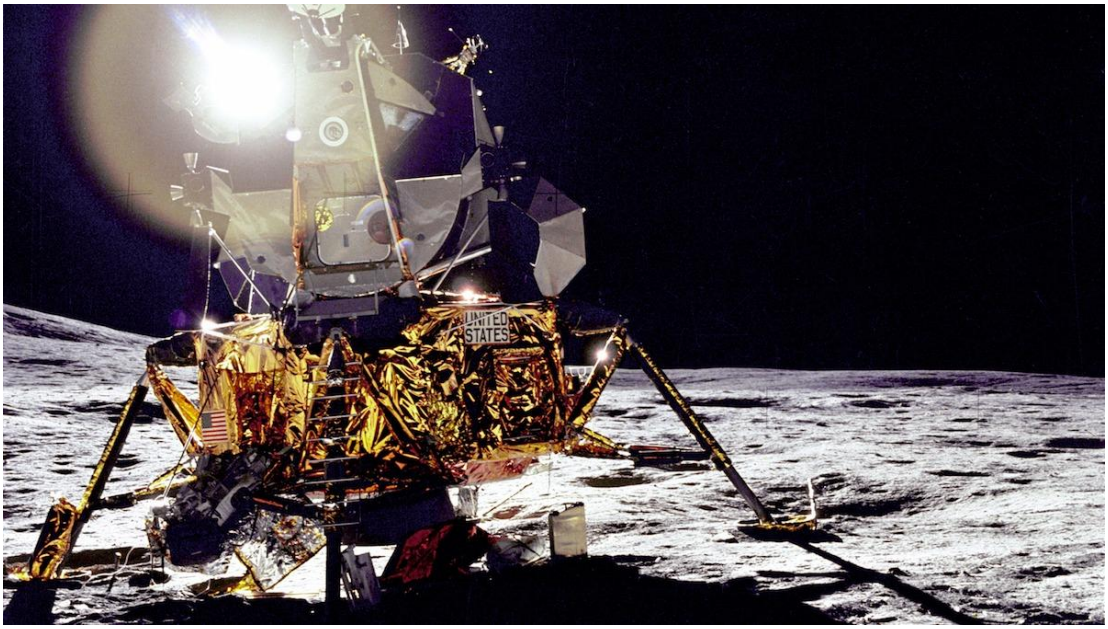
DELE CA2 Part B Submission

Lunar Lander: Learning to land a rocket!

Name	Class	Admin No.
Timothy Chia Kai Lun	DAAA/FT/2B/02	P2106911
Lim Jun Jie	DAAA/FT/2B/02	P2100788



Objectives

We are tasked with training an agent capable of landing the lunar lander safely onto a landing pad as per documentation, using reinforcement learning techniques. The environment is provided to us through the gymnasium library.

We aim to study the different behaviours executed in the Lunar Lander environment and attempt to optimize these behaviours such that we maximize on rewards.

> **Note**: Algorithms in this assigment were implemented with the help of pseudocode from Sutton & Barto's book

---

1. Project Setup

We have written modules for the algorithms used in this assignment as Python scripts including utility classes. These scripts are located in the `.\models` directory. The rendering of episodes will be stored in .gif and can be found in the `.\gifs` directory.

For the deep learning aspect of the asignment, we will be using TensorFlow 2. Weights of the model upon completion of training and history of metrics can be found in `.\assets` and `.\history` respectively.

In [ ]:
```
import os
import json
import numpy as np
import gymnasium as gym
from matplotlib import pyplot as plt

from models.dqn import DQN
from models.sarsa import SARSA
from models.dueling_dql import DuelingDQL
from models.utils import ReplayBuffer, EpisodeSaver
```

---

1. About The Environment

The Lunar Lander is a very interesting environment as it is described as a classic rocket trajectory optimization problem. The environment comes in two versions: discrete and continuous, with differing action spaces and the goal is to safely land the lunar lander on the launch pad located at (0,0).

For this group assignment, we will be applying different RL algorithms on the discrete version environment which contain 4 actions in the action space and the state space is represented as a vector.

1.1 States and Actions

$$\text{State Vector} \begin{cases} s_0 = \text{x-axis coord of agent} \\ s_1 = \text{y-axis coord of agent} \\ s_2 = \text{x-axis linear velocity} \\ s_3 = \text{y-axis linear velocity} \\ s_4 = \text{Agent's angle} \\ s_5 = \text{Agent's angular velocity} \\ s_6 = \text{Right leg touched ground} \\ s_7 = \text{Left leg touched ground} \end{cases} \qquad \text{Action Space} \begin{cases} a_0 = \text{Do nothing} \\ a_1 = \text{Fire left engine} \\ a_2 = \text{Fire main engin} \\ a_3 = \text{Fire right engin} \end{cases}$$

1.2 Reward Scheme

- The agent gains 100-140 points for landing on the launch pad and coming to a rest.
- Coming to a rest yields an additional 100 points.
- Each leg with ground contact earns an 10 points.
- Moving away from landing spot decreases the rewards.
- Crashing decreases the rewards by -100 points.
- Firing the main engine decreases rewards by -0.3 points.
- Firing the side engine decreases rewards by -0.3 points.

An episode is considered a solution when the episodic rewards obtained are greater than or equal to 200 points.

In [ ]:
```
env = gym.make('LunarLander-v2', continuous=False, render_mode='rgb_array')
state_size = env.observation_space.shape[0]
action_size = env.action_space.n
```

---

3. Reinforcement Learning Algorithms

In this assignment, we will be comparing and analysing the differences in application and performance of different reinforcement algorithms.

