Reinforcement Learning Assignment

Li Hantao, G2101725H, MSAI, hli038@e.ntu.edu.sg

In this project, you will learn how to implement one of the Reinforcement Learning algorithms (e.g., Qlearning, MC, SARSA, or even value iteration, policy iteration) to solve the BoxPushing grid-world game with various difficulty levels. Novel RL ideas are welcome and will receive bonus credit. In this assignment, you need to implement the code on your own and present convincing presentation to demonstrate the implemented algorithm.

I. Introduction

We implement two reinforcement learning algorithms, including **Q-Learning** and **SARSA**, to implement the solving of the Box-pushing grid-world game. In this report, we will show the basic environment and algorithms, with the corresponding curves and graphs to explain the experiment results.

A. Grid-world Environment

The environment is a 2D grid world as shown in Fig.1. The size of the environment is 6×14. In Fig. 1, **A** indicates the agent, **B** stands for the box, **G** is the goal position, and filling blocks means lava or cliff. (In the provided description document, the position of **G** is (4,13), while the position in the provided code is (4,12). Here, the position in the code shall prevail.)

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
0														
1														
2														
3														
4		B		▓	▓	▓	▓	▓	▓	▓	▓	▓	G	
5	A		▓	▓	▓	▓	▓	▓	▓	▓	▓	▓	▓	

*You can use the *DrawEmptyMap()* function, included in the code file, to generate such a figure.

Fig. 1. Sketch map of the box-pushing environment.

The game ends under three conditions:

1). The agent or the box steps into the dangerous region.

2). The current time step attains the maximum time step of the game.

3). The box arrives at the goal.

B. MDP Formulation

The MDP formulation is described as follows:

- State: The state consists of the position of the agent and the box. In Python, it is a tuple, for example, at time step 0, the state is ((5, 0), (4, 1)) where (5, 0) is the position of the agent and (4, 1) is the position of the box.

- Action: The action space is [1,2,3,4], which is corresponding to [up, down, left, right]. The agent needs to select one of them to navigate in the environment.

- Reward: The reward is calculated based on agent's movement. The final reward is a summation of three values at each time step. The three values are -1 for each movement, the negative value of the distance between the box and the goal as well as the negative value of the distance between the agent and the box. In addition to that, agent will also receive a reward of -200 if the agent or the box falls into lava and cliff. The (1) is the formulaic description of reward $\boldsymbol{r_t}$.

$$\boldsymbol{r_t} = \begin{cases} -1 - dist(\boldsymbol{B}, \boldsymbol{G}) - dist(\boldsymbol{B}, \boldsymbol{G}) & if\ no\ falling \\ -1 - dist(\boldsymbol{B}, \boldsymbol{G}) - dist(\boldsymbol{B}, \boldsymbol{G}) - 200 & if\ falling \end{cases} \quad (1)$$

- Transition: Agent's action can change its position and the position of the box. If a collision with a boundary happens, the agent or the box would stay in the same position.

II. Reinforcement Learning Algorithms

A. Q-Learning

The author of Q-learning is Watkins, who proposed this classical algorithm in his Ph.D. dissertation 'Learning from Delayed Rewards' in 1989. [1]

Q-learning is a value-based algorithm. It utilizes Q-table to record the estimated Q-values of different actions with distinct states, which indicates as (state, action) in the Q-table. Before exploring the environment, the Q-table will be initialized. (In this task, we initialize them to [0, 0, 0, 0].)

When the agent searches the environment, it will iteratively update Q(s, a) with the Bellman equation. With the increase of the iterations, the agent will know more about the 'world', and the Q-function can be fitted better until it converges or reaches the set number of iterations. The updating rule of the Q-values is shown in (2).

$$Q_{new}(S_t, A_t) = Q_{old}(S_t, A_t) + \alpha(R_{t+1} + \gamma \max_a Q_{old}(S_{t+1}, A_t) - Q_{old}(S_t, A_t)) \quad (2)$$

where α is the learning rate; γ is the discount factor; t is the current time step (episodes).

On the other hand, we can use another way to understand this equation, which is, thinking $(1-\alpha)Q_{old}$ as the proportion of the old Q-value take into the new Q-value, while the remaining part as the rewards learned in this action (by the action itself and potential rewards in the future). The pseudo-code of Q-learning

iterative process is shown in Fig. 2.

