ME5406 Deep Learning for Robotics

Project I

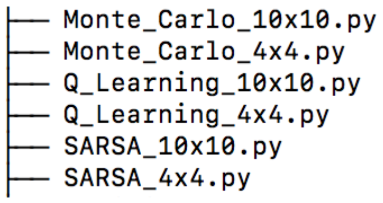
LIU QINGZHEN
A0225465A
E0576099@u.nus.edu

How to use?

As *Figure 1.1* shows, there are 6 files of 2 games with each algorithm to play.

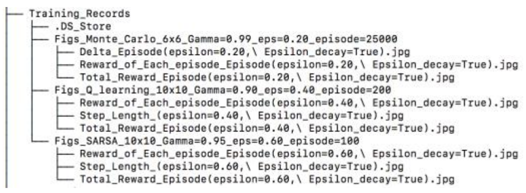
Training_Records in *Figure 1.2* is where training data were automatically saved after each successful training.

More details of the structure can be found in the file *ReadMe.md*.



```
├── Monte_Carlo_10x10.py
├── Monte_Carlo_4x4.py
├── Q_Learning_10x10.py
├── Q_Learning_4x4.py
├── SARSA_10x10.py
├── SARSA_4x4.py
```

Figure 1.1 File of Codes



```
├── Training_Records
│   ├── .DS_Store
│   ├── Figs_Monte_Carlo_6x6_Gamma=0.99_eps=0.20_episode=25000
│   │   ├── Delta_Episode(epsilon=0.20,\ Epsilon_decay=True).jpg
│   │   ├── Reward_of_Each_episode_Episode(epsilon=0.20,\ Epsilon_decay=True).jpg
│   │   └── Total_Reward_Episode(epsilon=0.20,\ Epsilon_decay=True).jpg
│   ├── Figs_Q_learning_10x10_Gamma=0.90_eps=0.40_episode=200
│   │   ├── Reward_of_Each_episode_Episode(epsilon=0.40,\ Epsilon_decay=True).jpg
│   │   ├── Step_Length_(epsilon=0.40,\ Epsilon_decay=True).jpg
│   │   └── Total_Reward_Episode(epsilon=0.40,\ Epsilon_decay=True).jpg
│   └── Figs_SARSA_10x10_Gamma=0.95_eps=0.60_episode=100
│       ├── Reward_of_Each_episode_Episode(epsilon=0.60,\ Epsilon_decay=True).jpg
│       ├── Step_Length_(epsilon=0.60,\ Epsilon_decay=True).jpg
│       └── Total_Reward_Episode(epsilon=0.60,\ Epsilon_decay=True).jpg
```

Figure 1.2 Structure of Training_Records

In *Pycharm*, it will work by first clicking the right button of mouse or touchpad and then click the button of *Figure 1.3*. Or in text file such as *Notepad* or *Sublime*, it will work with the hot key of your setting. Mine is ⌘+B.

During training, the calculated mean reward within certain episodes (which can be tuned by the parameter `mean_reward_calc_epi`) will be printed. However, it may be too fast to pass for an easy training. Then we should scroll up to check it if you want.

After running, you will see the *best policy, Q-table, map* of the game printed on the screen. Also, I use *Gym*'s *render()* to show the path the agent goes of one episode after training.

Finally, the successful training records will be saved in the local file as *FIGURE 1.4* Shows or *Jupyter Notebook* as *FIGURE 1.5* shows.

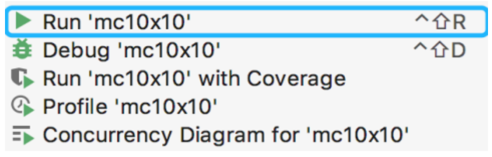
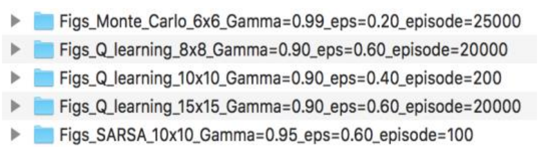
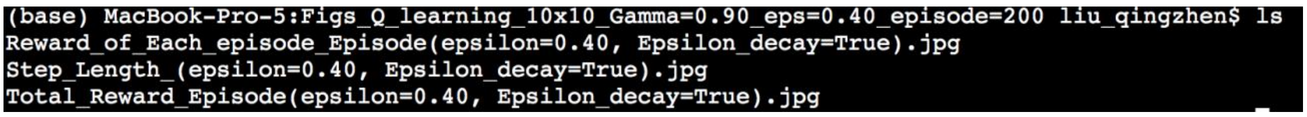

Run 'mc10x10' ^⇧R
Debug 'mc10x10' ^⇧D
Run 'mc10x10' with Coverage
Profile 'mc10x10'
Concurrency Diagram for 'mc10x10'

FIGURE 1.3 Run



```
▶ Figs_Monte_Carlo_6x6_Gamma=0.99_eps=0.20_episode=25000
▶ Figs_Q_learning_8x8_Gamma=0.90_eps=0.60_episode=20000
▶ Figs_Q_learning_10x10_Gamma=0.90_eps=0.40_episode=200
▶ Figs_Q_learning_15x15_Gamma=0.90_eps=0.60_episode=20000
▶ Figs_SARSA_10x10_Gamma=0.95_eps=0.60_episode=100
```

FIGURE 1.2 Saved Files



```
(base) MacBook-Pro-5:Figs_Q_learning_10x10_Gamma=0.90_eps=0.40_episode=200 liu_qingzhen$ ls
Reward_of_Each_episode_Episode(epsilon=0.40, Epsilon_decay=True).jpg
Step_Length_(epsilon=0.40, Epsilon_decay=True).jpg
Total_Reward_Episode(epsilon=0.40, Epsilon_decay=True).jpg
```

Figure 1.3 Files in Jupyter Notebook Terminal

Environment and methods

Gym

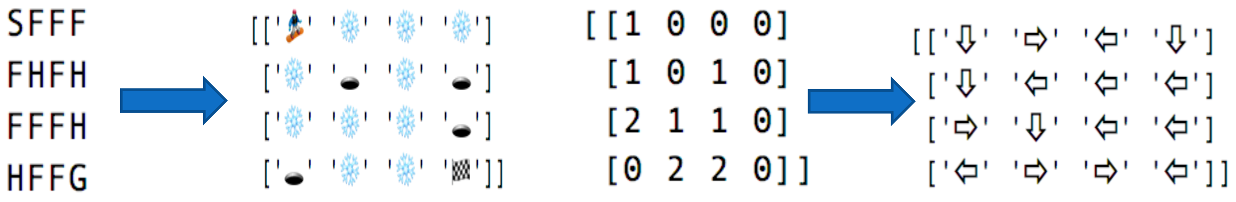
I applied *OpenAi Gym FrozenLake* environment in this project, and made some modifications to the environment, such as reward function and some other parameters. I will list the changes below.

