



ME5406 Deep Learning for Robotics

Project 1: The Froze Lake Problem and Variations

Submitted by

Fu Yanqing A0225413R

Email: e0576047@u.nus.edu

Department of Mechanical Engineering

Faculty of Engineering

National University of Singapore




Table of Contents

Chapter 1 Introduction.....	1
Chapter 2 Task 1:Basic implementation.....	1
2.1 Monte Carlo Control.....	1
2.2 SARSA with an ϵ -greedy.....	2
2.3 Q-learning with an ϵ -greedy.....	3
Chapter 3 Task 2:Extended implementation.....	4
3.1 Monte Carlo Control.....	5
3.2 SARSA with an ϵ -greedy.....	6
3.3 Q-learning with an ϵ -greedy.....	7
Chapter 4 Compare and Contrast.....	7
4.1 Operational complexity and time.....	7
4.2 Policy.....	8
Chapter 5 DQN Algorithm.....	9
Chapter 6 Difficulties and Initiatives.....	10

Chapter 1 Introduction

The purpose of this project is to use reinforcement learning to solve the basic ice lake problem. In this report, there are four reinforcement learning methods used: First-visit Monte Carlo Control, SARSA with an ϵ -greedy, Q-learning with an ϵ -greedy, DQN algorithm.

Environment:

Grid with F, G, H, S, where S is a start, G is a goal , F is frozen lake  and H is a hole . There is a choice of 4x4, 8x8, 10x10, 12x12 Frozen Lake environment maps could be chosen.

Agent: Robot 

States: Up, Down, Left, Right

Reward: In state F, the agent gets 0 reward, in state H -1 and in G it gets +1 reward.

For running this program, it is essential to install gym, itertools and tensorflow (DQN method). I add my own environment file which made by myself so that it will be convenient for us to modify the environment and add DIY maps.

Chapter 2 Task 1: Basic implementation

2.1 Monte Carlo Control

First Visit Monte Carlo: Average returns only for first time s is visited in an episode.

To evaluate state s

The first time-step t that state s is visited in an episode,

increment counter $N(s) \leftarrow N(s) + 1$

increment total return $S(s) \leftarrow S(s) + G_t$

(where $G_t = R_{t+1} + \gamma R_{t+2} + \dots + \gamma^{T-1} R_T$)

Value is estimated by mean reward $V(s) = S(s) / N(s)$

By law of large numbers, $V(s) \rightarrow v_{\pi(s)} V(s) \rightarrow v_{\pi(s)}$ as $N(s) \rightarrow \infty$

The Result of First_Visit MC:

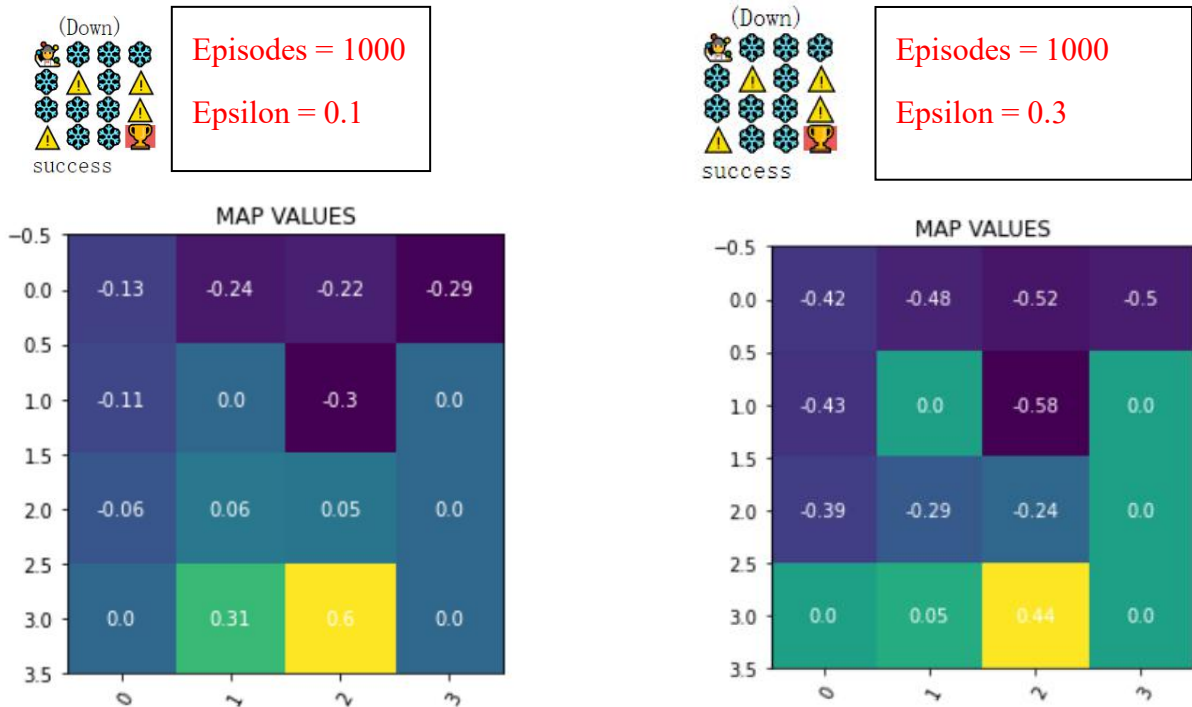


Figure 1: The Result of Monte_Carlo (map_size = 4*4)

In the basic Monte Carlo first-visit method, adding hyperparameter epsilon can eliminate exploring starts. In order to ensure that in the process of strategy selection, the agent will not fall into an infinite loop and reach the reward faster, I use the slippery update method, that is, when there is an epsilon exploration probability, another chance of random selection is added to get out of the infinite loop.

In the comparison of the above figure, we can find that in the case of different epsilon, the V value has a significant change, this is because the agent's exploration probability has changed.

2.2 SARSA with an ϵ -greedy

The algorithm flow of SARSA is:

```

Initialize  $Q(s, a)$  arbitrarily
Repeat (for each episode):
  Initialize  $s$ 
  Choose  $a$  from  $s$  using policy derived from  $Q$  (e.g.,  $\epsilon$ -greedy)
  Repeat (for each step of episode):
    Take action  $a$ , observe  $r, s'$ 
    Choose  $a'$  from  $s'$  using policy derived from  $Q$  (e.g.,  $\epsilon$ -greedy)
     $Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma Q(s', a') - Q(s, a)]$ 
     $s \leftarrow s'; a \leftarrow a'$ 
  until  $s$  is terminal
  
```

Source: https://blog.csdn.net/qq_39388410/article/details/88795124

So for SARSA

1. In the state s' , you know which a' to take, and actually take this action.
2. The selection of action a follows the ϵ -greedy strategy, and the calculation of the target Q value is also calculated based on the action a' obtained by the (ϵ -greedy) strategy, so it is on-policy learning.

