

## I Environment: Cliff Walking (10%)

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

Documentation: [Cliff Walking Documentation](#)

### Description

The game starts with the player at location [3, 0] of the 4x12 grid world with the goal located at [3, 11]. If the player reaches the goal the episode ends.

### Action Space

The action shape is (1,) in the range [0, 3] indicating which direction to move the player.

- 0: Move up
- 1: Move right
- 2: Move down
- 3: Move left

**(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.

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

`python Assignment-2/codebase_main.py -method test-cliff-walking`

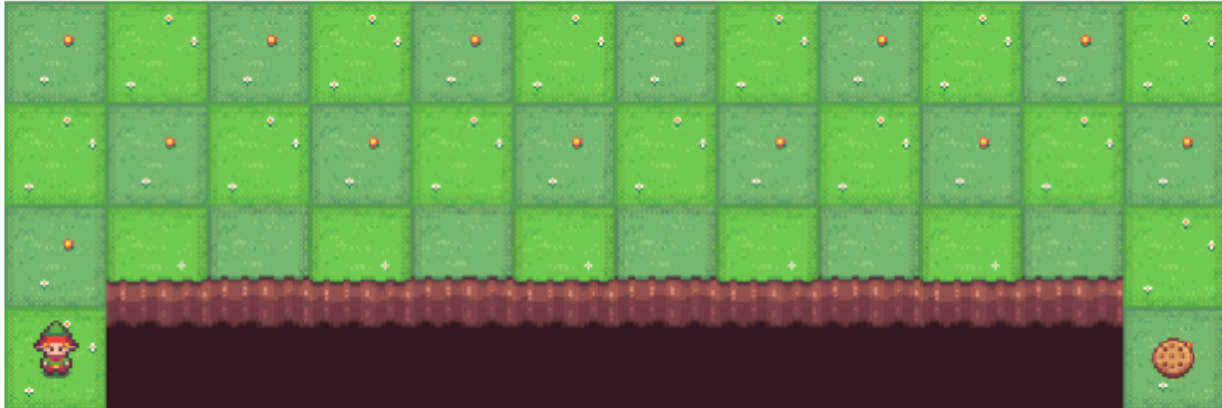
```
• (venv) marethu@Marethu:~/DS669_RL$ python Assignment-2/codebase_main.py -method test-cliff-walking
36
state:36, action:0, reward:-1, done:False, next_state:24
state:36, action:1, reward:-100, done:False, next_state:36
state:36, action:2, reward:-1, done:False, next_state:36
state:36, action:3, reward:-1, done:False, next_state:36
```

**(a)(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].)**

The initial state number is 36, because it starts in the bottom left corner. I know this because action:0 is up, action:1 is right, action:2 is down, action:3 is left. I know this because the projected next state is 36 for actions 1-3, and 24 for action 0. The game starts at [3, 0]

according to the documentation, which can be confirmed by visualizing the grid. The start position is 3 rows down from  $[0, 0]$  and in the same column, hence  $[3, 0]$ .

Environment looks like this:



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

To reiterate, action:0 is up, action:1 is right, action:2 is down, action:3 is left according to the documentation.

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

After taking action 0, the agent reaches state 24, because the action it takes is to move upward to  $[2, 0]$ . The reward is -1.

**(a)(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?**

Index  $[0 - 11]$  : 12 columns, Index  $[0 - 3]$  : 4 rows

States correspond to iteration through columns by row number, so position  $[0, 0]$  is state 0, position  $[1, 0]$  is state 12, position  $[2, 0]$  is state 24, and position  $[3, 0]$  is state 36.

That means that state number 0 corresponds to position  $[0, 0]$  (top left corner on grid), state number 11 corresponds to position  $[11, 0]$  (top right corner on grid), and 47 corresponds to  $[11, 3]$  (bottom right corner on grid).

(a)(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.)

## Description

The game starts with the player at location [3, 0] of the 4x12 grid world with the goal located at [3, 11]. If the player reaches the goal the episode ends.

A cliff runs along [3, 1..10]. If the player moves to a cliff location it returns to the start location.

The player makes moves until they reach the goal.

Adapted from Example 6.6 (page 132) from Reinforcement Learning: An Introduction by Sutton and Barto [1].

The cliff can be chosen to be slippery (disabled by default) so the player may move perpendicular to the intended direction sometimes (see `is_slippery`).