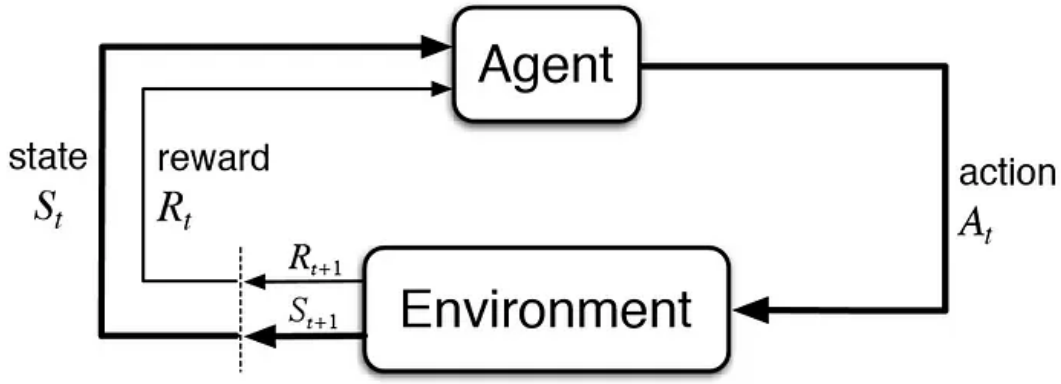
Reinforcement learning problems can be thought of as Markov Decision Processes (MDPs). MDPs are a way of fomarlizing sequential decision making and acts as the basis for structuring problems solved in reinforcement learning.

Components of an MDP

- Agent
- Environment
- State
- Action
- Reward signal

The decision maker, called an agent, interacts with the environment it's placed in. These interactions occur sequentially over time. At each time step, the agent will get some representation of the environment's state. Given this representation, the agent selects an action to take. The environment is then transitioned into a new state, and the agent is given a reward as a consequence of the previous action.

A visual representation of actions, states, and rewards can be seen below:



Iterative feedback loop (Bhatt, 2019)

The goal of an agent is to maximize cumulative future rewards and it's actions are defined in terms of a policy (probability distribution). There are two kinds of policies which we will be exploring:

On-policy

On-policy learning algorithms optimize the same policy (target policy) being used to select actions (behavioural policy) following some strategy (epsilon greedy). This means both target and behavioural policies are one and the same.

The policy that is used for updating and the policy used for acting is the same, unlike in Q-learning. SARSA (State-Action-Reward-State-Action) is an example of an on-policy learning algorithm that we will be implementing for this assignment.

Off-policy

Off-policy learning algorithms are different from on-policy methods because what is being learned/optimized is target policy and not the behavioural policy. Q-Learning is an example of an off-policy algorithm. We will be implementing Deep Q-Learning which is different from vanilla Q-Learning as a neural network is used to map the state vector to Q-values (state-action values).

In other words, it estimates the reward for future actions and appends a value to the new state without actually following any greedy policy.

3.1 Baseline: Deep Q-Learning (DQN)

Q-learning maintains a Q-table that is iteratively updated in order to maximize the reward for every state encountered. This strategy is captured by the Bellman Equation:

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

The optimal state-action value is the current reward and the discounted maximum future reward possible. In the case of finite discrete input spaces, the number of Q values to compute would be finite and can be solved iteratively with dynamic programming:

$$Q_{i+1}(s,a) = E[R_{t+1} + \gamma \max_{a'} Q_*(s',a') \mid s,a], \ \lim_{i \to \infty} Q_i = Q_*$$

However in the case of the Lunar Lander environment, the state representation are continuous vectors and maintaing a Q-table would need infinite space. This is where we use neural networks as function approximators to estimate the Q function:

$$Q(s,a;\theta) \approx Q(s,a)$$

The Deep Q-Network is trained by minimizing the difference between the predicted and actual Q-values (Mean-Squared Error) for the loss function:

$$L_i(\theta_i) = (\overbrace{(R_{t+1} + \gamma \max_{a'} Q(s',a';\theta_{i-1} \mid s,a))}^{\text{Target Q-Value}} - \overbrace{Q(s,a;\theta_i)}^{\text{Predicted Q-Value}})^2$$

Throughout multiple iterations, the Adam optimizer performs gradient descent to optimize the weights of the neural network by minimizing the loss of the Q-values and the target Q-values. The following are a few common concepts employed across our DQN agents.

3.1.1 Epsilon-Greedy Strategy

The idea of exploiting what the agent already knows versus exploring a random action is called the exploration-exploitation trade-off. When the agent explores, it can improve its current knowledge and gain better rewards in the long run. However, when it exploits, it gets more reward immediately, even if it is a sub-optimal behavior. As the agent can't do both at the same time, there is a trade-off.

Using the epsilon-greedy strategy, the agent has some chance to select the action with the highest estimated reward. This strategy dictates that a random action is taken with probability ϵ, and a network based action is taken otherwise.

This approach for action selection can be implemented in Python with a simple conditional statement:

In [ ]:
```
if np.random.random() < epsilon:
    # select action by exploration
else:
    # select action by exploitation
```

3.1.2 Experience Replay

Sequential states are strongly correlated as the actions the lander would take at any presetn state is stringly correlated to actions taken in the past/future. To break correlation, we store a batch of past experiences and randomly select a uniformly distributed sample from this batch.