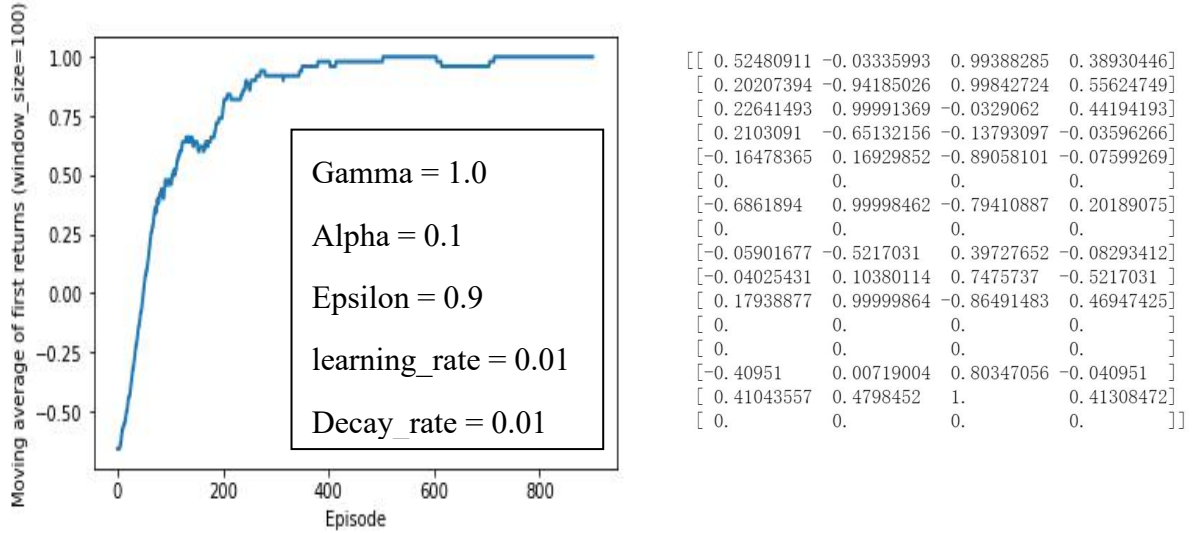


Figure 2: SARSA Result and Q_Table (map_size = 4*4)

The Q table can intuitively tell us the evaluation results of the agent's action selection in various directions after exploring the map. And we see that under the choice of this kind of hyperparameters, certain disturbances may occur in the process of rising rewards. I will optimize the choice of hyperparameters in Task2.

2.3 Q-learning with an ϵ -greedy

First, we will initialize a Q -table, which can be all 0s or other values, usually all 0s, and then we set the number of training episodes, where the initial state to the end state is counted as one round. Then in each round, we will have an initial state, and then we will continue to take actions, where each action is called a step. In each step, we select action A according to the current state through a certain strategy. Then after selecting action A , we can get the reward value R and the new state S , then the current q value is updated based on the following formula:

```

Initialize  $Q(s, a)$  arbitrarily
Repeat (for each episode):
  Initialize  $s$ 
  Repeat (for each step of episode):
    Choose  $a$  from  $s$  using policy derived from  $Q$  (e.g.,  $\epsilon$ -greedy)
    Take action  $a$ , observe  $r, s'$ 
     $Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$ 
     $s \leftarrow s'$ ;
  until  $s$  is terminal

