Project report – Reinforcement learning with OpenAI GYM

Table of Contents

The Self-Driving Cab problem

The self-driving cab problem is an example in which reinforcement learning can be applied for the Taxi to start driving itself and learning from rewards based on performing the right actions and learning what not to do when the agent gets penalised for performing the wrong actions. In the Problem scenario, we have a taxi (Agent) that is to interact with its environment in a way that allows it to correctly pick up a passenger while also: correctly

- Dropping them off at the right location
- Save the passengers time by taking the minimum time possible to drop off that Passenger (Quickest route)
- Take care of the passenger’s safety (Avoid hitting walls)

The problem consists of:

- A Starting state for the smart cab
- A State Space which is a set of all the possible situations our smart cab could partake in and contains:
    - A 5x5 grid
    - The 4 locations in which the passenger could be picked up from and dropped off at
    - The passenger itself and the 4 passenger locations equating to 5

So from this, it is clear that our taxi environment (State Space) is consisting of:

$$\frac{the\ grid}{5X5} X \frac{The\ 4\ pick\ and\ drop\ locations}{4} X \frac{Passenger\ agent\ in\ the\ cab\ and\ the\ drop\ off\ locations}{5}$$

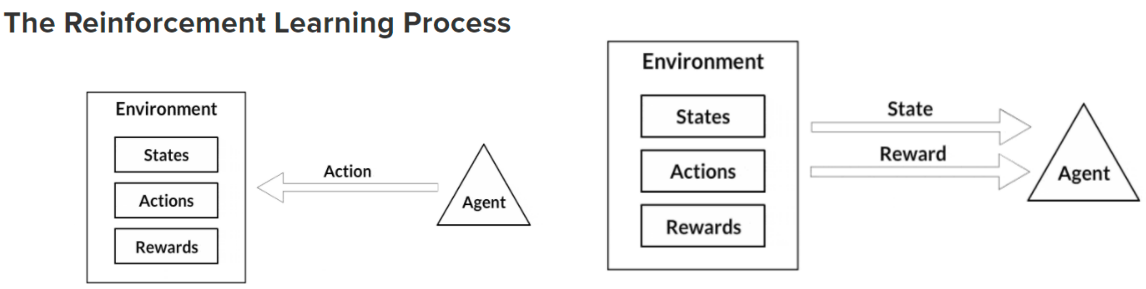
5x5x5x4 = 500 total possible states that could happen in this environment

- An Action Space is the actions that our self-driving cab (Agent) can do, namely:
    - Go North
    - Go South
    - Go East
    - Go West
    - Pick up Passenger
    - Drop off Passenger

- A rewards system in which allows the agent to learn the right thing to do and what the wrong thing to do by bonus points and penalisation points depending on its actions in its current state within the environment (Search Space)

When completing this environment, I will compare the results achieved between using reinforcement learning and without reinforcement learning further down.

Using reinforcement learning to create a self-driving agent



What is Reinforcement learning

Reinforcement learning is the process in which we train an agent to make the optimal choice from learning through past experiences when presented in the same state within the Space state environment. It would've learnt from its previous decisions within the action space and the rewards that are given from that choice which action would be the optimal decision to make.

When we consider to create/implement a Reinforcement learning algorithm for the problem we desire to solve, we need to account for the rewards, states and actions within the state space.

The process of creating an agent that is capable of learning from itself involves the following:

- The agent must observe the environment.
- Use an action within its action space determined via a strategy (possibly random choice).
- Receive a reward or penalty from the action it chose to take.
- Learn from the actions and refine the strategy to fit one with the highest reward and lowest penalisations.
- Iterate until the best strategy for the situation within the environment state space has been found with relation to the action space possible for the agent, we are training.

Implementing Q-learning

Q-learning uses the agent's rewards generated via the agent's action in the state space. Since there is a reward for every move possible from a given state in the state space, this reward, if more beneficial than the others generated, is saved into a Q-table as a Q-value indicating that the action was the most beneficial for that given state. Through every iteration of the environment, the Q-table and Q-values will be updated as a (state, action) for the given state. This is the best action with more and more iterations the computer will start to find the best action for every move by updating the Q-value in the Q-tables and recalling them when in that position later on.

The Mathematics within Q-Learning

$$Q(state, action) \leftarrow (1-\alpha)Q(state, action) + \alpha\Big(reward + \gamma \max_{a} Q(next\ state, all\ actions)\Big)$$

The maths behind Q-Learning is:

```
new_value = (1 - alpha) * old_value + alpha * (reward + gamma * next_max)
    q_table[state, action] = new_value
```

