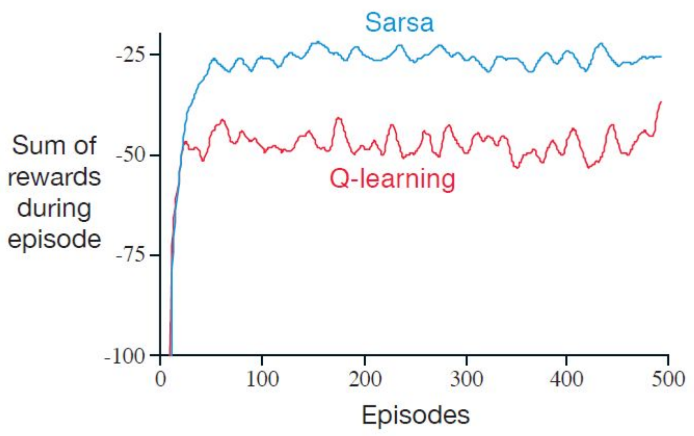
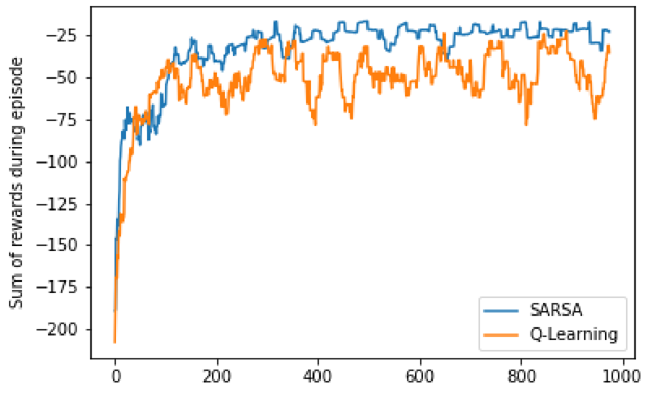
CS9670 – Assignment 3

Mathias Babin

The code used to generate each of the following answers can be found at: https://github.com/mbabin2/CS9670-Assignment-3

1a. Recreate the figure below in the text using the 'CliffWalking-v0' task in OpenGym.



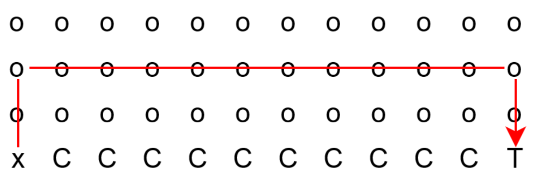


We trained both Sarsa and Q-Learning using the follow hyperparameters:

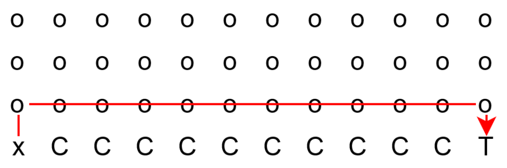
Num. episodes	1000
alpha	0.1
epsilon	0.1
gamma	1

As expected, Sarsa outperforms Q-Learning on this task due to the risk of restarting when taking the ‘optimal’ route when ε > 0. Below are the routes taken by both agents after training for 1000 episodes, for these tests we set ε = 0.

Sarsa:

