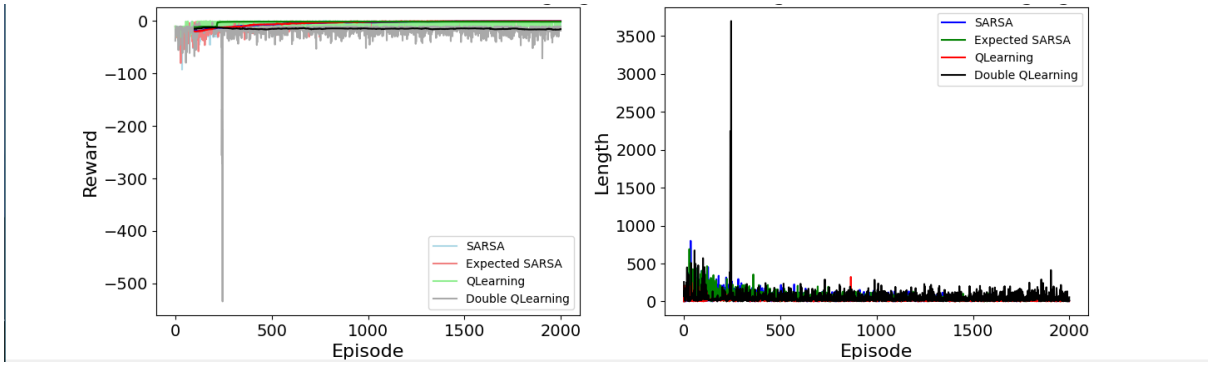
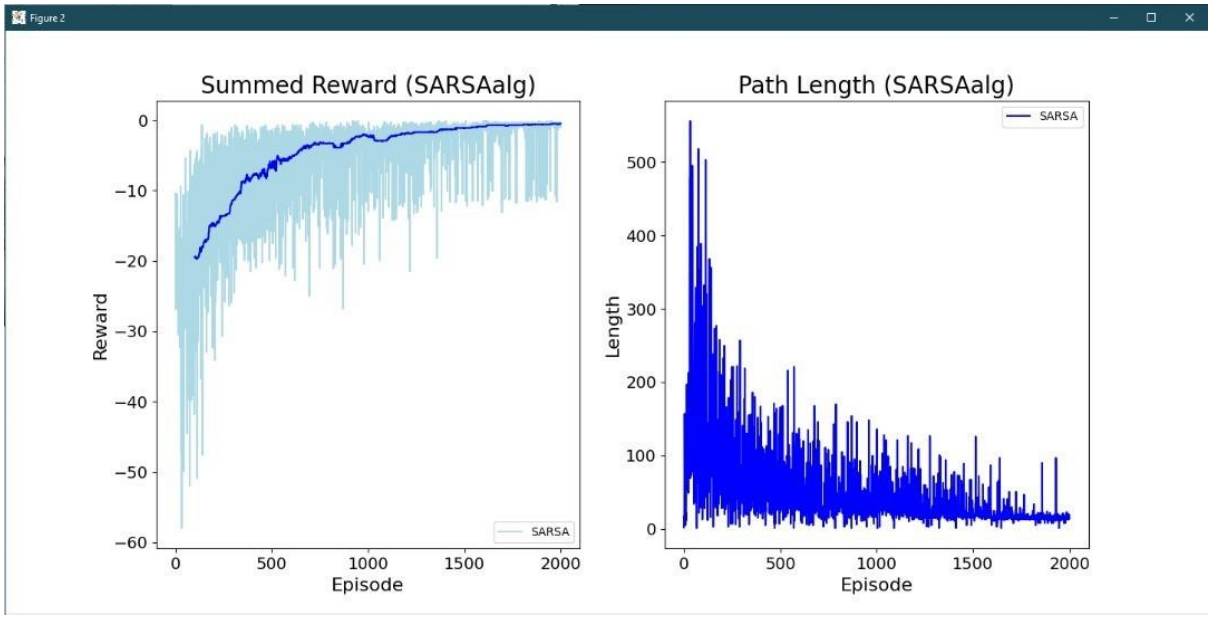
Assignment 2

