Solving common control problems using Reinforcement Learning

Owen Agius
Department of Artificial Intelligence
Faculty of Information and Communication Technology
University of Malta
owen.agius.19@um.edu.mt

1 Introduction

Back in 1937, a psychologist named B.F Skinner coined the term Operant Conditioning [17]. In essence, operant conditioning is the learning process in which animals or humans learn certain behaviours based on a positive-negative reward-punishment basis. But it did not take long for computer scientists to develop Skinner's theory as a building block of machine learning through Watkins' discovery of the Q-Learning algorithm in 1989, greatly improving the feasibility and practicality of reinforcement learning [20]. Stemming from traditional machine learning, reinforcement learning (RL) revolves around the training of models to make a sequence of decisions [7]. In reinforcement learning, an artificially intelligent agent attempts to achieve a goal by constantly learning in a completely alien, game-like environment. The agent in the environment learns through a trail-and-error approach. The artificial intelligence model influences the agent by rewarding either rewards or penalties based on the actions it performs, where the goal of the agent is to maximise the rewards as much as possible. Despite the developer setting the reward-punishment policy of the environment, it is completely in the hands of the model to figure out how to carry through in achieving the rewards.

The machine learning spectrum can be described as a product of supervised learning, unsupervised learning and reinforcement learning [16]. In supervised learning, the learning within the system is reliant on the training data, where the training data consists of a labeled pair on inputs and outputs [13]. The approach is coined as *supervised* because the training data acts as a supervisor which is guides the model from the labels of the inputs and outputs of the dataset. Moreover and similarly to supervised learning, in unsupervised learning the model is also reliant on the training data provided. But the main difference is that the data in the training set is not labeled, which forces the model to search for hidden patterns in the input. This is where reinforcement learning differs from other machine learning approaches. RL is not based off of training sets but it is based off of an algorithm where the model constantly receives feedback from the user through rewards or punishments.

Reinforcement learning has seen many improvements and enhancements over the years, rendering the machine learning approach as a viable implementation for many companies such as Facebook, Google and Amazon, among many others. Modern RL applications can be found in self-driving cars. Developed by Amazon, AWS DeepRacer [1] is an autonomous racing car that uses cameras to visualize the racetrack alongside an RL model which controls the throttle and direction of the vehicle. In industry reinforcement, RL based robots are used to perform a variety of tasks. Particularly, Google use Deepmind [11] in order to cool their data centers, which reduced a total of 40% of their energy expenditure. In the engineering field, Facebook make use of Horizon [6] which is capable of handling internal tasks. Internally, Horizon is used to personalize suggestions and deliver more suitable notifications. Considering a more multi-disciplinary approach, RL usage was explored by Stanford University with Microsoft Research to develop DeepRL [9] which is able to model future rewards in a chat-bot dialogue environment.

The aim of this experiment is to both explore and compare popular reinforcement learning algorithms, such as Q-Learning and SARSA on standard and non-complex OpenAI Gym [3] environments, being Taxi-v3 [19] and LunarLander-v2 [5]. Additionally, a variety of bandits will be tested and evaluated against each other.

2 Background

2.1 Reinforcement Learning Algorithms in Armed and Contextual Bandits

2.1.1 Multi-Armed Bandits

In reinforcement learning multi-armed bandit problems are generally some of the more straight-forward problems to solve. Consider an agent that has the power to pick actions, and each action has a reward that is distributed according to a certain probability distribution. The purpose of the game is to maximize your reward over a number of episodes (in this case, single acts) [18]. Consider a trivial example of a slot machine with k arms. Each time a specific arm is pulled, a specific reward is returned. The aim of the agent is to maximise how many rewards are pulled.

$Q_t(a) = \exists[R_n|A_n = a]$

More formally, whenever the estimate $Q_t(a)$ is estimated, an expected reward R_n and action A_n are taken at timestep n. For all of the underlying probability distributions that will be looked at, a Gaussian distribution will be used so that the mean corresponds to the true value. The simplest way to proceed is to perform a greedy action, or taking the action which is assumed to maximise the reward. This is done by swapping A_n to $max_a(Q_n(a))$.

2.1.2 *Contextual Bandits*

Contextual Bandits are commonly regarded as an extension of multi-armed bandits. The contextual bandit adds to the model by making the decision dependent on the current state of the environment 1.

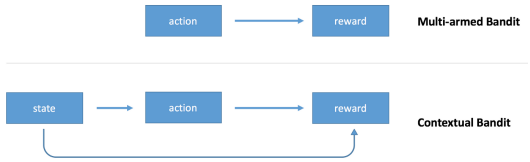

Figure 1: (https://medium.com/emergent-future/simple-reinforcement-learning-with-tensorflow-part-0-q-learning-with-tables-and-neural-networks-d195264329d0)

The algorithm analyzes the situation, makes a decision from a set of options, and then monitors the outcome of that decision. A reward is determined by the outcome. The goal is to increase the average reward as much as possible. For example, to improve click through rate, you can employ a contextual bandit to choose which news article to display first on your website's main page. The context is information about the user, such as where they came from, which pages of the site they previously visited, device information, location, and other relevant information. A choice of which news article to display is an action. A result is whether or not the user clicked on a link. A reward is binary: 0 if no click is made, 1 if one is made.

