

Frans Fourie

Intelligent Autonomous Systems

Reinforcement Learning on MDP

Environments

Contents

Introduction	4
Problem Formulation	4
Technical Approach.....	4
Part 1 Policy Iteration / Value Iteration	4
Value Iteration	5
Part 2 Q-learning.....	5
Action Policy.....	6
Optimal Q values.....	7
Performance Evaluation.....	7
Part 3 Q-learning.....	7
Reinforcement Learning.....	7
Results.....	8
Part 1 Results	8
Part 2 Results	9
Part 3 Results	21
Mountain Car	21
Acrobatic	25
Discussion.....	29
Part 1.....	29
Part 2.....	30
Part 3.....	30
Conclusion.....	30

Table of Tables

Table 1: Hyperparameters for part 1	8
Table 2: Optimal found policy	8
Table 3: Path followed during lucky run	9
Table 4: Part 2 hyperparameters	9
Table 5: Epsilon experimentations.....	10
Table 6: learning rate experimentations	14
Table 7: Evaluation graphs of experiments	18
Table 8: Mountain car hyperparameters	21

Table of Figures

Figure 1: Value iteration algorithm.....	5
Figure 2: Q-Learning algorithm	6
Figure 3: Epsilon greedy policy	6
Figure 4: RMSE formula	7
Figure 5: Reinforcement learning algorithm.....	8
Figure 6: Mountain Car Q-Learning steps per iteration.....	22
Figure 7: Mountain Car Q-Learning rewards per iteration	23
Figure 8: Mountain Car Reinforcement learning steps per iteration	24
Figure 9: Mountain Car Reinforcement learning rewards per iteration.....	25
Figure 6: Acrobatic Q-Learning steps per iteration.....	26
Figure 7: Acrobatic Q-Learning rewards per iteration	27
Figure 8: Acrobatic Reinforcement learning steps per iteration	28
Figure 9: Acrobatic Reinforcement learning rewards per iteration.....	29

Introduction

This report documents the use of Q-Learning and Reinforce algorithms in order to learn optimal policies for specific Markov Decision Process (MDP) environments. The first part of the project involves using value iteration or policy iteration in order to find the optimal values and policy for the given MDP as well as constructing to accompanying Q Table. The second part of the project involves using Q-Learning to solve the same MDP as that of part 1. The Q-Learning algorithm that was implemented makes use of a constant learning rate accompanied by the epsilon greedy action selection policy. The Q Table of this Q-Learning method can then be compared to that generated in part 1. The performance of the algorithm can also be evaluated by looking at the average reward and steps taken at each iteration. For part 3 of the project two specific MDP environments will be solved using both Q-Learning as well as a Reinforce learning algorithm. These approaches will be compared by evaluating there performance.

Problem Formulation

Given a specific MDP environment make use of value iteration and policy iteration to find the optimal policy and values. Also use Q-Learning to find the best policy by approximating the optimal solutions. Lastly use reinforcement learning to also try and find a good policy and approximate the optimal solutions. In order to best approximate the optimal solutions using both Q-Learning and Reinforcement learning experimentation needs to be done in order to find the best hyper parameters to use for the different algorithms and different MDP environments.

Technical Approach

The following is technical approaches followed in order to complete this project with an explanation of the algorithm used for each part. A large part of the implementation of these algorithms is finding the optimal hyperparameters to use. To this end extensive testing was done with multiple values to find the optimal values. The results of these experimentations are included in the results section.

Part 1 Policy Iteration / Value Iteration

In order to solve part 1 of this project value iteration is used. The program starts by declaring all the constants and parameters that will be used in order to solve this part of the project. The environment is initialized as the maze MDP environment. The slip of the environment is set to 0 for training of the model and calculating the optimal solutions.

Value Iteration

In order to find the optimal solutions Value iteration is used by implementing the following algorithm.

Value iteration

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

Repeat

$\Delta \leftarrow 0$

For each $s \in \mathcal{S}$:

$v \leftarrow V(s)$

$V(s) \leftarrow \max_a \sum_{s',r} p(s',r|s,a) [r + \gamma V(s')]$

$\Delta \leftarrow \max(\Delta, |v - V(s)|)$

until $\Delta < \theta$ (a small positive number)

Output a deterministic policy, $\pi \approx \pi_*$, such that

$\pi(s) = \arg\max_a \sum_{s',r} p(s',r|s,a) [r + \gamma V(s')]$

Figure 1: Value iteration algorithm

While doing value iteration we will manually account for the 10% probability of slipping by taking this into account when calculating the optimal Q values. The reason the environment slippage is turned off and slippage is manually added then is in order to make it predictable and make training the model reliable. The algorithm terminates and proclaims that it has found the optimal solutions when the change in the optimal values between iterations shrink to a value smaller than theta which in this application was 1e-200. During the algorithm, the maze environment is used in order to find the next state given the current state and an action. The environment is also used to find the reward given by the next state and to see when the maze has been completed and the goal reached.

Part 2 Q-learning

For this part we implemented the Q-learning algorithm. The program starts by initializing the maze environment as well as declaring constants and parameters. Now the Q-Learning algorithm is implemented running for the defined number of steps. The formula used for calculating the q values is the following.

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

Figure 2: Q-Learning algorithm

One of the initially defined parameters is the learning rate that will be used in the above formula to indicate the weight of past Q values when updating a Q value. Setting the learning rate to 0 will mean nothing new will be learnt by the algorithm and the values will stay constant. Setting the value to 1 will mean old values are completely discarded when updating the values. Because of this a learning rate decay is implemented meaning that the learning rate will decay over time this will allow the algorithm to take large steps and learn quickly at the start and then take smaller steps to stabilize and eventually converge at the end.

Action Policy

For this implementation, an epsilon greedy action selection policy will be used. The following is the formula used to implement this policy.

$$Action = \begin{cases} \max_a Q(s, a), & R > \epsilon \\ Random\ a, & R \leq \epsilon \end{cases}$$

Figure 3: Epsilon greedy policy

The reason for introducing the epsilon greedy policy is to allow the algorithm to 'explore'. Without this policy the algorithm might find a specific action very rewarding for a certain state at the start of the algorithm and this will lead to the algorithm never exploring the possibility of different actions having better rewards in later iterations. Thus, by introducing this policy we create an exploration factor which for a certain percentage of times will allow the algorithm to choose a random action. In the implementation we will make use of epsilon decay meaning that the value of epsilon will decrease every iteration. The reason for this decay is that at the start we want the algorithm to explore a lot in order to identify better paths but then later we want the algorithm to start following the optimal solution more closely.

Optimal Q values

In order to compare the results of the Q-Learning algorithm the root mean square error (RMSE) between the Q Table gotten from part one using value iteration and the Q Table gotten from part two using Q-Learning is calculated. The RMSE was calculated using the following formula.

$$\text{RMSE} = \sqrt{\frac{1}{n} \sum_{j=1}^n (y_j - \hat{y}_j)^2}$$

Figure 4: RMSE formula

Performance Evaluation

In order to evaluate the performance of the Q-Learning algorithm the evaluation.py file that was given is used. The epsilon greedy policy code is implemented in this file also. The policy is evaluated at every 50 iterations by the evaluation program. From this program we get back the average steps/10 and the average reward/10. This can now be plotted against the iterations to evaluate the performance of the algorithm. If the algorithm works as expected, we will see the rewards moving to a maximum while the steps tend to a minimum over the iterations.

Part 3 Q-learning

For part 3 of the project two new environments will be explored the mountain car and the acrobat MDP environments. For the mountain car the state vector consists of the position and velocity of the car with the goal being to get the car to reach the top of the hill and reach the flag. The acrobat problems state vector consists of 6 parameters which are the sin and cosine components for the different joints with the dot product of the directions. The goal of the acrobat problem is to get the arm to reach the line above the starting position. Both these MDP environments need to be approached using Q-Learning as well as Reinforcement learning and then their performances needs to be evaluated and compared. For Q learning we will use the same approach and formula as outlined in part 2. For the evaluation and comparing of these methods the same evaluation techniques as in part 2 will be used namely looking at the average amount of steps as well as average reward.

Reinforcement Learning

The following is the algorithm implemented in order to do reinforcement learning.

REINFORCE, A Monte-Carlo Policy-Gradient Method (episodic)

Input: a differentiable policy parameterization $\pi(a|s, \theta)$

Initialize policy parameter $\theta \in \mathbb{R}^{d'}$

Repeat forever:

 Generate an episode $S_0, A_0, R_1, \dots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot, \theta)$

 For each step of the episode $t = 0, \dots, T - 1$:

$G \leftarrow$ return from step t

$\theta \leftarrow \theta + \alpha \gamma^t G \nabla_{\theta} \ln \pi(A_t|S_t, \theta)$

Figure 5: Reinforcement learning algorithm

Results

The following are the results obtained from the project for each part.

Part 1 Results

The following hyperparameters where used for the completion of part 1.

Table 1: Hyperparameters for part 1

Hyperparameter	Value
Discount factor	0.9
Theta (value to check for convergence)	1e-200
Chance for slippage	0%
Algorithm	Value iteration

The following is the optimal policy that was found in part 1.

Table 2: Optimal found policy

Policy:
[1 1 1 1 1 1 1 1 3 3 3 3 3 3 3 3 1 1 3 1 3 1 3 3 3 3 3 3 3 3 3 3 3 3 0 0 3
3 0 0 2 2 0 2 0 2 0 0 0 1 0 0 1 1 0 1 0 3 0 0 3 3 0 3 0 3 0 0 3 0 0 0 1 1
2 2 3 3 3 3 1 1 1 2 1 0 0 0 1 1 1 0 1 0 0 0 1 0 0 0 0 0 0 0 2 0 2 0 0 0 0
0]

The following is the path followed by the robot given a lucky run meaning slippage was set to 0.

