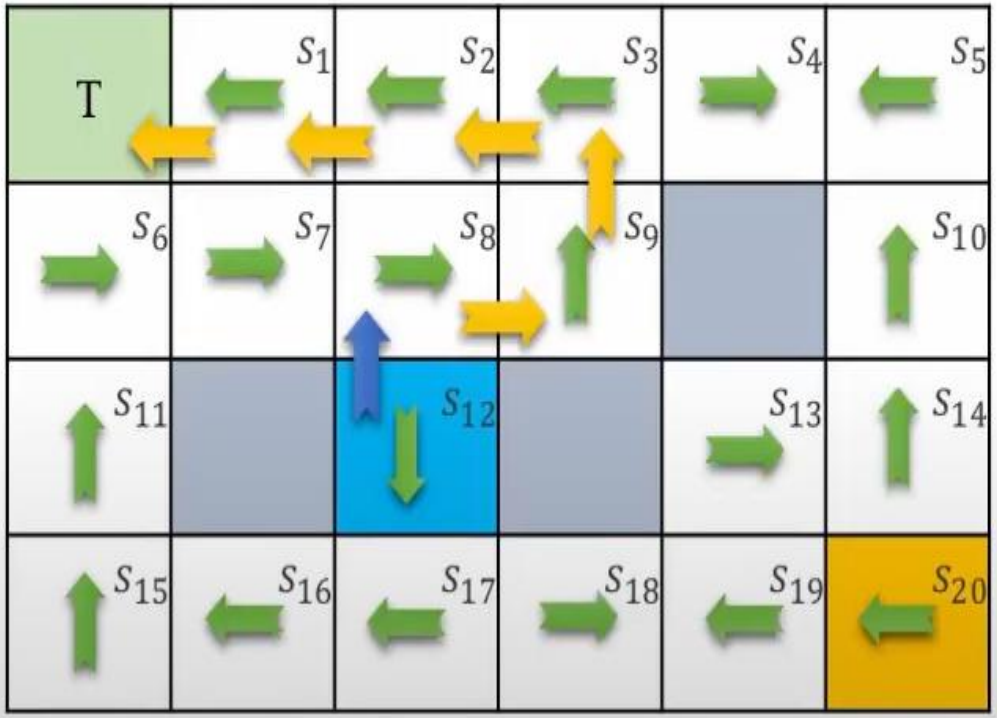
$Q(s_9, U)$=[70.19] — **Returns_list(s_9, U)=[70.19]**

$Q(s_8, R)$=[62.171] — **Returns_list(s_8, R)=[62.171]**

Episode 3



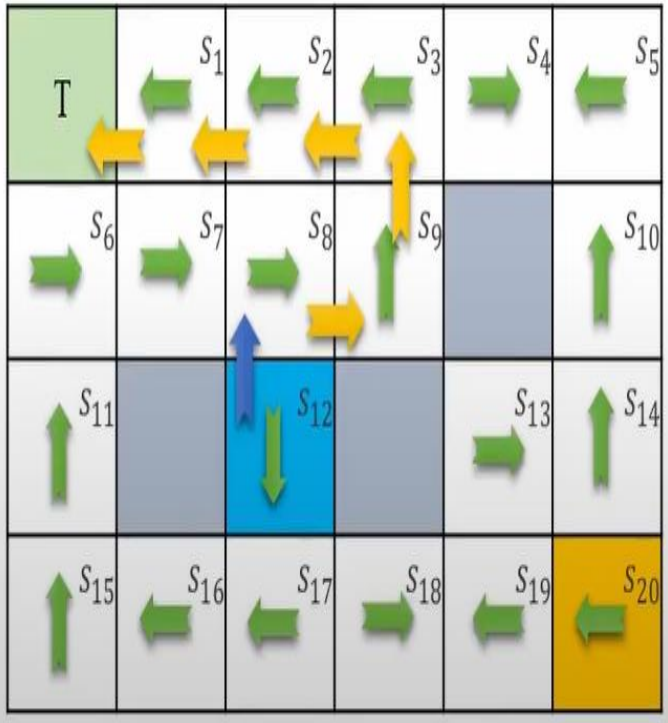
Generate episode (3):



$$(s_{12}, U) \rightarrow (s_8, R) \rightarrow (s_9, U) \rightarrow (s_3, L) \rightarrow (s_2, L) \rightarrow (s_1, L) \rightarrow (s_0, stop)$$

Discounted returns: $G = \gamma G + R_{t+1}$ $\gamma = 0.9$

$$(s_{12}, U) \rightarrow (s_8, R) \rightarrow (s_9, U) \rightarrow (s_3, L) \rightarrow (s_2, L) \rightarrow (s_1, L) \rightarrow (s_0, stop)$$



$(s_1, L) = (0.9 \times 0) + 100 = 100$

$(s_2, L) = (0.9 \times 100) - 1 = 89$

$(s_3, L) = (0.9 \times 89) - 1 = 79.1$

$(s_9, U) = (0.9 \times 79.1) - 1 = 70.19$

$(s_8, R) = (0.9 \times 70.19) - 1 = 62.171$

$(s_{12}, U) = (0.9 \times 62.171) - 1 = 54.953$

Returns_list(s_{17}, L)=[−1]

Returns_list(s_{12}, D)=[−1.9]

Returns_list(s_8, D)=[−2.71]

Returns_list(s_7, R)=[−3.439]

Returns_list(s_6, R)=[−4.095]

Returns_list(s_7, L)=[−4.685]

Returns_list(s_1, L)=[100, **100**]

Returns_list(s_2, L)=[89, **89**]

Returns_list(s_3, L)=[79.1, **79.1**]

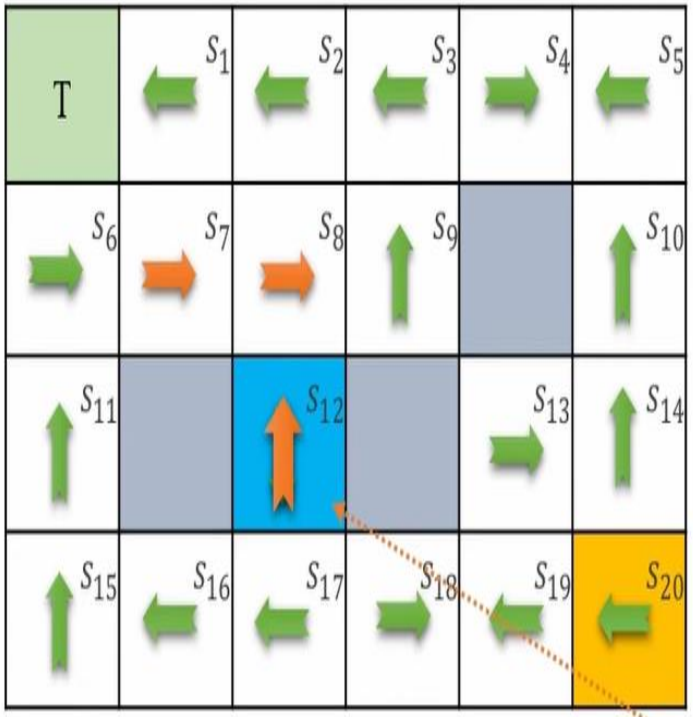
Returns_list(s_9, U)=[70.19, **70.19**]

Returns_list(s_8, R)=[62.171, **62.171**]

Returns_list(s_{12}, U)=[54.953]

e) Update the Policy

$$\pi(s) = argmax_a Q(s,a)$$



Update the Policy

$Q(s\_17, L)=[-1]$

$Q(s\_12, D)=[-1.9]$

$Q(s\_8, D)=[-2.71]$

$Q(s\_7, R)=[-3.439]$

$Q(s\_6, R)=[-4.095]$

$Q(s\_7, L)=[-4.685]$

$Q(s_1, L)=[100]$

$Q(s_2, L)=[89]$

$Q(s_3, L)=[79.1]$

$Q(s_9, U)=[70.19]$

$Q(s_8, R)=[62.171]$

$Q(s_{12}, U)=[54.953]$

Average of the list:

$Returns\_list(s_{17}, L)=[-1]$

$Returns\_list(s_{12}, D)=[-1.9]$

$Returns\_list(s_8, D)=[-2.71]$

$Returns\_list(s_7, R)=[-3.439]$

$Returns\_list(s_6, R)=[-4.095]$

$Returns\_list(s_7, L)=[-4.685]$

$Returns\_list(s_1, L)=[100, \mathbf{100}]$

$Returns\_list(s_2, L)=[89, \mathbf{89}]$

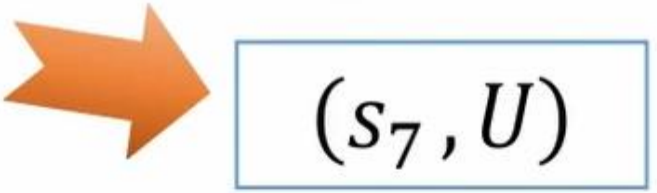
$Returns\_list(s_3, L)=[79.1, \mathbf{79.1}]$

$Returns\_list(s_9, U)=[70.19, \mathbf{70.19}]$

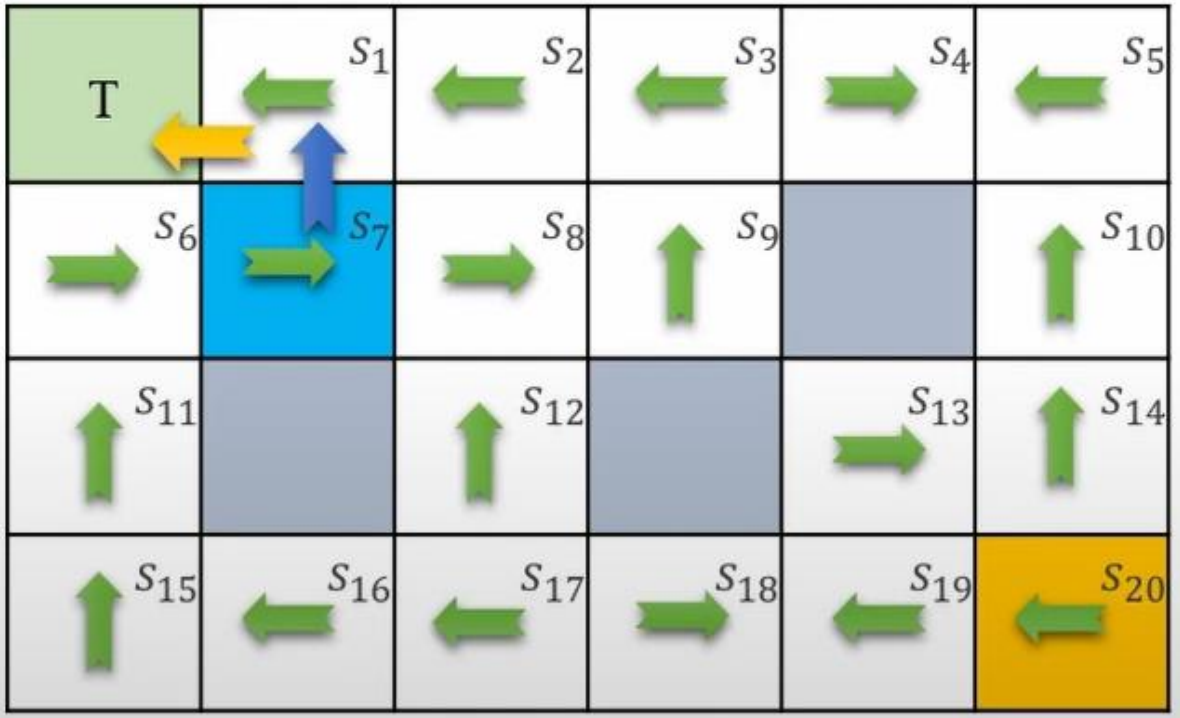
$Returns\_list(s_8, R)=[62.171, \mathbf{62.171}]$

$Returns\_list(s_{12}, U)=[54.953]$

Episode 4



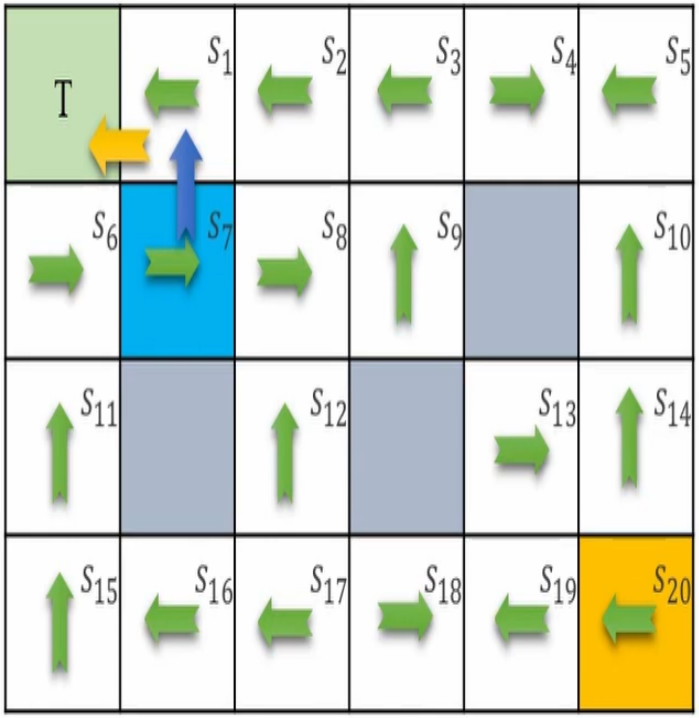
Generate an episode (4):



$$(\boldsymbol{s_7}, \boldsymbol{U}) \rightarrow (s_1, L) \rightarrow (s_0, stop)$$

Discounted returns: $G = \gamma G + R_{t+1}$ $\gamma = 0.9$

$(s_7, U) \rightarrow (s_1, L) \rightarrow (s_0, stop)$



$(s_1, L) = (0.9 \times 0) + 100 = 100$

$(s_7, U) = (0.9 \times 100) - 1 = 89$

Returns_list(s_17, L)=[−1]

Returns_list(s_12, D)=[−1.9]

Returns_list(s_8, D)=[−2.71]

Returns_list(s_7, R)=[−3.439]

Returns_list(s_6, R)=[−4.095]

Returns_list(s_7, L)=[−4.685]

Returns_list(s_1, L)=[100, 100, **100**]

Returns_list(s_2, L)=[89, 89]

Returns_list(s_3, L)=[79.1, 79.1]

Returns_list(s_9, U)=[70.19, 70.19]

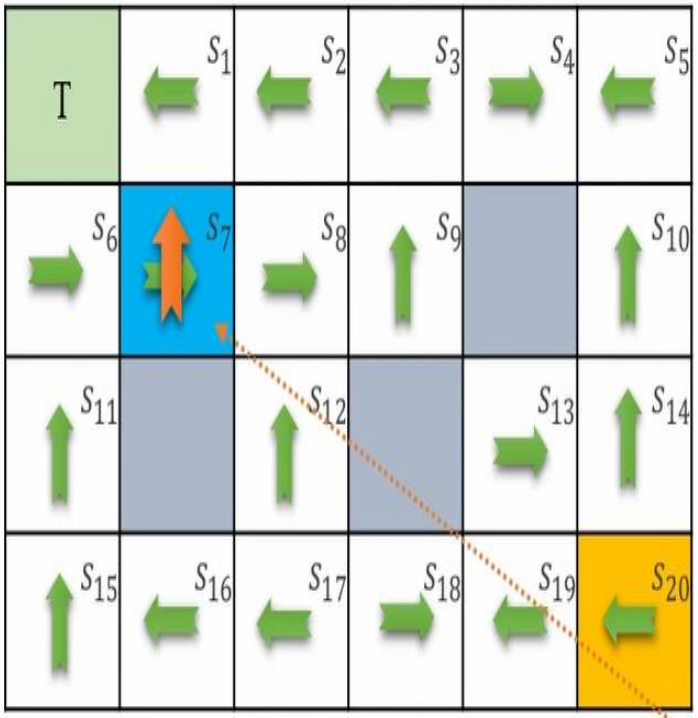
Returns_list(s_8, R)=[62.171, 62.171]

Returns_list(s_{12}, U)=[54.953]

Returns_list(s_7, U)=[89]

e) Update the Policy

$$\pi(s) = argmax_a Q(s,a)$$



Update the Policy

$Q(s_{17}, L)$=[−1]
$Q(s_{12}, D)$=[−1.9]
$Q(s_8, D)$=[−2.71]
$Q(s_7, R)$=[−3.439]
$Q(s_6, R)$=[−4.095]
$Q(s_7, L)$=[−4.685]
$Q(s_1, L)$=[100]
$Q(s_2, L)$=[89]
$Q(s_3, L)$=[79.1]
$Q(s_9, U)$=[70.19]
$Q(s_8, R)$=[62.171]
$Q(s_{12}, U)$=[54.953]
$Q(s_7, U)$=[89]

Average of the list:

Returns_list(s_{17}, L)=[−1]
Returns_list(s_{12}, D)=[−1.9]
Returns_list(s_8, D)=[−2.71]
Returns_list(s_7, R)=[−3.439]
Returns_list(s_6, R)=[−4.095]
Returns_list(s_7, L)=[−4.685]
Returns_list(s_1, L)=[100, 100, **100**]
Returns_list(s_2, L)=[89, 89]
Returns_list(s_3, L)=[79.1, 79.1]
Returns_list(s_9, U)=[70.19, 70.19]
Returns_list(s_8, R)=[62.171, 62.171]
Returns_list(s_{12}, U)=[54.953]
Returns_list(s_7, U)=[89]

In reinforcement learning (RL), **exploration** means that the agent **tries different actions**, even if they are not currently believed to be the best ones, so it can **gather more information** about the environment.

