



UNIVERSITY OF  
LIVERPOOL

2024/25

**Student Names:** Roy Triggs, Lucas Pavlou  
**Student ID:** 201272208, 201840900  
**Project Title:** COMP532  
Bio Inspired Optimization and  
Machine Learning Assignment 1  
**Lecturer:** Meng Fang

# DEPARTMENT OF COMPUTER SCIENCE

The University of Liverpool, Liverpool L69 3BX

# Bio-inspired Optimization and Machine Learning

## Assignment 1: Multi Armed Bandits

### Contents

Project Report.....	1
Abstract.....	1
Introduction.....	1
Model.....	2
Results.....	3
Exploitation vs Exploration.....	6
Action-Value Methods.....	7
References.....	11
Team Member Contributions.....	11

### Project Report

#### Abstract

Our project built a multi-armed bandit, with 5, 10 and 20 arms. using Python 3.11 in a Google Colab Jupyter Notebook. We found that the adjustment of the following parameters changed the average reward for 2000 bandits over 2000 steps: number of arms, near greedy epsilon value, actual values and reward variance. We investigated exploitation and exploration using greedy and near-greedy values  $\epsilon = 0, 0.01, \text{ and } 0.10$ , where the action values for each arm were distributed evenly between 0 and 2 according to the number of arms of the bandit. We studied action-value methods by implementing one type in our model and writing an essay about the others. We found that, for each n-armed bandit, the near-greedy methods settled on a higher action value arm in the long run compared to the absolute greedy method, despite the initial reward increase being equally steep for each  $\epsilon$  value.

#### Introduction

Let  $A = a_1, a_2, \dots, a_k$  be a set of all possible actions. Then we define the true value of an action ( $a$ ) to be the expected reward ( $R$ ) when that action is selected.

$$q(a) = E(R | A = a) \quad (\text{Equation 1})$$

and to estimate this value

$$Q_t(a) = \frac{\sum_{t=0}^N (R_t | A_t = a)}{N_t(a)}, \quad N_t(a) \neq 0 \quad (\text{Equation 2})$$

$Q_t(a)$  is equal to the sum of the sampled rewards from action  $a_i$  divided by the number of times that action has been sampled. Also known as simple averaging<sup>2,3</sup>. Accordingly, we will

combine the simple average with an action selection rule to form the action-value method used in our model. The most basic action selection rule is to select the action with the highest estimated action value. Appropriately named, the greedy action selection method<sup>2</sup>

$$A_t = \arg \max_a Q_t(a) \quad (\text{Equation 3})$$

$A_t$  (the action selected) is equal to the maximum estimated action value, with ties broken arbitrarily. Given current knowledge, this method exploits the highest returning action. Equations (1, 2 and 3) are the mathematical foundations of our multi-armed bandit model.

## Model

Our model is made up of 4 separate functions – *Bandit()*, *GreedyMethod()*, *RewardFunction()*, and *UpdateRewards()*. Each task is made up of a bandit of n-arms which sequentially loops through the last three functions described above 2000 times each. Each bandit uses  $\epsilon = 0, 0.01$  and  $0.10$ . After each task was completed, we plotted the cumulative rewards divided by the number of steps at time t to give us the average rewards at the given time point. This allowed us to visually compare the effects of different exploration rates.

### *Bandit() function*

This function is the bandit generator. It creates an n-armed bandit, where n can be determined by the user. The input of the function is the number of arms that the bandit has. Outputs are a list of starting rewards (set to 0 during the testing phase and optimistically set to 10 to incentivise exploration during the initial steps). This function also outputs a list of evenly incremented action values between -1 and 1 for the number of arms chosen. Alongside the actual values, we stored a 0 vector to act as a count for the number of times an action is selected.

### *GreedyMethod() function*

This is the greedy action selection method. This function selects the action (or one of the actions, if there is a tie) with the highest current estimated action value. One of the inputs for this function is the current rewards list as of the previous step during the task. The other input is the  $\epsilon$ -value, which determines the frequency at which the bandit randomly explores a different arm, despite the highest current action value (the highest action value arm is still included in the random choice). The output is the index of the next action i.e., the next lever to pull.