Table 3: Path followed during lucky run

	Step 1	Step 2	Step 3	Step 4	Step 5	Step 6
Action:	DOWN	RIGHT	RIGHT	UP	DOWN	RIGHT
Map:	SWFWG O O O O O W O O W F O W F W	SWFWG O O O O O W O O W F O W F W	SWFWG O O O O O W O O W F O W F W	SWFWG O O O O O W O O W F O W F W	SWFWG O O O O O W O O W F O W F W	SWFWG O O O O O W O O W F O W F W
If Slip:	SWFWG O O O O O W O O W F O W F W	SWFWG O O O O O W O O W F O W F W	SWFWG O O O O O W O O W F O W F W	SWFWG O O O O O W O O W F O W F W	SWFWG O O O O O W O O W F O W F W	SWFWG O O O O O W O O W F O W F W
	Step 7	Step 8	Step 9	Step 10	Step 11	Step 12
Action:	DOWN	DOWN	UP	UP	RIGHT	UP
Map:	SWFWG O O O O O W O O W F O W F W	SWFWG O O O O O W O O W F O W F W	SWFWG O O O O O W O O W F O W F W	SWFWG O O O O O W O O W F O W F W	SWFWG O O O O O W O O W F O W F W	SWFWG O O O O O W O O W F O W F W
If Slip:	SWFWG O O O O O W O O W F O W F W	SWFWG O O O O O W O O W F O W F W	SWFWG O O O O O W O O W F O W F W	SWFWG O O O O O W O O W F O W F W	SWFWG O O O O O W O O W F O W F W	SWFWG O O O O O W O O W F O W F W

If the robot slips anywhere along the optimal path it will follow the new optimal path that is outlined in the optimal policy. The Q Table is included as a .npy file. The Q Table has been normalized such that the largest number in the table is 1 indicating the probability of that action being the optimal action is 100%.

Part 2 Results

The following is a table of constant hyperparameters that were kept constant across all experiments and used for part 2.

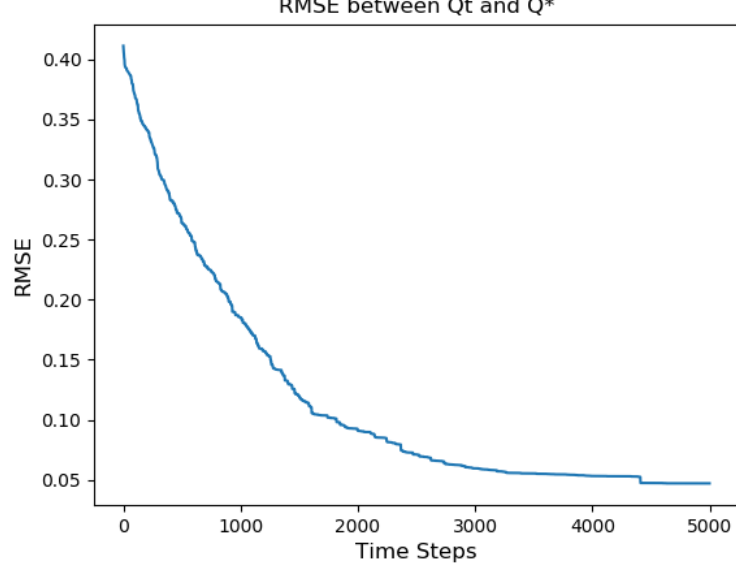
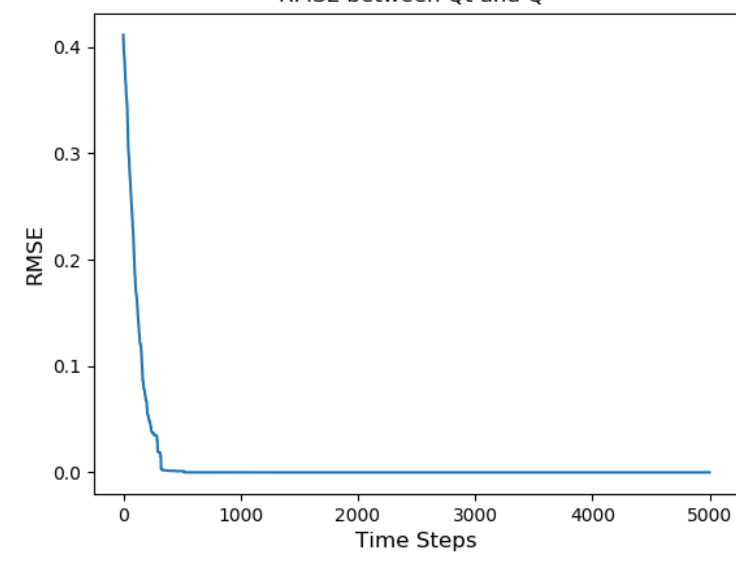
Table 4: Part 2 hyperparameters

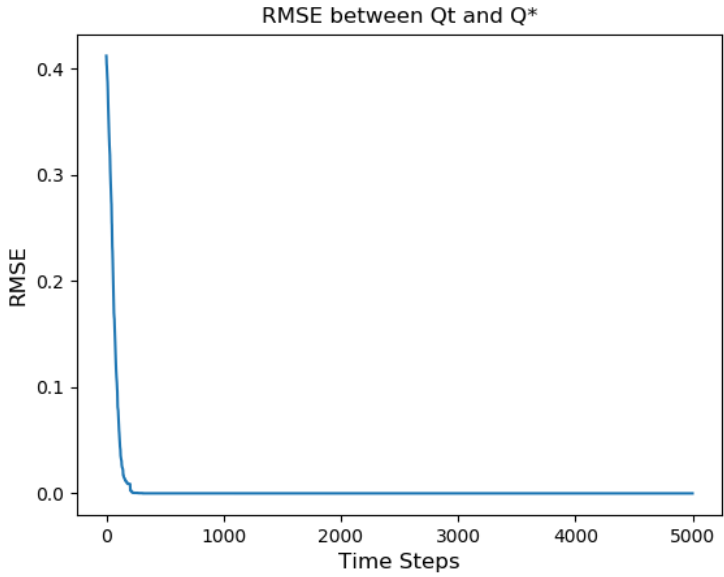
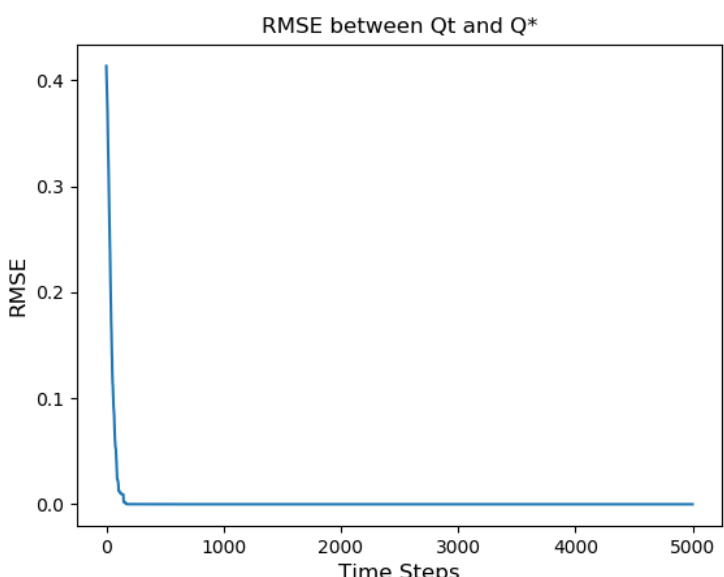
Hyperparameter	Value
Discount factor	0.9
Steps to run algorithm	5000
Chance for slippage	0%
Algorithm	Q-Learning

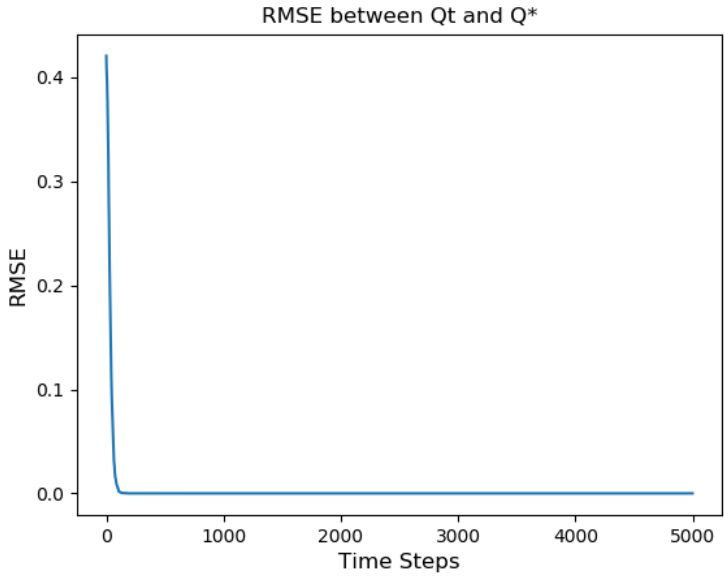
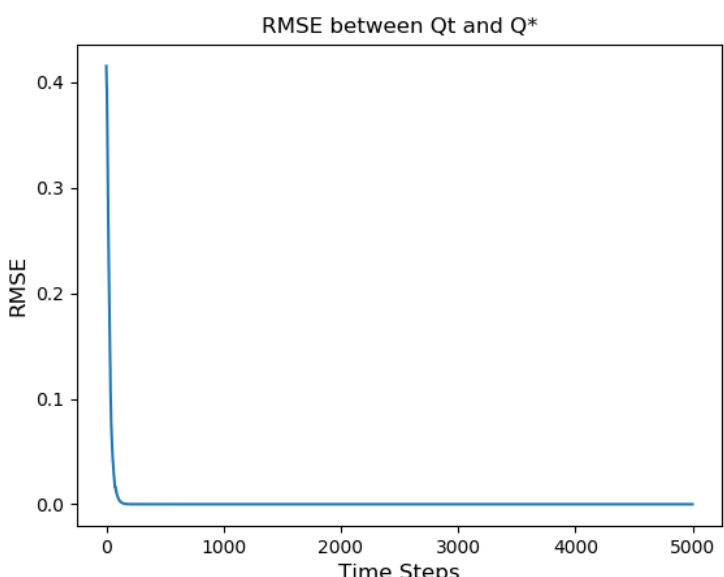
The following is the epsilon experimentation. For this experiment the learning rate was kept constant while the epsilon value was changed. At every value, the Q Table is compared to that of part 1's Q Table

