Université de Strasbourg

The Cliff-Walking Problem:

From Bellman's Equation to SARSA and Q-Learning algorithms

VASTA Francesca
VU Billy
2024-2025

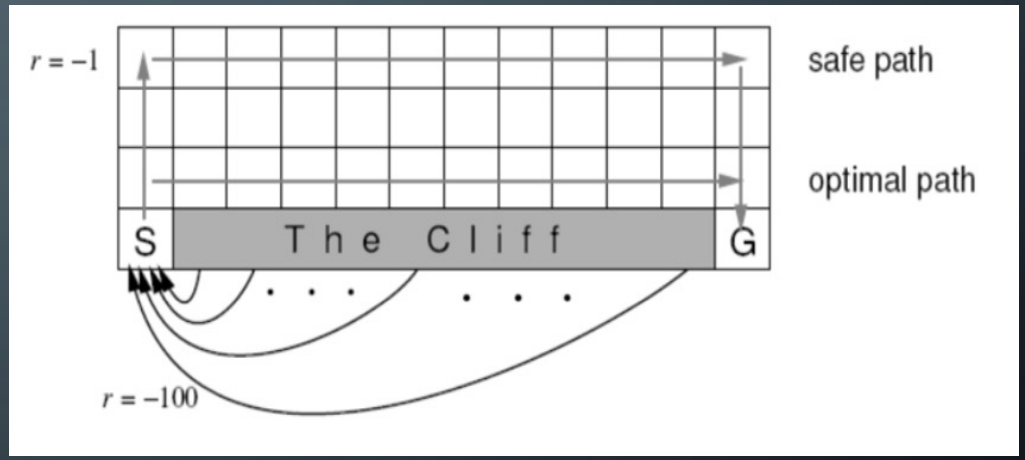
SUMMARY

1) The Cliff Walking Environment + Implementation
2) Introduction to Temporal Difference algorithms
3) SARSA - On-Policy Algorithm
4) Q-Learning - Off-Policy Algorithm
5) Code Improvement with Epsilon-Decay
6) Comparison of the performance of the two algorithms
7) Special case: Epsilon 100% greedy
8) Conclusion

THE CLIFF WALKING ENVIRONMENT

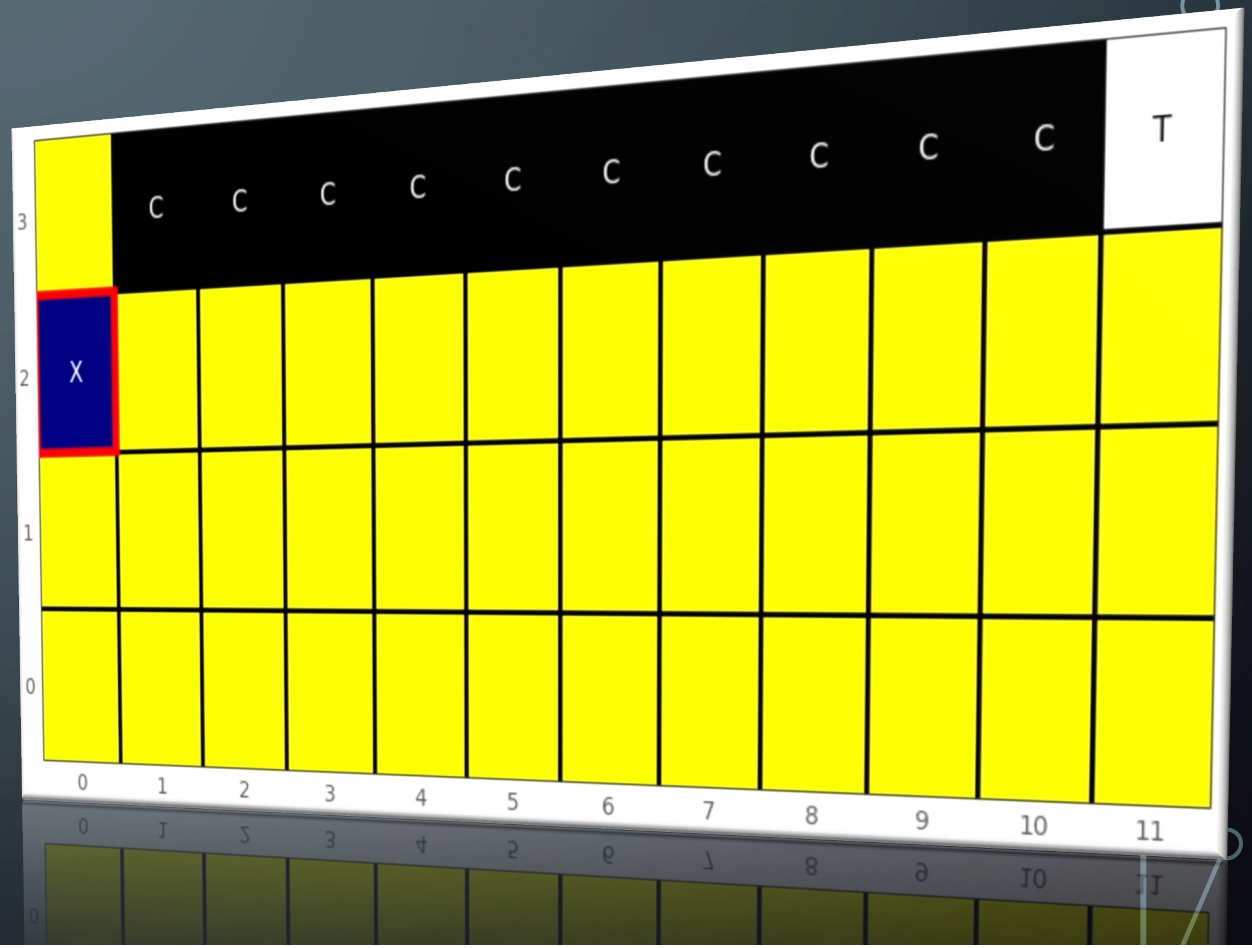
- The agent must reach the target without falling off the cliff

