Implementation of Q learning and SARSA in Cliff Walking Problem

Introduction

In this report, we seek to find the optimal solution to the “Cliff Walking problem” (Figure 1) by implementing the Q-learning algorithm. In Q-learning, we define an agent that moves across the states within an environment (domain).

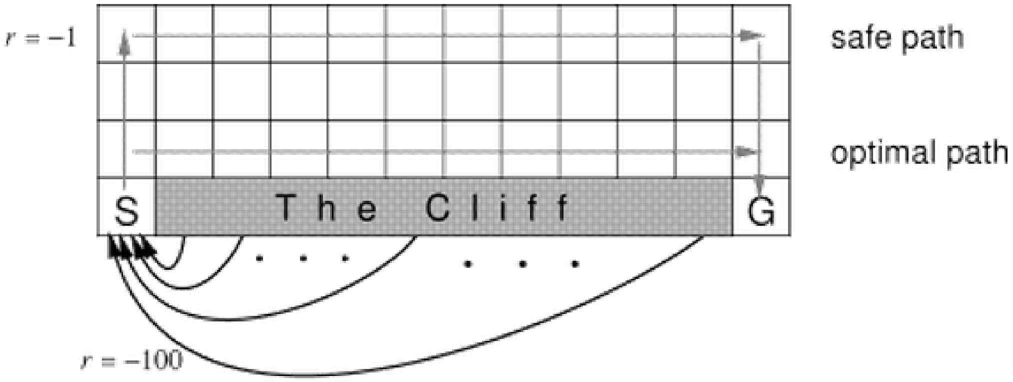


Figure 1 – Cliff Walking problem

The agent is initially positioned at the lower-left corner of the grid, and its task is to find the shortest way to reach the lower-right corner while avoiding the grey squares situated at the bottom. The agent is penalized by 1 reward unit each time it moves to a new state, 100 reward units by moving to the grey squares and is rewarded by 100 reward units by reaching the goal.

From a reinforcement learning perspective, the task of the agent is to find a policy that ensures the highest reward gained in a minimal number of steps. The challenge of this domain is that the agent is deployed far away from the goal, which creates difficulty for the agent’s learning as the reward observed (which will be discussed in later sections) requires time to propagate from the left side to the right side of the grid. In the optimal solution, the agent would need to take 13 steps to reach the goal, which is shown in Figure 1.

Code and code dependency

Python was selected as the programming language to conduct the experiment. The code was separated into 4 files: “Main_env.ipynb”, “Q_learning.py”,”Sarsa.py”, and “cliff.py”. “Main_env.py” executes the learning and visualisation process where the other files are the building blocks for the environment and algorithms. One key dependency of the code was OpenAI Gym (Brockman *et al.*, 2016), the “cliff.py” code depends on the Gym package as standard api to interact with agents. The “cliff.py” environment was originally created by OpenAI but unfinished, with incorrect reward assigned to the environment and failed to replicate the original experiment by Sutton and Barto, (1998). We modified the code to comply with the recent version of Numpy (a Python library) standard and bug fixing.

Representation

The domain is represented in form of a 4 x 12 grid, with each square is numbered and corresponds to a state. The agent is initially deployed at state 1 (blue) and seeks to reach state 12 (red) while avoiding states from 2 to 11 (grey). This representation as a grid facilitates the learning process of the agent with Q-learning while being identical to the initial domain.

37	38	39	40	41	42	43	44	45	46	47	48
25	26	27	28	29	30	31	32	33	34	35	36
13	14	15	16	17	18	19	20	21	22	23	24
1	2	3	4	5	6	7	8	9	10	11	12

🟦	Start	⬜	Normal states
🟥	Goal	▨	Cliff

Figure 2 – Representation of the Cliff Walking problem as a 4x12 grid

State transition function

The state transition function specifies the states that can be moved to by the agent given its current state and an action, which is as follow:

$$\mathbf{S_{t+1} = f\ (s_t,\ a_t)}$$

The agent is deployed at state 1 and is allowed to execute an action, which are "move up", "move down", "move left", and "move right" in order to reach adjacent states (if any). If the agent executes an action that would move it to the direction where no state exists, it would stay at the same position. The transition in this problem is always deterministic, i.e. the transition probability in transition function is always 1. The following tables describes the state transition matrix from each state to another:

State	Up	Down	Left	Right	State	Up	Down	Left	Right	State	Up	Down	Left	Right
1	13	1	1	2	**17**	29	5	16	18	**33**	45	21	32	34
2	14	2	1	3	**18**	30	6	17	19	**34**	46	22	33	35
3	15	3	2	4	**19**	31	7	18	20	**35**	47	23	34	36
4	16	4	3	5	**20**	32	8	19	21	**36**	48	24	35	36
5	17	5	4	6	**21**	33	9	20	22	**37**	37	25	37	38
6	18	6	5	7	**22**	34	10	21	23	**38**	38	26	37	39
7	19	7	6	8	**23**	35	11	22	24	**39**	39	27	38	40
8	20	8	7	9	**24**	36	12	23	24	**40**	40	28	39	41
9	21	9	8	10	**25**	37	13	25	26	**41**	41	29	40	42
10	22	10	9	11	**26**	38	14	25	27	**42**	42	30	41	43
11	23	11	10	12	**27**	39	15	26	28	**43**	43	31	42	44
12	24	12	11	12	**28**	40	16	27	29	**44**	44	32	43	45
13	25	1	13	14	**29**	41	17	28	30	**45**	45	33	44	46
14	26	2	13	15	**30**	42	18	29	31	**46**	46	34	45	47
15	27	3	14	16	**31**	43	19	30	32	**47**	47	35	46	48
16	28	4	15	17	**32**	44	20	31	33	**48**	48	36	47	48

Figure 3: Transition table

Reward function

The “reward” obtained by an agent by executing an action from a state is defined by a reward function, which is as follow:

$$r_{t+1} = r(s_t, a_t)$$

The specific reward is represented in a tabular format, which is denoted as the reward matrix, or “R-matrix”, which will be discussed in later sections. The Reward function in this case is deterministic: it returns a fixed amount of reward given a state and action.

Policy

Policy can be understood as a decision rule that determines the action that the agent would take given its current state. In this domain, we selected the ε-greedy policy, following which the agent would take a random action with a probability ε. Such random action allows the agent to explore the domain. When not taking a random action, the agent would execute an action that yields the highest possible reward given its “knowledge” about the domain, which is the Q-matrix in this case.

The probability ε is an hyperparameter, ranging from 0 to 1, controls how likely the agent will execute a random action, with 0 being not taking any random action (“pure exploitation or greedy policy”) and 1 implies that the agent would behave randomly (“full exploration”).(Sutton and Barto, 2017). In addition, ε is set to decrease over time by a factor, denoted as λ, which is a value between 0 to 1 that controls the decaying exploration factor. In specific, at the end of each epoch, ε is updated as follow:

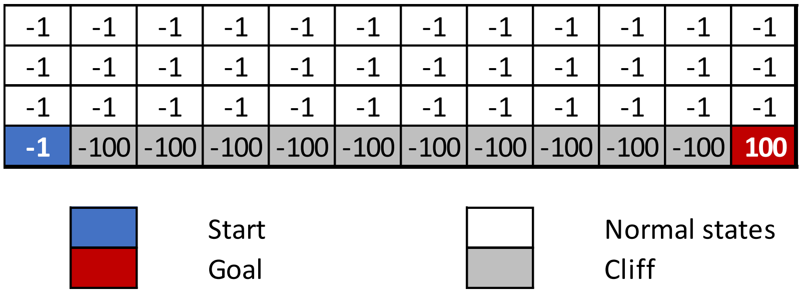
$$\varepsilon = \varepsilon (1 - \lambda)$$

By decreasing ε over time, the agent would start by taking random actions to explore the domain, then gradually seek to exploit its knowledge about the domain to execute the optimal set of actions that maximises the returns. In this experiment we adjust ε to observe the effect of exploration and exploitation on Q-learning. Exploitation represent the action that the agent would executive based on its knowledge of best action. The exploitation policy can be denoted as below (Mitchell, 1997):

$$\pi^*(s) = \text{argmax}\{Q(s_{valid}, a_{all})\}$$

The original R-matrix

The R-matrix specifies the rewards that the agent would get by reaching certain state after executing an action. In this case, the reward is fully observable, that is the deployer of the agent has full knowledge of the possible state transitions as well as their corresponding rewards. The original R-matrix is represented in Figure 4, in which the value of each square denotes the return that the agent would get by reaching a certain state.