With inspiration from: [https://github.com/dennybritz/reinforcement-learning/blob/master/lib/envs/cliff\\_walking.py](https://github.com/dennybritz/reinforcement-learning/blob/master/lib/envs/cliff_walking.py)

## Episode End

The episode terminates when the player enters state [47] (location [3, 11]).

After taking action 1, the agent would reach a (hypothetical) terminal state of failure if it were to attempt to transition to state 37 / position [3, 1] because it would walk off the side of the cliff. So instead, the agent is penalized heavily by a reward of -100 for trying to walk over the cliff, and it is actually sent back to the initial state / initial starting position (state 36 / position [3, 0]).

The documentation indicates that this is true, because it mentions this rule, and because the only listed terminal state is reached upon entering state 47. Counterintuitively, there is no terminal state upon walking over the edge of the cliff.

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

Actions 2 and 3 result in no grid position change, so the next state is the current state which is 36. This is because action 2 is down and action 3 is left, and both actions would result in the agent hitting the grid/wall boundaries of the environment and failing to actually move to a new position. This failure to move to a new position provides the agent with a reward of -1 because it is treated as transitioning from the current state to the current state, despite the lack of actual position change.

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

The transition and reward rules of the Cliff Walking environment are as follows:

- Actions that result in a state transition / position change that **is not a location reset** from falling off the cliff, such as moving from state 36 to state 24 (position [3, 0] to position [2, 0]), receive a penalty of -1.
- Actions that result in a state transition / position change that **is a location reset** from falling off the cliff, such as attempting to move from state 36 to state 37 (position [3, 0] to position [3, 1]), receive a penalty of -100 and a position reset to the initial state (36 with position [3, 0]).
- Actions that result in no state transition / position change, such as attempting to move outside of the grid dimensions, receive a reward of -1.

**(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.

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

**python Assignment-2/codebase\_main.py -method test-cliff-walking**

```
my_actions_list = [0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
```

```
action:0, next_state:24, reward:-1, done:False
action:0, next_state:12, reward:-1, done:False
action:0, next_state:0, reward:-1, done:False
action:1, next_state:1, reward:-1, done:False
action:1, next_state:2, reward:-1, done:False
action:1, next_state:3, reward:-1, done:False
action:1, next_state:4, reward:-1, done:False
action:1, next_state:5, reward:-1, done:False
action:1, next_state:6, reward:-1, done:False
action:1, next_state:7, reward:-1, done:False
action:1, next_state:8, reward:-1, done:False
action:1, next_state:9, reward:-1, done:False
action:1, next_state:10, reward:-1, done:False
action:1, next_state:11, reward:-1, done:False
Start State: 36
End State: 11
total reward:-14
```

**(b)(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.)

The highest total reward is -13, because this is the shortest path to get to the goal/terminal state from the initial state.

**(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.

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

**python Assignment-2/codebase\_main.py -method test-cliff-walking**

```
optimal_path_actions_list = [0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2]
```

**python Assignment-2/codebase\_main.py -method test-cliff-walking**

```
action:0, next_state:24, reward:-1, done:False
action:1, next_state:25, reward:-1, done:False
action:1, next_state:26, reward:-1, done:False
action:1, next_state:27, reward:-1, done:False
action:1, next_state:28, reward:-1, done:False
action:1, next_state:29, reward:-1, done:False
action:1, next_state:30, reward:-1, done:False
action:1, next_state:31, reward:-1, done:False
action:1, next_state:32, reward:-1, done:False
action:1, next_state:33, reward:-1, done:False
action:1, next_state:34, reward:-1, done:False
action:1, next_state:35, reward:-1, done:False
action:2, next_state:47, reward:-1, done:True
Start State: 36
End State: 47
total reward:-13
```

The highest total reward for an episode in this environment is equivalent to the highest reward path that an agent can take to get from the start state 36 at [3, 0] to the end state 47 at [3, 11]. This path consists of moving upwards once, moving all the way to the right, and then moving downwards once to reach the goal.