ECE 457C : Reinforcement Learning

Instructor : Mark Crowley

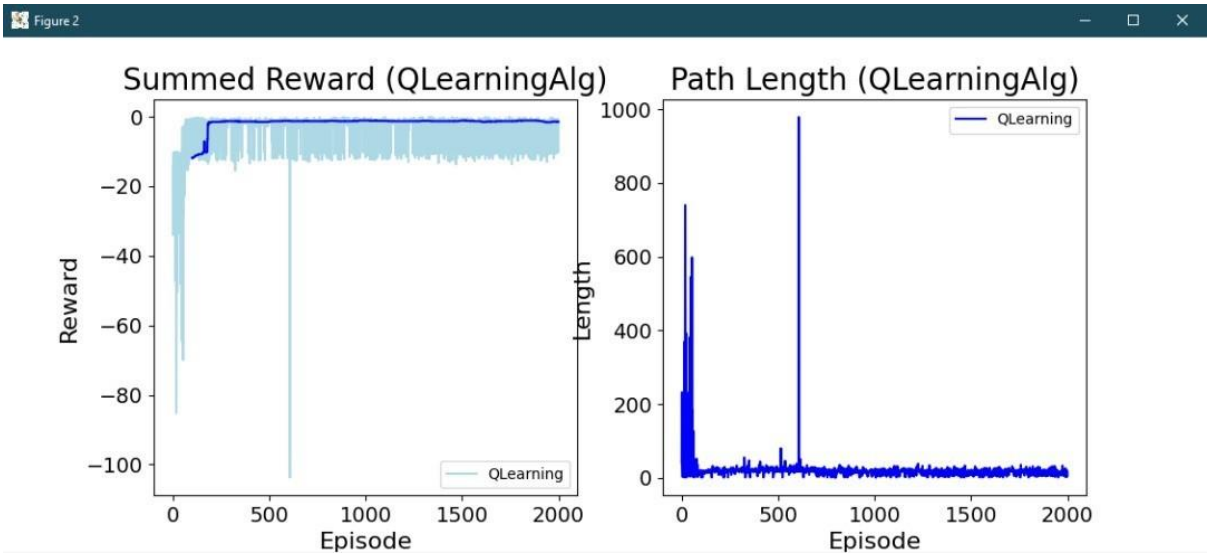
Student ID : Seoyoung Sim (20993717), Harish Parthasarathy (20793872)