$$R(si) = \begin{cases} -1 & \text{for all } s_i \text{ not on the cliff} \\ -100 & \text{for all } s_i \text{ on the cliff} \end{cases}$$



IMPLEMENTATION OF THE CLIFF WALKING ENVIRONMENT

1) Library '**gym** '/'**CliffWalkingEnv**'
2) Initializes a 4x12 grid with key default parameters :
   - Starting position
   - Cliff locations
   - Defined actions



`actions` :

- `0: "up"`
- `1: "right"`
- `2: "down"`
- `3: "left"`

TEMPORAL DIFFERENCE ALGORITHMS

- In TD algorithms the learning process is driven by the ***temporal difference***, which is the gap between the agent's current value estimate and the newly observed value following each action
- The agent uses it to incrementally update its estimate towards the latest observation in real time

TD algorithms overcomes the disadvantages of Monte Carlo Methods and Dynamic Programming

TD > MC

The estimated Q π (s,a) is updated at each step and not at the end of the episode

TD > DP

TD doesn't need a model of the environment, i.e. no need to have prior knowledge of transition probabilites

INTRODUCTION TO TEMPORAL DIFFERENCE ALGORITHMS

→ All in all, with TD algorithms the agent can learn from past experiences without needing a full sequence and also when the environment is unknown

SARSA

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

Q-LEARNING

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

- α is the learning rate,
- r is the immediate reward received after taking action a in state s
- γ is the discount factor,
- s' is the next state and a' is the action taken in state s'

→ They remind us of the Bellmann's Equation, but with some adaptations ☺

→ The red boxes highlight their different updating rules!

SARSA : ON-POLICY ALGORITHM

SARSA

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

ON-POLICY

BEHAVIOURAL POLICY = UPDATING POLICY

Q(s',a') is the value of taking a' in s' under the current policy π

- More conservative and safe policy
- Stability in the learning process
- Sub-optimal solution
- Slower learning pace

SARSA : ON-POLICY ALGORITHM

```python
def make_epsilon_greedy_policy(Q, epsilon, nA):
    """
    Creates an epsilon-greedy policy based on a given Q-function and epsilon.

    Args:
        Q: A dictionary that maps states to action values.
            Each value is a numpy array of length nA (number of actions).
        epsilon: The probability to select a random action, float between 0 and 1.
        nA: Number of actions in the environment.

    Returns:
        A function that takes an observation (state) as an argument and returns
        the probabilities for each action in the form of a numpy array of length nA.
    """
    def policy_fn(state):
        action_probabilities = np.ones(nA, dtype=float) * (epsilon / nA)
        #action probabilities onitializes a numpy array where each action has a base probability
        #of epsilon / nA (the probability of choosing a random action).
        best_action = np.argmax(Q[state])
        #best-action finds the action with the highest Q-value for the given state using np.argmax.
        action_probabilities[best_action] += (1.0 - epsilon)
        #we adjust the probability of choosing the best action
        #In particular, we increases the probability of selecting the best action by adding (1.0 - epsilon) to it.
        #This ensures that with probability 1 - epsilon, the agent selects the best-known action, while
        #with probability epsilon, it explores other actions.
        return action_probabilities


    return policy_fn
```

This is the function which allows us to make policy improvements!

The function makes the policy greedy with respect to the current value function, the current epsilon and the number of actions in the environment.

