Using Reinforcement Learning to Optimise LunarLander-V2 Scores in an OpenAI Gym Environment

Aditya Gupta (adityag@connect.hku.hk), Aggarwal Dhruv (aggdhruv@connect.hku.hk),
Bae Joonyoung (n99joon@hku.hk), Kwan Rafael Matthew Susanto (u3574242@connect.hku.hk),
Suntoso Sean Michael (ssuntoso@connect.hku.hk)

1. Introduction

Reinforcement learning is a Machine Learning technique differing from Supervised and Unsupervised learning where instead of a dataset, the algorithm is given positive rewards for desirable actions and negative penalties for undesirable actions. More abstractly, the entity 'making the decisions' is the agent, and everything it interacts with is its environment. Interactions between the agent and its environment at each time step t of its state provide it with some reward $R \in \mathrm{R}$ which needs to be maximized [1].

In this project, we will engineer multiple Deep Learning Neural Networks to achieve a high score on the "LunarLander-v2" game under the OpenAI Gym environment. This will be achieved via reinforcement learning, with the agent receiving a positive reward for landing in between the yellow flags, and negative rewards for crashing or landing outside the goal zone.

2. Problem Statement

We consider only the discrete action space where there are 4 possible actions: (1) do nothing (2) fire left orientation engine (3) fire main engine (4) fire right orientation engine [2]. An 8-dimensional vector represents the state at t with the coordinates x & y, its linear velocities in x & y, angle, angular velocity, and booleans that represent whether each leg has contacted the ground [2]. We aim to maximize the score of the landing of the lunar module through these methods.

To achieve this we will consider three classes of RL Deep Neural Networks. Namely, the Policy Gradient methods, DQN methods, and SARSA methods. With fine tuning and hyperparameter optimisation we aim to achieve as high of a score as possible with a reasonable training time.

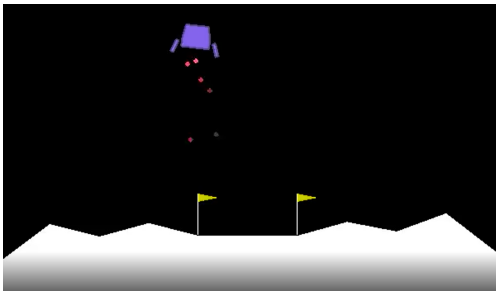

Figure 1: A snapshot of the game LunarLander-V2

3. Related Work

OpenAI's Lunar Lander v2 problem is a well tackled one. While our project focuses on the discrete action space, there exist solutions to solve the problem using the continuous action space. Verma's implementation of the DDPG algorithm using the actor-critic network architecture [3]. We also found multiple solutions using the relatively new Proximal Policy Optimization algorithm to solve the lunar lander problem on both the discrete and continuous action space [4]. However, these solutions require complex implementations of network architecture while obtaining very similar results to DQN and SARSA.

4. Policy Gradient

A Policy Gradient method attempts to maximize the "Expected" reward of a policy π, where the exact reward mechanism depends on the variation used. The reward is maximized via gradient ascent, and the neural network parameterized the policy function. In this section we consider 3 implementations of the Policy Gradient Algorithm: REINFORCE, Proximal Policy Optimization, and Actor Critic (A2C).

4.1. Reinforce

The first variant we use is the REINFORCE method implemented in the notebook provided in the tutorial. The pseudo-code is given as follows [5].

function REINFORCE
  Initialise θ arbitrarily
  for each episode $\{s_1, a_1, r_2, ..., s_{T-1}, a_{T-1}, r_T\} \sim \pi_\theta$ **do**
    for $t = 1$ to $T - 1$ **do**
      $\theta \leftarrow \theta + \alpha \nabla_\theta \log \pi_\theta(s_t, a_t) v_t$
    end for
  end for
  return θ
end function

Figure 2:: Policy gradient with REINFORCE pseudo-code

With 1000 updates, the network is able to achieve final rewards of around 80, which is a reasonable score. The architecture uses 2 fully connected hidden layers with 16 features, and output space of 4.

Figure 2: Policy Gradient Scores by Episode

To improve the score, RELU and Sigmoid activation functions were tested as alternatives, but consistently achieved worse results. The network was then tested with 96 features and the performance improved markedly. But the high variance drawback of Monte Carlo sampling is evident with sudden drops in the score towards the end of the training. More hyperparameter optimization in the future can yield better results.