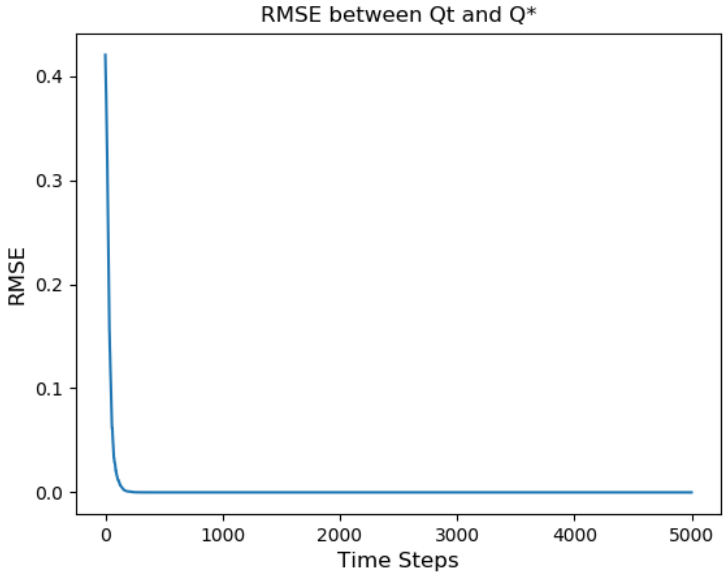
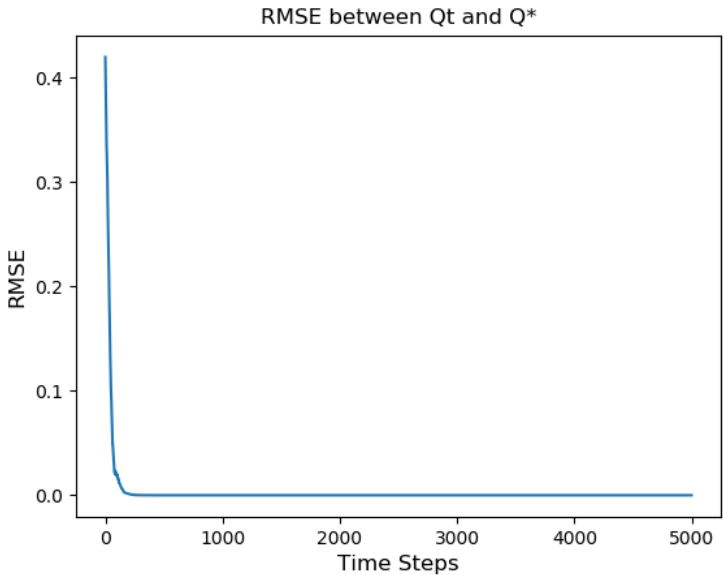
by calculating the RMSE and plotting the RMSE over iterations. For this implementation we make use of epsilon decay thus the value given below is the epsilon start value and will decay linearly until 0 at the last iteration.

Table 5: Epsilon experimentations

Epsilon	Learning Rate	RMSE	Graph
0.01	0.9	4.7e-2	 <p>The graph shows the Root Mean Square Error (RMSE) between Q_t and Q^* over 5000 time steps. The y-axis represents RMSE, ranging from 0.05 to 0.40. The x-axis represents Time Steps, ranging from 0 to 5000. The RMSE starts at approximately 0.40 at time step 0 and decreases rapidly, reaching a plateau of about 0.05 after approximately 3000 time steps.</p>
0.1	0.9	1.607e-16	 <p>The graph shows the Root Mean Square Error (RMSE) between Q_t and Q^* over 5000 time steps. The y-axis represents RMSE, ranging from 0.0 to 0.4. The x-axis represents Time Steps, ranging from 0 to 5000. The RMSE starts at approximately 0.40 at time step 0 and decreases very rapidly, reaching a value of 0.00 by approximately 500 time steps and remaining there until 5000 time steps.</p>

0.2	0.9	1.64e-16	
0.3	0.9	1.67e-16	

0.5	0.9	1.671e-16	
0.7	0.9	1.733e-16	

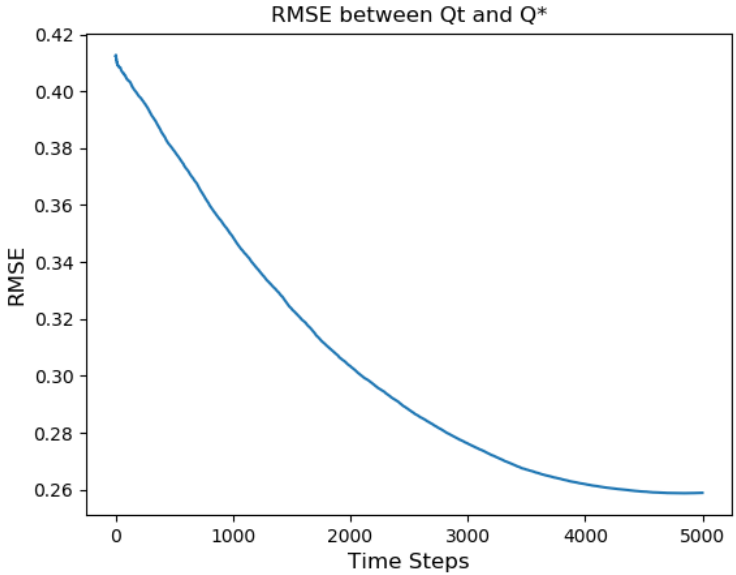
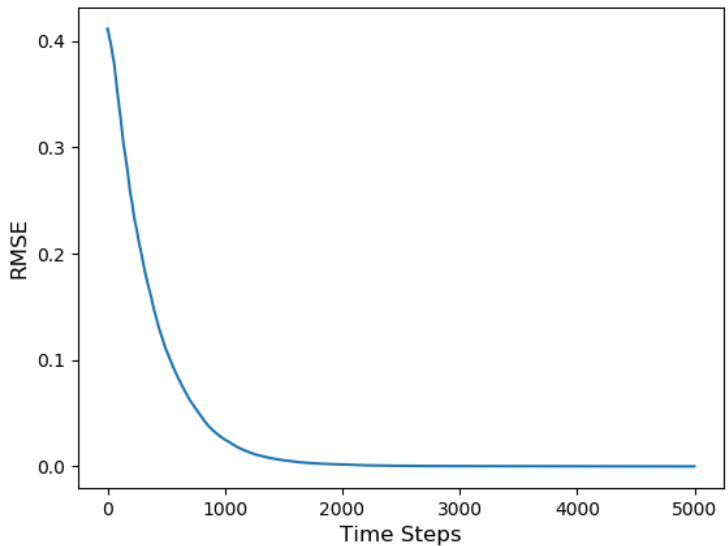
0.9	0.9	1.692e-16	
1	0.9	1.711e-16	

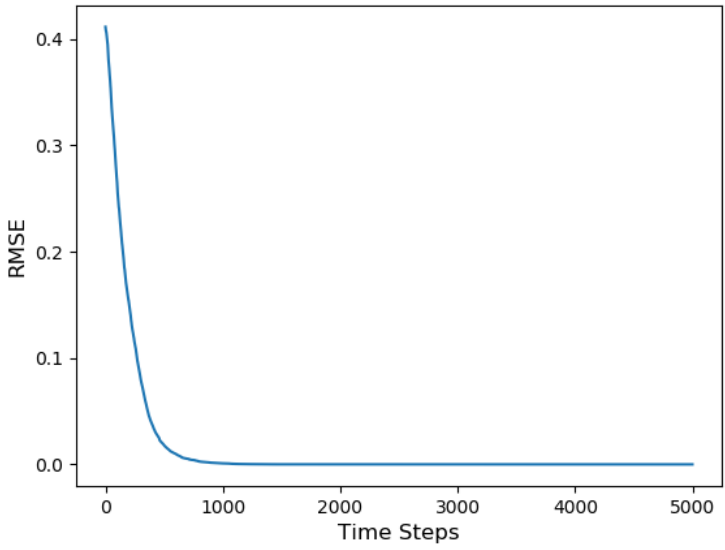
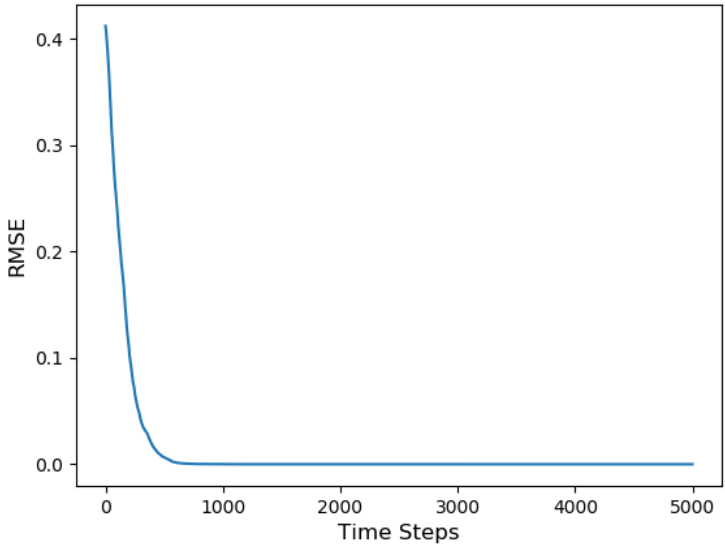
From the above table epsilon 0.2 has been chosen. Although this epsilon value gives the 2nd smallest RMSE and not the smallest it was chosen to ensure there is enough freedom to explore.