```

Source: https://blog.csdn.net/qg_39388410/article/details/88795124

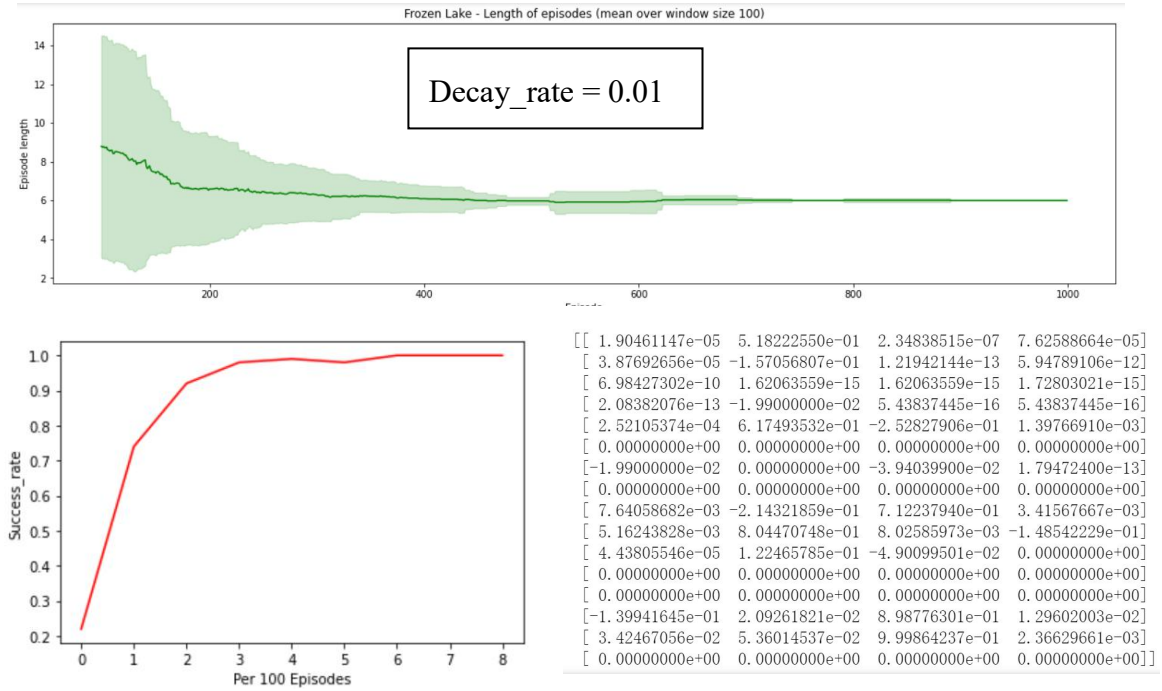


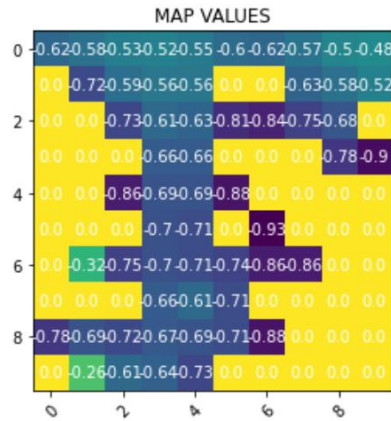
Figure 3: Q_learning Result and Q Table (map_size = 4*4)

We can see that the Q_Learning algorithm got an almost perfect strategy at 300 episodes, We compare with SARSA Q table and we can find that Q_learning Q table is different from SARSA Q table. And as the episodes increase, the number of steps the agent needs to walk decreases and gradually converges to a certain episode_length.

Chapter 3 Task 2:Extended implementation

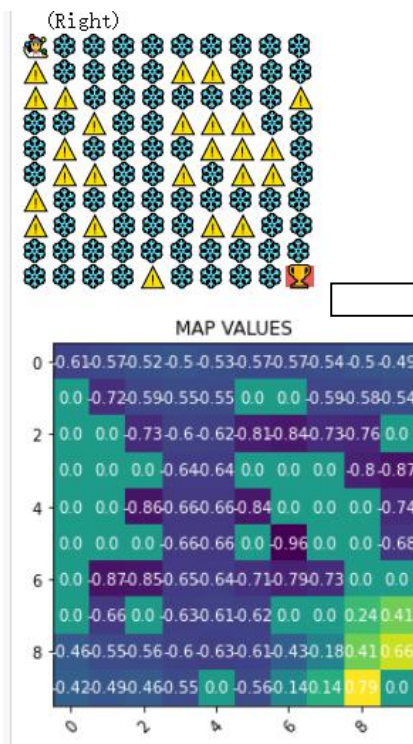
In order to reduce the amount of calculation, I used a 10×10 size map in this task. There is a map generator in the code I wrote that can directly generate maps of any size for use.

3.1 Monte Carlo Control



After increasing episodes to 10000, the agent still cannot find the correct solution, and the map has not been explored completely.

So continue to increase episodes to 45000 and increase epsilon to 0.5.



By increasing episodes and epsilon, you can increase the scope of your exploration of the map, from which you can see that all the places the map can reach have been updated, agent can find the right strategy, but requires a lot of steps. Without using SLIPPERY may cause the agent to fall into a dead loop and not be able to get out.

Figure 4: Monte_Carlo Result (map_size = 10*10)

For maps of different difficulty, Monte Carlo method requires different calculation steps. The convergence conditions of the Monte Carlo method are too harsh, and the strategy update method is too random to make Monte Carlo a good method.