### *RewardFunction() function*

This function takes the chosen arm from the *GreedyMethod()* function and adds a random value to it. This random value is taken from a Normal (Gaussian) distribution with mean 0 and a user-specified variance. This simulates a reward for the agent for a given action. The function outputs the reward for the action.

### *UpdatesRewards() function*

The update rule uses Q-learning concepts to update the current reward list. In other words, the estimated action value for the most recent action is updated with the most recent reward divided by the total number of times that action has been performed. The function outputs this updated current averaged rewards list: one averaged reward value per arm.

### Implementing tasks

We combined the above functions such that we create a bandit with  $n$  arms—this creates the true values list as described above. Then the user defines the number of iterations of the loop. The loop starts with the *GreedMethod()* function which arbitrarily chooses the first action to take, then temporarily saves the reward for the chosen action (generated by the reward function). We add this reward to the cumulative rewards list. Next, we call *updateRewards* function to revise the current estimate list. Then, the loop starts over until the desired number of time steps is achieved.

We ran this loop 3 times, each time with different values of epsilon (0, 0.01 and 0.1). In each case we collected the cumulative rewards list, divided by the time step to generate an average rewards list for each greedy/near-greedy agent.

We plotted the average rewards list against the number of time steps to visually compare the three agents' performance for each task  $\epsilon = 0, 0.01, 0.1$ .

## Results

In our investigation, we ran 2000 5, 10 and 20-armed bandits with  $\epsilon$ -values 0, 0.01 and 0.1 over 2000 time steps. We found that, for a given  $n$ -armed bandit, the long run average rewards changed depending on the value of epsilon. In general, we found that the lower the number of arms, the better the near-greedy methods performed compared to the purely greedy method although, in the long run, both of the near-greedy methods attained a higher average reward than the greedy method for all  $n$ -armed bandits.

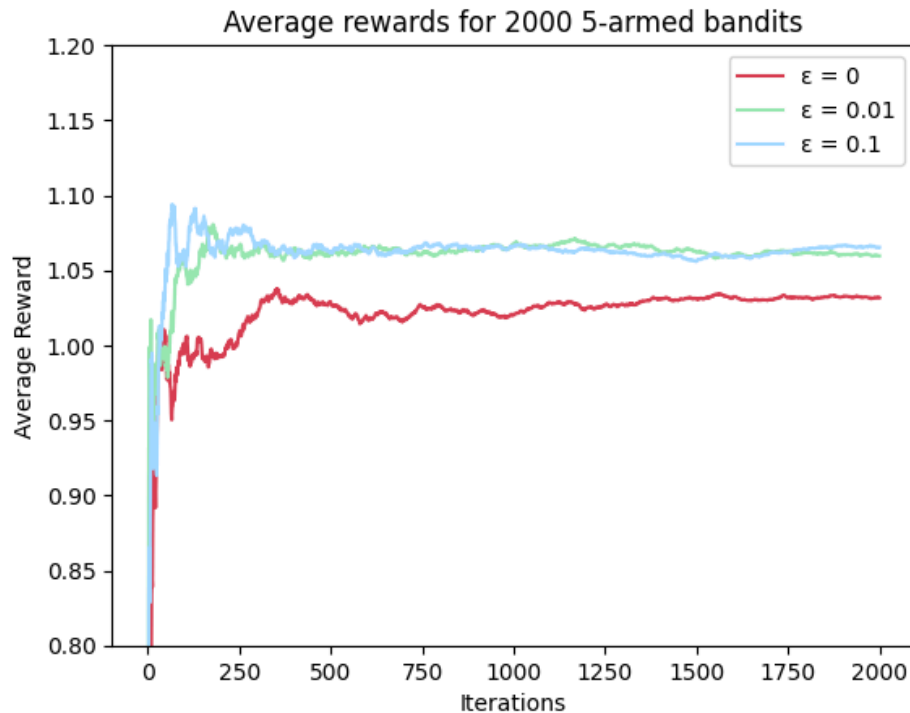
In all 3 graphs below, it can be seen that all bandits showed a significant increase in average reward in the early time points as they explored all the arms available to them. For each 5-armed bandit (*Figure 1*), this rapid increase of average reward started to level out after around 250 steps. This is likely because the relatively low number of arms led to the optimal action being found very quickly and, despite occasional exploration, the other actions would never yield sufficient reward to make exploiting these new actions worthwhile. However, as the number of arms increased, the number of steps taken to level out the average reward at each step also increased. This is because there were more actions for these bandits to test in order to find the optimal action. Hence, we also observed that the greatest average reward for each  $n$ -armed bandit decreased as  $n$  increased, as the reward average was being brought down by more extensive exploration of sub-optimal actions in the initial time steps. This also explains why, generally speaking, the absolute greedy ( $\epsilon = 0$ ) method levelled out quicker than the near-greedy methods.