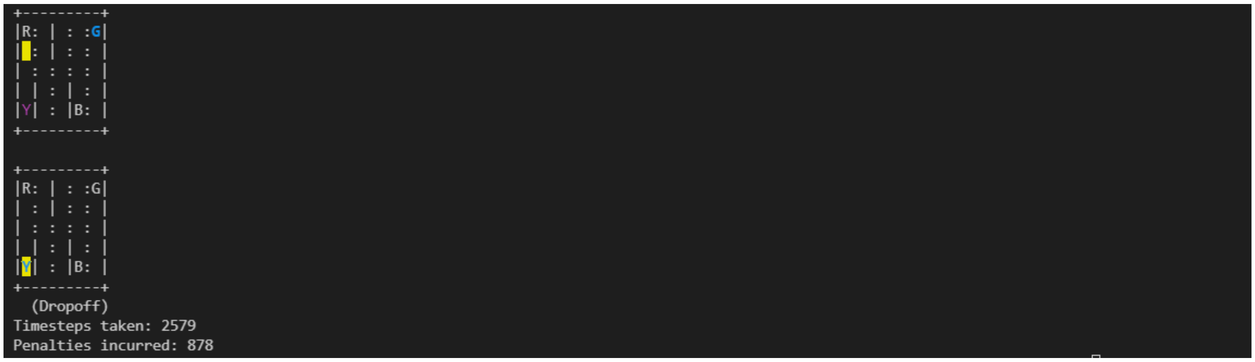
we are determining the new q-value of the agent's current state in the state space and action by taking the learning rate of the original q-value and then adding the learning rate (alpha) of the reward and discount factor(gamma) of the next state by all actions and updating the q-table with the action for that state that gave the highest reward from the old state. This is the math behind Q-Learning and how to solve the Self-driving cab.

In the case of not having a q-value for a given state due to not iterating through enough of the state space to generate one for that state, we use the Epsilon as a number to compare against the q-value to allow for exploration to complete the requirements for the given smart cab environment which will prevent the smart cab from getting stuck due to its allowance to be explored.

Findings from my self-driving cab solutions

I have created 3 files, one that brute forces the result, one that uses a complete Q-Table to get the final result, and one that uses the Q-table and Epsilon to allow the agent to complete the environment without a full Q-table and Q-values for all given positions.

Random policy solution (1 completed round):



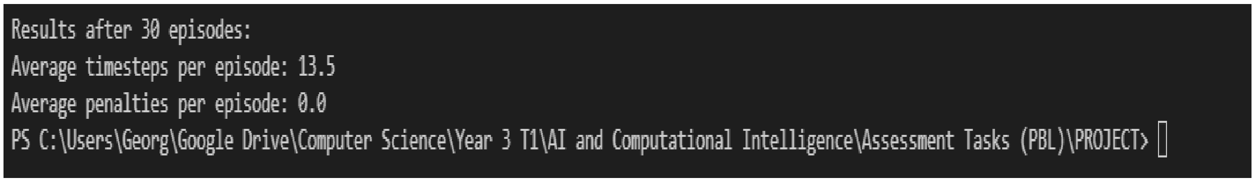
```
+---------+
|R: | : :G|
| : | : : |
| : : : : |
| | : | : |
|Y| : |B: |
+---------+

+---------+
|R: | : :G|
| : | : : |
| : : : : |
| | : | : |
|Y| : |B: |
+---------+
  (Dropoff)
Timesteps taken: 2579
Penalties incurred: 878
```

Timesteps taken: 2579

Penalties incurred: 878

Q-Learning solution with full Q-Table knowledge (30 completed rounds)



```
Results after 30 episodes:
Average timesteps per episode: 13.5
Average penalties per episode: 0.0
PS C:\Users\Georg\Google Drive\Computer Science\Year 3 T1\AI and Computational Intelligence\Assessment Tasks (PBL)\PROJECT>
```

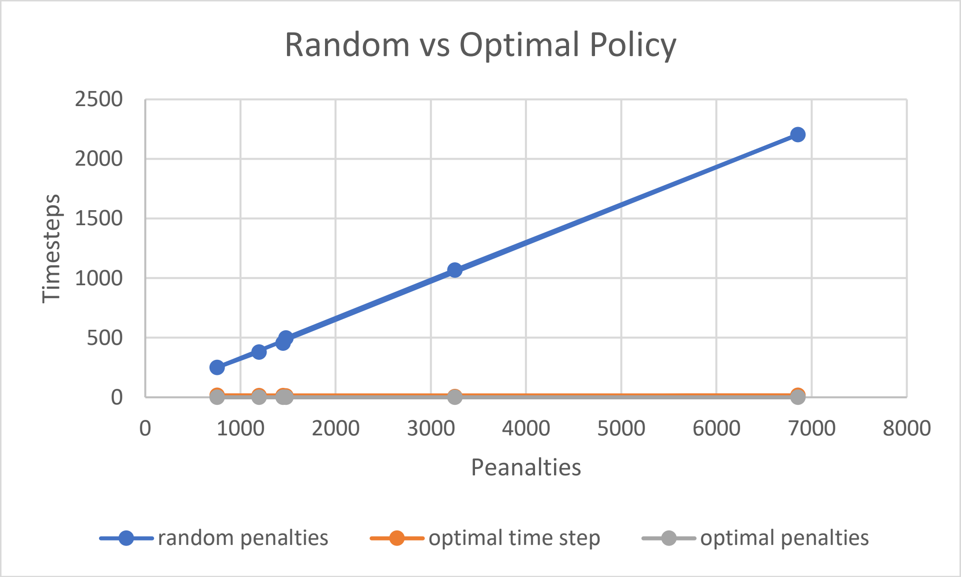
Results after 30 episodes:

Average timesteps per episode: 13.5

Average penalties per episode: 0.0

The Results between random policy and the optimal policy

The results clearly show that the random policy is extremely ineffective to be able to need 2579 steps incurring 878 penalties while the optimised q-learning policy implementing a q-table without the use of Epsilon (exploring) is 100% accurate in the fact that it incurs no penalties and has an average of 13.5 timesteps to be taken per episode showing just how much of an increase in efficiency it has when implemented into the problem as a solution.



