National University of Singapore

ME5406 Deep Learning for Robotics

Project 1

The Froze Lake Problem and Variations

Author:
Wang Yizhuo

Matriculation Number: A0225440R
E-mail: wy98@u.nus.edu

October 12, 2020

1 Task statement and modeling

This project report aims to solve the *Frozen Lake* Problem and its variations via model-free reinforcement learning approaches. Frozen lake problem basically depicts a grid-world scenario that a robot wants to move from the start point to the goal where lies a frisbee, while avoiding falling into holes on the frozen lake.

Conventional ways to do the grid-world path finding like Dijkstra, A* can almost find an optimal path towards the target. Reinforcement learning is a new approach to solve path finding problems while having good properties of scalability and universality to various maps and multi-agents scenarios (we do not discuss it in this project).

A 4×4 and 10×10 world are built separately to test the performance of each methods, namely first-visit Monte Carlo, SARSA and Q-learning (the 4×4 map is shown as Figure 1). The world consists of one start point at upper left of the map, one frisbee point at lower right, and with 25% of holes randomly distributed on the map (meanwhile, guarantee at least one passage to the target). At each timestep, the robot can move one step (N/E/S/W) to an adjacent location.

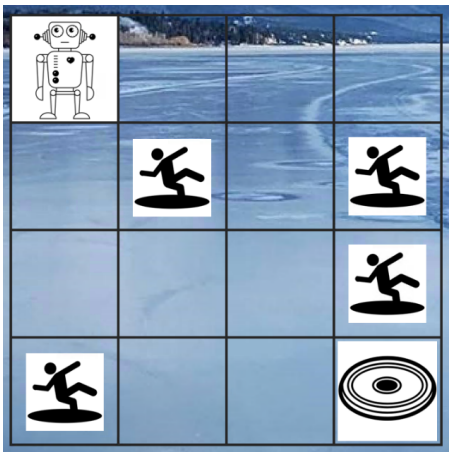


Figure 1: A robot moving on a 4×4 frozen lake.

This report will implement and compare these three algorithms via Python, give some theoretical explanations to the results, and furthermore show some improvement tricks to enhance the efficiency and robustness.

2 Implementations of reinforcement learning

2.1 Reinforcement learning cast

A problem like this should fit in RL structure for better analysis.

- *State space*: The size of state space $\mathcal{S}$ should be equal to $m \times n$, where m and n are the size of the map, 4×4 and 10×10 for here.
- *Action space*: For each timestep, the agent can either move Up/Down/Left/Right, so the size of action space $|\mathcal{A}|$ is 4, and for every state, $\mathcal{A}(\mathcal{S}_t) = \{Up, Down, Left, Right\}$.
- *Reward structure*: Agent receives a reward when it is in a hole or gets a frisbee.

  - Fall into a hole: -1
  - Reach the frisbee: +1

- *Details*:
  - The agent is confined in the grid, so even we do not restrict its action, any invalid actions will keep the agent stays where it is.
  - Once the agent is on the state of a hole or frisbee, the episode will be terminated.

2.2 Task 1: Implementation in a 4×4 map

2.2.1 First-visit Monte Carlo

The general idea of Monte Carlo prediction is to let the agent move around in episodes by following the given policy $\pi(s)$ while collecting the rewards along the way. Every episode will generate a sequence of $\{S, A, R\}$ series, and Monte Carlo prediction will calculate the return for each state-action pair. Compare to the every-visit method, first-visit only take the first visitation value in an episode into account when calculating the return. After that, Monte Carlo control will generate a new policy $\pi'(s)$ according to the new Q table. And it points out, after enough episodes, the extracted policy from the estimated Q table will tend to the optimal one $\pi^*(s)$.

So I implement the Monte Carlo algorithm in a 4×4 grid map for 10 times, and the results for each trail are as shown in Table 1. As we can see, in trail 5, an optimal policy has been generated just for 4 episodes! At the same time, 3 out of 10 trails do not finishes with an optimal policy or even without an successful exploration to the target. That is due to the fact that Monte Carlo is a method with high randomness, it updates the corresponding values only after an episode finishes, which is prone to low efficiency and high uncertainty.