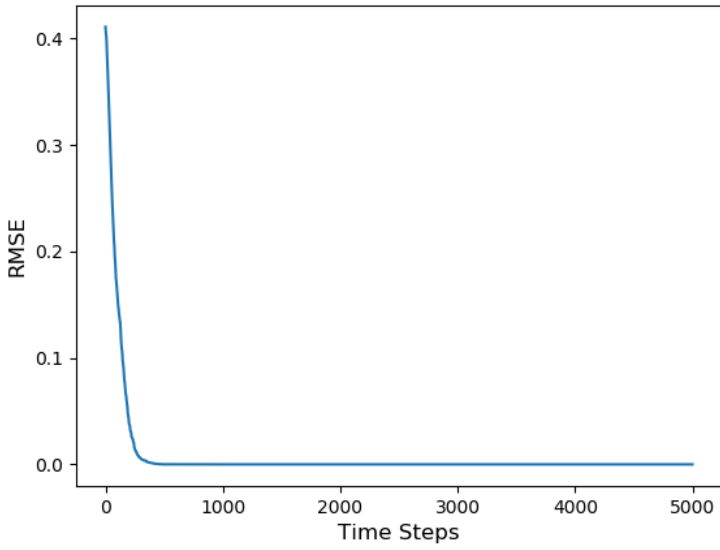
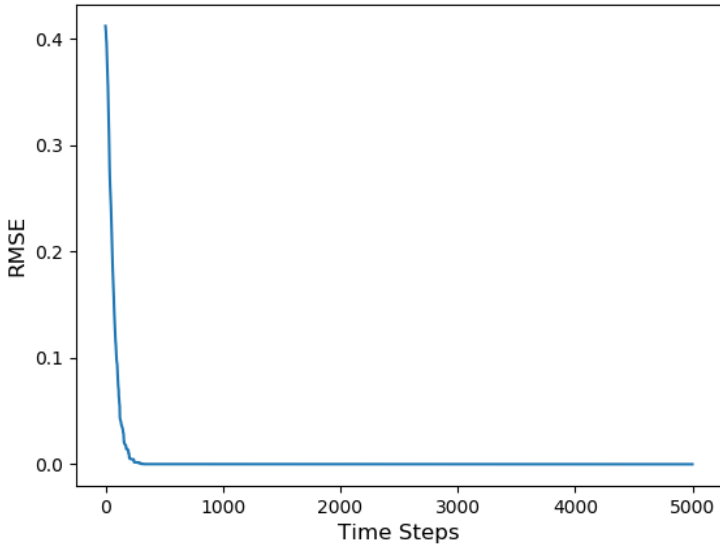
The following is the learning rate experimentation. For this experiment epsilon was kept constant while the learning rate value was changed. At every value, the Q Table is compared to that of part 1's Q Table by calculating the RMSE and plotting the RMSE over iterations. For this implementation we make use of

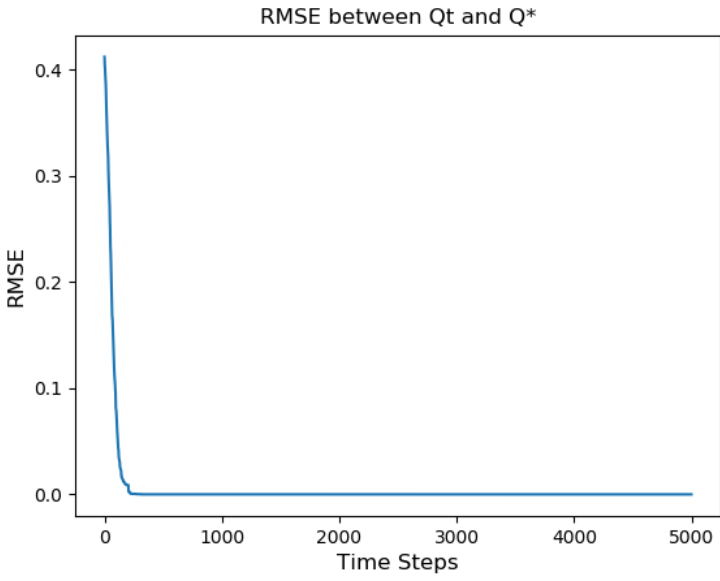
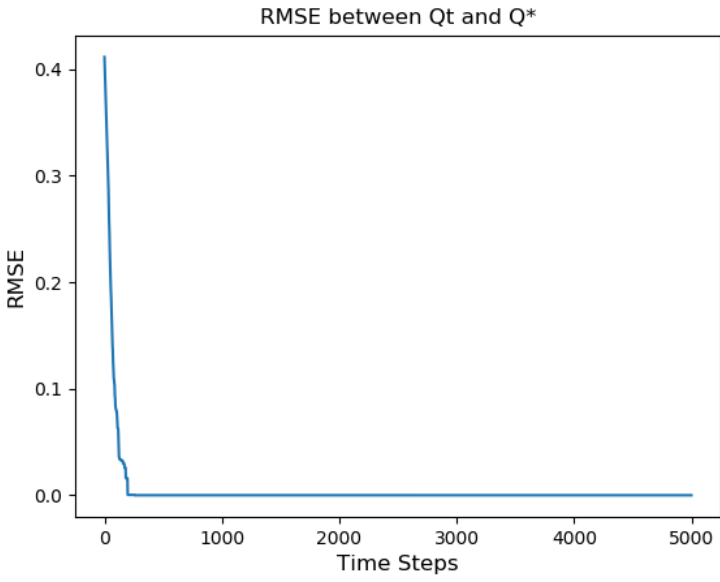
learning rate decay thus the value given below is the learning rate start value and will decay linearly until 0 at the last iteration.

Table 6: learning rate experimentations

Epsilon	Learning Rate	RMSE	Graph
0.2	0.01	2.589e-1	 <p>The graph shows the Root Mean Square Error (RMSE) between Q_t and Q^* over 5000 time steps. The y-axis represents RMSE, ranging from 0.26 to 0.42. The x-axis represents Time Steps, ranging from 0 to 5000. The RMSE starts at approximately 0.41 at time step 0 and decreases steadily, reaching approximately 0.26 by time step 5000.</p>
0.2	0.1	8.739e-5	 <p>The graph shows the Root Mean Square Error (RMSE) between Q_t and Q^* over 5000 time steps. The y-axis represents RMSE, ranging from 0.0 to 0.4. The x-axis represents Time Steps, ranging from 0 to 5000. The RMSE starts at approximately 0.41 at time step 0 and decreases rapidly, reaching near 0.0 by time step 2000 and remaining stable until time step 5000.</p>

0.2	0.2	6.177e-9	<p>RMSE between Q_t and Q^*</p>  <p>The graph displays the Root Mean Square Error (RMSE) between the estimated value Q_t and the target value Q^* over 5000 time steps. The y-axis represents RMSE, ranging from 0.0 to 0.4. The x-axis represents Time Steps, ranging from 0 to 5000. The RMSE starts at approximately 0.42 at time step 0 and decreases rapidly, reaching a value near 0.0 by time step 1000, and remains stable at that level until time step 5000.</p>
0.2	0.3	9.01e-13	<p>RMSE between Q_t and Q^*</p>  <p>The graph displays the Root Mean Square Error (RMSE) between the estimated value Q_t and the target value Q^* over 5000 time steps. The y-axis represents RMSE, ranging from 0.0 to 0.4. The x-axis represents Time Steps, ranging from 0 to 5000. The RMSE starts at approximately 0.42 at time step 0 and decreases rapidly, reaching a value near 0.0 by time step 1000, and remains stable at that level until time step 5000.</p>

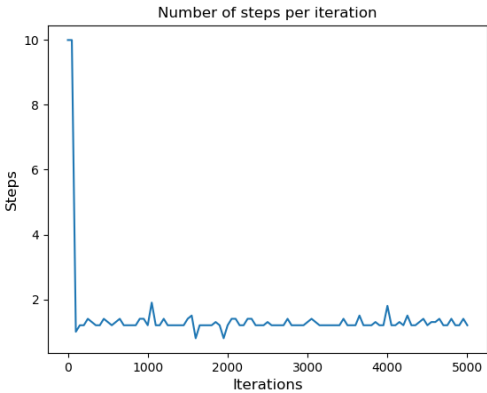
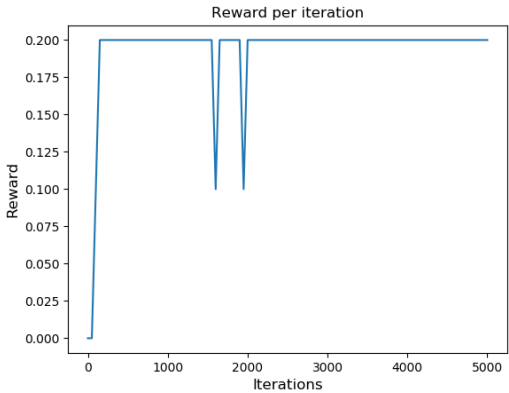
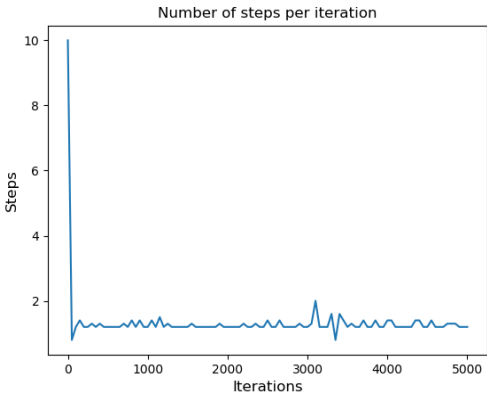
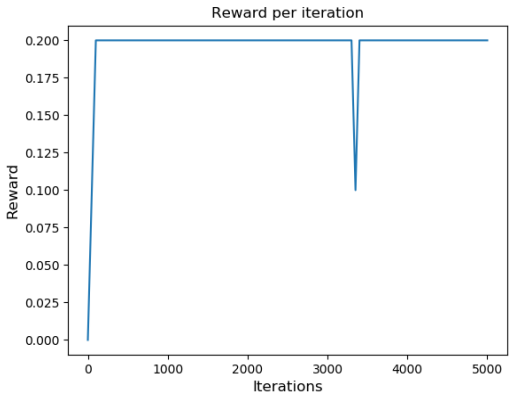
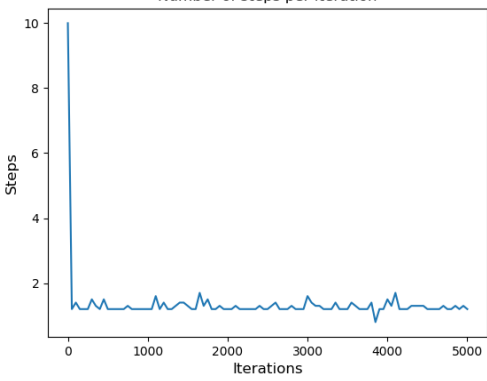
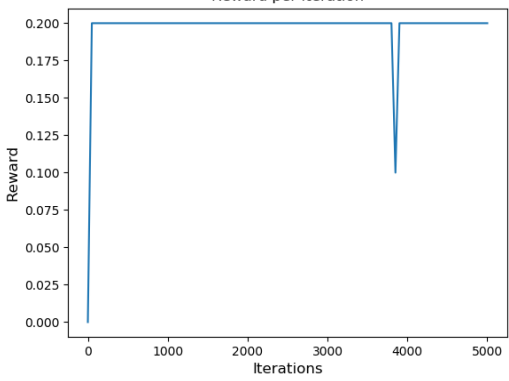
0.2	0.5	2.269e-16	<p>RMSE between Q_t and Q^*</p>  <p>The graph displays the Root Mean Square Error (RMSE) between the estimated value Q_t and the target value Q^* over 5000 time steps. The y-axis represents RMSE, ranging from 0.0 to 0.4. The x-axis represents Time Steps, ranging from 0 to 5000. The RMSE starts at approximately 0.42 at time step 0 and decreases rapidly, reaching a value near 0.0 by time step 500, and remains stable at that level until time step 5000.</p>
0.2	0.7	1.79e-16	<p>RMSE between Q_t and Q^*</p>  <p>The graph displays the Root Mean Square Error (RMSE) between the estimated value Q_t and the target value Q^* over 5000 time steps. The y-axis represents RMSE, ranging from 0.0 to 0.4. The x-axis represents Time Steps, ranging from 0 to 5000. The RMSE starts at approximately 0.42 at time step 0 and decreases rapidly, reaching a value near 0.0 by time step 500, and remains stable at that level until time step 5000.</p>