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

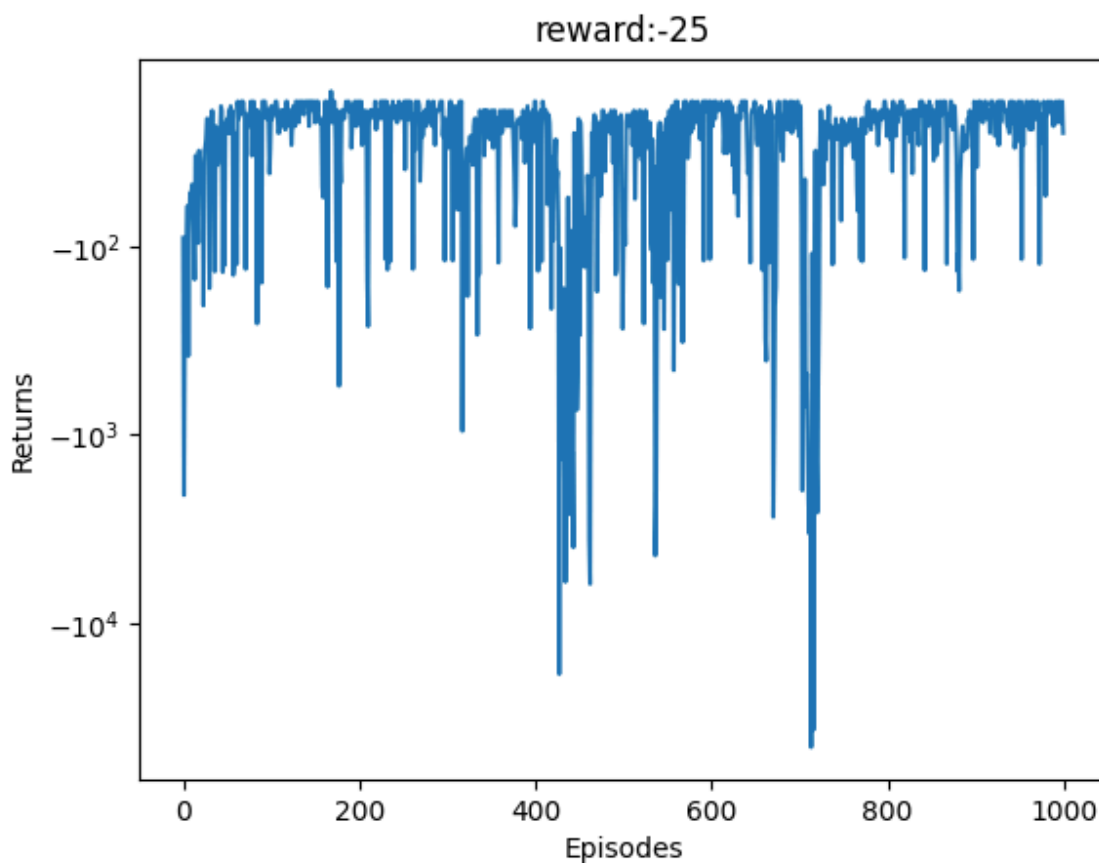
In this section, you need to read through and understand the codebase\_main.py, util.py, get\_args.py and codebase\_Sarsa.py in the project. Finish your code in codebase\_Sarsa.py between the pound sign lines denoted with "Your Code" in each function. Run the program in codebase\_main.py to test your 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 the 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 the 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.

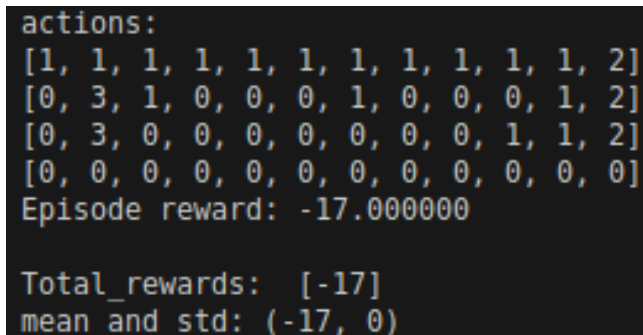
- Then call the `n_step` function to finish the n-step SARSA in the `sarsa` function. Please follow the guide under "Your Code".
- Set the parameter `method` in `get_args.py` as `sarsa`. Run the program in `codebase_main.py`, and the code will call the function `plot_return` to save a plot of how the 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`.

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



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

`python Assignment-2/codebase_main.py -method sarsa -seeds 1`

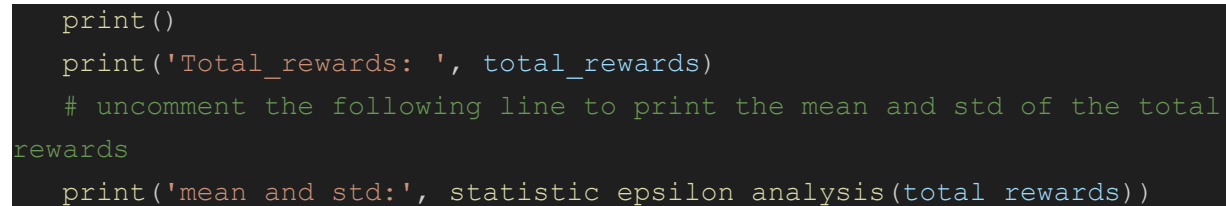


```
actions:
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2]
[0, 3, 1, 0, 0, 0, 1, 0, 0, 0, 1, 2]
[0, 3, 0, 0, 0, 0, 0, 0, 0, 1, 1, 2]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
Episode reward: -17.000000

Total_rewards: [-17]
mean and std: (-17, 0)
```

