Multi-Agent Systems - Homework Assignment 6 Resit

3 March 2023

Name: Carol-Sebastian Rameder
Student number: 2747982

2. Reinforcement Learning: Cliff Walking

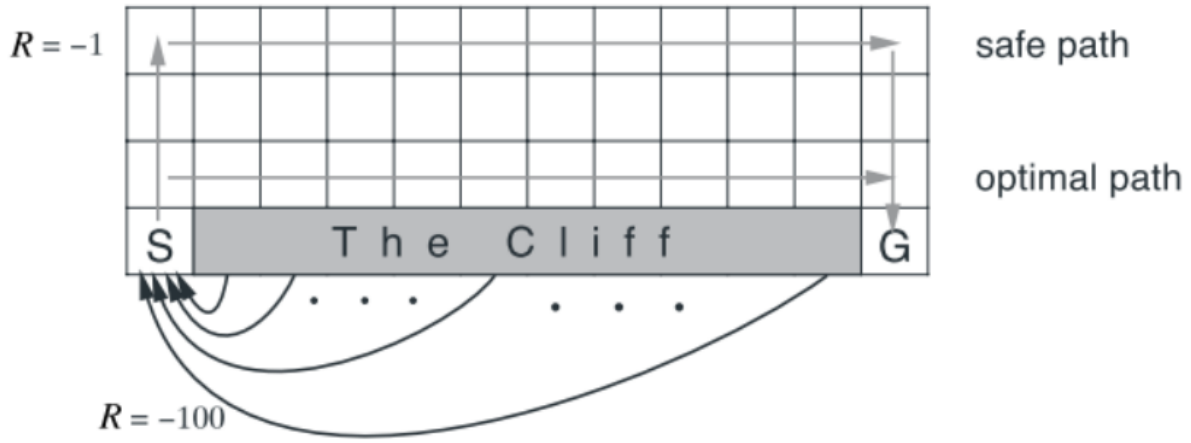
Reinforcement learning is a subfield of machine learning that deals with learning by trial and error through interaction with an environment. One of the popular methods of reinforcement learning is Q-learning, which is used to find an optimal policy for an agent in a given environment. SARSA is another reinforcement learning algorithm used to learn an agent's Q-value in an environment. In this assignment, we will discuss and compare the performances of Q-learning and SARSA on the Cliff Walking problem.

The Cliff Walking problem is a classic reinforcement learning problem that involves an agent navigating a grid world. The agent's goal is to reach the end of the grid while avoiding falling off a cliff in the process. The agent receives a negative reward for each step taken, and a more significant negative reward if it falls off the cliff. The agent also gets a positive reward upon reaching the goal. The problem is challenging because the agent must learn to balance the trade-off between taking shorter paths that are riskier versus longer paths that are safer.

In this assignment, we will first provide an overview of the Q-learning and SARSA algorithms. Then, we will discuss how these algorithms can be used to solve the Cliff Walking problem. We will compare and contrast the policies learned using the two algorithms and experiment with the replay buffer for Q-learning. We will experiment with the ϵ coefficient in the ϵ-greedy algorithm. Finally, we add a snake pit in the environment and compare the behaviour of both algorithms in this new context.

2.1 Comparing SARSA & Q-Learning. Replay Buffer

Implementation



The grid has 21 columns and 4 rows. As seen in the image (?!), the Start position is (3,0), the Goal is (3,21) and the Cliff is all the positions on the fourth line between the Start and the Goal. The rewards received after performing an action are according to the landing position. If the action gets the agent off the cliff, the reward is -100, to the goal 100 and -1 otherwise. In our case, this was implemented using a function which takes the current position and the chosen action as parameters and returns the reward accordingly. Getting off the grid was avoided by restricting the possible actions of the agent with a function. This takes the current position as the argument and returns the available actions of the agent accordingly, taking into consideration margins and corners.

The Q-values are stored in a 4x4x21 Numpy array, as there are 4 possible actions on each position on the grid - up, right, down and left. From one episode to the next one, the Q-values are remembered and the agent uses them to learn the environment. The values of the unavailable actions remain 0 throughout the whole learning process and are not taken into consideration when selecting the highest quality with the policy.

The ε-greedy policy implies choosing the action with the highest estimated value at each time step, with probability 1 - ε or a random action with probability ε. In our case, this was implemented by generating a random float between 0 and 1 and comparing it to ε. According to this small experiment, either a random action or the best action is returned. To choose the best action, firstly all possible actions from the current position are generated with a separate function. It takes all corners and margins into account, as not all four possible actions are available from any position. Given the set of possible actions, the one with the highest Q-values is returned. Otherwise, a random element from the list of available actions is returned. It is important to mention that this policy does not necessarily returns the actual best action, but the one with the highest quality as learned up to the current step.