-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
-1	-100	-100	-100	-100	-100	-100	-100	-100	-100	-100	100

🟦	Start	⬜	Normal states
🟥	Goal	▨	Cliff

Figure 4 – The original R-matrix

Parameters for Q-learning

The Q-learning algorithm implemented in this task has 5 parameters, which are as follow:

- Ɛ: Exploration factor
- $\boldsymbol{\lambda}$: Decay factor
- $\boldsymbol{\pi}$: Learning policy
- $\boldsymbol{\alpha}$: Learning rate
- $\boldsymbol{\gamma}$: Discount factor

As discussed, Ɛ and $\boldsymbol{\lambda}$ control the exploration tendency and $\boldsymbol{\pi}$ controls the learning of the agent.

Meanwhile, the learning rate $\boldsymbol{\alpha}$ determines the degree to which the agent takes into account the reward obtained at each state when updating the Q-matrix. $\boldsymbol{\alpha}$ ranges from 0 to 1, with 0 implies no learning is taken place while 1 means that the Q-matrix is updated fully by the value of the reward and estimated return.

Finally, the discount factor $\boldsymbol{\gamma}$ controls how the agent takes into account future rewards. This factor ranges from 0 to 1, with 0 implies that the agent does not consider future rewards but only immediate rewards, while 1 makes the agent treat immediate and future rewards equally.

We begin with a set of parameters to obtain preliminary results, based on which the performance of the agent was assessed. We then tried different parameters value then decide on the best setting that yields the best result. Figure 5 specifies the values selected in our settings.

Parameters		Initial value	Grid search values
Ɛ	Exploration factor	0.1	0.03, 0.05, 0.1, 0.2
$\boldsymbol{\lambda}$	Decay factor	0	0.01
$\boldsymbol{\pi}$	Learning policy	Ɛ-greedy	Ɛ-greedy
$\boldsymbol{\alpha}$	Learning rate	0.9	0.1, 0.2, 0.3, 0.9
$\boldsymbol{\gamma}$	Discount factor	0.99	0.99,0.9, 0.7, 0.2

Figure 5 – Parameter values for Q-learning

The original Q-matrix

The Q-matrix represent the current knowledge of the agent of the domain. It is composed by the Q-values, which represents the reward and returns of being in a state. As the agent move around the domain, it gains more knowledge and updates the Q-matrix in accordance with the transition rule.

In this case, the original Q-matrix is set to be a 48x4 matrix of zeroes, which implies that the agent has no prior knowledge/bias of the domain. For it to decide which action to take, it looks at the current Q-matrix and chooses the action that yields the highest possible returns (Even-Dar and Mansour, 2003). If the agent is set to explore the environment, it takes a random action regardless of the Q-matrix according to the Ɛ-greedy policy. The values of Q matrix are then updated after each action is taken place and the corresponding return is observed. The Q updated can be expressed as formula below (Sutton and Barto, 2017):

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\Big[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\Big].$$

A key remark to this formula is that the value is updated based on the learned estimated max Q value instead of the actual reward taken place. This is the unique property of off-policy learning.

In the following section, we demonstrate how the Q-matrix is updated in one learning episode as the agent perform actions and react to the environment.

Updating the Q-matrix in a learning episode

1. Agent is deployed at state 1, denoted as s_0 (initial state);
2. Agent defines the possible transition from s_0, which are “Up” , “Right” , “Left”, and “Down”;
3. Agent decides which action to take according to the Ɛ-greedy policy. The agent chooses to explore, taking a random action, which is “Up”;
4. Agent is taken to state 13, denoted as s_1
5. Agent defines the possible transition from s_1, which are “Up”, “Down”, and “Right”
6. Agent updates the Q-matrix, in specific $Q(s_0, a_0)$:

$$\begin{aligned}
Q_{new}(s_0, a_0) &= Q_{old}(s_0, a_0) + \boldsymbol{\alpha}\ [(r(s_0, a_0) + \boldsymbol{\gamma}.\text{argmax}\{Q(s_{valid}, a_{all})\}-Q_{old}(s_0, a_0)] \\
Q_{new}(1, up) &= Q_{old}(1, up) + \\
&\quad \boldsymbol{\alpha}[(r(13) + \boldsymbol{\gamma}.\text{argmax}\{Q(13, up), Q(13,down), Q(13,right),Q(13,left)\}- \\
&\quad Q_{old}(1, up)] \\
Q_{new}(1, up) &= 0 + 0.1\ [(\ -1 + 0.99.\text{argmax}\{0, 0, 0\}-0] \\
Q_{new}(1, up) &= -0.1
\end{aligned}$$