In the example, average reward plots from the text book by Sutton and Barto (2014), the findings for average rewards over time are quite similar to ours in many ways. The example shows that the absolute greedy method initially has a similarly rapid increase in average reward to the near-greedy methods, but ultimately settles on a sub-optimal action.

The example, however, suggests that, although the  $\epsilon = 0.1$  method initially settles on the best action of all three methods, in the very long term (over 1000 steps), the  $\epsilon = 0.01$  method eventually catches this method up and overtakes it in average reward. This would be because, initially, the  $\epsilon = 0.01$  method takes a long time to explore all actions and determine the optimal one, but once it has found the best action, it is less likely to deviate from it to

explore other actions, and eventually accumulates a higher average reward than the  $\epsilon = 0.1$  method.

However, we did not observe this effect for any of our bandit models, despite using more iterations in our data than the example. One reason for this could be that the distribution of potential rewards around the true values of our model were much narrower than in the example, making it very likely that the bandit will accurately approximate the actual value of a given arm after only a few trials and decide whether or not to continue exploiting it. This is an improvement that we could make to our multi-armed bandit implementation in the future.



*Figure 1: Average rewards against interactions (time steps) for one 5-armed bandit with three different exploration rates,  $\epsilon = 0$  (red), 0.01 (green) and 0.1 (blue). Both near-greedy methods achieved noticeable higher average rewards in the long run compared to the absolute greedy method.*

In our data, it is only when  $n = 20$  (Figure 3), that any observable difference occurs between the  $\epsilon = 0.01$  and  $\epsilon = 0.1$  methods. At this point, the number of possible actions that are required to be explored outweighs the narrow distribution of rewards surrounding each true action value. Therefore, the  $\epsilon = 0.01$  method takes far longer to find the optimal action than the  $\epsilon = 0.1$  method.

