

**Contents**

1. Lunar Lander Implementation..... 2

    1.1 Implementation Overview..... 2

    1.2 Features and Variables Used ..... 4

    1.3 Results and Discussion ..... 5

        1.3.1 Loss Function..... 5

        1.3.2 Learning Rate..... 6

        1.3.3 Epsilon Value..... 7

        1.3.4 Final Model ..... 8

2. Exploration and Exploitation for Deep Reinforcement Learning ..... 9

References ..... 11

# 1. Lunar Lander Implementation

Lunar lander is a reinforcement learning environment provided by OpenAI Gym, where a moon lander tries to land safely on the landing pad on the surface of the moon. The lander takes actions from an action space and aims to land on the landing pad in the shortest time possible. The lander earns rewards from each action chosen and the subsequent interaction with the environment, and the ultimate objective is to optimise the landing process to maximise the rewards earned.

## 1.1 Implementation Overview

In this project, we have used Tensorflow and Keras to implement deep reinforcement learning for the Lunar Lander environment available in OpenAI Gym. OpenAI Gym is a toolbox in python which contains a lot of different simulated environments to test and compare reinforcement algorithms. This project focuses on implementing the Lunar Lander environment, which is one of the most popular environments for deep reinforcement learning and is often used for evaluating the performance of different learning algorithms.

The premise is to land a lunar lander on the surface of the moon on a landing pad indicated by two flags. Each episode starts with the lander coming down from the top to the ground due to gravity. There are three engines on the lander - main engine, left engine and right engine - which have to be used to help the lander touchdown on the landing pad. The objective is to land safely with zero speed i.e without using any engine, on the landing pad. The reward for achieving this is 100 to 140 points. The reward for each leg touching the ground is 10 points. Using the main engine leads to a reward of -0.3 per frame and using the other two engines lead to a reward of -0.03 per frame. The reward if the lander crashes or comes to rest safely on the ground is -100 and +100 respectively, and this is taken as the end of the particular episode. The episode can be considered over also when the lander goes out of the defined environment of the game. The benchmark to deem the game a success is taken to be 200 points.

The agent in this task is the lander. The action space of this agent consists of 4 actions:

- Doing nothing
- Firing the main engine
- Firing the left engine
- Firing the right engine

The observation space or the state is an 8-dimensional vector, which denotes the coordinates (in x and y) of the lander, the linear velocity (in x and y) of the lander, the angle of the lander made with the ground, its angular velocity, and two boolean values indicating whether each leg is touching the ground or not.

The model is trained using a Deep Q-learning reinforcement learning algorithm which involves the mapping of a state-action pair to a Q-value using a neural network. The architecture of this algorithm consists of two neural networks, Q-network and a Target network, and a component known as an Experience Replay. The Experience Replay interacts with the environment by selecting an action using the  $\epsilon$ -greedy approach and getting the reward and the next state associated with that. Each experience consists of the current state, action taken, reward received, and next state observed. This data, for many experiences, is stored in a replay buffer as the training data on which the Q-network will be trained later. The training data is shuffled and a random batch is fed into the two networks. The Q-network is a neural

network which takes in the current state as input and outputs the estimated Q-values for each possible action. The Q-network trains on the data and estimates the Q-values for all actions for the current state, which are known as the predicted Q-values. The Target network gets the next state from all data samples and gives the best Q-value among all the actions from that state. This value, in addition to the reward from the sample is known as the target Q-value. The models are created as seen in Figure 1.

```
# Creating a model and a target model
def _createModels(self):
    model = Sequential([
        Input(shape=self.stateSpace),
        Dense(self.batchSize, activation=self.hiddenActivation),
        Dense(self.batchSize, activation=self.hiddenActivation),
        Dense(self.actionSpace, activation=self.outputActivation)])

    targetModel = Sequential([
        Input(shape=self.stateSpace),
        Dense(self.batchSize, activation=self.hiddenActivation),
        Dense(self.batchSize, activation=self.hiddenActivation),
        Dense(self.actionSpace, activation=self.outputActivation)])

    return model, targetModel
```

Figure 1: Creating the models

The predicted Q-values for the action taken is computed using the Bellman equation presented in (1), and the weights of the neural network are updated using backpropagation to minimise the difference between the target Q-values and the predicted Q-values. The predicted Q-values from the Q-network, the target Q-values from the Target network and the reward associated with each state-action pair are used to calculate the loss for training the Q-network at each timestep. This is done over many timesteps over many episodes. The code implementation of equation (1) is seen in Figure 2.