The Q-matrix (partly shown) is then updated as follow:

		State																	
		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	...	48
Action	Up	-0.1	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	...	-
	Left	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	...	-
	Right	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	...	-
	Down	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	...	-

7. From s_1 (state 13), agent decides which action to take according to the Ɛ-greedy policy. The agent chooses to explore, taking a random action, which is “Right”;
8. Agent is taken to state 14, denoted as s_2;
9. Agent defines the possible transition from s_2, which are “Up”, “Down”, “Left”, and “Right”;
10. Agent updates the Q-matrix, in specific $Q(s_1, a_1)$:

$$\begin{aligned}
Q_{new}(s_1, a_1) &= Q_{old}(s_1, a_1) + \boldsymbol{\alpha}\ [(r(s_1, a_1) + \boldsymbol{\gamma}.\text{argmax}\{Q(s_{valid}, a_{all})\}-Q_{old}(s_1, a_1)] \\
Q_{new}(13, right) &= Q_{old}(13, right) + \\
&\quad \boldsymbol{\alpha}[(r(14) + \boldsymbol{\gamma}.\text{argmax}\{Q(14, up), Q(14, left), Q(14, right), Q(14, down)\}- \\
&\quad Q_{old}(13, right\ )] \\
Q_{new}(1, up) &= 0 + 0.1\ [\ -1 + 0.99.\text{argmax}\{0, 0, 0\}-0] \\
Q_{new}(1, up) &= -0.1
\end{aligned}$$

The Q-matrix (partly shown) is then updated as follow:

		State																	
		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	…	48
Action	Up	-0.1	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	…	-
	Left	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	…	-
	Right	-	-	-	-	-	-	-	-	-	-	-	-	-0.1	-	-	-	…	-
	Down	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	…	-

1. From s_2, agent decides which action to take according to the Ɛ-greedy policy. The agent chooses to explore, taking a random action, which is "Down";
2. Agent is taken to state 2, denoted as s_3;
3. Agent defines the possible transition from s_3, which none, as state 14 was originally defined as a cliff, which penalises the agent by 100;
4. Agent updates the Q-matrix, in specific $Q(s_2, a_2)$:

$$Q_{new}(s_2, a_2) = Q_{old}(s_2, a_2) + \boldsymbol{\alpha}\ [(r(s_2, a_2) + \boldsymbol{\gamma}.\text{argmax}\{Q(s_{valid}, a_{all})\}-Q_{old}(s_2, a_2)]$$

$$Q_{new}(14, \text{down}) = Q_{old}(14, 2) + \boldsymbol{\alpha}[(r(2) + \boldsymbol{\gamma}.\text{argmax}\{Q(2, \emptyset)\}-Q_{old}(14, \text{down})]$$

$$Q_{new}(14, \text{down}) = 0 + 0.1\ [(-101 + 0.99.\text{argmax}\{0\}-0]$$

$$Q_{new}(14, \text{down}) = -10.1$$

The Q-matrix (partly shown) is then updated as follow:

		State																	
		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	…	48
Action	Up	-0.1	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	…	-
	Left	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	…	-
	Right	-	-	-	-	-	-	-	-	-	-	-	-	-0.1	-	-	-	…	-
	Down	-	-	-	-	-	-	-	-	-	-	-	-	-	-10.1	-	-	…	-

Since each learning episode would take a minimum of 11 steps, we believe that repeating the above analysis would add little value. The episode continues until the agent reach the goal which is state 12. The Q-matrix is then updated in a similar way to the above, the episode count is added by 1 and the run terminates.

Presentation and analyses of experimental results

In all of our experiments, all Q-matrices are initialised as 4x12 matrices of zeroes. The agents were required to learn the Q-values and the policy that gives the highest cumulative rewards.