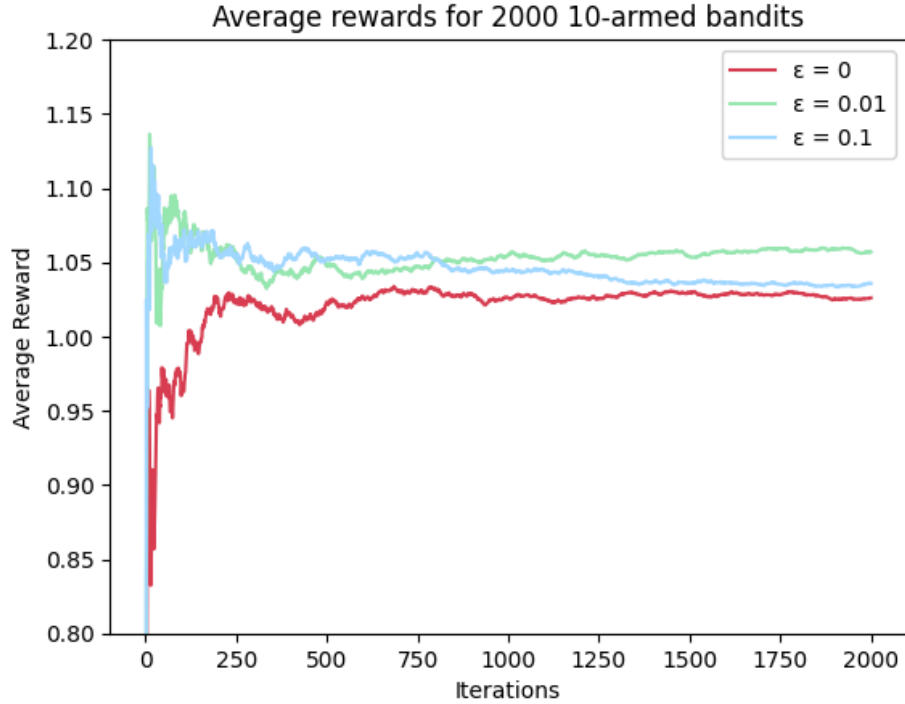


Figure 2: Average rewards over time for a 10-armed bandit with exploration rates,  $\epsilon = 0$  (red), 0.01 (green) and 0.1 (blue). The greedy was outperformed by both near-greedy methods, although the difference in average reward is not as significant as for 5-armed bandits.

Overall, our bandits and their average reward changes do compare fairly closely to the example given in the textbook by Sutton and Barto (2014). Hence, we feel that the Python programme that we built together is quite robust and gives a good approximation of the Q-learning action-value method for multi-armed bandits. In future, we would like to extend our investigation to widen the distribution of rewards surrounding each action value. We would then test this new distribution using the current n-armed bandits but also extend the study to include bandits with more arms, in order to discover how the different exploration values affect optimal action determination in the medium to long term.

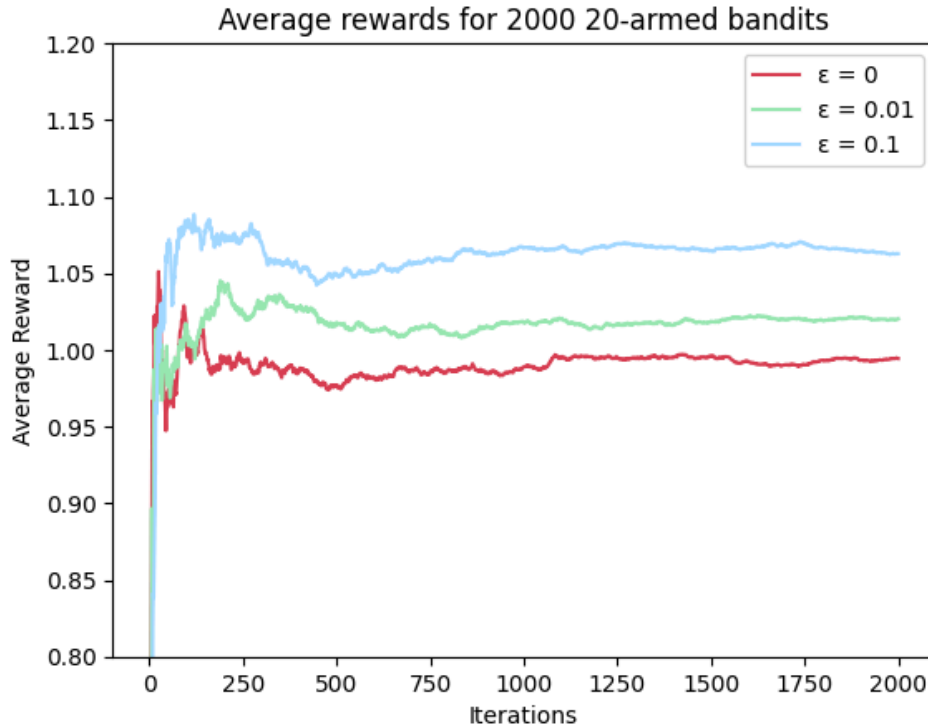


Figure 3: Average rewards per step for a 20-armed bandit with three exploration rates,  $\epsilon = 0$  (red), 0.01 (green) and 0.1 (blue). Here, both near-greedy action-value methods considerably out-performed the absolute greedy method in average reward accumulation in the long run, with  $\epsilon = 0.1$  performing the best overall.