Of course, each time the value function is updated, the policy is also update

This part of the function is the “greedy” part, once the best action is found, the action probability is updated

SARSA : ON-POLICY ALGORITHM

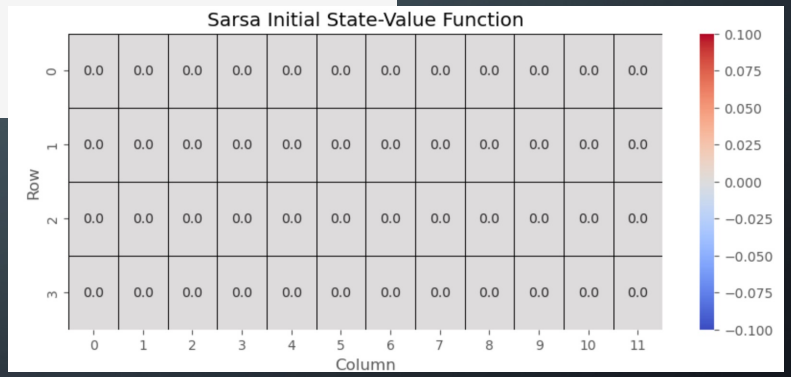
```python
def sarsa_decay(env, num_episodes, discount_factor=1.0, alpha=0.5, epsilon=1.0, epsilon_decay=0.995, epsilon_min=0.1):
    """
    SARSA algorithm: On-policy TD control. Finds the optimal epsilon-greedy policy.

    Args:
        env: OpenAI environment.
        num_episodes: Number of episodes to run for.
        discount_factor: Gamma (γ) discount factor.
        alpha: Step size (α), TD learning rate.
        epsilon: Initial probability to sample a random action, between 0 and 1.
        epsilon_decay: Decay rate of epsilon after each episode.
        epsilon_min: Minimum value that epsilon will decay to.

    Returns:
        A tuple (Q, stats_decay) where Q is the optimal action-value function,
        a dictionary mapping state -> action values, and stats_decay is an EpisodeStats
        object with two numpy arrays for episode_lengths and episode_rewards.
    """

    # Initialize the action-value function Q with zeros for all state-action pairs.
    Q_sarsa_decay = defaultdict(lambda: np.zeros(env.action_space.n))
```

Initialization of the state-action values:



SARSA : ON-POLICY ALGORITHM

TWO NESTED LOOPS

OUTER LOOP which applies to each episode

INNER LOOP which applies to each step of each episode

```python
for i_episode in range(num_episodes):
    # Decay epsilon after each episode, but ensure it doesn't go below the minimum epsilon
    epsilon = max(epsilon_min, epsilon * epsilon_decay)

    # Print progress every 100 episodes
    if (i_episode + 1) % 100 == 0:
        print(f"Episode {i_episode + 1}/{num_episodes} completed. Epsilon: {epsilon:.4f}")

    # Initialize the state
    state, _ = env.reset() if isinstance(env.reset(), tuple) else (env.reset(), {})

    # Choose action A from state S using the epsilon-greedy policy
    action_probabilities = make_epsilon_greedy_policy(Q_sarsa_decay, epsilon, env.action_space.n)(state)
    action = np.random.choice(np.arange(len(action_probabilities)), p=action_probabilities)
```

In the *OUTER LOOP*, for each episode:

1. We **initialize the state** by calling env.reset().
2. **Select an initial action** according to the epsilon-greedy policy
3. After each episode, **epsilon will be reduced** (= more exploitation)

SARSA : ON-POLICY ALGORITHM

TWO NESTED LOOPS

```
# Loop for each step of the episode
for t in itertools.count():
    # Take action A, observe reward R and next state S'
    next_state, reward, is_terminal, truncated, _ = env.step(action)

    # Choose next action A' from S' using the epsilon-greedy policy
    next_action_probabilities = make_epsilon_greedy_policy(Q_sarsa_decay, epsilon, env.action_space.n)(next_state)
    next_action = np.random.choice(np.arange(len(next_action_probabilities)), p=next_action_probabilities)

    # Calculate the TD target and TD error
    td_target = reward + discount_factor * Q_sarsa_decay[next_state][next_action]
    td_error = td_target - Q_sarsa_decay[state][action]

    # Update Q-value for current state-action pair
    Q_sarsa_decay[state][action] += alpha * td_error

    # Update statistics
    stats_sarsa_decay.episode_rewards[i_episode] += reward
    stats_sarsa_decay.episode_lengths[i_episode] = t + 1  # t starts from 0

    # Break if terminal state or truncated
    if is_terminal or truncated:
        break

    # Update state and action
    state = next_state
    action = next_action
```

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

In the *INNER LOOP*, for each step of the episode, the agent:

1. **Takes action A and observes the reward** and **next state** Chooses the next action A′ from S′ **based on the same epsilon-greedy policy**, using the Q-values for guidance.
2. **Updates Q-values** tfor the current state-action pair with the SARSA updating rule, using:

✓ Immediate reward

✓ The discounted temporal difference between future and current state-action values

Q-LEARNING : OFF-POLICY ALGORITHM

Q-LEARNING

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

OFF-POLICY

BEHAVIOURAL POLICY ≠ UPDATING POLICY
(100% greedy)

max_a' Q(s',a') represents the value of the best possible action in the next state s', irrespective of the current policy.