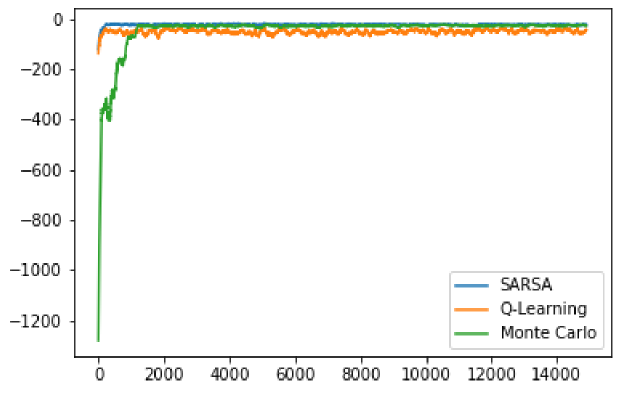


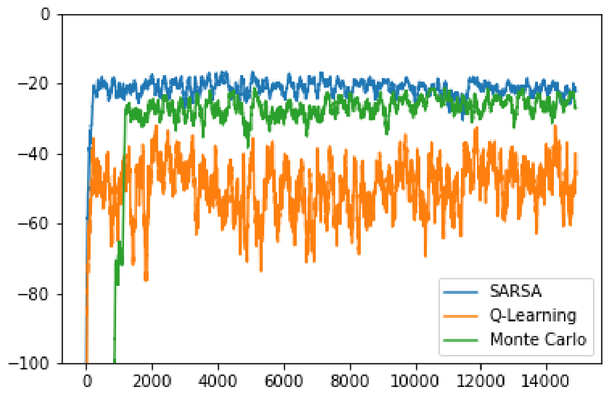
Q-Learning:



We can see that Sarsa does in fact learn to take a suboptimal path to avoid falling off the cliff, while Q-Learning does not due to it’s target function of *R+ γmaxQ(s’,a)* fixating on updates towards the best action found in the next state *s’*: it does not learn from falling off like Sarsa does.

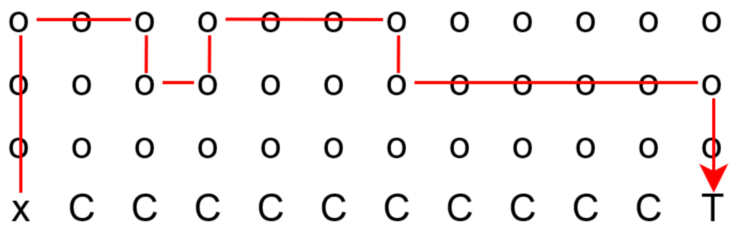
1b. Add on-policy Monte Carlo. How does it compare?





For this experiment, we kept the same hyperparameters as last time, but increase the number of episodes to 15, 000 as MC appears to be less efficient than TD methods. From these results we can see that our implementation of Every-Visit Monte Carlo (MC) outperforms Q-Learning during training for the same reasons that Sarsa did, i.e., it is taking a less risky route to the goal. What's interesting is that it appears to be performing suboptimally when compared to Sarsa with an average performance of around -20. An example of this can be seen from the path this agent takes during our test run which makes 2 suboptimal decisions which takes it to the top row:

Monte Carlo:



Perhaps given more episodes, we would see MC converge to match Sarsa's performance by taking less of these suboptimal actions.

2a. Implement one of the following to learn the Frozen Lake task in OpenGym.

For this task, we decided to implement Double Q-Learning and trained it with the following hyperparameters:

Num. episodes	15000
alpha	0.1
epsilon	1
gamma	0.99

As an off-policy learner Q-Learning (and Double Q-Learning) should be able to learn taking purely random actions. For many environments, 100% random actions may not be sufficient to explore the complete state space, however for this task, it is. After training, we test the agent with an ε = 0, for 1000 episodes and average the results. We find the agent is capable of solving the environment approximately 75% of the time, which should be fairly close to the optimal amount since the environment specifies a 20% chance for the agent to move in a different direction than specified.

2b. What is the reward function for the Frozen Lake task? How does this affect your agent's ability to learn? How might you make things easier for your agent? Modify the reward your agent receives from its actions. Can you achieve better performance?

The reward function for this task is extremely sparse with a 1 rewarded for the agent reaching the Goal state *G* and 0 otherwise. To make training easier on our agent, we specify a new reward function:

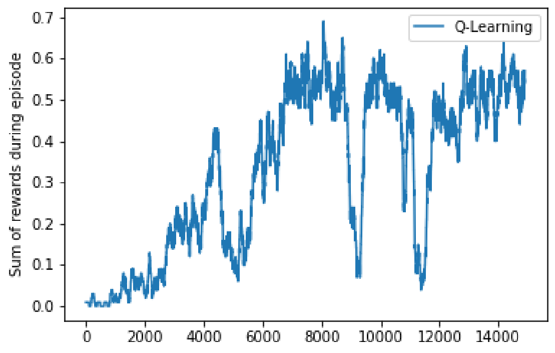
State	Reward
S	-0.1
F	-0.1
H	-10
G	10

Our new reward function penalizes the agent with a reward of -0.1 at each step, -10 for falling through the ice, and +10 for reaching the goal. Using this reward function we achieve a slightly worse average performance from our 1000 test runs when compared to our first attempt in part ***a*** with an average success rate of approximately 73%. However, using this reward function did allow us to reduce our training time by 50% from 15,000 episodes to only 7,500 episodes.