## Exploitation vs Exploration

Artificial intelligence (AI) mimics behavioural patterns and intelligence of humans or other living entities. AI encompasses all intelligent systems, a subset of which is machine learning (ML). Reinforcement learning (RL) is a technique of ML, in which agents learn from their environment by taking actions, receiving rewards and transitioning between states <sup>1</sup>.

To incentivise an explicit trial-and-error search for good behaviour. RL utilises reinforcement functions - rewards or punishments, i.e., evaluation feedback. Evaluation feedback returns an action value, but not whether that action was the best possible action; by constitution, evaluative feedback is dependent on the action taken <sup>1,2</sup>.

Coactively, instructive feedback - used in another technique of ML: supervised learning - denotes the correct action to take, independent of the action selected. To investigate differences and confluence of both feedback methods, closely related to exploration and exploitation, we built our n-armed multi-bandit testbed above.

Exploration lowers uncertainty that the current action selections are a suboptimal solution. Exploitation is dependent on the agent's current knowledge. Exploitation maximises the short-run rewards in a machine learning task. Reinforcement learning combines exploitation with exploration to optimise reward-based learning. In other words, practicing best actions whilst experimenting to look for potential better techniques.

Exploration and exploitation are mutually exclusive; therefore, there is a trade-off between them. Overexploitation can cause suboptimal action sets. While too much exploration can be wasteful, spending too much time sampling redundant actions.

For a reinforcement learning model to perform most effectively, data scientists must find an equilibrium between exploring and exploiting. This balance depends on the task at hand. The more actions there are to choose from can lower a purely greedy method's chances of exploiting the optimal action. Whereas, if the set  $A_t$  is small and the reward distributions are distinct (or known), a greedy bandit should identify the best action quickly. On the contrary, without exploration, there is zero chance that the greedy method will switch to the optimal action in the long run, should the best action have a 'disappointing run' early on,<sup>1</sup> i.e., lowering expectations of an optimal action below the average returns of a suboptimal solution. This highlights the importance of exploration.

Near-greedy action value methods (explorers) utilise The Law of Large Numbers. As time steps tend towards infinity, any model with an  $\epsilon > 0$  will tend towards the optimal action. This is because, eventually, all of the actions will be sampled. However, if there is a very large number of potential actions, this could take a very long time; as such, the number of time steps is also a determining factor in the pursuit of balancing exploration and exploitation.

## Action-Value Methods

In reinforcement learning, action-value methods use known rewards obtained from previous actions and use these to calculate the estimate of the reward of a specific action at the next step. At each step, the estimate is updated based on the “acquired knowledge” of rewards from previous steps.

Broadly speaking, there are two types of action-value methods – off-policy and on-policy. Off-policy methods (such as Q-learning) aim to learn the optimal policy for carrying out actions in an environment by *exploring* as many of those actions as possible. Example policies include exploratory and greedy methods. At each time step, Q-learning updates the Q-value (the expected reward at each time step) based on the best possible next action that could be taken, rather than the action that the agent chooses. On-policy methods will be described and discussed in more detail later, but the next few examples will be of off-policy Q-learning methods.

One off-policy action-value method ( $\epsilon$ -greedy) has already been described in this document. Although the  $\epsilon$ -greedy method is intuitive to understand and relatively simple to compute, the method requires the reward value for each and every action to be stored in memory, in order to calculate the average over time. This quickly becomes computationally expensive and leads to increased computational time in the long run. In practice, where thousands or even hundreds of thousands of trials are required to determine the optimal action (i.e. where the number of arms of the bandit is very large), it becomes unfeasible to use the estimated average method.



Thankfully, several other Q-learning methods exist for estimated reward computation. These alternatives come under the bracket of *recursive* methods, where the reward value from the previous time step is used to calculate the next reward.