3.2 SARSA with an ϵ -greedy

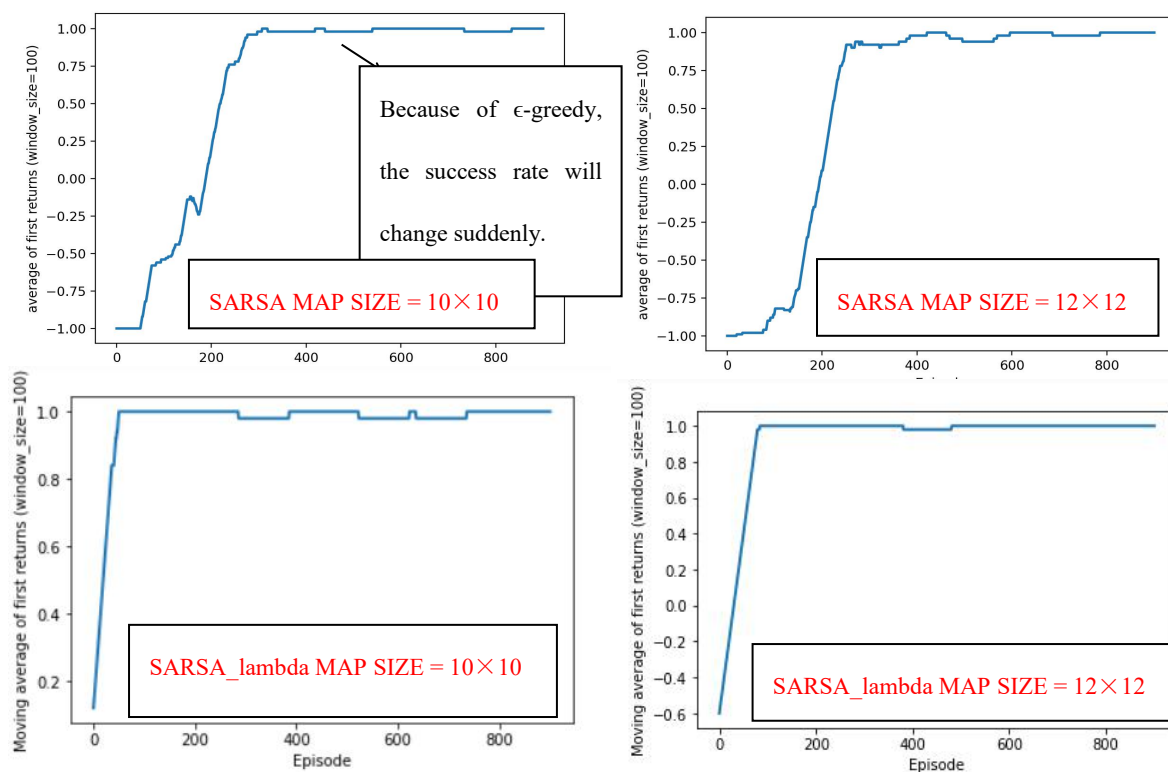


Figure 5: SARSA and SARSA_lambda results Contrast

SARSA is an on-policy reinforcement learning algorithm that estimates the value of the policy being followed. In this algorithm, the agent grasps the optimal policy and uses the same to act. The policy that is used for updating and the policy used for acting is the same, unlike in Q-learning. This is an example of on-policy learning. But if I use the faster global update, that is, the SARSA_lambda algorithm, it can make the agent find the best strategy faster. From these four pictures, we can find that the convergence speed of SARSA_lambda under the same map is much faster than SARSA.