2.1.3 *Full RL Problem*

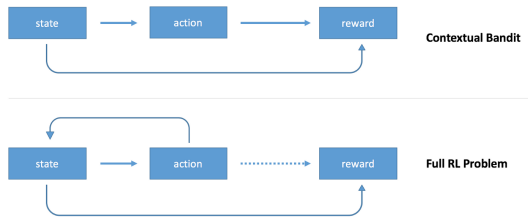

Figure 2: (https://medium.com/emergent-future/simple-reinforcement-learning-with-tensorflow-part-0-q-learning-with-tables-and-neural-networks-d195264329d0)

Additionally, reinforcement learning as an ideology can be viewed as an extension of the contextual bandit implementation 2. An agent which takes actions relevant to the state and rewards the policy is still present. The distinction is that the agent can perform numerous acts in a row, and reward data is scarce.

2.2 The Four Main Elements of Reinforcement Learning

A reinforcement learning algorithm cannot be implemented without its four main elements being the policy, the reward signal, the value function as well as the model of the environment. The policy is a strategy which the RL agent utilizes in the aim of achieving as many goals as possible. In essence, the policy highlights the actions which the agent must take as a function of the relative state of the agent within the environment. Formally, a policy can be defined as the policy $\pi(s)$ that accommodates the actions the agent should perform for every state in $s \in S$.

The whole gist of RL is made through the use of the reward signal. In a reinforcement learning problem, the goal is defined by a reward signal. The environment gives a single number, a reward, to the reinforcement learning agent at each time step. The agent's main goal in the long run is to maximize the overall reward it receives. The agent's favourable and undesirable events are thus defined by the reward signal. In a biological system, we might think of rewards as being either positive or punishments. They are the agent's direct and distinguishing characteristics of the dilemma.

Naturally, the algorithm cannot distinguish a positive or negative scenario by itself, and this is where the value function comes into play. The value function is used in order to determine *how good* a situation can be for the RL algorithm. Making reference to the policy π which is a mapping of all of the possible states and all of the possible actions to the probability $(\pi(s, a))$ of taking an action a when in state s. Mathematically, the value of a state under a policy is the expected return when starting in the state s and following the probability π. Therefore, using this reasoning the value function can be formally defined as:
$V^{\pi}(s) = E_{\pi}\{R_t|s_t = s\} = E_{\pi}\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1}|s_t = s\}$ within a time-step t.

Finally, the model of the environment is responsible for the prediction of the outcomes of the actions $a \in A$ that are used in correspondence or in addition to the reciprocal action with the environment to learn the optimal policies [14].

2.3 Key Bandit Algorithms

Multi-Armed Bandits and Contextual Bandits cannot learn by themselves, but with the help of a bandit algorithm, this is made possible. A variety of bandit algorithms are highlighted in Experiment 1, which will be discussed here, namely, Thompson Sampling, LinUCB, Epsilon-Greedy and Non-Contextual UCB. Kicking things off with the LinUCB [8]. The Linear Upper Confidence Bound algorithm is made up of two main building blocks. The first block is that the algorithm maintains estimates of the parameters of every arm with 'Linear Least Squares'. Secondly, the other block maintains confidence ellipsoids defined as the inverse covariance of the estimates acquired in the previous block. LinUCB's fundamental concept is "Optimism in the Face of Uncertainty." The agent includes exploration by increasing the variance of the estimates by an amount equal to the variance of the estimations. The confidence ellipsoids enter the scene at this point.

Thompson sampling (TS) is another popularly used algorithm to overcome the exploration-exploitation dilemma. The basic idea is to treat the average reward μ from each bandit as a random variable and use the data we have collected so far to calculate its distribution. Then, at each step, we will sample a point from each bandit's average reward distribution and select the one whose sample had the highest value. We subsequently get the reward from the selected bandit and update its average reward distribution.

Finally, the epsilon greedy method. A pure Greedy approach has a very high chance of choosing a sub-optimal socket and sticking with it. As a result, finding the best socket will be impossible. Adding an element of exploration

is an easy method to solve this problem. Epsilon-Greedy performs just that: Actions are chosen greedily by default. The selected action is the one with the highest estimated benefit. At each time step, however, an action may be chosen at random from the set of all potential actions. A probability is used to select a random action. Exploration is introduced to the basic Greedy algorithm in this fashion. Every action will be sampled several times throughout time to provide a more accurate assessment of its genuine reward value.

