ME5406: Deep Learning for Robotics

Part I Project Report

An Analysis of the Frozen Lake problem with Model-Free Reinforcement Learning Algorithms

---

Student Name:
Arijit Dasgupta

Student Number:
A0182766R

Student Email: **arijit.dasgupta@u.nus.edu**

October 17, 2020

Contents

1 Introduction

1.1 Problem Statement

For this project, the problem statement is split into two tasks. The rules of task 1 are as follows:

- Figure 1 illustrates a 4 by 4 grid in which the objective is for the robot (agent) to glide across the grid (environment) to reach the frisbee (goal & terminal state) without falling into any holes (terminal states). The robot emoticon represents the robot, the skull emoticon represents a hole, the small circle represents a non-terminal state in the grid and the double circle represents the frisbee.
- The action space is restricted to 4 actions: up, down, left, right. The state transition probabilities, $P(S'|S, A) = 1$ for all intended actions and states (e.g. left means the robot always moves **left**).
- The robot cannot leave the grid, any action that attempts the leave the grid will lead to a final state that is unchanged. This was not explicitly stated in the project requirements, but it shall be assumed.
- An episode will terminate if the robot ends up in a hole or reaches the frisbee.
- A +1 reward is sent to the agent when the robot reaches the frisbee, a -1 reward is received when the robot falls into a hole and a 0 reward for all other cases. This reward system is fixed and will be strictly followed in the course of this project.

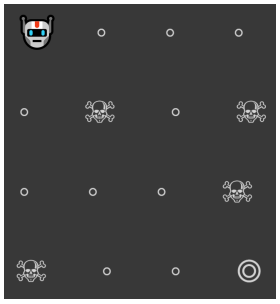


Figure 1: Command-Line UI representation of the 4 by 4 Frozen Lake Problem

The rules of task 2 are the same as task 1 with the following exceptions:

- The grid size is changed to 10 by 10 with the proportion of holes remaining at 25%.
- The holes are to be randomly assigned while ensuring that there is an unblocked and valid path from the robot to the frisbee. An example of such a grid is shown in Figure 2.

Figure 2: Command-Line UI representation of the 10 by 10 Frozen Lake Problem

1.2 Model-Free Algorithms

In this section, I will briefly touch on the three model-free reinforcement learning algorithms for the problem statement. It is to be noted that the four-argument P function, $P(S', R|S, A)$ is known, as the state transition probability, $P(S'|S, A)$, is equal to 1 for all state-action pairs and the reward system is fixed. Hence, we can technically use Dynamic Programming to solve this problem. But as the project requirements strictly stated Model-Free algorithms, I will stick to them. All three algorithms involve updating a Q table of the size (number of states, number of actions) in different manners.

The first algorithm, first-visit (FV) monte-carlo (MC) without exploring starts (ES) can only carry out all calculations and update the Q values after finishing an entire episode. A list is initialised for all state-action pairs before training, let us call this the "return" list. During training, the robot is forced to start from the top left coordinate given the "without ES" assumption. The robot follows an ϵ-greedy behaviour policy where by it chooses a random action at any state with probability ϵ, or a greedy action otherwise. After an episode ends, a variable G is initialised to 0 and the state, action, reward logs of the episode are backtracked. At each time-step, the G value is updated as $G = \gamma G + R_{t+1}$. If that state-action was visited for the first time in the episode, then the updated G value is appended to the "return" list and the Q value for that state-action pair is updated by taking the average of all the G values in the "return" list. This calculation repeats for all time steps (backtracked) in the episode and for every episode until the training ends.

The other two algorithms, SARSA and Q-learning (both Temporal Difference (TD) algorithms) are different as they allow for the updating of Q values before an episode ends. They can even update after every time step. That is implemented in the present project. The difference between the algorithms is the manner in which the Q value is updated. In SARSA, the q value for a state-action pair is updated as shown in Equation 1, while the one for Q learning is shown in Equation 2.

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

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

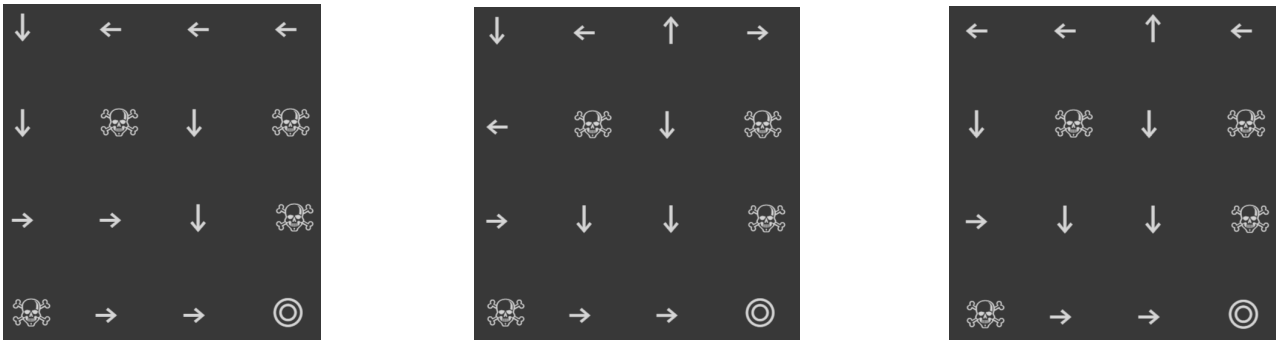
This means that SARSA follows an ϵ-greedy target policy while Q learning follows a greedy target policy.

