CS6700 Final Project Report

Chella Thiyagarajan N and Tarun Prasad
ME17B179 and ME17B114

Algorithm:

We have chosen the Q-Learning algorithm with state space discretization. In our RL CS6700 we have learnt two prominent algorithms which are Q-Learning and Policy Gradient algorithms. We decided to go forward and implement these algorithms as we were already fully familiar with the concepts and if we find any anomaly while evaluating the results we would be able to explain them.

Q-learning is a *values-based* learning algorithm. Value based algorithms update the value function based on an equation (particularly Bellman equation). Whereas the other type, *policy-based* estimates the value function with a greedy policy obtained from the last policy improvement. Q-learning is an *off-policy learner*. Means it learns the value of the optimal policy independently of the agent's actions. On the other hand, an *on-policy learner* learns the value of the policy being carried out by the agent, including the exploration steps and it will find a policy that is optimal, taking into account the exploration inherent in the policy.

After implementing both the algorithms, we found that both Q-Learning and Policy Gradient tend to converge over a long period of time, say 10000 to 20000 Episodes, we later found that with the right set of input features and right set of discretization bins Q-Learning has shown promise to converge with a lot less Episodes whereas policy gradient tend to converge over large number of episodes. Due to the short episode constraint of the aicrowd platform evaluator we selected Q-Learning as our go to algorithm.

Q-Learning:

Create a q-table: When q-learning is performed we create what's called a *q-table* or matrix that follows the shape of [state, action] and we initialize our values to zero. We then update and store our *q-values* after an episode. This q-table becomes a reference table for our agent to select the best action based on the q-value.

Action Exploration: An agent interacts with the environment in 1 of 2 ways. The first is to use the q-table as a reference and view all possible actions for a given state. The agent then selects the action based on the max value of those actions. This is known as *exploiting* since we use the information we have available to us to make a decision.
The second way to take action is to act randomly. This is called *exploring*. Instead of selecting actions based on the max future reward we select an action at random. Acting randomly is important because it allows the agent to explore and discover new states that otherwise may not be selected during the exploitation process. You can balance exploration/exploitation using epsilon (ε) and setting the value of how often you want to explore vs exploit.
<u>Pseudo-Code:</u> Fix a value for such that > 0. Usually, it is a small number. Pick a random number r such that r ∈ [0, 1]. If r < , then pick a random action from the action space with uniform probability. If not, pick the greedy action. An alternative approach is to make a function of t, say t such that t → 0 as t → 0. This approach is essentially reducing exploration as algorithm updates.

Learning: The updates occur after each step or action and ends when an episode is done. Done in this case means reaching some terminal point by the agent. The agent will not learn much after a single episode, but eventually with enough exploring (steps and episodes) it will converge and learn the optimal q-values or q-star (Q∗).

Here are the 3 basic steps:

1. Agent starts in a state (s1) takes an action (a1) and receives a reward (r1)
2. Agent selects action by referencing Q-table with highest value (max) OR by random (epsilon, ε)
3. Update q-values

We adjust our q-values based on the difference between the discounted new values and the old values. We discount the new values using gamma and we adjust our step size using learning rate (lr).

<u>Q-Value Update equation:</u>

$$Q^{new}(s_t,a_t) \leftarrow \underbrace{Q(s_t,a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\Bigg(\underbrace{\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1},a)}_{\text{estimate of optimal future value}}}_{\text{new value (temporal difference target)}} - \underbrace{Q(s_t,a_t)}_{\text{old value}}\Bigg)}^{\text{temporal difference}}$$

<u>Q-Learning Pseudo Code:</u>

```
Initialize $Q(s,a)$ arbitrarily
Repeat (for each episode):
   Initialize $s$
   Repeat (for each step of episode):
      Choose $a$ from $s$ using policy derived from $Q$
      Take action $a$, observe $r$, $s'$
      Update
        $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$
      $s \leftarrow s'$;
   Until s is terminal
```

States:

To converge the algorithm faster and to get a higher score we have tried a lot, like tuning the hyper-parameters, keeping parameters constant or decaying them, using different functions to decay the parameters, etc., Of all of them, selecting appropriate and targeted states proved to be producing better results. This tells us that feature engineering is a vital part and a strong domain knowledge can only help us to create better features.

Q-Table Approach 1:

Instead of simple linear discretization we first tried to discretize using random samples obtained from the simulator itself. We tried to generate 10000 sample states randomly using random actions from the simulator. We then sorted each state array in ascending order, then we tried to divide the whole array into equally spaced n_bins of arrays; the start and end values of these divided arrays are boundaries. So, now we have n_bins number of boundaries for each state, in order to create a q-table we need to permute all the boundaries between different states, therefore we would have finally discretized the whole state space.

Size of state space discretization = (n_bins)^(number of states)
Size of actions = actions

Size of Q-Table = ((n_bins)^(number of states))*(actions)

Finally we assign random values to all the entries in the q table.

Q-Table Approach 2:
Here we tried to linearly discretize the states. We hard coded the minimum and maximum possible limits of the state space and divided them with equally spaced n_bins of intervals for each state. The start and end values of these divided intervals are denoted as boundaries of a state. So, now we have n_bins number of boundaries for each state, in order to create a q-table we need to permute all the boundaries between different states, therefore we would have finally discretized the whole state space.

