DS 669
Reinforcement Learning
Programming Assignment $\#2$

In this assignment, you will get familiar with `CliffWalking-v0` which is an environment from the Gymnasium library. You will also implement the Sarsa method and Q learning method and run various experiments to understand the effects of different parameter settings.

Submission:

- You need to finish the assignment by finishing all parts in each section. If the task is marked as **Question**, please answer it in your words clearly. If the task asks for a screenshot, the screenshot must cover all the answers.
- Make sure you number each of your responses so it is easy to match responses with exercises.
- When finished, submit your report in **PDF** form and submit your code files `codebase_main.py`, `codebase_train.py`, `codebase_Sarsa.py`, `codebase_cliff_walking_test.py`, and `codebase_Q_learning.py` on Canvas. **Please DO NOT** zip all your files together.

Environment Instruction:

You need to call the `env.step` and `env.reset` in your code to take action and reset the environment back to the initial state. Here is a brief instruction of these two functions.

When you call the `env.step`:

- The input of `env.step`:
  - action (ActType): an action the agent provides to update the environment state.
- The returns of `env.step`:
  - observation (ObsType): the next observation after taking actions.
  - reward (SupportsFloat): The reward as a result of taking the action.
  - done (bool): Whether the agent reaches the terminal state (as defined under the MDP of the task) which can be positive or negative. If true, the user needs to call the reset.
  - truncated (bool): Whether the truncation condition outside the scope of the MDP is satisfied. (Not used in this assignment.)
  - info (dict): Contains auxiliary diagnostic information (helpful for debugging, learning, and logging). (Not used in this assignment.)

When you call the `env.reset`:

- The input of `env.reset`:
  - None
- The returns of `env.reset`:
  - observation (ObsType): the initial state.
  - info (dictionary): This dictionary contains auxiliary information complementing "observation". (Not used in this assignment.)

P.S.: The instructions here are different from the instructions listed in the official document of Gymnasium. The reason is that both the env.step and env.reset functions are called from `cliffwalking.py` and it rewrites these two functions.

I Environment: Cliff Walking (10%)

In this section, you will get familiar with `Cliff Walking` environment. Please read through `codebase_main.py`, `util.py`, `codebase_cliff_walking_test.py` and `get_args.py`.

(a) You do not need to write any code in this question. Please:

- set the parameter `method` in `get_args.py` as `test-cliff-walking`.
- run `codebase_main.py` and the `test_each_action` function.

(1) Please take a screenshot of all the output results from the code.

(2) **Question**: What is the initial state number? Where is it located on the map?
(**Hint:** The location is represented as [#row, #col], e.g., the top-left corner's location is represented as [0,0].)

(3) **Question**: What do the actions $0, 1, 2,$ and 3 each represent?

(4) **Question**: After taking action 0, which state does the agent reach? What is the reward?

(5) **Question**: Which position do the state numbers 0, 11, and 47 represent? Can you briefly describe how the state number corresponds with the position?

(6) **Question**: After taking action 1, which state does the agent reach? What is the reward? Is the agent terminated?
(**Hint:** The episode terminates when the player enters state [47] (location [3, 11]). You can check the definition of "Episode End" from the official Gymnasium documentation.)

(7) **Question**: After taking action 2 and 3, which state does the agent reach? What is the reward? Can you explain what does it mean?

(8) **Question**: Can you briefly describe the transition and reward rules of `Cliff Walking` environment?

(b) In this question, you need to finish your code in `codebase_cliff_walking_test.py` and `codebase_main.py` between the pound sign lines denoted with "Your Code" in each function, then run the program in `codebase_main.py` to test your code. First, you need to implement the function `test_moves` in the `codebase_cliff_walking_test.py` file.

- First, reset the environment.
- Then, take the action in the parameter `actions` one by one for each step.
- Print the `action`, `next state`, `reward`, and `done` for each step.

You need to call the `env.step` and `env.reset` to finish this function.

Then, in `codebase_main.py` file:

- Create a list of actions to direct the agent moving from the starting position to the most top-right corner position.
- Call `test_moves` function to test if your series of actions are correct.