1.3 Outline of Report

The first part of the report will go through a very detailed analysis of task 1 while the next part will focus on hyperparameter tuning with Q Learning for task 2. After that I will clearly outline and describe my own initiatives and features that I have added into the project. These initiatives are aspects of the project that were not strictly required. Afterwards I will go through the obstacles and difficulties faced and how I overcame them. I end the report with by highlighting what I learnt and ideas for extending this project.

2 Analysis of Task 1

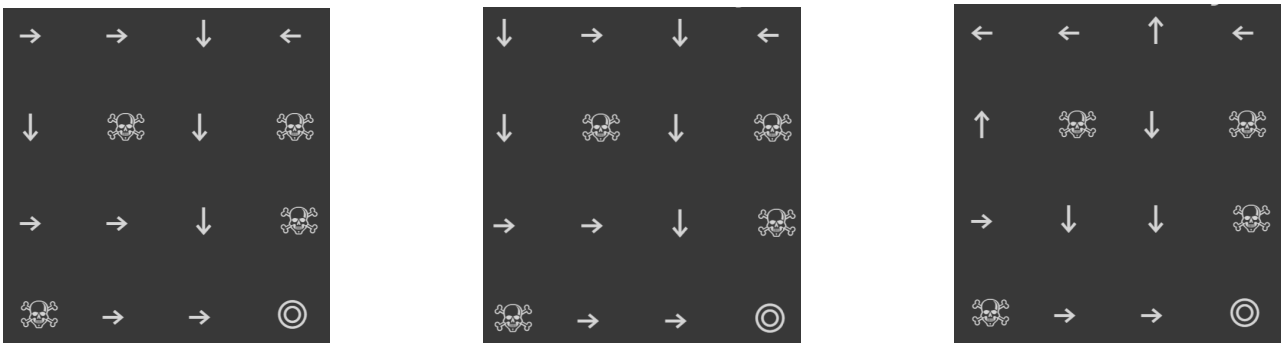
2.1 Comparing the three algorithms



(a) Q Learning (b) SARSA (c) FV-MC without ES

Figure 3: Policy after training for 1000 episodes with a constant ϵ of 0.5

As the 4 by 4 grid is a relatively small grid, I compare the performance of all three algorithms by running them at a relatively low 1000 episodes, with a discount rate (γ) of 0.9 and a learning rate (α) of 0.1. I also define custom ϵ values throughout training; an example is as follows: [0%: 1.0, 50%: 0.5, 90%: 0.1]. This means that from 0% (episode 0) of the training to just before 50% of the training (episode 499), the ϵ is 1.0, and so on. However for now, we start simple by using a constant ϵ value of 0.5 for all episodes. Figure 3 shows the final policies of all three algorithms. Note that all policies are derived by taking the greedy action for all non-terminal states after training. Notice that in the policy for Q learning in Figure 3a, no matter which non-terminal state you are in, you will always end up at the goal. However, it is not fully optimal yet as the policy in the top row unnecessarily makes the robot go to the left when it has a faster route down. It is obvious that the policies for SARSA and FV-MC without ES in Figures 3b & 3c are not optimal as in some states, the robot will never reach the goal. Nonetheless, one sign that some training has occurred is evident in the fact that **their policies never push the robot into a hole**. Let us repeat the training process for a larger 5000 episodes and a custom ϵ schedule of [0%: 1.0, 50%: 0.5, 90%: 0.1]. The policies are shown in Figure 4.



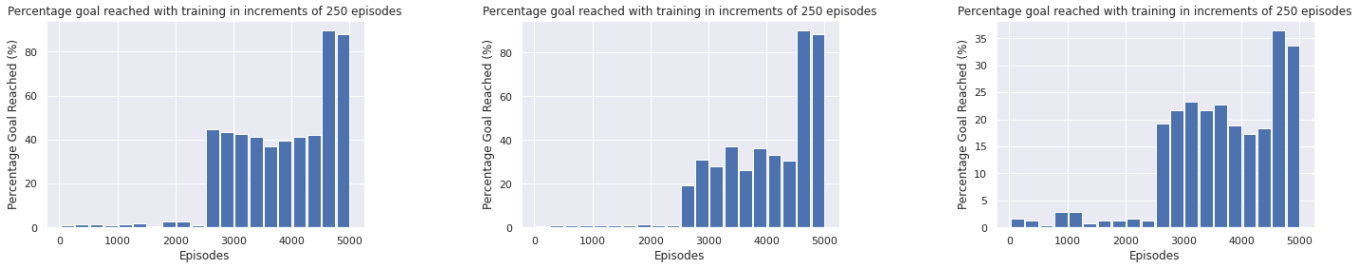
(a) Q Learning (b) SARSA (c) FV-MC without ES

