MASTER OF TECHNOLOGY - INTELLIGENT SYSTEMS
2020

INTELLIGENT SOFTWARE AGENT

FINAL PROJECT2 REPORT

LunarLander-v2 Solver

Team members:

Zhan Sheng (A0215253N)
Wu Jingxuan (A0215262N)
Lakshmi Subramanian (A0215255L)
Yalavarti Dharma Teja (A0215457A)

Contents

1. Executive Summary 3
    1.1. Actions 3
    1.2. State 3
2. Project Objective 4
3. Background & Introduction 4
    3.1. Reinforcement learning 4
    3.2. Evolutionary Algorithm / Genetic Algorithm 5
    3.3. Q-learning 6
    3.4. Dueling DQN Network 7
    3.5. SARSA 8
4. Methodology 9
    4.1. Q-Learning 9
    4.2. SARSA 10
    4.3. Dueling Deep Q-Learning 11
    4.4. Evolutionary Algorithms 12
        4.4.1. Using Evolutionary Strategy 12
        4.4.2. Using genetic algorithm to train Neural Network 13
5. Results & Analysis 13
    5.1. Q-Learning 13
    5.2. SARSA 14
    5.3. Dueling DQN 15
    5.4. Evolutionary Algorithm 16
        5.4.1. Using Evolutionary Strategy 16
        5.4.2. Using genetic algorithm to train Neural Network 17
6. Limitations & Future Work 18
7. Conclusion 19
References 20

1. Executive Summary

Lunar Lander is an interesting problem in OpenAIGym. In our project, LunarLander-v2 was chosen as the prototype. [1] The figure below shows the LunarLander-v2 game interface.

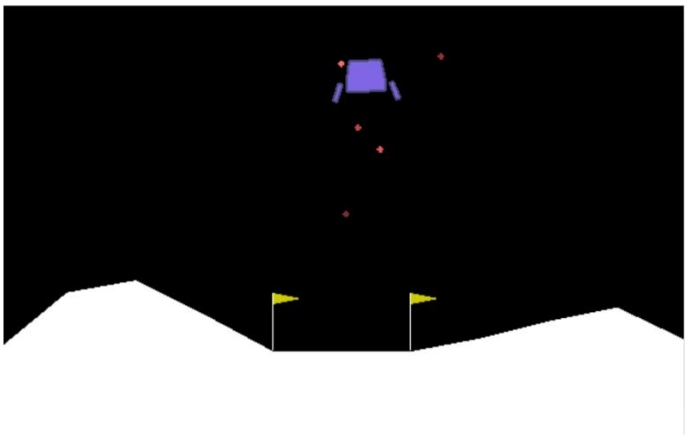


Figure 1: The screenshot of LunarLander-v2

Landing pad is always at coordinates (0,0). Coordinates are the first two numbers in the state vector. Reward for moving from the top of the screen to landing pad and zero speed is about 100 to 140 points. If the lander moves away from the landing pad it loses reward back. Episode finishes if the lander crashes or comes to rest, receiving additional -100 or +100 points. Each leg ground contact is +10. Firing main engine is -0.3 points each frame. Solved 200 points. Landing outside the landing pad is possible. Fuel is infinite, so an agent can learn to fly and then land on its first attempt.

1.1. Actions

There are four discrete actions the lander can take:

- 0: Do nothing
- 1: Fire left thruster
- 2: Fire main thruster
- 3: Fire right thruster

1.2. State

The state vector consists of eight variables between -1 and 1:

- Lander position in x
- Lander position in y
- Lander velocity in x
- Lander velocity in y
- Lander angle
- Lander angular velocity
- Contact left landing leg
- Contact right landing leg

By trying different algorithms to settle the LunarLander-v2 problem, our understanding of the RL and EL algorithms has undoubtedly improved.

2. Project Objective

To compare the performances of Lunarlander-v2 in OpenAIGym with the following four learning algorithms:

1. Q-Learning
2. Double DQN Network
3. SARSA
4. Evolutionary / Genetic Algorithm

3. Background & Introduction

3.1. Reinforcement learning

Reinforcement learning (RL) is learning by interacting with an environment. It is shown in figure 2. An RL agent learns from the consequences of its actions, rather than from being explicitly taught and it selects its actions on basis of its past experiences (exploitation) and also by new choices (exploration), which is essentially trial and error learning. The reinforcement signal that the Reinforcement Learning agent receives is a numerical reward, which encodes the success of an action's outcome, and the agent seeks to learn to select actions that maximize the accumulated reward over time.

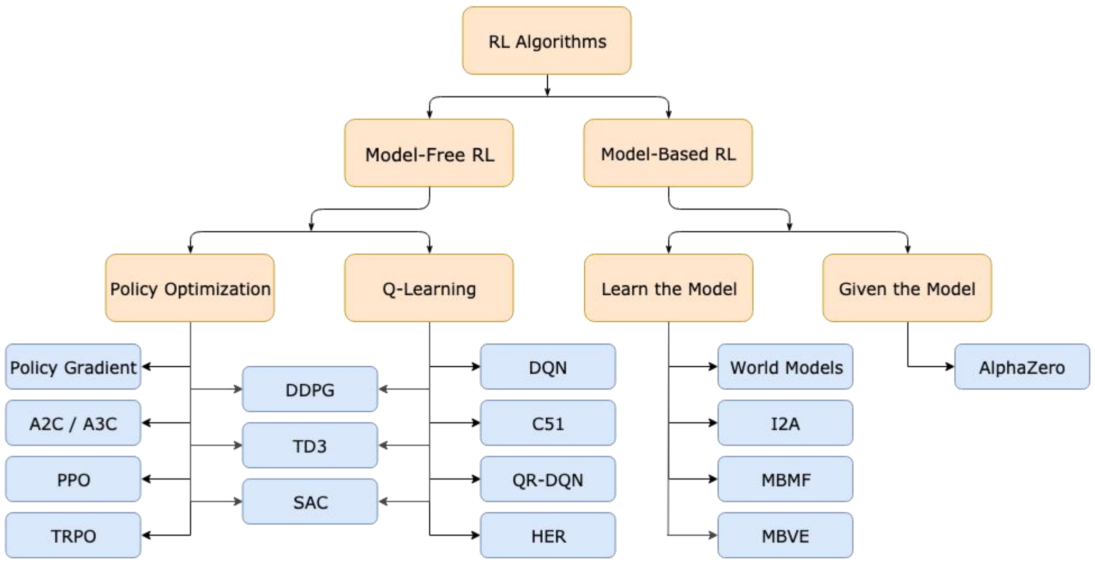
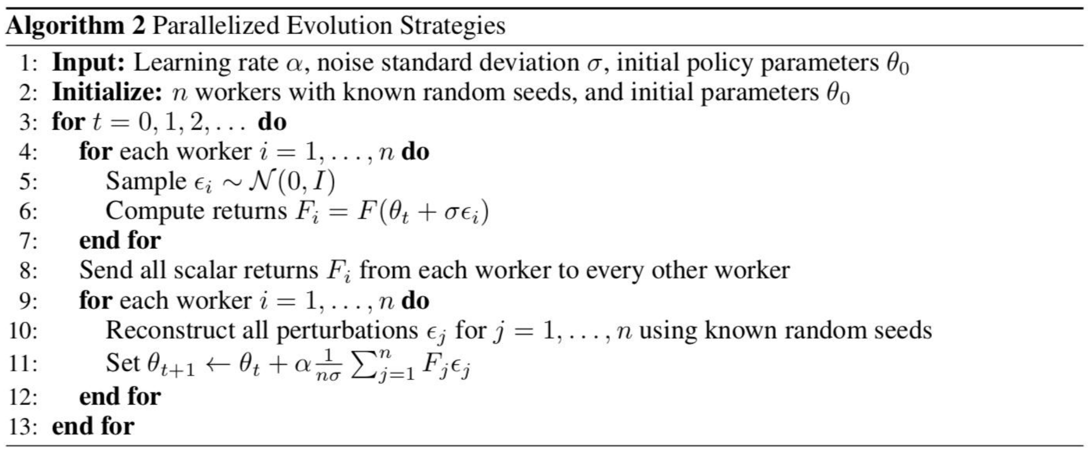