The Cart Pole Problem

The cart pole problem is one that requires a cart to move in a direction in order to keep a pole upwards, and these directions are left and right and via force, i.e. (1 or -1). The environment consists of 4 observations at any given state within the state space. The starting point of the cart, the angle of the pole, move cart right or move cart left.

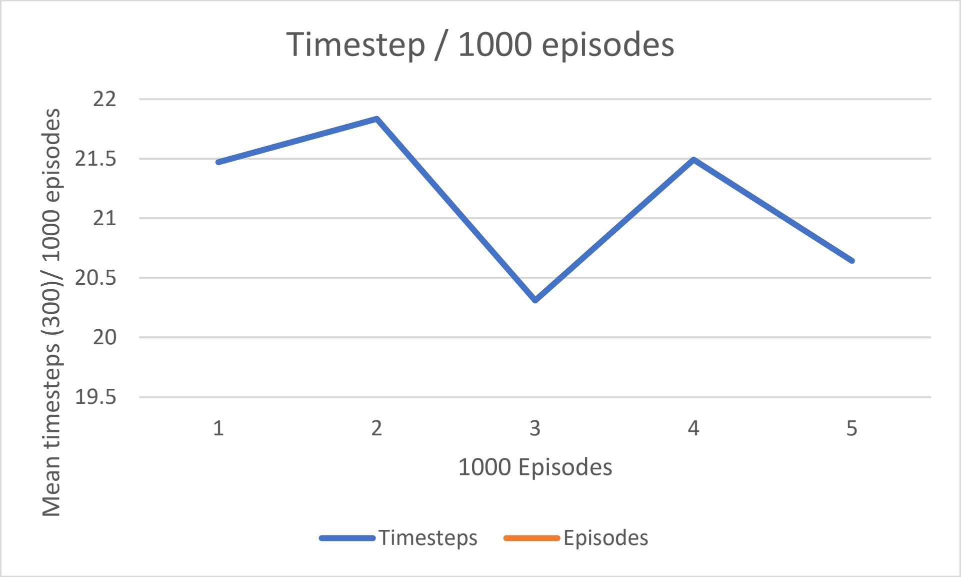
The Cartpole is different to the self-driving cab problem in a variety of ways. The smart cab consisted of penalties for its mistakes and rewards for its correct decisions which were saved to a q table, the self-driving cab problem also involved picking up a passenger and dropping them at the correct location taking the shortest correct path possible. While the penalties for dropping off and picking up a passenger at the wrong location was -10, while a -1 for every move, there was a +20 reward for when a correct drop off was achieved. In the Cartpole, there is not really a penalty for the Cartpole but rather a reward for every timestep that the pole remains vertical for the brute force, but also a mean reward for the Q-Learning algorithm implemented in the Q-Learning variant as of the Cartpole problem, this means reward was generated using the rewards generated from the episode and introduced with a decaying epsilon if the episode reward was greater than the one in the previous. This allows for a better learning algorithm with alpha (learning rate) gamma (discount factor) and the Epsilon to formulate a better Q-table for the Cartpole problem.

The Results between random policy and the Q-Learning (optimal) policy

The results for the random policy and Q-Learning policy are described below and compared after a thorough explanation of the results.

Random Policy

The random policy worked differently to the Q-Learning algorithm and proved to behave in a different way. Once the random policy was able to generate 300 reward points which consisted of 300 timesteps, it would then break and determine how many episodic iterations it would take to hold the pole up on average for 200 timesteps. This was calculated using the amount of time the pole was able to be held up for 300 timesteps / 1000 episodes.



From the chart, it is clear that although it didn't work with perfect accuracy, it was able to complete around 20 successful pole holds per 1000 episodes using a random policy.