$$Q^*(s_t, a_t) = r_t + \gamma(1 - d)\max_a Q(s_{t+1}, a_{t+1}) \quad (1)$$

```
def computeLoss(self, events):
    states, actions, rewards, nextStates, dones = events

    # Computes the max Q value of the next state
    maxQ = tf.reduce_max(self.targetModel(nextStates), axis=-1)

    # Bellman Equation
    yPred = rewards + (self.gamma * maxQ * (1-dones))
    qValues = self.model(states)

    # Selects qValues at the index of the corresponding action
    batch_size = tf.shape(qValues)[0]
    action_indices = tf.stack([tf.range(batch_size), tf.cast(actions, tf.int32)], axis=1)
    qValues = tf.gather_nd(qValues, action_indices)

    # Selected loss function
    h = tf.keras.losses.Huber()
    loss = h(yPred, qValues)

    return loss
```

Figure 2: Implementation of the Bellman Equation

An important thing to note is that the Target network is not trained. In spite of this, it has a relevant role in the training of the Q-network to obtain the optimal Q-values for all state-action pairs. In the absence of a Target network, the optimal direction of change for the predicted Q-values is likely to change after each update of the weights of the Q-network. This would mean that the ‘target’ Q-values fluctuate with each timestep and thus would make it difficult to reach the optimal Q-values. Hence, by keeping a Target network which does not get trained, the target Q-values can be kept constant, while the Q-network gets trained and the predicted Q-values become closer to the target Q-values. However, even these target Q-

values are not optimal and have to be improved, and hence they are updated after every certain number of timesteps by copying the Q-network weights to the Target network.

```
def updateNetwork(self):
    # Performs a soft update of the target model weights using the model weights and a softUpdate parameter
    for weightTarget, weightModel in zip(self.targetModel.weights, self.model.weights):
        newWeight = self.softUpdate * weightModel + (1.0-self.softUpdate) * weightTarget
        weightTarget.assign(newWeight)

    # Updates the weights based on batches of events
    def fitting(self, events):
        # Automatic computation of gradients to update the weights
        with tf.GradientTape() as tape:
            loss = self.computeLoss(events)

        # Loss value computed with respect to trainable variables
        gradients = tape.gradient(loss, self.model.trainable_variables)
        self.optimizer.apply_gradients(zip(gradients, self.model.trainable_variables))

    self.updateNetwork()
```

Figure 3: Training functions

Figure 3 shows a snippet of the training helper functions. The training process involves running multiple episodes, where each episode starts with a new randomly initialised state and continues until the game is over. The total reward obtained during training is printed at step-intervals to monitor the progress of the training.

## 1.2 Features and Variables Used

The Adam optimizer is commonly used in deep reinforcement learning and combines the advantages of both Adagrad and RMSprop characteristics. Adagrad optimizers alter the learning rate for each weight in the network based on gradient history. However, it suffers from diminishing learning rates, resulting in sluggish convergence. RMSprop addresses the problem of diminishing learning rates by employing a moving average of squared gradients, which dampens learning rates for heavier weights. However, it may still encounter the issue of noisy updates.

As a result, the Adam optimizer combines the advantages of Adagrad and RMSprop by keeping a moving average of the gradient and squared gradient and integrating a bias correction term to adjust for the initial bias estimations. This not only enables Adam to adaptively alter the learning rates for each weight based on the gradient history, but it also dampens the learning rates for the heavier weights. This improves the overall performance of reinforcement learning by increasing the pace of convergence and being computationally efficient.

The target network weights are updated gradually rather than being abruptly replaced with those of the training network when soft updates are used. This approach reduces the likelihood of the target network overfitting to the training data, which can happen when the target network weights are modified too frequently. This improves training performance and steadiness. How much the target network weights are updated to match the training network weights is controlled by the soft update parameter in the soft update algorithm. The target network weights are updated more quickly when a soft update is close to 1, and more slowly when it is close to 0. In this project, a soft update of 0.001 is used.

The Huber loss function is suitable for machine learning problems with regression, especially when dealing with noisy data and outliers. It combines the benefits of both the mean squared error (MSE) and the mean absolute error (MAE), penalising large errors more severely than the former while being more tolerant of tiny errors than the latter.

