Assignment 1: Reinforcement Learning
COMP532
J. McCarron, H.Williams

This document is intended to serve as a companion to the submitted codebase. Here, we can explain the choices behind our technical implementation, and further discuss the implications of our results as they relate to the field of Reinforcement Learning (RL).

PROBLEM 1

Problem 1 required us to reimplement and plot a comparison of a greedy method against 2 different $\mathcal{E}$-greedy methods; as per figure 2.2 *(Sutton and Barto, 2018).* $\mathcal{E}$ or epsilon, is a way of quantifying how the policy balances the exploration-exploitation dichotomy. Exploration & exploitation are mathematically defined as such:

$$a_t^* = argmax\ Q_t(a)$$

$$a_t = a_t^* \longrightarrow exploitation$$
$$a_t \neq a_t^* \longrightarrow exploration$$

Balancing exploration and exploitation is an issue at the very core of reinforcement learning. By exploiting learnt knowledge, the agent can guarantee a certain level of reward. However, it is then likely that an agent may 'lock-on to a sub-optimal action forever' *(Dey, 2018).* Conversely, excessive exploration prevents '[maximisation] of the short-term reward' *(Tokic, M. 2008).*

We have seen that neither sole exploration nor exploitation provide a satisfactory modus operandi for our agent. As such, a way of balancing the two is required. The $\mathcal{E}$-greedy method provides a means for the agent to generally exploit, but occasionally explore. The exploration aspect of the $\mathcal{E}$-greedy method considers all sub-optimal actions equally, that is to say, if an exploratory move is to take place, all actions other than the current maximum have an equal chance of taking place. As such, we can consider this to be random exploration.
Rank based approaches, such as SoftMax, take into account the estimated value of the sub-optimal actions, and use these values to determine which unideal actions may be worth exploring, while ignoring those that appear to have limited value.

With regard to our implementation, we elected to use Python. Specifically, we used the Anaconda virtual environment. The Anaconda environment comes preinstalled with a number of useful packages for RL processing; in this instance, NumPy & Matplotlib. The Jupyter Notebook provides an excellent means for developing the RL code in a modular & collaborative fashion.

In practice, $\mathcal{E}$ can be thought of as a threshold which a randomly generated number must fall below, in order for the agent to take a non-greedy action. This can be easily implemented using a single if/else statement.

```
for j in range(0,time):
    if epsilon[k] > np.random.random():
        action = np.random.randint(arms)
    else:
        action = np.argmax(q_Est)
```

As per the figure in Sutton & Barto, we set our experiment to be a 10-armed bandit, with 1000 time steps. The entire experiment is then run 2000 times. The average value for each timestep is plotted on the figure. In addition to the charts in Sutton & Barto, we have created a bar chart plotting the maximum average reward that each method obtained (Appendix B). The code for gathering the maximum value is very straightforward.

```
max = []

for i in range(len(reward)):

    max.append(np.max(reward[i]))

print(max)
```