2.4 The Bellman Expectation and Optimality Equations

When a policy is adopted, the value of a specific state is defined by the immediate reward plus the value of successor states, according to the below equation definition of the Bellman Expectation Equation.

$v_\pi(s) = E_\pi[R_{t+1} + \gamma v_\pi(S_{t+1}|S_t = s]$

The state-action Value function may be expressed as below. From the definition below it can be seen that a state's State-Action Value can be broken down into the immediate reward we receive for performing a certain action in state(s) and moving to another state(s') plus the discounted value of the state-action value of the state(s') with respect to some action(a) our agent will take from that state onwards [2].

$q_\pi(s, a) = E_\pi[R_{t+1} + \gamma q_\pi(S_{t+1}, A_{t+1})|S_t = s, A_t = a]$

The Bellman Optimality Equation (as viewed below) defines a time value *t*, for any state-action pair *(s,a)* expects the return from a starting state s by taking any action a which with the optimal policy equation $q_*(s', a')$ will be equal to the expected reward in R_{t+1}. This equation is essential in as it can aid in fitting a reinforcement learning algorithm which finds the action a which maximises the optimal policy π as much as possible. As can be viewed from the equation, the optimal policy equation is used recursively within the Bellman Optimality Equation

$q_*(s, a) = E[R_{t+1} + \gamma max q_*(s', a')]$

2.5 Model Based and Model Free Approaches

Traditional reinforcement learning systems can either be model-based or model-free. In a model-based reinforcement learning environment, the policy is directly affected through the output of a machine learning model. In this scenario, a model-based reinforcement learning environment is one that employs machine learning models such as random forest, gradient boost, neural networks, and others. A model-free RL system, on the other hand, does not have a policy based on the use of machine learning models; instead, its policy is directed by the use of non-ML techniques. For example, consider balancing a lever system that assures optimal effort and load stability when on the fulcrum. A simple algorithm may be built to ensure that the fulcrum moves left when the load exceeds the effort, and right when the load exceeds the effort [12]. A model-free system coins its name due to the absence of a machine learning model in the stability of the environment.

2.6 Value Function Approximation

In reinforcement learning, as problems get larger and more complex, it is quite common for a developer to encounter situations with extremely large problems or state spaces. This issue gives light to two main problems, where the system would have too many states or actions to store. But if storage is not a problem for the developer, the estimation of each separate value would be too computationally expensive for a feasible implementation. The value function approximation is the solution to this problem. A value function approximation application attempts to build a function in which the estimate of the true value functions is calculated. This value is calculated by creating a compact representation of the value function which uses a smaller amount of parameters.

$v'(s, w) \simeq v_\pi(s) \quad or \quad q'(s, a, w) \simeq q_\pi(s, a)$

The weights of the neural network are the vector of weights w that will be used to estimate the value function throughout the whole state/state-action space, which is a standard technique. The methods we've seen before, in Temporal-Difference learning, will be used to update this vector of weights. In a RL application the true value function is obviously unknown at first. Therefore it is substituted into a target. In Monte Carlo Learning the target is defined as $\triangle w = \alpha(G_t - v'(S_t, w)) \bigtriangledown_w v'(S_t, w)$ .Whereas, in TD Learning, the target is defined as $R_{t+1} + \gamma v'(S_{t+1}, w)$ therefore an update, by substituting the temporal difference target, it can be defined as:

$\triangle w = \alpha(R_{t+1} + \gamma v'(S_{t+1}, w) - v'(S_t, w)) \bigtriangledown_w v'(S_t, w).$

2.7 Value Based and Policy Gradient Approaches

Value Based methods and Policy Gradient methods are both theoretically derived from the Markov Decision Process construct, therefore as a result both use similar concepts. However, the two approaches are different internally. The most significant distinctions between the techniques are in how they approach action selection both during learning and as a result. The purpose of value-based approaches is to discover the largest value to learn a single deterministic action from a discrete sequence of activities. The purpose of policy gradients and other direct policy searches is to discover a stochastic map from state to action that works in continuous action spaces. Resultantly, policy gradient implementations can solve problems which are impossible for value-based methods (such as Q-Learning to solve):

- Stochastic Policies
- Large and Continuous Action Spaces

However, value-based methods still keep an edge over policy gradient methods when considering the speed and simplicity of the model execution.