The goal of both algorithms is to learn the Q-values of each possible action by learning the environment. To achieve this, a large number of episodes are simulated. During one episode, the agent starts from the predefined start position and chooses the action according to the policy and the Q-values available at the current step. runs until a final state is reached. This includes all positions in the Cliff and the Goal. At each iteration, an action is

chosen according to the ε-greedy policy, and the Q-value of this state and action is updated according to each algorithm-specific formula. The current state is updated with the landing position of the chosen action and the episode runs by repeating this loop until a final state is reached.

Initialize $Q(s, a)$ arbitrarily
Repeat (for each episode):
  Initialize s
  Repeat (for each step of episode):
    Choose a from s using policy derived from Q (e.g., ε-greedy)
    Take action a, observe r, s'
    $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$
    $s \leftarrow s'$;
  until s is terminal

Initialize $Q(s, a)$ arbitrarily
Repeat (for each episode):
  Initialize s
  Choose a from s using policy derived from Q (e.g., ε-greedy)
  Repeat (for each step of episode):
    Take action a, observe r, s'
    Choose a' from s' using policy derived from Q (e.g., ε-greedy)
    $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma Q(s',a') - Q(s,a)]$
    $s \leftarrow s'$; $a \leftarrow a'$;
  until s is terminal

One difference between the two algorithms is the method of computing the quality of the next state - *Q(S', A')*. In the case of Q-learning, the highest quality of the available actions is computed, whilst, in the case of SARSA, the quality of the next state is determined by the choice made using the ε-greedy policy and needs to be known, when the update is done. Therefore, we can state that the SARSA algorithm is “on policy” - the policy is used to update the Q-values - and that the Q-learning algorithm is “off policy”. In the implementation, computing the quality of the next state is done with functions similar to the ε-greedy policy - choosing the best or a random action is simulated without updating the next state. Here, we solve the following scenario: the next state is a final state and we need to compute the quality of an action taken - max_a *Q(S', a)* for Q-Learning and *Q(S', A')* for SARSA. The Quality of the next action, in this case, is 0.

We will also experiment with the replay buffer to check if reducing the correlation between subsequent experiences impacts the number of updates required to produce new policies when using the Q-Learning algorithm. In our case, the replay buffer is filled during a previous experiment to make sure it contains. It can also be filled with actions taken completely random, this does not have much importance. The buffer needs not be empty at the beginning of the learning process. Each tuple in the buffer contains the following:

- S: the current state
- A: the action chosen with ε-greedy policy

- R: the reward received when performing action A in state S
- S': the landing state

During the experiments, the agent produces the training data while interacting with the environment and stores it in a replay buffer of size 200, in our case. Instead of updating the Q values with the newly produced SARS tuple, a randomly sampled tuple from the buffer is used. This is called experience replay.

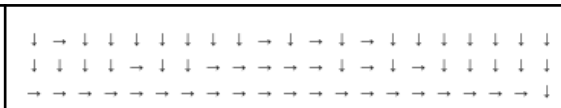
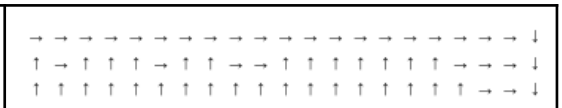
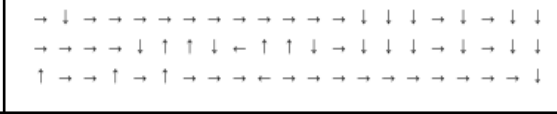
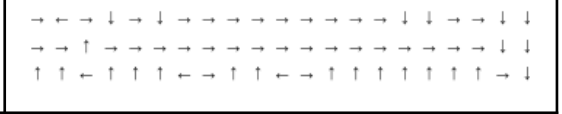
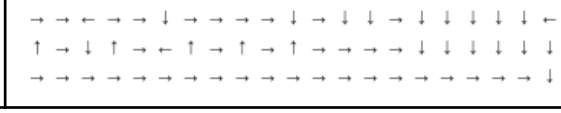
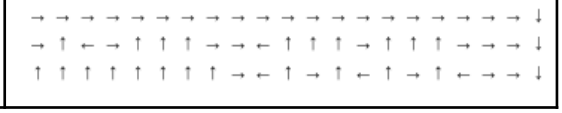
Question 1.

To assess the first question, we used the experimental setups for the hyperparameters from the table below. The second one allows more random actions (30%) and takes future rewards less into account - the rewards from the following states are discounted more than in the first experimental setup. In our environment, the vast majority of the qualities are negative as there are way more negative final states than positive ones - there are 19 "cliff" end states with a -100 negative reward and one end state with a positive reward. Thus we can affirm that the second experimental setup with a smaller gamma factor prioritizes the exploration, whilst the first one prioritises the exploitation.

Parameter	Experimental setup (1)	Experimental setup (2)
ε (epsilon)	0.2	0.3
γ (gamma)	0.95	0.8
Learning rate	0.1	0.2
Episodes	2000	1000

For the first experiment we ran all the 2000 episodes to let the agent fully learn the environment. The second experiment stops as soon as the optimal policy derived from the current Q-values leads the agent to the Goal position. In the table below, the optimal action for each position on the grid is shown. We computed this by choosing the highest quality from the available actions. The final policy was implemented by calling the e-greedy function with a negative exploration factor - here we do not allow random actions anymore. Starting from the bottom left corner (the Starting point of the agent), the path always leads to the bottom right corner - the Goal. The Early Stop method implies that the learning process stops when the current optimal policy leads the agent to the goal.