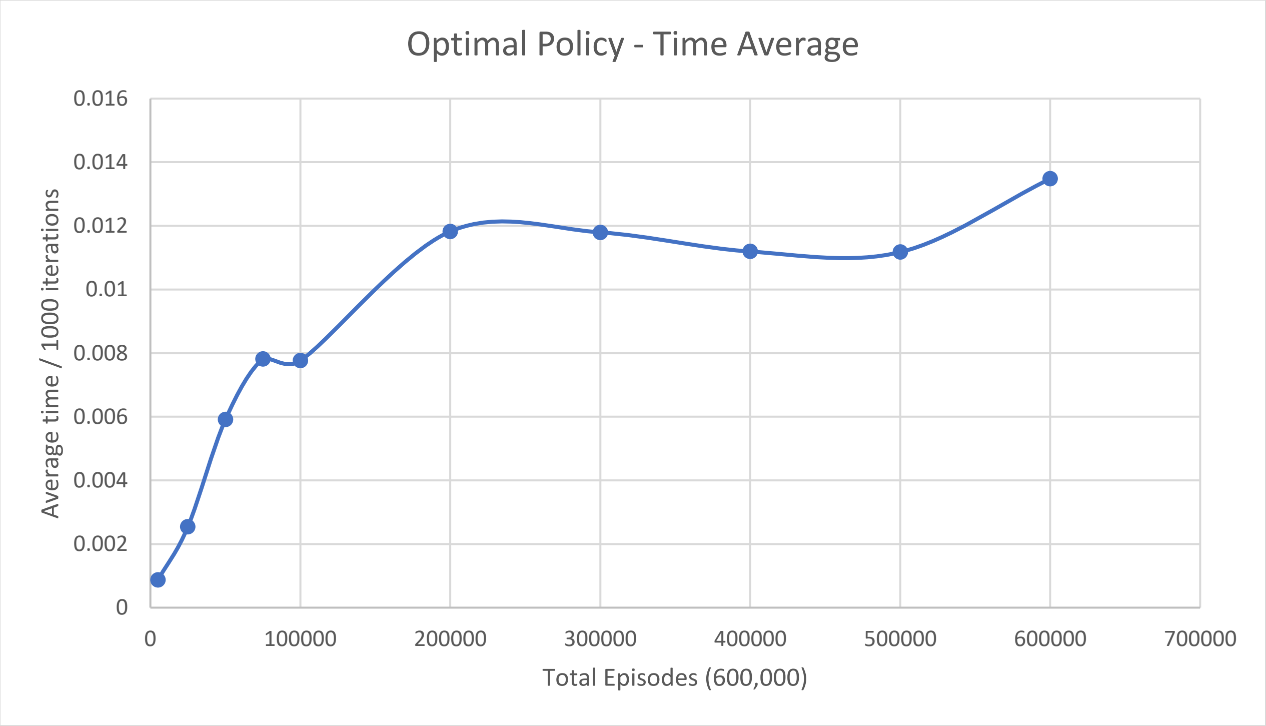
Optimal Policy

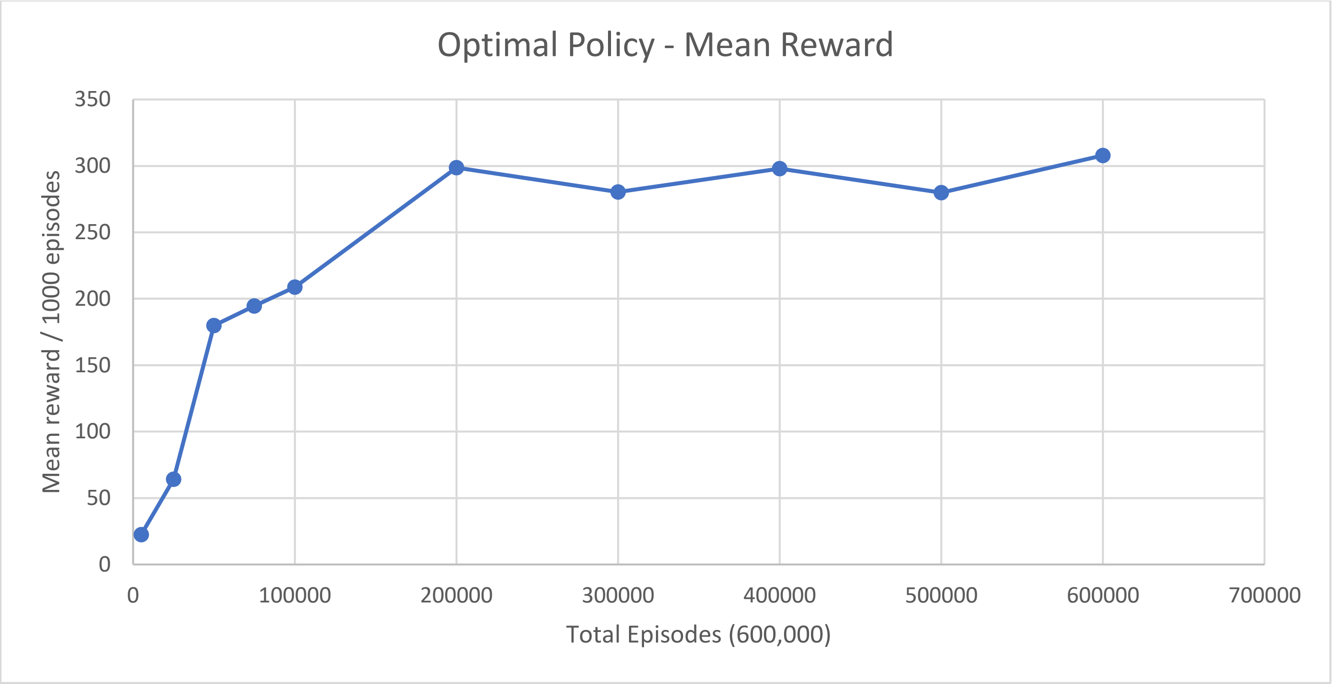
For the optimal policy, I used the Q-learning algorithm with a total of 600,000 iterative episodes to check the progress from iteration 1 to the final iteration. The success of this algorithm was deemed by the average reward generated and the time that it was able to hold up the pole on average for any given 1000 iteration/episodes. As the iterations increased along with the updated q-table and decaying Epsilon, I was able to generate an algorithm that was successfully learning to balance the pole longer as the iterations and q-table as the iterations were updated nearing the 600,000 episodes.