The first of the recursive methods that we will describe here is called incremental implementation. It turns out that the formula for the action value update rule can be derived directly from the sample average calculation given in *Equation 2*, which is repeated here for convenience.

$$Q_t(a) = \frac{\sum_{i=0}^N (R_i | A_i = a)}{N_t(a)} \quad \text{Equation 2}$$

$$Q_t(a) = \frac{R_1 + R_2 + \dots + R_{N_t(a)}}{N_t(a)}$$

To estimate the reward of the next action (*Equation 4*), we must use a recursive equation derived from *Equation 2*, which is as follows:

$$\begin{aligned} Q_{t+1} &= \frac{R_1 + R_2 + \dots + R_t}{t} \\ &= \frac{1}{t} \sum_{i=1}^t R_i \\ &= \frac{1}{t} \left( \sum_{i=1}^t R_i + R_N \right) \\ &= \frac{1}{t} (R_t + (t-1)Q_t + Q_t - Q_t) \\ &= \frac{1}{t} (R_t + tQ_t - Q_t) \\ &= Q_t + \frac{1}{t} [R_t - Q_t] \end{aligned} \quad \text{Equation 4}$$

In the fully simplified equation, We can see, therefore, that the formula to determine the action-value estimate  $Q_{t+1}$  only requires the average reward from the previous steps ( $Q_t$ ) and  $t$  (as  $R_t$ , the new reward, will be determined at each step). At time  $t = 1$ , this value is often set to either 0, or a random guess.

By evaluating the expression  $R_t - Q_t$  in *Equation 4*, we obtain the difference between the new reward and the current estimate. This value allows us to update the current reward estimate with. *Equation 3* also describes a step size factor  $\frac{1}{t}$ . We can clearly deduce that as time progresses,  $t \rightarrow \infty$  and hence  $\frac{1}{t} \rightarrow 0$ . This ultimately reduces the impact that each subsequent reward value has compared to the previous one in calculating the current estimate.

Ashis (2024) gives a good example of an application of both the full memory and incremental update methods, by tracking a sports team's scores over a season. Let's say a football team scores 3, 0, 1, 1, 3 and 0 points (reward values) after the first 6 games. We could use the full memory method to calculate the predicted reward in the next game as follows:

$$Q_t(a) = \frac{3+0+1+1+3+0}{6} = 1.333...$$

This method will always work; however, as the season progresses, we will have to store 38 values for Premier League teams and 46 for lower-division clubs. What if we want to calculate expected rewards for each of the 92 teams in the Football League? This would become very computationally heavy.

Instead, we can use the incremental update method for each club. Using the above reward values, we can calculate that, after 5 games, the average score is 1.6 points. This value can then be used to calculate the reward estimate for the 6<sup>th</sup> game as follows:

$$Q_{t+1} = 1.6 + \frac{1}{6} [0 - 1.6] = 1.333...$$

This method reaches the same answer, whilst reducing computational strain.

On the other hand, the incremental implementation requires that the current estimate is already known. This means that all previous reward estimates must be computed before calculating  $Q_{t+1}$ , as the update equation is iterative. With the full-memory method,  $Q_t(a)$  can be calculated *at any time*, as long as the rewards from each time step are known.

So far, we have discussed methods for tracking rewards in stationary environments. These are situations where the actual reward values for a given action do not change with time. However, many real-world situations do not behave in such a fixed way, and the actual values ( $q(a)$ ) change regularly. These are called *non-stationary environments (NSEs)*. An example of an NSE is with stock market trading. Here, several different conditions – such as investor behaviour, government policy and world events – can change over time. In this case, recent rewards (i.e. profits from trading decisions) should be considered as more influential in calculating  $Q_{t+1}$  than rewards from the past, as they are more representative of the present environment. Without the decay factor, the reward value prediction may over-fit to old data which are based on outdated environmental parameters.