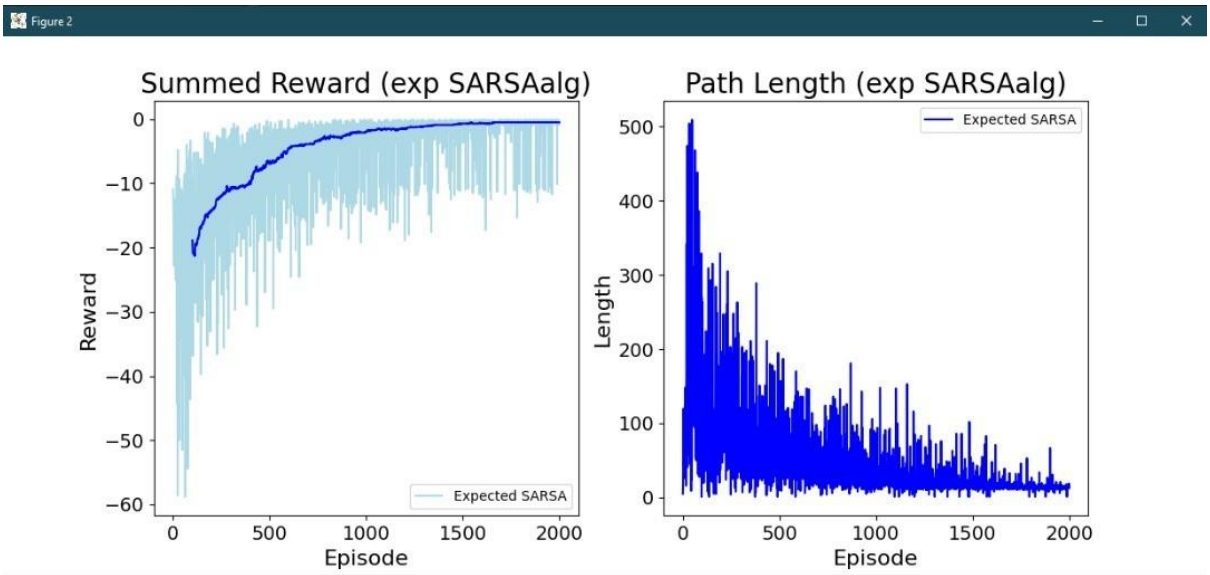
< SARSA >

$$\varepsilon = 0.1, \gamma = 0.1, \alpha = 0.01$$



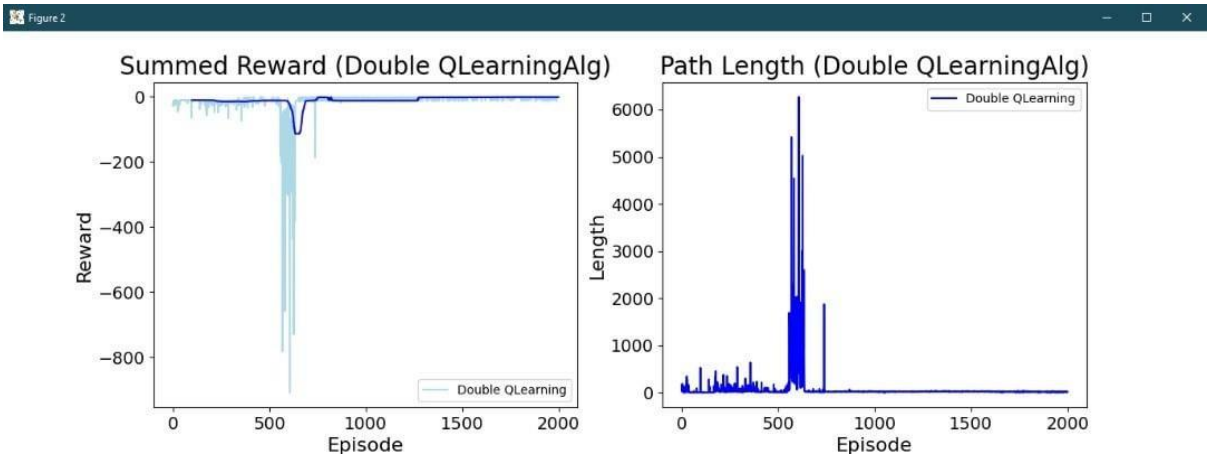
< Q-Learning >

$$\varepsilon = 0.1, \gamma = 0.1, \alpha = 0.8$$



< Expected SARSA >

$$\varepsilon = 0.1, \gamma = 0.1, \alpha = 0.01$$



< Double Q-Learning >

$$\varepsilon = 0.1, \gamma = 0.1, \alpha = 0.8$$

Hyperparameters used in code

1) α : Learning rate. This determines what extent the newly acquired information exceeds the old one. Depending on the learning rate, there may be cases that learning is not done properly so it is important to choose right value through experiences. If we choose too big learning rate, it overshoots so it does not learn/compute properly, if we choose too small learning rate, it needs to compute a lot so it takes too much time to end the episodes. Thus, we choose the proper learning rate which works well on task 2 through the experience.

2) γ : discount factor. This determines how much the agent cares about rewards in the distant future relative to those in the immediate future. If the discount factor is near 0, it is short-sighted which means that it only considers the next time step, if it is near 1, it is future-oriented which means that it considers future rewards important as much as the discount extends. So, if we choose too small discount factor, we cannot consider future rewards, and if we choose too big, we cannot consider each reward different through time steps. We choose the proper discount factor which works well on each algorithm on task 2 through the experience.

3) ε: It determines the probability to explore or exploit in ε-greedy policy.

Algorithms

1) SARSA

SARSA is an on-policy TD control algorithm that takes action from the current state. Then it will update the value function with the same action which is taken from the current state. Since SARSA is an iterative algorithm, the initial condition $Q(S_0, A_0)$ is implicitly assumed before the first update occurs. Whenever action is taken, the update rule increases the probability of selection by making it higher than other alternatives.

