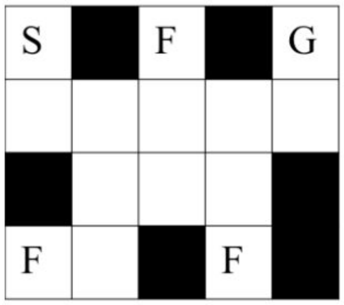
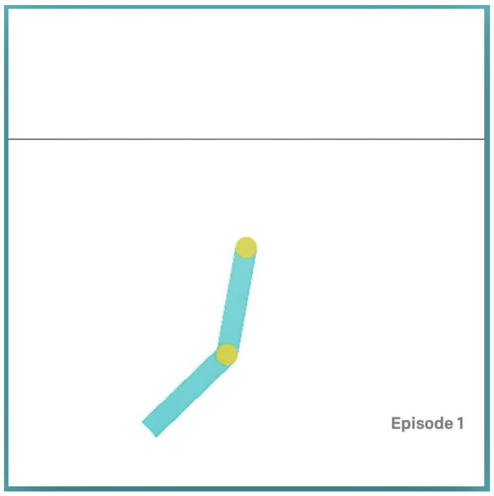
I/ Introduction

In this project, I implement various types of Reinforcement Learning for an agent to successfully navigate and achieve the goals in 3 environments. I solve this using Value Iteration, Q Learning, and Policy Gradient Descent using Neural Network.

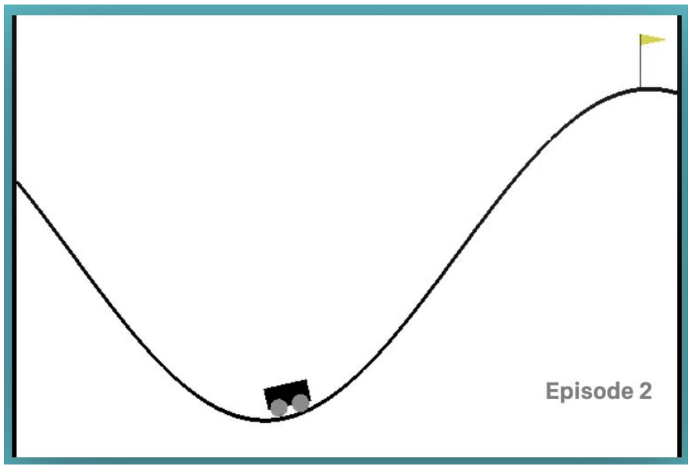
1) Maze: How to get from S to G in the Maze, and collect the most number of flags F along the way. There are 4 actions the robot can take: going UP, DOWN, LEFT, RIGHT.



2) Acrobot: How to swing the bot's arm to reach a certain height.

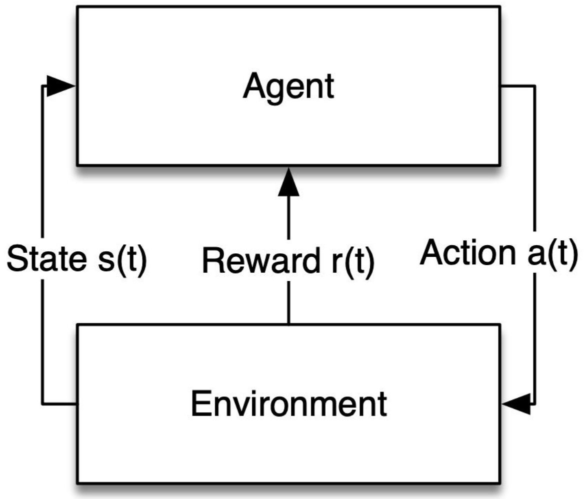


3) Mountain Car: How to push & pull the car to make it reach the mountain's top.



II/ Problem Formulation

All 3 problems can be formulated as a reinforcement learning MDP problem, in which an agent takes actions in an unknown environment, gets rewards and state feedback, then learns through trial and error.