Figure 4: Policy after training for 5000 episodes with an ϵ schedule of [0%: 1.0, 50%: 0.5, 90%: 0.1]

Figures 4a and 4b clearly show that both Q learning and SARSA have achieved optimal policies. However the policy by FV-MC without ES in Figure 4c still makes little sense. This is further evident in Figure 5 that shows how often the robot is able to reach the frisbee at different stages of training. One aspect common in Figures 5a to 5c is that the percentage goal reached is relatively extremely low till about half the episodes at which point the performance improves significatly, and the same happens at the 90% mark. This is expected as it aligns with the ϵ schedule. When the ϵ is set at 1.0, the agent is only choosing random actions, hence it is natural that the agent will not reach the goal most of the time. As the ϵ decreases, the likelihood that the agent will reach the goal by exploiting increases. At the final 10% of training, Q Learning reached the goal 88.0%, while SARSA did it 88.6% of the time and FV-MC without ES only reached the goal 33.6% of the time. This is also evident in Figure 5. Even though Q learning and SARSA achieved optimal policies, it did not achieve 100% as there was still a positive ϵ value of 0.1, hence sometimes the agent would take a random action that was sub-optimal.

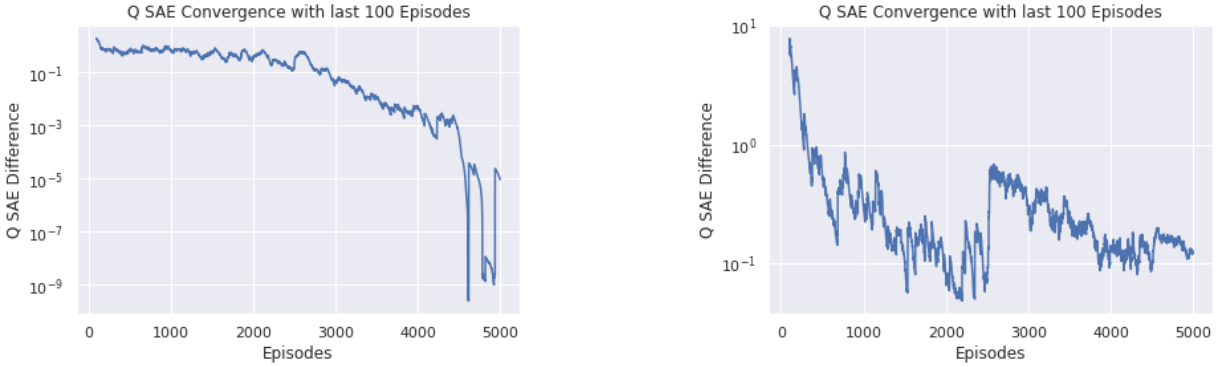
To further investigate why MC-FV without ES did not perform well, I looked at the moving-average convergence of the Q values. I did this by calculating the Sum Absolute Error (SAE) between the Q table in the current time step, and the averaged Q table of the last 100 time steps. This is like calculating convergence with a moving average.

Figure 6 illustrates the moving-average convergence of Q learning and FV-MC without ES.



(a) Q Learning (b) SARSA (c) FV-MC without ES

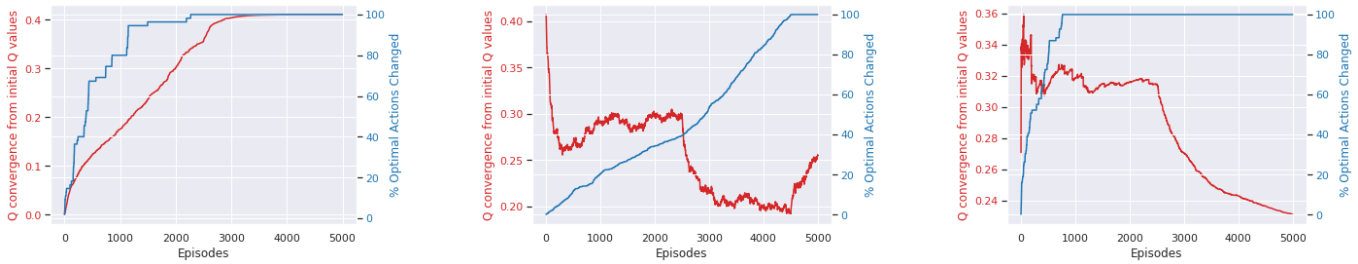
Figure 5: Percentage Goal Reached during training for 5000 episodes with an ϵ schedule of [0%: 1.0, 50%: 0.5, 90%: 0.1]



(a) Q Learning (b) FV-MC without ES

Figure 6: Moving-average convergence with last 100 time steps during training for 5000 episodes with an ϵ schedule of [0%: 1.0, 50%: 0.5, 90%: 0.1]

Figure 6a shows how the Q learning algorithm converges with training, but there are constant fluctuations in Figure 6b for the moving-average convergence of FV-MC without ES. Additionally, the Q convergence values are many magnitudes higher for FV-MC without ES. This may indicate that the number of episodes for training for the FV-MC witout ES algorithm is not enough. Other than this plot, I came up with two more ways to visualise the convergence of the different algorithms. The first is to measure how often the policy changes. This can be measured by the number of times an optimal action in any state for the greedy-policy determination changes with the episodes. The number of such changes can be cumulatively recorded with each episode and normalised to a 0 to 100% scale. This is represented by the blue lines in Figure 7. When the blue line reaches 100%, it means that the policy will no longer change for the remaining duration of training. The second way to visualise convergence is by calculating the Mean Squared Error (MSE) of the Q table at the current time step with the Q table before starting training. So instead of a moving average, the convergence is with respect to the Q table during initialisation. As the Q tables are initialised with zeros, this value is calculated by simply taking the mean of the sum of the squared Q values in the table. This is represented by the red lines in Figure 7.