Our implementation of a replay buffer can be found below:

In [ ]:
```
class ReplayBuffer:
    def __init__(self, max_length, state_size, action_size, is_sarsa=False):
        self.is_sarsa = is_sarsa
        self.memory_counter = 0
        self.max_length = max_length
        self.state_memory = np.zeros((self.max_length, state_size))
        self.new_state_memory = np.zeros((self.max_length, state_size))
        self.action_memory = np.zeros((self.max_length, action_size), dtype=np.i
        if is_sarsa:
            self.new_action_memory = np.zeros((self.max_length, action_size), dt
        self.reward_memory = np.zeros(self.max_length)
        self.done_memory = np.zeros(self.max_length, dtype=np.float32)

    def append(self, state, action, reward, new_state, done, new_action=None):
        idx = self.memory_counter % self.max_length

        self.state_memory[idx] = state

        actions = np.zeros(self.action_memory.shape[1])
        actions[action] = 1.0
        self.action_memory[idx] = actions

        if self.is_sarsa:
            new_actions = np.zeros(self.action_memory.shape[1])
            new_actions[new_action] = 1.0
            self.new_action_memory[idx] = new_actions

        self.new_state_memory[idx] = new_state
```

```
        self.reward_memory[idx] = reward
        self.done_memory[idx] = 1 - done
        self.memory_counter += 1

    def sample(self, batch_size):
        max_memory = min(self.memory_counter, self.max_length)
        sampled_batch = np.random.choice(max_memory, batch_size, replace=False)

        states= self.state_memory[sampled_batch]
        actions = self.action_memory[sampled_batch]
        rewards= self.reward_memory[sampled_batch]
        new_states = self.new_state_memory[sampled_batch]
        if self.is_sarsa:
            new_actions = self.new_action_memory[sampled_batch]
        dones = self.done_memory[sampled_batch]

        if not self.is_sarsa:
            return states, actions, rewards, new_states, dones
        else:
            return states, actions, rewards, new_states, new_actions, dones
```

3.1.3 Network Architecture

Our Q-network contains an input layer expecting batches of state vectors which are of size 8. We use 2 fully connected hidden layers with 256 nodes and ReLU activation each. The output layer is a fully connected layer with 4 nodes that are linearly activated andrepresents the Q-values for each of the 4 actions given a state.

In [ ]:
```
qnetwork = Sequential([
    Dense(units=256, activation='relu', input_shape=(state_size,)),
    Dense(units=256, activation='relu'),
    Dense(units=action_size, activation='linear')
])
```

3.1.4 Hyperparameters

For all models and experiments done in this assignment, we have chosen to train the agent for a total of 500 episodes and maximum of 1000 steps per episode. Due do hardware contraints of our machines, we have chose to store up to 10,000 experiences in the replay buffer to be sampled from.

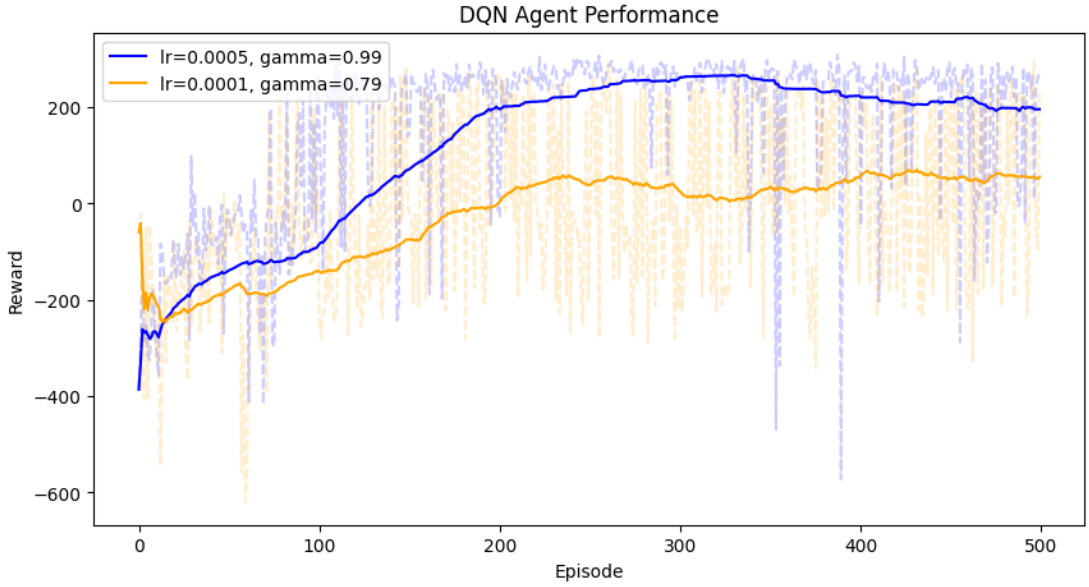
We experimented with higher and lower values of learning rate and discount factor. We chose to keep the exploration rate of 1 as we would want our agent to explore the evironment before starting to learn from past experiences as slowly decay epsilon by a factor of 0.99 with a minimum value of episolon being 0.01.