3.3 Q-learning with an ϵ -greedy

As can be seen from the figure below, after the local map increases, the episodes required to use the Q learning algorithm to obtain a good strategy increase, and a good strategy can be obtained after reaching 200 episodes (when decay_rate = 0.1):

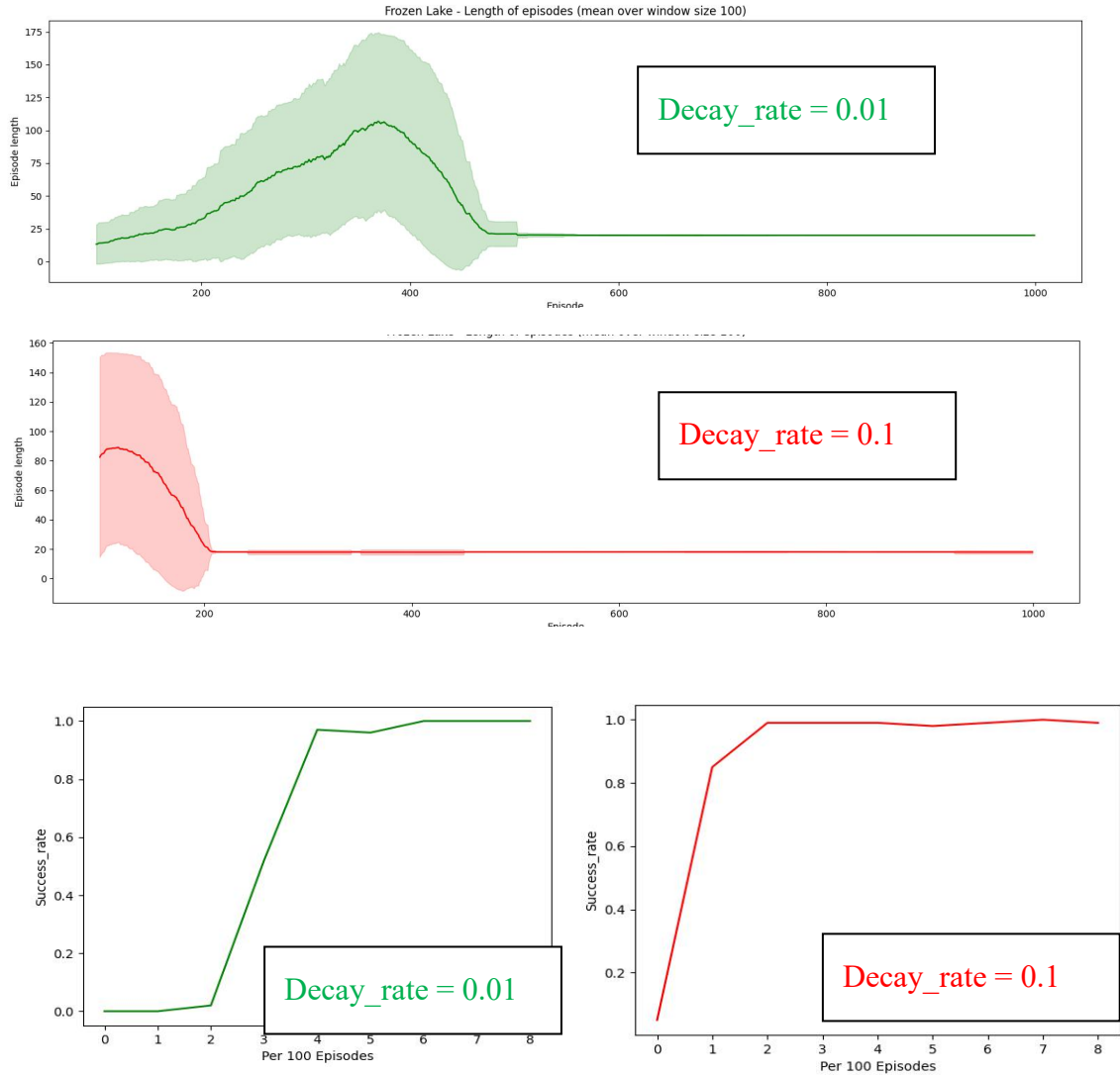


Figure 6: Different Decay_rate in Q_learning

In Q-Learning, the agent learns optimal policy with the help of a greedy policy and behaves using policies of other agents. Q-learning is called off-policy because the updated policy is different from the behavior policy, so Q-Learning is off-policy. In other words, it estimates the reward for future actions and appends a value to the new state without actually following any greedy policy.