Using the generalised formula for updating the average value  $Q_t$  (Equation 5), we can derive Equation 6 through a series of steps left to the reader to evaluate.

$$Q_{t+1} = Q_t + \alpha [R_t - Q_t] \tag{Equation 5}$$

$$Q_{t+1} = (1 - \alpha)^t Q_1 + \sum_{i=1}^t \alpha (1 - \alpha)^{t-i} \cdot R_i \tag{Equation 6}$$

Here, the step-size parameter  $\alpha$  is a predefined constant, where  $0 < \alpha \leq 1$ . The weight given to  $R_i$  (namely  $\alpha(1 - \alpha)^{t-i}$ ) is weighted as a result of how many steps ago ( $t - i$ ) the

reward was obtained and decreases over time as  $1 - \alpha$  will always be  $< 1$ . We can also observe that *Equation 5* is very similar to the perceptron update rule, where the weights can, indeed, change over time. Both the perceptron update rule and NSE equations include a learning rate as well as weighting for previously acquired information. However, the NSE equation has an explicit decay factor, whereas with the perceptron update rule, decay is implicit, reducing the relevance of old data on the most recent error correction. Most importantly, the NSE equation is recursive, allowing the model to consider the complete history of rewards, rather than just the most recent data point and its error.

As we have seen, Q-learning updates the Q-value (the expected reward at each time step) based on the optimal policy. By contrast, *on-policy* methods do not consider the best possible policy for agent learning. Instead, on-policy methods use rewards to update only the policy that the agent is currently following. The best-known on-policy method is state-action-reward-state-action (SARSA).

In SARSA, the agent starts at state  $s_t$ . From here, the agent chooses an action  $a_t$  from a set of actions defined by its policy  $\pi$ . As the agent moves to the new state  $s_{t+1}$ , it receives a reward  $r_t$ . The agent then chooses another action, called  $a_{t+1}$ , defined within  $\pi$ . Action  $a_{t+1}$  is then used to update action-value function the policy (*Equation 7*), as follows:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)] \quad \text{Equation 7}$$

Like the incremental update and NSE methods, the SARSA action-value function is *recursive*, relying on the Q-values calculated in the previous steps. Again,  $\alpha$  represents the step-size parameter for SARSA. Here,  $\gamma$  is a discount factor between 0 and 1 which determines how much the agent will prioritise immediate over future rewards. Values close to 0 prioritise immediate rewards, whereas a value closer to 1 takes future rewards into consideration. This value can be changed in order to optimise the learning speed of the agent, but may lead to “overfitting” in learning, where the agent settles on a sub-optimal series of actions if it deems them to be more profitable in the short-term.




We can observe that the action-value update depends on action-values for both the next and current steps. The difference between the next and current steps is calculated similarly to the incremental update method. Previously-calculated values (rather than a growing list of data) are used to determine the reward of the next step in an iterative fashion.

In practice, Q-learning algorithms are more likely to test all actions, especially at early time points. This makes agents that learn using this strategy more likely to acquire penalties throughout the learning process, as they test all actions. Whilst this can lead to longer learning times than with SARSA, Q-learning will usually converge to a near ideal strategy in the long term<sup>4</sup>. As time increases, the SARSA algorithm teaches the agent to avoid acquiring penalties at all costs. This causes the agent to learn a policy that reaches the goal within fewer learning iterations, but the path it finds would likely be sub-optimal.

## References

1. Russell SJ, Norvig P, Chang M-W. *Artificial intelligence: A modern approach*. 4th ed. Harlow, United Kingdom: Pearson; 2022.
2. Sutton RS, Barto AG. *Reinforcement Learning: An Introduction*. 2nd ed. Cambridge, MA: The MIT Press; 2018.
3. Rice JA. *Mathematical statistics and data analysis*. 3rd ed. Belmont, CA: Thomson; 2007.
4. GeekforGeeks. Differences Between Q-Learning and SARSA. 2025; <https://www.geeksforgeeks.org/differences-between-q-learning-and-sarsa/>. Accessed 17/03/25.

## Team Member Contributions

<i>Team Member</i>	<i>Programming</i>	<i>Report</i>	<i>Problem2</i>	<i>Problem3</i>
Lucas Pavlou				
Roy Triggs	