3.1.5 DQN Agent Performance

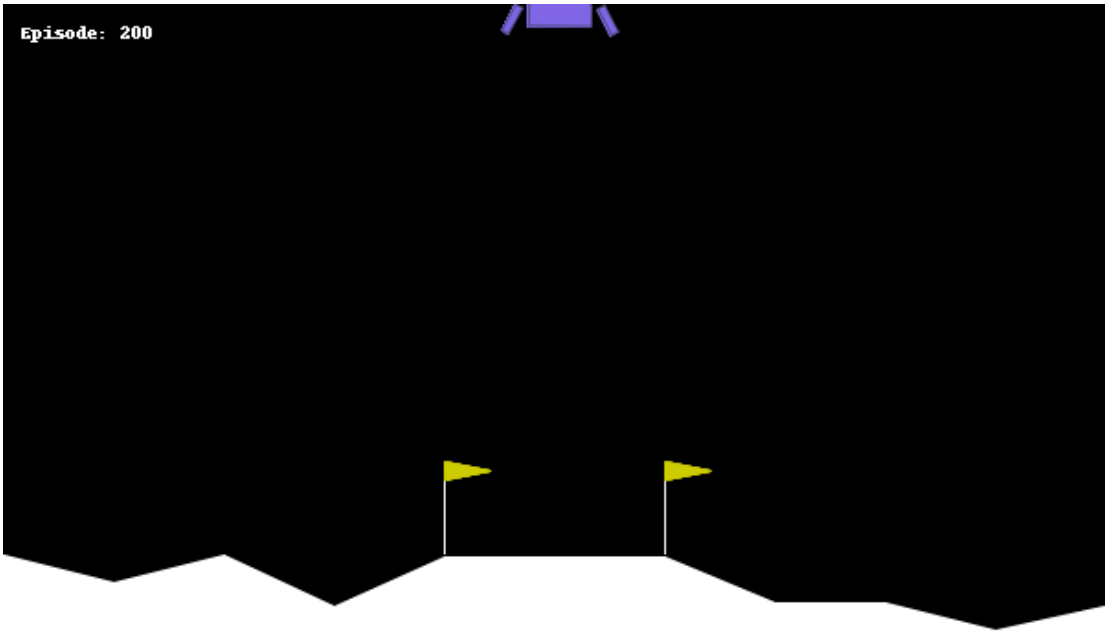
Using a DQN as our baseline, we experimented with different values for learning rate and the discount factor. We found that these hyperparameters do indeed affect the speed at wring the agent adapts to the environment and learns the optimal policy.

We observed that a DQN agent with a lower learning rate and discount factor performed much worse as compared to a DQN agent with higher learning rate and discount factor.

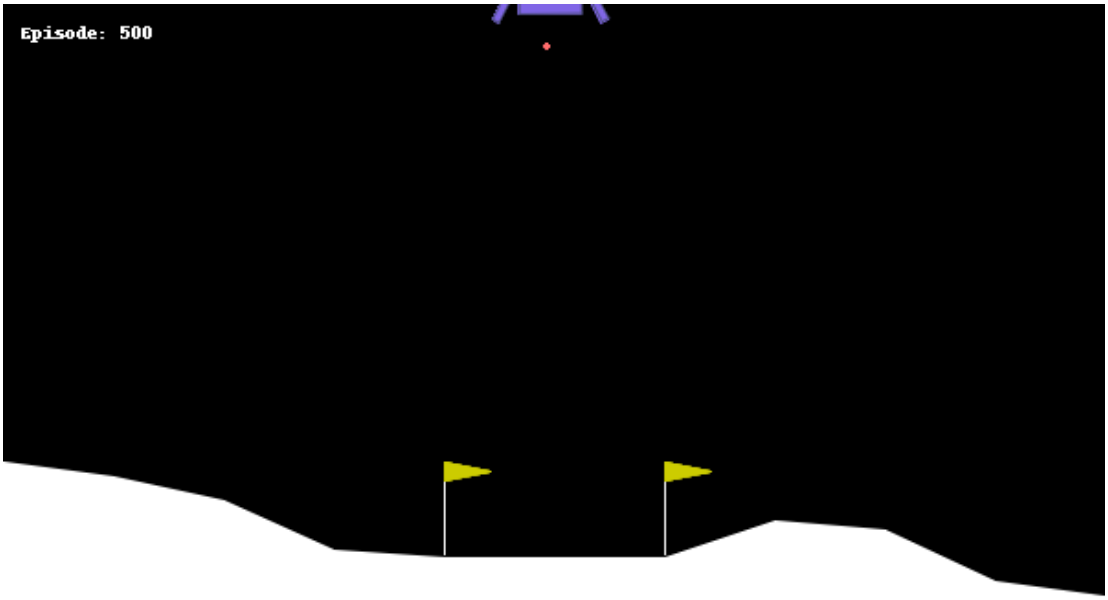
From searching, we have found the best combination of learning rate and discount factor to be 0.0005 and 0.9.



DQN agent with learning rate of 0.0005 and discount factor of 0.99 scored an average of 200 points at about 200 episodes.



However, after training for longer episodes we started to see the average rewards decrease slightly which could indicate that we could stop training early once a solution has been found.



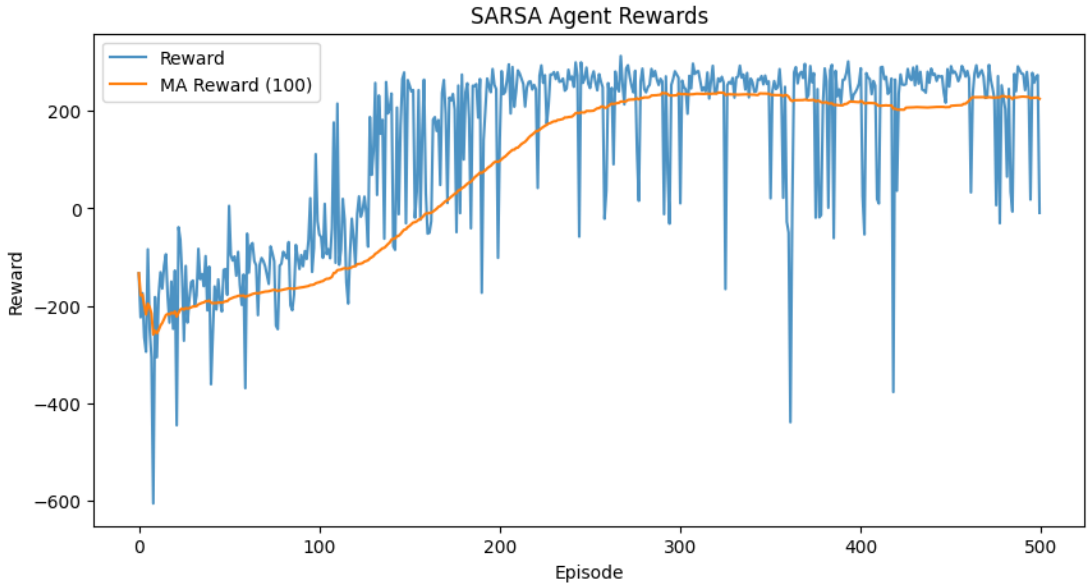
3.3 State-Action-Reward-State-Action (SARSA)