Reward: In the *FrozenLake* environment, there is only one reward which is at the final point when reaching at the frisbee. We need to add rewards to the state of holes.

is_slippery: This parameter is set to be Ture by default, which will make choosing actions non-deterministic, which means even if we set *Epsilon* to zero, there is still possibility that the action will not be the one with the largest value in *Q-table* of the state. I changed it to *False*, so we will reach the frisbee point for each episode after *Epsilon* decays to 0, if the agent was trained correctly. Definitely, it's ok to be set to be *True* to run the program. But I am not sure if it follows the setting of the game rules of this project.

Map size: by tuning the parameter *map_size* in my program directly, we can generate a random map of that size easily.

Mapping: I replaced the number of actions and alphabet of map with icons using *dictionary*



```
SFFF        [['🏂' '❄' '❄' '❄']        [[1 0 0 0]        [['⇩' '⇨' '⇦' '⇩']
FHFH         ['❄' '●' '❄' '●']          [1 0 1 0]         ['⇩' '⇦' '⇦' '⇦']
FFFH         ['❄' '❄' '❄' '●']          [2 1 1 0]         ['⇨' '⇩' '⇦' '⇦']
HFFG         ['●' '❄' '❄' '🏁']]         [0 2 2 0]]        ['⇦' '⇨' '⇨' '⇦']]
```

Figure 1.6 Mapping of Elements in Map and Printed Policy

Common functions and parameters in Gym

There are some frequently used functions such as *reset()*, *step(action)* can be found in file *core.py* in *Gym*'s file folder.

reset(): To refresh the state of the environment and return a new state

step(Action): Take this action and return the state, reward and whether the agent is at finishing point after taking this action.

render(): To visualize the current state the agent is in.

observation_space.n: Number of states

action_space.n: Number of actions of a state

Epsilon Decay

Epsilon = Max (Minimum Epsilon, Epsilon – Current Episode/ Total Episodes)

When we choose *Epsilon-Decay* method, the parameter *Epsilon* will decay with the growing of training episode.

What's good of this *Epsilon-Decay* method is that it can start with a high exploring rate to know more about the environment and then choose the 'right' action with larger possibility after some episodes of training. *Epsilon-Decay* is a good solution to the '*exploration and exploitation* problem'. Another advantage is that when *Epsilon* goes to 0, model will choose the best policy based on *Q-table* without exploring, where actions will be deterministic. If the model has learned the best policy, then we will see continuous reward of 1 for each episode, and 0 if it hasn't, which would be more intuitive. In practical use, there could be a problem when *Epsilon* goes to 0. We may see the program looks like not moving. It will not go to the next episode. This is because the action that *Q-table* leads to will not let it goes to a hole or goal point, so the episode will end till the largest iteration, which could be very slow. Another explanation is that the agent has learned to avoid holes, so the training steps for each episode will be more than steps at the start of training. More details will be talked in the *Difficulties* part.

Evaluation Method

The idea of how to evaluate whether the model has learned the best policy is to test for some episodes and check if it gets the positive reward at each episode. After training, the *Q-table* will be fixed, and we choose action with the max value of its state without updating *Q-table*, then the action the model choose is deterministic for each state. If the model has learned the best policy, it should get the positive reward at every episode if *is_slippery* was set to be *False*

Comparison of 3 methods

Method / Map	*Monte-Carlo* without ES	*Q-Learning*	*SARSA*
4x4	Good	Best	Better
10x10	Most Episodes Needed	Fast and Best	Good
larger	None	20x20 works well	Ranked 2nd

Theoretically, all methods will reach the best policy if the number of training episodes is large enough. However, in practical, limited to the performance of my CPU, there are differences in efficiency for 3 algorithms. *Q-Learning* definitely works the best, *SARSA* got the second, while *Monte-Carlo* ranked last.

3 Methods can deal with 4x4 *FrozenLake* game easily within the computing limitation of my CPU. However, when dealing with 10x10 *FrozenLake* game, *Monte-Carlo* method performs really bad. There is an interesting thing happening very often when I train the agent on 10x10 game with *Monte-Carlo* method. If I use *Epsilon-Decay* method to set our *Epsilon*, then when the episode is near or reaching the place where *Epsilon* goes to 0, then what looks like is the agent will never finish this episode, just stuck at one episode. I believe this is because at that time, agent will choose the action of largest value for that state according to the *Q-table*, which turns out that the agent will not either reach the Frisbee or run into a hole. This is caused by no exploration, which happens when *Epsilon* is about to 0. So it is better to set a minimum value of *Epsilon* such as 0.01 to avoid this situation.

Q-Learning and *SARSA* all belong to TD-Methods. *Q-Learning* is an off-policy method. The policy *Q-Learning* uses to update its value is different from the policy in choosing action, which is called behavior policy. *Q-Learning* chooses action based on *Epsilon-Greedy*, and updates based on *max (Q(s,a))*. While, *SARSA* is an on-policy method. It uses *Epsilon-Greedy* to choose action and update values in *Q-table*. The result of the difference between these two methods is that *Q-Learning* tends to choose the best policy, while *SARSA* will not choose the action that may decreases the value, which is safer. This tending of choice may increase the step length of steps to reach the goal during training.

We can prove it by generating a map which looks like *FIGURE 2.1*, the holes are on the upside and downside. From *FIGURE 2.2* we can find that *Q-Learning* choose to walk along the holes, while *SARSA* chooses a safer policy. It goes to the 3rd rows first and then turn right, then up to the positive reward point.

