ECE 493 Assignment 3 Report

Akshay Patel | Student ID: 20732500

Unless otherwise noted, all figures and algorithm explanations are from the second edition Reinforcement Learning textbook used for this course, written by Richard S. Sutton and Andrew G. Barto.

Domain Description

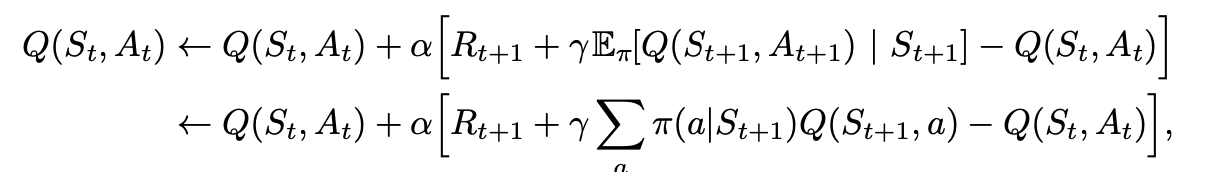
Maze World is the deterministic environment that was used for this assignment. States, actions, and dynamics are defined as follows. States are defined as the agent being in a specific cell of the maze, and they are represented by four coordinates. Tasks can have walls that the agent cannot pass through, pits that are terminal states where the agent receives a large negative reward, and all tasks have a goal state for which the agent receives a positive reward. The action space in this environment consists of {up, down, right, left} represented by integers {0, 1, 2, 3} respectively. The dynamics of this environment are that given a state *s*, and an action *a*, taking action *a* from a state *s* will result in the agent being in a state *s’*. However, if the action takes the agent into a wall or takes the agent out of the grid, then the state will remain the same. If the agent reaches the goal or runs into a pit, then the game ends as those are terminal states. The reward function is that reaching the goal gives you a reward of +1 and ends the episode, running into a pit yields a penalty of -10 and ends the episode, running into a wall will give you a penalty of -0.3, and apart from these, the agent will receive a -0.1 reward for reaching any other cell in the grid.

Maze World is deterministic, meaning that if the agent takes a specific action from a specific state, it will always go to the same next state as well as receive the same reward value. The objective of this environment given this reward function is to reach the goal state as quickly as possible to maximize the reward.

Algorithm Explanations

Expected Sarsa

Expected Sarsa is a learning algorithm that is similar to Q-Learning except for the fact that instead of taking the maximum over the next state-action pairs, Expected Sarsa uses the expected value which is calculated by taking into account how probable each action is given the current policy. Expected Sarsa has the same pseudocode as Q-Learning, except for the update rule which is changed, and given below (from the RL textbook page 133).



Given the next state, Expected Sarsa moves *deterministically* in the same direction as Sarsa moves *in expectation*, which is why this algorithm is called Expected Sarsa. Expected Sarsa is clearly more complex and uses more computational resources than Sarsa given the difference in update rules. Expected Sarsa can also be described as being more complex and more computationally expensive than Q-Learning as it requires more floating-point operations in the expectation calculation. However, in comparison to Sarsa, this extra computational expense results in the elimination of variance due to the random selection of the next action. Given the same amount of learning time, Expected Sarsa should perform slightly better than Sarsa.

What’s even more interesting about Expected Sarsa is that it shows a significant advantage over Sarsa across a wide range of values for alpha, the step-size parameter or learning rate. Expected Sarsa can **safely** set alpha = 1 without suffering from any degradation of asymptotic performance. Sarsa cannot perform very well without the learning rate (alpha) having a small value.

Expected Sarsa can be thought of as subsuming and generalizing Q-Learning while improving over Sarsa with the reduction in variance, albeit at the additional computational cost. Expected Sarsa may perform much better than both of Sarsa and Q-Learning. Also, Expected Sarsa can be both on-policy or off-policy.