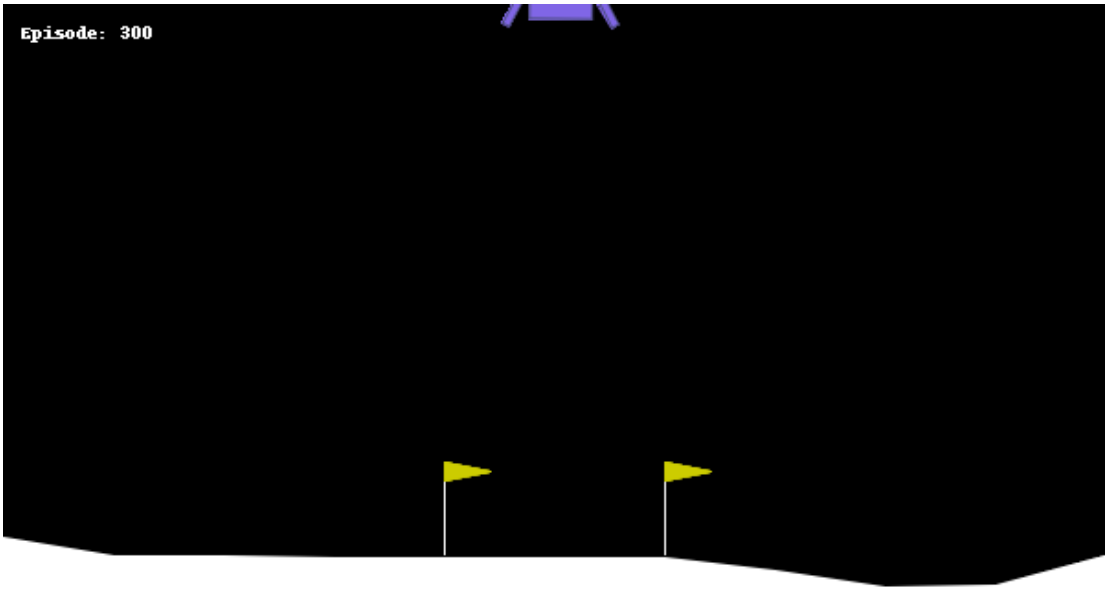
Rather than storing the state, action, reward and new state to be sampled from the replay buffer at each timestep of the episode, we generate and include the next action to be taken in the new state following the epsilon-greedy strategy and include the state, action, reward, new state and new action transition stored.

We implement the SARSA algorithm from pseucode code found in Sutton & Barto's book. The update rule for SARSA is:

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

When the SARSA agent was trained on the Lunar Lander environment, we observe that the agent is also able to solve the environment in about 300 episodes, which is slower than DQN. Although, average rewards of SARSA agent is still lower than the DQN agent.





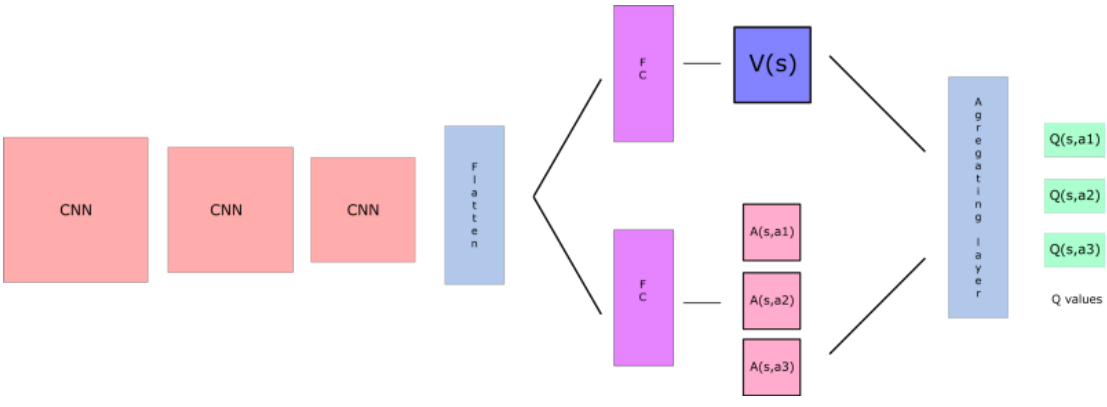
3.4 Dueling Deep Q-Learning (DDQN)

The Dueling DQN was first proposed in a paper in Dueling network architectures for deep reinforcement learning (Wang et al., 2016). The architectural change utilized two separate streams to explicitly separate the representation of state values and state-dependent action advantages.The two streams represent the value and advantage functions, and share a common feature learning module.

The advantage of the Dueling DQN is its ability to learn the value of the states without the need to learn the effect of each action for each of these states.

3.4.1 Network Architechture

> **Note**: The diagram below uses convolutional layers. However, since the state representation is a vector, we have adjusted the architecture to use fully connected layers instead.