Initialize $Q(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(terminal\text{-}state,\cdot) = 0$
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'
    $Q(S,A) \leftarrow Q(S,A) + \alpha\left[R + \gamma \max_a Q(S',a) - Q(S,A)\right]$
    $S \leftarrow S'$;
  until S is terminal

Fig. 2. Pseudo code of Q-learning iterative process. [2]

B. SASAR

SASRA (State-Action-Reward-State-Action) is an algorithm for learning Markov decision process strategies. Rummery and Niranjan introduced the algorithm in the technical paper '*Modified Connectionist Q-Learning,*' and Sutton mentioned the alias SASRA in the footnote. [3] This learning system was considered as the pioneer of Q-learning algorithm in 1997.

The 'State-Action-Reward-State-Action' clearly reflects the five values on which the learning update function depends, namely, the current state S_t, the action A_t selected in the current state, the reward r_t obtained, the state S_{t+1} obtained after executing A_t in S_t, and the action A_{t+1} to be executed in S_{t+1} state. The core idea of the algorithm can be simplified as follows:

$$Q_{new}(S_t, A_t) = Q_{old}(S_t, A_t) + \alpha(R_{t+1} + \gamma Q_{old}(S_{t+1}, A_{t+1}) - Q_{old}(S_t, A_t)) \quad (3)$$

We can notice that the significant difference between Q-learning and SARSA is the updating method, which means we use the original A_{t+1} in the state S_t instead of the optimize A after the Q-value is updated, like Q-learning method. The pseudo-code of SARSA iterative process is shown in Fig. 3.

Initialize $Q(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(terminal\text{-}state,\cdot) = 0$
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)
    $Q(S,A) \leftarrow Q(S,A) + \alpha\left[R + \gamma Q(S',A') - Q(S,A)\right]$
    $S \leftarrow S'$; $A \leftarrow A'$;
  until S is terminal

Fig. 3. Pseudo code of SARSA iterative process. [4]

III. Procedures

We use the above two algorithms for training, respectively. We use the **ε-Greedy** algorithm to speed up the learning rate and the randomness in the learning process; that is, whenever the agent decides the action, it has the probability of ε to explore randomly instead of using the action in the Q-table.

In this way, the algorithm can balance '*following experience*' and '*exploring new behavior.*' While not being conservative, trying the strategy with higher income as much as possible to speed up the progress of optimization strategy.

In the first training, we regard a training process with $\boldsymbol{T}$ times of iterations (episodes) as a complete experiment, and we repeat the whole experiment $\boldsymbol{N}$ times. The parameters are shown in Table 1.

TABLE I
Training Parameters

Learning Rate $\boldsymbol{\alpha}$	0.5
Discount factor $\boldsymbol{\gamma}$	0.99
Number of episodes $\boldsymbol{T}$	10000
Number of experiments $\boldsymbol{N}$	20
ε-greedy $\boldsymbol{\varepsilon}$	0.01

A. Learning Progress - Episode Rewards vs. Episodes

The episode rewards vs. episodes curves of the learning progress have been shown in Fig. 4. Due to the high density of $\boldsymbol{T}$ data points, sampling at an interval of 10 is adopted.

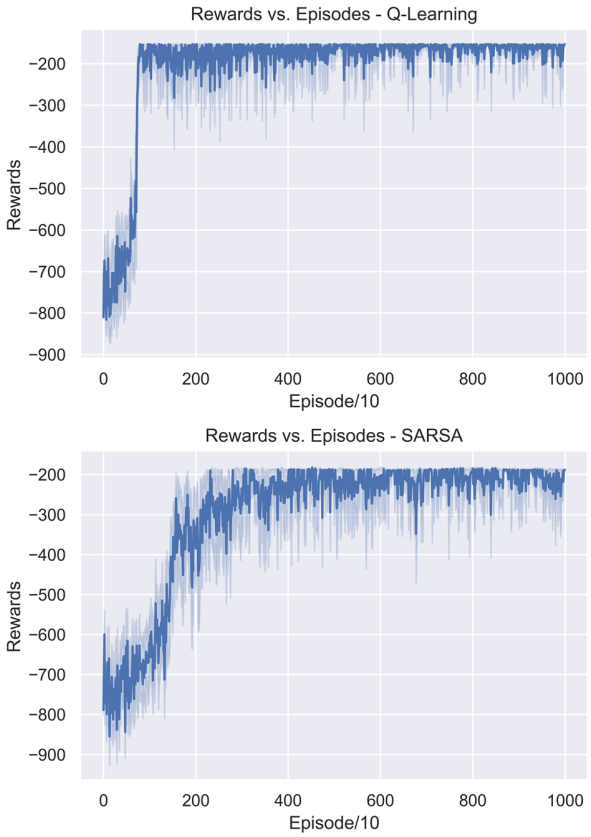

Fig. 4. Curve of training result with different algorithms.