Chapter 4 Compare and Contrast

4.1 Operational complexity and time

Compared with SARSA and Q_learning, the Monte_Carlo algorithm need more execution time and a lot of trial and error opportunities to find a successful strategy. The efficiency of its algorithm is much lower than Q-learning and SARSA.

We compare the pros and cons of SARSA, SARSA_LAMBDA and Q_learning algorithms under the same hyperparameters, and we can get the following results:

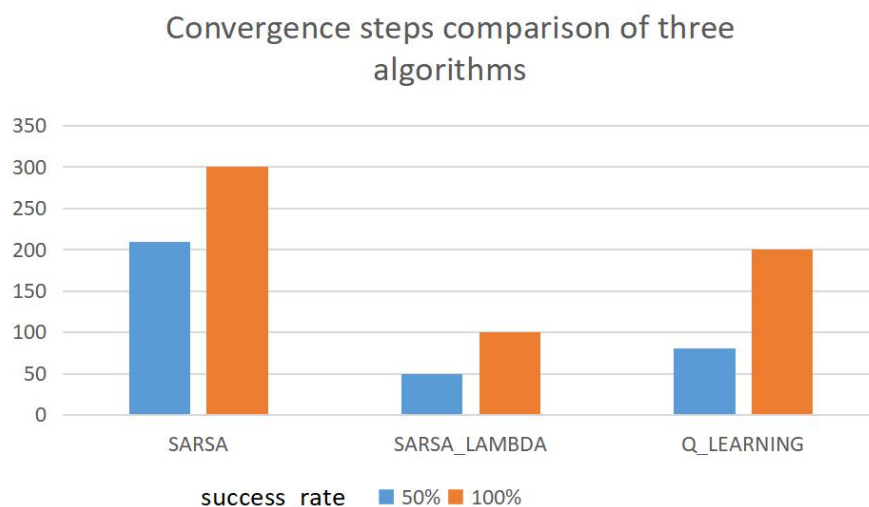


Figure 7: Convergence Comparison of Three Algorithms

The running time of these three algorithms is not much different, but the number of steps required to find the optimal solution is different. Using the same 10*10 map and using three algorithms to test under similar hyperparameters, we can get: The convergence speed of Q_learning is better than SARSA but weaker than SARSA_lambda.

4.2 Policy

Excluding the return route in the strategy, the strategy path obtained by the three algorithms is as follows:

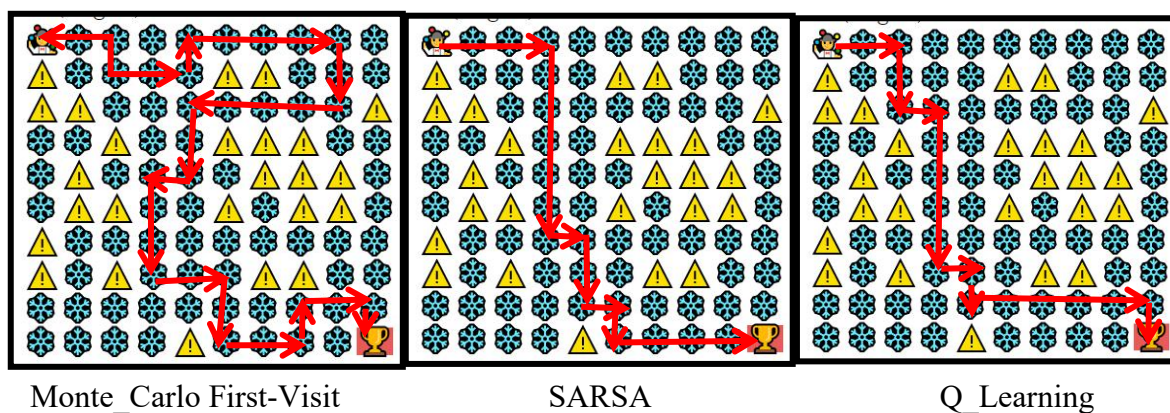


Figure 8: Comparison of Three Algorithm Strategies

SARSA chose a conservative strategy. He had planned his actions for the future when updating the Q value, and he was more sensitive to errors and death.