Figure 2: RL Algorithms classification

3.2. Evolutionary Algorithm / Genetic Algorithm

Genetic Algorithms (GA) starts with a population of randomly generated individuals/solutions, and uses the principle of natural selection to discover useful sets of solutions. The selection is usually fitness (usually, as per some pre-defined fitness function) proportional, with fitter individuals being allowed a higher probability of being selected into the subsequent generation. In order to search the solution-space, a proportion of individuals are subjected to mutation and crossover operations. The hope is that as the generations progress, fitter and fitter individuals get selected into the subsequent generations, which will ultimately deliver optimal or close to optimal solutions.

Evolutionary strategy is an optimization algorithm under the class evolutionary computation or artificial evolution methodologies. The search operators used in this evolutionary computation is mainly mutation and selection which is applied in each iteration and this is called generations. A simple Gaussian evolutionary strategy was chosen for the lunar lander problem.



Algorithm 2 Parallelized Evolution Strategies

```
1: Input: Learning rate $\alpha$, noise standard deviation $\sigma$, initial policy parameters $\theta_0$
2: Initialize: $n$ workers with known random seeds, and initial parameters $\theta_0$
3: for $t = 0, 1, 2, \dots$ do
4:     for each worker $i = 1, \dots, n$ do
5:         Sample $\epsilon_i \sim \mathcal{N}(0, I)$
6:         Compute returns $F_i = F(\theta_t + \sigma\epsilon_i)$
7:     end for
8:     Send all scalar returns $F_i$ from each worker to every other worker
9:     for each worker $i = 1, \dots, n$ do
10:        Reconstruct all perturbations $\epsilon_j$ for $j = 1, \dots, n$ using known random seeds
11:        Set $\theta_{t+1} \leftarrow \theta_t + \alpha \frac{1}{n\sigma} \sum_{j=1}^{n} F_j \epsilon_j$
12:    end for
13: end for
```

Figure 3: Pseudo code for Gaussian Evolutionary Strategy

Evolutionary algorithms can also be used to obtain the optimal weights of a neural network where the inputs are the states and the outlets are the actions. Instead of back propagation, evolutionary algorithms like genetic algorithms can be used to get the weights of the network.

3.3. Q-learning

Q-learning is a model-free reinforcement learning algorithm to learn quality of actions telling an agent what action to take under what circumstances. It does not require a model (hence the connotation "model-free") of the environment, and it can handle problems with stochastic transitions and rewards, without requiring adaptations.[2]

For any finite Markov decision process (FMDP), Q-learning finds an optimal policy in the sense of maximizing the expected value of the total reward over any and all successive steps, starting from the current state. Q-learning can identify an optimal action-selection policy for any given FMDP, given infinite exploration time and a partly-random policy. "Q" names the function that the algorithm computes with the maximum expected rewards for an action taken in a given state. [2]

```
$Q$-learning: Learn function $Q : \mathcal{X} \times \mathcal{A} \to \mathbb{R}$
Require:
  Sates $\mathcal{X} = \{1, \dots, n_x\}$
  Actions $\mathcal{A} = \{1, \dots, n_a\}$,      $A : \mathcal{X} \Rightarrow \mathcal{A}$
  Reward function $R : \mathcal{X} \times \mathcal{A} \to \mathbb{R}$
  Black-box (probabilistic) transition function $T : \mathcal{X} \times \mathcal{A} \to \mathcal{X}$
  Learning rate $\alpha \in [0, 1]$, typically $\alpha = 0.1$
  Discounting factor $\gamma \in [0, 1]$
  procedure QLEARNING($\mathcal{X}$, $A$, $R$, $T$, $\alpha$, $\gamma$)
      Initialize $Q : \mathcal{X} \times \mathcal{A} \to \mathbb{R}$ arbitrarily
      while $Q$ is not converged do
          Start in state $s \in \mathcal{X}$
          while $s$ is not terminal do
              Calculate $\pi$ according to Q and exploration strategy (e.g. $\pi(x) \leftarrow$
  $\arg\max_a Q(x, a)$)
              $a \leftarrow \pi(s)$
              $r \leftarrow R(s, a)$                                ▷ Receive the reward
              $s' \leftarrow T(s, a)$                               ▷ Receive the new state
              $Q(s', a) \leftarrow (1 - \alpha) \cdot Q(s, a) + \alpha \cdot (r + \gamma \cdot \max_{a'} Q(s', a'))$
              $s \leftarrow s'$
      return $Q$
```

Figure 3: Q-learning Pseudo Code [2]

3.4. Dueling DQN Network

This algorithm splits the Q-values in two different parts, the value function V(s) and the advantage function A(s, a).