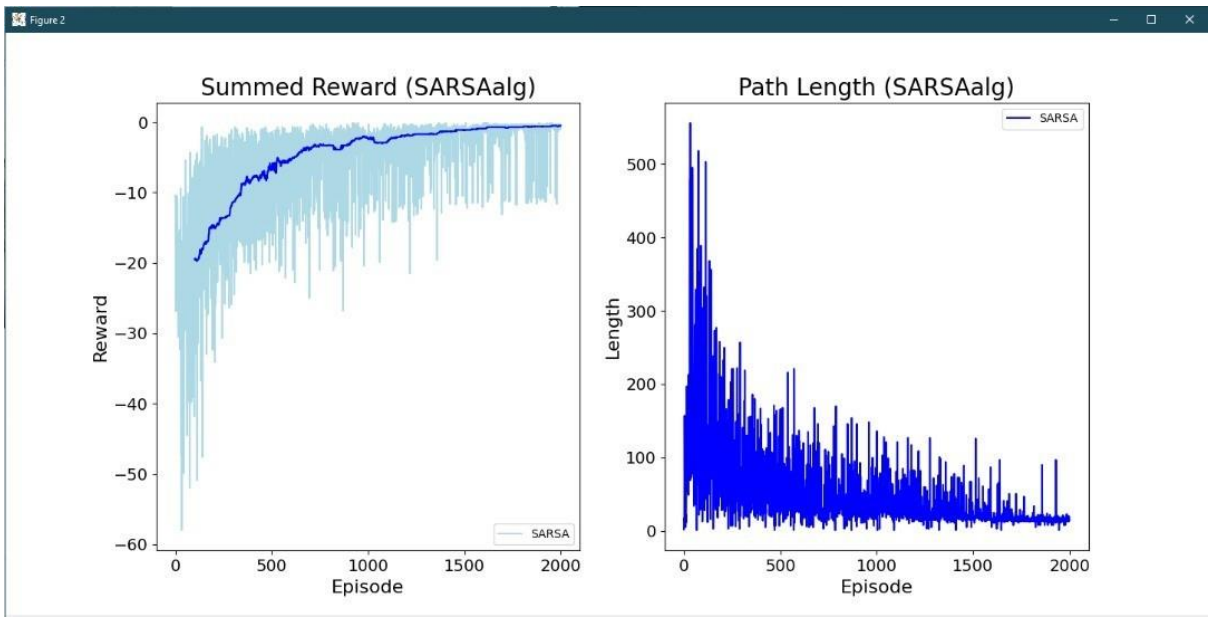
It chooses action a in state s by following Q-policy which is ε-greedy. During the episodes (in the loop) we checks states and rewards that are changed with every action. Then, choose next action a' by nest state s'. After this, the Q-value is updated and it continues until it raches the terminal state.

Value function : $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]$

The Q value of the state action is updated by an error and adjusted by the learning rate. The Q value represents the reward that can be received at the next time step to take the action and the future discounted rewards received at the action observation of the next state.

The policy implementation can be carried out immediately without going through all episodes, it is efficient if each episode is very long. So, SARSA can perform fast since it does not have to wait until all episodes end. But it can be inefficient since it is on-policy so it cannot reuse the data.

We choose $\varepsilon = 0.1$, $\gamma = 0.1$, $\alpha = 0.01$ as a parameter for this algorithm through the experience. We lower the learning rate so it can improve its performance.