(1) Please take a screenshot of your result. The screenshot must include the last state that your actions lead the environment to.

(2) **Question**: What is the highest total reward for an episode in this environment?
(**Hint**: An episode must start from the initial state and end in a terminal state. The top-right corner state in this question is not terminal. Each time step incurs -1 reward, unless the player stepped into the cliff, which incurs -100 reward.

(c) In the `codebase_main.py` file, please:

- Create a list of actions to direct the agent moving from the initial state to the terminal state (the bottom-right corner of the map) and has the highest total reward.
- Call the `test_moves` function to test if your series of actions are correct.

(1) Please take a screenshot of your result. The screenshot must include the last state that your actions lead the environment to.

II Implementation of Temporal Difference (TD) Control (60%)

In this section, you need to read through and understand the `codebase_main.py`, `util.py`, `get_args.py` and `codebase_Sarsa.py` in the project. Finish your code in `codebase_Sarsa.py` between the pound sign lines denoted with "Your Code" in each function. Run the program in `codebase_main.py` to test your code and obtain your results.

Please read the comments of each variable carefully and understand the program structure, the definition of inputs and outputs, and the notes under "Your Code".

(a) Implementation of n-step SARSA

- Before implementing the function of `sarsa` and `n_step` in `codebase_Sarsa.py`, you need to finish `epsilon_greedy` function in `codebase_train.py`. Please fully understand the epsilon greedy algorithm and follow the guide under "Your Code".
- Then please call `epsilon_greedy` in `n_step` function located in the `codebase_Sarsa.py` to finish the n steps-sampling to calculate the reward. You need to call `env.step` to finish the `n_step` function. Moreover, you need to take care of the case that the environment terminates before the n-step. In this case, you need to stop and record the actual number of steps that have been taken in the variable `acted_steps`. The `acted_steps` is needed in the `sarsa` function for updating the q-table. Please check the definition of input and output carefully under the function names.
- Then call `n_step` function to finish n-step SARSA in `sarsa` function. Please follow the guide under "Your Code".
- Set the parameter `method` in `get_args.py` as `sarsa`. Run the program in `codebase_main.py`, and the code will calls the function `plot_return` to save a plot of how the total reward changes with the increase of the number of episodes.
- You can use the `evaluate_policy` in `codebase_train.py` to evaluate your q-table. It can use the greedy method on the q-table to generate a policy, reset the environment, and run the policy on the environment to get an episode. It can print out the policy for the states and the total reward of the episode. The codebase calls `evaluate_policy` function by default.

- The code already defined a variable `tabular_q` as a table to store the q values. Each `tabular_q[state][action]` stores the q value for each pair of state and action.
- In this question, the epsilon is fixed as the value of `init_epsilon=0.1`.

(1) Please paste the generated plot of your result.

(2) Please take a screenshot of the policy and the episode reward after calling the `evaluate_policy`.

(3) **Question:** According to the generated plot, does the q-table converge? Why if it does not?

(4) **Question:** What are the disadvantages of using a constant value of ε?

(5) **Question:** Should be value of ε gradually increase or decrease as the number of episodes increases?

(b) Optimization of epsilon ε

- In this question, we adjust the value of ε to optimize the SARSA model and make it converge. According to the convergence condition, please use $\varepsilon = \frac{init\_epsilon}{k}$ to improve the algorithm. The `init_epsilon` is the default setting of epsilon in the `get_args.py`, and k is the kth episode generated in the `sarsa` function. Please implement this method in the sarsa function in `codebase_Sarsa.py` by replacing the line "`epsilon = init_epsilon`" with your code.
- Please modify the name of the saved plot accordingly if you do not want to overwrite the previously generated plot.

(1) Please paste the generated plot of your result.

(2) Please take a screenshot of the policy and the episode reward.

(3) **Question:** Please explain why this method can help SARSA to converge.

(4) **Question**: Is the total reward generated from part II.(b) the highest q value of this game? In other words, does SARSA generate the optimal policy with this setting?

