Reinforcement Learning - Assignment Chapter 4

61175024H 白偉辰 Nick

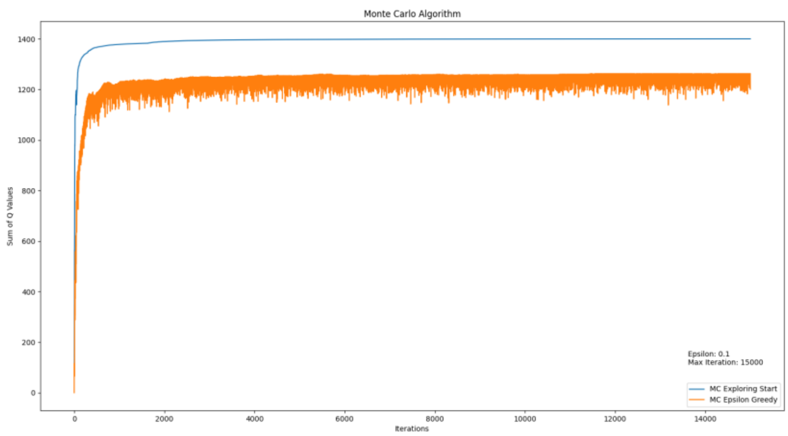
A.《 Implement MC-Epsilon Greedy and compare it with the given code example for MC-Exploring Start. 》

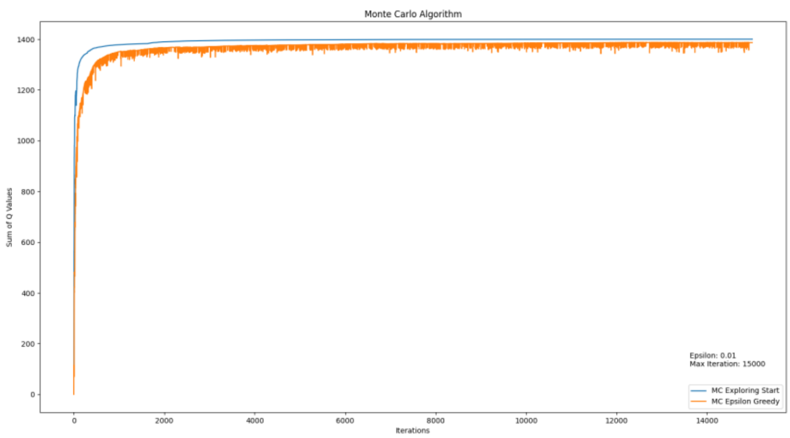
The main difference between class "MC_Exploring_Start" and class "MC_Epsilon_Greedy" in my code is function "policy" part.

After we get the "valid_actions" list for the current state coordinate, I added an if...else... statement and parameter "EPSILON" to implement Epsilon Greedy concept. Parameter "EPSILON" defined as 0.1. By using random() function to generate a float number between 0 and 1, for example let's call it "rand_num". If rand_num is bigger than the EPSILON, then choose the action which had maximum value in this state. If rand_num is smaller than the EPSILON, randomly choose an action from valid_actions.

By doing so, we can see the converged result from the figure shown below. Compare "MC Epsilon Greedy" with "MC Exploring Start", MC Epsilon Greedy has 10% chance to randomly select an action, so its line on chart is less compact to MC Exploring Start. If we decrease the EPSILON to 0.01, which means 1% chance to randomly select an action, then we will get closer range between MC Exploring Start and MC Epsilon Greedy. And the final optimal policy also shown as below.

```
Before (random policy), T = Target, W = Wall
------------------------------------------------------------
| T      | left   | right  | left   | left   | left   |
------------------------------------------------------------
| right  | right  | down   | up     | W      | up     |
------------------------------------------------------------
| down   | W      | up     | W      | down   | left   |
------------------------------------------------------------
| right  | right  | up     | left   | right  | left   |
------------------------------------------------------------

MC Exploring Start

Optimal policy, T = Target, W = Wall
------------------------------------------------------------
| T      | left   | left   | left   | left   | left   |
------------------------------------------------------------
| up     | up     | up     | left   | W      | up     |
------------------------------------------------------------
| up     | W      | up     | W      | right  | up     |
------------------------------------------------------------
| up     | left   | up     | left   | left   | up     |
------------------------------------------------------------

MC Epsilon Greedy

Optimal policy, T = Target, W = Wall
------------------------------------------------------------
| T      | left   | left   | left   | left   | left   |
------------------------------------------------------------
| up     | up     | left   | up     | W      | up     |
------------------------------------------------------------
| up     | W      | up     | W      | right  | up     |
------------------------------------------------------------
| up     | left   | left   | left   | left   | left   |
------------------------------------------------------------
```