- ✓ Optimal Policy
- ✓ Faster convergence towards optimal policy
- ✗ High risk taking
- ✗ Instability during learning process

Q-LEARNING : ON-POLICY ALGORITHM

```python
def make_epsilon_greedy_policy(Q, epsilon, nA):
    """
    Creates an epsilon-greedy policy based on a given Q-function and epsilon.

    Args:
        Q: A dictionary that maps states to action values.
            Each value is a numpy array of length nA (number of actions).
        epsilon: The probability to select a random action, float between 0 and 1.
        nA: Number of actions in the environment.

    Returns:
        A function that takes an observation (state) as an argument and returns
        the probabilities for each action in the form of a numpy array of length nA.
    """
    def policy_fn(state):
        action_probabilities = np.ones(nA, dtype=float) * (epsilon / nA)
        #action probabilities onitializes a numpy array where each action has a base probability
        #of epsilon / nA (the probability of choosing a random action).
        best_action = np.argmax(Q[state])
        #best-action finds the action with the highest Q-value for the given state using np.argmax.
        action_probabilities[best_action] += (1.0 - epsilon)
        #we adjust the probability of choosing the best action
        #In particular, we increases the probability of selecting the best action by adding (1.0 - epsilon) to it.
        #This ensures that with probability 1 - epsilon, the agent selects the best-known action, while
        #with probability epsilon, it explores other actions.
        return action_probabilities


    return policy_fn
```

This is the function which allows us to make policy improvements!

The function makes the policy greedy with respect to the current value function, the current epsilon and the number of actions in the environment.

Of course, each time the value function is updated, the policy is also update

This part of the function is the “greedy” part, once the best action is found, the action probability is updated

Q-LEARNING : ON-POLICY ALGORITHM

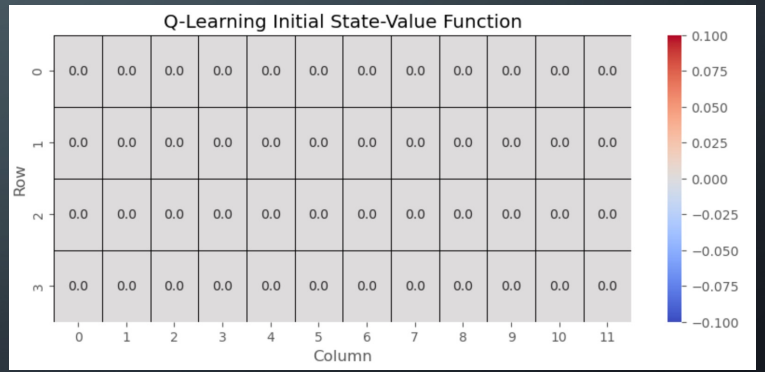
```python
def q_learning_decay(env, num_episodes, discount_factor=1.0, alpha=0.5, epsilon=0.1, epsilon_decay=0.995, min_epsilon=0.01)
    """
    Q-Learning algorithm with epsilon decay.

    Args:
        env: OpenAI environment.
        num_episodes: Number of episodes to run for.
        discount_factor: Gamma (γ) discount factor.
        alpha: Step size (α), TD learning rate.
        epsilon: Starting epsilon value for epsilon-greedy policy.
        epsilon_decay: The rate at which epsilon decays after each episode.
        min_epsilon: The minimum value that epsilon can decay to.

    Returns:
        Q, stats: Tuple containing the optimal Q-table and statistics.
    """

    # Initialize Q-table
    Q_q_decay = defaultdict(lambda: np.zeros(env.action_space.n))
```

Initialization of the state-action values:



Q-LEARNING : ON-POLICY ALGORITHM

TWO NESTED LOOPS

OUTER LOOP which applies to each episode

INNER LOOP which applies to each step of each episode

```python
# The policy derived from Q, using an epsilon-greedy approach
policy = make_epsilon_greedy_policy(Q_q_decay, epsilon, env.action_space.n)

for i_episode in range(num_episodes):
    # Decay epsilon after each episode, but ensure it doesn't go below the minimum epsilon
    epsilon = max(min_epsilon, epsilon * epsilon_decay)

    # Update the epsilon-greedy policy with the decayed epsilon
    policy = make_epsilon_greedy_policy(Q_q_decay, epsilon, env.action_space.n)

    # Print progress every 100 episodes
    if (i_episode + 1) % 100 == 0:
        print(f"Episode {i_episode + 1}/{num_episodes} completed. Epsilon: {epsilon:.4f}")

    # Initialize the state
    state, _ = env.reset() if isinstance(env.reset(), tuple) else (env.reset(), {})
```