The discount factor, gamma, is used to determine the importance of future rewards in the agent's decision-making process. The value of gamma ranges from 0 to 1, where a higher gamma value places more emphasis on future rewards, and a low gamma value places more importance on immediate rewards. Hence, gamma is the factor that allows the agent to trade off future and immediate rewards. The discount factor gamma is set to 0.99, which means that future benefits are reduced by a factor of 0.99. This value is commonly used in many reinforcement learning implementations, thus it was chosen for this assignment

The epsilon value determines the probability of selecting a random action, the exploration action, instead of selecting the action with the highest Q-value, the exploitation action. Selecting a greedy action is said to be exploiting or optimising the values of the actions up to that time step. When the agent selects another action other than the optimal one, it is said to be exploring and improving its estimates about the values of the non-greedy actions. The probability of exploration will affect the training and performance of the agent. Hence, three epsilon starting values are tested: [1, 0.1, 0.01]. When the epsilon value is 1, that suggests that the agent will execute a fully random exploration for the best policy while the agent executes an  $\epsilon$ -greedy approach for epsilon of 0.01 and 0.1.

Exploration is necessary in order to find the best policy. However, as the agent learns and gains experience, it is desirable to reduce the exploration rate in order to exploit the learnt policy. By applying a decay factor to the epsilon value at each time step, the epsilon value gradually decreases the exploration rate over time.

## 1.3 Results and Discussion

The goal of the game is to achieve a reward of 200, which signifies that the lander manages to land on the moon successfully. During the development phase, it is set that the environment is solved when the average reward of the past 100 episodes is more than or equal to 200. The averaging of rewards ensures that the model is well-trained and does not accidentally solve the environment by chance. The model development was executed using Google Colab's CPU.

### 1.3.1 Loss Function

Two loss functions, Huber and MSE, were tested to determine the most appropriate function to be used with the lunar lander environment. The results of the performances are shown in Table 1 below.

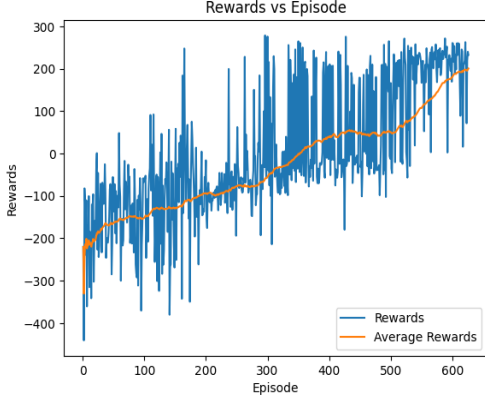
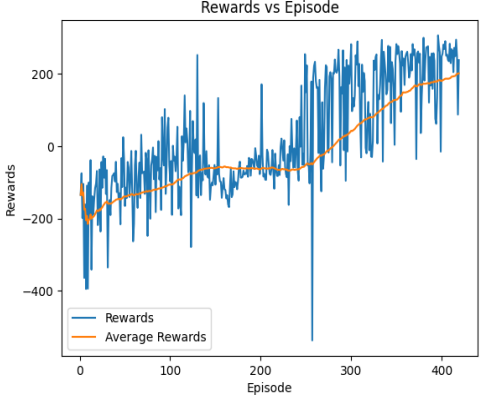
Loss Function	MSE	Huber
Graph		
Number of episodes	626	419

Table 1: Comparison of Loss Functions

Using either of the two loss functions as the loss function in the reinforcement learning model leads to a steady increase in the average reward with the increase in the number of episodes. From Table 1, although there are more episodes that achieve a reward score of more than 200 in the case of MSE, the average reward of the past 100 episodes only reaches more than 200 in the 626th episode. There are also huge variations in rewards throughout the training of the model. On the other hand, the model utilising the Huber loss manages to achieve an average reward of more than 200 at the 419th episode. This, however, may not mean that Huber loss is the better one to use on all similar tasks as it could also be due to the model having a higher starting reward as compared to the MSE model, which allowed the Huber model to achieve the goal faster.

### 1.3.2 Learning Rate

The learning rate is an important hyperparameter in reinforcement learning that governs the speed and stability of the model's weight changes during training. A fast learning rate causes the model to converge quickly, but it is less stable and more prone to overshooting the optimal answer. A lower learning rate may be more stable, but it may also result in slower convergence or becoming stuck in a local minimum. Three learning rates are tested, and their results are shown below while using the Huber loss function.