PseudoCode <policy> in class MC_Epsilon_Greedy
Input: state_coordinate

```
for all possible action in action dictionary
    if next state coordinate not equal to state coordinate
        add action into valid_action list
if random number bigger than threshold # exploitation
    for action in valid_action list
        add each action's Q value into Q_value list
    get max value from Q_value list
    for action in valid_actions list
        if max_value equal to action's Q value at current state coordinate
            get action's index via action_to_number dictionary and add into indexes list
    return >>> best action
else random number smaller than threshold # exploration
    for action in valid_action list
        get action's index via action_to_number dictionary and add into indexes list
return >>> random choose an action from action dictionary with indexes list
```

```python
    def policy(self, state_coordinate): #Optimal policy, find the maximun Q(s,a) and return action
        Q_value = []
        valid_actions = []
        state_coordinate = np.array(state_coordinate)
        indexes = []
        for action in self.env.action_dict.keys():
            # if state coordinate == next state coordinate, \
            # which means the action is invalid cause "transfer_state" return the same coordinate
            if not (state_coordinate == self.env.transfer_state(state_coordinate, action)).all():
                valid_actions.append(action)

        if np.random.random() > EPSILON: # exploit
            for valid_action in valid_actions:
                Q_value.append(self.Q_values[(tuple(state_coordinate), valid_action)])
            max_value = max(Q_value)

            for valid_action in valid_actions:
                if max_value == self.Q_values[(tuple(state_coordinate), valid_action)]:
                    indexes.append(self.env.action_to_number[valid_action])
            return self.env.direction_dict[random.choice(indexes)]
        else: # explore
            for valid_action in valid_actions:
                indexes.append(self.env.action_to_number[valid_action])
            return self.env.direction_dict[random.choice(indexes)]
```

B.《 In this regard, change the example slide 31 so that from s13 a drone can take our robot to s1, and from s9 same drone can take it to s6. (note that you need to add a new action called fly). 》

First, added "fly" action into action dictionary, with [0, 0] value (later will give it different value based on different state, s9 or s13).

```
        self.action_dict = {"up": [-1,0], "right": [0, 1], "down": [1,0], "left":[0,-1], "fly": [0,0]} # [row, column]
```

Second, at function "transfer_state", I determined whether the input state_coordinate is s9[1, 3] or s13[2, 4] or other state coordinate.

If state_coordinate is s9[1, 3], then see what's the random input action are given. If the random action is "fly", then the drone will take the robot from s9[1, 3] to s1[1, 0], which means robot need to stay on the same row but go left for 3 columns. As the code shown below, <state_coordinates + [0, -3]>. Otherwise, if the random action is not fly but other actions (up, down, right, left), then just add the moving action value according to the direction dictionary.

Same concept for s13, if the input state_coordinate is s13[2, 4], and the random action is "fly". Then, the robot will move from s13[2, 4] to s1[0, 1], which means the robot need to go up for 1 row and go left for 3 columns. As the code shown below, <state_coordinates + [-2, -3]>. Otherwise, if the random action is not fly but other action (up, down, right, left), then just add the moving action value according to the direction dictionary.