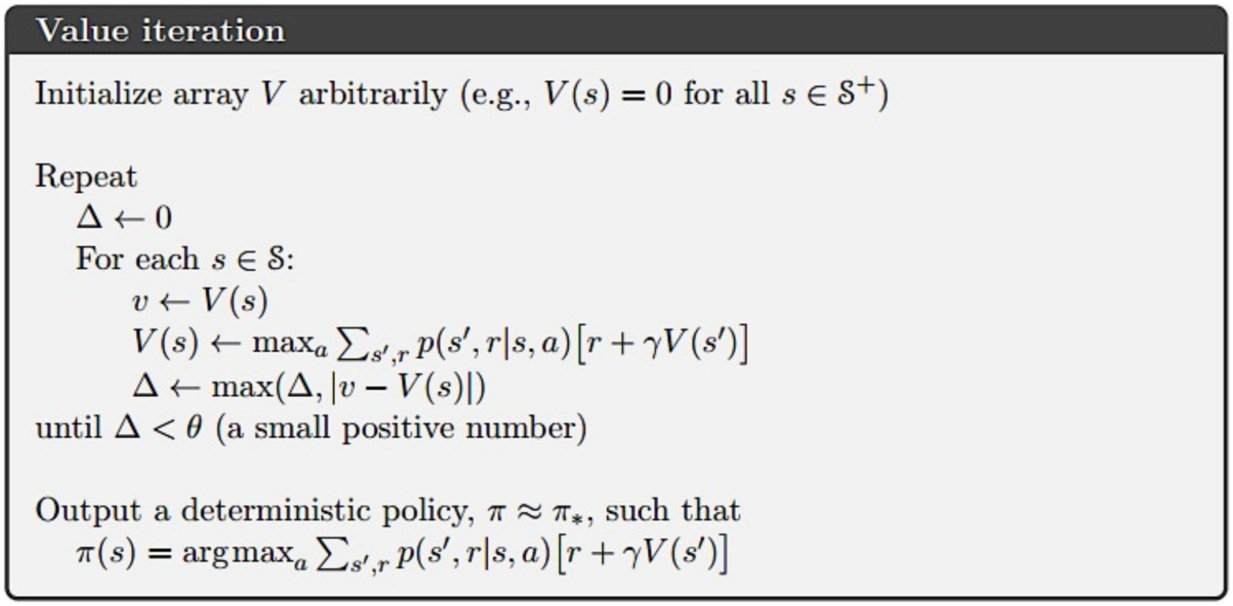


III/ Technical Approach

I experiment with the following approaches to solve reinforcement learning:

1) Value Iteration

The main idea is to start from the end state, then go backward and calculate the discounted reward for each state before that. Eventually this will converge to produce the optimal policy. This algorithm requires a model of all states in the environment.



```
Value iteration

Initialize array $V$ arbitrarily (e.g., $V(s) = 0$ for all $s \in \mathcal{S}^+$)

Repeat
    $\Delta \leftarrow 0$
    For each $s \in \mathcal{S}$:
        $v \leftarrow V(s)$
        $V(s) \leftarrow \max_a \sum_{s',r} p(s',r|s,a)\big[r + \gamma V(s')\big]$
        $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$ (a small positive number)

Output a deterministic policy, $\pi \approx \pi_*$, such that
    $\pi(s) = \mathrm{argmax}_a \sum_{s',r} p(s',r|s,a)\big[r + \gamma V(s')\big]$
```

2) Q-Learning

Q-Learning aims to work even when we don’t have a good model of the environment. The main idea is to let the agent play many episodes and explore the environment. Then based on the rewards from those episodes, try to estimate the Q-value, which is how good a state-action pair is.

This works well for environments with a few discrete states, such as Maze. For environments with continuous states like Acrobot and Mountain Car, I discretize the states first before feeding them into Q-learning algorithm.

The full Q-learning formula is as below:

Definitions: Value function and Q-value function

Following a policy produces sample trajectories (or paths) s_0, a_0, r_0, s_1, a_1, r_1, …

How good is a state?
The **value function** at state s, is the expected cumulative reward from following the policy from state s:

$$V^{\pi}(s) = \mathbb{E}\left[\sum_{t\geq 0} \gamma^t r_t | s_0 = s, \pi\right]$$

How good is a state-action pair?
The **Q-value function** at state s and action a, is the expected cumulative reward from taking action a in state s and then following the policy:

$$Q^{\pi}(s,a) = \mathbb{E}\left[\sum_{t\geq 0} \gamma^t r_t | s_0 = s, a_0 = a, \pi\right]$$

Value iteration algorithm: Use Bellman equation as an iterative update

$$Q_{i+1}(s,a) = \mathbb{E}\left[r + \gamma \max_{a'} Q_i(s',a') | s, a\right]$$

Q_i will converge to Q* as i -> infinity

3) REINFORCE - Policy Gradient Descent

REINFORCE tries to estimate a good policy directly, without keeping track of Q values. This will work even in environments with continuous states. This is done by letting the agent play many games and gradually adjust the policy function to map any state to the best action.

Function approximation is the strength of neural network, and that is what I use for my policy function. It's a simple network with 3 layers: Input layer (environment states) - Hidden layer (20 nodes) - Output layers (actions' probabilities).

The REINFORCE algorithm is as below:

```
function REINFORCE
    Initialise $\theta$ arbitrarily
    for each episode $\{s_1, a_1, r_2, ..., s_{T-1}, a_{T-1}, r_T\} \sim \pi_\theta$ do
        for $t = 1$ to $T-1$ do
            $\theta \leftarrow \theta + \alpha \nabla_\theta \log \pi_\theta(s_t, a_t) v_t$
        end for
    end for
    return $\theta$
end function
```