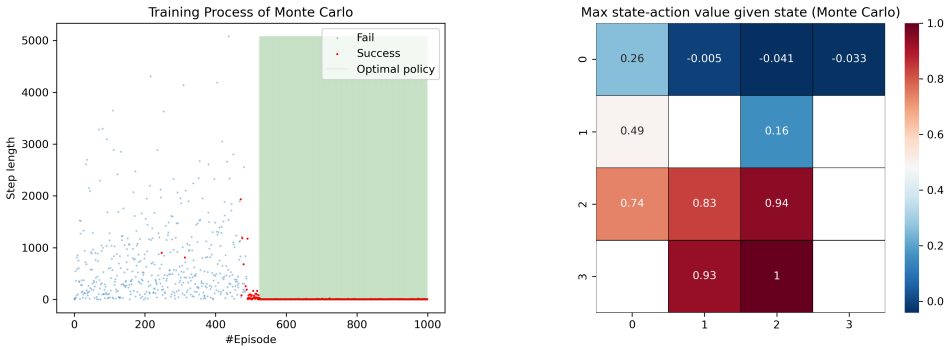
Table 1: I set the maximum episode number to be 1000, to make sure it can have enough training. **First reach** means the very first episode the agent gets a frisbee. **Optimal policy** means that by setting the policy to be the maximum state-action value: $\pi(s) = \arg\max_a Q(s, a)$, it can generate a policy leading the agent from start point all the way to the target. Apparently, episode of optimal policy is larger than the first reach.

Trail (max=1000 episodes)	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
First Reach (MC)	247	7	172	/	2	/	/	287	113	12
Optimal Policy (MC)	524	11	240	/	4	/	/	738	194	45

Training process of trail 1 is shown in Figure 2(a). The agent first reach the target at the 247th episode, which is represented as the red dot in the graph. The optimal policy tends to steady after 524 episodes, where are the green shadow lines in the graph. The heat map in Figure 2(b) gives a rough visualization of the Q table in each state.

2.2.2 SARSA and Q-learning

SARSA and Q-learning belongs to Temporal Difference learning, and they use bootstrapping to update the state-action value $Q(s, a)$ every one or few steps within an episode, which contributes to convergence. The most obvious discrepancy between SARSA and Q-learning is that SARSA



(a) Training process

(b) State value

Figure 2: First-visit Monte Carlo method. Left figure 2(a) shows the relation between the episode and the length of episode in training process. Blue dots are the episodes fall into the hole while red dots represent episodes end with reaching the frisbee. Green lines/shadowed area indicate an optimal policy has been reached, that is, where $\pi(s)$ will lead the agent to the target. Right figure 2(b) is the heat map of the state value, the number on which shows the maximum value of $Q(s, a)$ given state. Blank cells are either obstacles or the target.

is on-policy, while Q-learning is off-policy. That is to say, the behavior policy and target policy are the same for SARSA, different for Q-learning. The update rule of SARSA is

$$Q\left(S_t, A_t\right) \leftarrow Q\left(S_t, A_t\right) + \alpha[R_{t+1} + \gamma Q\left(S_{t+1}, A_{t+1}\right) - Q\left(S_t, A_t\right)] \tag{1}$$

The Q-learning just replaces the target policy term to a seemingly more greedy one.

$$Q\left(S_t, A_t\right) \leftarrow Q\left(S_t, A_t\right) + \alpha[R_{t+1} + \gamma \max_{a'} Q\left(S_{t+1}, a'\right) - Q\left(S_t, A_t\right)] \tag{2}$$

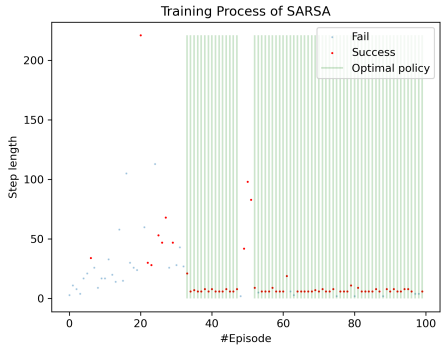
Implementation of each update strategy is shown in Table 2. We can see that SARSA reaches the target a bit faster than Q-learning, since it has a more explorative and safe updating rule. However, Q-learning tends to have faster convergence. Figure 3 illustrates the trail 1 training process and heat map of SARSA and Q-learning. They can both basically converge in 100 episodes.