Representing performance by episodes

Figure 6 shows that in general, the agent's performance improved over time as the number of episodes increases. The cumulative reward obtained by the agent with learning rate of 0.9 is represented by an orange line. Analysing the curve suggested that after 20 episodes, the optimal path – which yields highest cumulative reward - has been learnt by the agent. However, there is still fluctuations in the cumulative reward obtained by the agent, and it does not always reach the goal with the highest cumulative reward. Every spike of negative reward indicated the agent has stepped in a grey square (cliff) in some episodes.

We identified two issues that are attributed to unstable performance, which are stochastic control policy and the convergence of the Q-matrix. The stochastic control policy implemented was Ɛ-greedy policy, which ensures the agent not only exploits with currently knowledge but also explores to learn the underlying reward of the domain. However, one of its disadvantage is that short term return is sacrificed for long term improvement. The exploration-exploitation dilemma was well studied in the multi-bandit problem (Sutton and Barto, 1998) and proved to be an essential element to the agent in many settings. Q-learning is an off-policy algorithm, following which the optimal policy is learnt independently to the control policy, this implies that it does not take the stochastic nature of control in their optimal solution as demonstrated in figure 8. Although the optimal policy is learnt, the action of agent is not deterministic, meaning when agent is at row 1, there is one fourth of Ɛ probability that it will step in the grey squares due to the Ɛ-greedy policy. The off-policy properties of Q-learning allowed the agent to learn the optimal policy but at the same time preventing it from achieving optimal return. In the following section, we compare the performance of Q-learning with an advanced on-policy algorithm which is “SARSA”, which aims to address mentioned problem.

Figure 7 and 8 demonstrate the greedy policy evaluated by Q learning at episode 20 and episode 1000. At episode 20, it was shown that the agent understands how to navigate from the starting point to the goal optimally (the red arrow) but failed to suggest optimal actions in most of the other states. This is due to the limited exploration allowed by a small number of episodes. As Ɛ set to 0.03, the agent chose to exploit most of the time with little exploration, which slowed down the speed of convergence. After 1000 episodes, the agent explored more states from the environment and started to learn the optimal action in most of the states as displayed in figure 8.

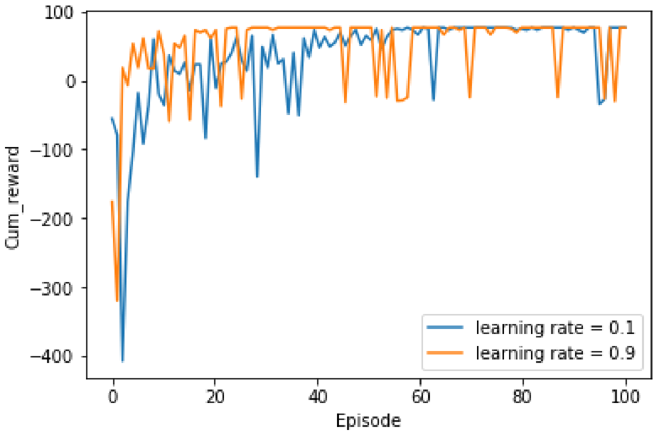
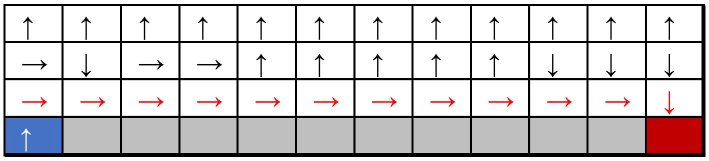
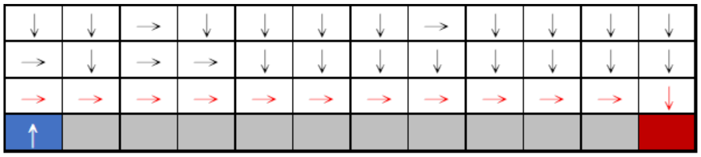


Figure 6 Performance vs episode



↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑
→	↓	→	→	↑	↑	↑	↑	↑	↓	↓	↓
→	→	→	→	→	→	→	→	→	→	→	↓
↑											