Calling `evaluate_policy` calls `actions_of_tabular_q` which prints 'actions:' + the actions derived from the q table, and then `evaluate_policy` takes the action table and evaluates it, conditionally printing a response and returning the `episode_reward`.

I added some code to print the total reward:



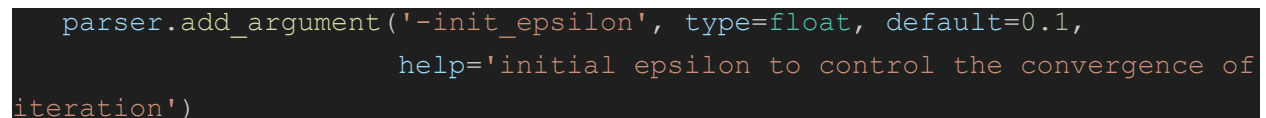
```
print()
print('Total_rewards: ', total_rewards)
# uncomment the following line to print the mean and std of the total
rewards
print('mean and std:', statistic_epsilon_analysis(total_rewards))
```

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

No, the Q-table does not converge. Over the course of the generated plot, there are many massive spikes in negative reward and no behavior that can be described as the leveling off / smooth decrease in magnitude of the negative reward. This means that there is a lack of convergence of the Q-table. The values do hover around a specific range of ~17-45, but this hovering is consistently interrupted by massive negative reward spikes. This is due to the static  $\epsilon$ .

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

The disadvantages of using a constant value of  $\epsilon$  is that the rate of exploration is held constant over the entire course of the episode. This is a problem, because it means that the model is restricted in terms of how much it can exploit learned knowledge versus how much it relies on random action expiration to determine its behavior.



```
parser.add_argument('-init_epsilon', type=float, default=0.1,
                    help='initial epsilon to control the convergence of
iteration')
```



In our case, `init_epsilon` is 0.1, which means that the exploration rate roughly corresponds to about 10% of choices in each episode being randomly generated through the function:

```
epsilon_greedy(tabular_q, state, epsilon=0.0):
```

What this means is that no matter what, 10% of the actions we take are randomly selected. This is a horrible approach because this low rate can hamper exploration in the beginning of each episode, where exploration would be more beneficial as we need to search for optimal actions through sheer trial and error. More importantly, however, this constant percentage is actually extremely detrimental to the model when it comes to being able to exploit learned information. Even if the agent learns optimal behavior that would allow it to reach the terminal state and for the q-table to converge to some consistent and optimal set of values, it would still result in the actual actions that an agent chooses to take being random 10% of the time, which can result in undesired behavior and/or an un-optimal policy such as the one pictured above.

I can even see that this is likely reflected in the actual negative rewards over the course of the episodes, where generally the values reduce down to ~17 - 40 with large spikes likely being the result of random choices resulting in the agent walking off of the cliff and being reset (hence large penalties). The fact that our  $\epsilon$  is constant is continually disrupting any chance for the Q-table to converge.

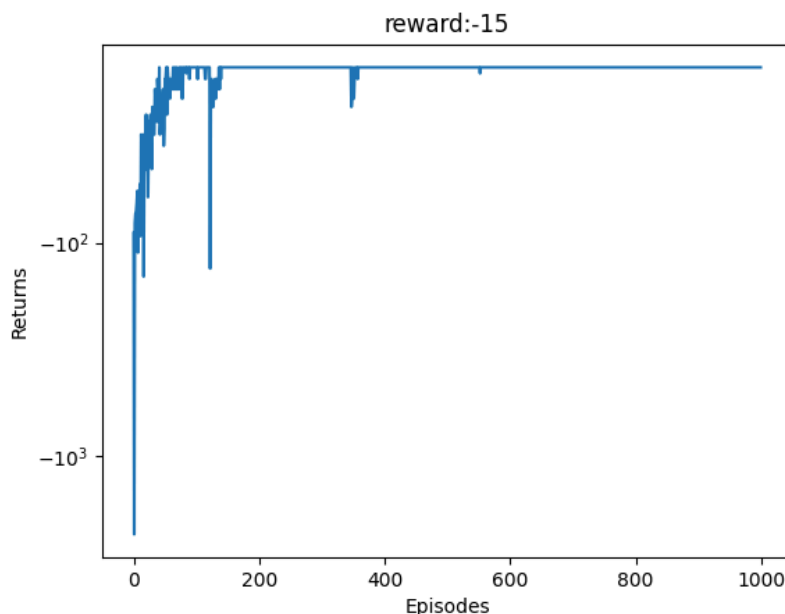
**(a)(5) Question: Should the value of  $\epsilon$  gradually increase or decrease as the number of episodes increases?**