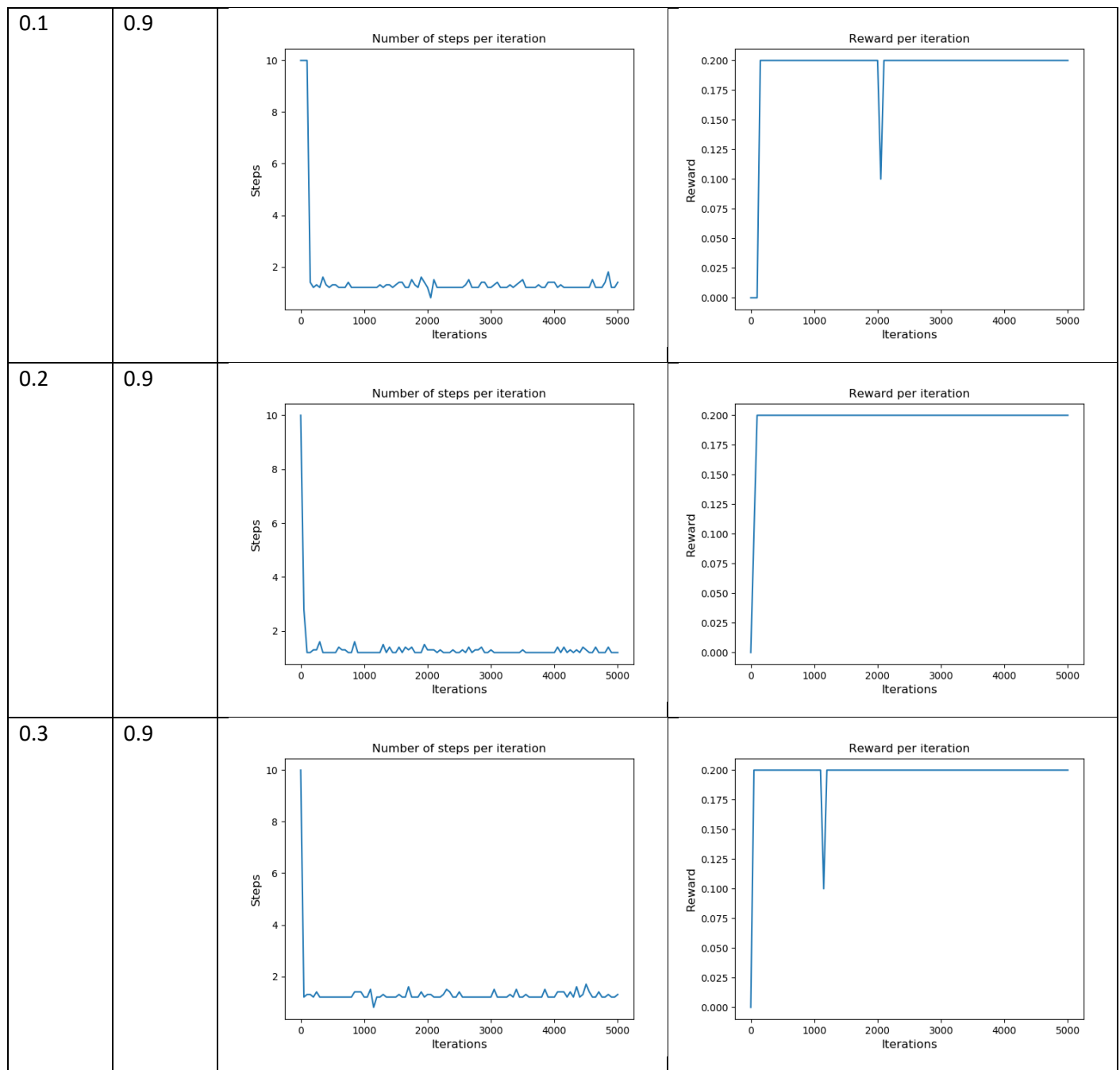
0.2	0.9	1.64e-16	
0.2	1	1.61e-16	

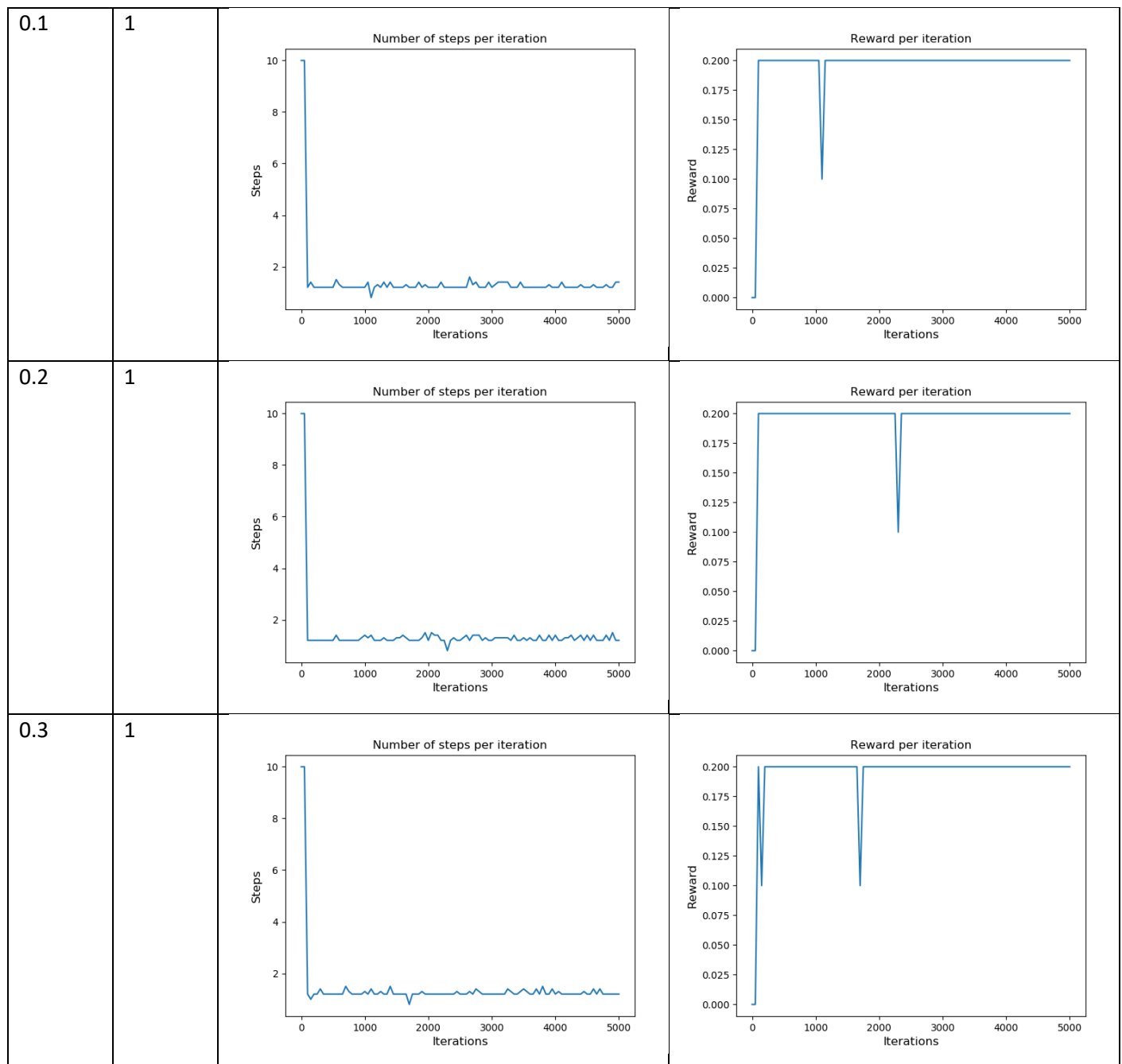
By looking at the above table the Learning rate is chosen as 0.9 although this is the 2nd smallest RMSE which is chosen due to the smoothness of the graph which means its learning is less erratic.