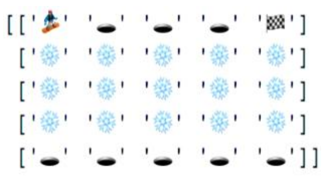
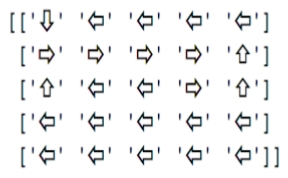
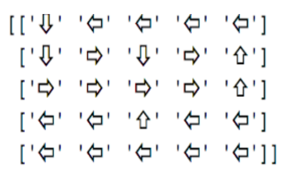


FIGURE 2.1 5x5 Map



```
[['⇩' '⇦' '⇦' '⇦' '⇦']
 ['⇨' '⇨' '⇨' '⇨' '⇧']
 ['⇧' '⇦' '⇦' '⇨' '⇧']
 ['⇦' '⇦' '⇦' '⇦' '⇦']
 ['⇦' '⇦' '⇦' '⇦' '⇦']]
```

FIGURE 2.2 Q-Learning's Policy



```
[['⇩' '⇦' '⇦' '⇦' '⇦']
 ['⇩' '⇨' '⇩' '⇨' '⇧']
 ['⇨' '⇨' '⇨' '⇨' '⇧']
 ['⇦' '⇦' '⇧' '⇦' '⇦']
 ['⇦' '⇦' '⇦' '⇦' '⇦']]
```

FIGURE 2.3 SARSA's Policy

Difficulties and my ideas

Non-deterministic behavior of agent after training

I started with *Q-Learning* and *SARSA* first as it is easier to program compared with *Monte-Carlo* method. However, I got stuck on *Q-Learning* for a long time. One problem appeared in training the agent in the *FrozenLake* 4x4 map. With the default setting of Gym, there is a parameter called *is_slippery* set to be *True*, which is created to imitate the slip of moving on lake. This will lead to non-deterministic of choosing actions even when *Epsilon* is 0. So even if the *Q-table* has been trained well, we still can't guarantee that the agent succeeds for every episode. To solve this problem, there are 3 methods. One is to set the parameter *is_slippery* to be False, then the choice of actions will be deterministic only depending on the *Q-table*. If the agent has been trained well, then it will succeed in each episode. This is the method I chose. Another is to set a success rate threshold in evaluating this model. For, example, if the agent succeeds 80 times in 100 tests, then we regard it as successful. Or, we can print the best policy based on *Q-table* and check if the model has been trained well to go to the final point, which may not be so intuitive.

Q-Learning can't solve 10x10 game using numpy.argmax()

The second problem appeared in training agent in 10x10 *FrozenLake* game with *Q-Learning*. It just didn't work at the beginning. I checked the *Q-table* and found that the *Q-table* was very sparse. The problem was caused by the *Argmax* function from *Numpy*. This function will return the index of the max number in an array. However, if there are many equal max numbers, then it will always return the index of the first max number. This means in the *Q-table*, if one state has 4 zeros, then it will always choose the 0th action, which is 'Left' in my game. This is definitely against the algorithm of *Q-Learning* in choosing action. So I rewrote a function to randomly return the index of the max number, and it works.

Agent got stuck in certain episode during training with Monte-Carlo method

As mentioned in the comparison of 3 methods, I was puzzled by the stuck of training in episode when *Epsilon* is near 0. It looks like there is something wrong with my program because my computer got hotter at that time. Well, just face the failure of this training and start another training will solve the problem. *Monte-Carlo* method has an extremely low success rate in playing 10x10 *FrozenLake* game without enough amount of training. For me, it requires more than 200 thousand episodes and even more to succeed.

My Ideas

By tuning the hyperparameters, we can improve the accuracy and speed of the model. The *Epsilon-Greedy* method really plays an important role in the training. High *Epsilon* will make the agent explore more of the environment, while also makes it hard to reach the goal with the increasing of the dimension of the map. Low value of *Epsilon* let the agent follow the known best path, but may cause not enough exploration. *Epsilon-Decay* is a solution, which is very useful.

Gamma should be large in order to make the *(state, action)* which is far from the reward point receive an evident update. Because the return will decay exponentially for steps before the reward.

By increasing the reward of reaching the goal, the agent may learn a feasible policy fast in this game. Too sparse of the setting of reward may make the training hard. It can be clearly found with the growing of size of map, where agent is hard to get the positive reward.

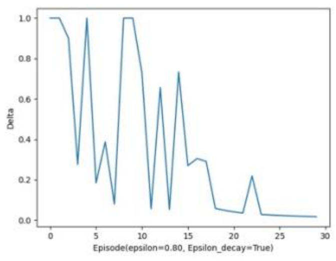
First-visit Monte-Carlo control without ES 4x4

Experiment Result

Episode = 30, Epsilon = 0.8, Epsilon_Decay = True, Gamma = 0.9

For 4x4 map, as the number of states is few, there is possibility of training agent to learn the best policy within such few episodes with *First-visit Monte-Carlo without ES*, but as the number of training episode is low, the success rate is not high.

We need to tune *Epsilon* to be high so that the model will explore more in choosing actions. It works well in 4x4, but with the dimension of the map goes up, the high *Epsilon* will lead the agent to holes more often. Let's say if we set *Epsilon* to be 1, then all actions will be taken randomly regardless of the *Q-table*. The possibility to 'guess' the right answer of 4x4 map could be $(1/4)^6$, while for a 10x10 map, the possibility would be $(1/4)^{18}$ on average, which equals 1/68719476736.



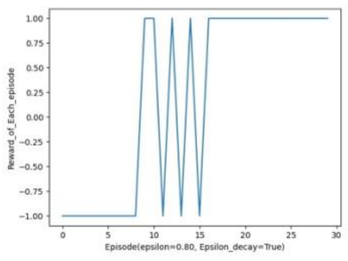
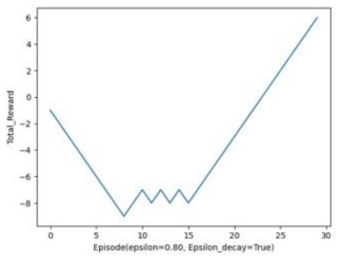
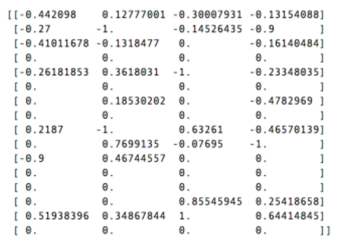


FIGURE 3.1 Delta of each episode   FIGURE 3.2 Reward of each episode with Epsilon-Decay

We can see from the *FIGURE 3.1* that delta goes to nearly 0 after training for some episodes.