Table 2: Randomly simulate in the 4×4 map for 10 times respectively for SARSA and Q-learning.

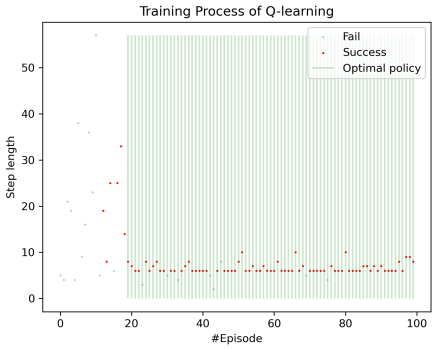
Trail (max=200 episodes)	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
First Reach (SARSA)	6	4	11	9	17	9	8	7	8	9
Optimal Policy (SARSA)	33	20	23	17	65	16	54	29	17	13
First Reach (Q-learning)	12	15	12	7	6	8	11	20	7	2
Optimal Policy (Q-learning)	19	26	24	20	14	14	15	30	15	14

Table 3: Randomly simulate in 10×10 map for 10 times each with SARSA and Q-learning. **First optimal policy** means the very first time an optimal policy has been generated, while **steady optimal policy** means the policy remains optimal for, say in our trails, $2000/100 = 20$ episodes.

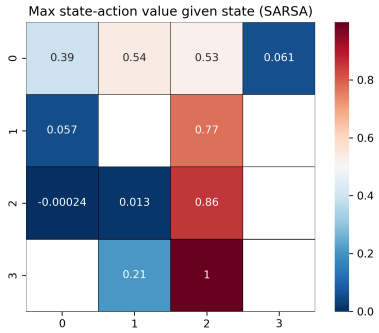
Trail (max=2000 episodes)	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
First Reach (SARSA)	274	266	379	236	227	188	144	207	173	284
First Optimal Policy (SARSA)	360	274	394	667	295	551	208	415	257	690
Steady Optimal Policy (SARSA)	883	436	611	952	/	1020	818	1048	759	1383
First Reach (Q)	632	207	691	750	22	620	735	937	660	1004
First Optimal Policy (Q)	961	773	928	1001	741	1034	859	1328	963	1407
Steady Optimal Policy (Q)	961	773	928	1001	741	1060	868	1337	963	1407



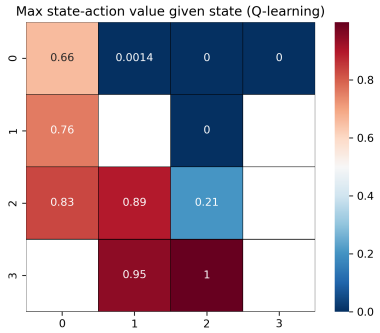
(a) Training process of SARSA



(b) Training process of Q-learning



(c) State value of SARSA

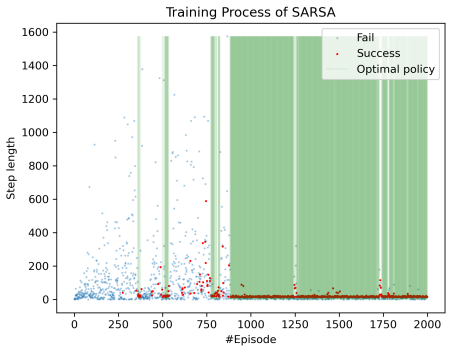


(d) State value of Q-learning

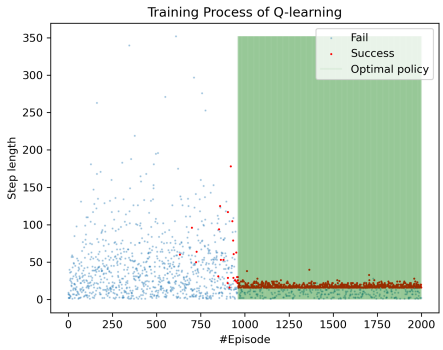
Figure 3: Training process and heat map of SARSA and Q-learning in a 4×4 map.