2.8 Evolutionary Methods to Reinforcement Learning

Recently a shift from traditional reinforcement learning to more evolutionary methods started to be explored. Sutton and Barto [18], as well as Kaelbling, Littman, and Moore [7], give useful overview of the area of reinforcement learning . They distinguish between techniques that search the space of value functions and methods that search the space of policies when it comes to reinforcement learning. The temporal difference learning is an example of the former, whereas the evolutionary approaches approach is an example of the latter [10]. Deriving from Darwin's theory of evolution [4], evolutionary methods in reinforcement learning grasp the concept of natural selection in the aim of building a global search. Evolutionary methods are general-purpose search techniques that have been used in a wide range of applications, including numerical function optimization, combinatorial optimization, adaptive control, adaptive testing, and

machine learning [15]. One of the reasons for the increased popularity of evolutionary methods is that there are minimal criteria for their use. An evolutionary method iteratively updates a population of possible solutions, which are frequently encoded in chromosomes. The evolutionary method assesses solutions and develops offspring depending on the fitness of each solution in the task environment throughout each repetition, referred to as a generation. The solutions' substructures, or genes, are then altered via genetic operators like mutation and recombination. The notion is that structures associated with successful solutions might be modified or merged in following generations to produce even better answers [10].

3 Methodology

3.1 Experiment 1

For the first experiment, the objective was to investigate whether adding context to the problem improved the long-term rewards. To validate this experiment multiple bandit algorithms (discussed in Chapter 2.3) being LinUCB, non-contextual UCB, Thompson Sampling and epsilon-greedy methods were compared to the LinUCB implementation.

The linear UCB contextual bandit, where the payout is believed to be a linear function of the context features, was created using the UCB algorithm, which is popular in the MAB (With a MAB approach, the variants are treated similarly) environment. A simulation experiment with varied alpha values of a LinUCB policy was also conducted.

All of the algorithms surrounding experiment 1 were all carried out in the context of the dataset provided, in which the arms and the rewards were segmented into separate lists in pre-processing.

3.2 Experiment 2 - Taxi-v3

The Taxi-v3 environment was introduced by [19] in order to highlight some issues that could be found in hierarchical reinforcement learning. The OpenAI Gym environment is host to 4 different locations in a small map where the job of the agent is to pick up a passenger at a location x and drop the passenger off at a location y. Generally, when tackling the Taxi-v3 experiment, the agent is rewarded 20 points for a successful drop-off with a 1 point deduction for every time-step used in travelling. Additionally, the agent is deducted 10 points for either illegal pick-ups or drop-offs.

In this experiment, the Taxi-v3 environment was handled using a Temporal Difference Learning (TD Learning) approach. Unsupervised TD learning predicts the predicted value of a variable in a sequence of states. TD Learning employs a mathematical method to substitute sophisticated future thinking with a straightforward learning procedure that achieves the same outcomes. Rather than computing the whole future reward, TD Learning seeks to forecast a mix of immediate reward and its own reward prediction for the following time interval. Formally, the main idea of TD Learning is the discounted return (as defined below). Where the reward at a time-step t is the summation of the future discounted rewards.

$R_t = r_{t+1} + yr_{t+2} + y^2 r_{t+3} + \cdots = \sum_{k=0}^{\infty} y^k r_{t+k+1}$

Moreover, the error metric, TD Error (defined below), is the difference between the best correct reward and the current prediction.

$E_t = V_t^* - V_t$

$E_t = r_{t+1} + \gamma * V_{t+1} - V_t$

Where the current value is iteratively updated by its value added by the learning rate and multiplied by the error.

Temporal Difference Learning approaches are reliant on three main parameters, the learning rate α, the discount rate γ and the rate of exploration ϵ. Starting off with the learning rate, this parameter indicates how much the inaccuracy should affect our estimates. The learning rate ranges from 0 to 1. A high learning rate adapts quickly and may result in erratic training outcomes rather than convergence. A lower learning rate adapts slowly, therefore convergence will take longer. With γ we are setting up how much the agent will value future rewards. Similarly to the learning rate, the discount rate is also set between 0 and 1. Moreover, the larger the γ value, the higher the valuation of future rewards. Finally, the ϵ value helps us explore new options with a probability value of ϵ whilst still staying at the current maximum defined with probability *1-*ϵ. Naturally, the larger the ϵ value, the greater the exploration while training.

In order to achieve temporal difference learning, the Q-Learning and the SARSA agents were implemented. Both agents store the rewards of the current state along with the respective action for future reference. But the difference between the two is very subtle and minimal. Q-Learning considers the optimal scenario depending if you get into the next state or not whereas the SARSA agent considers the reward if and only if the current policy is followed into the next state.

More formally, in Q-Learning (defined below) when passing the reward for the next state (s',a') to the current state, the maximum possible reward of the new state s is taken in order to ignore any policy which is being used.

$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma max_a Q(S_{t+1}, a) - Q(S_t, A_t)]$

On the other hand, SARSA (defined below) still follows the policy (epsilon-greedy) when computing the next state a' and passing the reward relative to the a' in the previous step.