(c) Effect of number of steps in n-step SARSA

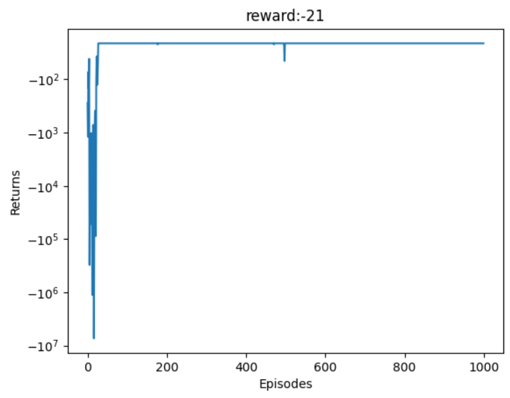
- A computer science student Steve conducted an experiment setting the value of `num_steps` to be 10 and 100, and respectively obtained the reward results and policy results shown in screenshots of Figures 1 and 2 . Based on his results, please finish the following questions.

  (1) **Question:** With the state-action pair (1,2) and (17,1), what will happen?
  (**Hint:** state-action pair is defined as (state number, action number). For example, the state-action pair (1,2) indicates that the agent's action is 2 at state 1, located at [0,1].)

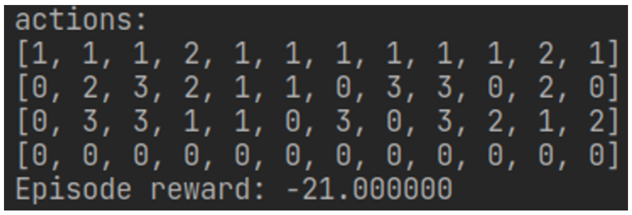
  (2) **Question**: How do the different numbers of steps affect the convergence of the q-table? Does a larger number of steps lead to faster or slower convergence?

  (3) **Question**: Are the total rewards of the 10-step and 100-step SARSA higher or lower than those of teh 1-step SARSA?

  (4) **Question**: There is an "wrong" action in the policy of the 10-step SARSA shown in the screenshot of Figure 1. Can you find the state-action pair and explain why the policy finally selected the wrong action for this state? You can print out and analyze the q values of all the 4 actions of this state.
  (**Hint:** "wrong" means that the action will lead the agent to get a worse reward.)



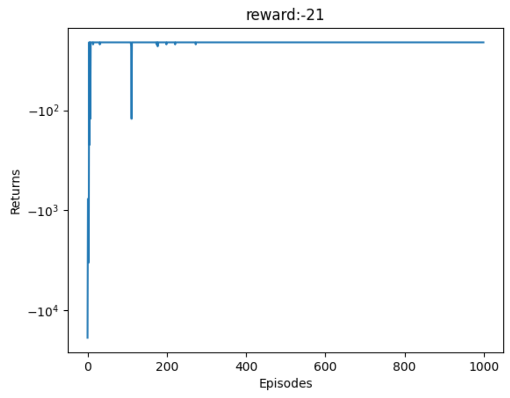
(a) reward result



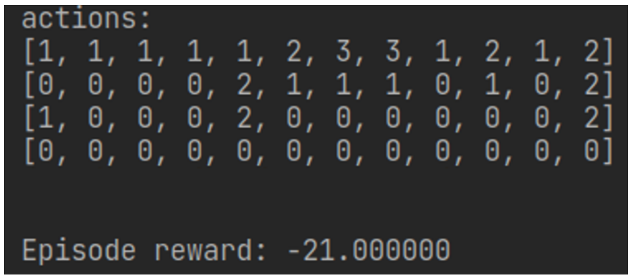
```
actions:
[1, 1, 1, 2, 1, 1, 1, 1, 1, 1, 2, 1]
[0, 2, 3, 2, 1, 1, 0, 3, 3, 0, 2, 0]
[0, 3, 3, 1, 1, 0, 3, 0, 3, 2, 1, 2]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
Episode reward: -21.000000
```