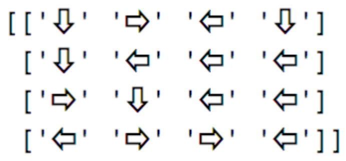




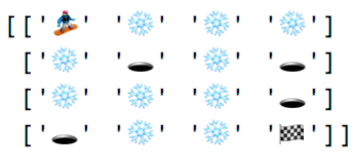
```
[[-0.442098    0.12777001 -0.30007931 -0.13154088]
 [-0.27       -1.         -0.14526435 -0.9       ]
 [-0.41011678 -0.1318477   0.         -0.16140484]
 [ 0.          0.          0.          0.        ]
 [-0.26181853  0.3618031  -1.         -0.23348035]
 [ 0.          0.          0.          0.        ]
 [ 0.          0.18530202  0.         -0.4782969 ]
 [ 0.          0.          0.          0.        ]
 [ 0.2187     -1.          0.63261    -0.46570139]
 [ 0.          0.7699135  -0.07695    -1.        ]
 [-0.9         0.46744557  0.          0.        ]
 [ 0.          0.          0.          0.        ]
 [ 0.          0.          0.          0.        ]
 [ 0.          0.          0.85545945  0.25418658]
 [ 0.51938396  0.34867844  1.          0.64414845]
 [ 0.          0.          0.          0.        ]]
```

FIGURE 3.3 Total Reward

FIGURE 3.4 Q-table



```
[['⇩' '⇨' '⇦' '⇩']
 ['⇩' '⇦' '⇦' '⇦']
 ['⇨' '⇩' '⇦' '⇦']
 ['⇦' '⇨' '⇨' '⇦']]
```

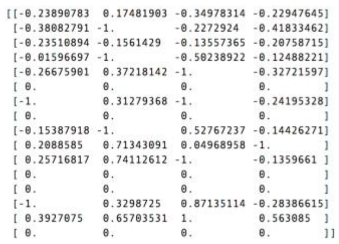


```
[['🏂' '❄️' '❄️' '❄️']
 ['❄️' '🕳' '❄️' '🕳']
 ['❄️' '❄️' '❄️' '🕳']
 ['🕳' '❄️' '❄️' '🏁']]
```

Figure 3.5 Best Policy

Figure 3.6 Map 4x4

Episode = 300, Epsilon = 0.7, Epsilon-Decay = True, Gamma = 0.9



```
[[-0.23890783  0.17481903 -0.34978314 -0.22947645]
 [-0.38082791 -1.         -0.2272924  -0.41833462]
 [-0.23510894 -0.1561429  -0.13557365 -0.20758715]
 [-0.01596697 -1.         -0.50238922 -0.12488221]
 [-0.26675901  0.37218142 -1.         -0.32721597]
 [ 0.          0.          0.          0.        ]
 [-1.          0.31279368 -1.         -0.24195328]
 [ 0.          0.          0.          0.        ]
 [-0.15387918 -1.          0.52767237 -0.14426271]
 [ 0.2088585   0.71343091  0.04968958 -1.        ]
 [ 0.25716817  0.74112612 -1.         -0.1359661 ]
 [ 0.          0.          0.          0.        ]
 [ 0.          0.          0.          0.        ]
 [-1.          0.3298725   0.87135114 -0.28386615]
 [ 0.3927075   0.65703531  1.          0.563085  ]
 [ 0.          0.          0.          0.        ]]
```

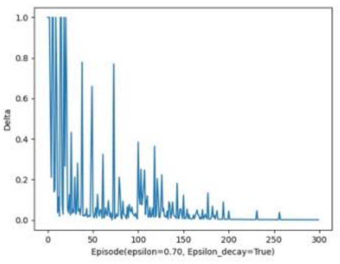
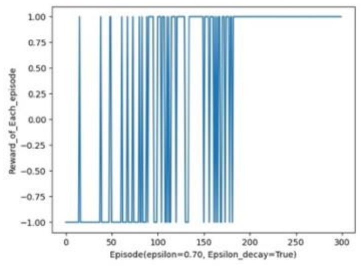


Figure 3.7 Q-table

FIGURE 3.8 Delta of each episode

As we can see, compared with training episodes of 30, this time the *Q-table* becomes less sparse as before. Those states with all zeros are all holes or ending point.



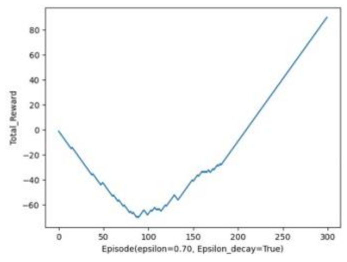


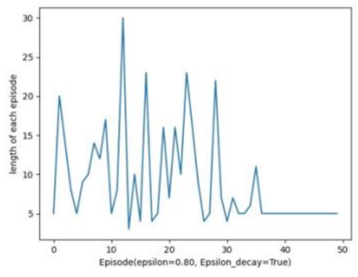
Figure 3.9 Reward of each episode with Epsilon-Decay

Figure 3.10 Total Reward

Q-Learning 4x4

Episode=50, Epsilon=0.8, Epsilon-Decay=True, Gamma=0.9, learning-rate=0.5

Q-Learning has a very high success-rate on 4x4 *FrozenLake* game within 50 episodes



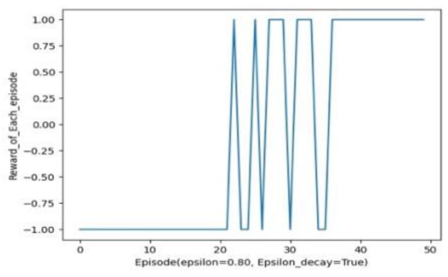
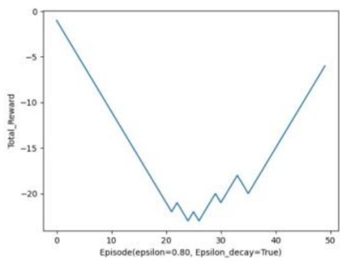
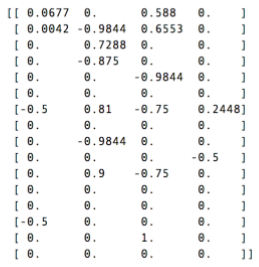