Experimental setup	Q-Learning	SARSA
1	↓ ↓ → ↓ → → ↓ → ↓ → ↓ ↓ ↓ ↓ ↓ ↓ ↓ → ↓ ↓ ↓ ↓ ↓ → → → → → → → → → → → ↓ ↓ ↓ ↓ → ↓ ↓ ↓ → → → → → → → → → → → → → → → → → → → ↓	→ → → → → → → → → → → → → → → → → → → ↓ ↑ ↑ ↑ ↑ ↑ ↑ ↑ ↑ → → ↑ → → → → → → → → → ↓ ↑ ↑ ↑ ↑ ↑ ↑ ← → ↑ ↑ ↑ ↑ ↑ ↑ ↑ → → ↑ ↑ → ↓

2		
1 + Early Stop		
2 + Early stop		

As shown in the resulting paths above, SARSA tends to learn the safe path and Q-Learning prefers the optimal path (both can be seen in the first image on the second page). This is directly caused by the foundational concepts of these two algorithms. SARSA avoids the optimal path because the policy may randomly choose to immediately fall off the cliff, as it also includes randomness in the next action. Additionally, when applying the Q-Learning update rule, the agent selects the action with the highest quality for the next step regardless of the action that was taken, thereby preventing it from falling off the cliff. This can lead to an overestimation of the Q-values and may result in a more optimistic policy. SARSA, on the other hand, updates its Q-values based on the Q-value of the next state and the action that was taken in the next state. This results in a more conservative estimate of the Q-values and may result in a safer policy. As a consequence, Q-learning tends to be more optimistic in the face of uncertainty, as it assumes that the Q-values are accurate and maximize the reward. SARSA, on the other hand, tends to be more cautious and pessimistic, as it takes into account the uncertainty in the Q-values and the potential risks associated with each action.

Early stop convergence results:

Experimental Setup	Q-Learning	SARSA	Replay buffer
1	1.04 seconds 227 steps	0.79 seconds 115 steps	10.91 seconds
2	0.406 seconds 215 steps	0.264 seconds 121 steps	7.65 seconds

Question 2.

The value of the exploration rate ϵ has a significant impact on the behaviour of the agent and the quality of the learned policy. The exploration rate determines the balance between exploration and exploitation in the agent's decision-making process. A high value of ϵ means that the agent is more likely to take random actions and explore the environment, while a low value of ϵ means that the agent is more likely to take the action with the highest Q-value and exploit the current knowledge. A smaller value of epsilon means that the agent

is less likely to take random exploratory actions and more likely to choose actions based on the current Q-values. This makes SARSA avoid falling off the cliff due to random actions and thus, prefer the optimal path.

Lowering the value for ϵ was the solution to the third question, as it makes the agent use the middle lane as its optimal policy in the case of SARSA. For Q-Learning, changing the ϵ value did not bring any change from the optimal path. The case of 0.05 can be seen in the image from “Question 3”. When the value is increased to ϵ = 0.1 the agent barely deviates from the safe path and when the value is higher than that, the safe path is followed entirely. When we decrease the value of ϵ to 0.01 it makes the agent follow the optimal path, as seen in the second image below. This is caused by making the SARSA method highly exploitative,

Optimal Path for ϵ = 0.1

Optimal path for ϵ = 0.01

Question 3.

To implement the new environment including the snake pit, we updated the reward function. The three possible actions landing the agent on the snake pit received a -100 reward. In addition, we added the snake pit to the final states. To test the new environment, we have made changes to the existing functions, which will need to be redefined before running the code (more instructions in the notebook file).

In the initial environment, the optimal path derived from the learned Q-values in the case of Q-Learning did not contain the position of the newly added snake pit. As expected, the optimal path after the addition does not change. The algorithm converges quickly using both previously mentioned experimental setups.

In the case of SARSA, the snake pit is placed right in the middle of the typical path learned. To make the algorithm converge in the new environment, it was required to run try several configurations of hyperparameters without success, as the model learned loops that caused runtime errors or reached the recursion limit. The solution to this was found by greedyfing SARSA. We aimed to keep the values of ε low and decrease the initial value over episodes to allow some exploration initially before focussing on exploitation. We implemented an epsilon scheduler that starts with the 0.05 value for ε initially and decreases by 0.0002 after each episode. We did not allow values lower than 0.01 to be able to escape possible loops with random actions. We found the values previously mentioned empirically. (this paragraph can be taken into consideration for the second question as well)

As seen in the image below, the optimal path goes through the middle lane. The agent learns that it is not efficient to follow the safe path anymore and chooses a solution that balances efficiency and safety.

We found the solution with 400 episodes. This method avoids overtraining successfully here. The learning rate was 0.1 and the gamma coefficient was 0.9.