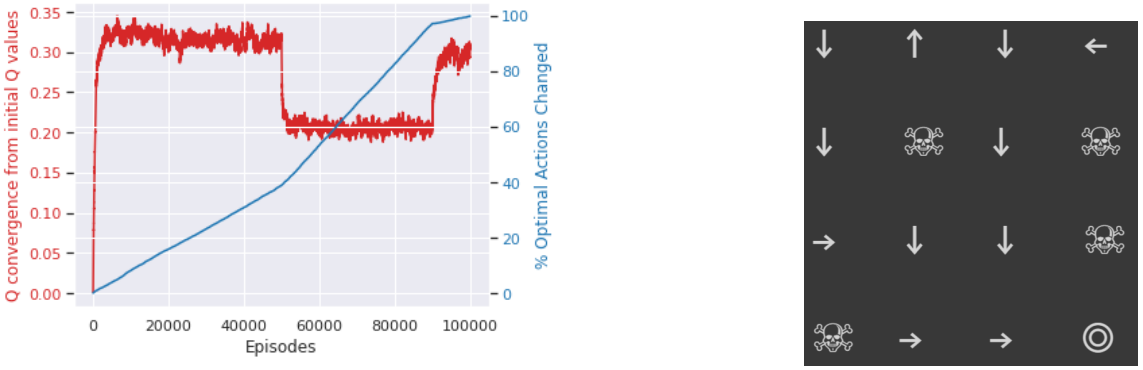


(a) Q Learning (b) SARSA (c) FV-MC without ES

Figure 7: Policy Convergence (blue) and Q Convergence (red) during training for 5000 episodes with an ϵ schedule of [0%: 1.0, 50%: 0.5, 90%: 0.1]

The first major difference is that the policy converged much faster for Q Learning at around 2250 episodes in Figure 7a as compared to the 4500 episodes in SARSA in Figure 7b. This means that Q Learning is able to arrive at the optimal policy the fastest. However, the Q convergence tells us that SARSA may not have truly converged as shown by the erratic changes of the red line in Figure 7b, showing no evidence of convergence. This same trend is seen for FV-MC without ES in Figure 7c. This is unlike the case for Q learning in Figure 7a where the red line visibly converges after 3700 episodes. It is possible that the SARSA algorithm has not had enough episodes to internally converge the Q values, but as the gridworld is so small, it is easier to obtain an optimal policy before requiring the Q values to converge. This specific possibility can be seen with Q Learning in Figure 7a, where the policy converges about 1500 episodes before the Q values converge. To check this, I tried running SARSA and FV-MC without ES for 100,000 (20x higher) episodes with the same ϵ schedule.

2.2 Pushing the limits of SARSA & Monte-Carlo



(a) Policy Convergence & Q Convergence

(b) Policy

Figure 8: SARSA training for 100,000 episodes with an ϵ schedule of [0%: 1.0, 50%: 0.5, 90%: 0.1]

Surprisingly, I found that the Q values for SARSA are not converging and that the policy is very unstable throughout training. Running SARSA for 100,000 episodes howevery yields a near-optimal policy as seen in Figure 8b (second arrow from left, top row is not optimal). Figure 8a illustrates the convergence for this run. The policy is always changing, up till the point the training ends. The policy is changing steadily when the Q convergence is fluctuating rapidly. It is possible that the agent is switching constantly between two possible optimal policies. Figures 4a and 4b are examples of two different optimal policies as no matter which state the robot is in, it will end at the goal with the same and least number of time steps. That could explain why the policy is seemingly constantly changing. Another crucial finding is that the Q values are extremely sensitive to ϵ , as the big shifts in Q convergence corresponds with the changes in ϵ .