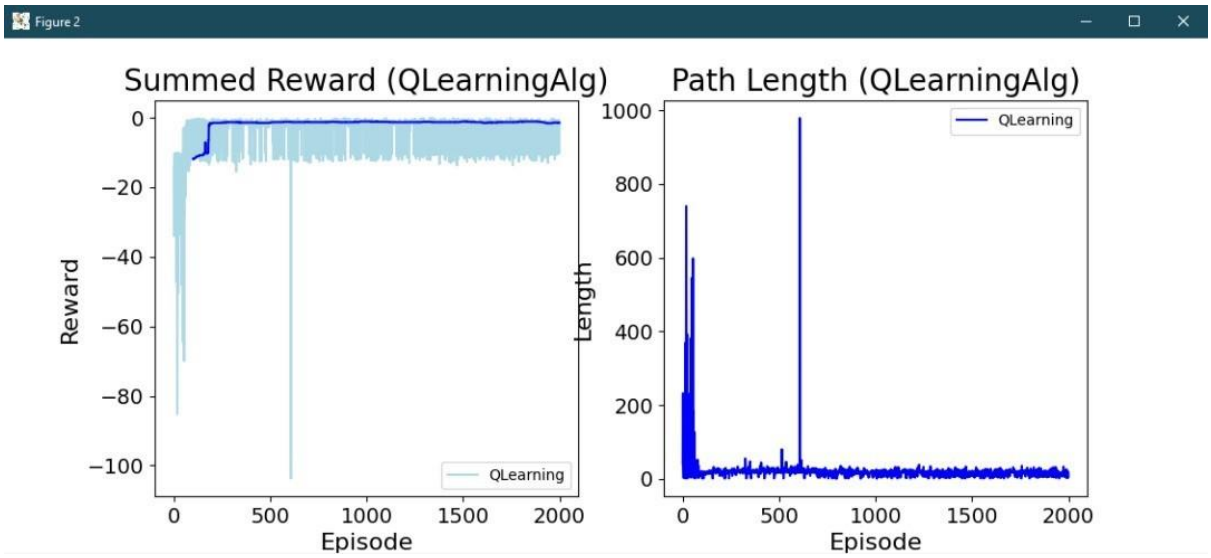
2) Q-Learning

Q-Learning is an off-policy TD control algorithm that chooses an action and sees the result. It will then update the value function with a different action, but chooses the best action based of that value function. There is no guarantee that Q-Learning will ever converge. The way that we have designed the code utilizes the main function that was given to us. For each episode, the initial action is decided outside of the individual episode loop, which is different from the Q-Learning pseudo code. However, this action can be used as the first action for the episode. In the while loop, we take the action, and observe the reward, and next state. We then call the actual learn function which uses the Q-Learning Bellman equation. This function first checks if the next state is an actual state where the agent can be (not a wall, or out of bounds). This also allows us to know if our state is terminal, in which case the algorithm can end. If it is not the terminal state, we can then update our Q table using the Q-learning equation. As well, we

can choose the next action for the next iteration of the loop here. This choose action function follows an epsilon greedy approach where we check if a random number is greater than epsilon (which is small number), we just explore in a random direction. If the number is less than epsilon, we exploit meaning the agent moves in the direction of the highest reward. This action is then used for the next iteration of the per episode loop. At the end, if the state is terminal, the per episode loop is ended and the next episode can begin.

The value function for Q-Learning looks like

$$Q(S,\ A)\ \leftarrow\ Q(S,\ A)\ +\ \alpha\left[R\ +\ \gamma\ \max_{a}(S',\ a)\ -\ Q(S,\ A)\right]$$



The performance for Q-Learning is faster when compared to SARSA as it is off-policy and able to use the past data to affect the result of the Q(S, A) lookup table.

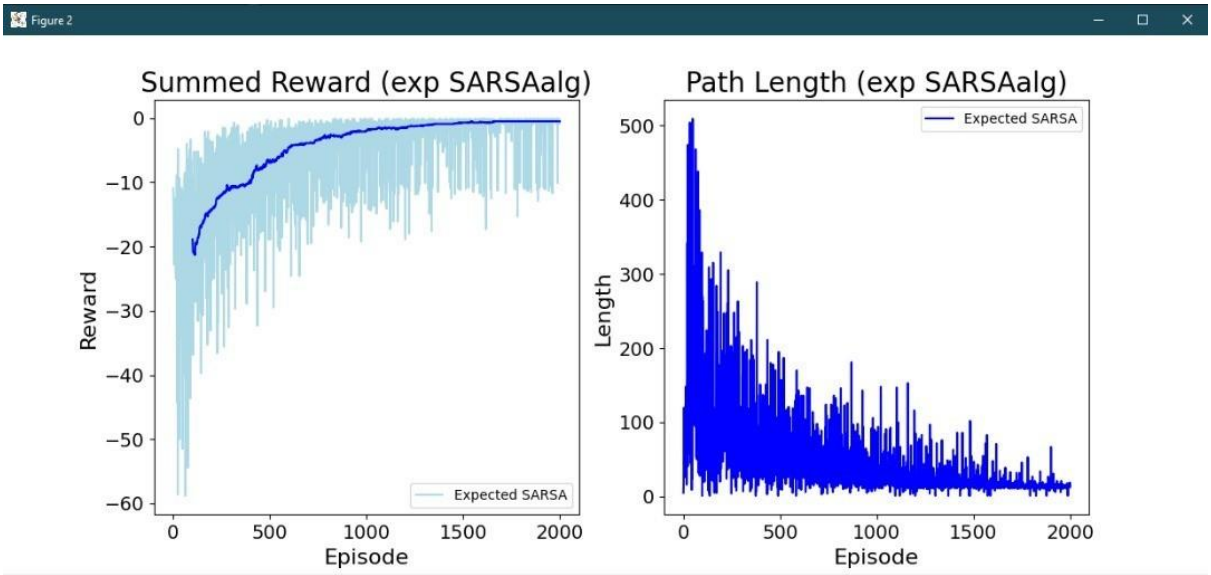
We choose $\varepsilon = 0.1,\ \gamma = 0.1,\ \alpha = 0.8$ as a parameter for this algorithm through the experiences.

3) Expected SARSA

Expected SARSA is a TD control algorithm that can be both on-policy and off-policy. Specifically, the expected SARSA is the conversion of SARSA into off-policy. It takes the expectation of Q-value to choose action so it shows some amount of deterministic similarly to Q-Learning while SARSA shows stochastic action. Because it has deterministic action, it has less variance in action than SARSA so the update is stable. However, the number of computations increases since it has to update Q-values for all actions when it updates the next state.

