EE619 Project #1

20233508 DoHyoung Lee

1. Implementation of Algorithms

- Q-learning : Q-learning is model-free and off-policy learning. Q-learning Agent updates the Q function according to the difference between the target and old-estimation.
- Double Q-learning : Double Q-learning uses two different Q functions. Each Q functions divides its target as selection and evaluation, and it prevents overestimation of value.

2. Implementation of Algorithms

```
import numpy as np
from collections import defaultdict

class Q_learning():
    def __init__(self, environment, epsilon=0.05, alpha=0.01, gamma=0.99):
        self.Q_table = defaultdict(lambda: np.zeros(4)) # Q_table =
{"state":(Q(s,UP),Q(s,DOWN),Q(s,LEFT),Q(s,RIGHT),)}
        self.epsilon = epsilon
        self.alpha = alpha
        self.gamma = gamma
        self.actions = environment.actions # ['UP', 'DOWN', 'LEFT', 'RIGHT']
        self.env_row_max = environment.row_max
        self.env_col_max = environment.col_max

    def get_Q_table(self):
        return self.Q_table

    def action(self, state):
        Q_table = self.get_Q_table()
        max_index = np.argmax(Q_table[state])
        probability = np.zeros(4)
        for i in range(0, len(probability)):
            if i == max_index:
                probability[i] = 1 - self.epsilon + self.epsilon/len(probability)
            else:
                probability[i] = self.epsilon/len(probability)
        action_index = np.random.choice(4, 1, p=probability).item()
        return self.actions[action_index]

    def update(self, current_state, next_state, action, reward):
        action_index = {"UP": 0, "DOWN" : 1, "LEFT" : 2, "RIGHT" : 3}
        self.Q_table[current_state][action_index[action]] =
(self.Q_table[current_state][action_index[action]] +
                                                self.alpha * (reward + self.gamma *
np.max(self.Q_table[next_state]) - self.Q_table[current_state][action_index[action]]))


    def get_max_Q_function(self):
        max_Q_table = np.zeros((self.env_row_max, self.env_col_max))
        Q_table= self.get_Q_table()
        for i in range(0, self.env_row_max):
            for j in range(0, self.env_col_max):
                max_Q_table[i][j] = np.max(Q_table[(i, j)])

        return max_Q_table
```

```
class Double_Q_learning():
    def __init__(self, environment, epsilon=0.05, alpha=0.01, gamma=0.99):
        self.Q1 = defaultdict(lambda: np.zeros(4)) # Q_table =
{"state":(Q(s,UP),Q(s,DOWN),Q(s,LEFT),Q(s,RIGHT),)}
        self.Q2 = defaultdict(lambda: np.zeros(4))
        self.Q_table = defaultdict(lambda: np.zeros(4))
        self.epsilon = epsilon
        self.alpha = alpha
        self.gamma = gamma
        self.actions = environment.actions # ['UP', 'DOWN', 'LEFT', 'RIGHT']
        self.env_row_max = environment.row_max
        self.env_col_max = environment.col_max

    def get_Q_table(self):
        return self.Q_table

    def action(self, state):
        Q_table = self.get_Q_table()
        max_index = np.argmax(Q_table[state])
        probability = np.zeros(4)
        for i in range(0, len(probability)):
            if i == max_index:
                probability[i] = 1 - self.epsilon + self.epsilon/len(probability)
            else:
                probability[i] = self.epsilon/len(probability)
        action_index = np.random.choice(4, 1, p=probability).item()

        return self.actions[action_index]

    def update(self, current_state, next_state, action, reward):
        action_index = {"UP": 0, "DOWN" : 1, "LEFT" : 2, "RIGHT" : 3}
        if np.random.randint(1) == 0:
            self.Q1[current_state][action_index[action]] =
(self.Q1[current_state][action_index[action]] +
                                                self.alpha * (reward + self.gamma *
self.Q2[next_state][np.argmax(self.Q1[next_state])] -
self.Q1[current_state][action_index[action]]))
        else:
            self.Q2[current_state][action_index[action]] =
(self.Q2[current_state][action_index[action]] +
                                                self.alpha * (reward + self.gamma *
self.Q1[next_state][np.argmax(self.Q2[next_state])] -
self.Q2[current_state][action_index[action]]))
        self.Q_table[current_state][action_index[action]] =
self.Q1[current_state][action_index[action]]
+  self.Q2[current_state][action_index[action]]

    def get_max_Q_function(self):
        max_Q_table = np.zeros((self.env_row_max, self.env_col_max))
        Q_table= self.get_Q_table()
        for i in range(0, self.env_row_max):
            for j in range(0, self.env_col_max):
                max_Q_table[i][j] = np.max(Q_table[(i, j)])

        return max_Q_table
```