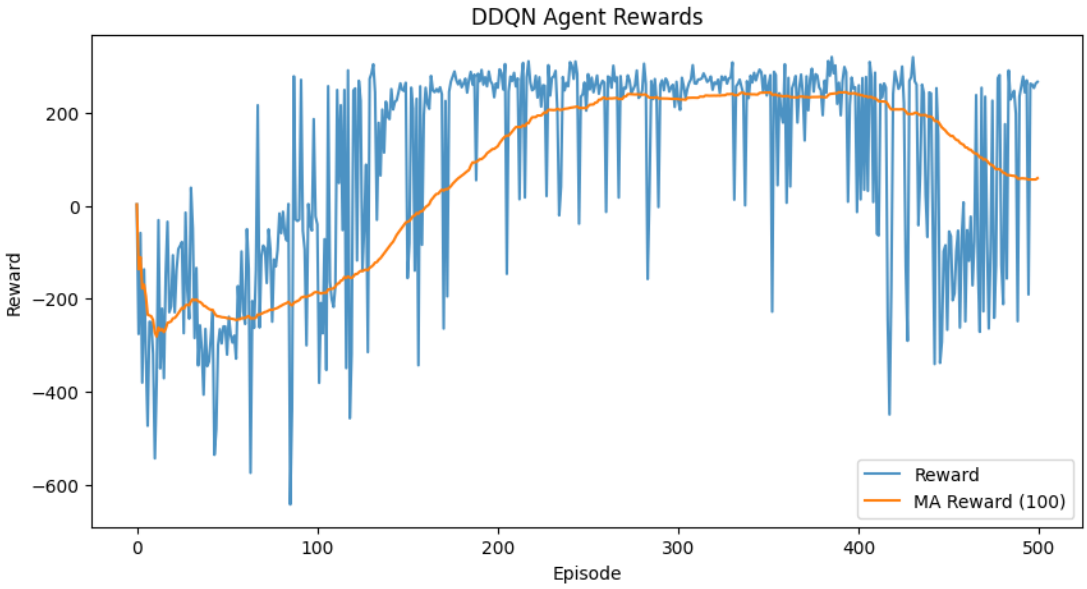


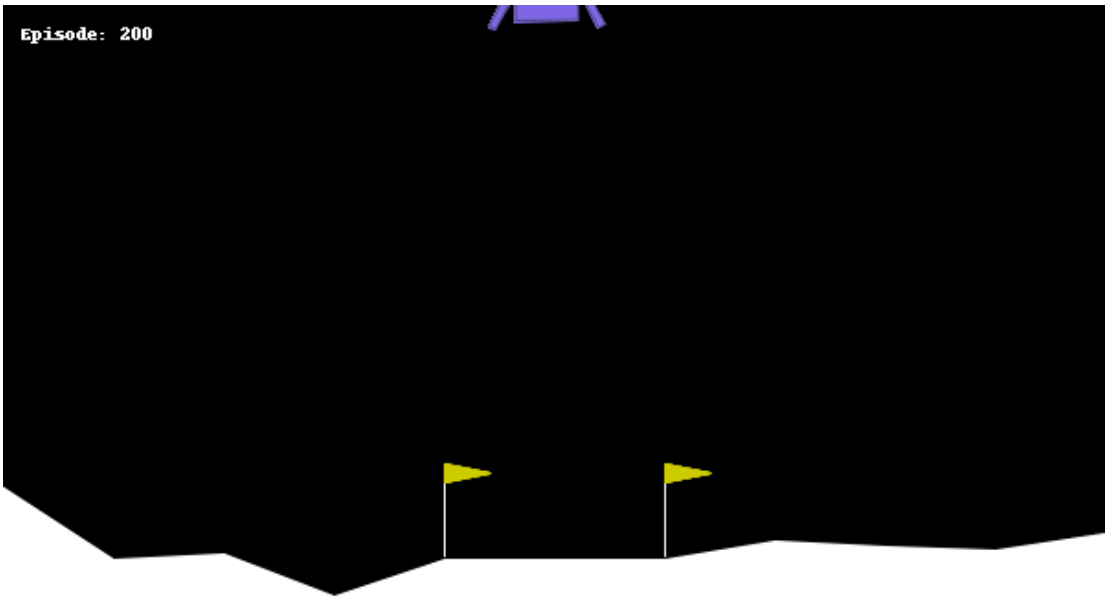
Dueling Architecture, (freeCodeCamp.org, 2018)

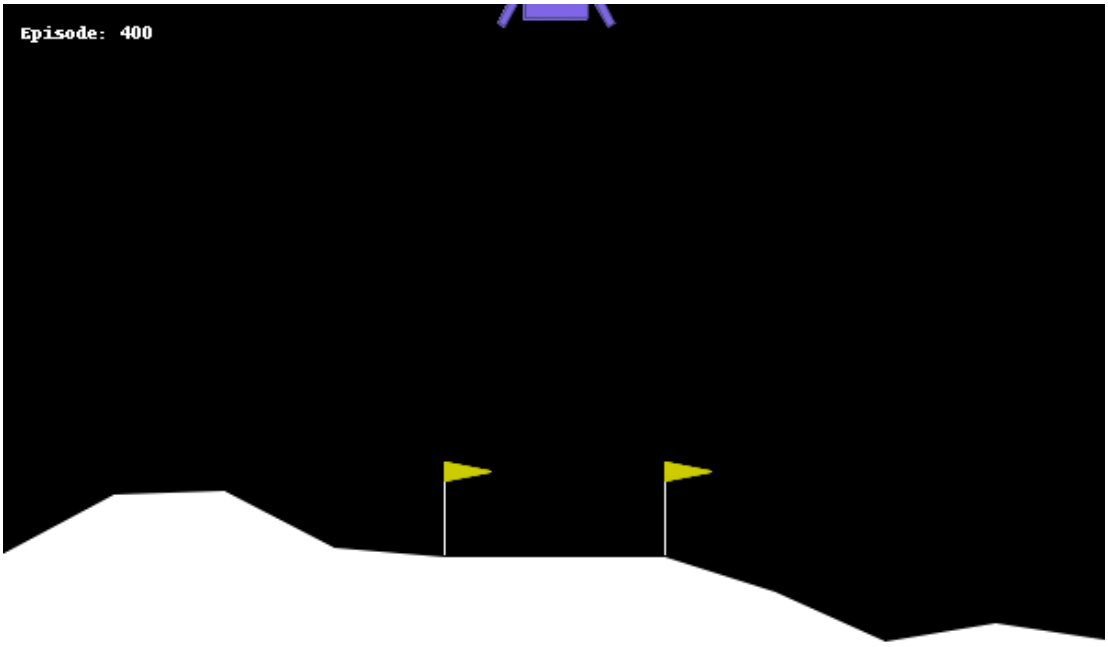
The state-value and advantage streams are aggregated through a forward mapping:

$$Q(s,a;\theta,\alpha,\beta) = V(s;\theta,\beta) + A(s,a;\theta,\alpha)$$

The Dueling DQN agent converges at around 200 episodes as well. However, at 400 episodes it reached its highest average reward of 239.







---

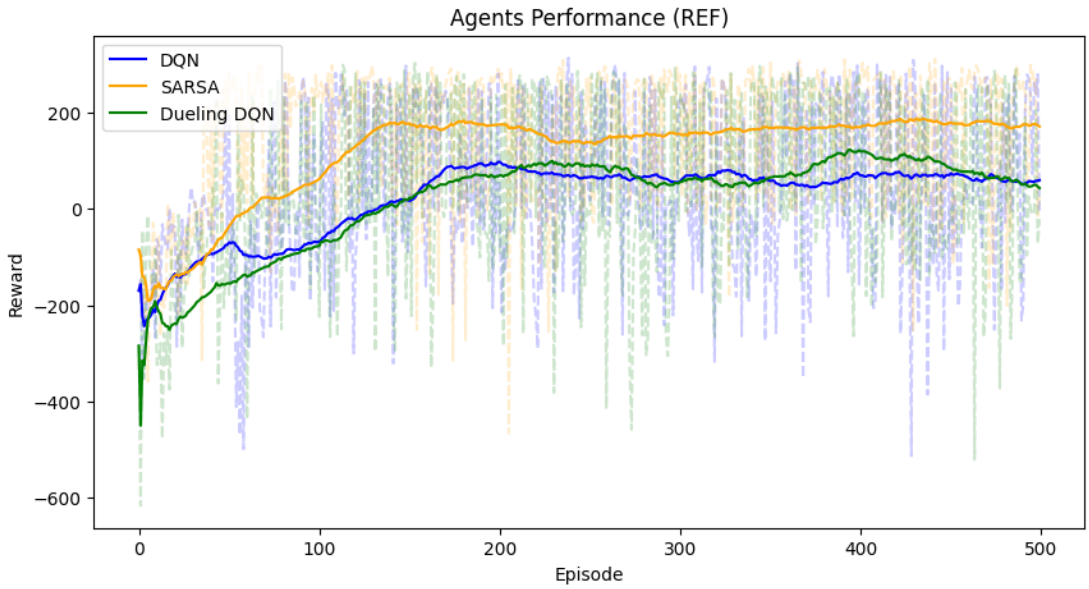
4. Experiments

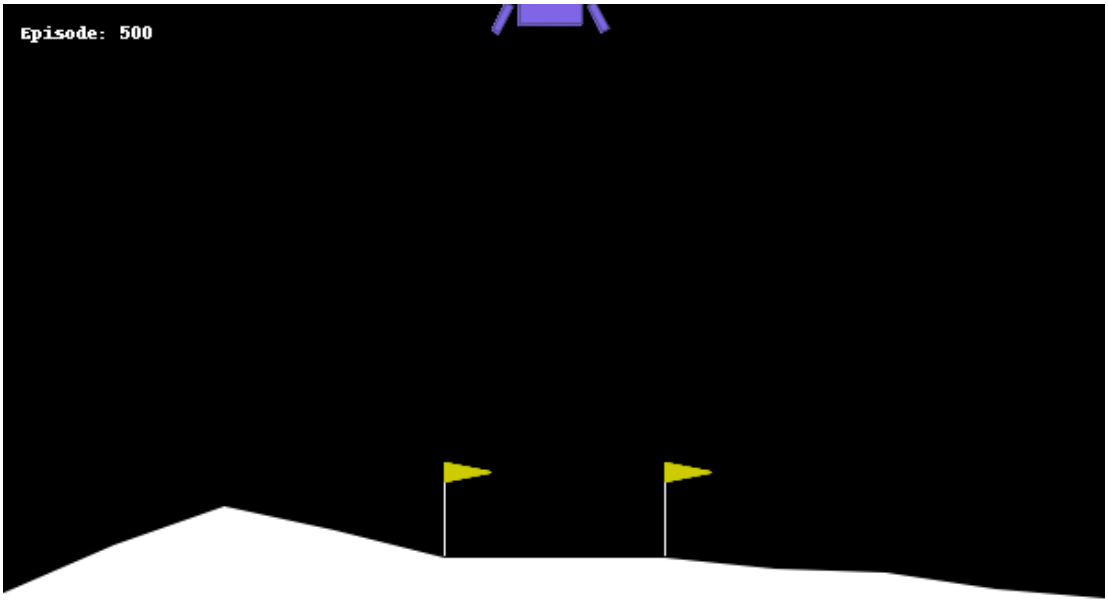
A creative component of this assignment was how we changed the environment to incorporate the concept of engine failure. We wanted to test the robustness of the agents in a noisy environment such as this.