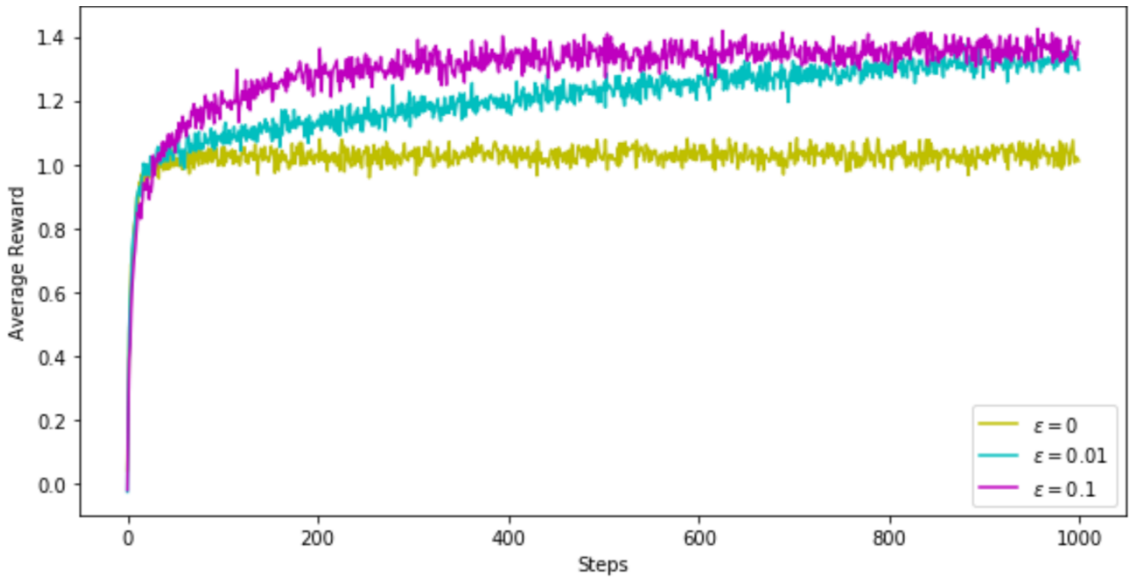
```
# Function to return average reward for each time step

def k_bandit(arms, time, runs, epsilon, returnOpt):

    final_reward = [[] for i in range(0,len(epsilon))]    # Averaged rewards at each time step, for each ε
    final_optimal_action = [[] for i in range(0,len(epsilon))]  # Optimal Action % for each epsilon

    for k in range(len(epsilon)):
        Rewards = np.zeros(time)
        OptimalAction = np.zeros(time)

        for i in range(runs):
            q_Star = np.random.normal(0,1,arms)      # Randomly initialises the Q values between 0 & 1
            best_Action = np.argmax(q_Star)          # Stores the action with the highest value to the best_Action
            q_Est = np.zeros(arms)                   # Updates the estimated value of each arm
            counts = np.zeros(arms)                  # The amount of times each arm is taken
            rew = []
            optimalAction = []

            for j in range(0,time):
                optimal_Count = 0
                if epsilon[k] > np.random.random():  # If the epsilon value is greater than RNG
                    action = np.random.randint(arms) # Make exploratory move
                else:
                    action = np.argmax(q_Est)        # Otherwise, exploit known knowledge.
                if action == best_Action:
                    optimal_Count = 1                # If the selected action is the best action, increase OC by 1

                reward = q_Star[action] + np.random.normal(0,1) # Gaussian - Mean = 0, Variance = 1
                counts[action] +=1
                q_Est[action] += (1/counts[action])*(reward - q_Est[action]) # Update action values estimates
                rew.append(reward)
                optimalAction.append(optimal_Count)

            Rewards += rew
            OptimalAction += optimalAction

        final_reward[k] = Rewards / runs # Calculates the average reward for each timetsep
        final_optimal_action[k] = OptimalAction / runs # No. times optimal action chosen / total number of runs

    # Chooses which array to return based on the returnOpt parameter

    if returnOpt == 1:
        return optimal_action
    else:
        return final_reward
```

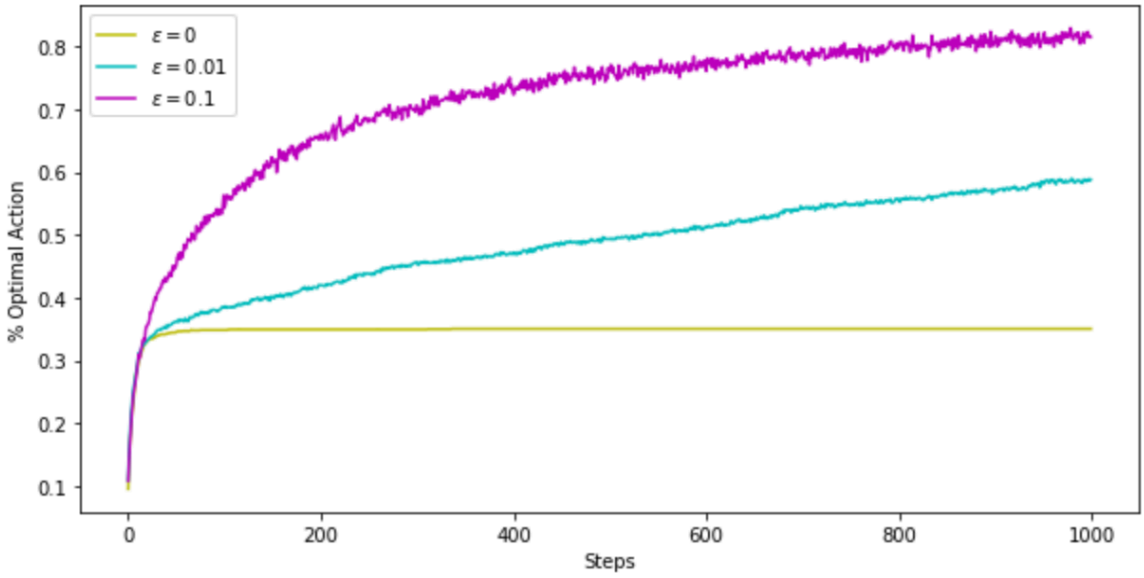
```
epsilon = [0,.01,.1] # The epsilon values to be tested
reward = k_bandit(10, 5000, 2000, epsilon, 0) # Runs the function defined above and stores it in variable reward
optAct = k_bandit(10, 5000, 2000, epsilon, 1) # Runs the function defined above and stores it in variable optAct
```

P1 FINDINGS

From our results, we can draw a number of conclusions. It is easy to see that an entirely greedy policy is suboptimal.
Even the introduction of $\mathcal{E} = 0.01$ has a noticeable effect on the agent's learning, and shows a slow but definite upward trend of the average reward. When the exploration rate is increased to $\mathcal{E} = 0.1$, we can see that the average reward is maximised considerably more quickly than its lower counterpart, yet they ultimately converge at a similar maximum. If the experiment

were to be continued over a greater number of timesteps, $\mathcal{E} = 0.01$ would ultimately perform better (Appendix A).