And if the input state_coordinate is neither s9 nor s13, then it must be other state in range s1, s2, ..., s8, s10, s11, s12, s14, ..., s20. In this case, just add the moving action value according to the direction dictionary. By doing so, we can calculate the sum of state_coordinate and action value, and get the next state coordinate.

After getting the coordinate for the next state, we determine whether next state coordinate is out of the board or not. Range from 0 to 3 for row, and 0 to 5 for column. If next state coordinate is out the board, then just return the value of the input state coordinate as we assign as "current_state_coordinate" at the very beginning.

At last, we can assure that the next state coordinate is in the grid, then we need to check whether the next state is a "wall" or not. If next state is wall, apparently we can not choose it as our next step, therefore just return the "current_state_coordinate".

The optimal policy is shown below, at state s9[1, 3] and s13[2, 4] both will take the fly action. Even s10[1, 5] and s18[3, 3] also influence by s13, both of the actions are try to go to state s13 for shorter route.

```
Find optimal policy using Monte Carlo algorithm with exploring starts
Before (random policy), T = Target, W = Wall
-------------------------------------------------------------
| T      | left   | right  | left   | left   | left   |
-------------------------------------------------------------
| right  | right  | down   | up     | W      | up     |
-------------------------------------------------------------
| down   | W      | up     | W      | fly    | down   |
-------------------------------------------------------------
| right  | right  | up     | left   | right  | left   |
-------------------------------------------------------------

Optimal policy, T = Target, W = Wall
-------------------------------------------------------------
| T      | left   | left   | left   | left   | left   |
-------------------------------------------------------------
| up     | up     | up     | fly    | W      | down   |
-------------------------------------------------------------
| up     | W      | up     | W      | fly    | left   |
-------------------------------------------------------------
| up     | left   | up     | right  | up     | left   |
-------------------------------------------------------------
```

61175024H 白偉辰Nick

PseudoCode <transfer_state>
Input: state_coordinate, action

current_state_coordinate = state_coordinate
if state_coordinate = s9[1, 3]
    if action = fly >>> next_state_coordinate = state_coordinate + [0, -3]
    else >>> next_state_coordinate = state_coordinate + action_dictionary[action]
else if state_coordinate = s13[2, 4]
    if action = fly >>> next_state_coordinate = state_coordinate + [2, -3]
    else >>> next_state_coordinate = state_coordinate + action_dictionary[action]
else >>> next_state_coordinate = state_coordinate + action_dictionary[action]

if next_state_coordinate is not in board[0~3, 0~5]
    return current_state_coordinate

if next_state equals to “wall”
    return current_state_coordinate
otherwise, return next_state_coordinate

```python
    def transfer_state(self, state_coordinates, action): #Input(state, action), Output(next state)
        current_state_coordinates = state_coordinates
        if np.array_equal(state_coordinates, np.array([1, 3])): # state_coordinate == s9[1, 3]
            if action == "fly":
                next_state_coordinates = state_coordinates + [0,-3] # fly from s9[1, 3] >>> [up=0, left=-3] >>> to s6[1, 0]
            else:
                next_state_coordinates = state_coordinates + self.action_dict[action]
        elif np.array_equal(state_coordinates, np.array([2, 4])): # state_coordinate == s13[2, 4]
            if action == "fly":
                next_state_coordinates = state_coordinates + [-2,-3] # fly from s13[2, 4] >>> [up=-2, left=-3] >>> to s1[0, 1]
            else:
                next_state_coordinates = state_coordinates + self.action_dict[action]
        else:
            next_state_coordinates = state_coordinates + self.action_dict[action]

        if next_state_coordinates[0] < 0 or next_state_coordinates[0] > 3 \
            or next_state_coordinates[1] < 0 or next_state_coordinates[1] > 5:# Out of board
            return current_state_coordinates
        next_state = self.grid_world[next_state_coordinates[0]][next_state_coordinates[1]]
        if next_state == "W": #Hit the wall
            return current_state_coordinates
        return next_state_coordinates
```