The value function V(s) tells us how much reward we will collect from state s. And the advantage function A(s, a) tells us how much better one action is compared to the other actions. [3]

What the Dueling DQN algorithm proposes is that the same neural network splits its last layer in two parts, one of them to estimate the state value function for state s (V(s)) and the other one to estimate the advantage function for each action a (A(s, a)), and at the end it combines both parts into a single output, which will estimate the Q-values. This change is helpful, because sometimes it is unnecessary to know the exact value of each action, so just learning the state-value function can be enough in some cases. [3]

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

Figure 4: The formula of Dueling DQN Network [3]

3.5. SARSA

SARSA very much resembles Q-learning. The key difference between SARSA and Q-learning is that SARSA is an on-policy algorithm. It implies that SARSA learns the Q-value based on the action performed by the current policy instead of the greedy policy.

The action a_(t+1) is the action performed in the next state s_(t+1) under current policy.

```
SARSA($\lambda$): Learn function $Q : \mathcal{X} \times \mathcal{A} \rightarrow \mathbb{R}$
Require:
  Sates $\mathcal{X} = \{1, \dots, n_x\}$
  Actions $\mathcal{A} = \{1, \dots, n_a\}$,        $A : \mathcal{X} \Rightarrow \mathcal{A}$
  Reward function $R : \mathcal{X} \times \mathcal{A} \rightarrow \mathbb{R}$
  Black-box (probabilistic) transition function $T : \mathcal{X} \times \mathcal{A} \rightarrow \mathcal{X}$
  Learning rate $\alpha \in [0, 1]$, typically $\alpha = 0.1$
  Discounting factor $\gamma \in [0, 1]$
  $\lambda \in [0, 1]$: Trade-off between TD and MC
  procedure QLearning($\mathcal{X}$, $A$, $R$, $T$, $\alpha$, $\gamma$, $\lambda$)
    Initialize $Q : \mathcal{X} \times \mathcal{A} \rightarrow \mathbb{R}$ arbitrarily
    Initialize $e : \mathcal{X} \times \mathcal{A} \rightarrow \mathbb{R}$ with 0.                ▷ eligibility trace
    while $Q$ is not converged do
      Select $(s, a) \in \mathcal{X} \times \mathcal{A}$ arbitrarily
      while $s$ is not terminal do
        $r \leftarrow R(s, a)$
        $s' \leftarrow T(s, a)$                                ▷ Receive the new state
        Calculate $\pi$ based on $Q$ (e.g. epsilon-greedy)
        $a' \leftarrow \pi(s')$
        $e(s, a) \leftarrow e(s, a) + 1$
        $\delta \leftarrow r + \gamma \cdot Q(s', a') - Q(s, a)$
        for $(\tilde{s}, \tilde{a}) \in \mathcal{X} \times \mathcal{A}$ do
          $Q(\tilde{s}, \tilde{a}) \leftarrow Q(\tilde{s}, \tilde{a}) + \alpha \cdot \delta \cdot e(\tilde{s}, \tilde{a})$
          $e(\tilde{s}, \tilde{a}) \leftarrow \gamma \cdot \lambda \cdot e(\tilde{s}, \tilde{a})$
        $s \leftarrow s'$
        $a \leftarrow a'$
    return $Q$
```

Figure 5: SARSA Pseudo Code

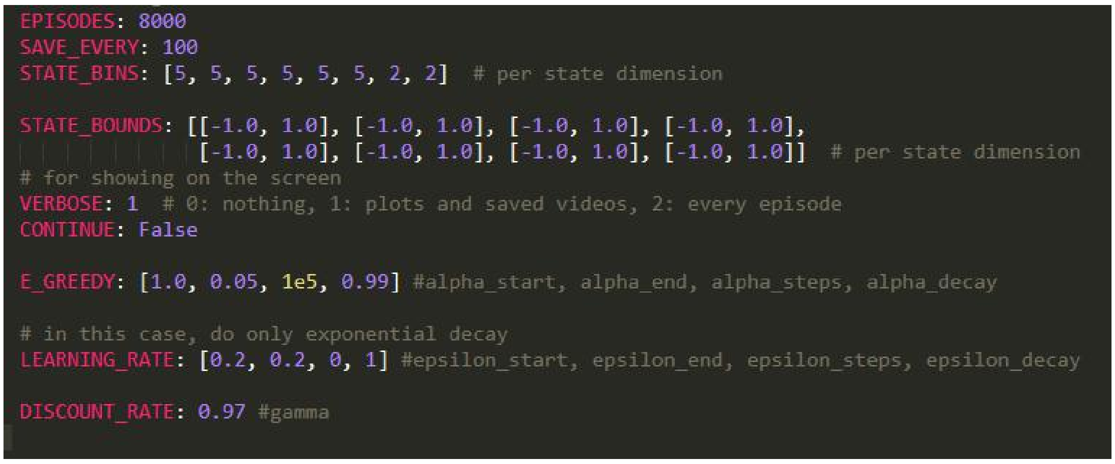
By contrast, Q-learning has no constraint over the next action, as long as it maximizes the Q-value for the next state. Therefore, SARSA is an on-policy algorithm.[5]

In this project, the 4 algorithms above were used to test the performance with the LunarLander-v2 game based on the OpenAIGym.

4. Methodology

The Lunar Lander game was implemented based on the openAIGym. In order to achieve smooth landing, four algorithms were used to test the performance of the Lunar Lander.