To further illustrate the exploration/exploitation dichotomy, we plotted a second chart (Appendix B) that introduced 3 additional $\mathcal{E}$ values – 0.2, 0.5 & 1 respectively. As one might expect, $\mathcal{E} = 1$ (total exploration) fails to make any significant progress, and truthfully, cannot be thought of as a learning agent. Even in the instance of $\mathcal{E} = 0.2$, there is a visible loss of stability in the line – which suggests there is an overuse of exploration. When $\mathcal{E} = 0.5$, there is a general upward trend, though too much time is spent exploring to useful exploit any learnt knowledge.

To conclude, a careful balancing of exploration and exploitation is required. Too much of either has a stark effect on the learning ability of the agent. Exploitation is essential for the agent to apply what it has learnt, while exploration is necessary to acquire an appropriate variety of experience, to determine optimal action strategies.

PROBLEM 2

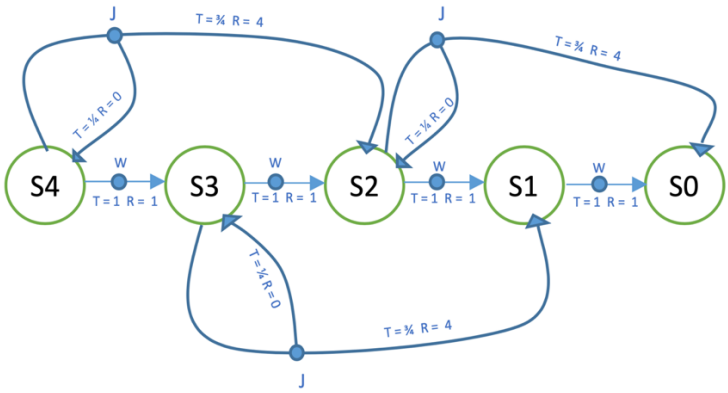


Diagram to illustrate the Markov Decision Process (MDP) outlined in Question 2.

Markov Decision Process with states = {4,3,2,1,0} where 4 (s4) is starting state and 0 (s0) is terminal state. Compute both $V^*(2)$ and $Q^*(3, J)$.
Reward $\mathrm{R}(\mathrm{s}, \mathrm{a}, \mathrm{s}') = (\mathrm{s} - \mathrm{s}')^2$.
Discount γ = ½.

$$\mathrm{V}^*(\mathrm{s}) = \max \sum_{s'} \mathrm{P}^{\mathrm{a}}_{\mathrm{ss}'} [\mathrm{R}^{\mathrm{a}}_{\mathrm{ss}'} + \gamma \mathrm{V}^*(\mathrm{s}')]$$

Where V*(s) is the value of state s under optimum policy

$\mathrm{P}^{\mathrm{a}}_{\mathrm{ss}'} = \Pr\{s_{t+1} = s' = s, a_t = a\}$ for all $s, s' \in$ S, a ∈ A(s).

Transition probability function of moving from state s to s' with action a.

$\mathrm{R}^{\mathrm{a}}_{\mathrm{ss}'} = E\{r_{t+1} \mid s_t = s, a_t = a, s_{t+1} = s'\}$ for all $s, s' \in$ S, a ∈ A(s).

Reward function of state s to state s' with action a.

Compute V*(2):

V*(0) = 0

V*(1) = max {1 (1 + γ V*(0))}
= max {1 + $\frac{1}{2}$(0)}
= max {1}
V*(1) = 1

The value of walk action from state 1(s) to state 0(s').

V*(2) = max {1 (1 + γV*(1)), $\frac{3}{4}$ (4 + +γV*(0)) + $\frac{1}{4}$ (0 + γV*(2))}
= max {1 + $\frac{1}{2}$ (1) , 3 + $\frac{1}{4}$.$\frac{1}{2}$ V*(2))}
= max { $\frac{3}{2}$, 3 + $\frac{1}{8}$ V*(2)}
= max {$\frac{3}{2}$, $3\frac{8}{7}$ }

The value of walk action from state 2 to state 1 and the sum of values of jump action from state 2 to state 0 and state 2 to state 2.

$V^*(2) = 3\frac{8}{7} = \frac{24}{7}$

Where if $V^*(2) = 3 + \frac{1}{8} V^*(2)$ was the maximum:

$V^*(2) = 3 + \frac{1}{8}V^*(2)$

$\frac{7}{8}V^*(2) = 3$

$V^*(2) = 3.\frac{7}{8}$

Compute Q*(3,J):

$$Q^*(s,a) = R_s^a + \gamma \sum P_{ss'}^p, V*(s')$$

Q*(s,a) gives the expected return for taking action a in state s and thereafter following an optimal policy. (Sutton and Barto, 2018, pp. 50).