In the *OUTER LOOP*, for each episode:

- **Initialize State**: by calling env.reset().
- **Epsilon is reduced** to encourage exploitation
- The **policy is updated** based on the Q-value and on the new value of epsilon

Q-LEARNING : O-POLICY ALGORITHM

TWO NESTED LOOPS

```
# Loop for each step of the episode
for t in itertools.count():
    # Choose action A from state S using policy derived from Q (epsilon-greedy)
    action_probabilities = policy(state)
    action = np.random.choice(np.arange(len(action_probabilities)), p=action_probabilities)

    # Take action A, observe R, S'
    next_state, reward, is_terminal, truncated, _ = env.step(action)

    # Update Q(S,A)
    td_target = reward + discount_factor * np.max(Q_q_decay[next_state])
    td_error = td_target - Q_q_decay[state][action]
    Q_q_decay[state][action] += alpha * td_error

    # Update statistics
    stats_q_decay.episode_rewards[i_episode] += reward
    stats_q_decay.episode_lengths[i_episode] = t + 1  # t starts from 0

    # End the episode if in terminal state
    if is_terminal or truncated:
        break

    # S ← S'
    state = next_state
```

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

In the *INNER LOOP*, for each step of the episode, the agent:

- **Takes action A** (based on the current state, using the epsilon-greedy policy) **and observes the reward** and **next state**
- **Updates Q-value f**or the current state-action pair is updated using the Q-learning update rule.

This update incorporates:

- ✓ The immediate reward.
- ✓ The discounted estimated maximum future reward from next_state.

CODE IMPROVEMENT WITH: *epsilon_decay*

- With an epsilon_decay there is early exploration at the beginning and gradual shift to exploitation
- Benefits : 1) Balanced Exploration-Exploitation Trade-off

  2) Efficient Learning

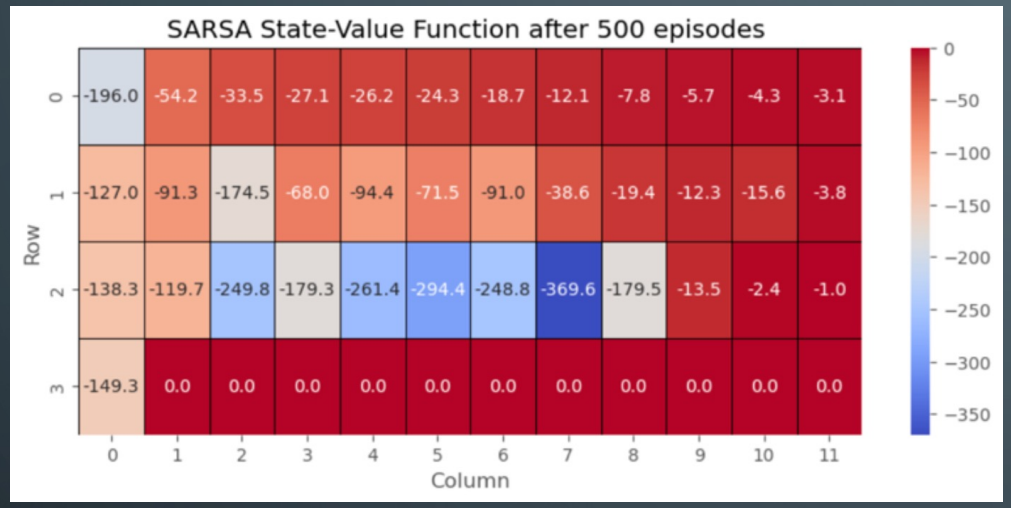
  3) Avoid being stuck in a Local Optimal

```
Episode 100/500 completed. Epsilon: 0.6058
Episode 200/500 completed. Epsilon: 0.3670
Episode 300/500 completed. Epsilon: 0.2223
Episode 400/500 completed. Epsilon: 0.1347
Episode 500/500 completed. Epsilon: 0.1000
```

→ 60.58% exploration and 39.42% exploitation
→ 36.7% exploration and 63.3% exploitation
→ 22.23% exploration and 77,77% exploitation
→ 13.47% exploration and 86.53% exploitation
→ 10% exploration and 90% exploitation