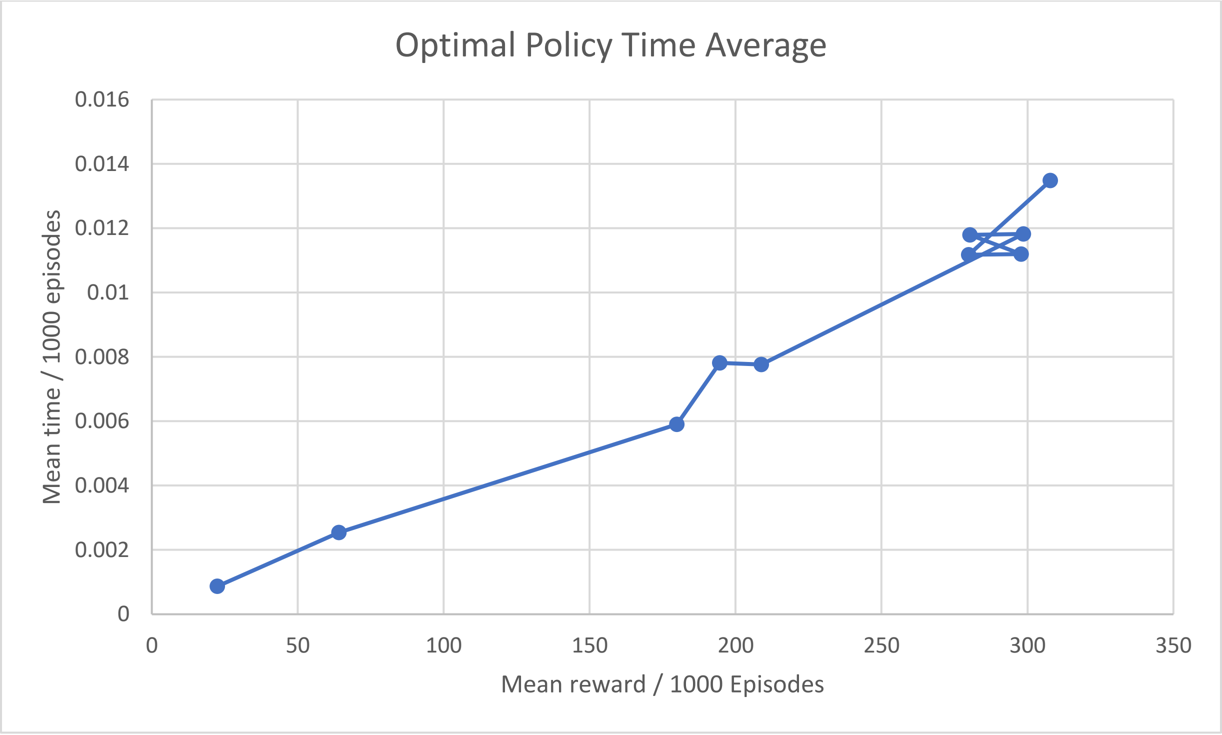
From the graphs below and the data collected, it is clear that as the algorithm worked towards the final iteration, there is a clear positive correlation between the episodic iterations against an increase of average time the pole was held up with an increasing mean reward showing that the algorithm was in fact working towards a successful Q-Learning implementation and table of successfully being able to hold the pole up longer each iteration through:

Optimal Policy

Episode	Time Average	Mean Reward
5000	0.000867717	22.443
25000	0.002542288	64.156
50000	0.005908398	179.858
75000	0.007812362	194.536
100000	0.007761622	208.871
200000	0.011824455	298.682
300000	0.011795043	280.29
400000	0.011192776	297.849
500000	0.011174463	279.879
600000	0.013480573	307.883







When we compare the random policy and the Q-learning policy it is clear that the random policy was able to generate successful holds of the pole however the Q-Learning algorithm was working towards being able to hold the pole up longer, if iterations for the Q-learning were put to around a few million and I had no time constraints I do not see a reason for the pole not being able to hold longer then 300 timesteps consistently as the Q-learning was developing more meaningful rewards and holding the pole up longer through every iteration through the environment and q-table update with q-table values.