R_s^a = ¾(4) + ¼ (0) = 3

Jump action k-2 = ¾ and reward is 4 and action J k = ¼ and reward is 0.

$Q^*(3,J) = 3 + \frac{1}{2}V^*(1)$
= 4 + ½
= 4.5

Taking action J from S3 to S1 and optimum policy from S1 to S0 ($V^*(1)$).

PROBLEM 3

Q1 - *What does the Q-Learning update rule look like in the case of a stateless or 1-state problem?*

A – Q-learning is a values-based learning algorithm in reinforcement learning (ADL, 2018). Primarily, it is concerned with mapping the expected reward of an action, when taken from a specific state. Initially, the environment is unknown to our agent, so it logically follows that these values are unknown. There are 2 main approaches towards 'initialising' Q-Values; Gaussian/Normal distribution, and optimistic initial Q. Each of these have their own implications for resolving the exploration/exploitation dichotomy.

In order for learning to take place, these values need to be updated, so that they reflect the agent's understanding of the environment. In pseudocode terms, the general format for the Q update rule is.

$$\boldsymbol{newEstimate = oldEstimate + stepSize[target - oldEstimate]}$$

In a MPD, with multiple states, the update rule is defined as the following:

$$Q^*(S_t, A_t) = \mathbb{E}[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3}, \dots \mid s_t, a_t]$$

$$NewQ(s,a) = Q(s,a) + \alpha[R(s,a) + \gamma\, maxQ'(s',a') - Q(s,a)]$$

In a MDP, there is an additional aspect, not included in the pseudocode formula defined above. The goal of our agent is to maximise its long term reward, however, in a single-state (or arguably, stateless) environment such as a K-Armed Bandit problem, the expectation of discounted future rewards is not factored in. As such, both the Q itself, and the update rule are defined more simply.

$$Q^*(a) = \mathbb{E}[R_t \mid A_t = a]$$

$$Q_k = \frac{r_1 + r_2 + r_3 + \cdots r_k}{k}$$

As state is not taken into account (and arguably, does not exist at all) the equation is simplified to just be based off the action value estimates. The most computationally efficient way of doing this is by taking a sample average based on the previously earned rewards, and using that to update the Q value. As such, the update rule can be written as follows:

$$Q_{k+1} = Q_k + \frac{1}{k+1}[r_{k+1} - Q_k]$$

Q2 – *Discuss the main challenges that arise when moving from single- to multi-agent learning, in terms of the learning target and convergence*

A – In the field of Reinforcement Learning (RL), single-agent systems are based on the notion of finding an optimal solution to solving a relatively simple task, using operant conditioning; an idea first solidified by Edward Thorndike. However, when we move towards systems where more than one agent is acting with autonomy, additional challenges arise. The RL methods for a single-agent system 'assume stationarity of the environment and cannot be directly applied [to multi-agent learning]' (Neto, 2005). In a single agent system, the state of the environment is only altered by the actions of that one agent, yet in a MAS, multiple agents will be interacting with and altering the environment. However, a MAS can yield a distributed solution to a task, which can be more efficient than a centralised single-agent system; therefore acknowledging and managing potential challenges is essential.

How we define our agents' behaviour is very important. For instance, the interactions between agents will be very different depending on whether they are programmed to be collaborating, or in competition with one another. Furthermore, the goal that our agents are pursuing may require the sacrifice of maximising individual reward, in favour of maximising the collective reward. Some parallels can be drawn between MAS and the moral philosophy of utility, put forward by English philosopher Jeremy Bentham. Bentham's theory of utility is concerned with attributing a moral value to a given action, based on its consequence. This value is determined by 'the overall happiness created by everyone affected by the action' (IEP, n.d.) This theory translates well to the idea of a MAS, whereby selfish actions taken by a single agent, may have adverse effects on the overall learning target, and vice-versa. One possible solution to this, is for all agents to have fixed policies, and individually determine a best response against those policies. Collectively, this is called the Nash equilibrium; 'a joint policy for all players, such that every agent's policy is a best response' (Silver, 2017).

As an MAS is inherently a non-stationary system, it logically follows that the learning target is also non-stationary. In other words, the learning target is constantly updated based on the actions of all agents. Therefore, the notion of convergence can't directly be applied, as there is no constant or stationary optimum to strive for.