Figure 4.1 Step length of each episode

Figure 4.2 REWARD OF EACH EPISODE

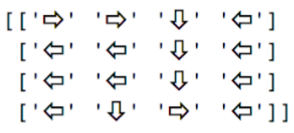




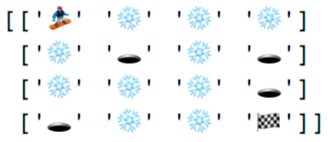
```
[[ 0.0677  0.      0.588   0.    ]
 [ 0.0042 -0.9844  0.6553  0.    ]
 [ 0.      0.7288  0.      0.    ]
 [ 0.     -0.875   0.      0.    ]
 [ 0.      0.     -0.9844  0.    ]
 [ 0.      0.      0.      0.    ]
 [-0.5     0.81   -0.75    0.2448]
 [ 0.      0.      0.      0.    ]
 [ 0.     -0.9844  0.      0.    ]
 [ 0.      0.      0.     -0.5   ]
 [ 0.      0.9    -0.75    0.    ]
 [ 0.      0.      0.      0.    ]
 [ 0.      0.      0.      0.    ]
 [-0.5     0.      0.      0.    ]
 [ 0.      0.      1.      0.    ]
 [ 0.      0.      0.      0.    ]]
```

Figure 4.3 Total reward

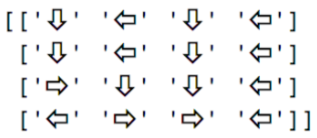
Figure 4.4 Q-table



```
[['⇨' '⇨' '⇩' '⇦']
 ['⇦' '⇦' '⇩' '⇦']
 ['⇦' '⇦' '⇩' '⇦']
 ['⇦' '⇩' '⇨' '⇦']]
```



```
[['⛷' '❄' '❄' '❄']
 ['❄' '🕳' '❄' '🕳']
 ['❄' '❄' '❄' '🕳']
 ['🕳' '❄' '❄' '🏁']]
```



```
[['⇩' '⇦' '⇩' '⇦']
 ['⇩' '⇦' '⇩' '⇦']
 ['⇨' '⇩' '⇩' '⇦']
 ['⇦' '⇨' '⇨' '⇦']]
```

FIGURE 4.5 Best Policy

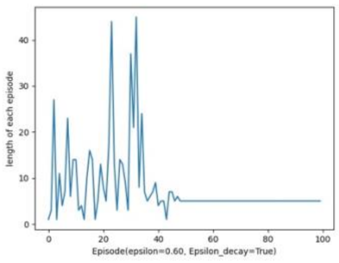
FIGURE 4.6 map 4x4

FIGURE 4.7 Best Policy of Another Training

After training, *Q-table* may generate different best policy for the agent. These 2 routes take same length of steps, are all the best policy. This may be caused by the different choice caused by *Epsilon-Greedy* method. When the agent explores one of the 2 best policies that will lead it to positive reward, then the *Q-table* will update the corresponding (state, action) value first. Then, the possibility of this (state, action) to be selected become larger.

SARSA 4x4

Episode=100, Epsilon=0.6, Epsilon-Decay=True, Gamma=0.95, learning-rate=0.5



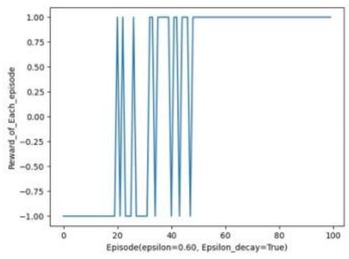
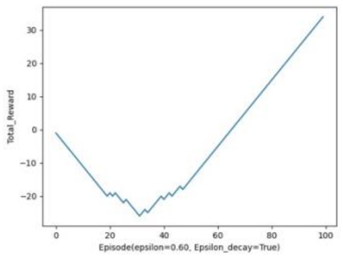
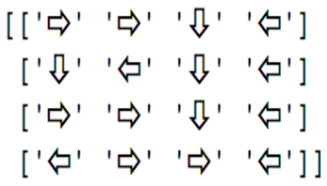
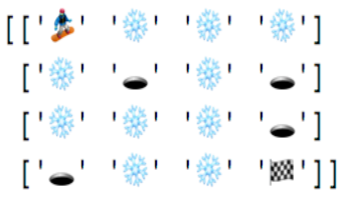


Figure 5.1 Step length of each episode   Figure 5.2 REWARD OF EACH EPISODE





```
[['⇨' '⇨' '⇩' '⇦']
 ['⇩' '⇦' '⇩' '⇦']
 ['⇨' '⇨' '⇩' '⇦']
 ['⇦' '⇨' '⇨' '⇦']]
```



```
[['⛷' '❄' '❄' '❄']
 ['❄' '🕳' '❄' '🕳']
 ['❄' '❄' '❄' '🕳']
 ['🕳' '❄' '❄' '🏁']]
```

Figure 5.3 Total reward   FIGURE 5.4 4x4 map best policy   Figure 5.5 4x4 map

First-visit Monte-Carlo control without ES 10x10

An easy map for testing

As I have low success rate in playing 10x10 *Frozenlake* game with Monte-Carlo, to make sure there is nothing wrong with my program, I tested with an easy map first: I trained it for 20000 episodes and it succeeded fast. It proved that there is no problem with my program. To make it work on a complex map, we just need to increase the training episodes.