We can notice that both algorithms can converge to their corresponding optimized path, completing the learning. (The specific paths will be discussed later.) In addition, we can see some distinctions between the two algorithms from Fig. 4.

Firstly, Q-learning's learning speed is faster than SARSA, which will also obtain a better path. Secondly, the oscillation in the later stage of SARSA is greater than that of Q-learning.

The phenomena seem to go against our intuition: SARSA tends to learn a longer but safer route, while Q-learning favors a better but dangerous path. This appearance often leads to more significant fluctuation of the Q-learning algorithm in the later stage of learning, and the rewards are lower because they often fall off the cliff. [2]

Subsequently, we will discuss the possible reasons why the phenomena in the experiment are diverse from the theoretical intuition.

B. *Final Q-Table*

Firstly, we should examine the Q-table finally learned by the two algorithms in the experiment and the final optimal path. Since the Q-table will contain four actions of each block in each state, it is unrealistic to show it in the report. Therefore, we have written the function DrawMap() to draw Q-table in a specific state in the code file. In Fig. 5, we show a Q-table in a particular state of the SARSA algorithm.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	←	←	←	←	→	→	↑	↓	→	↑	→	↓	↓	
1	←	→	→	→	→	→	←	←	←	↓	→	A	↓	
2	↓	↓	→	→	→	→	↓	↓	←	→	↓	↑	B	
3			↓	↓	↓	←	→	→	↑	←	←	↓	↑	
4				▓▓	▓▓	▓▓	▓▓	▓▓	▓▓	▓▓	▓▓	▓▓	G	
5			▓▓	▓▓	▓▓	▓▓	▓▓	▓▓	▓▓	▓▓	▓▓	▓▓	▓▓	

Agent Action Now: →

*You can use the *DrawMap()* function, included in the code file, to generate such a figure.

Fig. 5. Sketch map of the final Q-table in specific state.

It can be observed in Fig. 5 that in a specific state, the optimal action given by the Q-table is 'Agent Action Now,' which is, go to the RIGHT. Arrows in other locations in Fig. 5 represent the corresponding optimal action when the agent is in such a position while the box is in (2,12).

We can notice that the optimal actions of the agent are reasonable in the block near the box. However, there will be unreasonable actions at distant locations, such as (2,7), as a deadlock; since it is challenging for the algorithms to learn the strategy far from the optimal state under a small ε.

We can get the optimal route of the two algorithms by combining the Q-table visualization similar to Fig. 5 and the agent motion sequence output by the function. The optimal routes of the two algorithms have shown in Fig. 6.

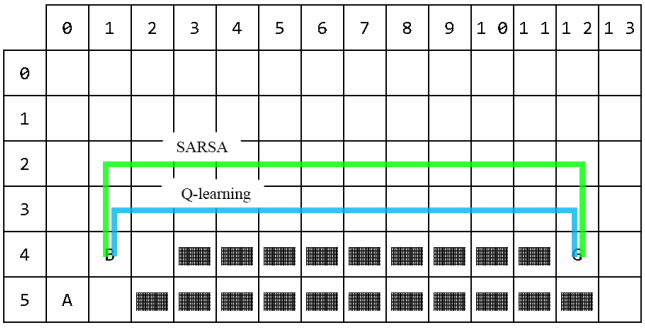

Fig. 6. Sketch map of the final policy(path).

We can notice that the paths learned by the two algorithms are consistent with the intuition discussed earlier. However, another intuition is inconsistent. We believe that the reason lies in the particularity of the reward function.

In this experiment, the reward is directly proportional to the distance between box and goal each time. Because the path of SARSA is more distant, requiring more operations, the reward obtained by the SARSA must be lower than that of the Q-learning, without considering the degree of ε-greedy. Secondly, we assume that the oscillation of the Q-learning algorithm is due to the negative reward caused by falling off the cliff when we are 'greedy,' which should be much more numerous than the negative reward caused by taking the wrong way. However, when it takes at least two actions to return to the original path every time we go wrong, and the negative return of falling off the cliff is not significant, the reward oscillation in the Q-learning algorithm cannot be observed in this experiment.