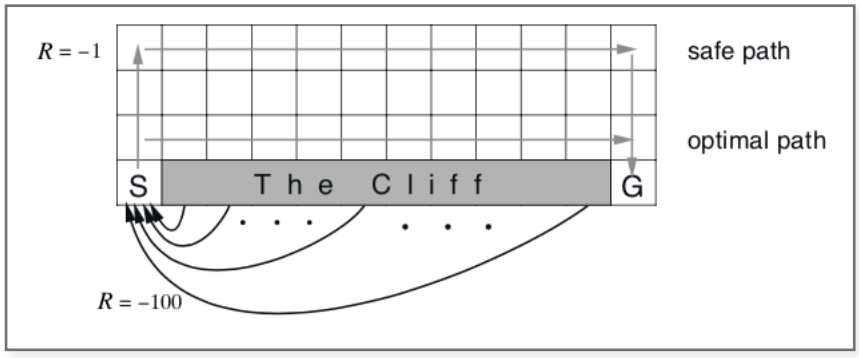
$$\sum_{n=1}^{\infty} \alpha_n(a) = \infty \quad and \quad \sum_{n=1}^{\infty} a_n^2(a) < \infty$$

The above figure describes the conditions required to assure convergence with probability 1. In a MAS, the second condition cannot be met, ' indicating that the estimates never completely converge but continue to vary in response to the most recently received rewards' (Sutton and Barto, 2018, pp. 26).

PROBLEM 4

Problem 4 required us to reimplement and plot a comparison of a Q-Learning algorithm against a Sarsa algorithm; as per figure 6.4 in Sutton & Barto. In particular, this figure shows how different RL algorithms navigate a gridworld environment with a 'cliff' that our agents can fall down. The aim is for our agent to get from the starting grid position, to a goal position, taking as few actions as possible. A variant of the cliff-walking problem exists with the agent having the ability move to diagonal squares, however, in this experiment, our agent can only go Up, Down, Left, or Right.

The cliff-walking gridworld can be visualised as such:



We can consider this a comparison between On/Off TD methods. Sarsa representing an 'on-policy' method, and Q-learning representing the 'off-policy' method. The diagram above shows that each 'safe-state' returns a reward of -1, with exception of 'the cliff', which returns a reward of - 100 and returns the agent to the start position.

The distinction between On/Off policy is subtle, yet important. In short, we can consider Q-Learning to be off-policy because it updates the state-action value based on the value of the next state's greedy action, regardless of whether a greedy policy is being used. However, in the case of SARSA, the estimate is defined as 'the return for state-action pairs assuming the current policy continues to be followed' (Neil, 2015).

SARSA Update Rule

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

Q-Learning Update Rule

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

As with Problem 1, we chose to implement using Python & the Jupyter Notebook. The only additional library used was 'TQDM', which is a library to computationally optimise iteration based experiments.

```
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
from tqdm import tqdm
%matplotlib inline
```

```
grid_Y  = 4
grid_X  = 12
epsilon = 0.1
alpha   = 0.5
gamma   = 1

# Start State, and Target State
startState = [3, 0]
goalState  = [3, 11]

# Available Actions
actionUp    = 0
actionDown  = 1
actionLeft  = 2
actionRight = 3
actionSet   = [actionUp, actionDown, actionLeft, actionRight]
```

```
# Returns the reward based off the state and action taken
def step(state, action):
    i, j = state
    if action == actionUp:
        nextState = [max(i - 1, 0), j]
    elif action == actionLeft:
        nextState = [i, max(j - 1, 0)]
    elif action == actionRight:
        nextState = [i, min(j + 1, grid_X - 1)]
    elif action == actionDown:
        nextState = [min(i + 1, grid_Y - 1), j]
    else:
        assert False

    reward = -1
    if (action == actionDown and i == 2 and 1 <= j <= 10) or (
        action == actionRight and state == startState):
        reward = -100
        nextState = startState

    return nextState, reward

# Choose an action based on the epsilon greedy algorithim
def chooseAction(state, Q_Value):
    if np.random.binomial(1, epsilon) == 1:
        return np.random.choice(actionSet)
    else:
        values_ = Q_Value[state[0], state[1], :]
        return np.random.choice([action_ for action_, value_ in enumerate(values_) if value_ == np.max(values_)])
```