IV/ Results

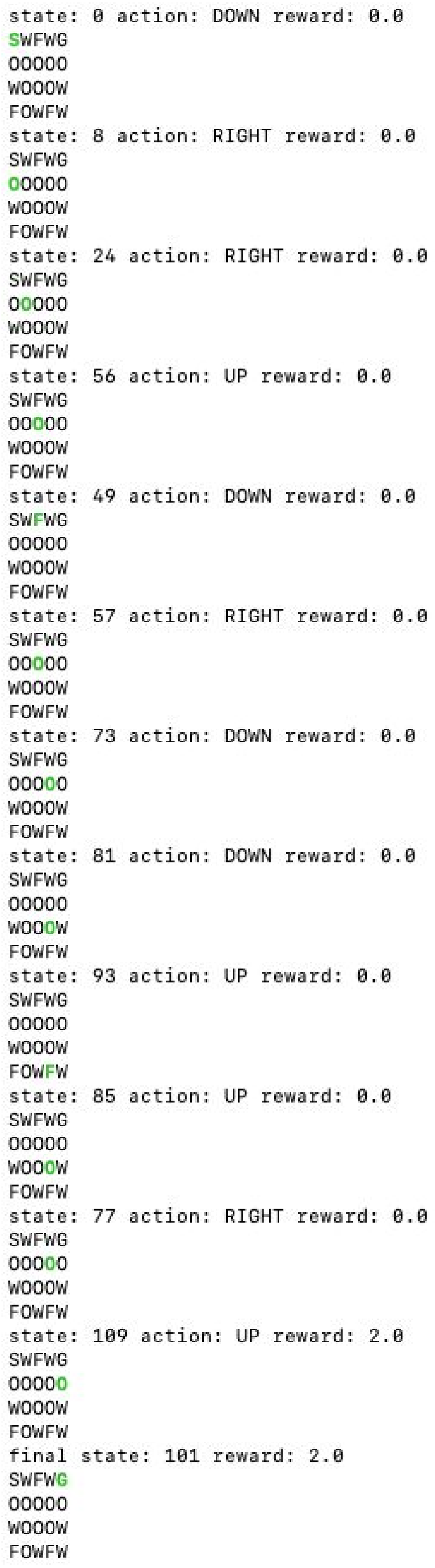
1) Value Iteration in Maze game

Value discount rate: 0.9

Starting at State 0, the optimal policy is: Down, Right, Right, Up, Down, Right, Down, Down, Up, Up, Right, Up

Total rewards: 2 flags

See the picture of the full path on next page.

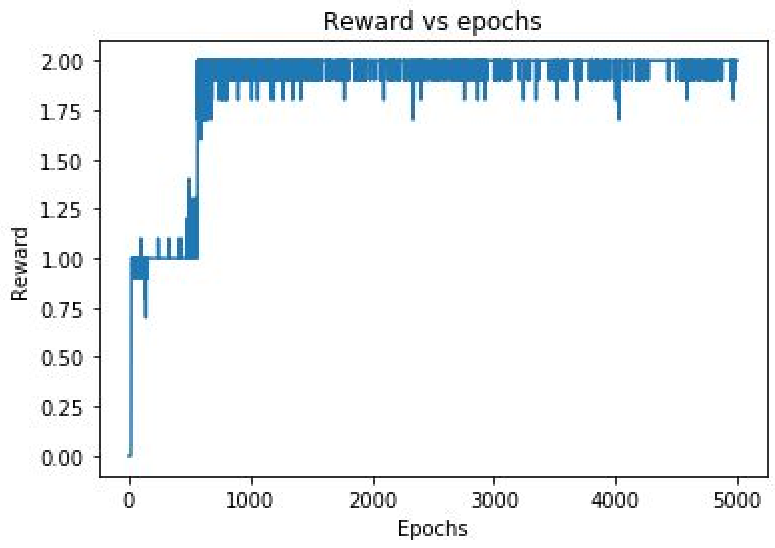


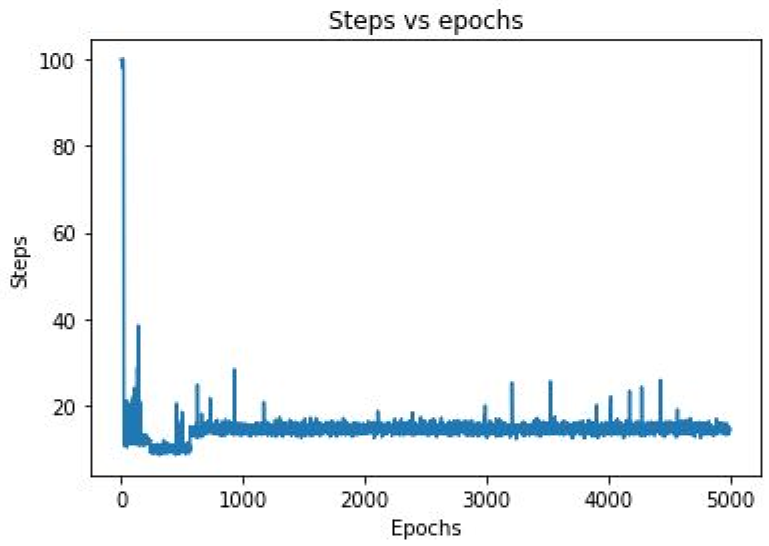
```
state: 0 action: DOWN reward: 0.0
SWFWG
OOOOO
WOOOW
FOWFW
state: 8 action: RIGHT reward: 0.0
SWFWG
OOOOO
WOOOW
FOWFW
state: 24 action: RIGHT reward: 0.0
SWFWG
OOOOO
WOOOW
FOWFW
state: 56 action: UP reward: 0.0
SWFWG
OOOOO
WOOOW
FOWFW
state: 49 action: DOWN reward: 0.0
SWFWG
OOOOO
WOOOW
FOWFW
state: 57 action: RIGHT reward: 0.0
SWFWG
OOOOO
WOOOW
FOWFW
state: 73 action: DOWN reward: 0.0
SWFWG
OOOOO
WOOOW
FOWFW
state: 81 action: DOWN reward: 0.0
SWFWG
OOOOO
WOOOW
FOWFW
state: 93 action: UP reward: 0.0
SWFWG
OOOOO
WOOOW
FOWFW
state: 85 action: UP reward: 0.0
SWFWG
OOOOO
WOOOW
FOWFW
state: 77 action: RIGHT reward: 0.0
SWFWG
OOOOO
WOOOW
FOWFW
state: 109 action: UP reward: 2.0
SWFWG
OOOOO
WOOOW
FOWFW
final state: 101 reward: 2.0
SWFWG
OOOOO
WOOOW
FOWFW
```