The following table shows the performance evaluations of the different tested epsilon and learning rate values. It shows the average steps over iterations and the average reward over iterations.

Table 7: Evaluation graphs of experiments

Epsilon	Learning Rate	Steps per iteration	Reward per iteration
0.1	0.8		
0.2	0.8		
0.3	0.8		





During the RMSE experiments a range for the optimal epsilon and optimal learning rate was found with epsilon being in the range 0.1-0.3 and learning rate being between 0.8-1. By then observing the above experiment and looking at the performance evaluation graphs the optimal values were chosen as epsilon equal 0.2 and the learning rate as 0.9.

Part 3 Results

The following are the results obtained in Part 3. Using the optimal found hyperparameters.

Mountain Car

The following where the chosen hyperparameters for the mountain car problem.

Table 8: Mountain car hyperparameters

Hyperparameter	Value
Epsilon	0.2
Learning Rate	0.9
Iterations	3000
Number of states	200

The following is the evaluation figures generated by the program.

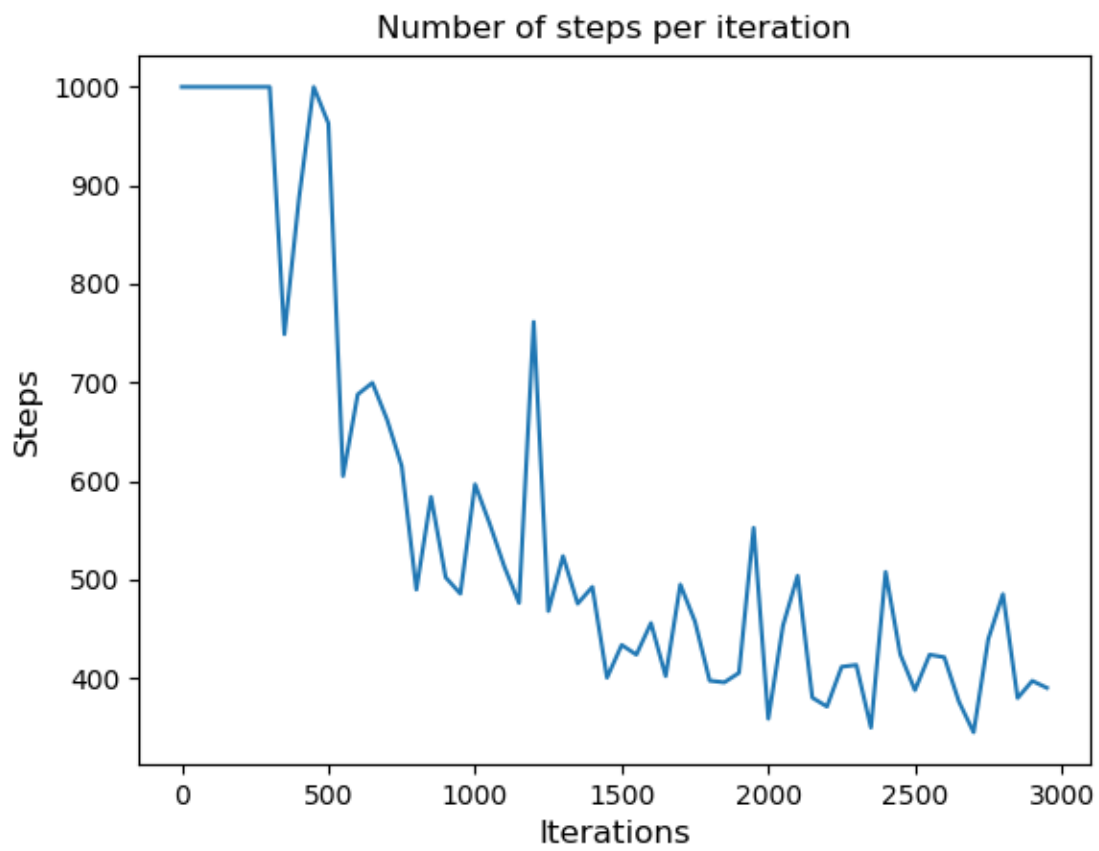


Figure 6: Mountain Car Q-Learning steps per iteration

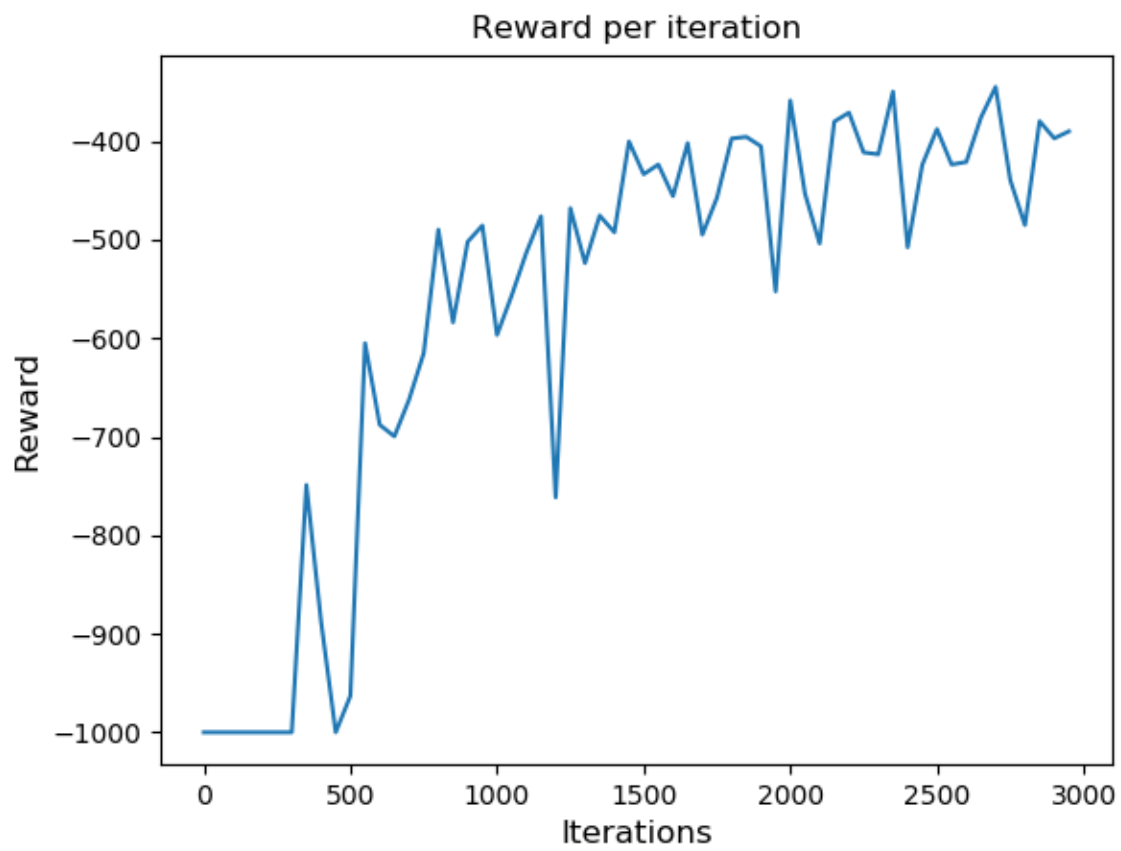


Figure 7: Mountain Car Q-Learning rewards per iteration

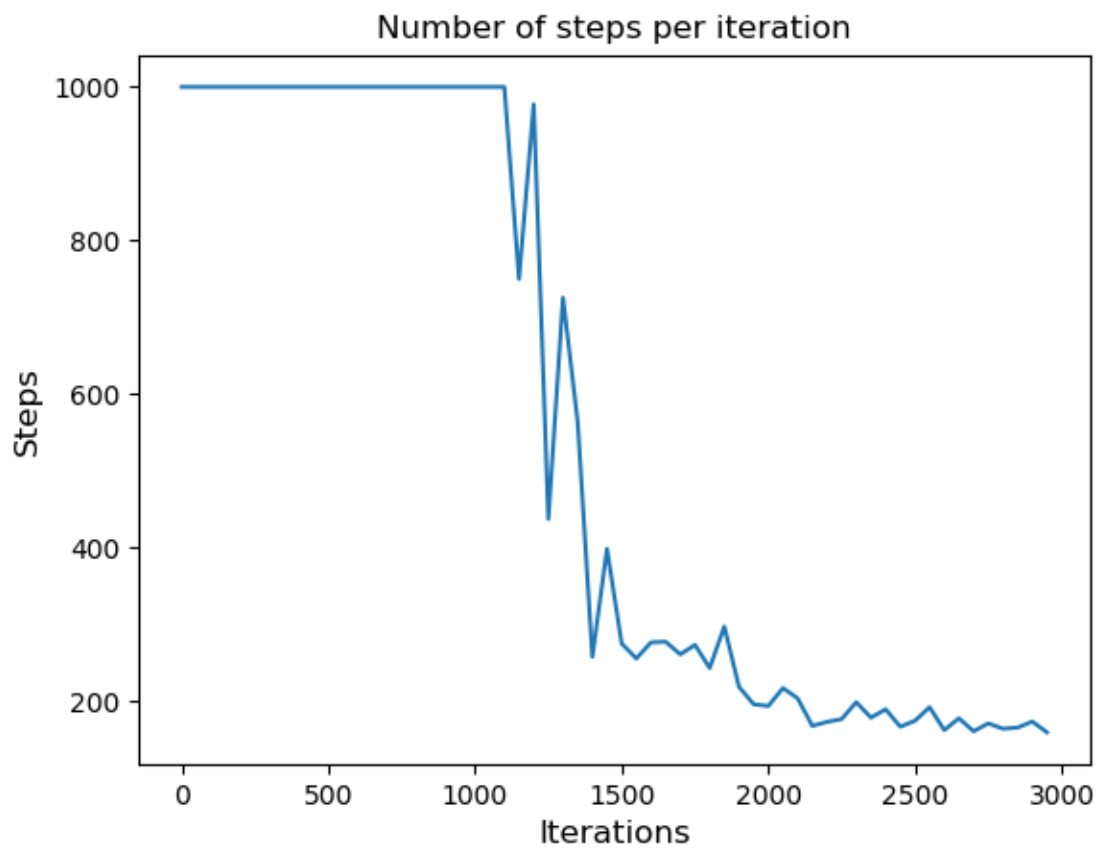


Figure 8: Mountain Car Reinforcement learning steps per iteration

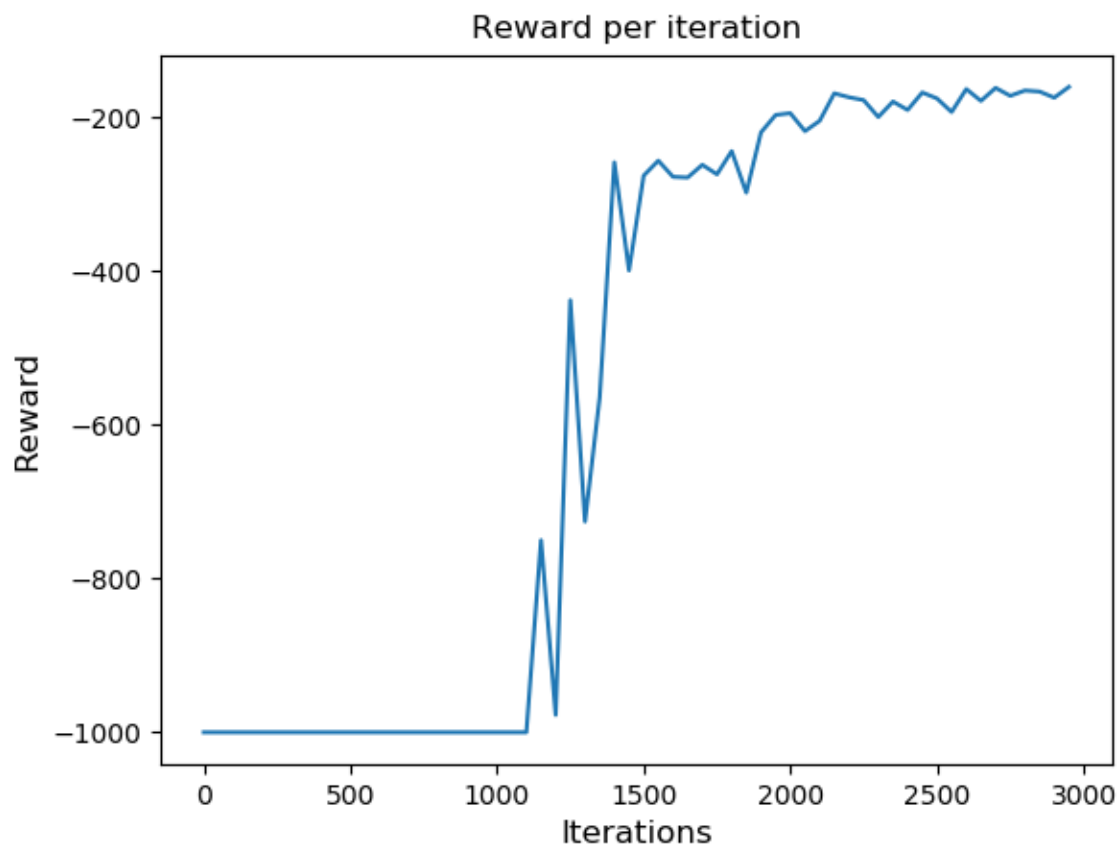


Figure 9: Mountain Car Reinforcement learning rewards per iteration

There is also gifs showing the goal being reached these are included in the folder with this document.

The name of the Q-Learning gif: car_Q_Learning

The name of the Reinforcement learning gif: car_REINFORCE

Acrobatic

The following where the chosen hyperparameters for the acrobatic problem.

Hyperparameter	Value
Epsilon	0.2
Learning Rate	0.9
Iterations	3000

Number of states	30
------------------	----

The following is the evaluation figures generated by the program.