The value of  $\epsilon$  should gradually decrease as the number of episodes increases. This is because we should try to explore space early on and then slowly begin to exploit knowledge about the environment that we have learned through exploration as we progress through the episodes. If we have learned valuable information about the environment, and our Q-table reflects this, we don't want  $\epsilon$  to be large because it will cause spiking behavior in the magnitude of the negative reward, even after convergence to some reward.

**(b) Optimization of epsilon  $\epsilon$**

- In this question, we adjust the value of  $\epsilon$  to optimize the SARSA model and make it converge. According to the convergence condition, please use  $\epsilon = \text{init\_epsilon} / k$  to improve the algorithm. The `init_epsilon` is the default setting of epsilon in the `get_args.py`, and `k` is the `k`th 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.

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



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

python Assignment-2/codebase\_main.py -method sarsa -seeds 1

```
actions:
[1, 1, 2, 1, 1, 1, 1, 2, 1, 1, 2, 2]
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2]
[0, 3, 3, 1, 1, 1, 0, 1, 1, 0, 1, 2]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
Episode reward: -15.000000

Total_rewards: [-15]
mean and std: (-15, 0)
```

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

This method helps SARSA to converge because it allows for a gradual shift from learning about the environment to exploiting learned information to obtain the most reward in the environment. In this case,  $\epsilon$  starts out at 0.1 and decreases over the course of the episodes to a value near zero in accordance with  $\epsilon / \text{number of episodes}$ . That means that the probability of choosing a random action decreases as the number of episodes increases. This allows for the agent to explore the environment in the beginning, but over time gradually shifts towards a more stable behavior, which is shown by the graph. In addition to the higher reward, the behavior shown on the graph is much more stable with respect to part II.(b)'s dynamic  $\epsilon$  than that of the SARSA with the constant  $\epsilon$  in part II.(a).

**(b)(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?**

SARSA does not generate the optimal policy with this setting. This is because the best reward possible is -13, as described/shown in the first section's question (b)(2) and (c)(1). So the total reward generated from part II.(b) is not the highest q value of this game, and SARSA does not generate the optimal policy with this setting.

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

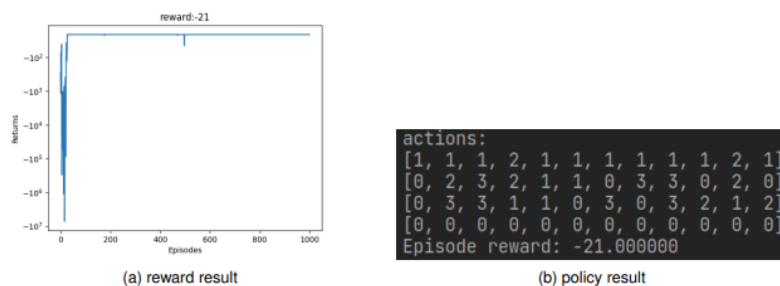


Figure 1: num\_step=10

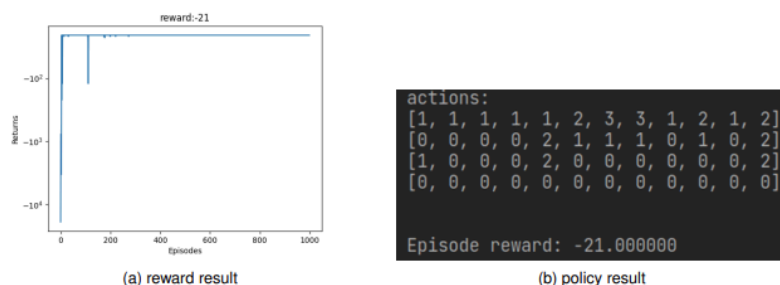


Figure 2: num\_step=100

• 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.