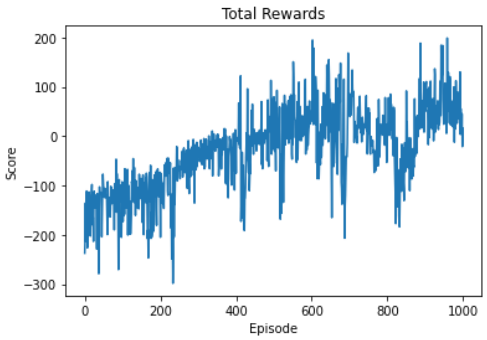

Figure 3: Optimised Policy Gradient Scores by Episode

Further, due to the nature of random sampling, REINFORCE is subject to high variance even with a large number of episodes. We can see that gradient ascent converges slowly due to opposing directions while training gradient ascent.

4.2. Proximal Policy Optimisation (PPO) Method

As stated above, REINFORCE is subject to high variance and slow convergence. In order to avoid this, the PPO algorithm was considered instead. PPO uses an off-policy approach by limiting the reward term to avoid sudden large steps which affect the convergence. The objective is defined as [6]:

$$\hat{\mathbb{E}}_t\Big[\min(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)\Big]$$

where $$r_t(\theta) = \frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{\text{old}}}(a_t \mid s_t)}$$

is the probability ratio. A current implementation of PPO was modified with some hyperparameter tuning and used on the LunarLander-V2 module.

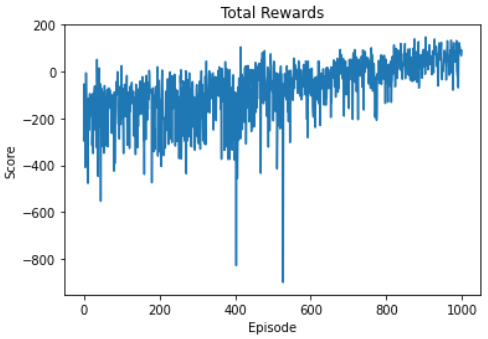

Figure 4: PPO Scores by Episode

After training on 1000 episodes, the results were quite disappointing, with a max score of around 100 reached in the last 50 episodes. Convergence of the PPO algorithm was steady, but slow and can lead to a good score but after too many episodes of training. To get a better result, another algorithm was considered.

4.3. Advantage Actor Critic (A2C)

The psuedocode for the A2C model is as follows [7]:

1. sample $\{\mathbf{s}_i, \mathbf{a}_i\}$ from $\pi_\theta(\mathbf{a}|\mathbf{s})$ (run it on the robot)
2. fit $\hat{V}^\pi_\phi(\mathbf{s})$ to sampled reward sums
3. evaluate $\hat{A}^\pi(\mathbf{s}_i, \mathbf{a}_i) = r(\mathbf{s}_i, \mathbf{a}_i) + \gamma \hat{V}^\pi_\phi(\mathbf{s}'_i) - \hat{V}^\pi_\phi(\mathbf{s}_i)$
4. $\nabla_\theta J(\theta) \approx \sum_i \nabla_\theta \log \pi_\theta(\mathbf{a}_i|\mathbf{s}_i)\hat{A}^\pi(\mathbf{s}_i, \mathbf{a}_i)$
5. $\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$

Here we use a "critic" to receive an approximation for the value function, and then use the Policy Gradient algorithm to move in the direction suggested by the critic. We use these approximations of the Q-value of different states to avoid the high variance from Monte Carlo sampling and move steadily towards the minima.

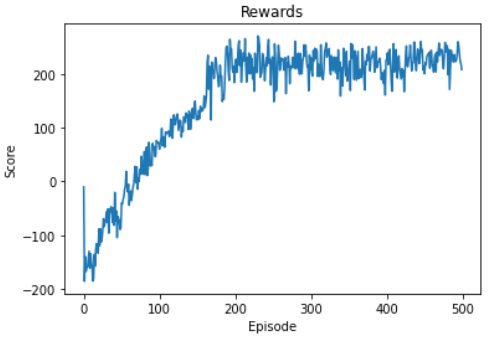

Figure 5: A2C Scores by Episode

The results are much more promising, and in only 500 episodes a steady score of >200 is seen. Training beyond 200 does not seem to increase the score much, but convergence is quick and variance is low.