2c. Now let's achieve better performance without cheating. Frozen Lake is about exploration. How could you promote better exploration? What's the best performance you think you can get, and why?

To solve this environment without modifying the default reward function, we modify our Double Q-Learning algorithm used in part 2a to now decay ε from 1 down to 0.05 over the course of training; however, in order to ensure that the agent is still properly exploring the environment, we only begin decaying ε once the agent has reached the goal state at least once.

Results from training using this method are shown on the right. While we do train this method for the same duration as our first attempt (15,000 episodes), it is clear that training could have been halted at around the 7500 episode mark as there appears to be no improvement past that point. Coincidentally, this is the same number of episodes needed to train using our modified reward function. From these results, we can conclude that decaying ε did in fact help our agent learn to solve this environment in a more timely manner.



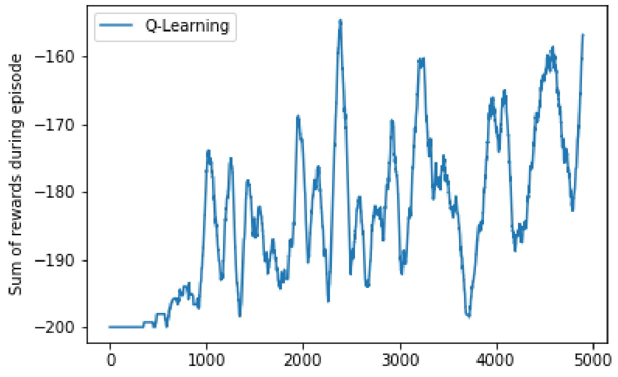
3. Implement tabular Q-Learning to solve the Mountain Car task in OpenGym.

Like with the Frozen Lake task, our solution the Mountain Car task uses Q-Learning (not Double Q-Learning this time) with a decaying ε, the hyperparameters we use are:

Num. episodes	5000
alpha	0.1
epsilon	0.9
gamma	0.99

Like before, we decay ε to a minimum of 0.05 starting once the agent has reached the goal at least once. In our initial experiments we found that completely random actions were not capable of reaching the goal, so to account for this we start ε at a value of 0.9 instead of taking completely random actions.

In order to use Tabular methods with this task, the state-space must be discretized. The simplest approach to this is to simply round each of the state's features to integers. The only issue with this environment is that most of these values fall below 1, resulting in a majority of states rounding to 0. To resolve this issue we simply multiply our states' values by 10 and 100 respectively such that the values of our state our now have at least 1 digit before the decimal point. Results for training are found below:



Once training is complete, we again test our agent on 1000 levels and found that our agents could solve the environment with the following results:

Avg. Reward	-152.7
Min. Reward	-171.0
Max. Reward	-140.0
% Success	100%

Discussion

I learned a fair bit during this assignment, I was mostly surprised with the amount of variance with each of these training methods. As an example, my solution for both Frozen Lake, and Mountain Car can have dramatically different outcomes for each run even when hyperparameters remain unchanged. This led to some difficulties during the implementation of these algorithms as I was left unsure whether the agents were not learning due to programming errors, or whether I had to slightly adjust a parameter like learning-rate or ε. I was also surprised how long it took me to get Monte Carlo to train considering how quickly I got both Sarsa and Q-Learning to work. It wasn't until I drastically increased the amount of training episodes that I saw any positive results out of MC.

Finally, I was surprised to see how effective simply rounding the state-space of Mountain Car is. In the past I've used K-means to cluster the state-space, but I figured this task was simple enough that rounding may work, and in fact it did. This said, I did notice that on some of my training runs the car will 'waggle' back and forth that the bottom of the slope, this is perhaps due to the fact that the agent can't properly differentiate between some of these states do to too many decimal points getting chopped off. In the future I can experiment with increasing how many of these decimal points are conserved (drastically increasing the number of states I have by an order of at least 10), and increasing the number of training of episodes that I use.