(b) policy result

Figure 1: num_step=10



(a) reward result



```
actions:
[1, 1, 1, 1, 1, 2, 3, 3, 1, 2, 1, 2]
[0, 0, 0, 0, 2, 1, 1, 1, 0, 1, 0, 2]
[1, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0, 2]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]


Episode reward: -21.000000
```

(b) policy result

Figure 2: num_step=100

- In this sub-question, we try to find out in which case the SARSA fails to find a policy. To prevent an infinite loop, modify the condition of your while loop to limit the episodes to a maximum of 10000 steps. This means the q table stops updating when the episode is done or the total steps of the episode are greater than 10000. Then set the `num_steps` as 10 and rerun the code.

  (5) Please paste the plot. Please take a screenshot of the policy and the total reward.

  (6) **Question**: Are there any traps in the policy based on the screenshot you got in II.(c)(5)? If so, could you provide the state-action pairs associated with these traps? (**Hint:** A trap means the policy might lead the environment to a dead end, resulting in non-termination.)

III Implementation of Q-learning (30%)

In this section, you need to read through `main.py`, `util.py`, `get_args.py`, and `codebase_Q_learning.py`. Finish your code in the function `q_learning` in `codebase_Q_learning.py` between the pound sign lines denoted with "Your Code" in each function. Run the program in `codebase_main.py` to test your code.

(a) Implementation of Q-learning

- Please reserve your optimization of ε in and by copying your implementation of I.(b) to the same place in the function of `q_learning`. Reset the `init_q_value` as 0.1 and `seeds` as 1. Set the `method` as `q-learning`
- To implement Q-learning, you need to call `epsilon_greedy` function in `codebase_train.py` (n-step is not needed in this question).
- The implementation is quite similar to SARSA. You just need some simple modifications. Please follow the guide under "Your Code".

(1) Please paste the generated plot. Please take a screenshot of the policy and the total reward.

(2) **Question**: Compared with the result you got from partII.(b)(1), does Q-learning converge faster or slower? Provide your explanation for why it is faster or slower.
(**Hint:** faster or slower here refers to `num_step`, not the actual running time.)

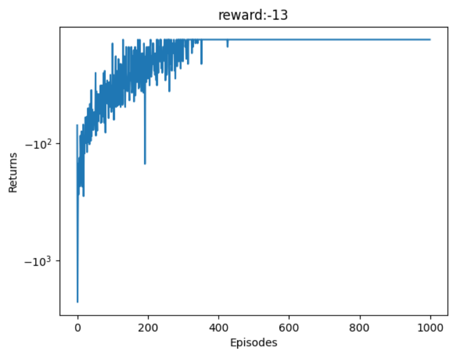
(3) **Question**: Does the total reward have a higher total reward than the SARSA? Please explain why it is higher or lower.

(4) **Question**: Is the total reward generated here the highest of the cliff walking?

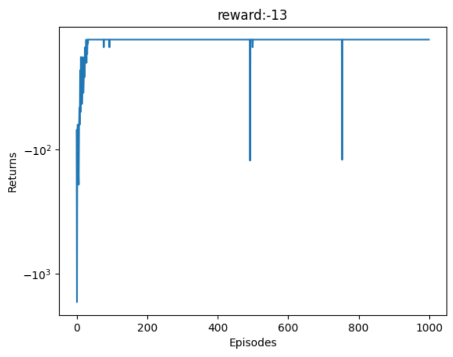
(b) Effect of Alpha α

- Continuing with Steve's experiment, he tested for different values for α to see the effect on Q-learning. He set α to be 0.1 and 0.9, and obtained the reward results separately, as shown in Figure 3. Based on his results, please finish the following question.

(1) **Question**: Do the different values of α affect the total reward of Q-learning? Please explain the reason.



(a) α = 0.1



(b) α = 0.9

Figure 3: reward results for different α values

(2) **Question**: Which α help Q-learning to converge faster? A larger α or a smaller α? Can you explain why?

(3) **Question**: Do you observe fluctuation in the plot after convergence? Can you explain how α affects the stability of the q-table values?