Figure 7 Greedy policy at episode 20 α =0.9



↓	↓	→	↓	↓	↓	↓	→	↓	↓	↓	↓
→	↓	→	→	↓	↓	↓	↓	↓	↓	↓	↓
→	→	→	→	→	→	→	→	→	→	→	↓
↑											

Figure 8 Greedy policy at episode 1000 α =0.9

Effect of Ɛ on performance

Increasing Ɛ would increase exploration thus gain more information about the underlying return matrix faster. This might increase the speed of optimality, at the cost of unnecessarily large negative reward. (Kaelbling *et al.*, 1996) Our results show that a higher Ɛ value would destabilise the agent. Figure 9 compares the performance of agents by different values for the Ɛ parameter. Higher Ɛ would cause the agent fail to reach target state more frequently incurring large negatives results. In Q-learning, the agent would converge to the optimal path in theory with infinite episodes. This is proven by the Bellman equation (Sutton and Barto, 2017). This means that in order to improve performance of the agent, one way is to have a decaying exploration strategy.

Here we took an alternative solution: We tried out different values for Ɛ and observe the result. Figure 9 shows unstable performance with high Ɛ value, be 0.05 and 0.01, while with Ɛ value of 0.03, the cumulative rewards obtained in each episode quickly averaged near 100. As such, it can be concluded that in this case, lower Ɛ value helps the agent converge more quickly and stabilises its performance. The long-term consequence of larger Ɛ is that the algorithm is unstable in long term and the cumulative reward curve is volatile due to the Ɛ greedy policy.

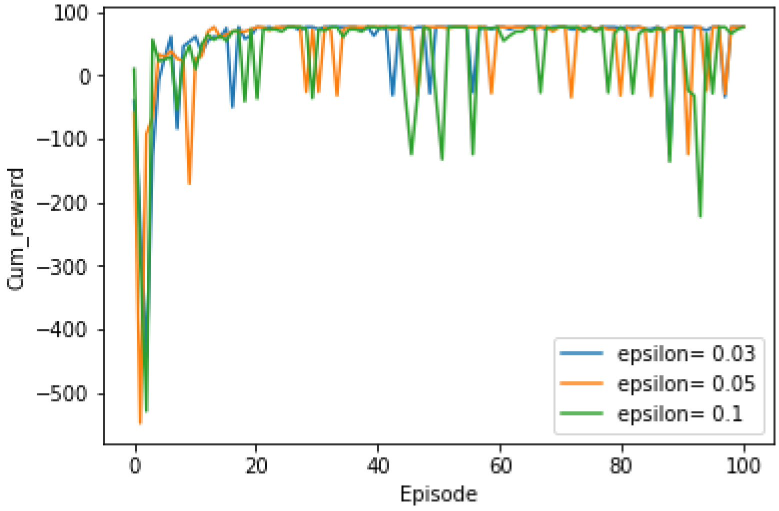


Figure 9 - Effect of epsilon value in Q-learning

<u>Effect of Ɛ, α, γ on performance</u>

We have also tried out different parameter value and observed the effect on the agent's performance. In all experiments, the agent was trained in 500 episodes. The results are summarised in Figure 10 as follow:

Ɛ	α	γ	Average reward	Episode taken to learn the optimal path
0.2	0.9	0.99	7.25	5
0.2	0.1	0.99	-34.04	11
0.2	0.9	0.2	-44.86	18
0.2	0.1	0.2	-79.01	36
0.03	0.9	0.99	56.58	12
0.03	0.1	0.99	32.08	21
0.03	0.9	0.2	51.32	8
0.03	0.1	0.2	-17.44	80

Figure 10 – Results obtained by varying Ɛ, α, γ parameters

It can be observed that learning rate α and exploration factor Ɛ contributed the most to the how well the agent performs in the long run. Learning rate of 0.9 and epsilon of 0.03 provides the best result. It shows that higher learning rate contributed to faster convergence as the agent adapted more quickly to the environment and adjusted any incorrect estimation of Q-value more quickly. It is known that Q-learning has the problem of over estimating bias (Hessel *et al.*, 2017) and using a small learning rate would preserves more prior results and thus taking longer for the algorithm to correct the reward value. Figure 6 shows the difference in learning speed when different learning rates are used. The agent with higher learning rate adapted to the environment more quickly, which results in improvement in performance, hence faster convergence. The result is consistent with previous research (Tau andIl, 2003) where higher learning rate is preferable in a Markov Decision Process context.