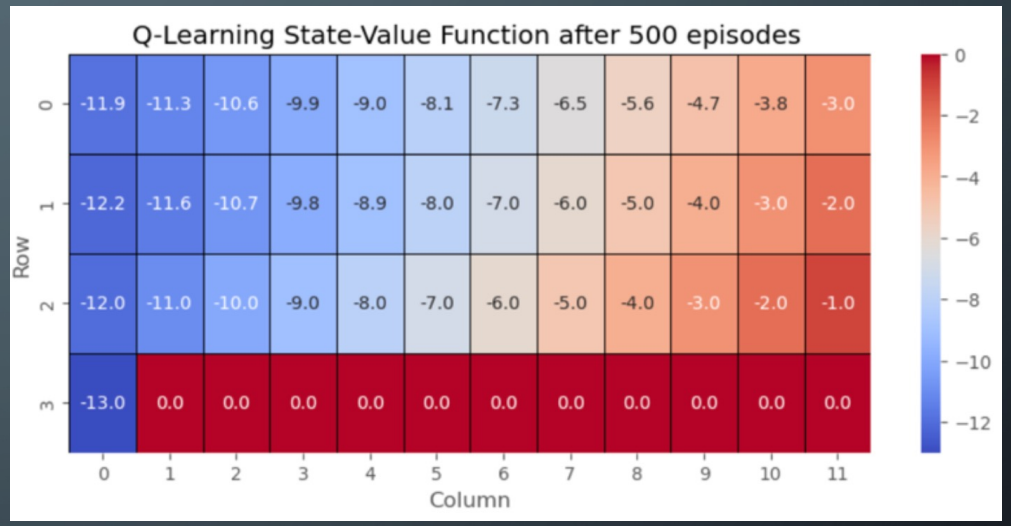
COMPARISON OF SARSA AND Q-LEARNING PERFORMANCE (1/3)

SARSA :



→ SARSA assigns higher values to states which are far from the cliff

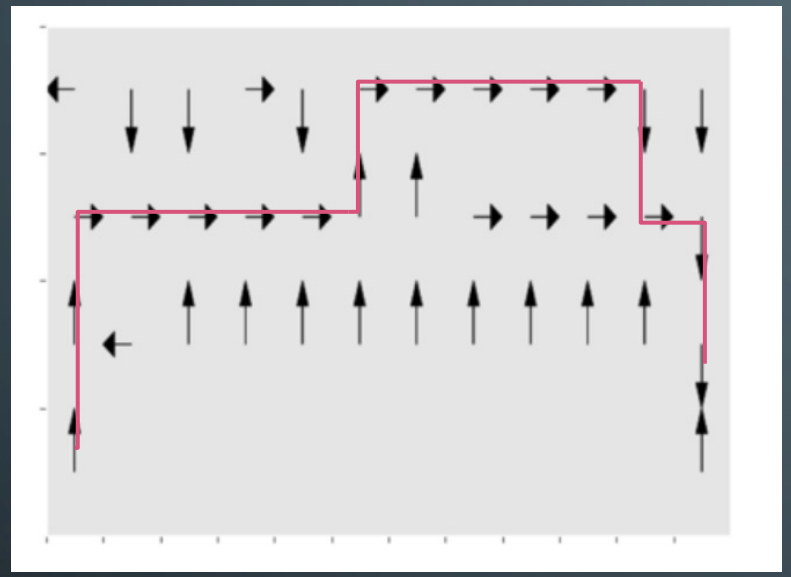
Q-Learning :



→ Q-Learning assigns higher values to states which are closer to the cliff

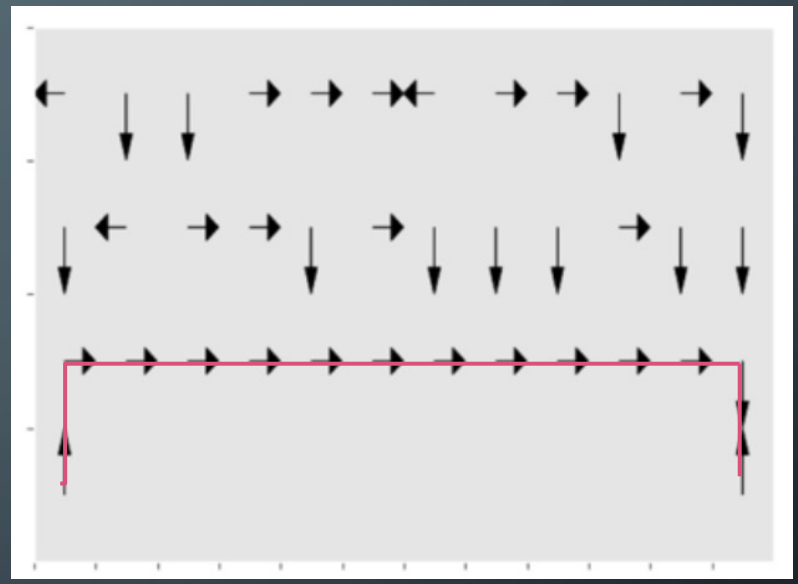
COMPARISON OF SARSA AND Q-LEARNING PERFORMANCE (2/3)

SARSA :



→ The optimal policy avoids states closer to the cliff. This policy is sub-optimal, if compared to Q-Learning.