C. *Discussion*

To verify the point of view, we designed another group of controlled experiments. We will use it this time ε= 0.1, while the other parameters are consistent. The experimental results are shown in Figure 7.

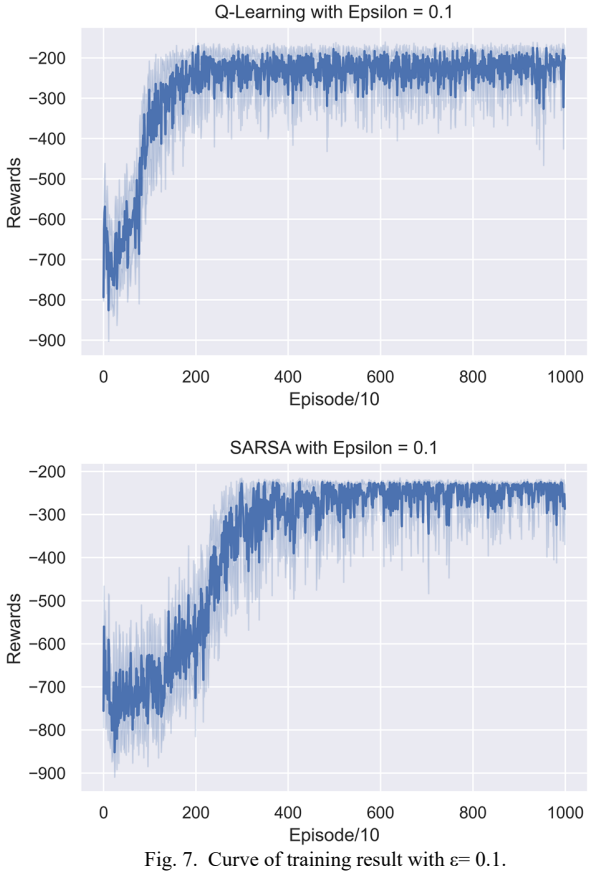

Fig. 7. Curve of training result with ε= 0.1.

When we increase the ε-greedy probability, the agent has a bigger chance of falling off a cliff or taking the wrong way; thus,

the reward gap, which is not evident in the original two cases, will be widened, making the oscillation phenomenon of Q-learning algorithm more obvious. We can see from Fig. 7 that the amplitude of oscillation in the second half of the Q-learning algorithm is significantly more apparent than in the SARSA, where even cannot be fitted to the original optimal rewards, indicating that our inference is correct.

D. Learning Progress - Q Values vs. Episodes

In addition to the above discussion, we can also observe the changes of Q-value corresponding to a specific state during the training. We have written the code to draw Q-values vs. episodes curves, which can output the development of Q-value corresponding to four actions under a specific state with iterations.

Fig. 8 shows an example. We select the state where the agent is in (4,1) and the box (3,1). This state is the branch point of separate paths of the two algorithms, that is, the Q-learning algorithm, which tends to take risks, will choose to move left in this state to push the box along the cliff, while the SARSA algorithm, which tends to be safe, will choose to continue to move up.