The grid search method allowed us to identify the optimal parameter choice, however computationally expensive. We tried out 4 different values for 3 parameters for Q learning, resulting in 64 combinations. This computation grows exponentially as we add more parameters and values.

Effect of Ɛ-decay factor on performance

From the above experiments, we have identified the learning rate of 0.9 and discount factor of 0.99 yield the best result. We ran 3 experiments using these 2 parameters value, set the Ɛ-decay factor $\boldsymbol{\lambda}$ at 0.01, and varying the exploration factor and observed the result, which is as follow:

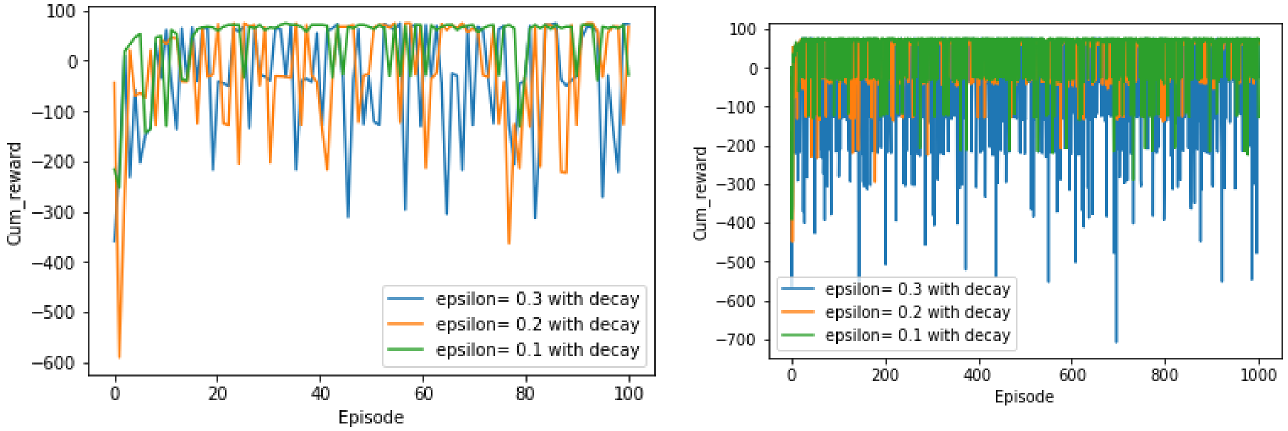


Figure 11 – Cumulative reward vs episode with different decayed value Ɛ

The plot suggests that implementing Ɛ-decay approach gives inferior performance to that obtained by using a constant Ɛ of 0.03 approach. Convergence was not achieved even after 1000 iterations. We identified that changing q-learning to decay exploration requires some fine turning grid search method. In consideration of expensive computational cost and no foreseeable improvement on performance, we choose an alternative algorithm SARSA over exploration decay q-learning for our advance algorithm presented below.

Selecting SARSA as learning algorithm

Despite sharing similarities with Q-learning as an iterative learning process, the key difference between SARSA and Q-learning is that SARSA is an on-policy algorithm: in SARSA, the Q-matrix is updated based on the action performed by the current control policy, while in Q-learning, it is updated in accordance with the greedy policy. In specific, the update equation is as follow:

$$\mathbf{Q_{new}(s_t,\ a_t) = Q_{old}(s_t,\ a_t) + \boldsymbol{\alpha}\ [r_{t+1} + \boldsymbol{\gamma}.Q(s_{t+1},\ a_{t+1}) - Q_{old}(s_t,\ a_t)]}$$

In SARSA, the next action (denoted as a_{t+1}) is the action performed in the next state (denoted as s_{t+1}) under the current control policy. By contrast, Q-learning has no constraint over the next action. The key different between current control policy update and max Q update is that, updating Q takes Ɛ -greedy into account.

Figure 13 and Figure 14 represent the results obtained by implementing the SARSA algorithm. SARSA does not output the optimal (shortest) path, but a path that is in between the safest and the optimal path. The parameter values for SARSA algorithm are as follow:

Parameters		Value
ε	Exploration factor	0.03, 0.2
λ	Decay factor	0
α	Learning rate	0.2
γ	Discount factor	0.99

However, SARSA doesn't guarantee generalisability. By learning the safer path, it does not guarantee to discover the optimal policy which leads to the optimal path. Figure 13 shows SARSA has superior performance over Q-learning in this task. Specifically, in long term performance, SARSA output a much smoother cumulative reward curve. In addition, there is a learning speed – performance trade-off. In comparison to Figure 6 and 9, SARSA learns much slower until the cumulative reward curve approach convergence asymptotically. This is arguable in the context that whether theoretical convergence is crucial to the agent.