Q-learning chooses the direction that maximizes Q every time it is updated, and then re-selects the action in the next state. Q-learning is a reckless, bold, and greedy algorithm that does not care about death and mistakes.

In practice, if you care more about machine damage, use a conservative algorithm, which can reduce the number of machine damage during training.

Chapter 5 DQN Algorithm

Because I am interested in DQN algorithm, I tested the effect of the DQN algorithm, but I found that the DQN algorithm requires a lot of computer computing power, so I only used the DQN algorithm on a 4×4 map.

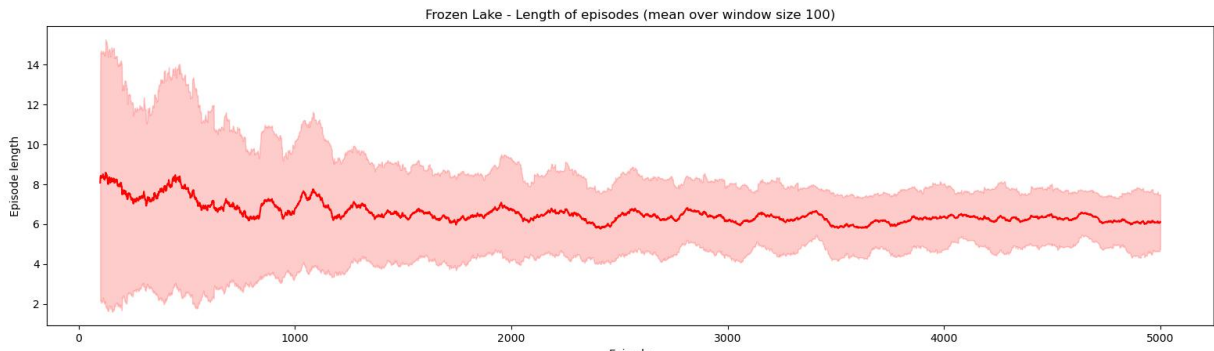


Figure 9: DQN Result

The training effect of using the DQN algorithm is not as good as expected. On the one hand, because the neural network requires more calculation steps, it will take more time to solve the problem by using DQN. On the other hand, it is prone to overfitting during the training process, which reduces its accuracy.

Chapter 6 Difficulties and Initiatives

I encountered many problems in this project, here I want to focus on these problems and describe how I solved them:

1. How to build and visualize the environment:

After reading the GYM library, I learned how to write an environment by myself by reading the environment source code, and wrote a map generator of any size. Through my own exploration, I found out how to modify the reward to -1, 0, 1 in the source environment. By consulting the relevant code on Github, I learned to map characters to string type pictures in RENDER. Then call in the main function to achieve the purpose of visualization.

2. Selection and comparison of hyperparameters:

Obviously, when selecting hyperparameters, different hyperparameters will affect the convergence of the algorithm. How to choose the appropriate hyperparameters requires multiple attempts and comparisons with the graph. And when comparing different algorithms, you should ensure that the hyperparameters are the same as possible, otherwise the impact of hyperparameters will directly change the results of the algorithm comparison.

3. Algorithm optimization and improvement

When using the SARSA algorithm, I found that it has many optimization directions, so I learned the SARSA_LAMBDA algorithm, and the convergence speed of the algorithm is greatly improved through the global update of the Q table. There are many ways to improve the algorithm, such as adding punish for traps or adding a little negative reward to blank cells can effectively improve the efficiency of the algorithm.