$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)]$

3.3 Experiment 3 - LunarLander-v2

In the LunarLander-v2 [5] environment, the landing pad is always located at the coordinates (0,0). The first two digits in the state vector are the coordinates. Moving from the top of the screen to the landing pad at zero speed earns you from 100 to 140 points. If the lander goes away from the landing pad, it forfeits the prize. If the lander crashes or comes to a stop during the episode, you will receive an extra -100 or +100 points. Each leg's ground contact is worth ten points. The cost of firing the primary engine is -0.3 points every frame. It is possible to land outside of the landing pad. Because fuel is boundless, an agent may learn to fly and land on the first try. There are four options: do nothing, fire the left orientation engine, fire the main engine, and fire the right orientation engine.

The handling of the LunarLander-v2 environment was taken care of using value function approximation (highlighted in Chapter 2.7. The alterations made to the main equation of value function approximation is the usage of different targets within the implementation. The experiment uses both Q-Learning and SARSA targets which are defined as the respective following.

$G = r + \gamma max_{a'} Q(s', a')$

$G = r + \gamma Q(s', a')$

The parameters used within the implementations of this experiment consisted of α and the discounting factor γ. But this experiment was also tackled using an evolutionary algorithm in the Cross Entropy Method (CEM). The CEM starts off by takeing a variety of inputs, look at the outputs, pick the inputs that resulted in the best outputs, then tweak them till you are happy with the results. This process can be better explained by viewing 3.

Initialize $\mu \in \mathbb{R}^d, \sigma \in \mathbb{R}^d$
for iteration $= 1, 2, \ldots$ **do**
  Collect n samples of $\theta_i \sim N(\mu, \text{diag}(\sigma))$
  Perform a noisy evaluation $R_i \sim \theta_i$
  Select the top p% of samples (e.g. $p = 20$), which we'll call the **elite set**
  Fit a Gaussian distribution, with diagonal covariance, to the elite set, obtaining a new μ, σ.
end for
Return the final μ.

Figure 3: Pseudocode of Cross Entropy Method. (Source: MLSS 2016 on Deep Reinforcement Learning by John Schulman)

4 Results and Discussion

4.1 Experiment 1

4.1.1 Thompson Sampling

The Thompson Sampling algorithm was fed the arms and the probabilities extracted from the dataset provided. With this in mind, over 2000 trails 4, the bandit algorithm returned a total reward of 2000 with an overall win-rate of 1.0. Additionally, the number of times each bandit was selected is as follows: [119, 106, 215, 202, 260, 250, 431, 52, 345, 20]. But when iterating in 4000 trails 5, the bandit selection alters quite heavily [10, 273, 1390, 1142, 211, 320, 299, 92, 79, 184] whilst still keeping the same win-rates and rewards earned.

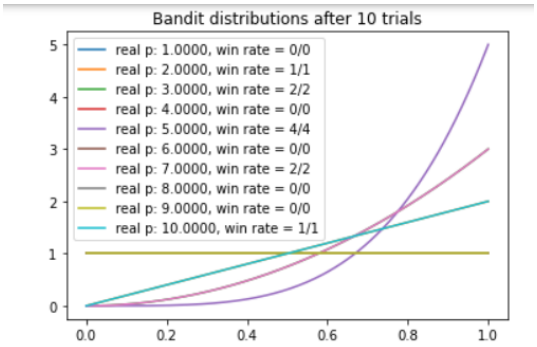

Figure 4: Thompson Sampling 2000 Iterations. (Source: Owen Agius)

4.1.2 Non-Contextual UCB

Over 2000 trails, the Non-Contextual UCB performed just shy of the other implementations in Epsilon Greedy and Thompson Sampling. The Non-Contextual UCB returned 1919 as the total reward accrued alongside a 0.95 win-rate. When testing out with 4000 trails, the win-rate increases slightly to 0.97 alongside a natural increase in the total reward accrued in 3911.

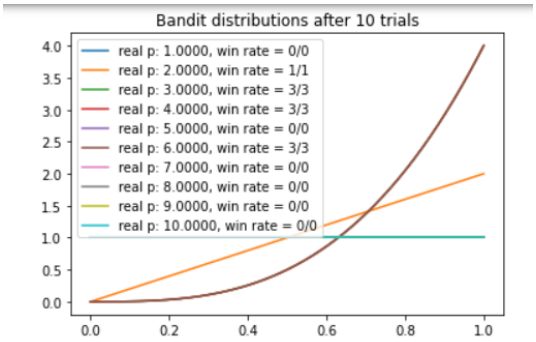

Figure 5: Thompson Sampling 4000 Iterations. (Source: Owen Agius)

4.1.3 Epsilon Greedy

Similarly to Thompson Sampling, the Epsilon Greedy implementation selected bandit 10 as the optimal bandit of the solution over 2000 trails. Moreover, the bandit algorithm also returned a total reward of 2000 alongside an overall win-rate of 1.0. Additionally, for more information regarding the optimal bandit. The bandit was explored 398 times (slightly higher than the Thompson Sampling approach) and was exploited 1602 times. When iterating over the epsilon greedy implementation for 4000 iterations, the optimal bandit remained the same, in contrast to the Thompson Sampling implementation.