(a) $\gamma = 0.9$

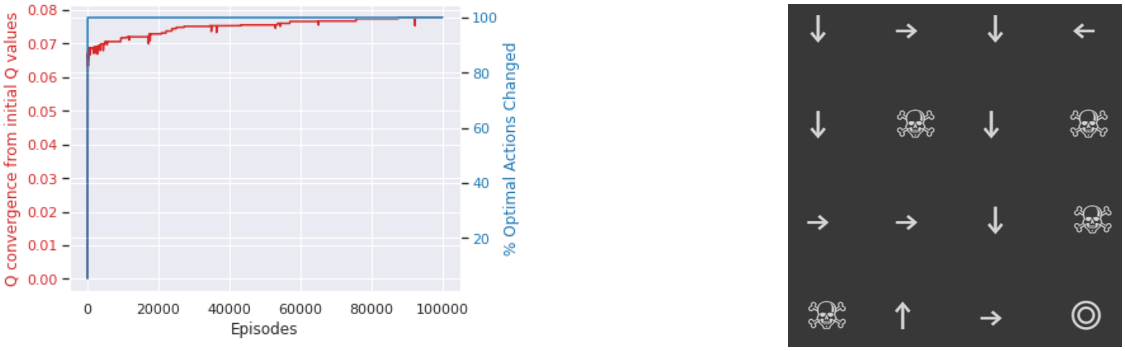
(b) $\gamma = 1.0$

Figure 9: Policy of FV-MC without ES training for 100,000 episodes with an ϵ schedule of [0%: 1.0, 50%: 0.5, 90%: 0.1]

In the case of FV-MC without ES, an interesting finding I had was that the discount factor made a big difference to the performance of the algorithm. Figure 9 shows the policies derived from running at $\gamma = 0.9$ and $\gamma = 1.0$. Figure 9a shows that at $\gamma = 0.9$, the agent is unable to get the optimal policy, including the top left grid (initial position), which is pointing left. My hypothesis is that, during training, the agent would often bump left in the first few time steps while exploring. Then eventually if it ended up at the hole, the multiple steps (of moving left at initial position) of discounting would mean any negative reward of moving left would be heavily diminished as only the first visit is considered. If the robot moves right or down, the robot can fall into the hole in the next time step, meaning that the

negative reward for those states are much more pronounced with less discounting. This inadvertently made the agent think that going left at the initial position will lead to better long term discounted reward. One could point out that this would work the opposite way, where the positive rewards to moving left in the initial position is more diminished as compared to moving right or down. However the latter effect does not balance the former as the agent is more likely to fall into holes at the start of training than reach the goal. Figure 9b shows that no discounting ($\gamma = 1.0$) gave an optimal policy, shedding some confirmation on my hypothesis. A way to avoid this effect would be to have negative rewards for attempting to leave the gridworld. Moving back to the case of SARSA, given that it is so sensitive to the ϵ, I decided to run it again for 100,000 episodes, but with an ϵ value of $\epsilon = \frac{1}{t}$. Hence the ϵ is always decreasing steadily instead of having abrupt changes.

2.3 Experimenting with ϵ for SARSA

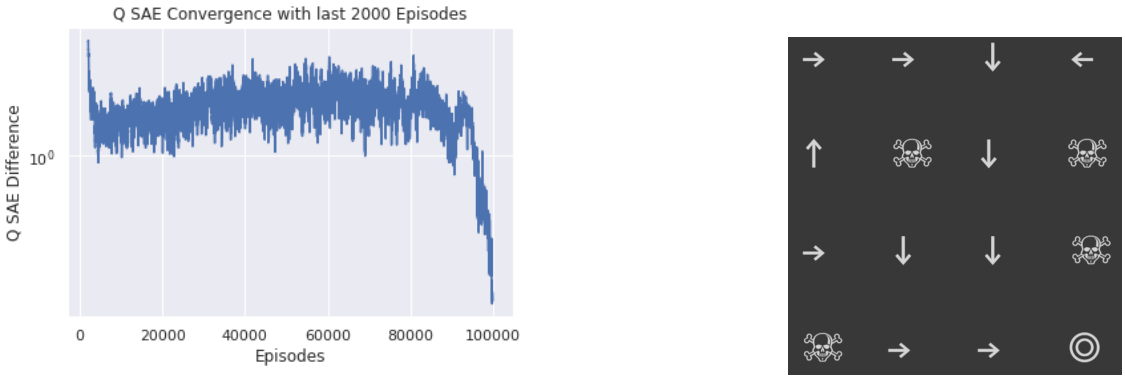


(a) Policy Convergence & Q Convergence

(b) Policy

Figure 10: SARSA training for 100,000 episodes with $\epsilon = \frac{1}{t}$

By implementing the new ϵ schedule in SARSA, Figure 10a shows that the policy converges rapidly, with a near-optimal policy as seen in Figure 10b. It is not surprising that the policy converged so fast as the non-linear relationship in $\epsilon = \frac{1}{t}$ means that ϵ drops very quickly. Hence I wanted to try this again but with a steadily & linearly decreasing ϵ from 1 to 0 with the relation of $\epsilon = \frac{T-t}{T}$ where T is the total number of time steps and t is the current time step. Figure 11 shows the results for this run. Figure 11a shows that the SARSA finally converges well with the training. Similar to previous runs, SARSA gave a near-optimal policy as seen in 11b. At this point, the results appear to agree with the notion that SARSA usually gives sub-optimal solutions that are near to the optimal, while Q Learning ends up converging to the optimal.