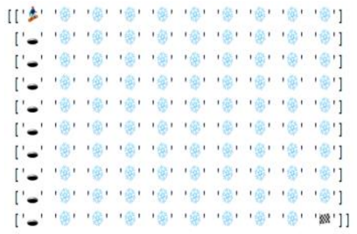
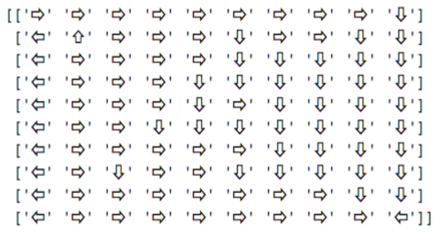


Figure 6.1 Tested 10x10 Map



```
[['⇨' '⇨' '⇨' '⇨' '⇨' '⇨' '⇨' '⇨' '⇨' '⇩']
 ['⇦' '⇧' '⇨' '⇨' '⇨' '⇩' '⇨' '⇨' '⇩' '⇩']
 ['⇦' '⇨' '⇨' '⇨' '⇨' '⇩' '⇩' '⇩' '⇩' '⇩']
 ['⇦' '⇨' '⇨' '⇨' '⇩' '⇩' '⇩' '⇩' '⇩' '⇩']
 ['⇦' '⇨' '⇨' '⇨' '⇩' '⇨' '⇩' '⇩' '⇩' '⇩']
 ['⇦' '⇨' '⇨' '⇩' '⇩' '⇩' '⇩' '⇩' '⇩' '⇩']
 ['⇦' '⇨' '⇨' '⇨' '⇨' '⇨' '⇩' '⇩' '⇩' '⇩']
 ['⇦' '⇨' '⇩' '⇨' '⇨' '⇩' '⇩' '⇩' '⇩' '⇩']
 ['⇦' '⇨' '⇨' '⇨' '⇨' '⇨' '⇨' '⇨' '⇩' '⇩']
 ['⇦' '⇨' '⇨' '⇨' '⇨' '⇨' '⇨' '⇨' '⇨' '⇦']]
```

Figure 6.2 Best Policy of Tested Map Delta

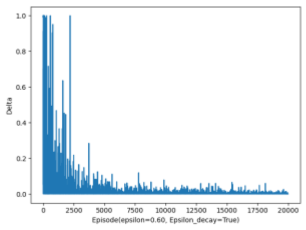


Figure 6.3 Delta

Episode = 3,000,000, Epsilon = 0.4, Epsilon-Decay = False, Gamma = 0.99

Monte-Carlo method has a hard time with playing *FrozenLake* 10x10 game, success rate is extremely low without a large number of training episodes. Detailed training process can be seen on my *Github Gist* here PRESS TO SEE THE TRAINING PROCESS ON GITHUB.

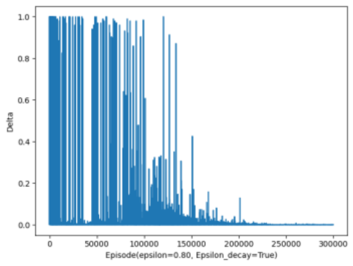
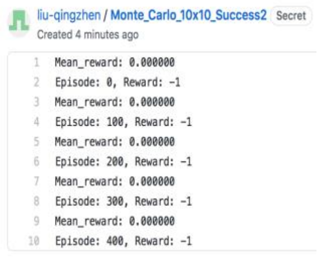


Figure 6.1 Delta of Map 10x10



liu-qingzhen / Monte_Carlo_10x10_Success2 Secret
Created 4 minutes ago

```
Mean_reward: 0.000000
Episode: 0, Reward: -1
Mean_reward: 0.000000
Episode: 100, Reward: -1
Mean_reward: 0.000000
Episode: 200, Reward: -1
Mean_reward: 0.000000
Episode: 300, Reward: -1
Mean_reward: 0.000000
Episode: 400, Reward: -1
```

Figure 6.2 Record on Github

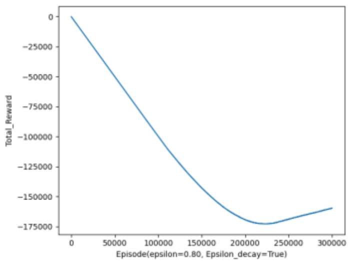
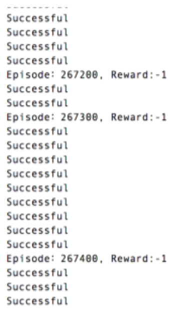


Figure 6.3 Total Rewards



```
Successful
Successful
Successful
Successful
Episode: 267200, Reward:-1
Successful
Successful
Episode: 267300, Reward:-1
Successful
Successful
Successful
Successful
Successful
Successful
Successful
Successful
Successful
Episode: 267400, Reward:-1
Successful
Successful
Successful
```

Figure 6.4 Experiment Record

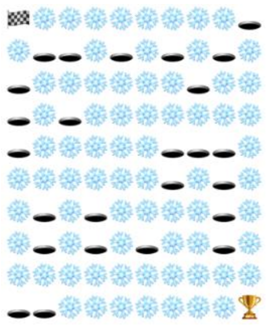


Figure 6.5 Map 10x10

```
[[2 2 2 1 0 1 0 0 0 0]
 [3 0 0 1 0 1 0 3 0 2]
 [0 2 2 1 1 0 0 0 2 1]
 [0 1 0 2 2 1 0 0 0 2]
 [0 2 2 3 2 1 0 0 0 3]
 [2 3 3 2 2 1 0 1 0 3]
 [1 0 3 0 2 2 1 1 0 1]
 [0 0 3 0 3 0 2 1 0 1]
 [0 0 0 1 1 2 2 2 2 1]
 [0 0 2 2 1 2 2 2 2 0]]
```

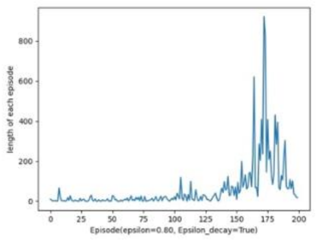
Figure 6.6 Best Policy

0 REPRESENTS LEFT; 1 REPRESENTS DOWN; 2 REPRESENTS RIGHT; 3 REPRESENTS UP

Q-Learning 10x10

Episode=200, Epsilon=0.4, Epsilon-Decay=True, Gamma=0.9, learning-rate=0.5

Q-Learning performs the best on 10x10 *FrozenLake* game among 3 methods. There could be failures with this parameter, but very few, because the map generated each time is not the same. Just add the training episode and tune those parameters.