Value function :

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \Sigma_{a'} \pi(a'|S_{t+1}) Q(S_{t+1}, a') - Q(S_t, A_t)]$$



SARSA updates the behavioral value function calculated in accordance with the current policy and Q-learning updates the maximum behavioral value function regardless of the policy. However, expected SARSA takes a different way. In its update, the target is changed to the expected value (the average of the next behavioral value functions) in the next state-action pair.

It costs more and is slower than SARSA but it can update policy more stable than SARSA since it consider all actions. Also, it can use a bigger step size because the influence from policies that are likely to be formed in the wrong direction in the early stages is relatively less than SARSA which means that it can be easily used even in environments with limited experience. Compared to SARSA, it shows higher performance as it reaches the terminal faster than SARSA.

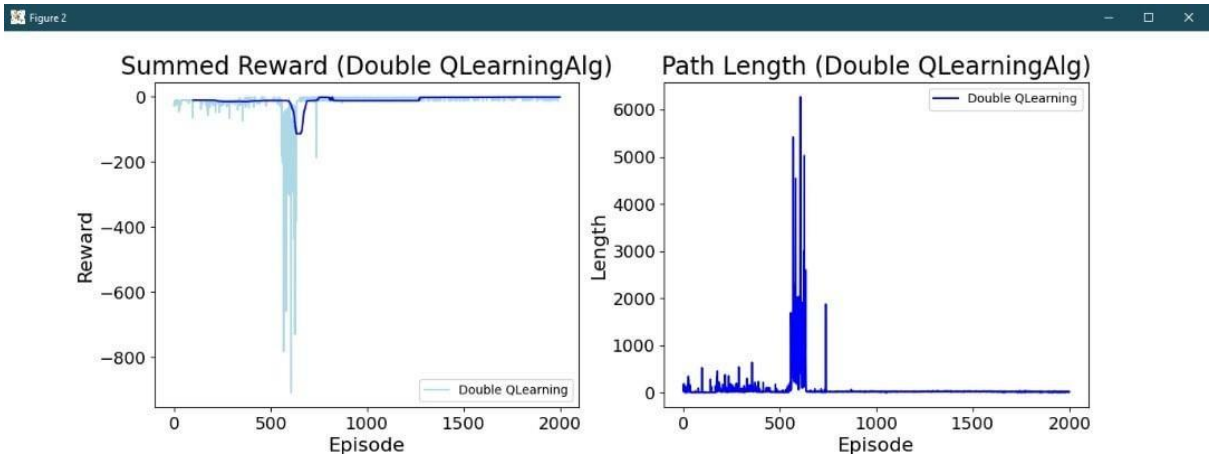
We choose $\varepsilon = 0.1$, $\gamma = 0.1$, $\alpha = 0.01$ as a parameter for this algorithm which is the same as SARSA through the experience. We lower the learning rate so it can improve its performance.

4) Double Q-Learning

Double Q-Learning is based of Q-Learning but instead of training on 1 action-value function, we train on 2 functions instead. What happens is on each step of the episode, we independently update one of the two functions, based on a random number generator. We use the future state of the other value function to update the value of the current state. We continue to use an epsilon greedy policy when running this.

$$Q_1(S, A) \leftarrow Q_{1(S,A)} + \alpha\left[R + \gamma Q_2(S', \underset{a}{\operatorname{argmax}} Q_1(S', a)) - Q_1(S, A)\right]$$

$$Q_2(S, A) \leftarrow Q_{2(S,A)} + \alpha\left[R + \gamma Q_1(S', \underset{a}{\operatorname{argmax}} Q_2(S', a)) - Q_2(S, A)\right]$$



We choose$\varepsilon = 0.1$, $\gamma = 0.1$, $\alpha = 0.8$ as a parameter for this algorithm which is the same as Q-Learning through the experiences.

It costs more than Q-Learning due to needing to store two action-value functions. However, it is much faster as it can use both action-values to find the value of the current state. Compared to the graph for Q-Learning, you can see that the function reaches the terminal state very quickly.

As we can see from the graphs above, Double Q-Learning shows the highest performance as it reaches the terminal state very quickly. After that, it showed good performance in the order of Q-Learning, Expected SARSA, SARSA. Since Double Q-Learning and Q-Learning is off-policy algorithm and can use past data, it can compute faster than SARSA. Expected SARSA showed better performance than SARSA since it has lower variance than SARSA.