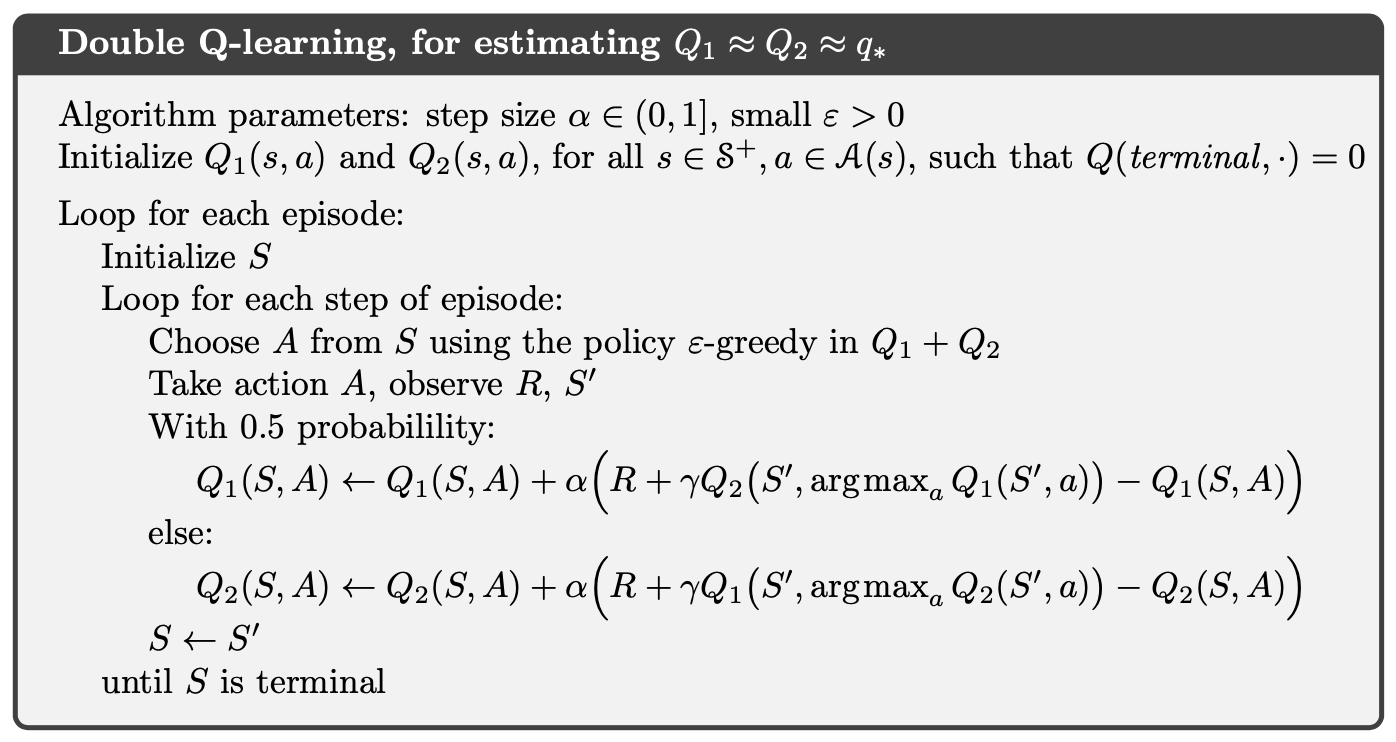
In my implementation, I use essentially the same code as my Q-Learning algorithm except that the learn function is changed to use the update outlined above using expectation of the value from the next state-action pairs. I also use an epsilon-greedy selection mechanism, which over time gradually decreases as I decay epsilon by 1% every episode. By doing so, I can encourage exploration in the beginning episodes, and encourage exploitation as the agent has more experience in the environment. However, in the plots below, I mistakenly did not call the epsilon decay function after every episode, so we don’t see the benefits of this mechanism there.

Double Q-Learning

Double Q-Learning is a double learning variant of Q-Learning that tries to avoid maximization bias (see the RL book for an explanation of maximization bias). Double Q-Learning divides the time steps in two, updating one of two Q-tables depending on a random selection between the two. The two approximate value functions are treated symmetrically, and the behaviour policy can use both action-value estimates from both Q-tables. An epsilon-greedy policy for Double Q-Learning can be based on the sum or average of the two action-value estimates. In my implementation, I use the sum of the two action-value estimates from the two Q-tables. Compared to Q-Learning, Double Q-Learning can eliminate the harm caused by maximization bias leading to better learning.

Double Q-Learning clearly requires more computational space as it requires the storing of two Q-tables and the summing of all corresponding Q-table values during the choice of an action. In an environment with a large state-space and a large action-space, the necessity to store two entire Q-tables can prove to be much more computationally expensive than Q-Learning or Sarsa.

Complete pseudocode for the Double Q-Learning algorithm is presented below, upon which my implementation is based. I use an epsilon-greedy selection mechanism, which over time gradually decreases as I decay epsilon by 1% every episode. By doing so, I can encourage exploration in the beginning episodes, and encourage exploitation as the agent has more experience in the environment. However, in the plots below, I mistakenly did not call the epsilon decay function after every episode, so we don’t see the benefits of this mechanism there.



Sarsa(Lambda) – Eligibility Traces

Eligibility traces are one of the basic mechanisms of reinforcement learning that almost any TD method, including Sarsa or Q-Learning, can be combined with to produce a more general algorithm that might learn more efficiently. They unify and generalize TD and Monte Carlo methods. When TD methods are combined with eligibility traces, they produce a set of methods spanning a spectrum with Monte Carlo methods at one end, when lambda = 1, and one-step TD methods at the other end, when lambda = 0. In the middle are intermediate methods that can often be better than either extreme method when lambda is 0 or 1. Eligibility traces also offer an elegant, algorithmic mechanism with significant computational advantages through a short-term memory vector, the eligibility trace. This is better than n-step methods in terms of computational expense because eligibility traces only require a single trace vector rather than a store of the last n feature vectors. The trace decay factor, lambda, which is in [0,1] interval, determines the rate at which the trace falls. See the RL book for a more thorough and in-depth explanation of the different kinds of eligibility traces that exist.