2.3 Task 2: Implementation in a 10×10 map

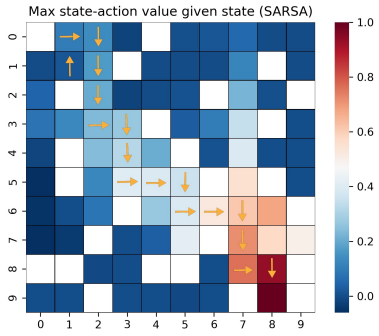
I further implemented the map to 10×10, while keeping 25% obstacle rate and set the start at $(1, 1)$ and frisbee at $(9, 9)$. From Figure 4 we can see that the SARSA reaches an optimal policy



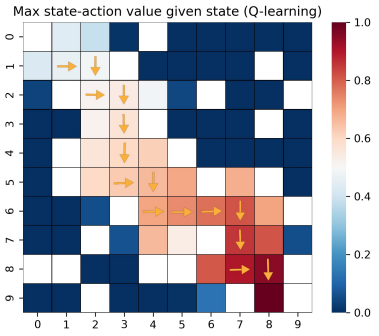
(a) Training process of SARSA



(b) Training process of Q-learning



(c) State value and derived policy of SARSA



(d) State value and derived policy of Q-learning

Figure 4: Training process and plicy heat map of SARSA and Q-learning in a 10×10 map.

earlier than Q-learning, but it is less stable, and that results in the discrete lines in Figure 4(a). Q-learning seems to converge slower, but once an optimal policy has been generated, it remains stable, and it is reflected in the solid pure green area in Figure 4(b). Table 3 also gives the same idea. The optimal policy firstly reach, for more than half of the cases, is the same as the steady episode.

First-visit Monte Carlo has been run for a couple of times, neither of which has reached the target within 50000 episodes. It indicates the low computational efficiency of pure Monte Carlo method.

3 Comparisons and discussions between each algorithm

First-visit Monte Carlo method is conducted off-line - it updates value functions only after an episode finishes. Once it finished, it will trace back all the state-action pairs and calculate the return for each one. Theoretically, it will indeed converge to an optimal policy, but in practice, especially in large-scale environment, it can easily become intractable due to intensive computation.

Online algorithms like SARSA and Q-learning utilize TD(0) to update the value each step instead of each episode, which can lead to better performance. Some visualizations are made to

illustrate the exploration and training process as in Figure 5.

The learning curve in Figure 5(a) and 5(b) show the relationship between the number of episode and the number of total step. The intuition is to observe the smoothness of the curve. If the curve grows squiggly, it indicates the algorithm is still exploring and yet to train. Once fully trained, the curve will grow in a relatively flat and smooth way. Hence, we can see that in 4×4 map, SARSA and Q-learning can quickly generate an optimal policy, despite there is a small disturbance in around the 600th episode of SARSA, which is the same as what we have been discussed: the on-policy update rule can make SARSA less stable than Q-learning. The *Cliff Walking* example also points out that the SARSA will take risk into consideration. Therefore, it might occur to us that, SARSA might lose its optimal policy during some updates, and converge again afterwards. That is why there occurs a sudden drop of SARSA's success rate in Figure 5(d).

In comparison, Monte Carlo method converges way slower than TD methods. Success rate in Figure 5(c) and 5(d) also proves this. Success rate even remains zero in 10×10 map (Note here success means an episode ends with target, so the ϵ-greedy exploration strategy needs to be taken into account).