Without exploration, the agent might:

- Keep repeating the same actions (**exploitation**),
- Miss discovering better actions,
- And get stuck in a **suboptimal policy**.

So — exploration ensures **all actions in all states** get some chance to be tried.

Monte Carlo Control with Exploring Starts (MC-ES)

MC-ES enforces exploration **artificially** by assuming that:

- Every episode starts from a **randomly chosen state–action pair** (a random start).
- At the **beginning of each episode**, the environment picks **any (s, a)** pair with **nonzero probability**.
- This ensures that **every possible state–action pair** will **eventually be visited**.

Advantage:

- Guarantees **complete exploration** — the algorithm will eventually sample returns for every (s, a) pair.
- This makes learning of **Q(s, a)** for all pairs possible.

Limitations:

- Not realistic in most real-world problems — we **can't actually start** from *any* (s, a) we like (e.g., a robot can't start in mid-air or teleport to any state).

Monte Carlo Control without Exploring Starts

- Since starting randomly from any (s, a) isn't practical, we use a **different way** to ensure exploration:
- Use a **stochastic policy**, such as:
  - **ε-soft policy** (or **ε-greedy** policy)
- With **probability (1 – ε)**, choose the **greedy action** (best known so far).
- With **probability ε**, choose a **random action** (to explore).
- So exploration happens **naturally** during episodes, not through random starting states.

Advantage:

- **More practical** — works with any environment where the agent starts normally.
- Guarantees that **every action** has a **nonzero probability** of being chosen eventually.

Limitations:

- Exploration is **slower** and **less systematic** than in MC-ES.
- The quality of learning depends on the value of **ε** (too high → too random; too low → not enough exploration).

	MC with Exploring Starts	**MC without Exploring Starts**
Exploration	Explicitly forces exploration via starts	Achieved via ε-soft policy
Feasibility	Requires control over initial states	Works in real environments
Convergence	Theoretical guarantee	Empirical but effective
Example Algorithm	MC-ES	ε-soft Monte Carlo control

Issues with Monte Carlo Control with Exploring Starts (MC-ES)

1. **Averaging Without Policy Dependence:** In MC-ES, the returns for each state–action pair are averaged **without explicitly considering the policy** under which those returns were collected. This makes the learning process **less directed and slower**.

2. **Slow Convergence:** Although MC-ES can theoretically converge to the **optimal policy and optimal value function**, in practice the convergence is **slow**, especially when state–action spaces are large or when some actions are rarely explored.

3. **Lack of Formal Proof:** There is **no rigorous formal proof** guaranteeing convergence of MC-ES under all conditions. It is largely supported by intuition and empirical results.

Monte Carlo Control with Exploring Starts **usually converges in practice**, but **no universal mathematical proof** exists showing that it will **always** converge to the optimal policy in every case. Its reliability is **based on intuition, empirical testing, and experience**, rather than complete theoretical proof.

Issues with Monte Carlo Control with Exploring Starts (MC-ES) (Cont'd)

4. Dependence on Policy Improvement: The algorithm assumes that if it gets trapped in a **suboptimal policy**, the **value function estimates** will eventually cause the policy to change and continue improving. However, this assumption may not always hold.

5. Stability Condition: The algorithm becomes **stable** only when both the **policy** and **value function** have reached their **optimal (best possible) state**, meaning neither can improve further.

Monte Carlo Control without Exploring Starts

- Monte Carlo (MC) Control methods are used in **Reinforcement Learning** to find the **optimal policy** and **optimal action-value function** based on sampled episodes.
- The traditional **MC Control with Exploring Starts (MC-ES)** requires that *every state–action pair has a nonzero probability of being selected as a starting point*. However, this assumption is often **impractical** in real-world problems.
- Hence, **Monte Carlo Control without Exploring Starts** was introduced to remove this unrealistic assumption.

Monte Carlo Control without Exploring Starts

The **Exploring Starts** assumption ensures all state–action pairs are explored, but:

•It is **not always possible** to start an episode from any arbitrary state–action pair.

•It limits the applicability of MC methods to **synthetic or controlled environments**.

- To overcome this, **soft or ε-greedy policies** are used to ensure **adequate exploration** without relying on arbitrary start states.
- Instead of exploring starts, the agent uses a **soft policy** (e.g., ε-greedy) during learning.

•The policy chooses:

  - The **best action** (according to the current Q-values) most of the time.
  - A **random action** with small probability (ε) to ensure exploration.

This ensures that **all actions** in **all states** continue to be explored, satisfying the **exploring requirement** in a more realistic way.

Monte Carlo Control without Exploring Starts

Goal: Find the optimal policy and action-value function using ε-soft (exploratory) policies.

Initialize:

- Q(s, a) arbitrarily
- π(s) as an ε-soft policy (e.g., ε-greedy w.r.t Q)

Repeat for each episode:

1. **Generate an episode** following the current ε-soft policy π.
2. **Compute returns (G)** for each state–action pair in the episode.
3. **Update Q-values:**

$$Q(s,a) \leftarrow Q(s,a) + \alpha[G - Q(s,a)]$$

4. **Improve the policy:**

   Make the policy greedy with respect to the updated Q-values while keeping it ε-soft:

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

Monte Carlo Control without Exploring Starts

5. Convergence

The **ε-soft Monte Carlo control algorithm** converges to the **optimal policy** (π^*) and **optimal Q-values** (Q^*) provided that:

- All state–action pairs continue to be explored, and
- The learning rate (α) satisfies certain conditions.

Advantages

- Removes the unrealistic **exploring starts assumption**
- Works in **practical environments**
- Guarantees **continued exploration** via ε-soft policy
- Can be used in **model-free settings**

On Policy vs Off Policy

In **Reinforcement Learning (RL)**, the distinction between **on-policy** and **off-policy** methods lies in how the **data (experience)** used for learning is generated and which **policy** is being improved.

1. On-Policy Methods

- On-policy methods **evaluate or improve the same policy** that is used to make decisions and generate data through interaction with the environment.
- The agent **follows a single policy** — it both **acts according to this policy** and **updates/improves it** based on the feedback (rewards) received. Thus, the behavior policy (used for exploration) and the target policy (being optimized) are **the same**.

•Example:

Imagine you are exploring restaurants in your city to find the best one.
You **try different restaurants yourself (generate data)** and **use your own experience** to decide which one is better next time.
→You are **evaluating and improving your own decisions** — that's on-policy learning.

•Usage:

On-policy methods can be applied in both **model-based** and **model-free** reinforcement learning.

•Common Algorithms:

- SARSA (State–Action–Reward–State–Action)
- Policy Gradient methods

2. Off-Policy Methods

- Off-policy methods **evaluate or improve a policy different** from the one used to generate the data.
- There are two distinct policies:
  - **Behavior policy:** used to **collect data** and **explore the environment**.
  - **Target policy:** the policy being **learned or optimized**.

•The agent learns from experiences generated by the behavior policy, but **tries to optimize a different (target) policy**.

•**Example:**

Suppose you rely on **Google's restaurant recommendations** (behavior policy) to explore options, but **you ultimately choose your favorite restaurant** based on what you've learned (target policy).

→ You are learning from **someone else's suggestions** while optimizing **your own choices** — that's off-policy learning.

•**Usage:**

Off-policy methods are mainly used in **model-free** reinforcement learning.

•**Common Algorithms:**

- Q-Learning
- Deep Q-Networks (DQN)

Monte Carlo Control with Epsilon-soft

- **On-policy methods**
  - attempt to evaluate or improve the policy that is used to make decisions
- In on-policy control methods the policy is generally soft
  - $\Pi(a|s) > 0$ for all $s \epsilon S$ and all $a \epsilon A(s)$
  - but gradually shifted closer and closer to a deterministic optimal policy
- **ε- greedy policies**
  - most of the time they choose an action that has maximal estimated action value

On-policy First-visit MC control

- **On-policy**: The policy being learned (target policy) is the same policy that is used to generate episodes.
- **First-visit MC**: The algorithm updates the value function only the first time a state-action pair is encountered in an episode.
- **ε-soft policy**: A policy that ensures all actions have a non-zero probability of being selected, facilitating **exploration**.

On-policy First-visit MC control

On-policy first-visit MC control (for ε-soft policies), estimates $\pi \approx \pi_*$

Algorithm parameter: small $\varepsilon > 0$

Initialize:

$\pi \leftarrow$ an arbitrary ε-soft policy

$Q(s,a) \in \mathbb{R}$ (arbitrarily), for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$

$Returns(s,a) \leftarrow$ empty list, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$

Repeat forever (for each episode):

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

$G \leftarrow 0$

Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:

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

Unless the pair S_t, A_t appears in $S_0, A_0, S_1, A_1 \ldots, S_{t-1}, A_{t-1}$:

Append G to $Returns(S_t, A_t)$

$Q(S_t, A_t) \leftarrow \text{average}(Returns(S_t, A_t))$

$A^* \leftarrow \arg\max_a Q(S_t, a)$ (with ties broken arbitrarily)

For all $a \in \mathcal{A}(S_t)$:

$$\pi(a|S_t) \leftarrow \begin{cases} 1 - \varepsilon + \varepsilon/|\mathcal{A}(S_t)| & \text{if } a = A^* \\ \varepsilon/|\mathcal{A}(S_t)| & \text{if } a \neq A^* \end{cases}$$

//Non-greedyaction with the minimal probability of selection

Algorithm parameter: small $\varepsilon > 0$

This ϵ controls the amount of exploration in the policy. It ensures that the policy is ϵ-soft, meaning every action has at least a small probability $\frac{\epsilon}{|A(s)|}$ of being chosen.

$$\text{For all } a \in \mathcal{A}(S_t):$$
$$\pi(a|S_t) \leftarrow \begin{cases} 1 - \varepsilon + \varepsilon/|\mathcal{A}(S_t)| & \text{if } a = A^* \\ \varepsilon/|\mathcal{A}(S_t)| & \text{if } a \neq A^* \end{cases}$$

- **If a=A* (the greedy action):**

$$\pi(a|S_t) \leftarrow 1 - \epsilon + \frac{\epsilon}{|\mathcal{A}(S_t)|}$$

This means the policy assigns a high probability of selecting the greedy action, but still maintains some probability of choosing other actions.

- **If a not equal to A* (non-greedy actions)**

$$\pi(a|S_t) \leftarrow \frac{\epsilon}{|\mathcal{A}(S_t)|}$$

This ensures that each non-greedy action is chosen with a small probability $\frac{\epsilon}{|A(s)|}$, encouraging exploration.

Off-policy Monte Carlo Prediction

Off-policy MC prediction (policy evaluation) for estimating $Q \approx q_\pi$

Input: an arbitrary target policy π

Initialize, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$:

$Q(s,a) \in \mathbb{R}$ (arbitrarily)

$C(s,a) \leftarrow 0$

Loop forever (for each episode):

$b \leftarrow$ any policy with coverage of π

Generate an episode following b: $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$

$G \leftarrow 0$

$W \leftarrow 1$

Loop for each step of episode, $t = T-1, T-2, \ldots, 0$, while $W \neq 0$:

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

$C(S_t, A_t) \leftarrow C(S_t, A_t) + W$

$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{W}{C(S_t, A_t)} [G - Q(S_t, A_t)]$

$W \leftarrow W \frac{\pi(A_t|S_t)}{b(A_t|S_t)}$

Incremental Implementation

- MC can be implemented incrementally
  - saves memory
- Compute the weighted average of each return

$$V_n = \frac{\sum_{k=1}^{n} w_k R_k}{\sum_{k=1}^{n} w_k}$$

non-incremental

$$V_{n+1} = V_n + \frac{w_{n+1}}{W_{n+1}}\left[R_{n+1} - V_n\right]$$

$$W_{n+1} = W_n + w_{n+1}$$

$$V_0 = W_0 = 0$$

incremental equivalent

Off-policy Monte Carlo Control

Off-policy MC control, for estimating $\pi \approx \pi_*$

Initialize, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$:

$Q(s,a) \in \mathbb{R}$ (arbitrarily)

$C(s,a) \leftarrow 0$

$\pi(s) \leftarrow \operatorname{argmax}_a Q(s,a)$ (with ties broken consistently)

Loop forever (for each episode):

$b \leftarrow$ any soft policy

Generate an episode using b: $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$

$G \leftarrow 0$

$W \leftarrow 1$

Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:

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

$C(S_t, A_t) \leftarrow C(S_t, A_t) + W$

$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{W}{C(S_t, A_t)} [G - Q(S_t, A_t)]$

$\pi(S_t) \leftarrow \operatorname{argmax}_a Q(S_t, a)$ (with ties broken consistently)

If $A_t \neq \pi(S_t)$ then exit inner Loop (proceed to next episode)

$W \leftarrow W \frac{1}{b(A_t|S_t)}$

Temporal Difference Learning

Unit II

Monte Carlo Methods: Monte Carlo prediction, Monte Carlo estimation of action values, Monte Carlo control, Monte Carlo Control without exploring starts, Off-policy prediction via importance sampling, incremental implementation, off-policy Monte Carlo Control

Temporal Difference Learning: TD prediction, Advantages of TD Prediction Methods, Optimality of TD(0), Sarsa: On-Policy TD Control, Q-learning: Off-policy TD Control, Expected Sarsa, Maximization Bias and Double Learning.

Tutorials:

Implementing Monte Carlo control algorithm in Python and applying it to a simple environment.

Implementing off-policy prediction using importance sampling in Python and analysing its effectiveness.

Implementing TD prediction algorithm in Python and applying it to a simple environment.

Purpose of studying three methods:

- All three methods — **DP**, **MC**, and **TD** — are **ways to estimate the value functions** and **improve policies** in Reinforcement Learning.
- They form the **core foundation** of how an agent learns from interaction with its environment.
- We study them because they represent **three fundamental approaches** to solving the **same RL problem**, but under **different assumptions** and **data availability conditions**.

Dynamic Programming

- Used when the model of the environment is known (you know all state transitions and rewards).
- It helps to understand the structure of RL problems and the concept of Bellman equations.
- Forms the theoretical basis of RL — everything else builds on this foundation.