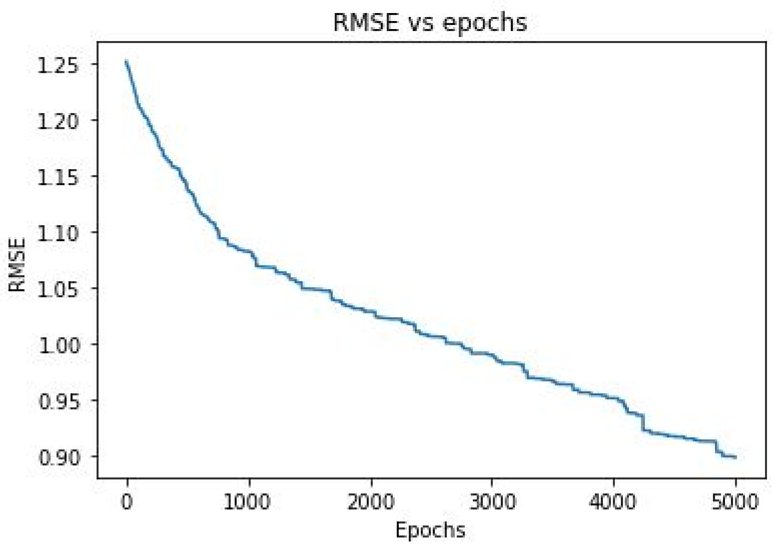
2) Q-learning in Maze game

The best parameters are: Learning rate = 0.2; Epsilon = 0.5. Achieved a stable reward of 2 after ~700 episodes. Converged near Q* after 5000 episodes.