(a) moving-average convergence with last 2000 time steps

(b) Policy

Figure 11: SARSA training for 100,000 episodes with $\epsilon = \frac{T-t}{T}$

2.4 Hyperparameter Tuning for Q-Learning: Learning & Discount rates

Before ending the analysis for part 1, I implemented some hyper-parameter tuning for the Q Learning algorithm for two hyper-parameters, learning rate & discount rate. All tuning runs are done for 100,000 episodes with an ϵ schedule [0%: 1.0, 50%: 0.5, 90%: 0.1]. While varying the learning rate, the discount rate is fixed at 0.9. Figure 12 shows the summary of varying the learning rate from 0.001 to 0.5, with **all agents converging to the optimal policy**. The results clearly show one conclusion, a higher learning rate leads to faster overall convergence. 12a shows clearly that a higher learning rate converges very quickly and as the learning rates go down, it will converge to the same Q MSE value, but at a later episode. 100,000 episodes are not even enough for a learning rate of 0.001 to converge. Figure

12b further confirms this same conclusion that higher learning rates converge faster. This finding is not surprising as a higher learning rate directly indicates faster learning.

(a) (b) moving-average convergence with last 100 time steps

Figure 12: Q Learning tuning for different learning rates for 100,000 episodes

Finally, to vary the discount rate from 0 to 1 in intervals of 0.1, the learning rate is fixed at 0.1. Figure 13 highlights the results of the tuning. The agent with a discount rate of 0 failed to come up with an optimal policy, while all other discount rates converged to the optimal policy. This is not surprising as a 100% myopic view of the state-action values would mean that certain states (like the robot starting state) would have no update of its Q values as it is not directly adjacent to any terminal state. Moving on to the plots, the first observation to notice in both Figures 13a and 13b is that the agent in all discount rates converge at the same rate. This means that the rate of convergence is dependent on learning rate and not the discount rate. The final observation from Figure 13a is that all agents converge at a different Q MSE value, with a higher discount rate leading to generally higher Q MSE values. This is indicative that a higher discount rate generally leads to Q values of a higher magnitude. A mathematical proof of this would be fascinating, if it were always true. Nonetheless, expecting different Q values for different discount rates is expected as the discount rate directly controls the Q values. This concludes the analysis for task 1.

(a) Varying Learning Rates (b) moving-average convergence with last 100 time steps

Figure 13: Q Learning tuning for different discount rates for 100,000 episodes

3 Analysis of Task 2

3.1 Varying the Number of Episodes

For task 2, I decided to focus on conducting a hyperparameter tuning analysis on Q learning only as I already focused a lot on SARSA and FV-MC without ES in the previous section In particular, I will look at how the training and policy differs for the following hyperparameters: [number of episodes, learning rates, ϵ schedules]. First, I varied the number of episodes while keeping the ϵ schedule at [0%: 1.0, 50%: 0.5, 90%: 0.1]. Note that the same frozen lake grid from Figure 2 is used by fixing the random seed to 42.

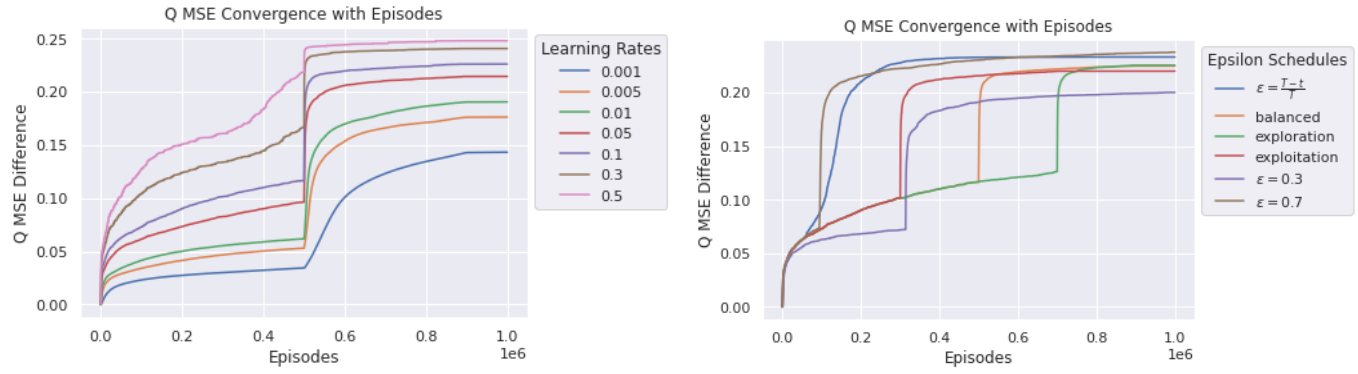
(a) 30,000 Episodes (b) 300,000 Episodes (c) 1,000,000 Episodes

Figure 14: Policy after training Q Learning in the 10 by 10 frozen lake from Figure 2

Figure 14 illustrates the greedy policy after training for different number of episodes. The "∼" in the map means that the Q values are all equal to each other, and thus there is no definite policy. This can happen to states where it is impossible for the robot to reach as it is surrounded by holes (e.g. 4th grid from the right of the bottom row of Figure 2) or the agent has not had enough episodes to explore all states in the frozen lake. This is very apparent in the bottom half of Figure 14a where most of the states are unexplored. In fact, even the robot starting region appears to be "untrained". If this was a monte-carlo algorithm, the robot starting area would have been certainly trained as the backtracking looks at all states visited (either first or every). However in this Q learning, the update is done after every time step, so it may take more episodes for the robot starting region to receive the effect of this update. This is evident in Figure 14b where training 10 times more at 300,000 episodes shows a greater degree of training. In fact the policy is nearly optimal, however some regions marked in red don't yet show the optimal action. Finally after training for 1,000,000 episodes as in Figure 14c, the policy becomes perfectly optimal, taking 15 minutes to run on Google Colab.