Monte Carlo Method

- Used when the model is unknown, but you can simulate or experience complete episodes.
- It learns from actual experience without requiring knowledge of transition probabilities.
- It estimates values as the average of observed returns.

Temporal Difference

- Also model-free, like MC. But unlike MC, it can learn incomplete episodes by updating after each step — bootstrapping from the current value estimate.
- More practical for continuous tasks or infinite-horizon problems

- In RL, a **model** means **the agent's understanding of the environment's dynamics**

- **State transition function:**

  $P(s'|s,a)$ → probability of reaching next state s' when taking action a in state s.

- **Reward function:**

  $R(s,a)$ → expected reward for taking action a in state s.

- If the agent **knows or learns** these two things, it's said to have a **model of the environment**

<u>Model-Based Reinforcement Learning</u>

Model-based RL **uses or builds** a model of the environment to plan and make decisions.

It can:

- Predict **what will happen next**, and
- Choose actions by simulating possible futures before acting.

Model-Free Reinforcement Learning

- Model-free RL **does not know or learn** the transition or reward model.
- It **learns directly from trial-and-error experience** with the environment.
- The agent interacts with the environment:

$$s_t \xrightarrow{a_t} r_{t+1}, s_{t+1}$$

and directly updates:

- **Value functions** (how good a state or action is), or
- **Policy** (which action to take), using experience samples.

•**Model-Based RL:** Learns or knows how the world works → *plans before acting.*
•**Model-Free RL:** Doesn't care how the world works → *learns what works by experience.*
•**TD learning, Q-learning, SARSA** → all **model-free**.
•**Dynamic Programming, Dyna-Q** → **model-based**.

Temporal Difference

Temporal Difference (TD) learning is a **model-free reinforcement learning** technique that learns directly from **experience** (interacting with the environment) — without knowing the transition probabilities or rewards beforehand.

Goal:

It estimates the **value function** (how good it is to be in a particular state) using information from consecutive time steps.

- "Temporal" → refers to **time steps** (successive moments of experience).
- "Difference" → refers to the **change or error** between the predicted value of the current state and the estimated value of the next state.
- This difference is used to update the current estimate.

This is the **TD error**, denoted as:

$$\delta_t = R_{t+1} + \gamma V(S_{t+1}) - V(S_t)$$

- R_{t+1}: reward received after taking an action
- $V(S_t)$: current estimate of state value
- $V(S_{t+1})$: next state's estimated value
- γ: discount factor

The learning update rule is:

$$V(S_t) \leftarrow V(S_t) + \alpha\, \delta_t$$

Difference used in learning process

- The **TD error** (δ_t) acts as a signal that tells how much the agent's prediction was off.
- If the next state was **better than expected**, the TD error is **positive**, and the value of S_t increases.
- If it was **worse than expected**, the TD error is **negative**, and the value decreases.

So, TD = Monte Carlo + DP advantages:

•Like **MC**, TD doesn't need to know the environment's dynamics — it's **model-free**.

•Like **DP**, TD updates its estimates based on **bootstrapping** — using the next state's *estimated* value instead of waiting for a final return.

•TD methods learn from **actual interactions** — sequences of states, actions, and rewards — rather than from a model of the environment. It can start learning immediately after each step, without waiting for an episode to finish.

•Unlike Monte Carlo, which waits until the **end of an episode** to calculate the complete return, TD learning updates **after every step**, using the **current estimate** of the next state's value.
This process of using "an estimate to update another estimate" is called **bootstrapping**.

Feature	Monte Carlo (MC)	Dynamic Programming (DP)	Temporal Difference (TD)
Requires model?	❌ No	✅ Yes	❌ No
Updates after full episode?	✅ Yes	❌ No	❌ No
Uses bootstrapping (estimates from other estimates)?	❌ No	✅ Yes	✅ Yes

Use a **simple 3-state world**: S1 → S2 → S3 (terminal)

- Each move gives a reward of **0**,
- Reaching the terminal state **S3** gives **+1** reward.
- Discount factor $\gamma = 1$
- Learning rate $\alpha = 0.5$
- Initial value estimates:

V(S1) = 0.0
V(S2) = 0.0
V(S3) = 0.0 (since terminal)

TD(0) Update Formula

$$V(S_t) \leftarrow V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)]$$

🔷 Step 1: From S1 → S2

- $S_t = S1,\ R_{t+1} = 0,\ S_{t+1} = S2$
- Current:
  $V(S1) = 0.0,\ V(S2) = 0.0$
- TD Error:

$$\delta = 0 + 1 \times 0 - 0 = 0$$

- Update:
  $V(S1) = 0 + 0.5 \times 0 = 0.0$

✅ So no change yet.

🔷 Step 2: From S2 → S3 (terminal)

- $S_t = S2,\ R_{t+1} = 1,\ S_{t+1} = S3$
- $V(S3) = 0$ (terminal)
- TD Error:

$$\delta = 1 + 1 \times 0 - 0 = 1$$

- Update:
  $V(S2) = 0 + 0.5 \times 1 = 0.5$

After Episode 1

State	Old V	New V
S1	0.0	0.0
S2	0.0	0.5
S3	0.0	0.0

Next Episode (S1 → S2 → S3 again)

Step 1: From S1 → S2

- $R_{t+1} = 0$
- $V(S2) = 0.5$
- TD Error:

$$\delta = 0 + 1 \times 0.5 - 0 = 0.5$$

- Update:

$$V(S1) = 0 + 0.5 \times 0.5 = 0.25$$

Step 2: From S2 → S3

- Same as before →

$$\delta = 1 - 0.5 = 0.5$$

$$V(S2) = 0.5 + 0.5 \times 0.5 = 0.75$$

⬅ END After Episode 2

State	V(S)
S1	0.25
S2	0.75
S3	0.00

- You can see how the **values propagate backward**:
- S2 learns directly from the terminal reward.
- S1 learns indirectly via S2's updated value.
- This gradual backflow of value is the **core idea of Temporal Difference learning** — improving estimates step-by-step without waiting for the final return.

Difference between TD and MC methods

Monte Carlo (MC)	Temporal Difference (TD)
1. MC learns from experience	1. TD learns from experience (can be limited)
2. MC must wait until end of the episode with complete sequences (to get the returns)	2. TD can learn before knowing the final outcome (in TD methods one needs to wait only one time steps)
3. MC only works for episodic environments	3. TD can work in environments with infinite horizon

Temporal Difference Prediction:

- Temporal Difference (1)
- Temporal Difference (0)
- Temporal Difference (λ)

SARSA: on policy TD control

Q-learning: off policy TD control

Expected SARSA (both on policy or off policy TD control)

- In reinforcement learning, the notation **TD(λ)** refers to a family of Temporal Difference (TD) methods that use different degrees of bootstrapping to update the value function.
- The parameter λ (lambda) controls the extent to which the method combines one-step updates (TD(0)) with multi-step updates (Monte Carlo methods).

Here's a breakdown of **TD(λ)** methods:

- **TD(0)**: This is the simplest case, where λ=0. It updates the value function after every step using information from the next state. It is a one-step method, meaning the update only considers the immediate next state and reward.
- **TD(1)**: This is at the opposite end of the spectrum, where λ=1. TD(1) is equivalent to the Monte Carlo method. It waits until the end of the episode and then uses the total return from the current state to the end of the episode to update the value function. In essence, it is a full, multi-step method, using the entire episode to update the value function.

Temporal Difference TD(1)

- TD (1) is for estimating V_π (value of policy)
- Marks an update to the values at the end of an episode but not limited to.
- Therefore we can update if we stop for any reason or use it in continuing tasks
- Similar to the Monte Carlo, with using the experience to solve the prediction problem but can be online with n-steps.
- It is similar to the MC approaches because it uses G_t

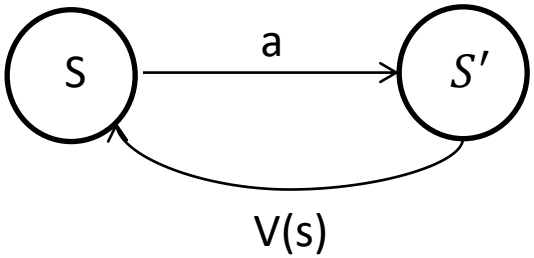
$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots + \gamma^{T-t} R_T$$

- **Use the sum of discounted rewards G_t (from whole episodes) and subtract from the prior estimate (we call it TD error)**

$$V(S_t) \leftarrow V(S_t) + \alpha \Big[G_t - V(S_t) \Big]$$

Temporal Difference TD(0)

- TD(0) is a one-step TD method, meaning it updates the value function after every step using information from the next state.
- TD(0) is for estimating V_π
- **In TD(0) algorithm, we do not need to wait until the end of the episode to update Q-value** (in contrast to MC approaches)
- In TD(0) algorithm we can apply an action and move to the next step then update the value (update the value of the previous state).



- The TD(0) is also known as one-step TD.
- The update rule for value at Temporal Difference (TD(0):

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

Temporal Difference TD(0)

Tabular TD(0) for estimating v_π

Input: the policy π to be evaluated
Algorithm parameter: step size $\alpha \in (0, 1]$
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(\mathit{terminal}) = 0$

Loop for each episode:
    Initialize S
    Loop for each step of episode:
        $A \leftarrow$ action given by π for S
        Take action A, observe R, S'
        $V(S) \leftarrow V(S) + \alpha\big[R + \gamma V(S') - V(S)\big]$
        $S \leftarrow S'$
    until S is terminal

Temporal Difference TD(0)

•TD(0) **blends Monte Carlo and Dynamic Programming ideas**:
•Like **Monte Carlo**, it learns from raw experience (no model of environment needed).
•Like **Dynamic Programming**, it **bootstraps** — updates current value estimates based on other current estimates V(S′).
•It updates after **each step**, not after the whole episode, making it more efficient and incremental.

$$V(S_t) \leftarrow V(S_t) + \alpha\left[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)\right]$$

- $V(S_t)$: current estimate of state value
- $R_{t+1} + \gamma V(S_{t+1})$: target value
- $[\text{Target} - \text{Estimate}]$: TD error

Monte-carlo method and TD method

Monte-carlo method

$$V(S_t) \leftarrow V(S_t) + \alpha\left[G_t - V(S_t)\right]$$

Monte carlo method doesn't allow you to learn in between episode. Values for all the states are updated to the actual outcome.

TD method

$$V(S_t) \leftarrow V(S_t) + \alpha\left[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)\right]$$

One step update, i.e. values are updated based on the next state estimates.

Advantages of TD

- TD methods better than:
- **DP methods** do not require a model of the environment, of its reward and next-state probability distributions
- **Monte Carlo methods**
  - as they can be implemented in an online, fully incremental fashion
  - Need to wait only 1 time step for updation
- both TD and Monte Carlo methods converge asymptotically to the correct predictions

On Policy vs Off Policy

In **Reinforcement Learning (RL)**, the distinction between **on-policy** and **off-policy** methods lies in how the **data (experience)** used for learning is generated and which **policy** is being improved.

1. On-Policy Methods

- On-policy methods **evaluate or improve the same policy** that is used to make decisions and generate data through interaction with the environment.
- The agent **follows a single policy** — it both **acts according to this policy** and **updates/improves it** based on the feedback (rewards) received. Thus, the behavior policy (used for exploration) and the target policy (being optimized) are **the same**.

•Example:

Imagine you are exploring restaurants in your city to find the best one.
You **try different restaurants yourself (generate data)** and **use your own experience** to decide which one is better next time.
→You are **evaluating and improving your own decisions** — that's on-policy learning.

•Usage:

On-policy methods can be applied in both **model-based** and **model-free** reinforcement learning.

•Common Algorithms:

- SARSA (State–Action–Reward–State–Action)
- Policy Gradient methods

2. Off-Policy Methods

- Off-policy methods **evaluate or improve a policy different** from the one used to generate the data.
- There are two distinct policies:
  - **Behavior policy:** used to **collect data** and **explore the environment**.
  - **Target policy:** the policy being **learned or optimized**.

•The agent learns from experiences generated by the behavior policy, but **tries to optimize a different (target) policy**.

•**Example:**

Suppose you rely on **Google's restaurant recommendations** (behavior policy) to explore options, but **you ultimately choose your favorite restaurant** based on what you've learned (target policy).

→ You are learning from **someone else's suggestions** while optimizing **your own choices** — that's off-policy learning.

•**Usage:**

Off-policy methods are mainly used in **model-free** reinforcement learning.

•**Common Algorithms:**

- Q-Learning
- Deep Q-Networks (DQN)

When a Reinforcement Learning Agent is playing a game, it is doing two things

1. taking action – **Behaviour Policy**

2. learning which actions are good and which actions are bad in a given state. Using this learning the agent updates its estimates of Q values. The agent has to use a policy to update its estimate of Q values – **Target Policy**

Behaviour Policy vs Target Policy

Behaviour Policy: Behaviour policy is the policy that the agent uses to determine its action (behaviour) in a given state.

Target Policy: Target policy is the policy that the agent uses to learn from the rewards received for its actions, i.e., to determine updated Q value.

Off Policy vs On Policy Agent (Learner)

If the target policy is different from behaviour policy than the agent (learner) is called **off policy** learner.

If the target policy is same as behaviour policy than the agent (learner) is called **on policy** learner.

Sarsa: On-policy TD Control

SARSA (State Action Reward State Action) Learning

It is a modified Q-learning algorithm where target policy is same as behaviour policy. The two consecutive state-action pairs and the immediate reward received by the agent while transitioning from first state to next state determine the updated Q value, so this method is called SARSA : **S**tate (s) **A**ction *(a)* **R**eward *(r)* **S**tate (s') **A**ction (a'). As target policy is same as behaviour policy, SARSA is an **on policy** learning algorithm.

SARSA stands for **State–Action–Reward–State–Action**, referring to the sequence of events used to update the Q-value.
It's an **on-policy Temporal Difference (TD) control** algorithm — meaning that:

- The same **policy** is used to both **choose actions** and **evaluate/update Q-values**.
- The update depends on the **actual action taken** by the current policy.

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

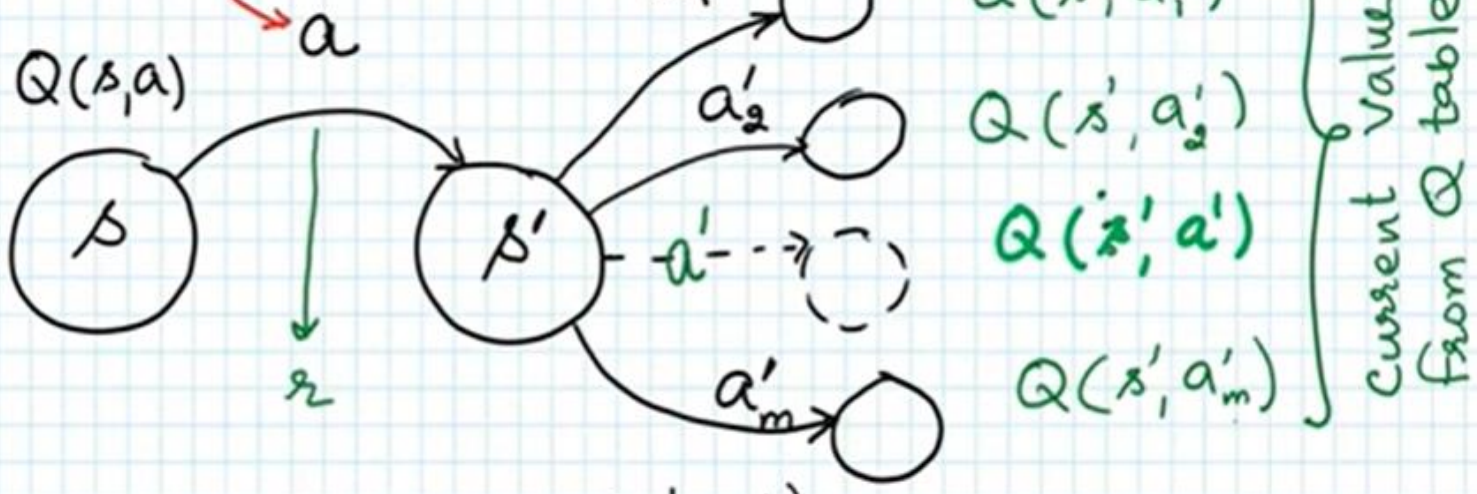
where:

- s = current state
- a = current action (chosen using behavior policy)
- r = reward received after taking action a
- s' = next state
- a' = next action chosen **according to the same policy**
- α = learning rate
- γ = discount factor

SARSA (State Action Reward State Action)

Action is not taken in s' but considered for learnin according to target policy

according to behaviour policy



$$\text{target } Q(s,a) = r + \gamma Q(s', a')$$

target policy is same as behaviour policy

a' = action according to behaviour policy in s'

Sarsa: On-policy TD Control

- The agent is currently at state **s** and takes action **a** (as per the **behavior policy**).
- It then receives a **reward (r)** and transitions to **s'**.
- From **s'**, it chooses an **action a'** (again according to the same policy).
- The update target becomes:

$$Q_{\text{target}}(s, a) = r + \gamma Q(s', a')$$

- The **target policy = behavior policy** (hence SARSA is **on-policy**).