The derived policy from SARSA and Q-learning is shown in Figure 4(c) and 4(d). Due to the obstacle setting and map constraints, the policies are basically the same. But the policy derived by SARSA is not the best policy, and we can still see a trend that SARSA is exploring more and doing safe, while for Q-learning, there are still many states have not been visited anymore, represented as navy deep blue in the graph.

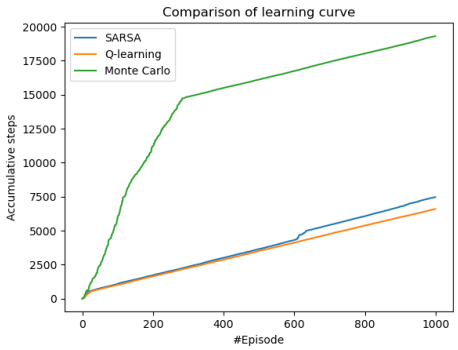
4 Improvements

From last section, it appears to us that every algorithms have its disadvantages, some of them are severely fatal. So in this section, I will present some methods I've been tried to enhance performance. Unfortunately, neither of these methods can make first-visit Monte Carlo find the path to the target in 10×10 map within 10000 episodes. So I would incline to say, the core idea of conventional Monte Carlo is not suitable for this kind of scenarios.

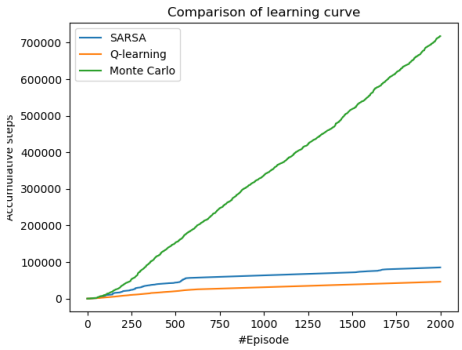
- **Add maximum step per episode.** We can discover an interesting phenomenon of the step length distribution in, for instance, Figure 4(a). Some step length is much higher than we expected, even over 1000 steps in a 10×10 map! The factor causes this is, as I would call it, the 'recurrent policy'. This happens when some blocks of the map form a closed-loop pathway. For instance, the policy of two adjacent states point to each other, or the policy of four or more states form a closed-loop. That would tremendously increase the step length, and extend the training time. So we can simply add a maximum step to constrain the length. However, this value should be chosen carefully, since it could be very hard for an agent to find the target within a limitation of small step length.

  If we limit the maximum step length, it usually needs more episodes to reach an optimal policy, but the good aspect is that it reduces ineffective training and in most cases, the training is faster and more predictable. This strategy is in essence to trade space for time. Empirically, we can set the maximum step length to at least two times of the number of state.

- **Improve reward structure.** Previously, the problem only have two kinds of rewards, namely, reach the frisbee for +1, and fall into the hole for -1, which is somewhat too simple. Here are some ways we can do to help.

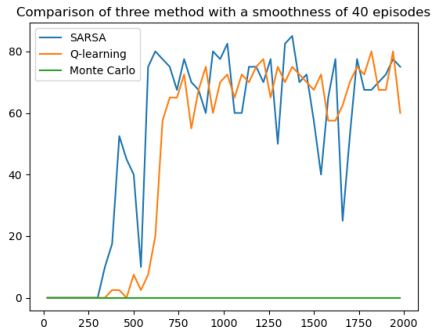


(a) Learning curve in 4×4 map



(b) Learning curve in 10×10 map

(c) Success rate in 4×4 map



(d) Success rate in 10×10 map

Figure 5: Comparison between First-visit Monte Carlo, SARSA and Q-learning. 5(a) and 5(b) show the relation between the number of episode and the total accumulative step. Success rate is to see the rate of exploration ends with reaching the frisbee. Note it is an average value over a certain number of episodes.