3.2 Varying Learning Rate & ϵ Schedule

This may not necessarily mean that the Q values have converged. Figure 15a shows the effect of varying the learning rate from 0.001 to 0.5 while running for 1,000,000 episodes each time. Similar to the conclusion from task 1, a higher learning rate led to a higher rate of change in Q values. However, the plot shows that even one million episodes is not enough for learning rates below 0.1 to come close to convergence. Given that all other hyperparameters like discount rate were kept constant ($\gamma = 0.9$), the Q values should converge to the same MSE value form initial. As the run in Figure 14c was done with a learning rate of 0.1, I conclude that the run may have converged to the optimal policy but not yet achieved Q convergence. In fact, only the runs with a learning rate of 0.1 or higher achieved the optimal policy while the other runs had one or two two states with sub-optimal actions.



(a) Varying Learning Rates (b) Varying ϵ Schedules

Figure 15: Q Learning tuning for 10 by 10 grid for 1,000,000 episodes

Finally, I tried adopting different ϵ schedules while keeping the learning rate at 0.1 while running for 1,000,000 episodes. I tried three custom schedules. The balanced schedule: [0%: 1.0, 50%: 0.5, 90%: 0.1], the exploration-focused schedule: [0%: 1.0, 70%: 0.5, 90%: 0.1] & the exploitation-focused schedule: [0%: 1.0, 30%: 0.5, 70%: 0.1].

Additionally, I tested keeping the ϵ fixed at 0.3 and 0.7 and implemented the $\epsilon = \frac{T-t}{T}$ relation. Figure 15b illustrates the Q convergence for the different ϵ schedules. Surprisingly, simply using a constant ϵ of 0.7 converged the fastest and performed the best. This shows that learning agents may not always learn as we expect. Performing almost as well was the $\epsilon = \frac{T-t}{T}$ implementation. The change in Q MSE value for the different custom schedules aligned with when the schedule changed ϵ from 1.0 to 0.5. This means that the exploitation-heavy converged faster than the other two. This is unsurprising given the huge number of episodes being run. Even in the exploitation schedule, the agent was allowed to go in full exploration mode ($\epsilon = 1.0$) for 300,000 episodes. I believe if the number of episodes run was drastically decreased, the exploration schedule could prove to be a better option. This concludes the analysis for task 2.

4 My Initiatives and Features

Other than meeting the bare requirements for the project, I have added many additional initiatives and features in my code to extend the capability of the model-free reinforcement learner for the frozen lake. Not all the capabilities were used to present the work so far, but they allow for more flexible and interesting use for anyone to play and test the model-free reinforcement learner. They are as follows:

1. The entire frozen lake environment was manually coded from scratch instead of using any existing OpenAI gym environment or even using OpenAI gym at all. Only necessary libraries like numpy, matplotlib, random etc. were imported. This gave me full access to control all aspects of the environment and agent and I was able to carry out low-level features and algorithms.

2. In the ϵ-greedy implementation of the code, if the greedy option has to be chosen and all the 4 actions have the same Q values, the action is also chosen randomly. In principle, if the Q values are all the same, it does not matter what action is chosen. I made this modification to avoid the code from auto-choosing the first index if all Q values are equal, which is to move up. Thus at the start of the training, when the robot enters a state where all Q values are zero and a greedy action is to be chosen, it would have always chosen up. Given that the Q values are all the same, it would probably help the agent more to take an exploratory move at this step instead of blindly moving up. It would still respect the greedy choice, as all actions would be equally greedy in this case. Hence, I made this modification.

3. Other than the "hardcoded" option of the 4 by 4 grid given in the project description, I added the ability to generate a custom N by M grid. I have the ability to choose N & M and even choose the starting coordinates of the robot and the frisbee. Additionally, I can also specify the fraction of states being holes. In this project, task 2 was kept at M = 10, N = 10, with the robot at the top left corner and frisbee at the bottom right corner. Additionally, the fraction of holes was fixed at 0.25 (25%) to match task 1.

4. When randomly assigning holes in the "custom" option, I developed an efficient depth first search function to ensure that there is a valid path from the robot to the frisbee, without searching online for help. At any grid, the algorithm recursively checks all vaild adjacent grids (if it is not yet checked), if the robot is there. This starts from the frisbee grid. As long as any leaf node from this recursive tree returns True, a valid path is present. The recursion methodology made the algorithm very fast and ensured that any custom grid or Exploring Starts presented in the command line is always valid.

5. Methods to measure and record policy convergence, moving-average Q Convergence from last x time steps and Q Convergence from the initial Q table were added to collect data on convergence during training. Data on whether the goal was reached was also collected.