Sarsa: On-policy TD Control

Sarsa (on-policy TD control) for estimating $Q \approx q_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:
  Initialize S
  Choose A from S using policy derived from Q (e.g., ε-greedy)
  Loop for each step of episode:
    Take action A, observe R, S'
    Choose A' from S' using policy derived from Q (e.g., ε-greedy)
    $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma Q(S', A') - Q(S, A)\big]$
    $S \leftarrow S'$; $A \leftarrow A'$;
  until S is terminal

The two consecutive state-action pair and the immediate reward received by the agent while transitioning from first state to next state determine the updated Q-value, this is called as SARSA.

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

Steps followed:

1. **Start at state** s and choose an **action** a using a policy (like ε-greedy).
2. **Take the action**, receive a **reward** r and observe the **next state** s'.
3. From s', choose the **next action** a' using the *same* policy.
4. **Update Q(s, a)** using the rule above.
5. **Move to state** s' and repeat until the episode ends.

🔷 Difference from Q-Learning

Aspect	SARSA	Q-Learning
Policy type	On-policy	Off-policy
Target	$r + \gamma Q(s', a')$ (action from current policy)	$r + \gamma \max_{a'} Q(s', a')$ (greedy action)
Behavior	Learns what the **agent actually does**	Learns what the **agent should ideally do**
Risk	Safer learning (accounts for exploratory actions)	More aggressive (optimistic estimates)

Difference in learning between SARSA and Q-Learning

Aspect	**SARSA (On-policy)**	**Q-Learning (Off-policy)**
Update target	Based on **action actually taken** (including exploratory moves)	Based on **best possible next action** (even if not taken)
Effect	Learns a **cautious** policy — avoids risky edges (like near the cliff) because it experiences those bad outcomes during exploration.	Learns an **optimistic** policy — assumes it will always take the best action, so it may choose paths *close to the cliff* that are risky in reality.
Stability	More stable when random exploration can lead to large penalties	Can be unstable or unsafe if the environment punishes mistakes heavily

Q-Learning: Off Policy TD Control

- One of the early breakthroughs in reinforcement learning was the development of an off-policy TD control algorithm known as Q-learning (Watkins, 1989)
- It learns a specific action/value function, following a specific policy for performing different movements in a different situations.
- Q-learning model consists of an agent, state (s) and set of action for each of the situations.
- Agents goal is to maximize the total reward.
- Q-learning is a off-policy control policy.

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

- The learned action-value function, Q, directly approximates q*, the optimal action-value function, independent of the policy being followed.

Q-learning

Updated Q Value | Current Q Value | Target Q Value | Current Q Value

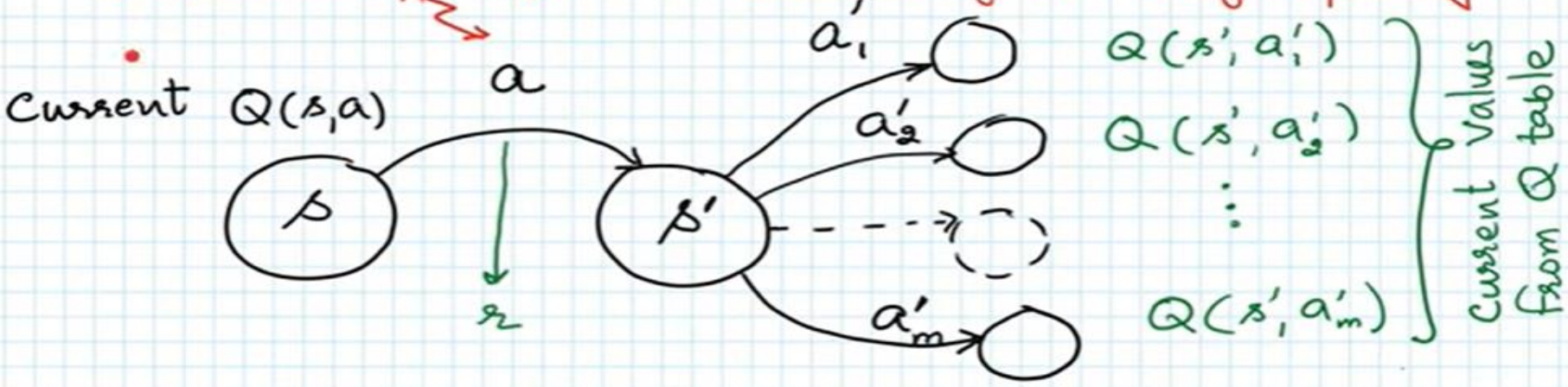
$$Q(s,a) = Q(s,a) + \alpha\left[r + \max_{a'} \gamma Q(s',a') - Q(s,a)\right]$$

Learning Rate

$$0 < \alpha < 1$$

according to behaviour policy

Action is not taken in s' but considered for learnin according to target policy



$$\text{target } Q(s,a) = r + \gamma \max_{a'} Q(s', a')$$

always greedy policy for Q-learning

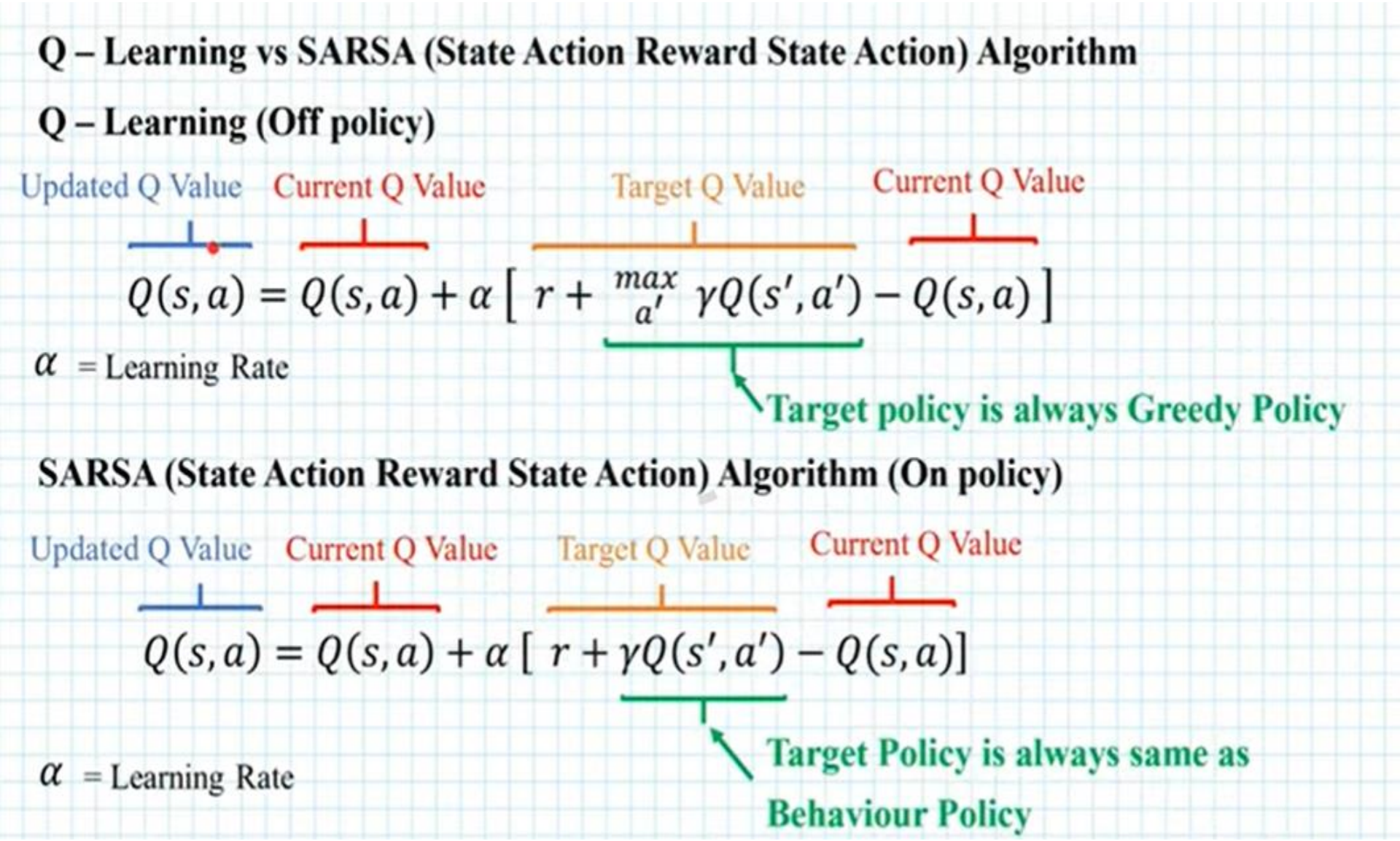
Q-Learning: Off Policy TD Control

Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:
  Initialize S
  Loop for each step of episode:
    Choose A from S using policy derived from Q (e.g., ε-greedy)
    Take action A, observe R, S'
    $Q(S, A) \leftarrow Q(S, A) + \alpha\left[R + \gamma \max_a Q(S', a) - Q(S, A)\right]$
    $S \leftarrow S'$
  until S is terminal

Differences between **SARSA** and **Q-learning (Cont'd)**

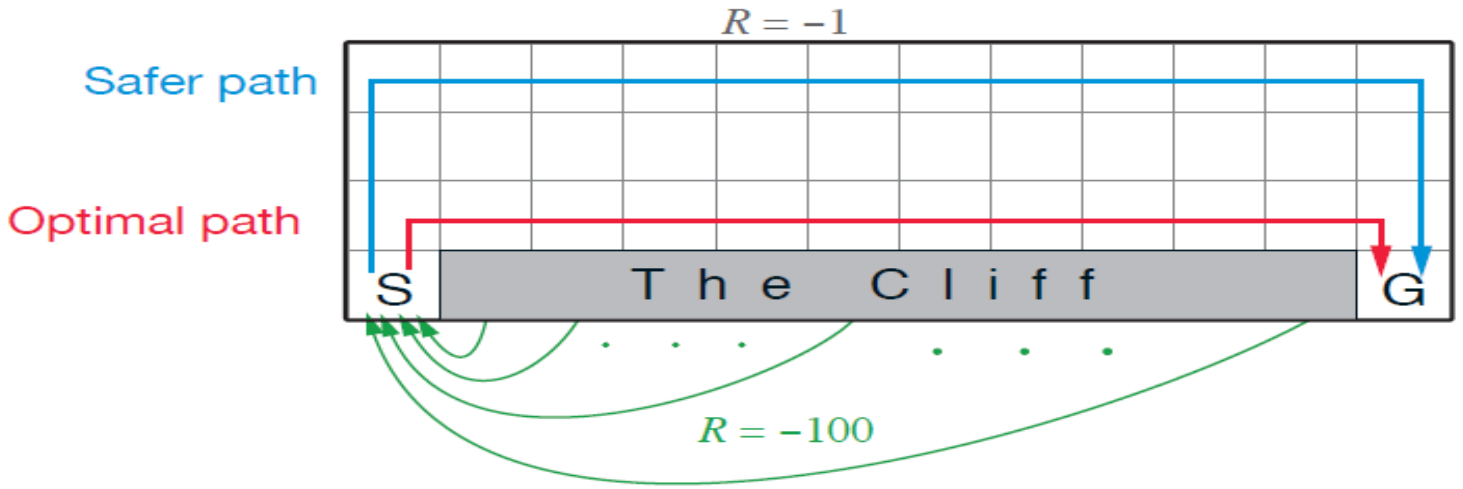


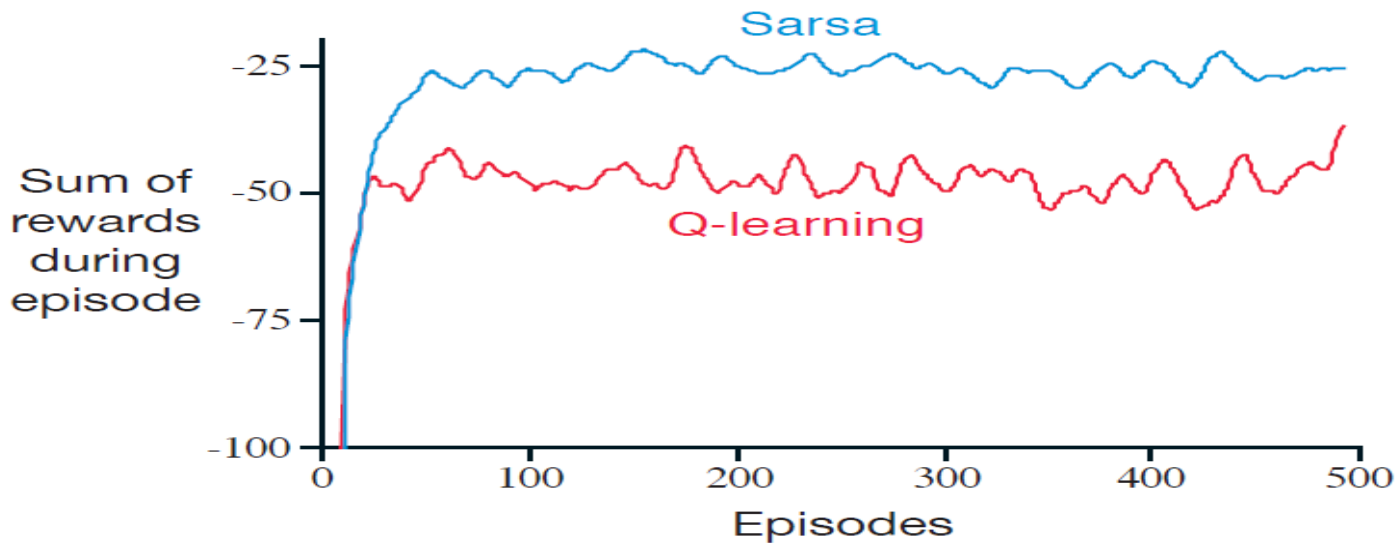
Comparison of QL and SARSA

- Both approach work in
  - finite environment
  - or discretized continuous environment
- QL directly learns the optimal policy while SARSA learns a “near” optimal policy.
- QL is a more aggressive agent, while SARSA is more conservative.
  - Example- In Cliff Walking
  - QL will take the shortest path because it is optimal while SARSA will take the longer, safer route
- In practice
  - For fast-iterating environment, QL should be your choice
  - If mistakes are costly , then SARSA is the better option

Cliff Walking

The cliff wall is of 4x12; goal is to start at S and reach G without falling into the cliff. The reward is -1 for all the transitions and -100 if the agent falls into the cliff and after which an episode terminates.