- Distance penalty. From Figure 4(c) we can see that SARSA does not gives the best policy, instead, it makes a detour. Why? Because we do not have a term to restrain the distance. That results to a longer path basically makes no difference from the short one. Just as we discussed in the last item, it could easily fall into a 'recurrent policy'. Agent would prefer to linger around rather than fall into the hole, before it finds the target. Therefore, adding a constant small penalty at each step will make it incline to a shorter path to a great extent. The result is in Table 4. With a constant step penalty of -0.1 in SARSA, the average and maximum step length drop down dramatically. Since the average step length is positively correlated with computation time, so it also saves a lot of time[1]. Meanwhile, the results remains basically the same, sometimes even better.
- Higher target reward. The reward of getting a frisbee is low compared to fall into the hole, since it is much easier to fall into the hole during exploration. Sometimes the agent arrives at the target for the first time, but its impact on previous state-actions

[1]For your information, the result is run under Intel Core i7-9750H.

is too low to change the policy, making it slow to train. Hence, it would be better to set a higher target reward, say +10.

  - Marginal penalty. This is an optional choice, since it just helps a little for the training. We can add a small penalty when the agent is taking actions that are about to out of the bound. This term can somehow reduce invalid trails to hit on the wall.

- **Decaying learning rate.** In the textbook, it introduces a decaying learning rate $\alpha_t = 1/(t+1)$, where t is the episode number. But in practice, I find it is not a good decaying strategy at least in 10×10 path finding - the learning rate decays too fast. Hence, we can set a minimal residual learning rate to make it 'at least' learn something from each episode. For instance, set $\alpha_t = 0.05 + 0.95/(t+1)$. In this way, the learning rate will drop from 1 to tend to 0.05 step by step. In practice, I find that it will contribute to the stability of the algorithm.

- **Decaying ϵ-greedy.** The ϵ-greedy policy is to trade-off between exploitation and exploration. Intuitively, we want the agent behaves 'bravely' before finding the target, and be conservative after generating an optimal policy. So likewise, we can set $\epsilon_t = 0.05 + 0.95/(0.1t+1)$.

Table 4: Effect of adding a small penalty per step for SARSA. Trainings with and without penalty are each executed for 10 times and gives an average. Each training has a episode number of 2000.

Items	**Without Penalty**	**With Penalty of -0.1**
Average step length	69.2	17.4
Maximum step length	2080.7	135.6
Time spent	4.32s	1.52s

5 Program details and requirements

5.1 Environments and requirements

The program is run under the environment of Python 3.6. The environment is all written by myself, data structures and some of the functions are inspired by `https://github.com/gucino`. For scientific calculation and visualization, there are few packages need to installed as below, you can use either `pip` or `conda` to manage.

```
conda create -n <env_name> python=3.6   # create virtual environment
conda activate <env_name>               # activate the environment
conda install numpy=1.16.0              # for scientific calculation
conda install matplotlib                # for plotting graphs
conda install seaborn                   # for heat map visualization
python run.py                           # run the file in terminal
```

Files are zipped along with this report, or you can clone it from my GitHub account at: `https://github.com/wyzh98/FrozenLake_NUS`. It includes the following files below.

- `run.py`: Main file, you should run this. This file contains all the classes of algorithms. Python class `FirstMonteCarlo`, `SARSA` and `QLearning` are all inherited from base class `FindPathBase`.
- `frozenlake_env.py`: Environment file, where `step()`, `reset()` and `render()` functions are defined. Reward structure can be tuned here, along with the I/O of maps.
- `map_4x4.txt, map_10x10.txt`: Map file. 0 for ice, 1 for hole, 2 for start and 3 for target.
- `requirements.txt`: Packages and their version of my Python environment, some of them are not a necessity for this project.
- `README.md`: Description file, which you can ignore.

5.2 Execution of the program

To execute the program, you should run `run.py`. The 10×10 map is set by default, and it will automatically run SARSA, Q-learning and first-visit Monte Carlo in sequence. As a result, it will print the first-reach episode result on the screen (if any), and followed by the policy of the last episode. Meanwhile, some figures are plotted, which are basically the same as the figures shown in the report. One training process and one heat map for each method, and two figures of comparison. If you do not want to run all the three methods, comment the line with `<env_name>.run()` correspondingly and the render functions in `__main__()`.