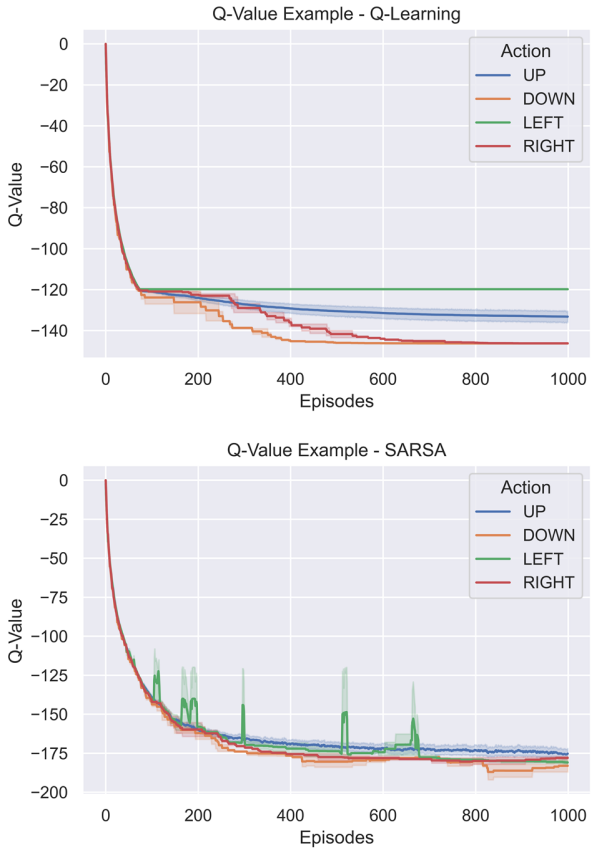

Fig. 8. Curve of Q-value in specific state.

The above conclusions can be examined from the results of the four actions in Fig. 8. Moreover, in the Q-learning algorithm, the Q-values of the four actions have been clearly distinguished from each other; that is, the trend of going left as the optimal strategy is 'solid.' However, the model seems to have no significant tendency towards the optimal strategy of going up in SARSA.

We believe that the reason for this result is the one discussed above. The design of the reward function makes the SARSA unable to judge the meaning of the 'safe' path, since taking the wrong road and falling off a cliff look equally 'unsafe.' Therefore, the strategy of SARSA for this critical branch state has been struggling.

IV. Improvement

In previous experiments, we noticed an obvious problem: the second half of the two algorithms continued to oscillate instead of fitting the best reward. The reason is the existence of the ε-greedy algorithm. After the model has found the globally optimal route, the ε-greedy still misguides the model with a constant probability, which leads to the failure of the terminated fitting.

It is relatively uncomplicated to solve this problem. We introduce a method used in scheduling the learning rate in the gradient descent algorithm, namely the cosine annealing. When applied to ε, it should make the model converge to the best policy in the later stage without affecting the exploration in the early stage of model learning.

A. Cosine Annealing

Cosine annealing is a learning rate schedule method, where the specific expression of the learning rate is (3).

$$\varepsilon_t = \varepsilon_{min} + \frac{1}{2}(\varepsilon_{max} - \varepsilon_{min})(1 + \cos\frac{t}{T}\pi) \tag{3}$$

where $\eta_{max,min}$ is the initial η and final η we set; T is the number of total epochs. Here we use T = 10000, ε_{max}=0.01, ε_{min}=0:

$$\varepsilon_t = 0 + \frac{1}{2}(0.01 - 0)(1 + \cos\frac{t}{10000}\pi)$$
$$= 0.005(1 + \cos\frac{t}{10000}\pi) \tag{4}$$

The specific image is a cosine waveform of half-cycle, as shown in Fig. 9. In the cosine annealing, with the increase of epoch, the ε first decreases slowly so that the model will not fall into the local minimum quickly, then accelerates the decline, and then decreases slowly again. This decline mode can cooperate with the ε-greedy, producing satisfactory results.

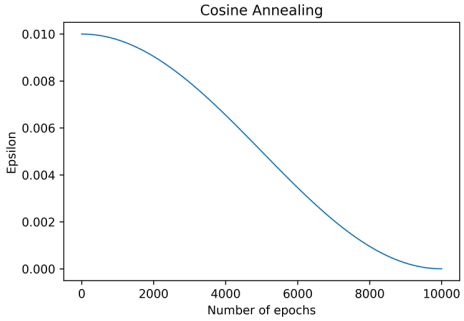

Fig. 9. Curve of cosine annealing.

B. *Experiments*

On the premise that other conditions remain unchanged, we apply cosine annealing on ε. The training results of the two algorithms are shown in Fig. 10.