The parameters configuration for Q-Learning and SARSA algorithms was shown in figure 6 below.



```
EPISODES: 8000
SAVE_EVERY: 100
STATE_BINS: [5, 5, 5, 5, 5, 5, 2, 2]  # per state dimension

STATE_BOUNDS: [[-1.0, 1.0], [-1.0, 1.0], [-1.0, 1.0], [-1.0, 1.0],
               [-1.0, 1.0], [-1.0, 1.0], [-1.0, 1.0], [-1.0, 1.0]]  # per state dimension
# for showing on the screen
VERBOSE: 1  # 0: nothing, 1: plots and saved videos, 2: every episode
CONTINUE: False

E_GREEDY: [1.0, 0.05, 1e5, 0.99] #alpha_start, alpha_end, alpha_steps, alpha_decay

# in this case, do only exponential decay
LEARNING_RATE: [0.2, 0.2, 0, 1] #epsilon_start, epsilon_end, epsilon_steps, epsilon_decay

DISCOUNT_RATE: 0.97 #gamma
```

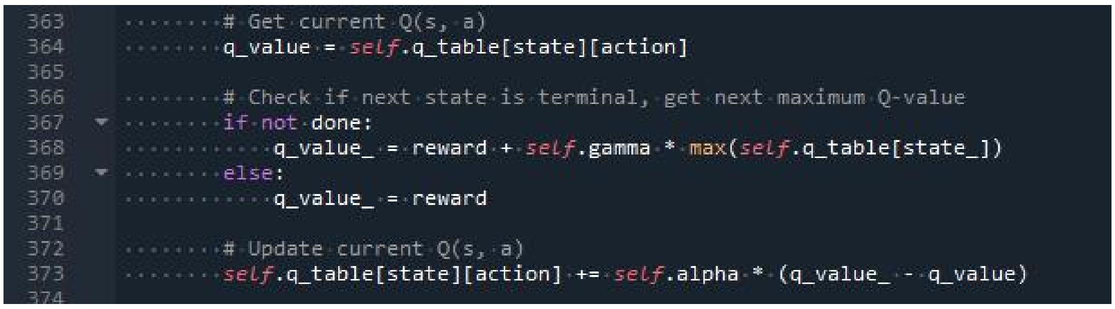
Figure 6: Configuration of Q-Learning and SARSA.

4.1. Q-Learning

Q-Learning is a value-based algorithm in reinforcement learning algorithms. Its algorithm was implemented as the pseudo code in figure 3. It firstly gets the current action value under the state in the Q-Table, and then selects the action that can get the most reward according to Q value.

The state that Lunar Lander comes into contact with at the beginning is very small, and if the Lunar Lander is executed according to the Q-Table that has been learned, then it is likely to make mistakes. At the same time, it is hoped that the Lunar Lander will land randomly at the beginning and get in touch with more states. Based on the above reasons, the Lunar Lander will not run exactly according to the results of Q learning at the beginning. It works with a certain probability epsilon, randomly selects actions instead of selecting actions based on max Q-Value. Then with continuous learning, this random probability is reduced and a decay function is used to reduce the epsilon.

In this case, the discount rate (gamma) was set as 0.97 and the epsilon decay was set as 0.99. After each iteration, the process updates current Q(s,a) in the Q-Table. The purpose of Q-Learning based on the Lunar Lander game is to find a strategy that can obtain the most reward. The python code was shown in figure 7 below.



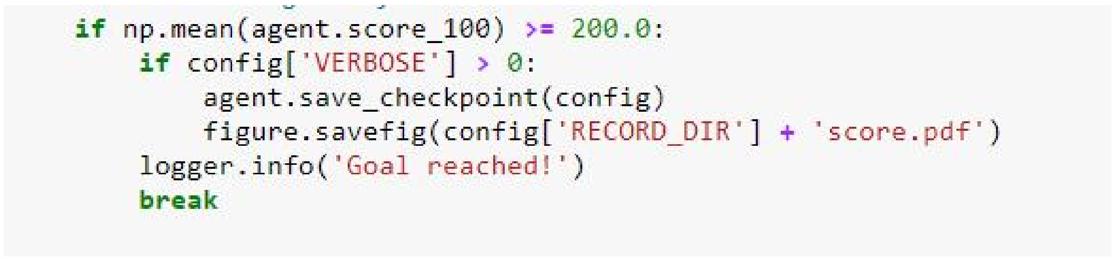
```
        # Get current Q(s, a)
        q_value = self.q_table[state][action]

        # Check if next state is terminal, get next maximum Q-value
        if not done:
            q_value_ = reward + self.gamma * max(self.q_table[state_])
        else:
            q_value_ = reward

        # Update current Q(s, a)
        self.q_table[state][action] += self.alpha * (q_value_ - q_value)
```

Figure 7: The q-values in Q-learning algorithm.

The processing shown in figure 8 will end either the agent's average reward score is larger than 200 or after 8000 episodes.



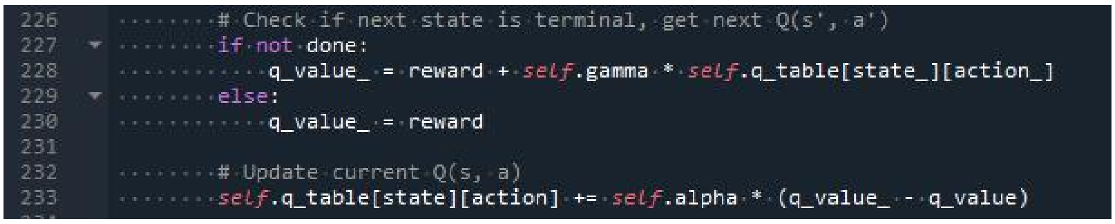
```
    if np.mean(agent.score_100) >= 200.0:
        if config['VERBOSE'] > 0:
            agent.save_checkpoint(config)
            figure.savefig(config['RECORD_DIR'] + 'score.pdf')
        logger.info('Goal reached!')
        break
```

Figure 8: The condition of finishing the processing.

4.2. SARSA

SARSA stands for State-Action-Reward-State-Action. SARSA algorithm is an On-Policy algorithm for Temporal Difference (TD) Learning. The main difference between SARSA and Q-Learning is that the maximum reward for the next state is not necessarily used to update the Q-values. Instead of that, a new action, and reward, is selected using the same policy that has determined the original action in the first place. The name SARSA actually comes from the functionality that the updates are being done using the quintuple Q(s, a, r, s', a'). Here, s and a are the original state and action, r is the reward received/observed in the following state and s', a' are the new state and action pair.

In this case, the discount rate (gamma) was set as 0.97 and the epsilon decay was set as 0.99, which are the same as the values for Q-Learning. The purpose of SARSA based on the Lunar Lander game is to find a strategy that can obtain the most reward. The python code was shown in figure 9 below.