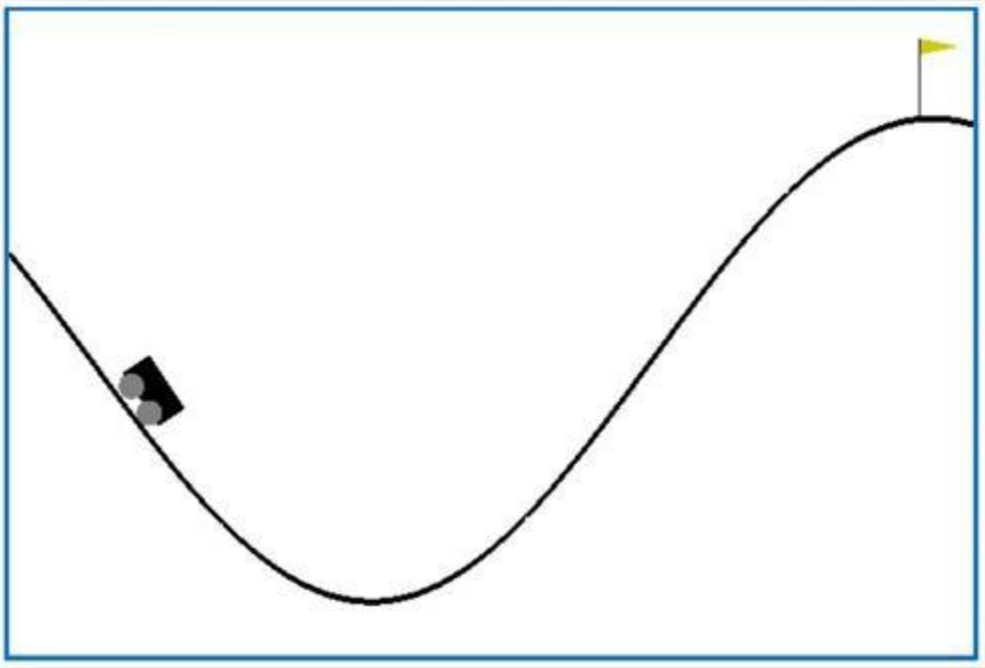
Although the implementation of the random policy was not a bad result, it did not save the values in which it was able to hold up the pole for that duration of timesteps (300), if an implementation of random policy and Q-table/Q-values Q-learning algorithm had been implemented to include both the successful random results for the given state to the q-table and then again for larger timesteps as the random policy successfully fills the gap for the q-table past the previous successful time step then I can see a very efficient algorithm forming that would be able to hold up the cartpole for a much larger period of timesteps in a much more efficient manner requiring less episodic iterations to get desired the effective and quality results from and to input into the q-table.

The Mountain Car Problem

What is it?

The mountain car problem is one that consists of a car that sits in the middle of two mountains, the objective of this environment and the goal is to solve a way in which the car can reach the flag pole at the top however its engine is not strong enough to drive up the mountain in one go and must therefore build momentum by sliding back and forth between the 2 mountains to generate enough momentum that it can reach the flag pole by scaling the hill successfully.

The mountain car is the agent, this agent has a vector which includes its horizontal position and velocity for any given state within the agent's state space.



Solving the Mountain Car problem

During every render of the environment the car starting at the bottom of the valleys at position -0.5, the episode ends after the car reaches the flagpole or after 200 moves/timesteps. At each timestep/move the mountain cars action space has 3 choices for that state, it can either do nothing, or be pushed left or be pushed right via a force. For every move that is taken of the 200 moves a penalty of -1 is incurred to its total reward. A max of -200 for a total reward is given for every unsuccessfully completed render.

To solve this environment, we need to set up the agent to be able to explore the environment with regards to the agent's state space using the agents action space.

Mountain Car Mathematical model

The state space is a box with 2 float values (2-dimensional box) and the action space consists of 3 actions (discrete values). These actions are represented as 0,1 and 2.

The 2-dimensional box representing the state vector can take in:

First float value range: (-1.2 and 0.6) [CART POSITION}

Second float value range: (-0.07 and 0.07) [CART VELOCITY]

[-1.2 , -0.07] = cart positions and velocity min values

[0.6 , 0.07] = cart position and velocity max values

Since we are dealing with a continuous state space this results in there being an infinite state-action pairs making convergence impossible to happen for every state-action pairs in the state space.

To create divergence, we are going to round the first state value to the nearest 0.1 value and for the second float value to the nearest 0.01, we will then multiply the cart position value by 10 and multiply the cart velocity value by 100. This way we can now create divergence since we have no mathematically reduced the number of state-action pairs for the agent to 855 which allows for the Q-learning algorithm to converge.

Since we are dealing with a 2-dimensional state space we will need to replace the initial Q (state, action) with Q (state1, state2, action).

The Q-Learning algorithm will become this:

$$Q'(s1, s2, a) = (1 - alpha)*Q(s1, s2, a) + alpha*(reward+gamma*Q(s1', s2', argmax\ a'\ Q(s1', s2', a')))$$

What this is saying is that we initialise the state 1 and 2 and action to small random values, observe the initial 2 states, based on the Epsilon choose an action, from that action we analyse the resulting reward and the new state of the environment for that given 2-dimensional state space and update the Q-table with the Q-table values for the initial position depending on the rule we use to update, (alpha, gamma, Epsilon)

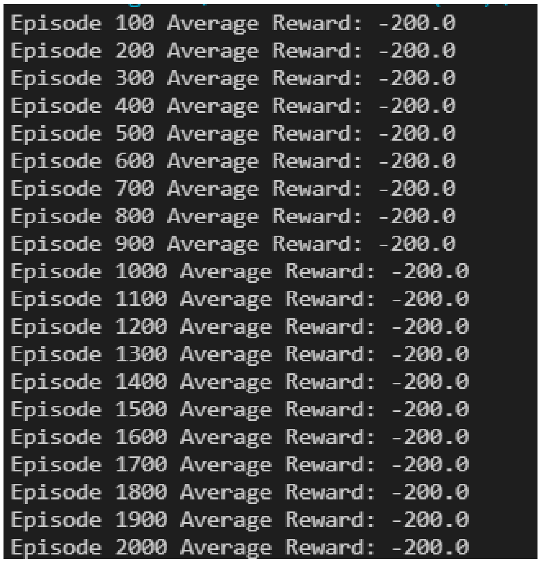
We will repeat this process until convergence within the Q-table from the updated Q-tables happens enough.