a) Learning rate = 0.2; Epsilon = 0.5

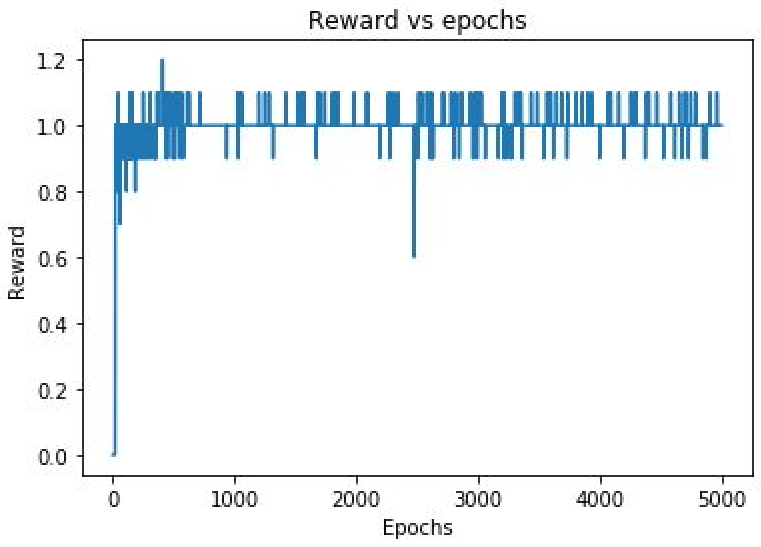


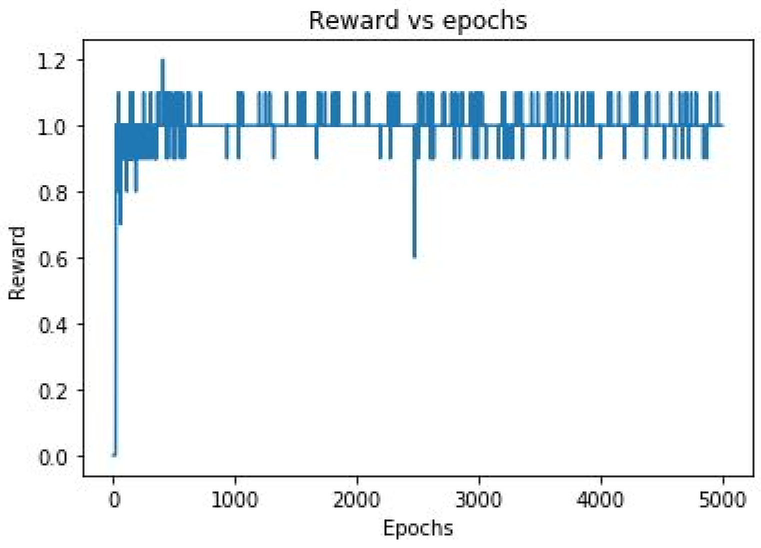


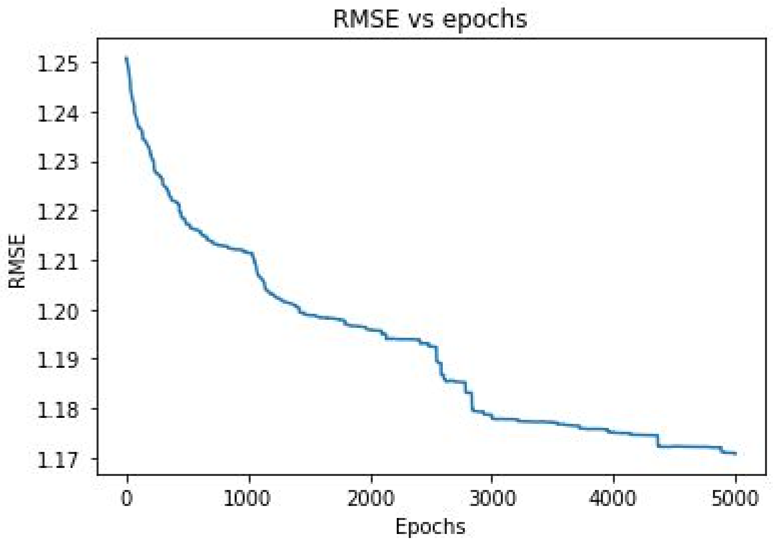


b) Learning rate = 0.2, Epsilon = 0.1

Less exploration seems to hurt this experiment.