4.1.4 LinUCB

Compared to the other bandit implementations, the LinUCB performs quite poorly when considering the amount of total rewards and the win-rate. Several test cases took place for the LinUCB algorithm which tests the implementation out on varying values of alpha. As can be viewed in the images 6 and 7 it can be concluded that the LinUCB has a tendency of performing with greater results the lower the alpha is during runtime. As alpha is 2, the win-rate is outputted at 0.36 but as alpha is set as 1, the win-rate is outputted as 0.469.

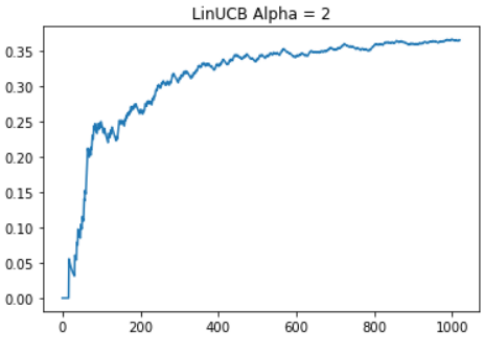

Figure 6: LinUCB with Alpha Value 2. (Source: Owen Agius)

Despite, not being illustrated, more test cases were taken with varying alpha, the results will be listed below:

- Alpha = 1.5, Winrate = 0.41

- Alpha = 0.5, Winrate = 0.39

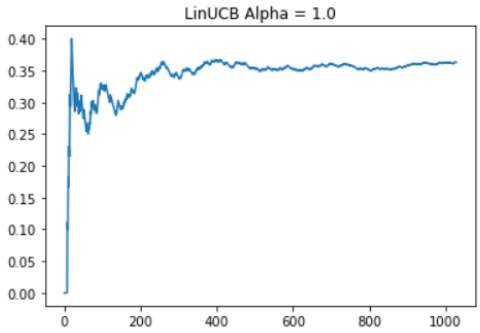

Figure 7: LinUCB with Alpha Value 1. (Source: Owen Agius)

4.2 Experiment 2 - Taxi-v3

4.2.1 Q-Learning

Starting off with Q-Learning, the first test case started off with the hyper-parameters set as:

- Epsilon = 0.00001
- Gamma = 0.95
- Learning Rate = 1

over 5000 episodes iterating over 200 test-episodes. The Q-Learning implementation took just slightly over 7 minutes to iterate over all of the episodes, but the environment was not solved. Curiously, after around 1500 episodes, the average reward remained stagnant being seemingly bound by the 7.5 and 8.2 values.

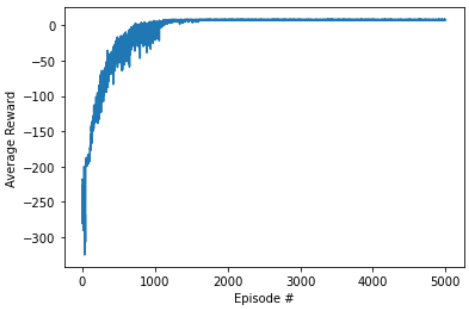

Figure 8: Unsuccessful Average Reward Graph of Q-Learning. (Source: Owen Agius)

Table 1: Q-Learning Unsuccessful Attempt

Epsilon Value	Gamma Value	Learning Rate
0.00001	0.95	1
Episodes	Test Episodes	Time Taken
5000	200	7m:28s

Continuing with Q-Learning, the second test case kicked off with the hyper-parameters set as:

- Epsilon = 0.000000001
- Gamma = 0.95
- Learning Rate = 0.69

over 20000 episodes iterating over 200 test-episodes. This step was taken in order to test out whether decreasing the gradient for the learning rate alongside greatly increasing the number of episodes would solve the environment or not. The Q-Learning implementation took just slightly over 23 minutes to iterate over all of the episodes, but the environment was still not solved. Similarly to the first test case, after around 1500 episodes, the average reward remained stagnant being seemingly bound by the 7.5 and 8.2 values.

Table 2: Q-Learning Unsuccessful Attempt

Epsilon Value	Gamma Value	Learning Rate
0.000000001	0.95	0.69
Episodes	Test Episodes	Time Taken
20000	200	23m:16s

4.2.2 SARSA

Starting off with SARSA, the first test case started off with the hyper-parameters set as the first test case in the Q-Learning implementation:

- Epsilon = 0.00001
- Gamma = 0.95
- Learning Rate = 1

over 5000 episodes iterating over 200 test-episodes. The SARSA implementation took just slightly over 5 minutes to iterate over all of the episodes, but the environment was not solved. Similarly to the Q-Learning test case of these parameters, after around 1500 episodes, the average reward remained stagnant being seemingly bound by the 7.5 and 8.2 values.