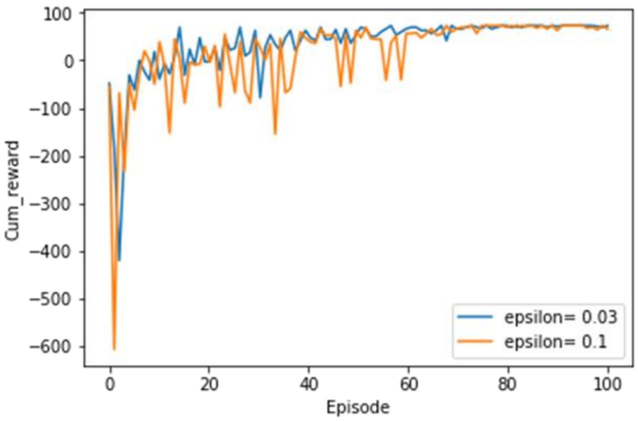


Figure 13
Performance vs episode - SARSA algorithm

↓	↓	→	↓	↓	↓	↓	→	↓	↓	↓	↓
→	→	→	→	→	→	→	→	→	→	→	↓
↑	←	↑	↑	↑	→	↑	←	↑	↑	↑	↓
↑	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	🟥

Figure 14 – Policy at episode 100, epsilon = 0.03

Conclusions

In this report, we critically evaluated two reinforcement learning models: Q-learning and SARSA. Both models managed to discover a path to the target state with different approaches. Q-learning guaranteed faster convergence to the optimal path at the cost of large negative rewards in most episode. By implementing different parameters for the exploration factor Ɛ, we noticed that smaller Ɛ leads to slower but asymptotic convergence and less volatile results. An alternative learning algorithm we adopted was SARSA, which eliminates the effect of stochastic control in the algorithm. SARSA output a safer path to target state that is in between the "safest" and the optimal path.

In both algorithms, smaller value of Ɛ is necessary to achieve good performance. To find out the set of parameters that leads to best agent's performance, we explored a range of value for such parameters in both models. This is not the ideal solution to model free learning as the purpose of model free learning is to let the agent automatically react to the environment. It would be interesting to explore other methods that require less fine-turning of the model.

References

Brockman, G. *et al.* (2016) 'OpenAI Gym'. Available at: http://arxiv.org/abs/1606.01540 (Accessed: 5April2018).

Even-Dar, E. and Mansour, Y., 2003. "Learning rates for Q-learning". *Journal of Machine Learning Research*, 5(Dec), pp.1-25.

Hessel, M. *et al.* (2017) 'Rainbow: Combining Improvements in Deep Reinforcement Learning'. Available at: http://arxiv.org/abs/1710.02298 (Accessed: 19March2018).

Kaelbling, L. P. *et al.* (1996) 'Reinforcement Learning: A Survey', *Journal of Artiicial Intelligence Research Submitted*, 4(9995), pp. 237–285. Available at: http://www.jair.org/media/301/live-301-1562-jair.pdf (Accessed: 16March2018).

Mitchell, T. 1997, *Machine learning*, McGraw-Hill, London.

Sutton, R. S. and Barto, A. G. (1998) *Reinforcement learning : an introduction*. MIT Press. Available at: https://mitpress.mit.edu/books/reinforcement-learning (Accessed: 16December2017).

Sutton, R. S. and Barto, and A. G. (2017) *Reinforcement Learning: An Introduction*. 2nd edn. MIT Press, Cambridge. Available at: http://incompleteideas.net/book/the-book-2nd.html (Accessed: 19March2018).

Tau, M. A. andII (2003) 'Learning Rates for Q-learning', *Journal of Machine Learning Research*, 5, pp. 1–25. Available at: http://www.jmlr.org/papers/volume5/evendar03a/evendar03a.pdf (Accessed: 11April2018).

Watkins, C.J. and Dayan, P., 1992. Q-learning. *Machine learning*, 8(3-4), pp.279-292.