5.3 APIs and parameters settings

If you want to change some parameters, the APIs are all set in my programs. All you need to do is to change these numbers, and here are some of them.

- To change the map, you should look at `run.py -> __main__() -> map_idx`, it locates at the last few lines of `run.py`. You can set the variable 0 for 4×4 map, and 1 for 10×10 map.
- To change the number of episodes of training, you should change the number at `run.py -> __main__() -> n`.
- To change the discount rate γ, ϵ-greedy and maxinum step length, you should look at `run.py -> FindPathBase:class -> __init__() -> self.GAMMA, self.EPSILON, self.MAX_STEP`. They have already been set by default, respectively, 0.95, 0.1 and infinity. All the simulations in this report are based on these numbers.
- To change the reward structure, you should look at `frozenlake_env.py -> FrozenLake: class -> step() -> reward`. The reward structure is constructed by some if-else statement, where you can simply change them in corresponding conditions.
- To try decaying ϵ or learning rate, you should look at `run.py -> SARSA/QLearning:class -> run()`, and uncomment the line with `self.EPSILON` or `self.LR`.

The programs are well-commented, with the I/O explanations of almost every function, you can also refer to them.

6 Difficulties and conclusion

Although I had some prior knowledge about reinforcement learning, since my ME5001 project is about searching via recurrent attention model based RL, I still struggled and learned a lot from this project. Because it really teaches me how to write from the very first line of the code to the tower of a hole project. I have been considered and thought for a long time about what are the input/output of each file, class or the function, what are each of them in charge of respectively, and where are the boundaries.

One thing that has been puzzled me for a while is that what kind of data structure should I use to store the sequence of state, action and reward. Thanks to Tisana Wanwarn, from whom I had some inspiration to store it via a dictionary composed of state tuple as keys and action list as values. This is really brilliant and fast to compute. Another one is the visualization.In order to present the full knowledge of data while making it nice to look, I've checked some of the visualizations from online resources but found most of them either imprecise or not elegant to look. So I check the usage of `matplotlib` and `seaborn`, then made that scatter graph with crossing shadowed lines, and the heat map. Hope that would make people clear about what is happening during the training. Dealing with 'first-visit' of Monte Carlo also got me for a while. I have to detect whether it is the first state-action pair in this episode. If it is, then I can store it into the state-action list for the whole training and calculate the average. Therefore, I made a temporary list to judge whether it is the first time to come. It seems easy now, but really took me a while when handling this.

As a conclusion, first-visit Monte Carlo is less efficient than TD algorithms like SARSA and Q-learning. Although theoretically Monte Carlo can converge to an optimal policy, but in practice, it never gives me a positive result when dealing with 10×10 maps. Maybe it can converge after millions of episodes, but I just did not try that much, and I don't think it is necessary, for during the 10000 episodes training can we see the randomness it has. The SARSA and Q-learning are both efficient, mostly within hundreds of episodes can they extract an optimal policy from Q table. The difference is that, SARSA explores the map more than Q-learning does. Even when SARSA has formed a stable policy, it might lost it during some episodes. That is because the consideration of potential risk can give a value that leads to a huge impact on the adjacent state-actions when it happens to update them, and it causes the lost of optimal policy during the training. However, in Q-learning, you will always see many unexplored states with Q table to be zero. And it behaves more stable once it has generated an optimal policy. That is because it uses the maximum value of the next state instead of still following the next state-action pair. That makes it behaves optimal, steady, and at the same time, dangerous.

Other methods that combines the multi-layer perceptron is more generalizable, like deep Q network, A2C, etc. And these methods will be implemented in the second part of the lecture.

At last, I want to express my thank to Peter. The lecture notes are comprehensive, meticulous and of great quality. It is the best reinforcement learning guide I have ever been read. Thanks for your work, it is awesome! And thank you for reading till here.