**(c)(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].)**

State-action pair (1, 2) indicates that the agent is at [0, 1], and the action is 2 which is 'move down'. So the agent would move down to state 13 at [1, 1].

- For policy in Figure 1, state 13 as a starting point for adhering to the policy would lead to the terminal state and reward would be -22.
- For policy in Figure 2, state 13 as a starting point for adhering to the policy would lead to the terminal state and reward would be -18.

State-action pair (17, 1) indicates that the agent is at [1, 5], and the action is 1 which is 'move right'. So the agent would move right to state 18 at [1, 6].

- For policy in Figure 1, state 18 as a starting point for adhering to the policy would lead to the terminal state and reward would be -8.
- For policy in Figure 2, state 18 as a starting point for adhering to the policy would lead to the terminal state and reward would be -9.

**(c)(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?**

Higher-step updates incorporate more future rewards into each update, which provides a more accurate estimate of the long-term return. That is why Steve's example is able to get a faster convergence of the Q-table and a subsequent faster convergence to the policy with a higher number of steps. In contrast, the lower step updates incorporate only more immediate future rewards into the updates, which leads to the increased variability in the start of the learning process that is pictured in the 10 step plot.

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

The total rewards of the 10-step and 100-step are both -21; the total rewards of my own 1-step SARSA with dynamic  $\epsilon$  are -15. So, the total rewards of the 1-step SARSA with dynamic  $\epsilon$  is higher than the total rewards of 10-step and 100-step in this instance. That being said, total rewards of 1-step SARSA with static  $\epsilon$  seem lower, however I think this is more by chance rather than one being more optimal than the other, because SARSA with static  $\epsilon$  fluctuates much more than any of the other examples.

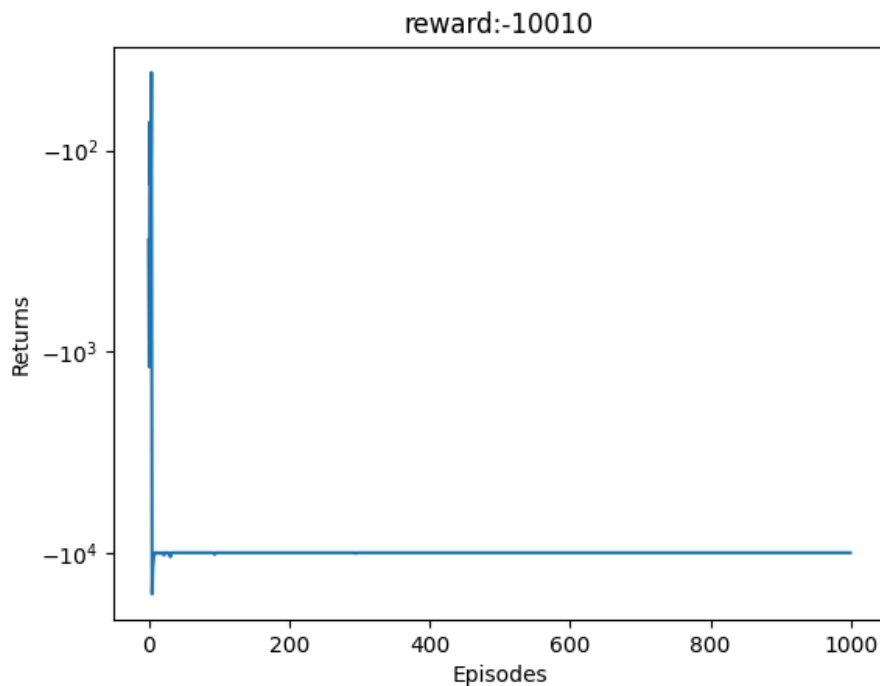
**(c)(4) Question: There is an "wrong" action in the policy of the 10-step SARSA shown in the screenshot of Figure 1. Can you find the state-action pair and explain why the policy finally selected the wrong action for this state? You can print out and analyze the q values of all the 4 actions of this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

A wrong action is located at state 3, position [0, 3] with action `2` or 'down'. This is a wrong action, because it diverts the agent from the otherwise more optimal path along the current policy of simply going to the right which would result in a reward of -19 to a path that forces a detour that costs the agent -2 additional negative reward. If it was `1` instead, this would remedy the wrong action and cause the agent to get a better reward.

- 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.