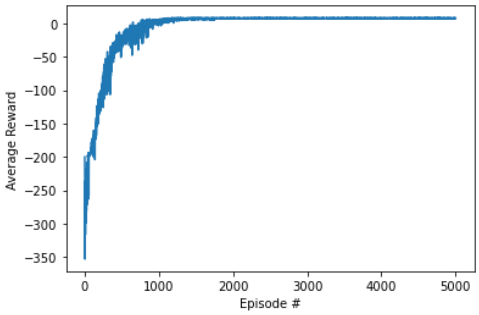

Figure 9: Unsuccessful Average Reward Graph of SARSA. (Source: Owen Agius)

A different concept was tested out in the second test case with a simple increase in the episode count of the implementation. With 40000 episodes, the environment was still not solved. But, the system behaved differently from any other test case seen up to now. After around 20000 iterations, the SARSA algorithm seemingly started to get confused and large dips and spikes in the average score could be viewed.

Naturally, as the episode count increased greatly, the time taken to iterate increased as well with around 32 minutes to execute.

Table 3: SARSA Unsuccessful Attempt

Epsilon Value	Gamma Value	Learning Rate
0.00001	0.95	1
Episodes	Test Episodes	Time Taken
5000	200	5m:44s

Table 4: SARSA Unsuccessful Attempt

Epsilon Value	Gamma Value	Learning Rate
0.001	0.95	1
Episodes	Test Episodes	Time Taken
40000	200	31m:28s

The same approach was taken as the second test case of the Q-Learning implementation with the parameters set as so:

- Epsilon = 0.000000001
- Gamma = 0.95
- Learning Rate = 0.69

over 20000 episodes iterating over 200 test-episodes. This step was taken in order to test out whether decreasing the gradient for the learning rate alongside greatly increasing the number of episodes would solve the environment or not. The SARSA implementation took just slightly over 16 minutes to iterate over all of the episodes, but the environment was still not solved. Similarly to the first test case and the Q-Learning approach, after around 1500 episodes, the average reward remained stagnant being seemingly bound by the 7.5 and 8.2 values.

Table 5: SARSA Unsuccessful Attempt

Epsilon Value	Gamma Value	Learning Rate
0.000000001	0.95	0.69
Episodes	Test Episodes	Time Taken
20000	200	16m:39s

4.2.3 Comparisons

Similarly, both implementations despite tweaking of the hyper-parameters could not solve the Taxi-v3 environment. But from the tables plotted, it can be spotted that the SARSA implementation performs much faster than that of Q-Learning. For the first test-case the Q-Learning algorithm executed in 7 minutes and 23 seconds whereas the SARSA equivalent finished in just over 5 minutes, rendering the SARSA policy to be 25% faster than the latter. Considering the second test case, the SARSA algorithm still keeps an edge in time, rendering the SARSA policy to be 30% faster in this test case.

4.3 Experiment 3 - LunarLander-v2

4.3.1 Q-Learning

The Q-Learning implementation starts off with an alpha value of 0.1 with a discounting factor of 0.99. With the use of these hyper-parameters the LunarLander-v2 environment was rendered to be solved in 649 episodes with an average score of 195.64 over 9 minutes and 24 seconds.

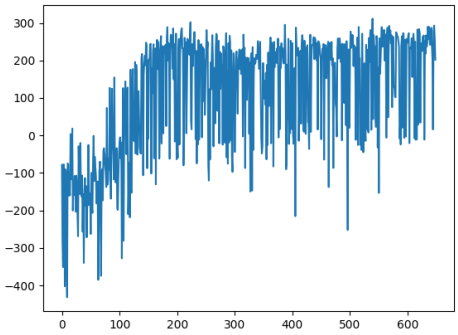

Figure 10: Average Reward Graph of Q-Learning. (Source: Owen Agius)

4.3.2 SARSA

The SARSA implementation started off with an alpha value of 0.1 alongside a discount factor of 0.99. These parameters helped in rendering the application of SARSA to successfully solved the LunarLander-v2 environment. Overall, the SARSA algorithm solved the environments in 378 episodes with an average score of 197.12 over 8 minutes and 42 seconds.