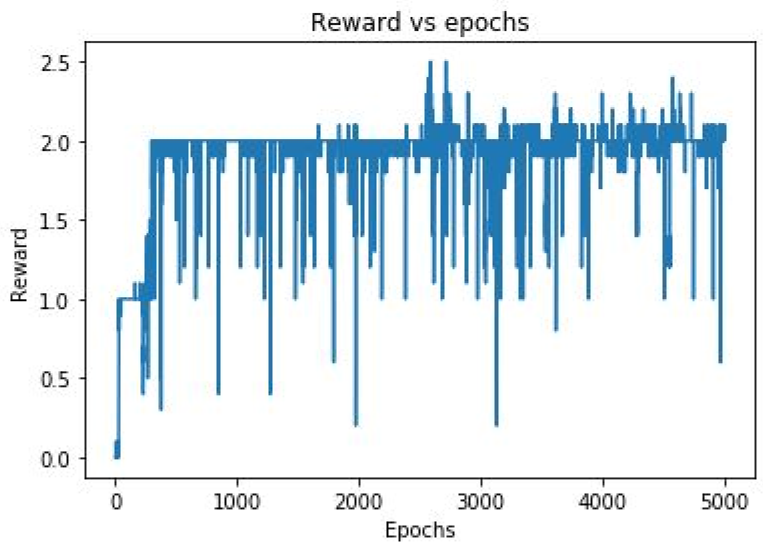


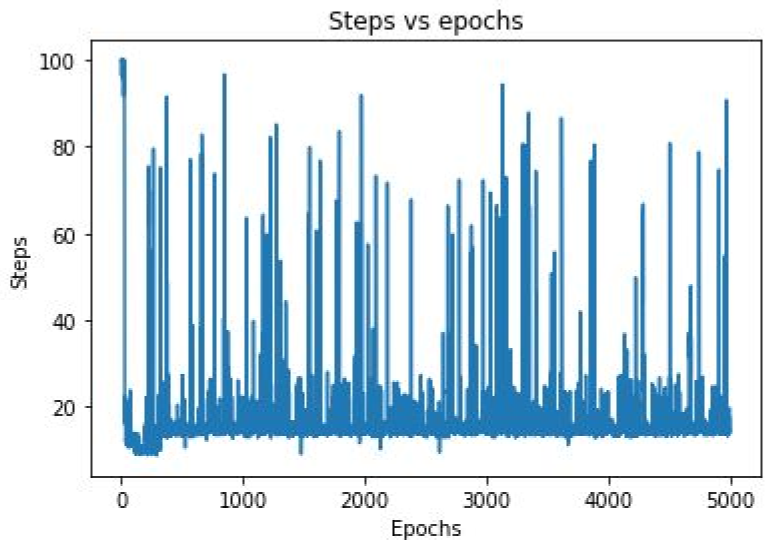


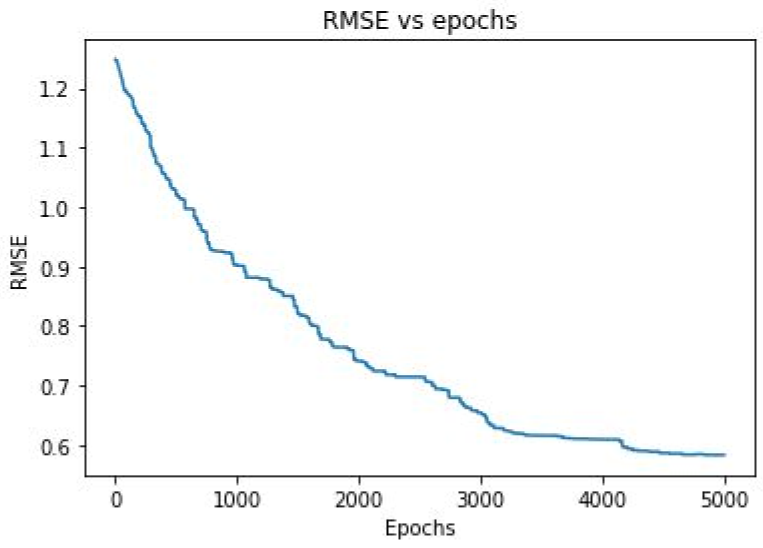


c) Learning rate = 0.5; Epsilon = 0.5

Learning rate too high seems to create unstable results. While Q converge to Q* faster (based on RMSE), the rewards keep fluctuating even after 1000 episodes.


Reward vs epochs
Reward
Epochs
0.0
0.5
1.0
1.5
2.0
2.5
0
1000
2000
3000
4000
5000


Steps vs epochs
Steps
Epochs
20
40
60
80
100
0
1000
2000
3000
4000
5000


RMSE vs epochs
RMSE
Epochs
0.6
0.7
0.8
0.9
1.0
1.1
1.2
0
1000
2000
3000
4000
5000

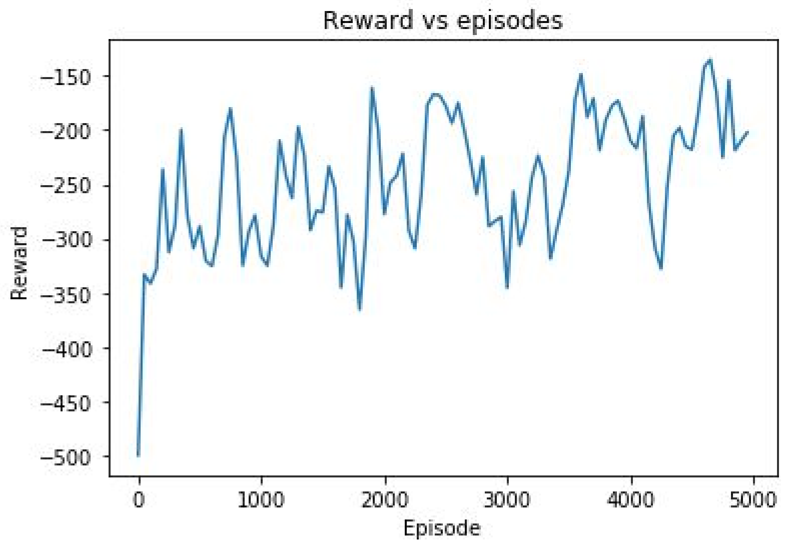
3) Q-learning in OpenAI Gym

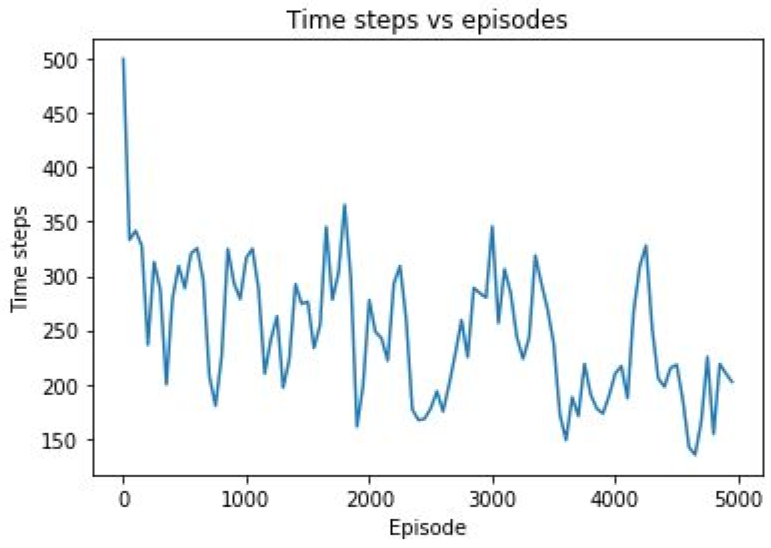
a) Acrobot

Video result: https://youtu.be/3pWVL1ZAAIY

The model achieves a decent performance after just ~20 episodes. Eventually win the game at ~200 timesteps.