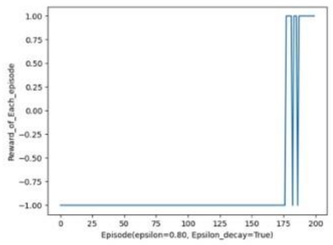
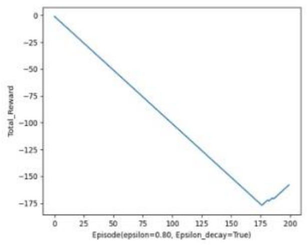
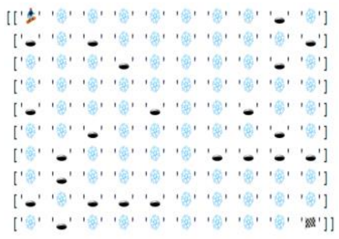
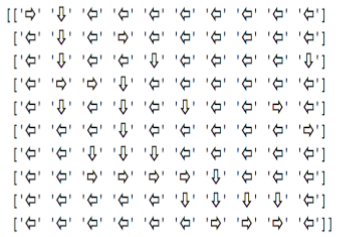


FIGURE 7.1 Step Length of Each Episode FIGURE 7.2 Reward of Each Length



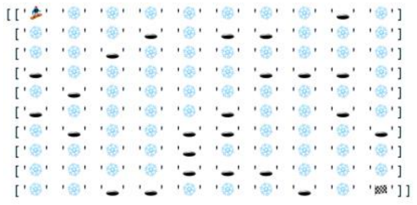


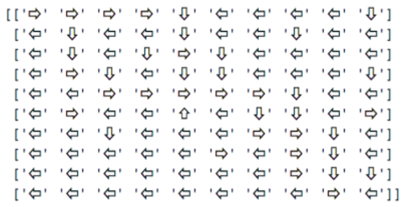


```
[['⇨' '⇩' '⇦' '⇦' '⇦' '⇦' '⇦' '⇦' '⇦' '⇦']
 ['⇦' '⇩' '⇦' '⇨' '⇦' '⇦' '⇦' '⇦' '⇦' '⇦']
 ['⇦' '⇩' '⇦' '⇦' '⇩' '⇦' '⇦' '⇦' '⇦' '⇩']
 ['⇦' '⇨' '⇨' '⇩' '⇦' '⇦' '⇦' '⇦' '⇦' '⇦']
 ['⇦' '⇩' '⇦' '⇩' '⇦' '⇩' '⇦' '⇦' '⇨' '⇦']
 ['⇦' '⇦' '⇦' '⇩' '⇦' '⇦' '⇦' '⇦' '⇦' '⇨']
 ['⇦' '⇦' '⇩' '⇩' '⇩' '⇦' '⇦' '⇦' '⇦' '⇦']
 ['⇦' '⇦' '⇨' '⇨' '⇨' '⇨' '⇩' '⇦' '⇦' '⇦']
 ['⇦' '⇦' '⇦' '⇦' '⇦' '⇩' '⇩' '⇩' '⇩' '⇦']
 ['⇦' '⇦' '⇦' '⇦' '⇦' '⇦' '⇨' '⇨' '⇨' '⇦']]
```

FIGURE 7.3 total reward FIGURE 7.4 Map 10x10 Figure 7.5 Best Policy

Another randomly-generated 10x10 map, parameters are same as above





```
[['⇨' '⇨' '⇨' '⇨' '⇩' '⇦' '⇦' '⇦' '⇦' '⇩']
 ['⇦' '⇩' '⇦' '⇦' '⇩' '⇦' '⇦' '⇩' '⇦' '⇦']
 ['⇦' '⇩' '⇦' '⇩' '⇨' '⇩' '⇦' '⇦' '⇦' '⇦']
 ['⇦' '⇨' '⇩' '⇦' '⇩' '⇩' '⇦' '⇦' '⇦' '⇩']
 ['⇦' '⇦' '⇨' '⇨' '⇨' '⇨' '⇨' '⇩' '⇦' '⇦']
 ['⇦' '⇨' '⇦' '⇦' '⇧' '⇦' '⇩' '⇩' '⇦' '⇨']
 ['⇦' '⇦' '⇩' '⇦' '⇦' '⇦' '⇨' '⇨' '⇩' '⇦']
 ['⇦' '⇦' '⇦' '⇦' '⇦' '⇨' '⇦' '⇨' '⇩' '⇦']
 ['⇦' '⇦' '⇦' '⇦' '⇦' '⇦' '⇦' '⇨' '⇩' '⇩']
 ['⇦' '⇦' '⇦' '⇦' '⇦' '⇦' '⇦' '⇦' '⇨' '⇦']]
```

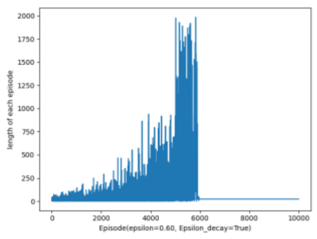
FIGURE 7.6 Map 10x10 Figure 7.7 Best Policy

SARSA 10x10 *FrozenLake*

Experiment Result

Episode=100, Epsilon=0.6, Epsilon-Decay=True, Gamma=0.95, learning-rate=0.5

Sometimes, we need to increase the number of episodes to make it successful. It doesn't succeed every time with episode of 100 for randomly-generated 10x10 maps.



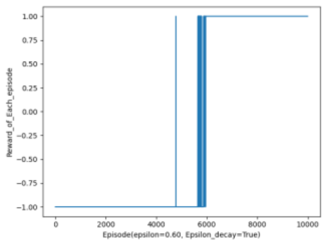


Figure 8.1 Step Length of Each Episode Figure 8.2 Reward of Each Episode

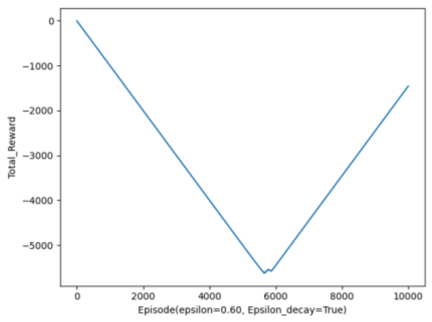
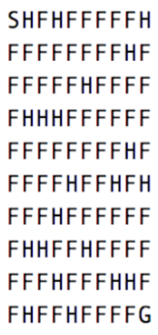
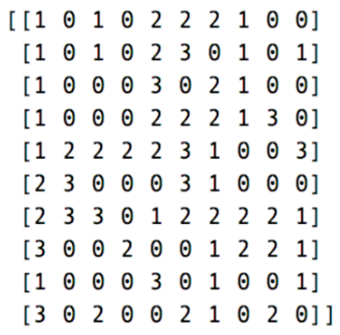


Figure 8.3Total Reward



```
SHFHFFFFFH
FFFFFFFFHF
FFFFFHFFFF
FHHHFFFFFF
FFFFFFFFHF
FFFFHFFHFH
FFFHFFFFFF
FHHFFHFFFF
FFFHFFFHHF
FHFFHFFFFG
```