This gridworld environment is used to compare on-policy (SARSA) and off-policy (Q-learning) learning.

Setup

- The agent starts at **S (Start)** and must reach **G (Goal)**.
- The dark gray region labeled "**The Cliff**" represents dangerous states:
  - Stepping into the cliff gives a **reward of −100** and sends the agent back to the start.
- Every normal step gives a **reward of −1**.
- Objective: **maximize total reward** (i.e., find the path that minimizes penalties).

Paths

- **Red Path (Optimal Path):**
  - Shortest route to the goal, right along the cliff edge.
  - Risky: a small mistake (exploration move) can fall into the cliff (−100).
- **Blue Path (Safer Path):**
  - Slightly longer route, avoids the cliff.
  - More stable and consistent reward, but slightly less optimal in pure reward terms.

The Learning Curves

- **SARSA (blue line)**:
  - Achieves *higher average reward* (closer to −25).
  - Takes the safer path; less variation, more stability.
- **Q-learning (red line)**:
  - Learns a more *optimal but riskier policy*.
  - Slightly lower average reward due to falling into the cliff during exploration.
  - More fluctuations due to risky exploration near the cliff.

(a) SARSA (On-Policy TD Control)

- **On-policy:** learns the value of the *policy being followed* (including exploration actions).
- Update rule:

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

- Here, both A and A' come from the **current ε-greedy policy**.
- As a result, SARSA **learns safely**, accounting for the possibility of risky exploratory actions.

→ **Outcome:**

SARSA learns the *safer path* (blue). It's more conservative because it evaluates the actual exploratory behavior of the policy.

(b) Q-Learning (Off-Policy TD Control)

- **Off-policy:** learns the value of the *optimal greedy policy* while still exploring.
- Update rule:

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

- It assumes the next action will always be the *best possible one* (greedy), regardless of actual exploration.

→ **Outcome:**

Q-learning converges to the *optimal path* (red) that hugs the cliff — because it optimistically assumes the agent will act optimally in the future, not explore and fall.

Key differences between **SARSA** and **Q-learning** in reinforcement learning are based on how they update the Q-values and the exploration-exploitation trade-off.

1. Update Method

- **SARSA (State-Action-Reward-State-Action)**: It is an **on-policy** learning algorithm. SARSA updates the Q-value based on the action taken following the agent's current policy. The update rule is:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left(r + \gamma Q(s',a') - Q(s,a)\right)$$

  Here, $Q(s',a')$ represents the Q-value for the next state-action pair following the current policy.

- **Q-learning**: It is an **off-policy** learning algorithm. Q-learning updates the Q-value based on the maximum possible reward from the next state, regardless of the policy the agent is following. The update rule is:

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

  Here, $\max_{a'} Q(s',a')$ represents the maximum Q-value over all possible actions in the next state, assuming the agent acts optimally.

differences between **SARSA** and **Q-learning (Cont'd)**

2. Policy Type

SARSA: It is an **on-policy** algorithm, meaning it learns the Q-values based on the agent's current policy. The action a' used to update the Q-value is the one chosen by the agent's policy, which could include exploration.

Q-learning: It is an **off-policy** algorithm, meaning it learns the Q-values assuming the agent will act optimally (taking the best possible action) in the future. It updates Q-values using the maximum possible Q-value for the next state, even if the agent is exploring.

3. Exploration-Exploitation Behavior

SARSA: Since it updates based on the actual action taken, SARSA is more sensitive to the exploration-exploitation trade-off. If the agent is exploring (e.g., **using an ϵ-greedy policy**), the Q-values reflect this exploratory behavior.

Q-learning: Q-learning assumes the agent is always exploiting (choosing the **action with the highest Q-value**) when updating, even though it might still be exploring. This can lead to more aggressive exploitation of learned policies.

differences between **SARSA** and **Q-learning (Cont'd)**

4. Convergence Behavior

SARSA: Since it updates based on the current policy, it tends to converge to a **safer** policy. It accounts for exploration during learning, which can lead to more conservative behavior, especially in environments where exploration leads to suboptimal or risky outcomes.

Q-learning: Since it updates based on the maximum possible reward, Q-learning converges faster to an **optimal policy** in many cases. However, this can result in aggressive behavior if the agent explores poorly.

Differences between **SARSA** and **Q-learning (Cont'd)**

Aspect	SARSA	Q-learning
Learning Type	On-policy	Off-policy
Update Formula	Uses next action from current policy	Uses maximum Q-value for next state
Exploration Sensitivity	Considers exploration during learning	Assumes optimal actions for updates
Convergence	Tends to be safer and more conservative	Often converges faster to optimal policy
Behavior in Risky Environments	Learns safer, exploration-aware policies	Learns more aggressive, exploitation-based policies

Q-learning

Q learning algorithm

For each s, a initialize the table entry $\hat{Q}(s, a)$ to zero.
Observe the current state s
Do forever:

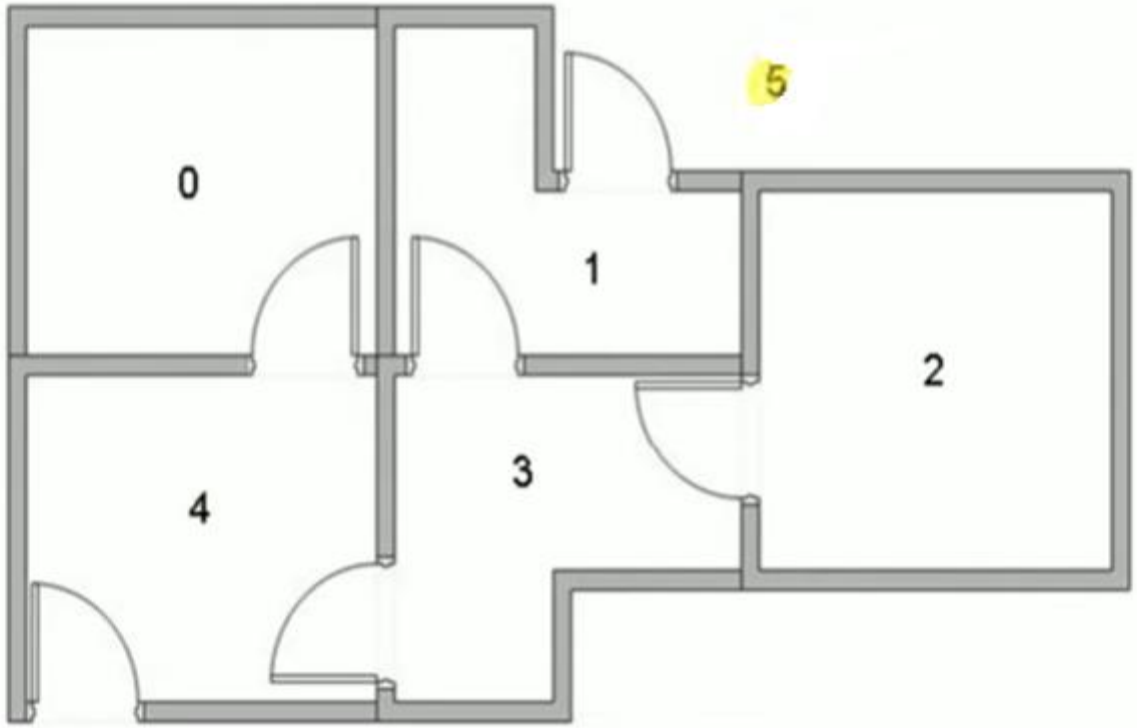
- Select an action a and execute it
- Receive immediate reward r
- Observe the new state s'
- Update the table entry for $\hat{Q}(s, a)$ as follows:

$$\hat{Q}(s, a) \leftarrow r + \gamma \max_{a'} \hat{Q}(s', a')$$

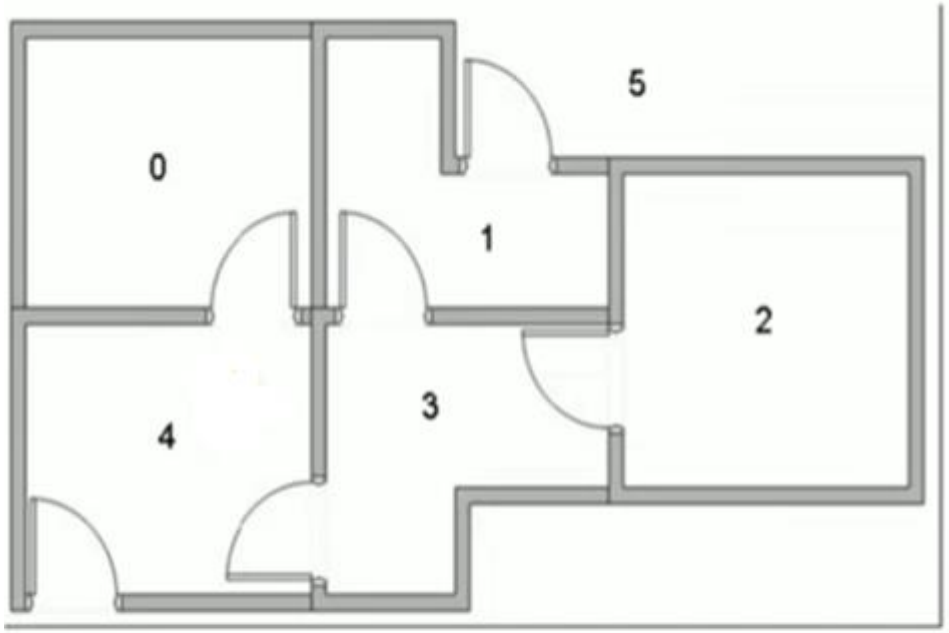
- $s \leftarrow s'$

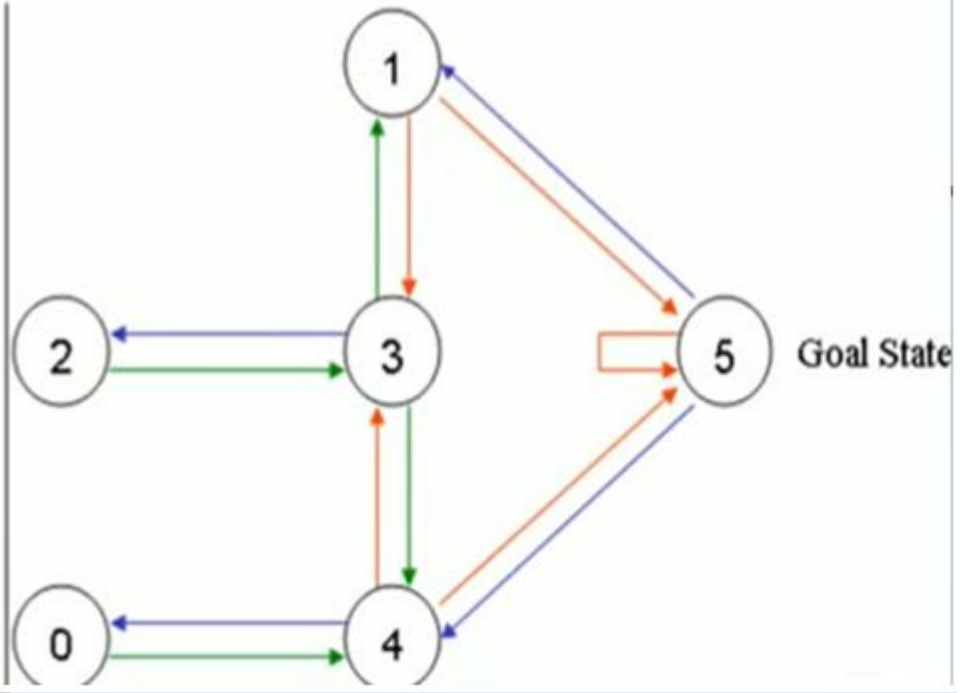
Q-learning: Problems and solution-1

- Suppose we have 5 rooms in a building connected by doors as shown in the figure below. We'll number each room 0 through 4. The outside of the building can be thought of as one big room (5). Notice that doors 1 and 4 lead into the building from room 5 (outside).

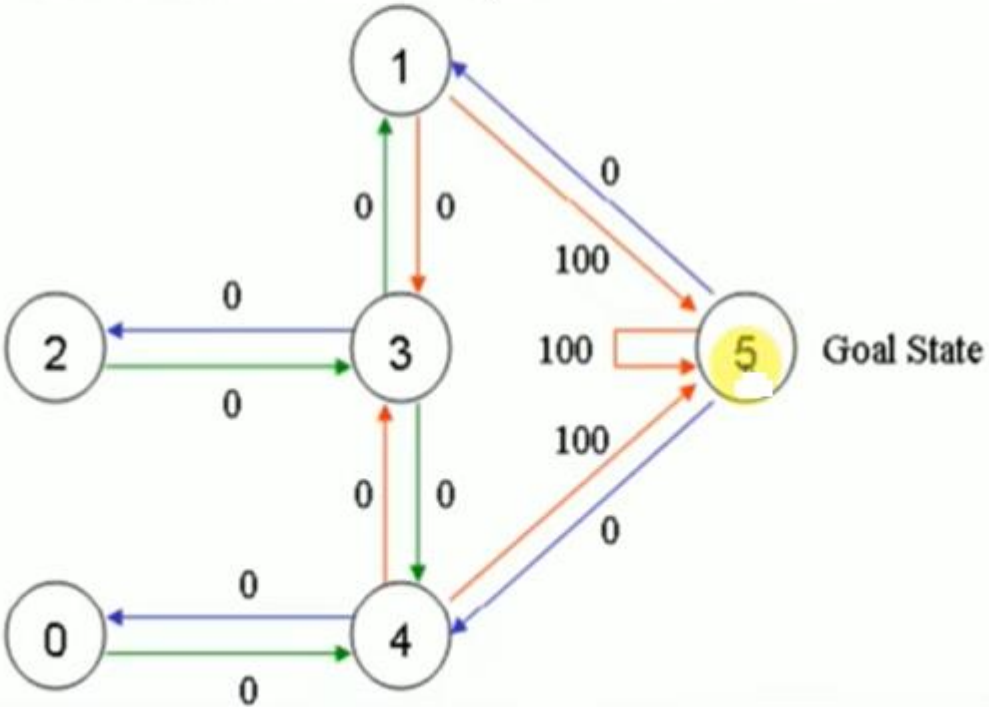


- We can represent the rooms on a graph, each room as a node, and each door as a link.

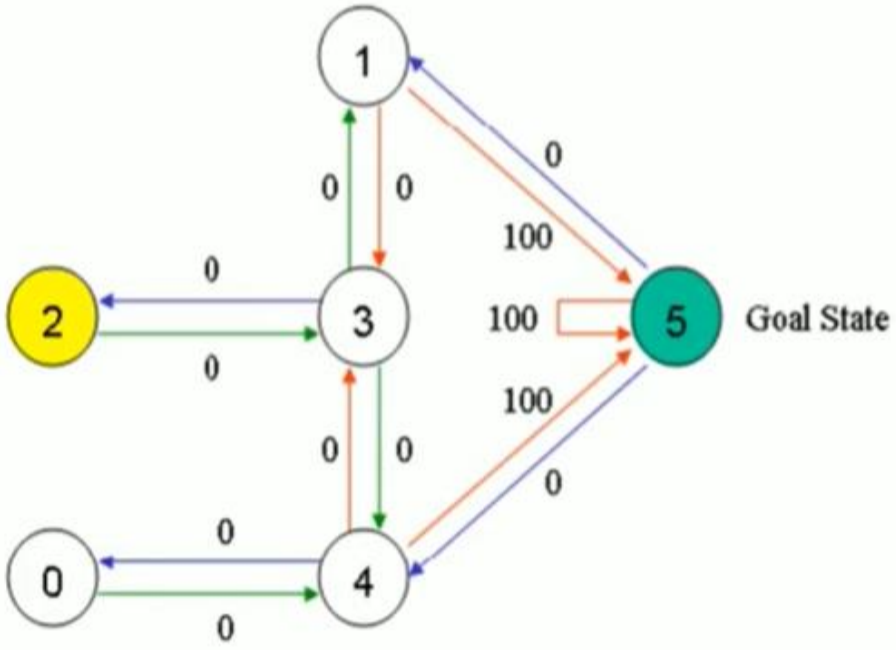




- The goal room is number 5
- The doors that lead immediately to the goal have an instant reward of 100. Other doors not directly connected to the target room have zero reward.
- Each arrow contains an instant reward value, as shown below:



- We can put the state diagram and the instant reward values into the following reward table, "matrix R". The -1's in the table represent null values (i.e.; where there isn't a link between nodes). For example, State 0 cannot go to State 1.