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

**python Assignment-2/codebase\_main.py -method sarsa -seeds 1 -num\_steps 10**



```
actions:
[1, 2, 2, 2, 1, 2, 1, 2, 1, 3, 2, 1]
[1, 0, 2, 3, 0, 2, 3, 1, 2, 2, 2, 0]
[0, 0, 3, 2, 1, 2, 0, 1, 1, 1, 2, 3]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
The agent didn't reach a terminal state in 100 steps.

Total rewards: [-100]
```

**(c)(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.)**

The trap is at state 35, position [2, 11]. The action is '3', which corresponds with 'move left'. The reason that this is a trap is due to the fact that agents must go to this location in order to get to the terminal state. There is no way to finish the episode without traversing downwards at state 35. So, state 35 in this specific instance is most definitely a trap.

### **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.

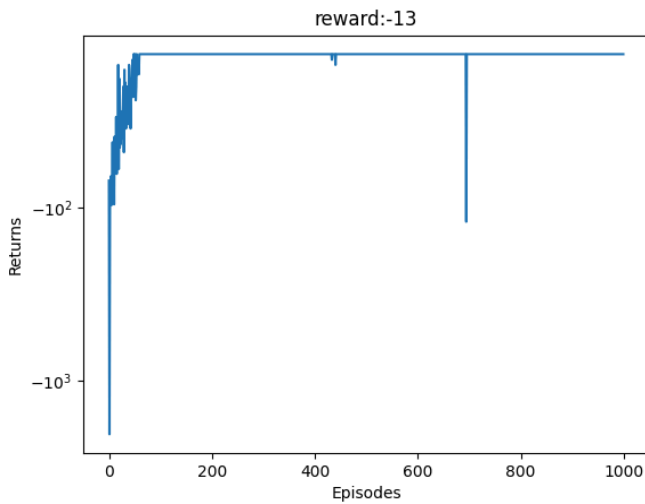
#### **(a) Implementation of Q-learning**

- Please reserve your optimization of  $\epsilon$  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".

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

**Note, that the init\_q\_value is never changed from 0.1 anywhere in the assignment, and default is 0.1 so I just didn't change anything for this part of this assignment.**

**python Assignment-2/codebase\_main.py -method q-learning -seeds 1**

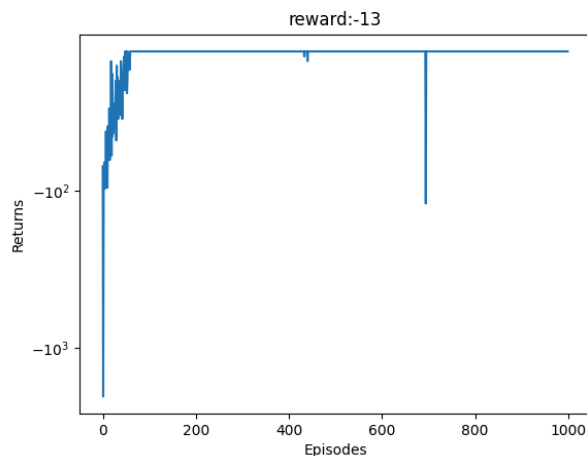
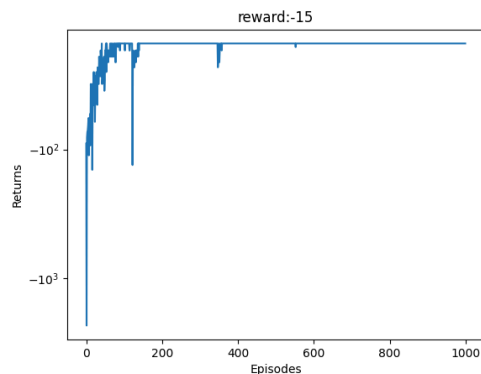


```
actions:
[1, 0, 3, 3, 1, 1, 2, 1, 2, 0, 2, 0]
[1, 0, 2, 0, 1, 1, 2, 2, 1, 1, 2, 2]
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
Episode reward: -13.000000
```

**(a)(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.)

I think Q-learning converges faster because it seems to take fewer episodes to converge to the optimal policy. Q-learning is off policy, so it learns optimal policy greedily independently of exploration ratio  $\epsilon$ . SARSA is on policy, which means its updates depend on the actions taken, which is greatly affected by the exploration ratio  $\epsilon$ .

SARSA with dynamic  $\epsilon$  (part II.(b)(1)):      |      Q-Learning with dynamic  $\epsilon$ :



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

Yes, Q-Learning has a higher total reward than the SARSA method because Q-Learning's path to the goal is shorter than SARSA. This is because q-learning is able to detect the path that goes along the cliff edge and exploit it, while SARSA is not, which results in SARSA taking extra steps to complete an episode compared to Q-Learning.