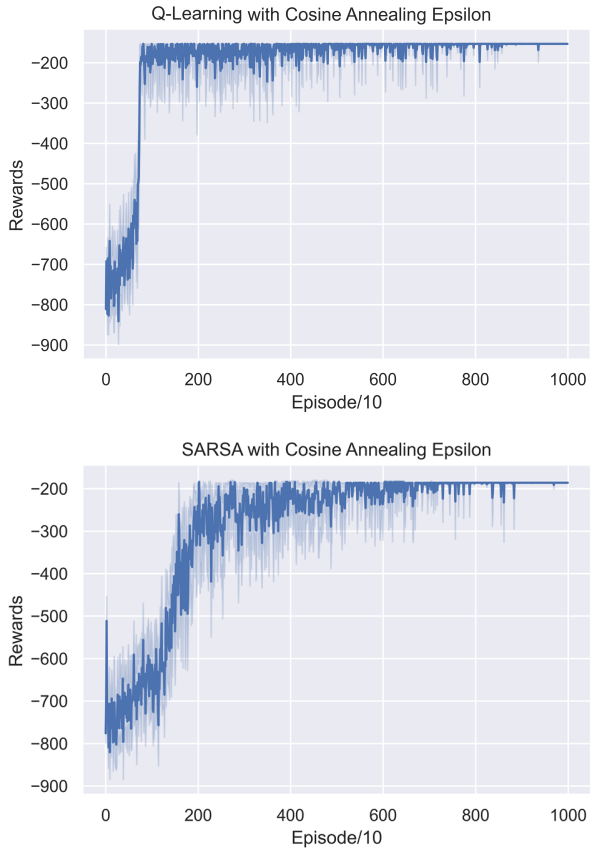

Fig. 10. Curve of training result with cosine annealing.

We can notice that the drastic oscillation disappears in the final stage of the two curves. In addition, the algorithm's strategy can still make the reward reach the original level, which shows that cosine annealing can both ensure the original learning effect of the algorithm and not be affected by continuous ε-greedy.

Moreover, the Q-value results of this experiment are also shown in Fig. 11.

Cosine annealing can also eliminate the continuous oscillation in the second half of the Q-value. Compared with Fig. 8, we can observe that the model with cosine annealing has robust stability, with no strategy conversion of Q-table due to unpredictable oscillation.

Even so, we still need to realize that it will increase the probability of falling into local optimization incorrectly, no matter which way to reduce ε we apply. Nevertheless, cosine announcing is still a scheduling method worthy of application.

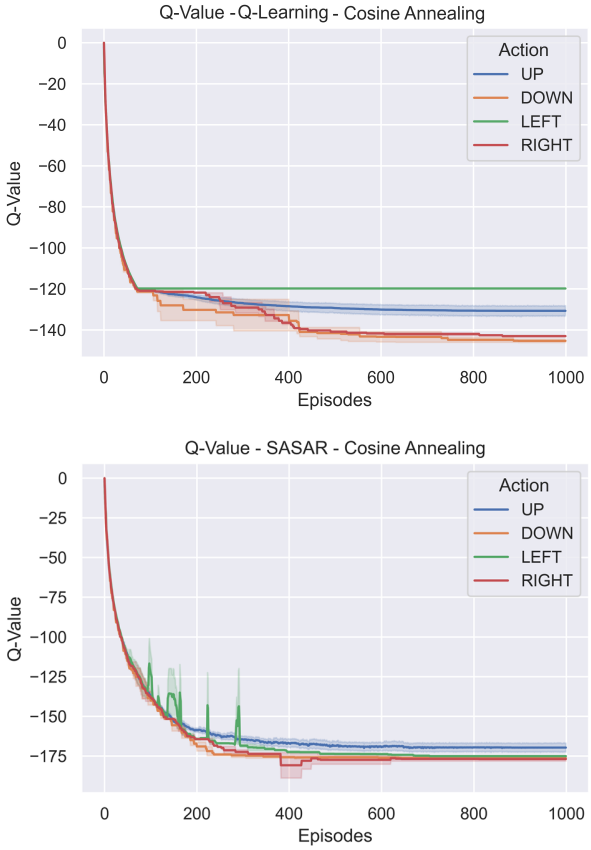

Fig. 11. Curve of Q-value in specific state with cosine annealing.

V. Code Availability

We improved the code on the original '*Q_learing.py*' file. Q-learning algorithm and SARSA algorithm are supplemented, and several functions for drawing curves and maps are added.

References

[1] Watkins, Christopher. (1989). Learning From Delayed Rewards.

[2] Richard S. Sutton and Andrew G. Barto (2015). Reinforcement Learning: An Introduction. Second edition. (pp. 158).

[3] Rummery, G. & Niranjan, Mahesan. (1994). On-Line Q-Learning Using Connectionist Systems. Technical Report CUED/F-INFENG/TR 166.

[4] Richard S. Sutton and Andrew G. Barto (2015). Reinforcement Learning: An Introduction. Second edition. (pp. 155).