FIGURE 8.4 10x10 Map

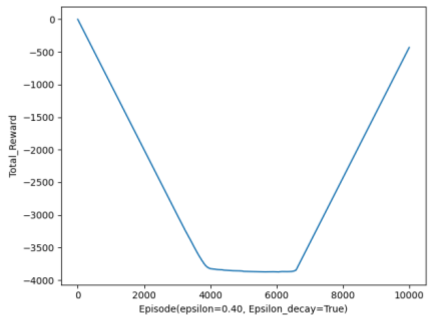


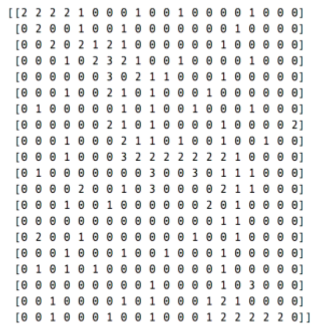
```
[[1 0 1 0 2 2 2 1 0 0]
 [1 0 1 0 2 3 0 1 0 1]
 [1 0 0 0 3 0 2 1 0 0]
 [1 0 0 0 2 2 2 1 3 0]
 [1 2 2 2 2 3 1 0 0 3]
 [2 3 0 0 0 3 1 0 0 0]
 [2 3 3 0 1 2 2 2 2 1]
 [3 0 0 2 0 0 1 2 2 1]
 [1 0 0 0 3 0 1 0 0 1]
 [3 0 2 0 0 2 1 0 2 0]]
```

FIGURE 8.5 Best Policy

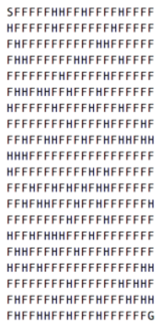
Map 20x20 Q-Learning

As *Pycharm* will have problem in showing the map replaced with those icons with dimension of 20x20, I used the original map and policy with numbers and alphabets.





```
[[2 2 2 2 1 0 0 0 1 0 0 1 0 0 0 0 1 0 0 0]
 [0 2 0 0 1 0 0 1 0 0 0 0 0 0 0 1 0 0 0 0]
 [0 0 2 0 2 1 2 1 0 0 0 0 0 0 1 0 0 0 0 0]
 [0 0 0 1 0 2 3 2 1 0 0 1 0 0 0 0 1 0 0 0]
 [0 0 0 0 0 0 3 0 2 1 1 0 0 0 1 0 0 0 0 0]
 [0 0 0 1 0 0 2 1 0 1 0 0 0 1 0 0 0 0 0 0]
 [0 1 0 0 0 0 0 1 0 1 0 0 1 0 0 0 1 0 0 0]
 [0 0 0 0 0 0 2 1 0 1 0 0 0 0 1 0 0 0 0 2]
 [0 0 0 1 0 0 0 2 1 1 0 1 0 0 1 0 0 1 0 0]
 [0 0 0 1 0 0 0 3 2 2 2 2 2 2 2 1 0 0 0 0]
 [0 1 0 0 0 0 0 0 0 3 0 0 3 0 1 1 1 0 0 0]
 [0 0 0 0 2 0 0 1 0 3 0 0 0 0 2 1 1 0 0 0]
 [0 0 0 1 0 0 1 0 0 0 0 0 0 2 0 1 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0]
 [0 2 0 0 1 0 0 0 0 0 0 0 1 0 0 1 0 0 0 0]
 [0 0 0 1 0 0 0 1 0 0 1 0 0 0 1 0 0 0 0 0]
 [0 1 0 1 0 1 0 0 0 0 0 0 0 0 1 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 1 0 0 0 0 1 0 3 0 0 0]
 [0 0 1 0 0 0 0 1 0 1 0 0 0 1 2 1 0 0 0 0]
 [0 0 1 0 0 0 1 0 0 1 0 0 0 1 2 2 2 2 2 0]]
```



```
SFFFFFHHFFHFFFFHFFFF
HFFFFFHFFFFFFFHFFFFF
FHFFFFFFFFFFHHFFFFFF
FHHFFFFFFHHFFFFHFFFF
FFFFFFFHFFFFFHFFFFFF
FHHFHHFFHFFFHFFFFFFF
HFFFFFHFFFFHFFFHFFFF
FFFFFFFFHFFFFHFFFFHF
FFHFFHHFFFHFFHFHHFHH
HHHFFFFFFFFFFFFFFFFF
HFFFFFFFFFFFHFHFFFFF
FFFHFFHFHFHFHHFFFFFF
FFHFHHFFFHFFHFFFFFFH
FFFFFFFFHFFFFHFFFFFF
HFFHFHHHFFFHFFFFFFFF
FHHFFFHFFHFFFHFFFFFF
HFHFHFFFFFFFFFFFFFHH
FFFFFFFFHFFFFFFFHFHHF
FHFFFFHFHFFFHFFFHFHH
FHFFHHFFHFFFHFFFFFFG
```

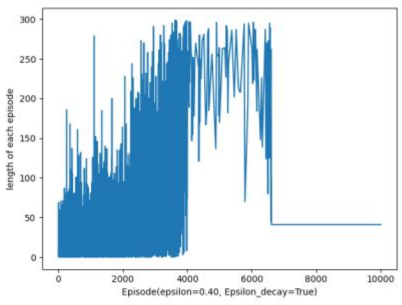
Figure 9.1 Total Reward

Figure 9.2 Best Policy

Figure 9.3 Map of 20x10

0 REPRESENTS LEFT; 1 REPRESENTS DOWN; 2 REPRESENTS RIGHT; 3 REPRESENTS UP

S REPRESENTS START, H REPRESENTS HOLES, G REPRESENTS FRISBEE



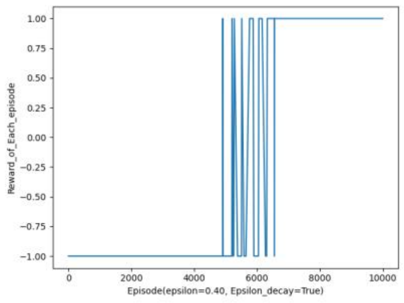


Figure 9.4 Length of Steps Per Episode

Figure 9.5 Reward of Each Episode Figure