After enough episodes we will use a decaying epsilon for a specified number of episodes to change the strategy of the algorithm from exploring into exploiting the current q-table to get more value from our implemented algorithm and policy.

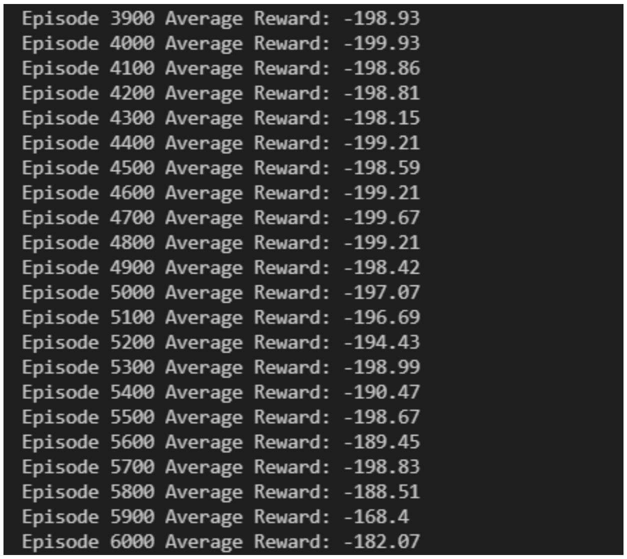
This means that until the end of the algorithm the agent will be exploring all the possible state-actions possibilities, creating convergence and then using the knowledge gained to fulfil the goal required (Reaching the top of the mountain).

Results

Due to the exploration aspect of the algorithm, we can see that there is no reduction in penalties until the last 2 thousand episodes, this is a part of the decaying Epsilon that has been implemented into the algorithm.

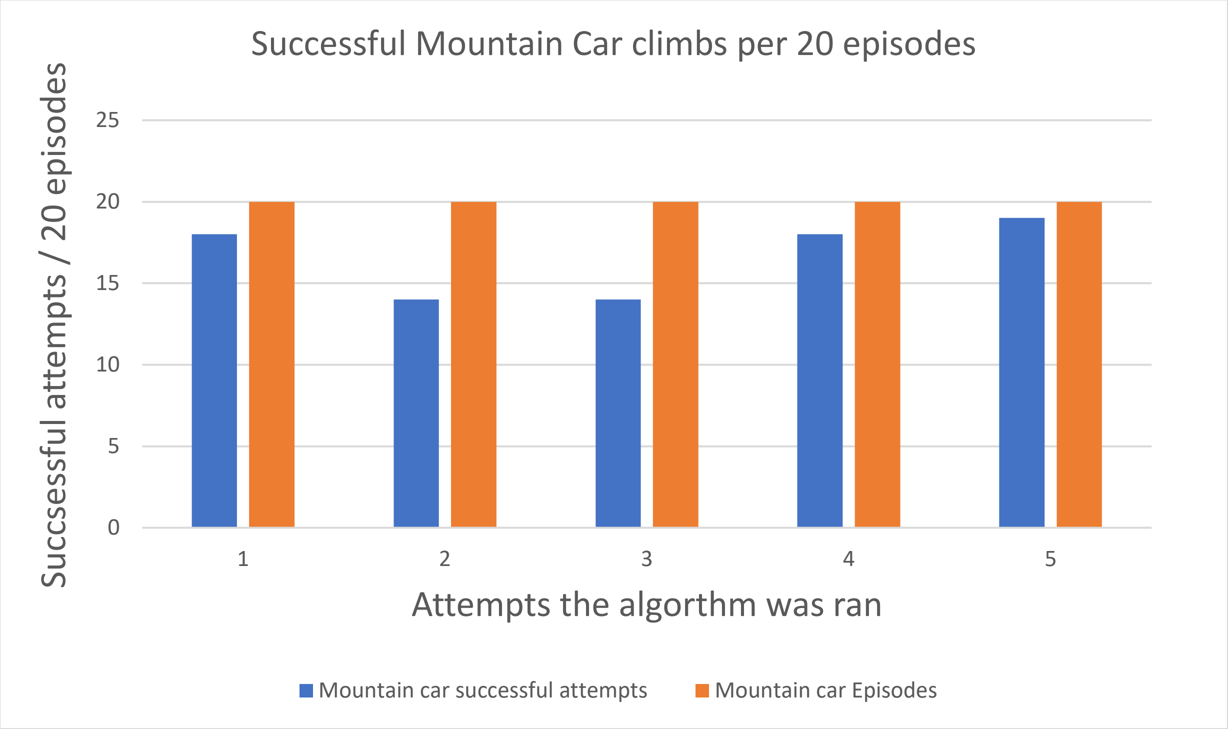


```
Episode 100 Average Reward: -200.0
Episode 200 Average Reward: -200.0
Episode 300 Average Reward: -200.0
Episode 400 Average Reward: -200.0
Episode 500 Average Reward: -200.0
Episode 600 Average Reward: -200.0
Episode 700 Average Reward: -200.0
Episode 800 Average Reward: -200.0
Episode 900 Average Reward: -200.0
Episode 1000 Average Reward: -200.0
Episode 1100 Average Reward: -200.0
Episode 1200 Average Reward: -200.0
Episode 1300 Average Reward: -200.0
Episode 1400 Average Reward: -200.0
Episode 1500 Average Reward: -200.0
Episode 1600 Average Reward: -200.0
Episode 1700 Average Reward: -200.0
Episode 1800 Average Reward: -200.0
Episode 1900 Average Reward: -200.0
Episode 2000 Average Reward: -200.0
```



```
Episode 3900 Average Reward: -198.93
Episode 4000 Average Reward: -199.93
Episode 4100 Average Reward: -198.86
Episode 4200 Average Reward: -198.81
Episode 4300 Average Reward: -198.15
Episode 4400 Average Reward: -199.21
Episode 4500 Average Reward: -198.59
Episode 4600 Average Reward: -199.21
Episode 4700 Average Reward: -199.67
Episode 4800 Average Reward: -199.21
Episode 4900 Average Reward: -198.42
Episode 5000 Average Reward: -197.07
Episode 5100 Average Reward: -196.69
Episode 5200 Average Reward: -194.43
Episode 5300 Average Reward: -198.99
Episode 5400 Average Reward: -190.47
Episode 5500 Average Reward: -198.67
Episode 5600 Average Reward: -189.45
Episode 5700 Average Reward: -198.83
Episode 5800 Average Reward: -188.51
Episode 5900 Average Reward: -168.4
Episode 6000 Average Reward: -182.07
```