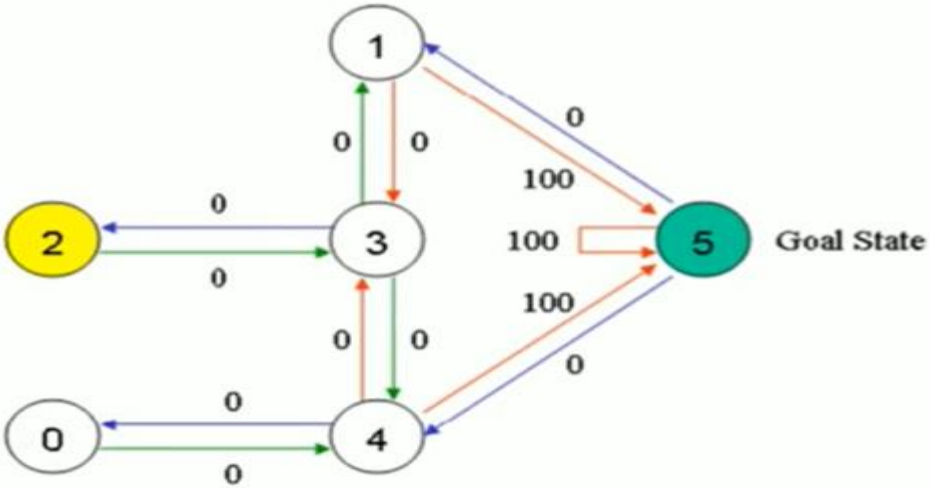


$$R = \begin{array}{c} \text{State} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \text{Action} \\ \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} -1 & -1 & -1 & -1 & 0 & -1 \\ -1 & -1 & -1 & 0 & -1 & 100 \\ -1 & -1 & -1 & 0 & -1 & -1 \\ -1 & 0 & 0 & -1 & 0 & -1 \\ 0 & -1 & -1 & 0 & -1 & 100 \\ -1 & 0 & -1 & -1 & 0 & 100 \end{bmatrix} \end{array}$$

- Learning rate = 0.8 and the initial state as Room 1.
- Initialize matrix Q as a zero matrix:

$$R = \begin{array}{c} \\ \text{State} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cccccc} \multicolumn{6}{c}{\text{Action}} \\ 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} -1 & -1 & -1 & -1 & 0 & -1 \\ -1 & -1 & -1 & 0 & -1 & 100 \\ -1 & -1 & -1 & 0 & -1 & -1 \\ -1 & 0 & 0 & -1 & 0 & -1 \\ 0 & -1 & -1 & 0 & -1 & 100 \\ -1 & 0 & -1 & -1 & 0 & 100 \end{bmatrix} \end{array}$$

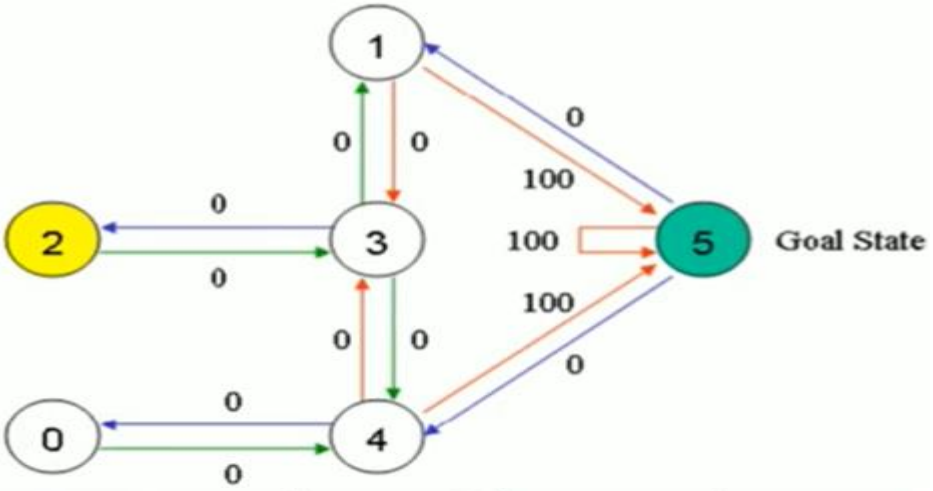
$$Q = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{array}$$



Select the state 1

- Look at the second row (state 1) of matrix R.
- There are two possible actions for the current state 1: go to state 3, or go to state 5.
- By random selection, we select to go to 5 as our action.

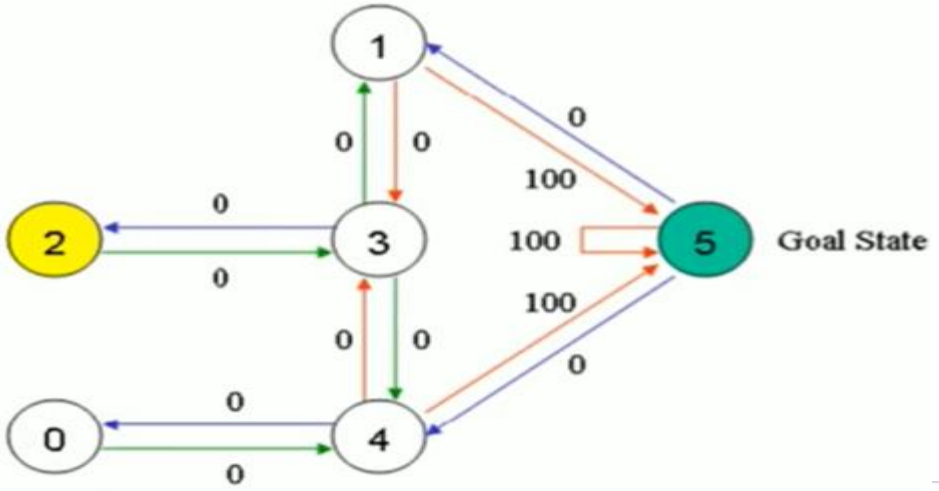
$$R = \begin{array}{c} \\ \text{State} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cccccc} & & & \text{Action} & & \end{array} \\ \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} -1 & -1 & -1 & -1 & 0 & -1 \\ -1 & -1 & -1 & 0 & -1 & 100 \\ -1 & -1 & -1 & 0 & -1 & -1 \\ -1 & 0 & 0 & -1 & 0 & -1 \\ 0 & -1 & -1 & 0 & -1 & 100 \\ -1 & 0 & -1 & -1 & 0 & 100 \end{bmatrix} \end{array}$$



- Now let's imagine what would happen if our agent were in state 5 (next state).
- Look at the sixth row of the reward matrix R (i.e. state 5).
- It has 3 possible actions: go to state 1, 4 or 5.
- Q(state, action) = R(state, action) + Gamma * Max[Q(next state, all actions)]
- Q(1, 5) = R(1, 5) + 0.8 * Max[Q(5, 1), Q(5, 4), Q(5, 5)] = 100 + 0.8 * 0 = 100

$$R = \begin{array}{c} \\ \text{State} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \text{Action} \\ \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} -1 & -1 & -1 & -1 & 0 & -1 \\ -1 & -1 & -1 & 0 & -1 & 100 \\ -1 & -1 & -1 & 0 & -1 & -1 \\ -1 & 0 & 0 & -1 & 0 & -1 \\ 0 & -1 & -1 & 0 & -1 & 100 \\ -1 & 0 & -1 & -1 & 0 & 100 \end{bmatrix} \end{array}$$

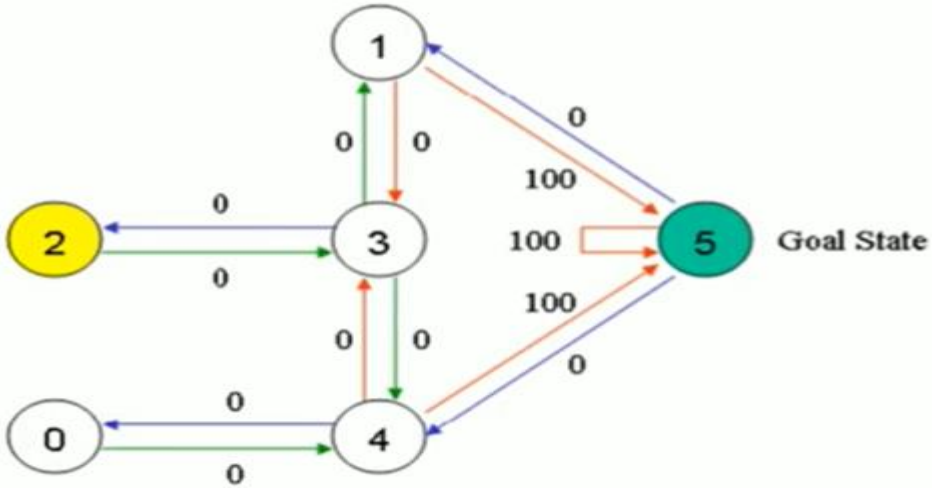
$$Q = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{array}$$



- Now let's imagine what would happen if our agent were in state 5.
- Look at the sixth row of the reward matrix R (i.e. state 5).
- It has 3 possible actions: go to state 1, 4 or 5.
- Q(state, action) = R(state, action) + Gamma * Max[Q(next state, all actions)]
- Q(1, 5) = R(1, 5) + 0.8 * Max[Q(5, 1), Q(5, 4), Q(5, 5)] = 100 + 0.8 * 0 = 100

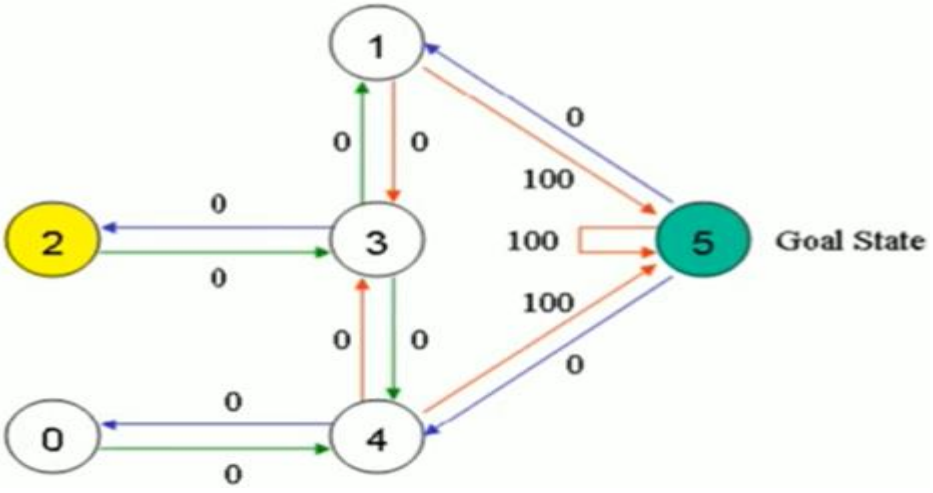
$$R = \begin{array}{c} \\ \text{State} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \text{Action} \\ \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} -1 & -1 & -1 & -1 & 0 & -1 \\ -1 & -1 & -1 & 0 & -1 & 100 \\ -1 & -1 & -1 & 0 & -1 & -1 \\ -1 & 0 & 0 & -1 & 0 & -1 \\ 0 & -1 & -1 & 0 & -1 & 100 \\ -1 & 0 & -1 & -1 & 0 & 100 \end{bmatrix} \end{array}$$

$$Q = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 100 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{array}$$



- The next state, 5, now becomes the current state.
- Because 5 is the goal state, we've finished one episode.

$$Q = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 100 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

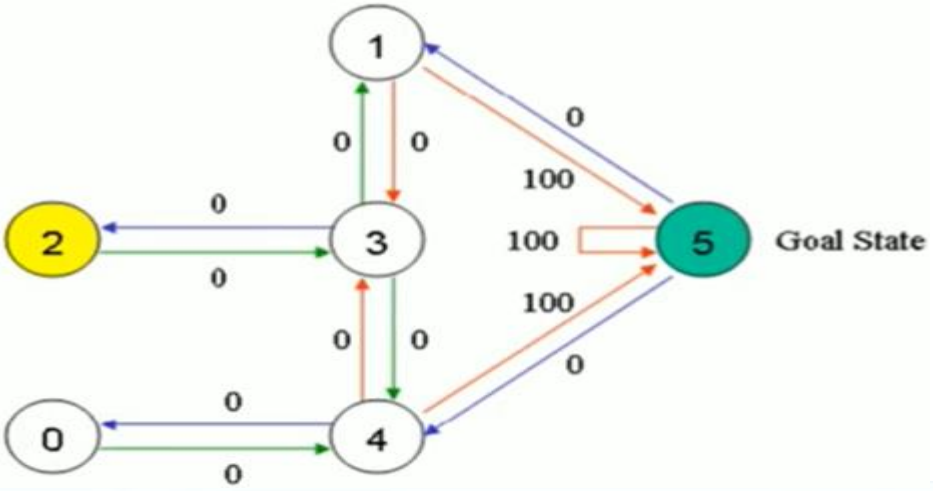


Choose Next Episode

- For the next episode, we randomly choose the initial state – say 3 (can go to 1, 2 & 4)

$$R = \begin{array}{c} \\ \text{State} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cccccc} & & & \text{Action} & & \end{array} \\ \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\ \left[\begin{array}{cccccc} -1 & -1 & -1 & -1 & 0 & -1 \\ -1 & -1 & -1 & 0 & -1 & 100 \\ -1 & -1 & -1 & 0 & -1 & -1 \\ -1 & 0 & 0 & -1 & 0 & -1 \\ 0 & -1 & -1 & 0 & -1 & 100 \\ -1 & 0 & -1 & -1 & 0 & 100 \end{array}\right] \end{array}$$

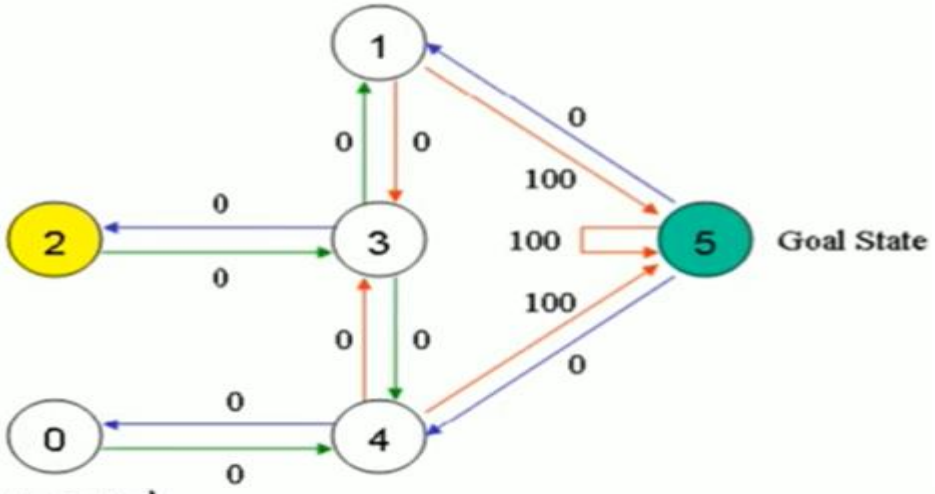
- Now we imagine that we are in state 1 (next state).