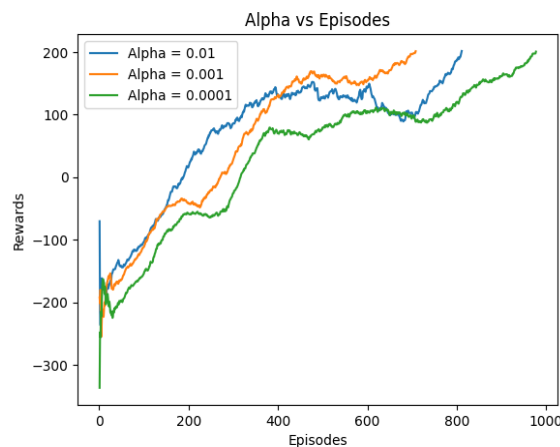


Figure 4: Graph of Alphas vs Episodes

Alpha	0.01	0.001	0.0001
Number of episodes	812	709	978

Table 2: Alpha tested and the number of episodes to reach goal

From Figure 4 and Table 2, it is seen that the learning rate of 0.001 achieves the goal in the shortest number of episodes compared to the other learning rate values. Although all three learning rates display similar trends as seen in Figure 4, the rate of change of the graph where alpha is 0.001 is the steepest from episode 1 to approximately the 450th episode. Figure 5 shows the plot of the individual episode rewards and average rewards when the alpha is 0.001.

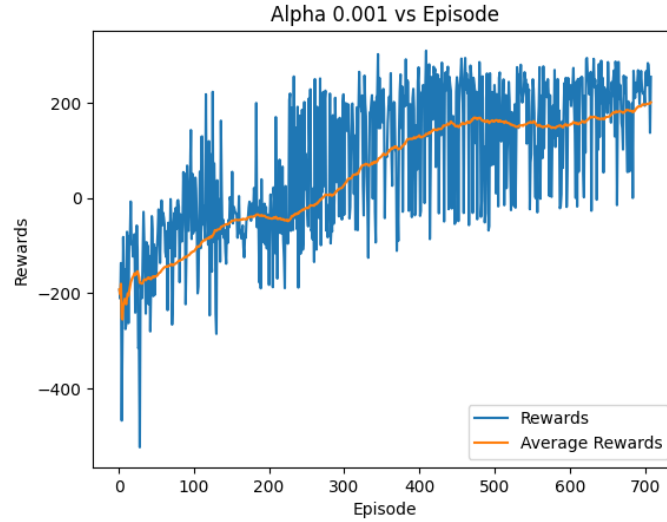


Figure 5: Best alpha vs Episodes

### 1.3.3 Epsilon Value

The starting epsilon value is an important parameter for managing the balance between exploration and exploitation throughout training as it slowly decays based on the epsilon decay rate. The results on three different starting epsilon values, while using the Huber loss and learning rate of 0.001, are presented in Figure 6 and Table 3 below.

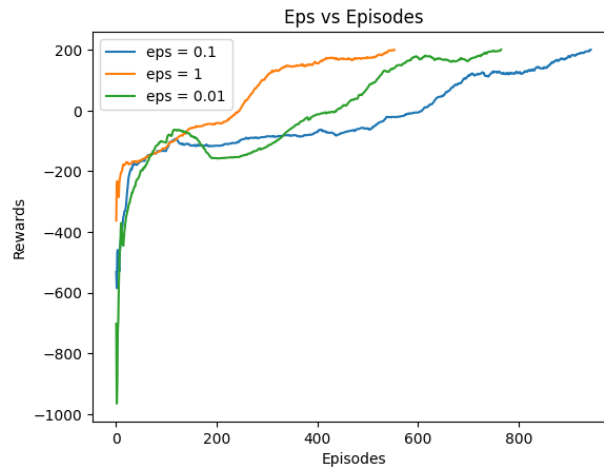


Figure 6: Starting epsilon values vs Episodes

Starting epsilon values	0.1	1.0	0.01
Number of episodes	944	554	766

Table 3: Starting epsilon values tested and the number of episodes to reach goal

The initial epsilon value of 1.0 produces the greatest results in terms of completing the goal in the fewest episodes feasible, out of the three evaluated. The model's initial random exploration allows it to quickly find an optimal policy and exploit it when epsilon decay decreases the epsilon value below 1.0. This allows the reward gained in earlier episodes to be higher than in other models, allowing the model to attain the goal faster. Figure 7 shows the rewards from the individual episodes and the average rewards obtained from the training phase when the initial epsilon is taken to be 1.