We implemented this change to the environment to simulate engine failure that happens in the real world by allowing an 80% chance that the actions the agent chose follow through, while there is a 20% chance the engine shuts off/does nothing.

We can see the agents performance under this noisy environment from the plot below. It can be noted that all three agents were not able to converge to an optimal policy as the average rewards gained over 500 episodes never reached 200 or above.

However, it is interesting to see that the SARSA agent was able to still reach an average reward of 186 points after about 430 episodes. This could be explained because SARSA is an on-policy learning algorithm, which means that it updates the action-value function based on the action that is actually taken, rather than the best possible action. This makes it more robust to noise and less likely to be thrown off track by chance events.





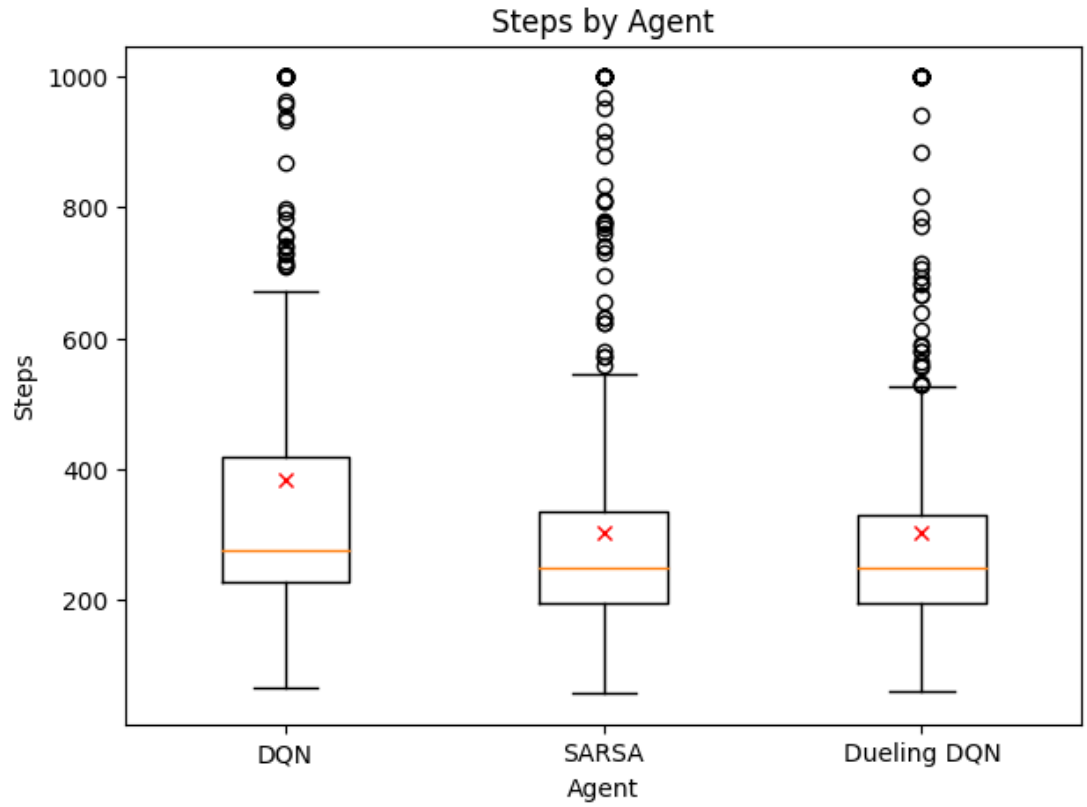
SARSA agent in noisy environment

---

5. Evaluation

From trying different algorithms to solve the rocket trajectory task of the base lunar lander environment, we can see that a standard DQN is sufficient and scores much higher than the other models in terms of average rewards.

The mean steps the DQN agent takes is much higher than the other agents, suggesting that the agent performs more cautiously.



---

6. Conclusion

In conclusion, we have implemented three different reinforcement learning algorithms and have found the DQN agent to be the best out of three due to the higher aveerage rewards received after 500 episodes and also that it takes more steps in order to safely land itself on the landing pad.

In the case of a noisy environment, we have found that on-policy methods like SARSA perform much better although never reaching a solution. However, it remains to be seen if longer training episodes will see the algorithm converge to an optimal policy.

Given more time, we would like to try more methods to better improve or speed up converge of each algorithm such as:

- State discretization
- Prioritized Experience Replay
- Increased Memory Capacity
- Adjust Training Episodes
- Hyperparameter Tuning

---

7. References

- Sutton, R.S. and Barto, A.G. (1992) Reinforcement learning: An introduction, Reinforcement Learning: An Introduction. Available at: http://www.incompleteideas.net/book/the-book-2nd.html (Accessed: January 31, 2023).
- Plaat, A. (2022) Deep Reinforcement Learning, a textbook, arXiv.org. Available at: https://arxiv.org/abs/2201.02135 (Accessed: January 31, 2023).
- Bhatt, S. (2019) Reinforcement learning 101, Medium. Towards Data Science. Available at: https://towardsdatascience.com/reinforcement-learning-101-e24b50e1d292 (Accessed: February 5, 2023).
- Wang, Z. et al. (2016) Dueling network architectures for deep reinforcement learning, arXiv.org. Available at: https://arxiv.org/abs/1511.06581 (Accessed: February 5, 2023).
- freeCodeCamp.org (2018) Improvements in deep Q learning: Dueling Double DQN, prioritized experience replay, and fixed..., freeCodeCamp.org. freeCodeCamp.org. Available at: https://www.freecodecamp.org/news/improvements-in-deep-q-learning-dueling-double-dqn-prioritized-experience-replay-and-fixed-58b130cc5682/ (Accessed: February 5, 2023).