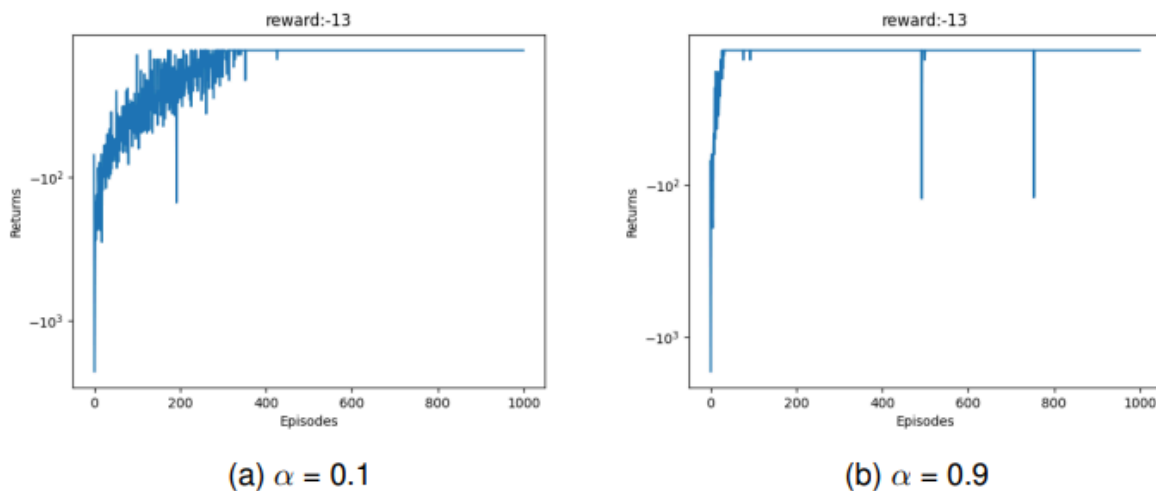
**(a)(4) Question: Is the total reward generated here the highest of the cliff walking?**

Yes, this total reward is the highest possible reward in the cliff walking environment because it takes the shortest possible path to the terminal state goal position from the initial state starting position. This path was provided in part 1(c)(1).

## **(b) Effect of Alpha $\alpha$**

• Continuing with Steve's experiment, he tested for different values for  $\alpha$  to see the effect on Q-learning. He set  $\alpha$  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.

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



**Figure 3: reward results for different  $\alpha$  values**

In Steve's experiment, there is no difference in the final reward of Q-learning between the two different values 0.1 and 0.9 because both learning rates result in convergence to the same reward of -13. However, if the total number of episodes was more limited to around, ~370, Steve would observe a difference in total reward. This is because  $\alpha = 0.1$  results in slower reward improvement, while  $\alpha = 0.9$  learns more quickly and reaches the optimal reward earlier. So, in



Steve's case, differences in  $\alpha$  affect the speed of learning more than the final outcome, assuming enough episodes are allowed.

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

A higher  $\alpha$  helps Q-learning to converge faster. This is because this parameter places a greater weight on recent experiences, which allows Q-value updates to be more significant after each step. This accelerates the rate at which the optimal q-value table is learned during the beginning of training, which is illustrated by the graph where it is shown that  $\alpha = 0.9$  leads to the almost immediate convergence to a reward of -13.

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

There are definitely fluctuations with respect to the plot depicting the behavior of the reward for  $\alpha = 0.9$ . This does not occur for the plot for  $\alpha = 0.2$ . This happens because a small  $\alpha$  slows down the learning process while a large  $\alpha$  speeds up the learning process.

The tradeoff between the two has to do with how much new information is weighted versus the old learned information in the Q-table. This is evident by looking at the tails of both graphs. At  $\alpha = 0.1$ , the reward takes longer to converge to -13, however once it does, there are no massive deviations in reward, which means that the Q-table for  $\alpha = 0.1$  is stable. This is evidenced by the fact that the tail is completely smooth after ~420 episodes. On the other hand, at  $\alpha = 0.9$ , the reward converges almost immediately to -13, however there are large, seemingly random spikes in negative reward in the tail of the graph. This is because the higher learning rate causes instability in the Q-table and large fluctuations based on noisy, heavily weighted immediate values that override optimal, learned Q-table values at the given episodes where the negative reward spikes occur.