- Now we imagine that we are in state 1 (next state).
- Look at the second row of reward matrix R (i.e. state 1).
- It has 2 possible actions: go to state 3 or state 5.
- Then, we compute the Q value:
- Q(state, action) = R(state, action) + Gamma * Max[Q(next state, all actions)]
- Q(3, 1) = R(3, 1) + 0.8 * Max[Q(1, 3), Q(1, 5)] = 0 + 0.8 * Max(0, 100) = 80

$$R = \begin{array}{c} \text{State} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \text{Action} \\ \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} -1 & -1 & -1 & -1 & 0 & -1 \\ -1 & -1 & -1 & 0 & -1 & 100 \\ -1 & -1 & -1 & 0 & -1 & -1 \\ -1 & 0 & 0 & -1 & 0 & -1 \\ 0 & -1 & -1 & 0 & -1 & 100 \\ -1 & 0 & -1 & -1 & 0 & 100 \end{bmatrix} \end{array}$$

$$Q = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 100 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{array}$$



- Now we imagine that we are in state 1 (next state).
- Look at the second row of reward matrix R (i.e. state 1).
- It has 2 possible actions: go to state 3 or state 5.
- Then, we compute the Q value:
- Q(state, action) = R(state, action) + Gamma * Max[Q(next state, all actions)]
- Q(3, 1) = R(3, 1) + 0.8 * Max[Q(1, 3), Q(1, 5)] = 0 + 0.8 * Max(0, 100) = 80

$$R = \begin{array}{c} \text{State} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \text{Action} \\ \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} -1 & -1 & -1 & -1 & 0 & -1 \\ -1 & -1 & -1 & 0 & -1 & 100 \\ -1 & -1 & -1 & 0 & -1 & -1 \\ -1 & 0 & 0 & -1 & 0 & -1 \\ 0 & -1 & -1 & 0 & -1 & 100 \\ -1 & 0 & -1 & -1 & 0 & 100 \end{bmatrix} \end{array}$$

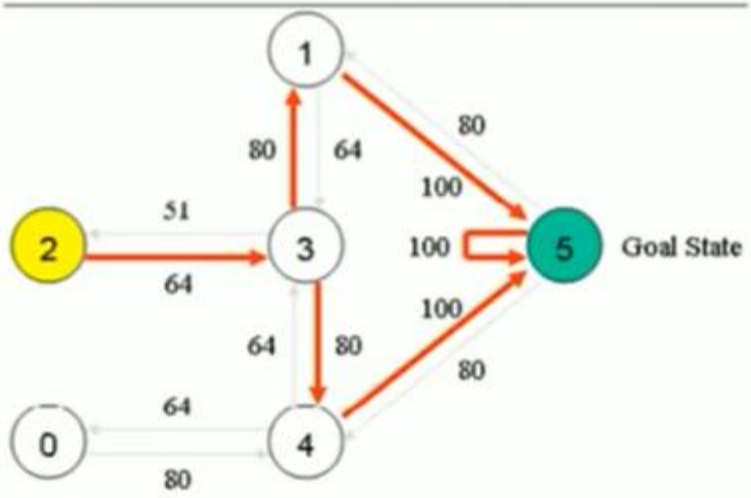
$$Q = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 100 \\ 0 & 80 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

After all iterations

- If our agent learns more through further episodes, it will finally reach convergence values in matrix Q like:

$$Q = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & 0 & 80 & 0 \\ 0 & 0 & 0 & 64 & 0 & 100 \\ 0 & 0 & 0 & 64 & 0 & 0 \\ 0 & 80 & 51 & 0 & 80 & 0 \\ 64 & 0 & 0 & 64 & 0 & 100 \\ 0 & 80 & 0 & 0 & 80 & 100 \end{bmatrix} \end{array}$$

- Tracing the best sequences of states is as simple as following the links with the highest values at each state.



Q-learning

Updated Q Value | Current Q Value | Target Q Value | Current Q Value

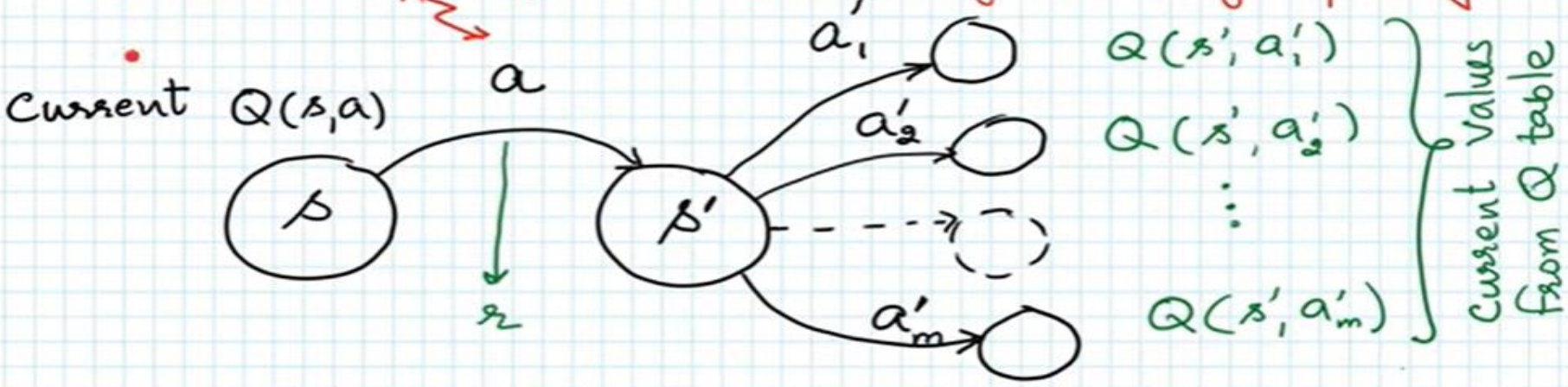
$$Q(s,a) = Q(s,a) + \alpha\left[r + \max_{a'} \gamma Q(s',a') - Q(s,a)\right]$$

Learning Rate

$$0 < \alpha < 1$$

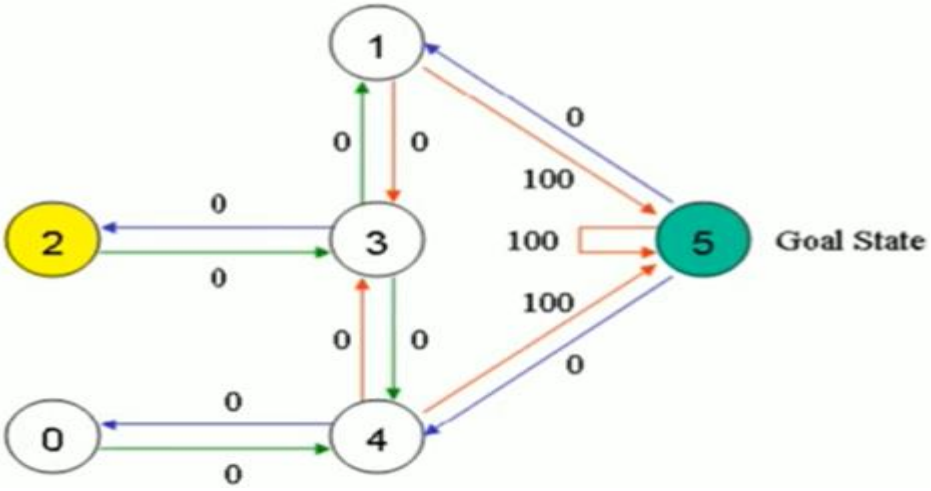
according to behaviour policy

Action is not taken in s' but considered for learnin according to target policy



$$\text{target } Q(s,a) = r + \gamma \max_{a'} Q(s', a')$$

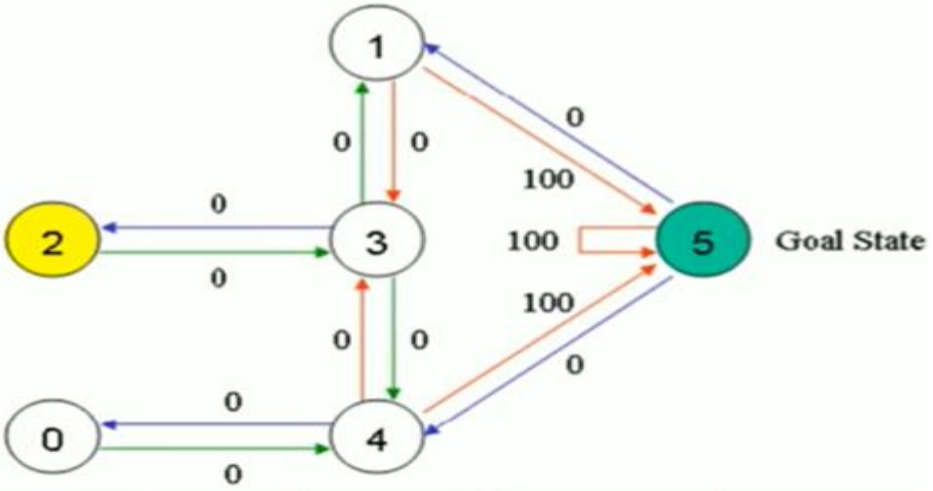
always greedy policy for Q-learning



Select the state 1

- Look at the second row (state 1) of matrix R.
- There are two possible actions for the current state 1: go to state 3, or go to state 5.
- By random selection, we select to go to 5 as our action.

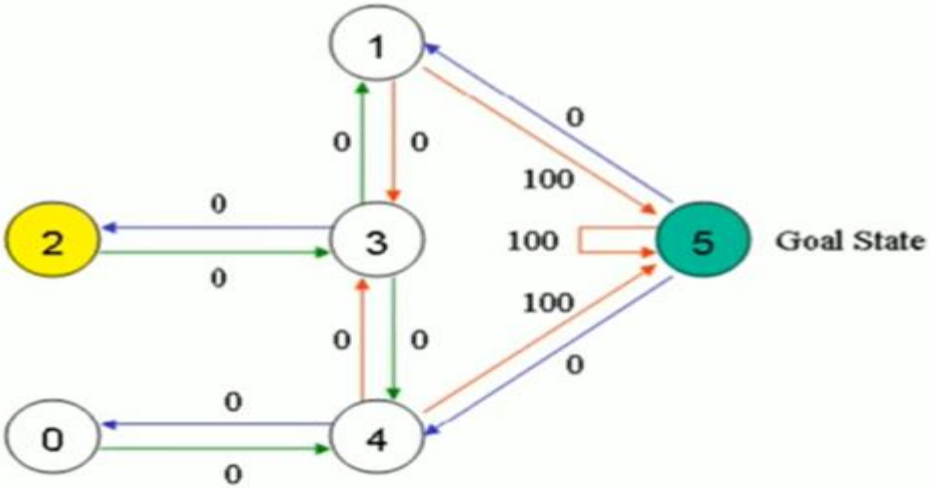
$$R = \begin{array}{c} \\ \text{State} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cccccc} & & & \text{Action} & & \end{array} \\ \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} -1 & -1 & -1 & -1 & 0 & -1 \\ -1 & -1 & -1 & 0 & -1 & 100 \\ -1 & -1 & -1 & 0 & -1 & -1 \\ -1 & 0 & 0 & -1 & 0 & -1 \\ 0 & -1 & -1 & 0 & -1 & 100 \\ -1 & 0 & -1 & -1 & 0 & 100 \end{bmatrix} \end{array}$$



- Now let's imagine what would happen if our agent were in state 5 (next state).
- Look at the sixth row of the reward matrix R (i.e. state 5).
- It has 3 possible actions: go to state 1, 4 or 5.
- Q(state, action) = R(state, action) + Gamma * Max[Q(next state, all actions)]
- Q(1, 5) = R(1, 5) + 0.8 * Max[Q(5, 1), Q(5, 4), Q(5, 5)] = 100 + 0.8 * 0 = 100

$$R = \begin{array}{c} \\ \text{State} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \text{Action} \\ \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} -1 & -1 & -1 & -1 & 0 & -1 \\ -1 & -1 & -1 & 0 & -1 & 100 \\ -1 & -1 & -1 & 0 & -1 & -1 \\ -1 & 0 & 0 & -1 & 0 & -1 \\ 0 & -1 & -1 & 0 & -1 & 100 \\ -1 & 0 & -1 & -1 & 0 & 100 \end{bmatrix} \end{array}$$

$$Q = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

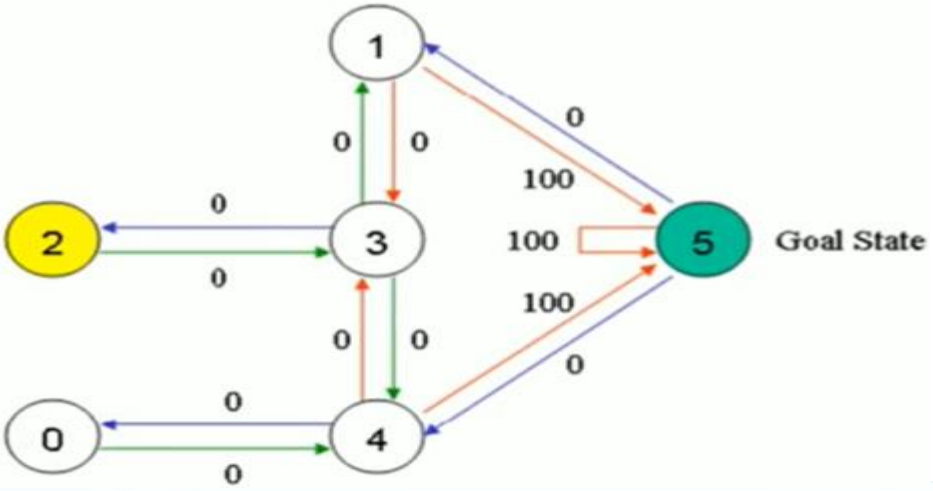


Choose Next Episode

- For the next episode, we randomly choose the initial state – say 3 (can go to 1, 2 & 4)

$$R = \begin{array}{c} \\ \text{State} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cccccc} & & & \text{Action} & & \end{array} \\ \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\ \left[\begin{array}{cccccc} -1 & -1 & -1 & -1 & 0 & -1 \\ -1 & -1 & -1 & 0 & -1 & 100 \\ -1 & -1 & -1 & 0 & -1 & -1 \\ -1 & 0 & 0 & -1 & 0 & -1 \\ 0 & -1 & -1 & 0 & -1 & 100 \\ -1 & 0 & -1 & -1 & 0 & 100 \end{array}\right] \end{array}$$

- Now we imagine that we are in state 1 (next state).