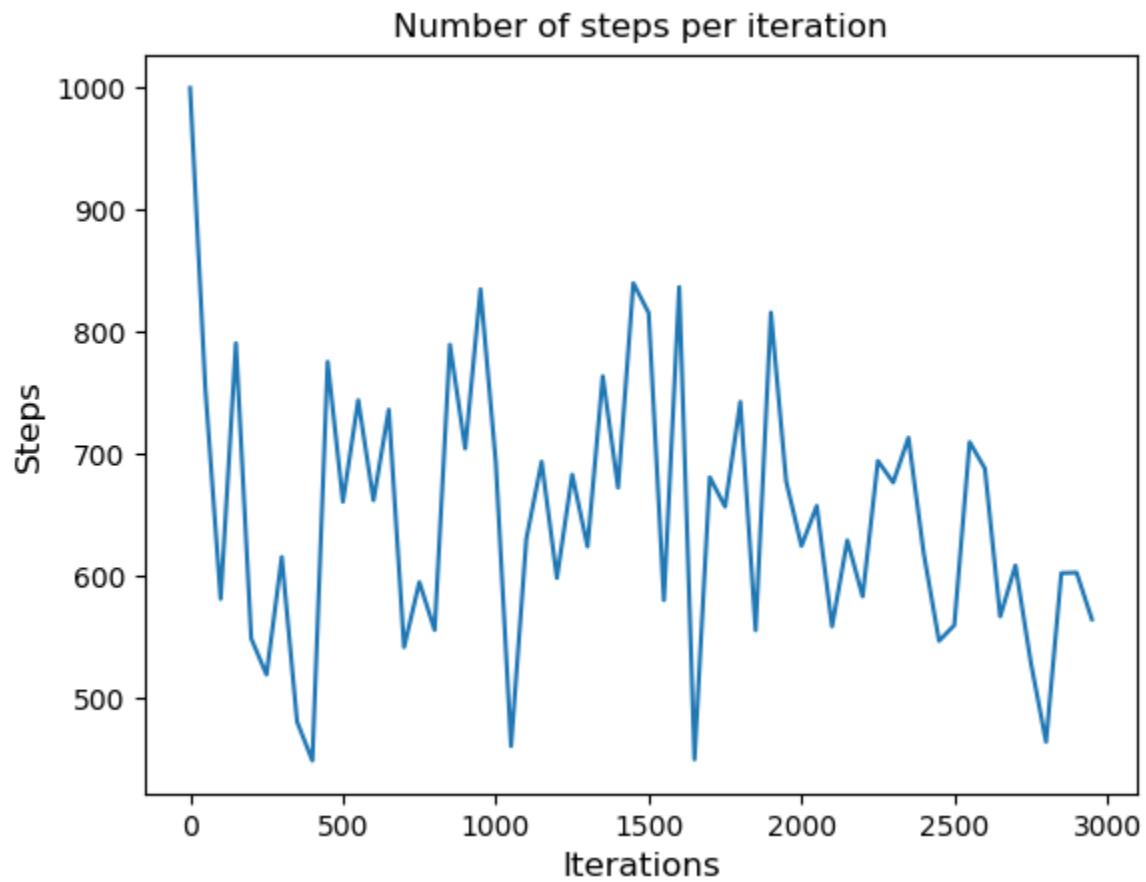


Figure 10: Acrobatic Q-Learning steps per iteration

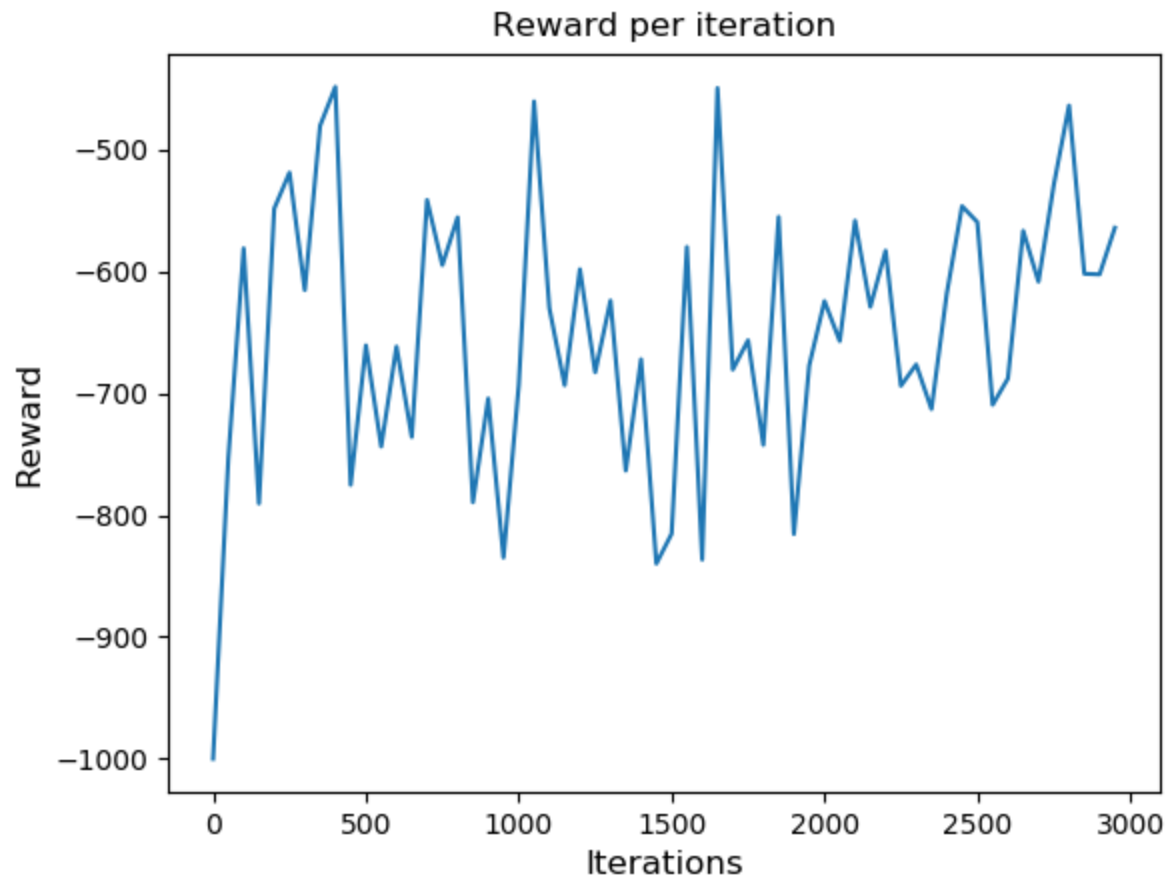


Figure 11: Acrobatic Q-Learning rewards per iteration

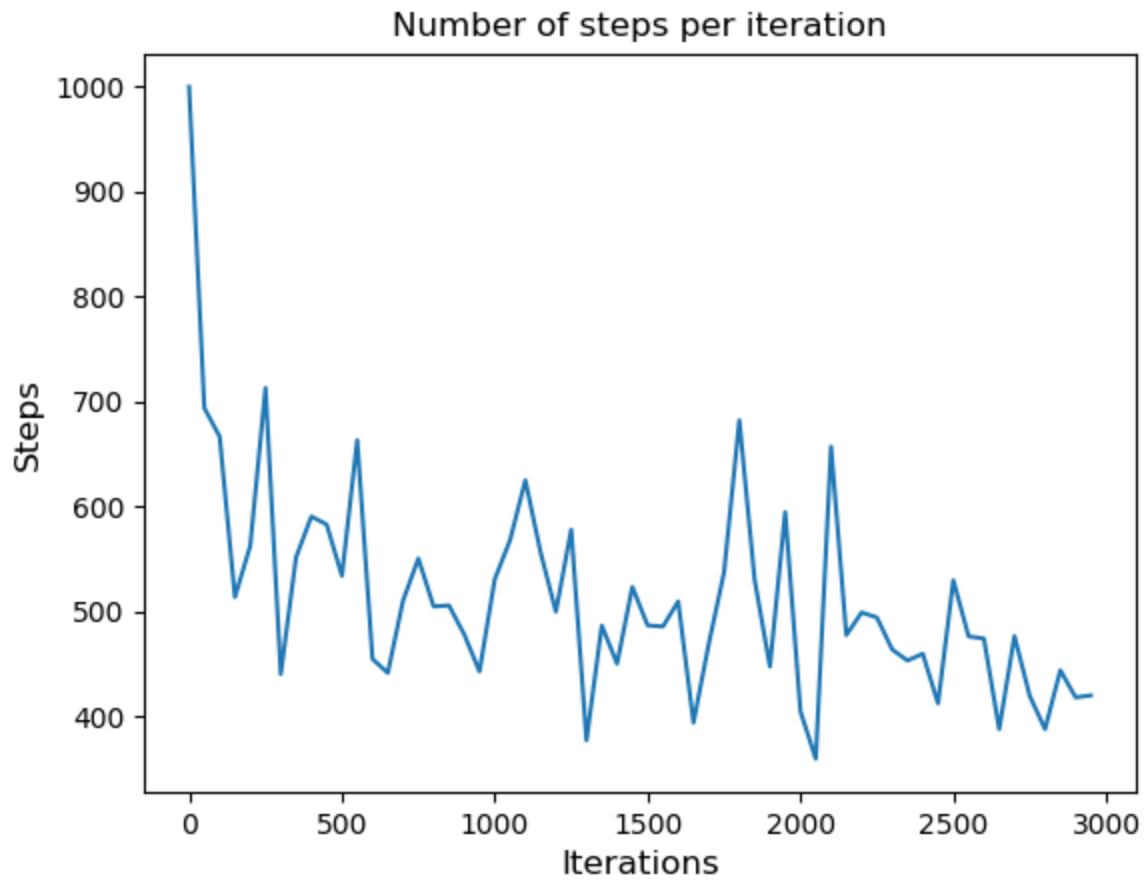


Figure 12: Acrobatic Reinforcement learning steps per iteration

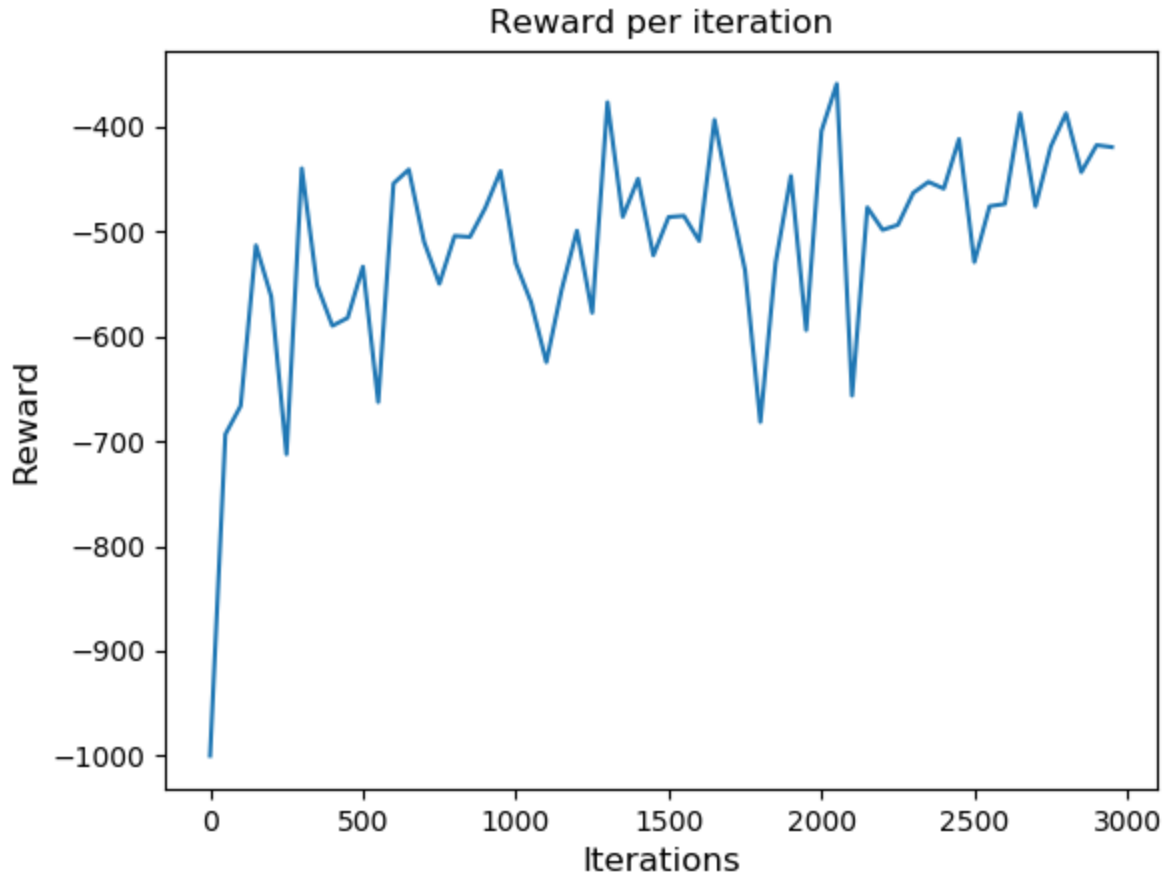


Figure 13: Acrobatic Reinforcement learning rewards per iteration

There are also gifs showing the goal being reached these are included in the folder with this document.

The name of the Q-Learning gif: acro_Q_Learning

The name of the Reinforcement learning gif: acro_REINFORCE

Discussion

The results above show how different methods of learning can be used to solve MDP environments. The following is the discussion of each parts results.

Part 1

In part 1 of the project it is shown that by implementing value iteration it is possible to quickly find the optimal policy and Q values table. The optimal policy allows for the robot to start or be placed into any of

the 112 states and the robot will know the optimal path to follow from that specific state that would maximize its possible reward.

Part 2

Part 2 of the project explores Q-Learning and how it compares to the methods used in Part 1. By experimenting with different Learning rates and epsilon values it was possible to estimate optimal values that allow for the best performance by the algorithm. The epsilon and learning rate chosen at the end was quite large but due to the program using decay on both these values starting with large values have better results due to freedom this gives the program at the start and allowing then for precise changes at the end. A very small RMSE was obtained at the end, this RMSE could shrink even further by increasing the number of steps and decreasing theta at part 1. But the small value obtained was deemed sufficient and thus there is no need to reduce the performance of the program by increasing the iterations.

Part 3