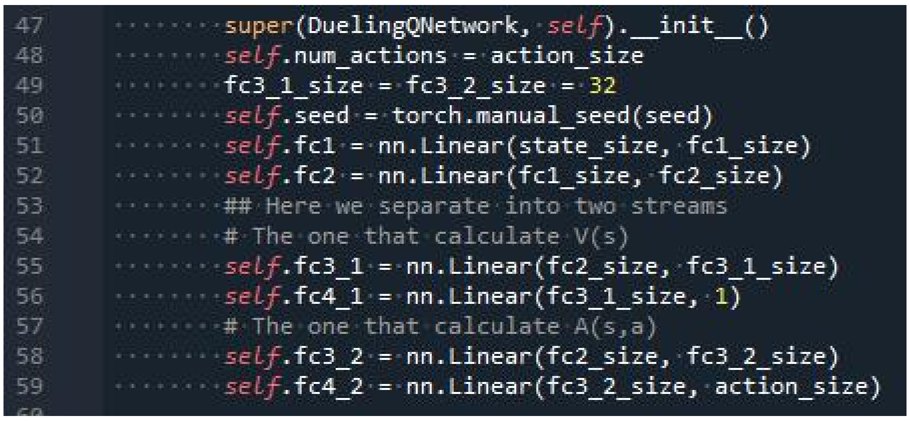
```python
        # Check if next state is terminal, get next Q(s', a')
        if not done:
            q_value_ = reward + self.gamma * self.q_table[state_][action_]
        else:
            q_value_ = reward

        # Update current Q(s, a)
        self.q_table[state][action] += self.alpha * (q_value_ - q_value)
```

Figure 9: The q-values in SARSA algorithms

The processing will end if it meets the same conditions as that with Q-Learning.

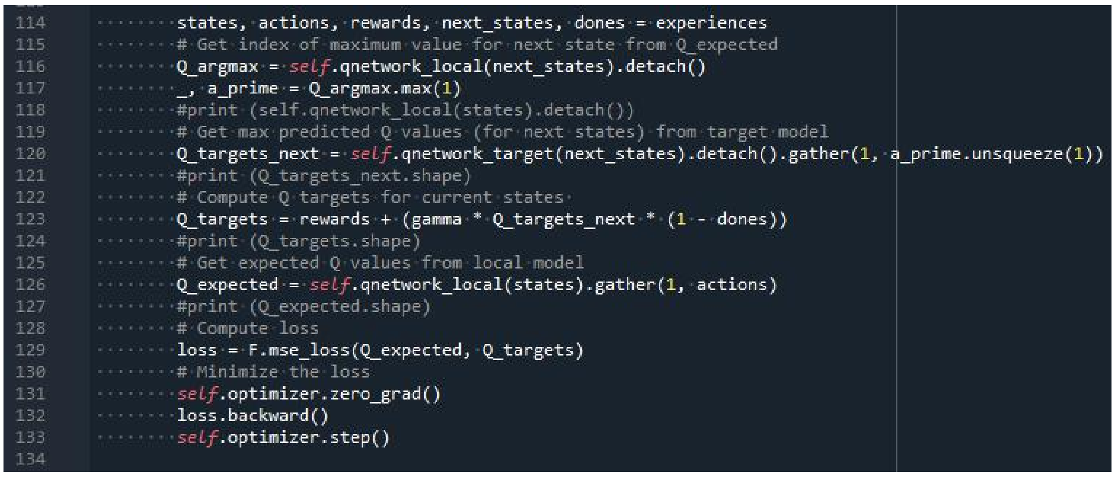
4.3. Dueling Deep Q-Learning



```python
        super(DuelingQNetwork, self).__init__()
        self.num_actions = action_size
        fc3_1_size = fc3_2_size = 32
        self.seed = torch.manual_seed(seed)
        self.fc1 = nn.Linear(state_size, fc1_size)
        self.fc2 = nn.Linear(fc1_size, fc2_size)
        ## Here we separate into two streams
        # The one that calculate V(s)
        self.fc3_1 = nn.Linear(fc2_size, fc3_1_size)
        self.fc4_1 = nn.Linear(fc3_1_size, 1)
        # The one that calculate A(s,a)
        self.fc3_2 = nn.Linear(fc2_size, fc3_2_size)
        self.fc4_2 = nn.Linear(fc3_2_size, action_size)
```

Figure 10: The network design in Dueling DQN algorithm.

Dueling DQN is also an easy to implement variant of DQN. The only difference between dueling DQN and DQN is the difference in network structure,which enables dueling DQN to learn more quickly.



```
        states, actions, rewards, next_states, dones = experiences
        # Get index of maximum value for next state from Q_expected
        Q_argmax = self.qnetwork_local(next_states).detach()
        _, a_prime = Q_argmax.max(1)
        #print (self.qnetwork_local(states).detach())
        # Get max predicted Q values (for next states) from target model
        Q_targets_next = self.qnetwork_target(next_states).detach().gather(1, a_prime.unsqueeze(1))
        #print (Q_targets_next.shape)
        # Compute Q targets for current states
        Q_targets = rewards + (gamma * Q_targets_next * (1 - dones))
        #print (Q_targets.shape)
        # Get expected Q values from local model
        Q_expected = self.qnetwork_local(states).gather(1, actions)
        #print (Q_expected.shape)
        # Compute loss
        loss = F.mse_loss(Q_expected, Q_targets)
        # Minimize the loss
        self.optimizer.zero_grad()
        loss.backward()
        self.optimizer.step()
```

Figure 11: The Q Values and Loss functions in Dueling DQN algorithm.

The Dueling DQN can be divided into three parts:

The first part: it is used to process and learn data like ordinary DQN.

The second part: calculate state value, which is the average value of network estimation.

The third part: calculate the value. Like state value, we input it from the H2 layer. Then we normalize the value, that is to increase the limit of "the average value of a value is 0".

The normalization process is very simple. We can get the average value of a value, and then subtract the average value from the value of A. A-mean(A)

The implementation of Dueling DQN is very simple, only needing to modify the network architecture of Q network. It can also be shared with other dqn techniques, such as experience playback, fixed network, dual network computing target, etc.

In this case, the discount rate (gamma) was also set as 0.97 and the epsilon decay was set as 0.99.

4.4. Evolutionary Algorithms

The Evolutionary Algorithms were implemented in the game based on the OpenAIGym. Two main methods are illustrated below.

4.4.1. Using Evolutionary Strategy

A simple Gaussian evolutionary strategy was chosen for the lunar lander problem. Here we have chosen the standard deviation of the noise that is added to the weight matrix as sigma = 100, learning rate alpha = 0.00025 for population size of 100 and the number of generations executed was for 100.