- Now we imagine that we are in state 1 (next state).
- Look at the second row of reward matrix R (i.e. state 1).
- It has 2 possible actions: go to state 3 or state 5.
- Then, we compute the Q value:
- Q(state, action) = R(state, action) + Gamma * Max[Q(next state, all actions)]
- Q(3, 1) = R(3, 1) + 0.8 * Max[Q(1, 3), Q(1, 5)] = 0 + 0.8 * Max(0, 100) = 80

$$R = \begin{array}{c} \text{State} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \text{Action} \\ \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} -1 & -1 & -1 & -1 & 0 & -1 \\ -1 & -1 & -1 & 0 & -1 & 100 \\ -1 & -1 & -1 & 0 & -1 & -1 \\ -1 & 0 & 0 & -1 & 0 & -1 \\ 0 & -1 & -1 & 0 & -1 & 100 \\ -1 & 0 & -1 & -1 & 0 & 100 \end{bmatrix} \end{array}$$

$$Q = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 100 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

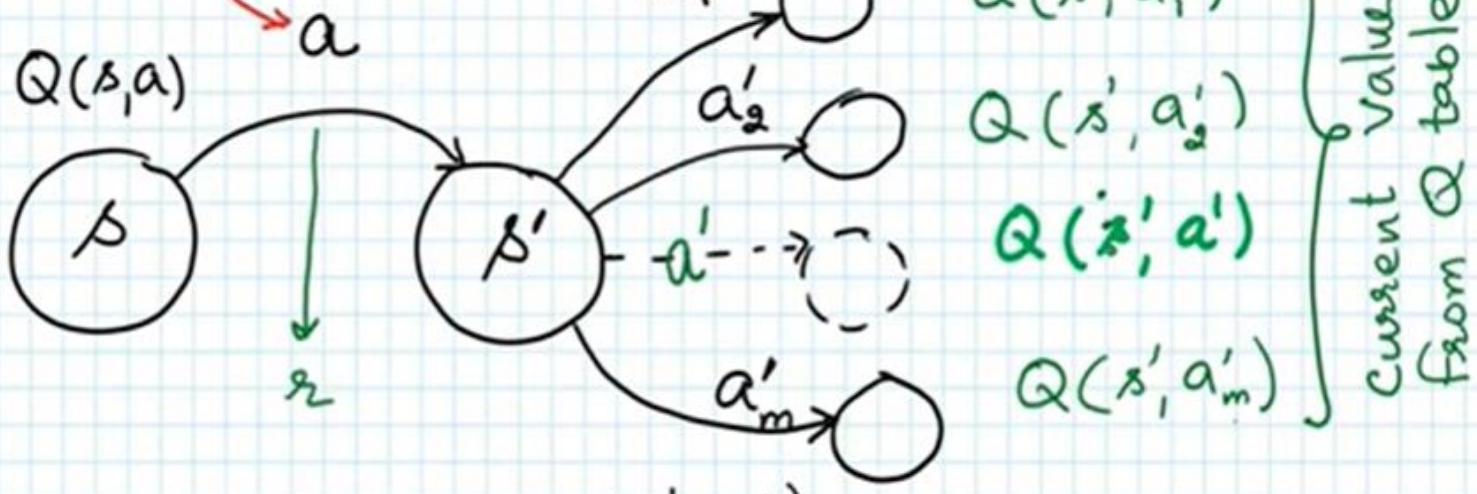
SARSA (State Action Reward State Action) Learning

It is a modified Q-learning algorithm where target policy is same as behaviour policy. The two consecutive state-action pairs and the immediate reward received by the agent while transitioning from first state to next state determine the updated Q value, so this method is called SARSA : **S**tate (s) **A**ction *(a)* **R**eward *(r)* **S**tate (s') **A**ction (a'). As target policy is same as behaviour policy, SARSA is an **on policy** learning algorithm.

SARSA (State Action Reward State Action)

Action is not taken in s' but considered for learnin according to target policy

according to behaviour policy



$$\text{target } Q(s,a) = r + \gamma Q(s', a')$$

target policy is same as behaviour policy

a' = action according to behaviour policy in s'

Q – Learning vs SARSA (State Action Reward State Action) Algorithm

Q – Learning (Off policy)

Updated Q Value | Current Q Value | Target Q Value | Current Q Value

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

α = Learning Rate

Target policy is always Greedy Policy

SARSA (State Action Reward State Action) Algorithm (On policy)

Updated Q Value | Current Q Value | Target Q Value | Current Q Value

$$Q(s,a) = Q(s,a) + \alpha \left[r + \gamma Q(s',a') - Q(s,a) \right]$$

α = Learning Rate

Target Policy is always same as Behaviour Policy

Expected SARSA

- **Expected SARSA** is a variant of the SARSA algorithm used in **Reinforcement Learning** (RL).
- Both SARSA and Expected SARSA are on-policy Temporal Difference (TD) learning algorithms, but Expected SARSA incorporates a more sophisticated approach to **updating the Q-values by considering the expected value over possible future actions**, rather than the Q-value corresponding to the next action chosen by the agent.
- Instead of the maximum over next state–action pairs it uses the expected value, taking into account how likely each action is under the current policy.

Expected SARSA (Cont'd)

- In **Expected SARSA**, instead of using the Q-value of the action a′ actually taken in the next state, we use the expected Q-value over all possible actions in the next state.
- This means we compute the weighted sum of Q-values for all actions in the next state, based on the agent's policy.

The update rule for Expected SARSA is:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \Big[R_{t+1} + \gamma \mathbb{E}_\pi [Q(S_{t+1}, A_{t+1}) \mid S_{t+1}] - Q(S_t, A_t) \Big]$$
$$\leftarrow Q(S_t, A_t) + \alpha \Big[R_{t+1} + \gamma \sum_a \pi(a|S_{t+1}) Q(S_{t+1}, a) - Q(S_t, A_t) \Big],$$

- In simpler terms, instead of using just one action a′ to calculate the TD target as in SARSA, Expected SARSA uses the average (expected) Q-value over all actions, weighted by the probability that each action will be selected according to the current policy π.

Advantages of Expected SARSA

- **Less Variance**: By using the expected value of the next action rather than the actual action taken, Expected SARSA reduces the variance in updates. This can lead to more stable learning.
- **More Robust**: Expected SARSA is often more robust to randomness in the policy, since it considers the average over all actions rather than relying on one specific action.
- **Better Convergence**: In practice, Expected SARSA can sometimes converge faster or more reliably than SARSA due to its smoother updates.

Key Differences Between SARSA and Expected SARSA

SARSA:

- Updates the Q-value based on the Q-value of the next action a′ actually taken in the next state.
- It is a sample-based approach and can be more sensitive to individual actions.

Expected SARSA:

- Updates the Q-value based on the expected value of all possible actions in the next state.
- It provides a smoother update since it averages over all actions rather than using just one.

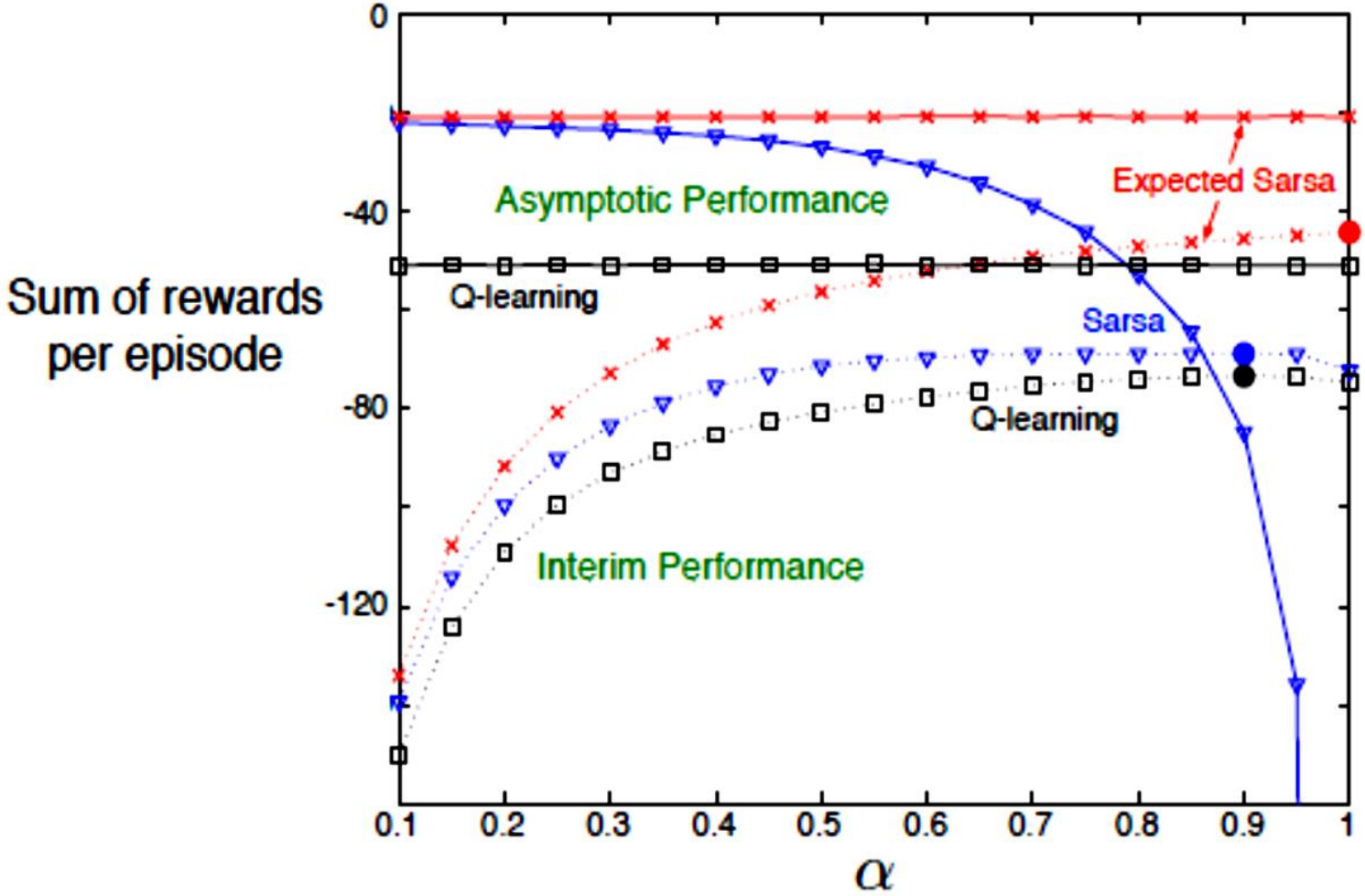


Figure 6.3: Interim and asymptotic performance of TD control methods on the cliff-walking task as a function of α. All algorithms used an ε-greedy policy with $\varepsilon = 0.1$. Asymptotic performance is an average over 100,000 episodes whereas interim performance is an average over the first 100 episodes. These data are averages of over 50,000 and 10 runs for the interim and asymptotic cases respectively. The solid circles mark the best interim performance of each method. Adapted from van Seijen et al. (2009).

When to Use Expected SARSA?

- **When you want lower variance in updates**: If the environment is highly stochastic or the agent's policy involves significant randomness (like an ε-greedy policy with a high ε), Expected SARSA might provide more stable learning.
- **In smooth, stable environments**: Since Expected SARSA takes an average over all actions, it tends to work well in environments where the noise or randomness in action selection is not too high.

Summary

- **Expected SARSA** is an improvement over SARSA in terms of stability and reduced variance. It incorporates the expected value of the Q-function over all possible actions, making it more robust to policy exploration.
- The tradeoff is that Expected SARSA can be computationally more expensive, as it requires calculating the expected value over all actions at each step. However, the smoother updates often lead to better performance in practice.

Maximization Bias and Double Learning

Maximization Bias in Reinforcement Learning refers to the systematic error that arises when a control algorithm, like Q-learning, uses the **maximum of estimated values** as a proxy for the maximum of true values. This error results in an **overestimation** of action values, leading to suboptimal decision-making.

Why Does Maximization Bias Occur?

- Maximization bias happens because of a basic statistical principle: **estimates are inherently uncertain**. When you take the **maximum** of a set of uncertain values, you are more likely to overestimate the true maximum due to this uncertainty.
- Each action a has an estimated value Q(s,a) which is a noisy approximation of the true value q(s,a).
- Since we take the **maximum of these noisy estimates** when selecting the best action, this tends to result in overestimating the true value of the best action.
- This is particularly problematic in states where many actions have similar (or zero) true values because small fluctuations in the estimates can lead to large differences in the chosen maximum.

Maximization Bias and Double Learning (Cont'd)

Imagine this situation:

- **True Q-values** for state s:
  - $q(s, a_1) = 0$
  - $q(s, a_2) = 0$
  - $q(s, a_3) = 0$
- **Estimated Q-values**:
  - $Q(s, a_1) = 0.1$
  - $Q(s, a_2) = -0.2$
  - $Q(s, a_3) = 0.05$

In this case, the agent would choose a_1, since its estimated Q-value (0.1) is the maximum. But the **true value of all actions is 0**, so this selection is biased. The agent thinks it's choosing a good action, but it's really being misled by overestimates in its Q-values.

Maximization Bias and Double Learning (Cont'd)

What Happens Next

Since the agent's goal is to **maximize the future reward**, it chooses the action with the **highest estimated Q-value**, which is a_1 (with $Q(s, a_1) = 0.1$).

But here's the problem:

- The agent chose a_1 because its estimated Q-value is **positive (0.1)**.
- However, the **true value of** a_1 is actually **0**, just like all the other actions (remember, $q(s, a_1) = 0$).

Even though none of the actions are better than the others in reality, the agent **overestimates** the value of a_1 because of the random noise in its learning process. This is an example of **maximization bias**: the agent has picked an action based on a **maximum estimated Q-value**, but that estimate is **biased** (too high) because it comes from a noisy learning process.

Impact of Maximization Bias

- **Overestimation**: Over time, maximization bias can lead to **consistent overestimation** of the values of certain actions.
- **Suboptimal policies**: The agent may repeatedly favor actions that it **incorrectly believes are better** due to this bias.
- **Instability**: In some cases, maximization bias can slow down learning or make the learning process unstable.

Mitigating Maximization Bias

1. Double Q-learning
2. Averaging Q-values
3. Lower learning rates

Double Learning

- One of the most effective methods to reduce maximization bias is **Double Q-learning**.
- Instead of using one Q-value function, we maintain **two Q-value functions**, Q1(s,a) and Q2(s,a), and use them in a way that reduces the bias in updates.
- Specifically, one Q-value function is used to select the action, and the other Q-value function is used to evaluate the value of that action.
- This helps reduce the overestimation that arises from taking the maximum over a single set of Q-values.
- **Double Q-learning** is an improvement over the traditional **Q-learning** algorithm that aims to address the problem of **maximization bias**.
- As discussed earlier, maximization bias occurs in standard Q-learning when an agent overestimates the value of actions because it uses the same set of Q-values both for selecting the best action and for evaluating it. Double Q-learning reduces this bias by using two separate Q-value estimates.

Double Learning

Double Q-learning, for estimating $Q_1 \approx Q_2 \approx q_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q_1(s, a)$ and $Q_2(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, such that $Q(\mathit{terminal}, \cdot) = 0$

Loop for each episode:
  Initialize S
  Loop for each step of episode:
    Choose A from S using the policy ε-greedy in $Q_1 + Q_2$
    Take action A, observe R, S'
    With 0.5 probabilility:

$$Q_1(S, A) \leftarrow Q_1(S, A) + \alpha\Big(R + \gamma Q_2\big(S', \operatorname{argmax}_a Q_1(S', a)\big) - Q_1(S, A)\Big)$$

    else:

$$Q_2(S, A) \leftarrow Q_2(S, A) + \alpha\Big(R + \gamma Q_1\big(S', \operatorname{argmax}_a Q_2(S', a)\big) - Q_2(S, A)\Big)$$

    $S \leftarrow S'$
  until S is terminal

Example:

Suppose an agent is trying to solve a grid-world problem where it must navigate to a goal, and it can choose between several actions at each step.

- In **Q-learning**, if one action has a slightly higher Q-value due to random noise, the agent will repeatedly favor that action, leading to overestimation.
- In **Double Q-learning**, because the agent uses one Q-value estimate to choose the action and another to evaluate it, the influence of random noise is reduced, leading to more accurate learning of action values.

Benefits of Double Q-learning

- **Reduced overestimation bias**: By decoupling action selection and evaluation, Double Q-learning mitigates the overestimation of action values, leading to better policies and faster convergence.
- **More stable learning**: Double Q-learning is less sensitive to the fluctuations in Q-value estimates, which can result in more stable learning.

References

1. Russell S., and Norvig P., Artificial Intelligence A Modern Approach (3e), Pearson 2010

2. Richard S Sutton, Andrew G Barto, Reinforcement Learning, second edition, MIT Press

3. https://www.coursera.org/learn/fundamentals-of-reinforcement-learning/home/week/1

END OF UNIT-II