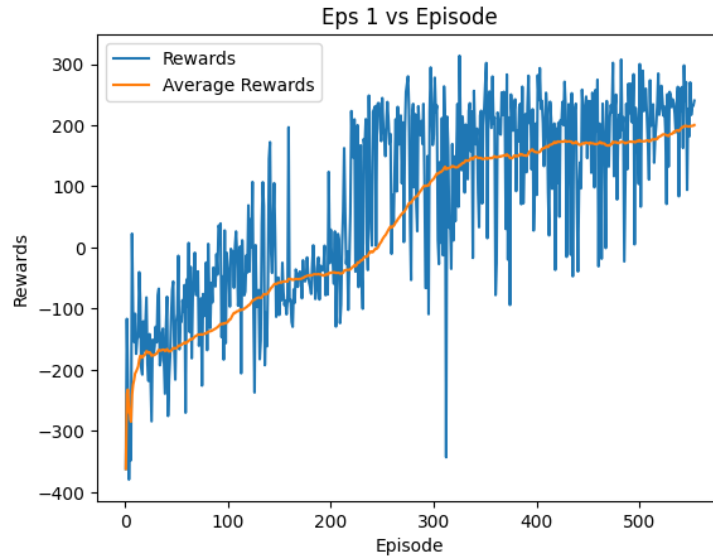


Figure 7: Best starting epsilon value vs Episodes

### 1.3.4 Final Model

The best model tested is using Huber loss with a learning rate  $\alpha = 0.001$  and a starting  $\epsilon = 1.0$ . The following graph in Figure 8 is the result of using the optimized hyperparameters tested.

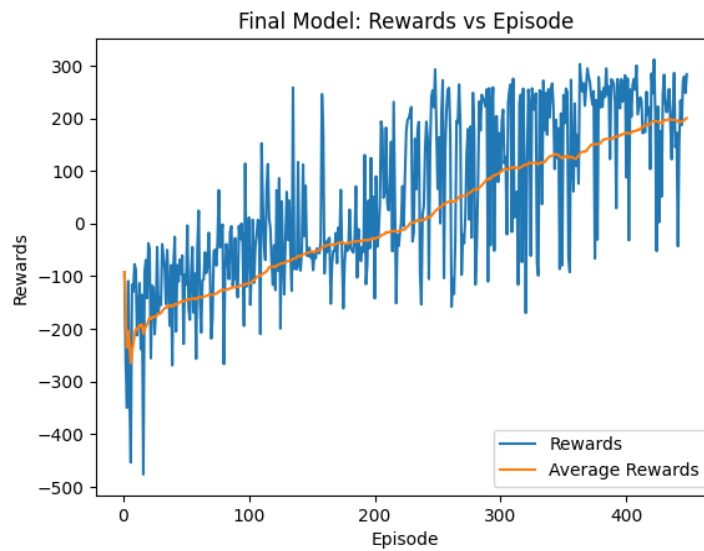


Figure 8: Final model results



The final model reaches the goal in 448 episodes with a training time of 45 minutes running on Google Colab's CPU.

## 2. Exploration and Exploitation for Deep Reinforcement Learning

The objective of a reinforcement learning agent is to compute the optimal policy that maximises the total reward earned over time. Therefore, finding the right trade-off between exploitation and exploration is fundamental to achieving optimal results in deep reinforcement learning. In order to maximise returns, an agent must choose actions that produce the highest reward. As the agent has no prior knowledge about the reward values for each action, the agent needs to perform different actions to discover the optimal choice. [5] A deep reinforcement learning algorithm balances exploitation and exploration by using its current policy to exploit the best action and adding random noise to explore other non-optimal actions which could lead to better long-term payoff. [3]

Exploitation occurs when the greedy action, which is the action with the highest estimated reward, is chosen. This relies on the agent's current knowledge of the actions that have been performed previously and the estimated rewards associated with those actions. While exploitation maximises the reward in the short term, it can lead to suboptimal results in the long run as the action with the actual highest reward may never be discovered. Hence, it is important for the agent to explore other possible actions to determine whether they are optimal or not. [5]