We also tried to alter the hyper parameter value setting to understand the effect of population size and learning rate. for the seconds simulation we used a relatively larger population size of 200 and learning rate alpha = 0.001.

4.4.2. Using genetic algorithm to train Neural Network

Here the network setup was structured such a way that inputs nodes of the network is defined to be 8 which are the states of the system and there are two hidden layers 10 nodes each along with biases in each layer and the output node is defined to be 4 which is the number of value sin the action space of the environment. Functions for generating networks for evolutionary mutation are used to generate fitter offspring networks. The mutation rate is set as 0.002 and the population size is 100. The setup was run for 500 generations.

5. Results & Analysis

The Lunar Lander game based on the OpenAIGym was implemented with four algorithms named SARSA, Q-Learning, Dueling DQN and Evolutionary Strategies. With the algorithms mentioned above, the Lunar Lander can keep balance and land smoothly as shown in figure 12 below. Each of the algorithms performs better than that without any algorithms.

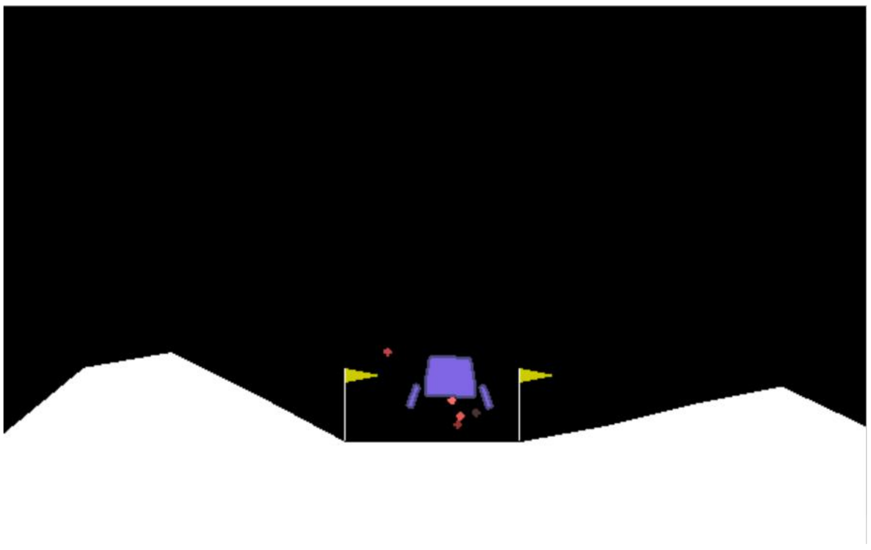


Figure 12: Landing with algorithms.

5.1. Q-Learning

After testing the Lunar Lander game with different parameters of gamma, alpha and epsilon, the best result in the tests of landing based on the algorithm of Q-Learning was shown in figure 13 below. The Lunar Lander can land smoother after 2000 episodes. In addition, the processing ended after 8000 episodes, which means the average rewards of each performance was not good enough based on the Q-Learning algorithm.

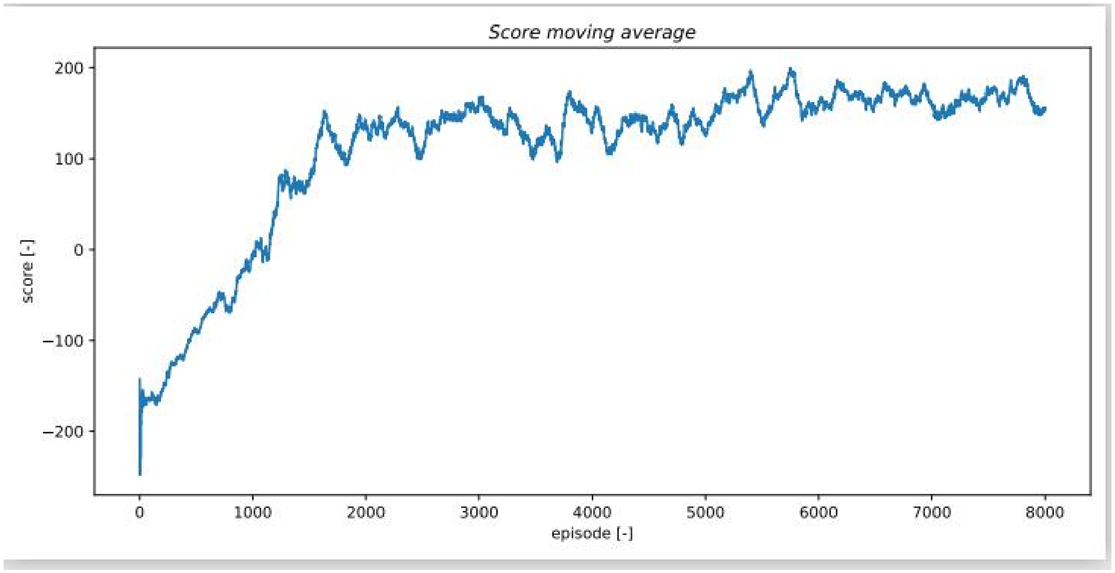


Figure 13: The moving average score plot based on q-learning algorithm

In the process of using different parameters to compare, epsilon value helped find a trade-off between exploration and execution when landing the Lunar Lander. Alpha helped weigh the results of the last learning and the results of current learning. If the alpha was set too low, the process only cared about the previous experience rather than accumulating new rewards. Gamma illustrated the impact of future rewards on the present. If the value of gamma is small, the process might not learn a strategy to reach the end. [2]

5.2. SARSA

After using different parameters of alpha, epsilon and gamma for testing the Lunar Lander game the below plot reflects the moving average score based on the episodes. The Lunar Lander was able to land well after 1300 episodes and much smoother after 4500 episodes as shown in figure 14. The execution comes to an end after 8000 episodes.

While using different parameters, the epsilon value helped in finding a trade-off between the exploration and exploitation when landing the Lunar Lander. Moreover, the parameters Alpha and Gamma have the same meaning and serve the same functionality as in Q-learning.