Size of state space discretization = (n_bins)^(number of states)
Size of actions = actions
Size of Q-Table = ((n_bins)^(number of states))*(actions)

Finally we assign 0 value to all the entries in the q table.

Result:
We tried the first approach first, we got a smooth learning curve which promised us that the whole state and algorithm framework works, but the curve rises very slowly. It took around 20000 iterations for the cartpole to converge to the maximum score. When we tried the second approach it showed a sudden bump or rise in the rewards and stayed the same till the end of 1000 Episodes. Due to the constraints posed by aicrowd to run the algorithm only upto 1000 iterations we used approach 2 for the cartpole and mountain car, whereas in the catch example we used approach 1. We are absolutely not sure what was the cause of sudden rise in the rewards for the approach 2, we suspect that this may be caused by 2 reasons, one reason is the random generation of states by choosing a random action may have not explored some potential spaces of state space and may have followed a different distribution or the other reason is the random initialization might have caused to assume a wrong prior initially. Overall the reason is too much randomness in approach 1 didn’t give us reliable scores.

CartPole:

This environment implements a version of the classic Cartpole task, where the cart has to counter the movements of the pole to prevent it from falling over.
The observation is a vector representing: (x, x_dot, sin(theta), cos(theta), theta_dot, time_elapsed)
We decided to eliminate the time_elapsed from the state space as we assumed it to be irrelevant, we also tried multiple combinations of the states to decide which set of less number of states would produce the best result. By using the domain knowledge theta and theta_dot would be the most important states to consider as they determine the angle of the inverted pendulum problem. We created a new state theta by extracting the angle from sin(theta) and cos(theta) using arc_tan function.

Mountain Car:
This environment implements a version of the classic Mountain Car problem where an underpowered car must power up a hill.
The observation is a vector representing: (x, x_dot, time_elapsed)
We haven’t done much variation in choosing and modulating states as the given number of observations is itself small, we only tend to change things when the number of states is high because it causes the number of entries in the Q-Table to blow up which makes it difficult for the algorithm to optimize. We just eliminated the time_elapsed feature as we assumed it to be irrelevant and we used both x and x_dot as functions.

Catch:
In this environment , the agent must move a paddle to intercept falling balls. Falling balls only move downwards on the column they are in.

The observation is an array shape (rows, columns), with binary values: 0 if a space is empty; 1 if it contains the paddle or a ball.
By running the environment randomly we observed that there is only one apple that could be falling from the sky at a time-instant, this gave us the inference that there could be only finite number of states and it is small. Rather than hard coding each states we tried approach 1 and tried to get a 10000 random sample of states and tried to extract only unique states from the samples. Luckily, the code captured all the unique states of the environment everytime. The total number of states are 45*5 = 225.

Hyper-Parameter Tuning:

There are fundamentally 4 hyperparameters that require tuning. They are gamma, epsilon, learning rate and number of bins.

Number of Bins: This tells us the level of discretization and the size of the Q-Table to be obtained. Increasing it gives us less score as the table size increases and it's very difficult to optimize, decreasing it also gives us less score as we may not capture all the motion details required to optimize the RL algorithm. Therefore the right value lies somewhere in the middle, so we empirically found out the sweet spot by trying out a lot of values for the number of bins for each environment.

Gamma: Gamma is the value of future reward. It can affect learning quite a bit, and can be a dynamic or static value. If it is equal to one, the agent values future reward JUST AS MUCH as current reward. This means, in ten actions, if an agent does something good this is JUST AS VALUABLE as doing this action directly. So learning doesn't work that well at high gamma values. Conversely, a gamma of zero will cause the agent to only value immediate rewards, which only works with very detailed reward functions. All the learning rates of all the environments lie between 0.9 and 1.

Epsilon: Epsilon tuning the second vital part of hyper parameter tuning next to tuning number of bins. Epsilon controls the exploration and exploitation aspects in the Q-Learning algorithm. Our strategy is to get a good score with constant epsilon. This constant epsilon value is set as the minimum value of the algorithm and the maximum value is varied empirically to get a better score. We have decided on using the linear function to decay the epsilon value. Catch environment didn't require much tuning, constant epsilon gave us very high accuracy upto 0.99 score. We used a very low epsilon of 0.0001 as catch required exploitation rather than exploration. Mountain car too worked under a constant epsilon 0.01, this too requires much exploitation, but we are not sure about that explanation as it provides us a score upto 0.8 max. In the cartpole environment we have used linear decay from 1 to 0.1. Cartpole seemed to work well with the initial exploration strategy.

Learning Rate: Learning rate tells the magnitude of step that is taken towards the solution. It should not be too big a number as it may continuously oscillate around the minima and it should not be too small of a number else it will take a lot of time and iterations to reach the minima.
The reason why decay is advised in learning rate is because initially when we are at a totally random point in solution space we need to take big leaps towards the solution and later when we come close to it, we make small jumps and hence small improvements to finally reach the minima. Linear decay from 1 to 0.1 worked well for the catch and cartpole environment. Learning rate decay strategy somehow worsened the mountain car scores so we resorted to using constant values.