Average amount of times the cart was able to successfully go up the mountain:



Evaluation of results

From the findings of the results, it would be fair to say that the implemented Q-Learning algorithm was fairly accurate at exploiting its gained q-table after its exploration phase and that the decaying epsilon implementation was very useful in allowing for the algorithm to successfully achieve the end goal at a sustainable level of nearing 100% and holding onto around the 75% mark of efficiency for the agent to be able to complete the given environment requirements. It is clear that some attempts where the algorithm was run had a much higher efficiency than others but none the less the results prove to show a fairly accurate and efficient Q-learning algorithm when taking into account it only used 6000 episodes to converge its found results into the Q-table using the algorithm, all while being said that this algorithm was adjusted to work with a 2-dimensional vector/box state space as opposed to its original single state space vector, although the 2-dimensional state space could've acted as a limitation in my algorithms ability and effectiveness to allow the agent to complete the environment It was able to work around that issue to prevent it being so as described above in the mathematical model of the algorithm which can be seen above.

Although this algorithm is different to the ones used in the previous mentioned environment solutions, this algorithm still follows the same Q-Learning Formula with the use of Q-values and Q-Tables, as well as the structure in which the Q-Learning algorithm is used, the adjustment to the algorithm to allow for it to work with a 2-dimentional state space did not change how the algorithm worked and the process in which it worked but rather used it to adapt and solve for a 2-dimensional state space while keeping the functionality of the Q-learning algorithm intact and near exactly the same. This shows just how powerful and useful the Q-learning algorithm and mathematics behind the Q-Learning algorithm can be for a range of Artificial intelligence problems in which Q-learning can be implemented into in order to solve the environment for a given agent, state space, action space and state-action pairs.

Bibliography

Brendan Martin, S. K., 2021. *Reinforcement Q-Learning from Scratch in Python with OpenAI Gym.* [Online]
Available at: https://www.learndatasci.com/tutorials/reinforcement-q-learning-scratch-python-openai-gym/
[Accessed 24 May 2021].

Fakhry, A., 2020. *Using Q-Learning for OpenAI's CartPole-v1.* [Online]
Available at: https://medium.com/swlh/using-q-learning-for-openais-cartpole-v1-4a216ef237df
[Accessed 25 May 2021].

Frans, K., 2016. *openai-cartpole.* [Online]
Available at: https://github.com/kvfrans/openai-cartpole/blob/master/cartpole-random.py
[Accessed 25 May 2021].

FRANS, K., 2016. *Simple reinforcement learning methods to learn CartPole.* [Online]
Available at: http://kvfrans.com/simple-algoritms-for-solving-cartpole/
[Accessed 23 May 2021].

Hayes, G., 2019. *Getting Started with Reinforcement Learning and Open AI Gym.* [Online]
Available at: https://towardsdatascience.com/getting-started-with-reinforcement-learning-and-open-ai-gym-c289aca874f
[Accessed 23 May 2021].

Ihvy, L., 2020. *Taxi-v3-Q-Learning.* [Online]
Available at: https://github.com/lhvy/Taxi-v3-Q-Learning/blob/master/agent.py
[Accessed 24 May 2021].

Wikipedia, 2021. *Q-learning.* [Online]
Available at: https://en.wikipedia.org/wiki/Q-learning
[Accessed 25 May 2021].