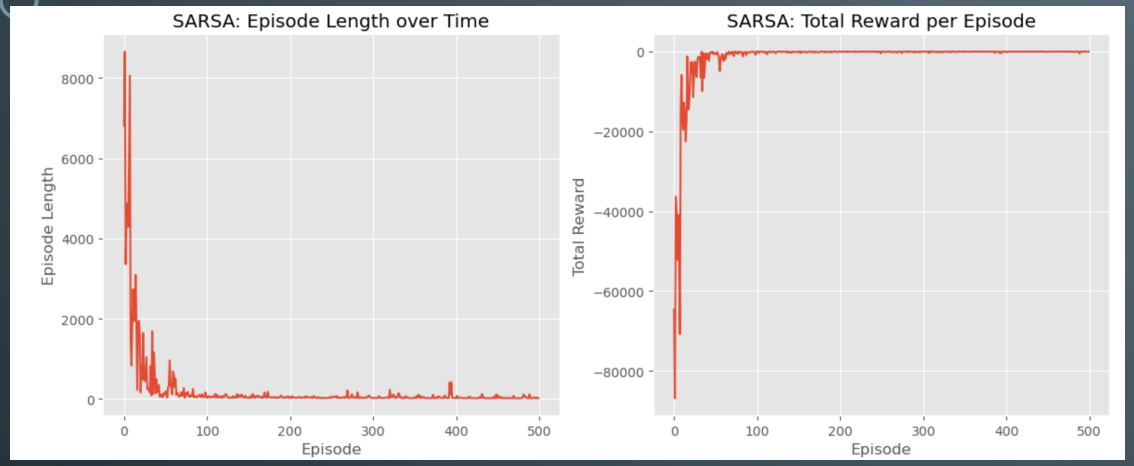
Q-Learning :



→ The optimal policy is close to the cliff and it is the shortest and fastest path to the target state

COMPARISON OF SARSA AND Q-LEARNING PERFORMANCE (3/3)

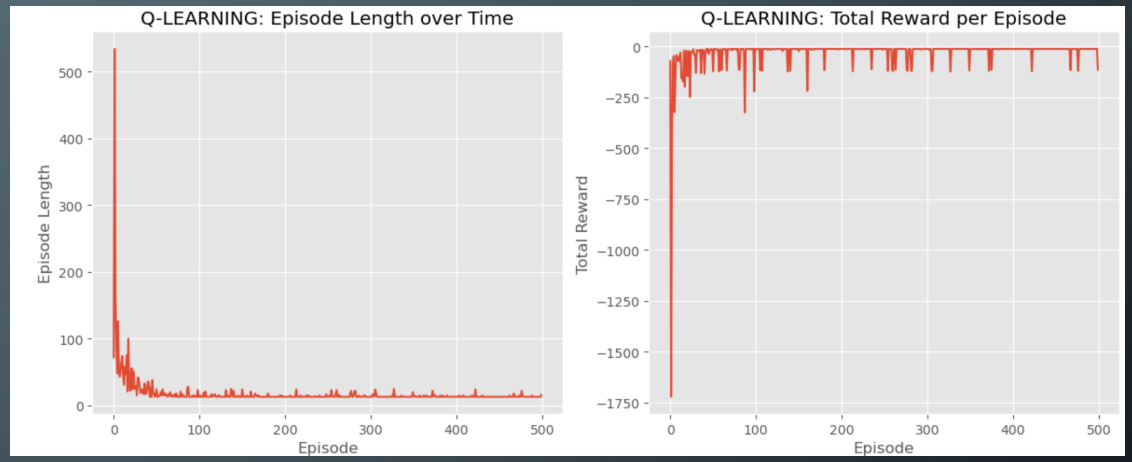
SARSA :




```
SARSA Average reward over 50 evaluation episodes: -17.0
```

- SARSA is slower at reaching converges (needs more time to find the optimal policy) because it is more conservative
- During training, the total rewards are more stable

Q-Learning :