6. A visualise method was added to the frozen lake class to display all plots for all the convergences and percentage goal reached.

7. The code allows for the monte-carlo algorithm to adopt an every-visit mode instead of first visit. The code also allows for the monte-carlo algorithm to adopt exploring starts. This gives the me more algorithms to try if I wanted to.

8. I can easily specify a custom ϵ schedule to implement custom schedules like [0%: 1.0, 50%: 0.5, 90%: 0.1] with one line, and my code will automatically generate a dictionary that holds the ϵ for every episode which is used when running the episodes. I can also specify this as "inverse" for $\epsilon = \frac{1}{t}$ and "linear_drop" for $\epsilon = \frac{T-t}{T}$.

9. I created methods to print the frozen lake and greedy policy in an aesthetic manner in the command line UI. Although this is not functionally relevent, it helps me or anyone running my code to easily understand the frozen lake map and the policy determined after training.

5 Conclusion

5.1 Obstacles and Difficulties

In general, the whole project went generally smoothly and there was no moment where I was disastrously stuck. That being said, there were indeed moments where I had to jump over hurdles. They are as follows:

- For this project, I challenged myself to not use OpenAI gym or any other libraries to aid me in creating the environment. Additionally, I only used built-in python data structures like lists and dictionaries (and numpy array for the Q table) to store all aspects of gridworld. It was quite difficult at first as I had to plan out everything carefully. For instance how do I represent the grid, the coordinates, the reward mappings etc? This became even more challenging as I designed the "custom" grid feature. I managed to overcome this with careful planning and patience and a lot of testing at every stage of development.
- I also challenged myself to implement all the model-free algorithms without looking online. This is difficult but I felt it was necessary to ensure that I understand all aspects of the algorithm. I overcame this by reading carefully through the notes and translating the pseudocode into python algorithms. The monte-carlo algorithms were particularly more difficult, but I managed to handle it using list mappings.
- Another obstacle was in making the code run faster. I overcame this by making certain changes to algorithms and data structures to eliminate any redundancy. In particular, I felt that the recursive depth first search function to check if there is a valid path from the robot to the frisbee was a huge time saver.
- The final challenge was dealing with randomness. As there are many random choices to be made with probability ϵ, every run of the code would yield different outcomes, although they were almost unnoticeable. It would frequently change the frozen lake map (10 by 10) with randomly assigned holes. I controlled for this by fixing the random seed in the python environment.

5.2 What I learnt from this Project

Ultimately, the most important part of this project is what I learnt from it. In all honestly, I have learnt more from part 1 of this module than I have from most NUS modules. The first main lesson was the entire theory of reinforcement learning, which spanned from the problem definition to Markov Decision Processes to dynamic programming, model-free methods and deep learning (although not needed for this project). For this project specifically, I gained a lot of confidence in my ability to code in Python, implement algorithms and efficiently use data structures. As I used the pseudo-code description from the part 1 notes to code the algorithms, I felt that my understanding of the algorithms were more solidified, including the other monte-carlo algorithms I added and were not necessary for the project. I also learnt how to manage my time in handling computational experiments. Many of the runs, especially hyperparameter tuning, took an immensely long time to run.

Other than my experience handling the project, I also gained more insights into the workings of the RL algorithms in the frozen lake problem. The findings solidified my understanding that Q learning converges to the optimal policy while SARSA usually adopts a near-optimal policy. FV-MC without ES works well with $\gamma = 1.0$ due to the reward structure of this project. Ultimately, I have shown that the Q values of all three algorithms can converge. I also learnt that higher learning rates converge the learning faster, while ultimately leading to the same Q values no matter what (in this case). I also learnt that SARSA is highly sensitive to the ϵ value for some reason, and that a higher discount rate appears to lead to Q values of higher magnitudes. I am not entirely sure why this is the case. I also found that using an ϵ with the relation $\epsilon = \frac{T-t}{T}$ worked very well for SARSA (Figure 11a) and Q Learning (Figure 15b, while the $\epsilon = \frac{1}{t}$ relation forces the learning to converge too fast. However, I also realised that learning agents may often not work the way you expect them to, revealing the empirical nature of the algorithms.

5.3 Further Work

There were many aspects of the project that I wanted to further explore, but time constraints and page limit constraints meant that I have to leave those ideas to this section. Some more interesting ideas would be to rethink the way this problem is structured. One very important aspect of how we expect the agent to learn is the way the rewards are structured. I would experiment by trying different reward structures, such as a negative reward when the robot takes an action that signals to leave the grid. One could also implement a time-based reward that discourages the robot from looping round the same path without reaching any terminal state. Also, I would try solving this problem with value iteration and policy iteration and see how they compare to the model-free methods. Another interesting idea is to see how changing the fraction of holes would affect the way the agent learns. Once could also considering carefully analysing how the speed of training is affected by the grid size and ϵ values. I noticed that the number of episodes per second reduced with the larger grid as well as with a lower ϵ value. This may be because the agent does not fall in to the hole easily, and takes more time-steps per episode. I understand my involvement with this project ends here, but I do hope students in the future will be given these ideas to implement and test in their projects. This concludes the report.