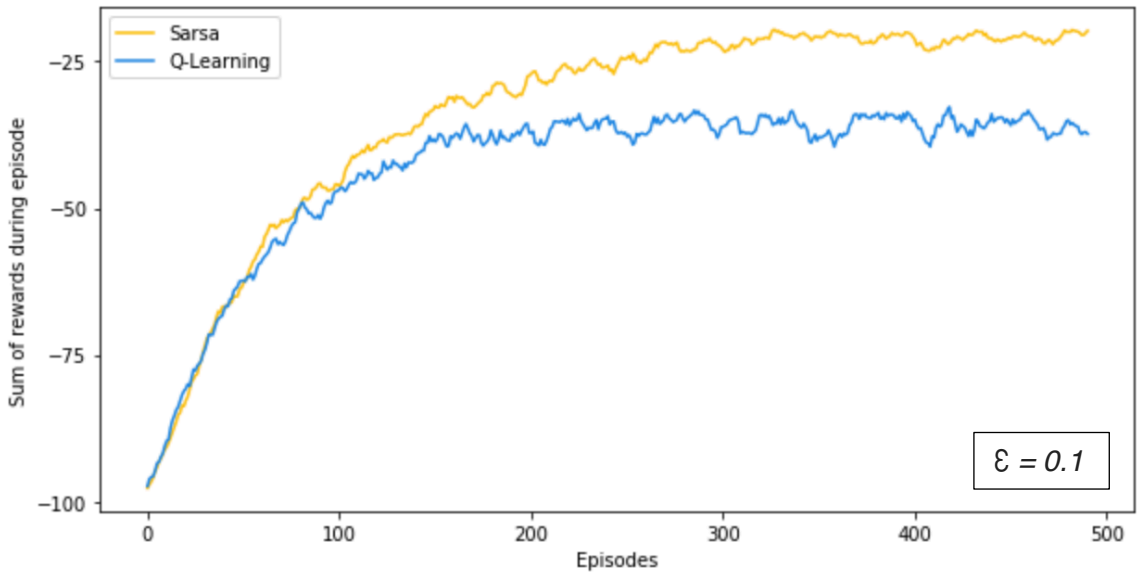
```
# Sarsa algorithm
def sarsa(Q_Value, stepSize = alpha):
    state   = startState
    action  = chooseAction(state, Q_Value)
    rewards = 0

    while state != goalState:
        nextState, reward = step(state, action)
        nextAction = chooseAction(nextState, Q_Value)
        rewards = rewards + reward

        # Expected value of new state
        target    = 0
        Q_Prime   = Q_Value[nextState[0], nextState[1], :]
        maxAction = np.argwhere(Q_Prime == np.max(Q_Prime))

        for action_ in actionSet:
            if action_ in maxAction:
                target += ((1.0 - epsilon) / len(maxAction) + epsilon / len(actionSet)) * Q_Value[nextState[0], nex
            else:
                target += epsilon / len(actionSet) * Q_Value[nextState[0], nextState[1], action_]

        Q_Value[state[0], state[1], action] += stepSize * (reward + target - Q_Value[state[0], state[1], action])
        state = nextState
        action = nextAction
    return rewards

```

```python
# Q-Learning algorithm
def q_learning(Q_Value, stepSize = alpha):
    state   = startState
    rewards = 0
    while state != goalState:
        action   = chooseAction(state, Q_Value)
        nextState, reward = step(state, action)
        rewards = rewards + reward

        # Q-Learning update
        Q_Value[state[0], state[1], action] += stepSize   * (reward + gamma * np.max(Q_Value[nextState[0], nextStat
        state = nextState
    return rewards
```

```python
def figure_6_4():
    # Episodes for the run
    eps  = 500

    # Perform x many runs
    runs = 20

    # Initialises empty arrays for the rewards
    q_learning_rewards = np.zeros(eps)
    sarsa_rewards      = np.zeros(eps)

    # Simultaneously runs the Sarsa and Q-Learning algorithim on the same gridworld

    for r in tqdm(range(runs)):
        sarsa_q      = np.zeros((grid_Y, grid_X, 4))
        q_learning_q = np.zeros((grid_Y, grid_X, 4))

        for i in range(0, eps):
            sarsa_rewards[i]      = sarsa_rewards[i] + max(sarsa(sarsa_q), -100)
            q_learning_rewards[i] = q_learning_rewards[i] + max(q_learning(q_learning_q), -100)

    # Takes a 10 point moving average
    sarsa_rewards      = np.convolve(sarsa_rewards, np.ones((10,)) / 10, mode='valid')
    q_learning_rewards = np.convolve(q_learning_rewards, np.ones((10,)) / 10, mode='valid')

    # Average
    sarsa_rewards      = sarsa_rewards / runs
    q_learning_rewards = q_learning_rewards / runs

    # Draw reward lines
    plt.figure(figsize = (10,5))
    plt.plot(sarsa_rewards, label = 'Sarsa', color = '#FFC107')
    plt.plot(q_learning_rewards, label = 'Q-Learning', color = '#1E88E5')
    plt.yticks(np.arange(-100, 0, step = 25))
    plt.xlabel('Episodes')
    plt.ylabel('Sum of rewards during episode')
    plt.legend()
```