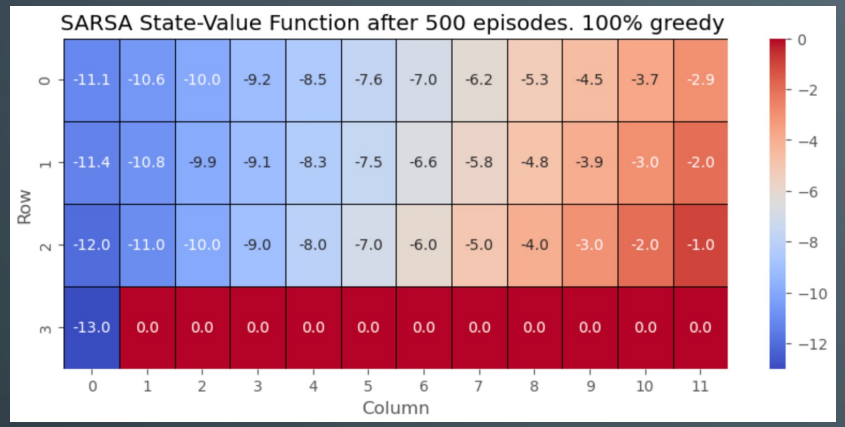



```
Q_learning Average reward over 50 evaluation episodes: -13.0
```

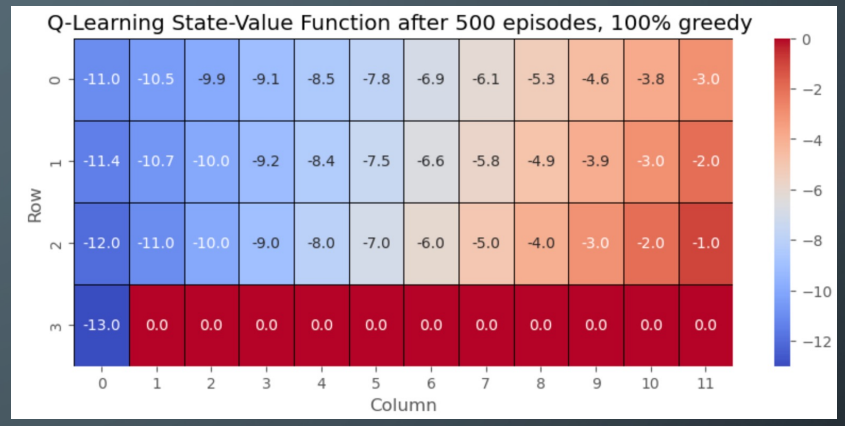
- Q-Learning reaches converges faster (episodes become shorter after 50 trials) because it is more aggressive and takes more risks
- During training, the total rewards are unstable because the agent falls off the cliff

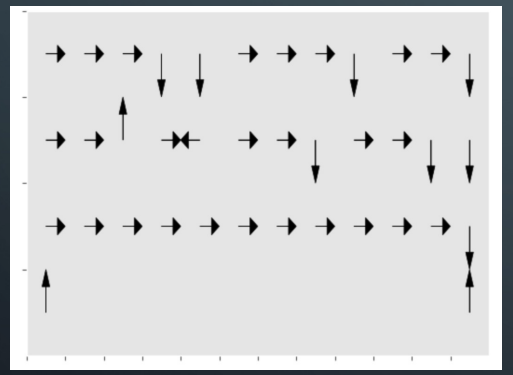
SPECIAL CASE : EPSILON 100% GREEDY `epsilon=0`

SARSA :

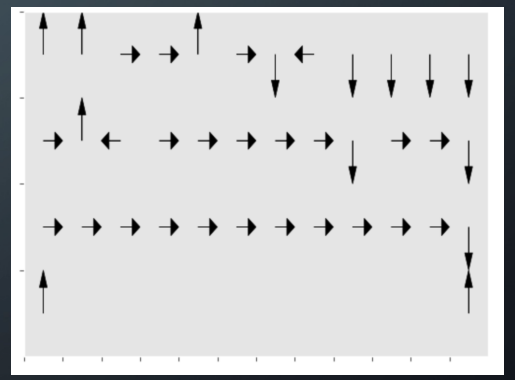


Q-Learning :





The essential difference between SARSA and Q-Learning lies in their exploration strategies rather than in their convergence capabilities.
When they adopt the **same updating policy (100% greedy)** they **converge to the same solution**



CONCLUSIONS

SARSA offers ***stability*** and ***safety*** in risky environments.

Preferable for scenarios where the agent is risk-avoidant

Q-Learning achieves ***faster adaptation*** and higher rewards when risks are managed.

Suitable for scenarios where the agent is risk-taker and the strategy exploits higher-risks to obtain higher returns

- ✓ When they employ a 100% greedy updating policy, they reach the same optimal policy
- ✓ Essential for the performance of the two algorithms is the right-tuning of parameters
- ✓ Future research can focus on *modifying the grid size, adjusting the reward structure, or introducing obstacles*

THANK YOU FOR YOUR ATTENTION :)

We wish you good luck with all your learning processes!