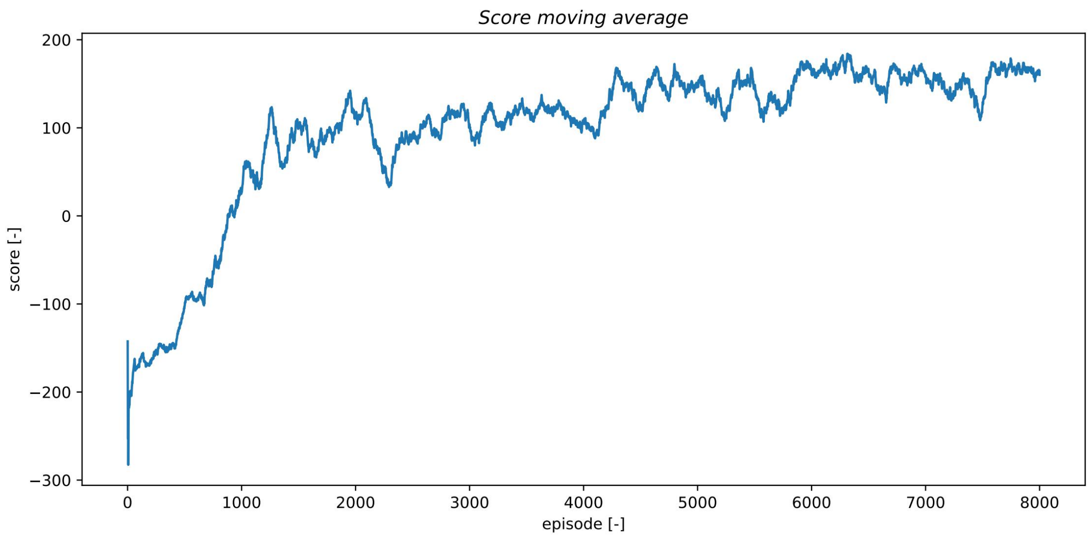


Figure 14: The moving average score plot based on SARSA algorithm

5.3. Dueling DQN

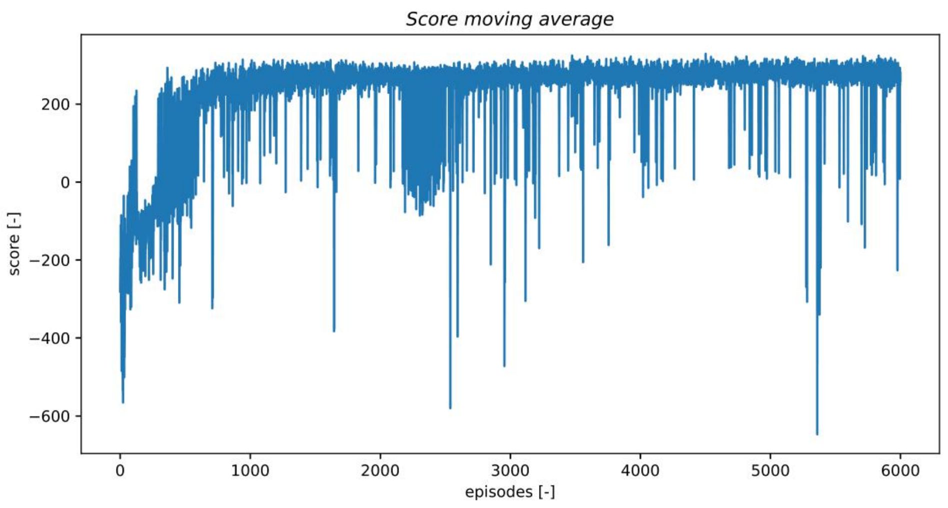


Figure 15: The moving average score plot based on Dueling DQN algorithm

The result of dueling DQN is shown in figure 15. Compared with the other three algorithms, dueling DQN algorithm has the following characteristics:

1.By using the Dueling DQN algorithm, the Lunar Lander score can be improved quickly.

2.However, due to the high complexity of the algorithm, it takes more time to run than other algorithms. It takes about 80min to run this 6000 times.

3.As can be seen from the burr on the graph, the result is not particularly stable and can produce many unexpected results.

4.Generally, Dueling DQN algorithm can achieve better results, and the score can reach more than 270 after 6000 episodes if it has a suitable parameter setting.

5.4. Evolutionary Algorithm

The use of different methods based on Evolutionary Algorithms generated different results.