3. Policy Figures

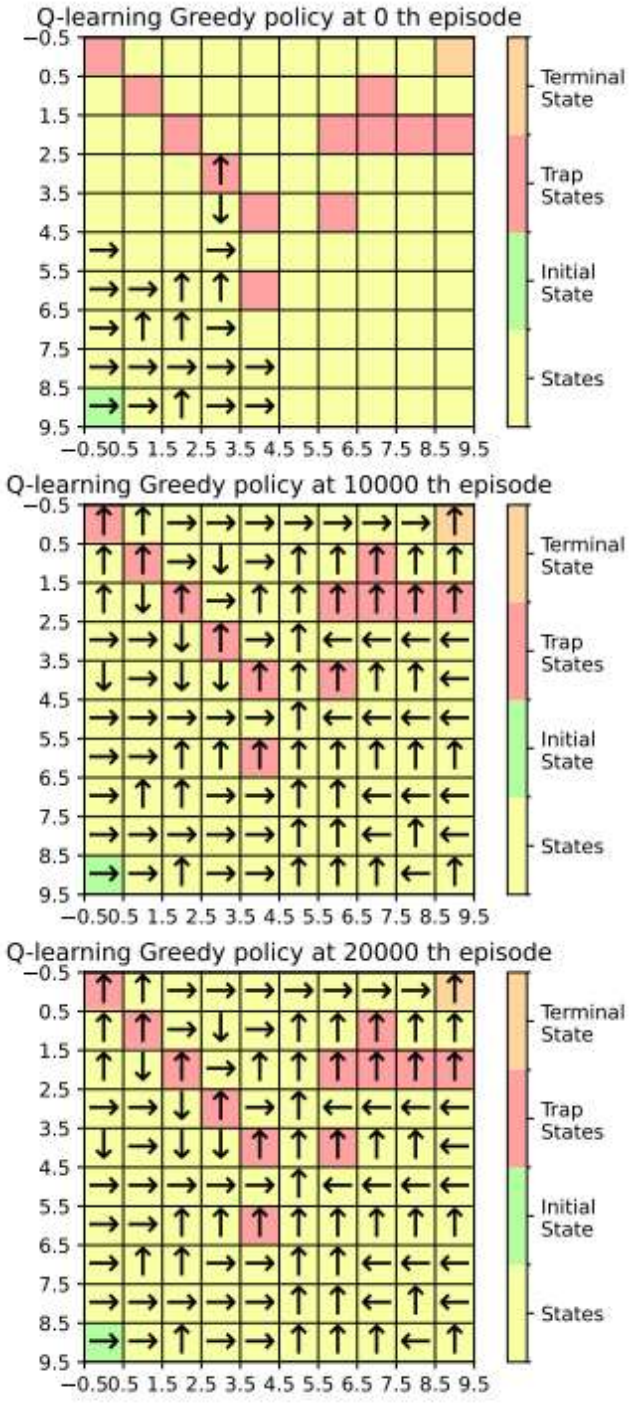

Figure 1. Q-learning Greedy Policy

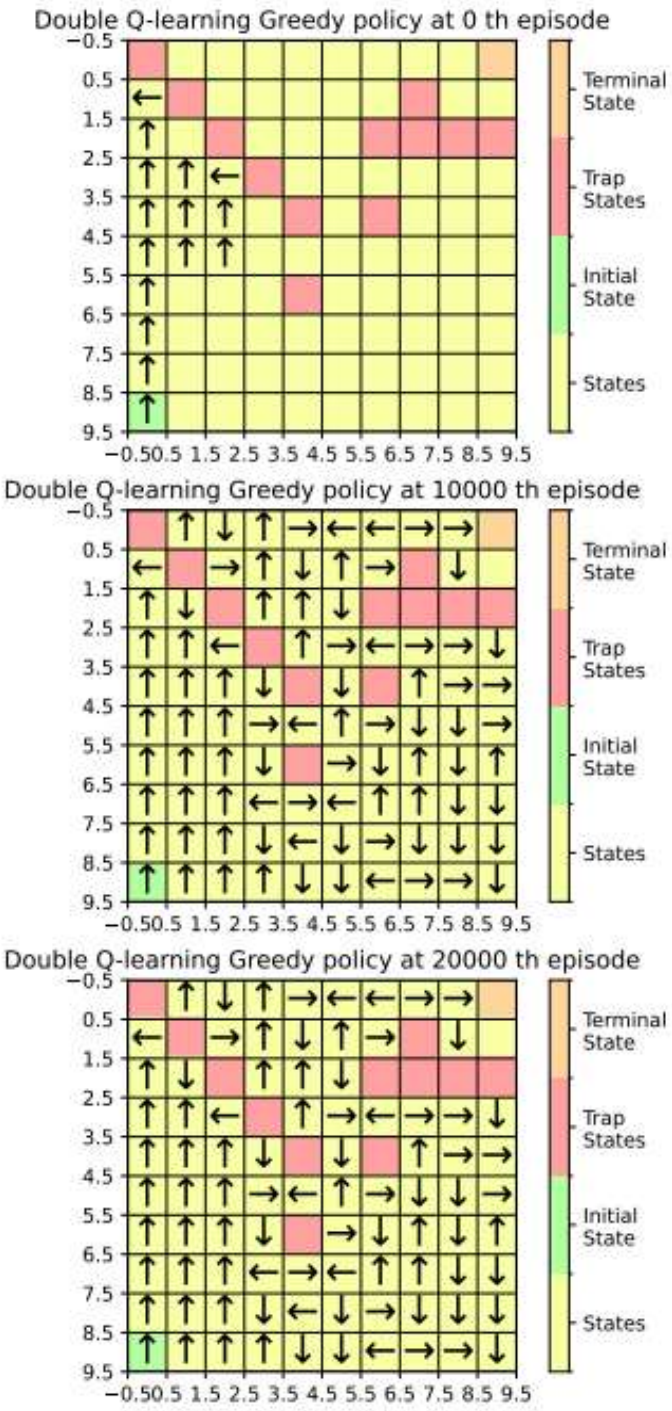

Figure 2. Double Q-learning Greedy Policy

4. Learning Curve and Max_Q for Each State

4-1)

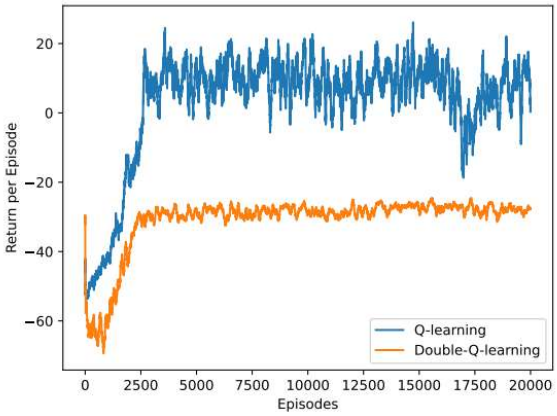

Figure 3. Learning Curve of Q-learning and Double Q-learning

In Figure 3, both Q-learning and Double Q-learning converges to specific return, the expectation of rewards, and that of Q-learning is more than that of Double Q-learning. Also, in Q-learning, Q-learning had a significantly different return per episode than Double Q-learning. The convergence of Double Q-learning into specific return is meaningful since that means the policy from Double Q-learning suggests specific way to visit the terminal state.