Exploration is the process of selecting new actions that have not been previously performed in order to gain a better understanding of the environment. By exploring, the agent is able to improve its estimates for the reward values of each action over time. Exploration can be achieved by adding random noise to actions or through exploration strategies like  $\epsilon$ -greedy. This results in the occasional selection of random actions which allows the agent to discover new actions that are potentially better. [3]

The greedy method is a pure exploitative approach where the agent always chooses the action with the largest estimated reward value. The greedy algorithm always exploits and never explores, since it prioritises the selection of actions that maximise the short-term payoff. This can be seen as a limitation in problems where the actual reward values are not known to the agent, which would require the agent to explore all the actions.

As mentioned earlier, a common exploration strategy is the  $\epsilon$ -greedy method. The  $\epsilon$ -greedy method balances exploitation and exploration by choosing greedy actions based on the agent's current knowledge while also exploring other actions occasionally through random selection. [2] The probability of exploring random actions is determined by  $\epsilon$  and the probability of exploiting the action with the maximum reward value is  $1 - \epsilon$ . In order to determine the best trade-off between exploration and exploitation, the parameter needs to be experimented with different values.

Q-learning is a well-known off-policy temporal difference (TD) control algorithm in reinforcement learning. An off-policy algorithm is one where the agent uses two different policies. One policy is used for selecting actions and generating episodes while the other policy is improved iteratively in order to find the optimal policy. Q-learning uses two policies, an  $\epsilon$ -greedy policy, and a greedy policy since it is an off-policy method. The  $\epsilon$ -greedy policy is used for action selection, involving both exploitation and exploration. The greedy policy always exploits and is used while computing the estimated reward value, also called the Q value, of the next state-action pair, which in the case of this assignment is done by the

Target network. [2] One limitation of Q-learning is its use of tables to record Q values which is inconvenient for large state spaces. This problem can be addressed by using a popular deep reinforcement learning algorithm called deep Q-learning. [4]

The deep Q-learning (DQL) algorithm is a variation of Q-learning that uses a deep neural network to approximate the Q values. The concept of exploitation and exploration in deep Q-learning is similar to that in Q-learning, with slight differences in the application of the exploration strategies. In addition to exploration strategies like adding noise and  $\epsilon$ -greedy, techniques such as experience replay can be used to explore a minibatch of experiences from the replay buffer through random sampling to train the network. [2,4] This allows the agent to explore and exploit the environment more effectively.

By finding the right balance between exploitation and exploration using methods such as deep Q-learning and  $\epsilon$ -greedy, deep reinforcement learning agents can learn optimal policies and obtain optimal results in complex environments.

The implementation of the lunar lander in this project tests three different epsilon values for the  $\epsilon$ -greedy method. The epsilon values tested are **1**, **0.1** and **0.01**, where a value of epsilon equating to **1** suggests a pure exploratory approach. Furthermore, a decay factor is applied to the epsilon value at each time step to gradually decrease the probability of exploration. This allows the agent to exploit the learnt policy more and choose actions that return a high reward. These methods help to manage the balance between exploration and exploitation while shifting towards more exploitation with each time step as the agent acquires more knowledge about the environment.

# References

- [1] Openai, “Gym/cartpole.py at master · Openai/Gym,” *GitHub*, 04-Oct-2022. [Online]. Available: [https://github.com/openai/gym/blob/master/gym/envs/classic\\_control/cartpole.py](https://github.com/openai/gym/blob/master/gym/envs/classic_control/cartpole.py). [Accessed: 18-Apr-2023].
- [2] S. Ravichandiran, *Deep Reinforcement Learning with Python*. 2<sup>nd</sup> ed. Packt Publishing, 2020.
- [3] C. Colas, O. Sigaud, and P. Oudeyer, “*GEP-PG: Decoupling Exploration and Exploitation in Deep Reinforcement Learning Algorithms*.” Available: <http://proceedings.mlr.press/v80/colas18a/colas18a.pdf>
- [4] P. Wei, “*Exploration-Exploitation Strategies in Deep Q-Networks Applied to Route-Finding Problems*.” *Journal of Physics: Conference Series*, 2020. doi: <https://doi.org/10.1088/1742-6596/1684/1/012073>
- [5] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.