P4 Findings

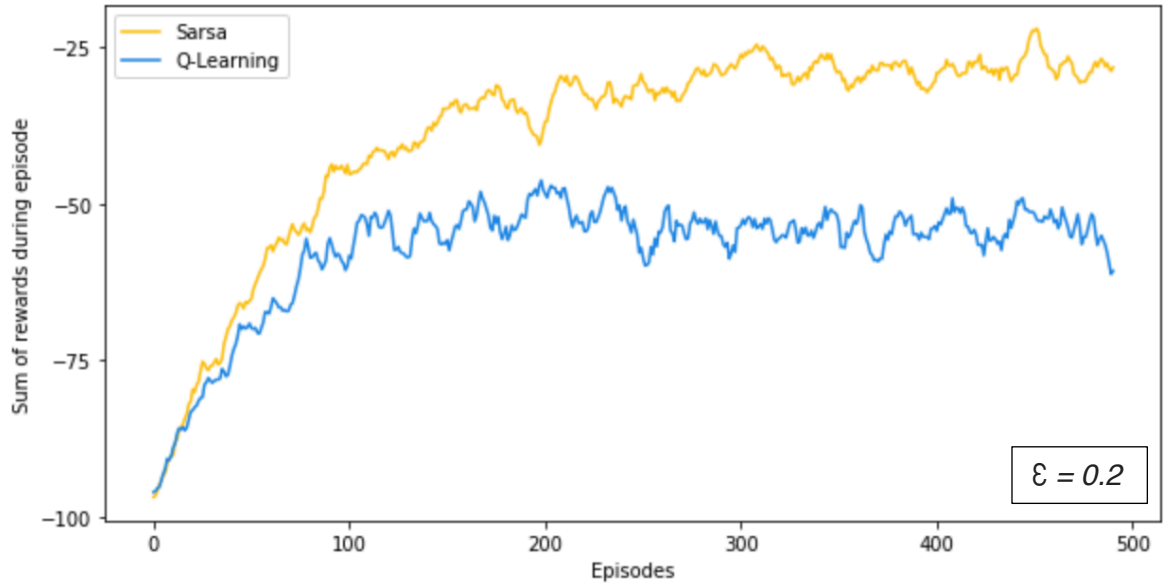
Figure: Comparison of average rewards yielded by Sarsa and Q-learning when applied to the cliff-walking task. ε = 0.1



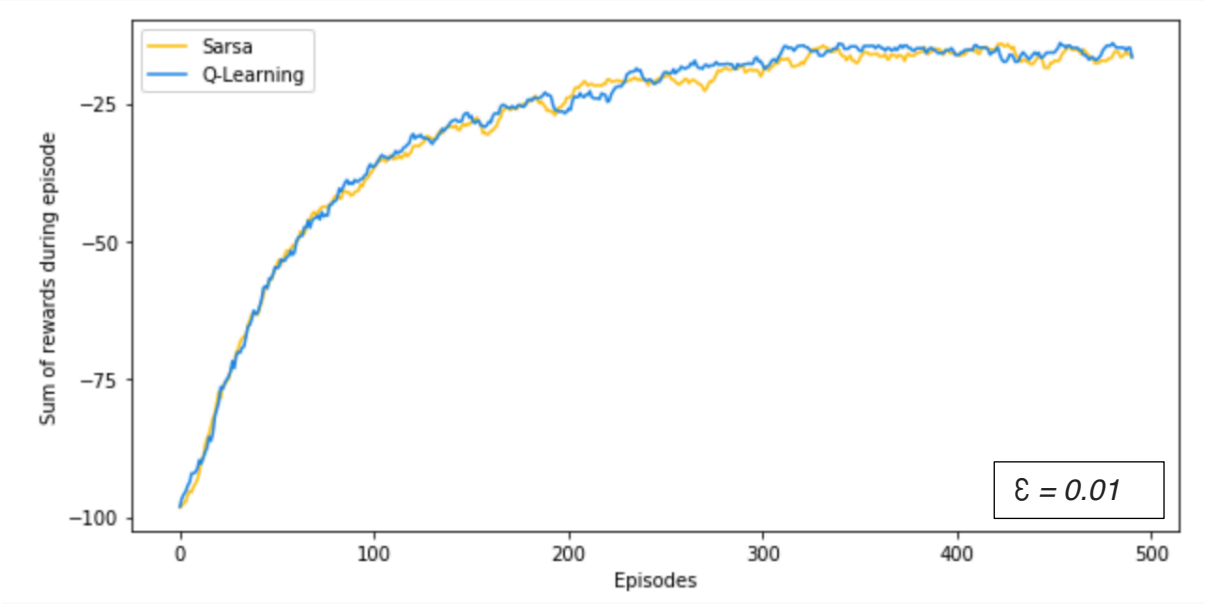
Without careful examination, it would appear that Sarsa converges towards the optimal path, whereas Q-learning converges towards the safe path, however this is not the case.

In actuality, Q-learning tends towards the optimal solution, however, due to the exploration directive, this increases the risk of falling off the cliff, thus incurring the significant reward penalty that comes with it. This explains the downward spikes present in the Q-Learning curve. Conversely, Sarsa yields higher average rewards despite having converged toward a sub-optimal (safe) policy.