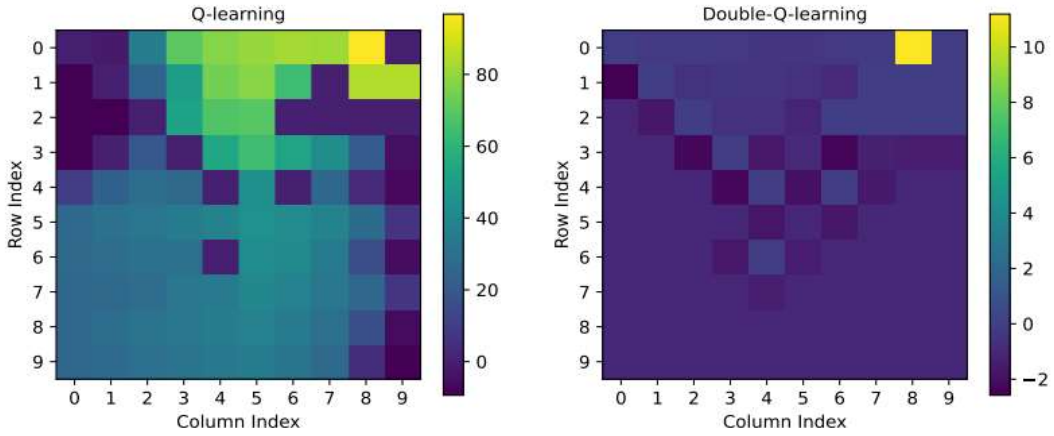

Figure 4. Maximum value of Q Function of Each State

Figure 4 shows that the maximum value of Q function of (Double) Q-learning. Both Q-learning and Double Q-learning algorithm draw max Q matrix. We can see Double Q-learning matrix values are remarkably low than that of Q-learning matrix. Figure 5 is a numerical representation of Figure 4, rounded to the second decimal place. According to Figure 5, max values of Q function in each state acquired from Double Q-learning are in the range of [-4, 1]. It means division of action selection and action evaluation really works. For example, assume that there are Q1 and Q2 functions which are used for Double Q-learning. When Q1 updates, the target of Q1 is sum of reward when the agent take some action and Q2 value of next state. That means Q1 cannot consider the expected reward of Q1 in the next state. Q1 value cannot be exploded, and same with Q2.

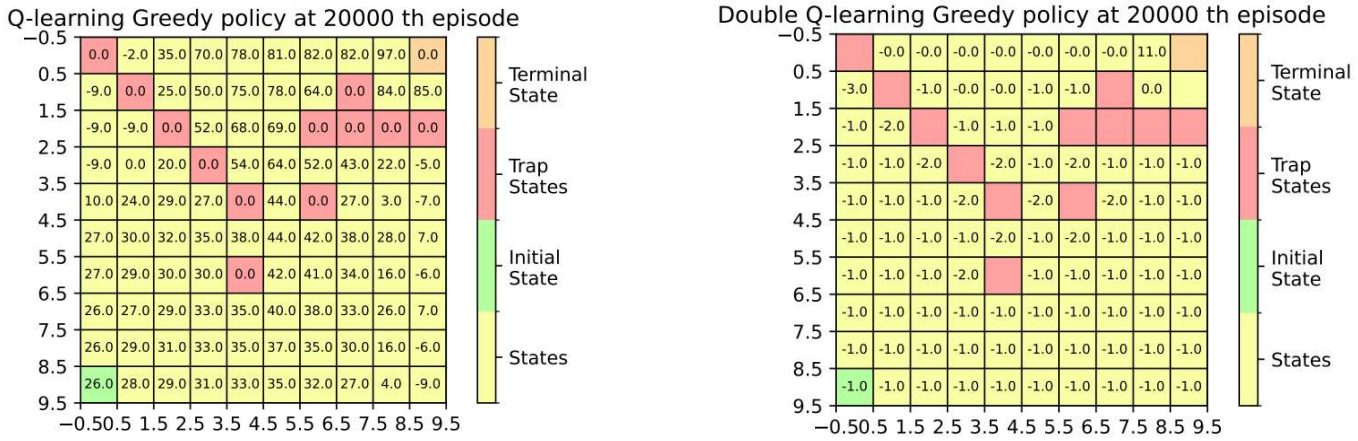

Figure 5. Numerical Data of Maximum Value of Q Function

4-2)

Then, is Q-table overestimates the expected reward? The answer is yes. We already know that Double Q-learning is unbiased, and the Q-table values are much higher than that of Double Q-table. It suggests that Q-learning is overestimated.

However, whether to overestimate is not a significant issue in this situation. In Figure 3, when we compare the average return, sum of discounted rewards and each rewards are really taken through exploring of the agents, we can see return of Q-learning is significantly higher than that of Double Q-learning. That means Q-learning suggests 'shorter' path to reach the terminal state, and we can see this tendancy after 3000th episode of Figure 3.

To sum up, we can presume that Q-table is overestimated by comparison with Double Q-table. However, in this situation, overestimation is not important since Q-learning agent found shorter pathway to terminal state, and this can be analyzed from return per epochs.