Parameters: Learning rate = 0.4; Epsilon = 0.03 to 0.001 (gradually decay after every 100 episodes)



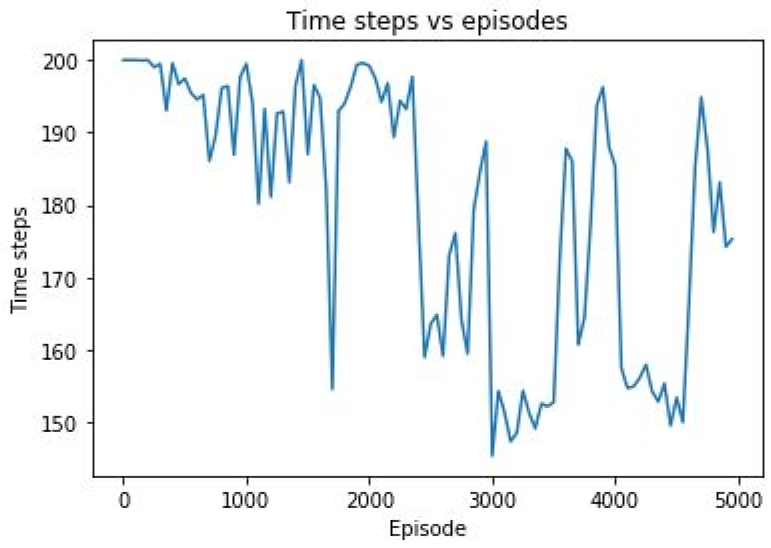


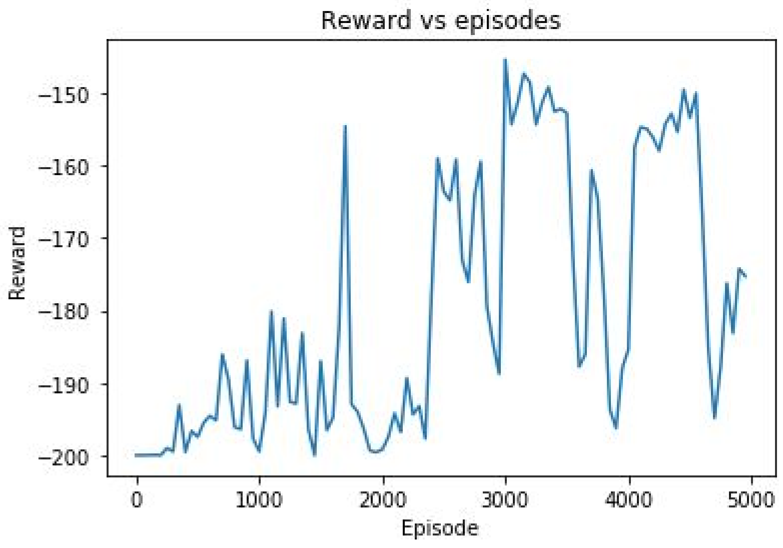
b) Mountain Car

Video result: https://youtu.be/DeB7d-cF0gI

The model achieves a decent result after ~3000 episodes, though it's a bit unstable over different episodes. Eventually win the game at ~180 time steps.

Parameters: Learning rate = 0.4; Epsilon = 0.03 to 0.001 (gradually decay after every 100 episodes)