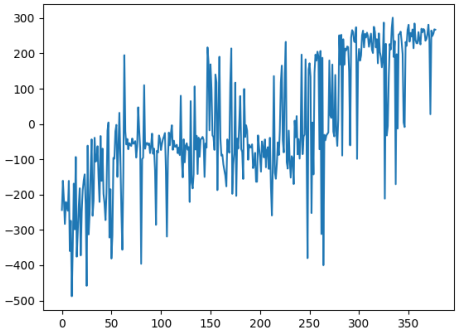

Figure 11: Average Reward Graph of SARSA Learning. (Source: Owen Agius)

4.3.3 Cross Entropy Method (CEM)

Due to the way CEM was implemented, it does not require as many iterations as the other implementations used for this experiment being Q-Learning and SARSA. For this test run, the parameters used for CEM were 200 on the population, extra_std set as 2.0 and extra_decay_time set as 5. Using these parameters, the environment was solved in 12 episodes with an average score of 206.4 over 52m:20s. As can be seen from 12 the algorithms starts off with a sudden spike and crash in the average rewards but after then a steady incline in average rewards was the trend, rendering the environment solved in 12 episodes.

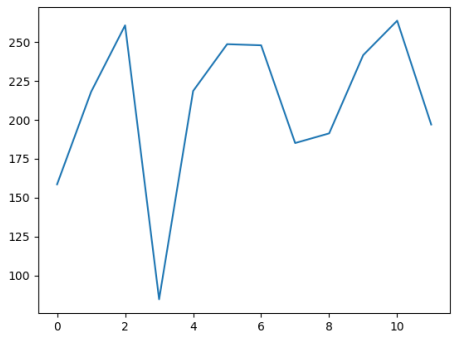

Figure 12: Cross Entropy Method Average Reward Graph. (Source: Owen Agius)

4.3.4 Comparisons

Comparatively, the SARSA implementation keeps an edge over the Q-Learning algorithm in the cases of how fast the environment was rendered solved in terms of episodes and also in terms of time. The times achieved in 8 minutes 42 seconds and 9 minutes and 24 seconds for the SARSA and Q-Learning implementations respectively renders the SARSA implementation to be 9% faster than the latter. Considering, the Cross Entropy Method, this implementation outperforms both the SARSA and Q-Learning techniques in terms of how quick the environment can be solved in relation to the episodes iterated. But this is not the case for the time taken to solve, with just slightly over 52 minutes to solve, it can be concluded that despite the CEM is much more computationally expensive than the traditional techniques in SARSA and Q-Learning.

5 References

[1] C. M. Au. Aws deepracer. 2021.

[2] blackburn. Reinforcement learning: Bellman equation and optimality (part 2), 2019.

[3] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba. Openai gym, 2016.

[4] A. Ellegård. *Darwin and the general reader: the reception of Darwin's theory of evolution in the British periodical press, 1859-1872*. University of Chicago Press, 1990.

[5] S. Gadgil, Y. Xin, and C. Xu. Solving the lunar lander problem under uncertainty using reinforcement learning. In *2020 SoutheastCon*, volume 2, pages 1–8. IEEE, 2020.

[6] J. Gauci, E. Conti, Y. Liang, K. Virochsiri, Y. He, Z. Kaden, V. Narayanan, X. Ye, Z. Chen, and S. Fujimoto. Horizon: Facebook's open source applied reinforcement learning platform. *arXiv preprint arXiv:1811.00260*, 2018.

[7] L. P. Kaelbling, M. L. Littman, and A. W. Moore. Reinforcement learning: A survey. *Journal of artificial intelligence research*, 4:237–285, 1996.

[8] L. Li, W. Chu, J. Langford, and R. E. Schapire. A contextual-bandit approach to personalized news article recommendation. *CoRR*, abs/1003.0146, 2010.

[9] B. Liu, G. Tur, D. Hakkani-Tur, P. Shah, and L. Heck. End-to-end optimization of task-oriented dialogue model with deep reinforcement learning. *arXiv preprint arXiv:1711.10712*, 2017.

[10] D. E. Moriarty, A. C. Schultz, and J. J. Grefenstette. Evolutionary algorithms for reinforcement learning. *Journal of Artificial Intelligence Research*, 11:241–276, 1999.

[11] A. Nandy and M. Biswas. Google's deepmind and the future of reinforcement learning. In *Reinforcement Learning*, pages 155–163. Springer, 2018.

[12] E. Odemakinde. Model-based and model-free reinforcement learning: Pytennis case study, 2021.

[13] S. Ravichandiran. *Deep Reinforcement Learning with Python - Second Edition*. 2020.

[14] S. Ray and P. Tadepalli. *Model-Based Reinforcement Learning*, pages 690–693. Springer US, Boston, MA, 2010.

[15] G. Rudolph. Evolution strategies. *Evolutionary computation*, 1:81–88, 2000.

[16] S. Russell and P. Norvig. Artificial intelligence: a modern approach. 2002.

[17] B. F. Skinner. Operant conditioning. *The encyclopedia of education*, 7:29–33, 1971.

[18] R. S. Sutton, A. G. Barto, et al. Introduction to reinforcement learning. 1998.

[19] E. T. T Erez, Y Tassa. Hierarchical reinforcement learning with the maxq value function decomposition. 2011.

[20] C. J. C. H. Watkins. Learning from delayed rewards. 1989.