5.4.1. Using Evolutionary Strategy

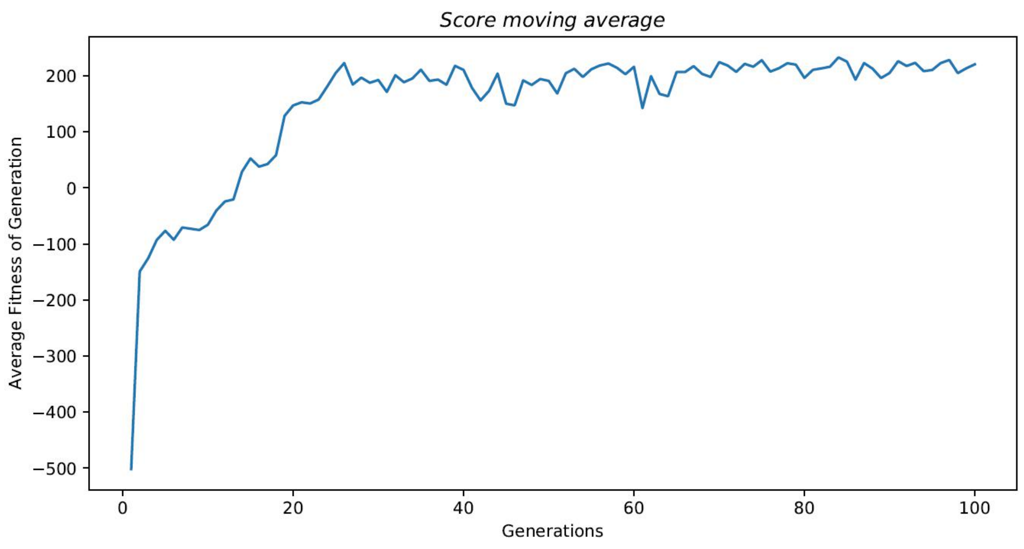


Figure 16: The moving average score plot based on evolutionary strategies case 1

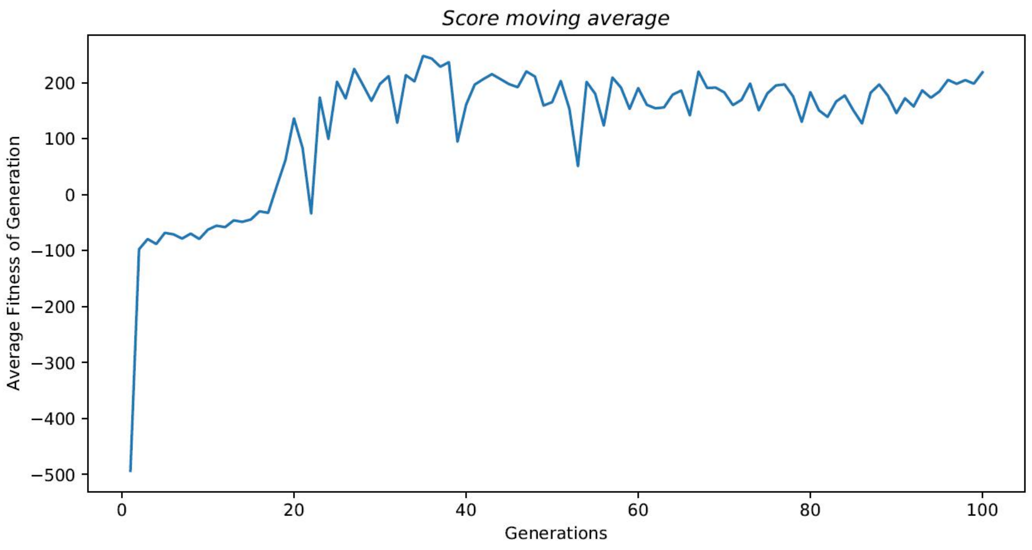


Figure 17: The moving average score plot based on evolutionary strategies case 2

A stable population mean is obtained with fitness reward of nearly 200 (which is good as mentioned in the gym documentation) in case of lower population size which can be inferred from the first graph.
In both the hyper parameter setup the peak is reached typically around 25 generations of training.

Here we observe from the above two graphs that doubling the population size and increasing the learning rate does not have a positive effect towards achieving higher and steady average population fitness. Since mutation is the only search operator we might be losing out the fitter actions steps over generations.

5.4.2. Using genetic algorithm to train Neural Network

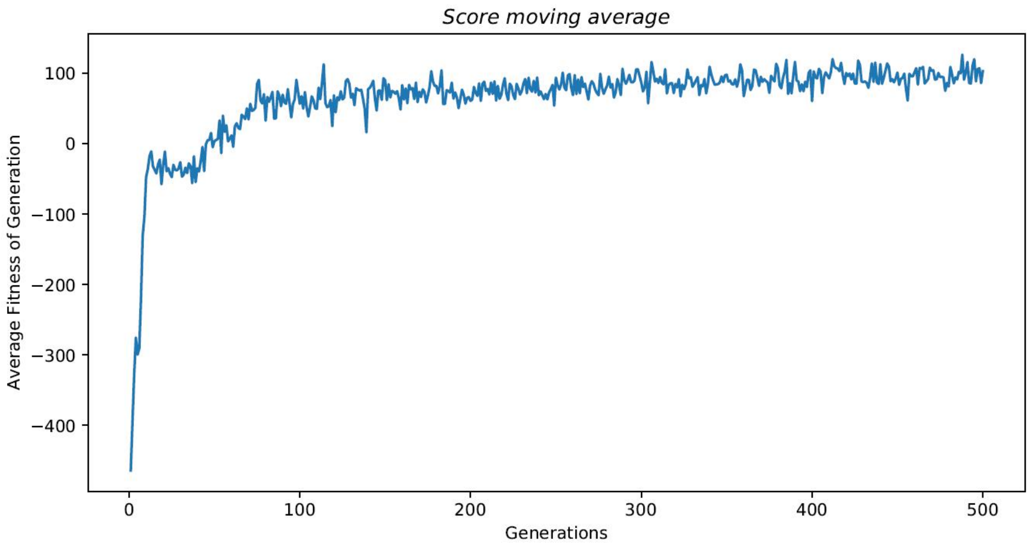


Figure 18: The moving average score plot based on network optimization using genetic algorithm

Compared to all the other algorithms this approach seems to have delivered a slightly poor performance. The mean population fitness has been achieved at relatively later generations after the start of training and the population mean is observed to be non improving for several generations.
Unlucky crossovers and Mutations could result in a negative effect on the program’s accuracy, and therefore make the program slower to converge or reach a certain loss threshold. [4]

6. Limitations & Future Work

Due to the shortage of time, some models in our project were not trained so completely to face all situations.

(1) There is no standard way to measure how good each algorithm is.
(2) Because the parameters of each algorithm are not the same, it is very difficult to ensure that all algorithms have the same parameters.
(3) The complexity of 4 algorithms is not consistent, and the efficiency of 4 algorithms is obviously different.

If we had a longer time frame to work on this project, we would have worked upon the following points of improvement:

1) Try more algorithms on Reinforcement Learning or Evolutionary Algorithms.
2) Try more combinations of parameter variations.
3) Try games with more complex environments in OpenAIGym.

7. Conclusion

Our group has spent a great time on this project. It is found that whatever to use the RL algorithms or EA algorithms, each algorithm has its own advantages and limitations. Generally speaking, most problems in a simple environment can be solved by using the SARSA algorithm or Q-learning algorithm. However, once the environment becomes a little more complex, the performances of these two algorithms are not good enough.

In this case, using the Dueling DQN algorithm or EA algorithm is more likely to achieve higher scores. However, it is difficult to get the best parameters setting, which requires rich machine learning experience to set parameters better.

In sum, there is no best RL or EA algorithm that can handle every problem. The only best way is to try a variety of machine learning algorithms, to try to change the combinations of a variety of variables, in order to find the algorithm that has better performance.

References

[1] Environments of LunarLander-v2 [online]
URL:https://gym.openai.com/envs/LunarLander-v2/

[2] Q-learning in Wikipedia [online]
URL:https://en.wikipedia.org/wiki/Q-learning

[3] Markel Sanz Ausin, Introduction to Reinforcement Learning. Part 4: Double DQN and Dueling DQN [online]
URL:https://medium.com/@markelsanz14/introduction-to-reinforcement-learning-part-4-double-dqn-and-dueling-dqn-b349c9a61ea1

[4] Neural Network and Genetic Algorithm [online]
URL:https://towardsdatascience.com/using-genetic-algorithms-to-train-neural-networks-b5ffe0d51321

[5] Reinforcement Learning
URL:https://www.cse.unsw.edu.au/~cs9417ml/RL1/algorithms.html