By increasing the $\mathcal{E}$-greedy value, we can expect that the lines will be more noisy and show a higher variance in reward. The figure below shows the same test but with an $\mathcal{E} = 0.2$. We can see that both Q-learning & Sarsa tend towards the same solutions as last time, however we see more significant downward spikes. This logically follows from our findings in the previous test; the $\mathcal{E}$-greedy value is useful to find the optimal policy, but is actively unhelpful once an optimum has been found.



The 2 experiments above may suggest that a lower $\mathcal{E}$-greedy value would yield better results. The figure below shows the same experiment but with $\mathcal{E} = 0.01$. In this instance, the Q-learning and Sarsa converge on the suboptimal but safe solution. These examples highlight how even in more sophisticated situations, the exploration/exploitation dichotomy is an issue at the very core of reinforcement learning.



One way of handling this would be for the ε-greedy value to decay over time. In other words, significant exploration would be encouraged early on, but become less likely to occur as the task continues. Therefore, the benefits of exploration would help the agent reach the optimal policy, without hindering it once it has arrived there.

REFERENCES

ADL (2018). *An introduction to Q-Learning: reinforcement learning*. [online] freeCodeCamp.org. Available at: https://medium.freecodecamp.org/an-introduction-to-q-learning-reinforcement-learning-14ac0b4493cc [Accessed 18 Feb. 2019].

Dey, S. (2018). *Some Reinforcement Learning: The Greedy and Explore-Exploit Algorithms for the Multi-Armed Bandit Framework in Python*. [online] sandipanweb. Available at: https://sandipanweb.wordpress.com/2018/04/03/some-reinforcement-learning-the-greedy-and-explore-exploit-algorithms-for-the-multi-armed-bandit-framework/ [Accessed 15 Feb. 2019].

Internet Encyclopaedia of Philosophy (n.d.). Bentham, Jeremy | Internet Encyclopaedia of Philosophy. [online] Available at: https://www.iep.utm.edu/bentham/ [Accessed 19 Feb. 2019].

(Neil G, 2015) (https://stats.stackexchange.com/users/858/neil-g), What is the difference between off-policy and on-policy learning?, URL (version: 2018-11-10): https://stats.stackexchange.com/q/184794

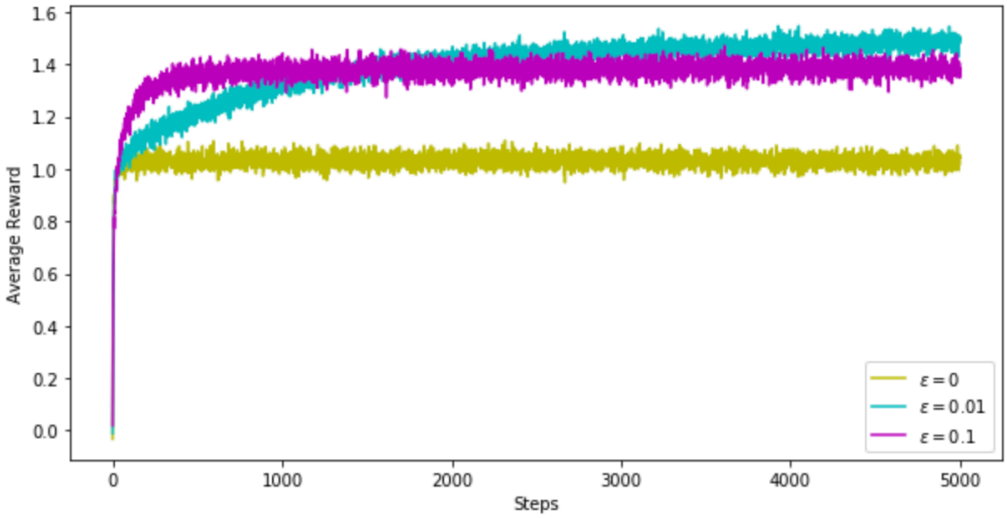
Neto, G. (2005). From Single-Agent to Multi-Agent Reinforcement Learning: Foundational Concepts and Methods. [online] Available at: http://users.isr.ist.utl.pt/~mtjspaan/readingGroup/learningNeto05.pdf [Accessed 19 Feb. 2019].

Silver, D. (2017). *RL Course by David Silver - Lecture 10: Classic Games*. [video] Available at: https://www.youtube.com/watch?v=N1LKLc6ufGY&t=927s [Accessed 21 Feb. 2019].

Sutton, R. and Barto, A. (2018). *Reinforcement Learning: An Introduction*. 2nd ed. Cambridge, MA: MIT Press, p.23,26.

APPENDICIES

Appendix A: *10 armed bandit test over 5000 steps. In this instance, the 1% exploration rate supersedes the maximum reward yielded by the 10% rate.*



Appendix B: *An expanded version of the original testbed, and maximum reward yielded by each policy.*