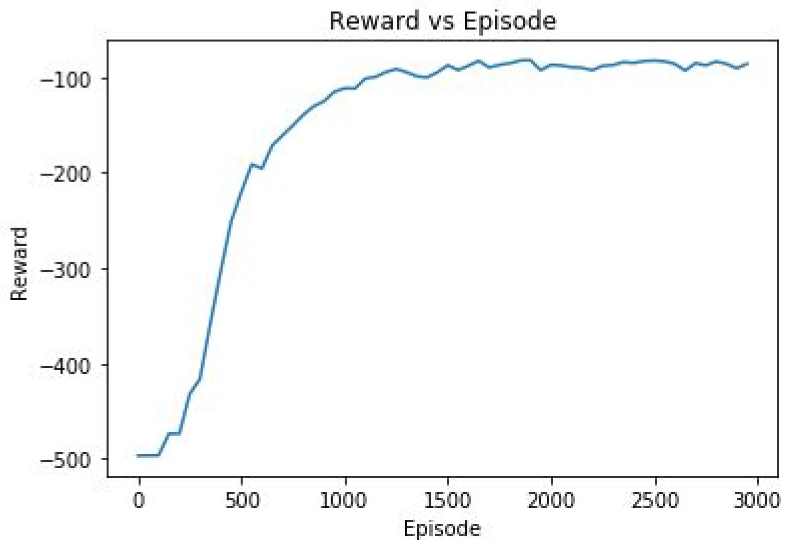
4) REINFORCE in OpenAI Gym

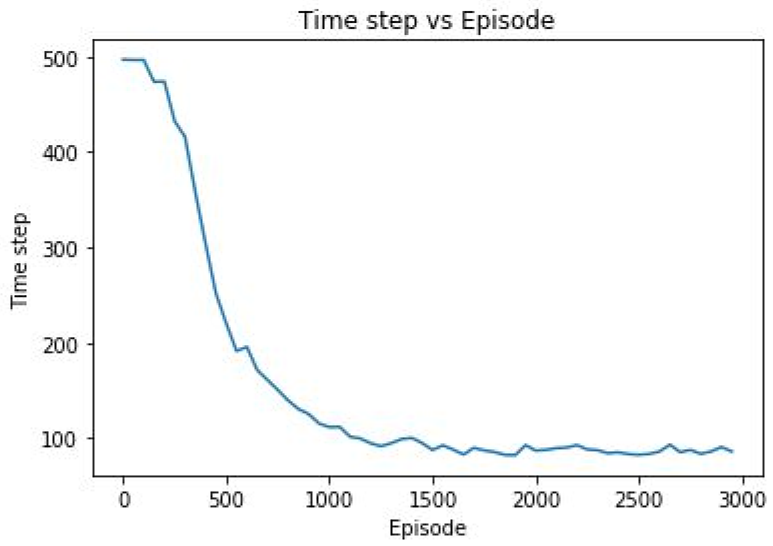
a) Acrobot

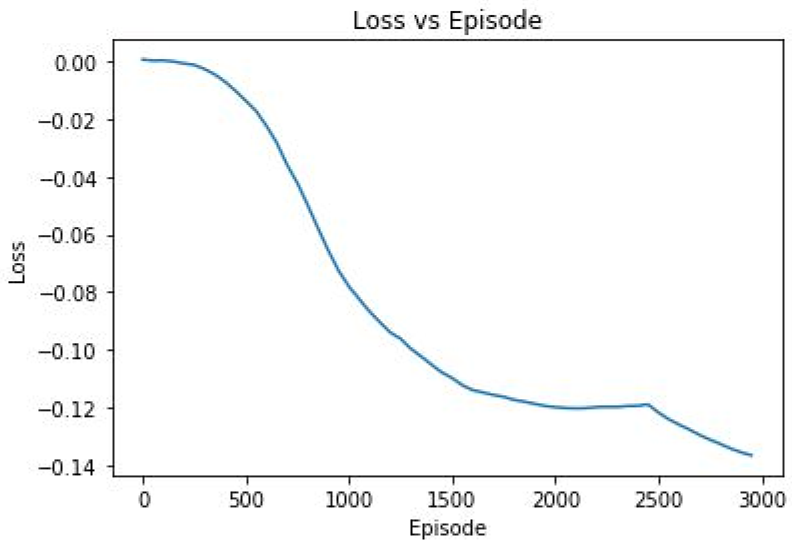
Video result: https://youtu.be/ecF-w1arABs

The model converges after ~1000 episodes. Eventually win the game after ~100 time steps, outperforming Q-learning!

Parameters: Learning rate = 0.01 (Adam optimizer); Batch size = 50; Value discount = 0.99







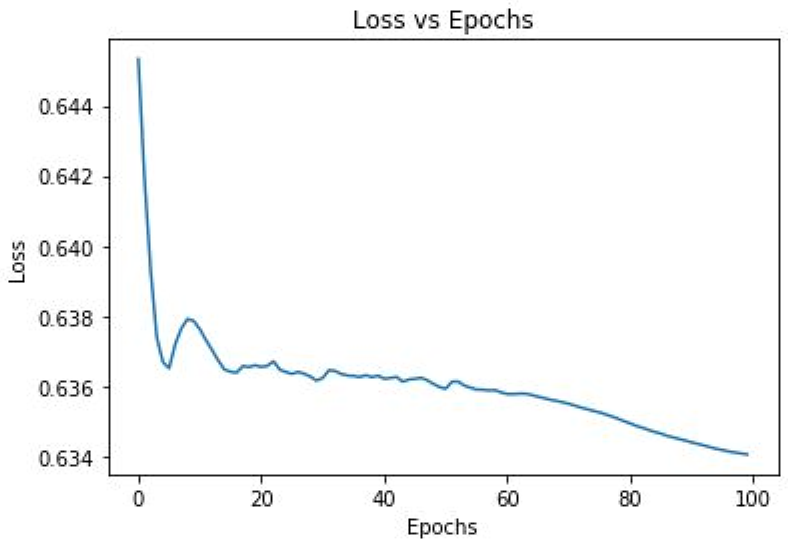
b) Mountain Car

Video result: https://youtu.be/weExxc8X95Q

Mountain Car is a lot more challenging than Acrobot. It's very rare for the car to reach mountain top by chance, so the model doesn't have enough success examples to learn. Using vanilla REINFORCE, the training loss doesn't decrease even after 5000 episodes.

I try a different approach to collect more success examples. I play the games randomly for 100,000 times, then record the "good games" where the car successfully reachs the mountain top. There are only 50 good games, which I use to train the policy network. It quickly converges after 100 epochs & learning rate 0.01 (Adam optimizer).

The final model wins the game in ~120 time steps, outperforming Q-learning.



V/ Discussion

Deep Reinforcement Learning clearly outperforms traditional Q-learning and Value Iteration in a more complex environment with continuous states. The challenge is the amount of data needed to train Deep RL model is a lot more, as seen in the case of REINFORCE on Mountain Car. One way to shorten the training might be to feed human-engineered features into the model, such as acceleration. Anyway, I’m hopeful for the potential of Deep Reinforcement Learning in real-life robotics!