Part 3 of the project requires the use of the same algorithms as used in part 2 with the addition of also solving the problem using reinforcement learning. By looking at the evaluation graphs it is clear that Reinforcement Learning works better than this implementation of Q-Learning. This can be seen by the fact that the Reinforcement Learning algorithm produces higher rewards than that of Q-learning. In order to obtain the hyperparameters that were used to generate the results for this part experiments were done. The experiments done were used to determine the optimal values for the following parameters epsilon, learning rate, iterations and the number of steps. Limited experiments were done to determine the optimal value for epsilon and the learning rate since the optimal ones for the Q-Learning algorithm was already found in the previous part. The more the number of states used the better, but the values chosen were chosen due to memory and processor limitations. The number of iterations were chosen due to evaluation graphs starting to converge at this point. The results for this part of the project were greatly improved by the introduction of decay. The decay functions especially improved the reinforcement learning results by a great margin.

Conclusion

This project shows how MDP environments can be solved using a wide variety of different approaches and methods. By comparing the results obtained by these methods it is clear that all the methods have advantages and disadvantages depending on the requirements of the application. With certain methods

such as value iteration being extremely fast whereas reinforcement learning takes a lot longer. Also depending on the available hardware will depict which methods will work best considering value iteration can be run with low level hardware whereas reinforcement learning performance increases as the complexity increases by increasing things such as states which can become hardware intensive. Possibly even better results could have been obtained when using Q-Learning and Reinforcement learning by performing even more experiments and doing finer tuning on the hyperparameters.

From this project it is clear that given an MDP environment an optimal policy can be found that maximizes rewards.