5. Deep Q-Network (DQN)

Next, we deploy the Deep Q-Network (DQN), a DNN used to estimate Q-Values using an off-policy algorithm, i.e. the greedy approach, and then compare its outcomes with an improved DNN - Double DQN. Furthermore, since DQN is an active field of research, there are a few variations that stabilize and speed up its training. Some of the relatively cutting edge improvements of DQN methods, such as Prioritized Experience Replay [8] and Dueling DQN [9] introduced in 2015, are also utilized to effectively deal with potential downfalls of a traditional DQN and maximize the score of the agent in the future.

5.1. Experience Replay

DQN uses previous estimation results for the consequent estimations. However, using only the latest experiences in sequence highly increases the correlations between the experiences in a training batch, which harms training efficiencies. Each experience is stored as a transition tuple of the form (s_t, a_t, r_t, s_{t+1}, d) with current state, action, rewards, successor state and completion status, at each time step t. Thus, storing all experiences in a *replay buffer* (or *replay memory*) and uniformly sampling a random training batch from it at each training iteration boosts the overall performance and is called *Experience Replay* [10].

5.2. Epsilon-Greedy Policy and Epsilon Decay

While updating the Q-values, we have to make a trade-off between *exploration*, exploring random action, and *exploitation*, greedily selecting the action with the highest estimated Q-value. In a specific term, it is called the *ε-greedy policy*, where at each step the model acts randomly with probability of ε, or greedily with probability of 1-ε.

In the early stage of training, it is more beneficial to explore randomly as it can reach out to many interesting parts of the environment and prevents from getting stuck at local minima. At the later stage, when Q-values are near optimum, lower ε values are preferred. Thus, *Epsilon Decay* is deployed, where ε value is decreased at decay rate after every step until preset ε base value [11].

5.3. DQN

Deep Q Learning networks can be used to effectively learn the expected reuters from state action value pairs in real-world scenarios with a large number of states [12]. Our implemented network uses a standard neural network - Q network to produce the optimal state action values with the experience replay interacting with the environment to produce more training data. For an input state, the DQN uses the following equation to generate Q values for all possible actions while exploring using the *ε-greedy* approach [12].

Bellman Equation for DQN:

$$Q(s,a;\theta) = r + \gamma Q(s', argmax_{a'} Q(s', a'; \theta); \theta) \quad (1)$$

With the parameter settings as, number of episodes = 250, ε = 1.0, ε decay rate = 0.99, ε end base value = 0.01, our DQN model was able to achieve a score of ~200 while training over a period of 250 episodes, clearly outperforming results seen while using Policy Gradient.

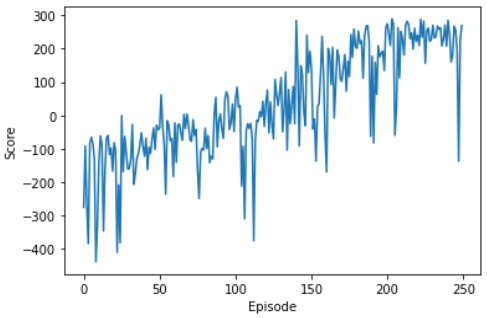

Figure 6: DQN Scores by Episodes

Further hyperparameter tuning or training over a longer episode period may yield better results, and will be implemented.

5.4. Double DQN

One of the improvements that could be made to the original DQN model was the tendency of DQN to overestimate certain Q-values. For instance, when there actions that are equally good, since Q-values are updated with approximations, some values get measured slightly greater than others, which lead to more subsequent emphasis on that value in greedy selection procedures.

In 2015, Double DQN was introduced by DeepMind researchers, which resolves this issue by using the online model, rather than the target model, during the selection process of the best actions for the next states, and using the target model only during the estimation calculations of the Q-values for the selected best actions [13].

Bellman Equation for Double DQN:

$$Q(s,a;\theta) = r + \gamma Q(s', argmax_{a'} Q(s', a'; \theta); \theta') \quad (2)$$

The only difference of equation (1) and (2) is that the main online neural network's weight parameters θ while selection is distinguished from the target network's parameters θ' while value evaluation.

Figure 7 is the run of the program with the same parameters as above DQN, but with the Double DQN model and over 400 training episodes.

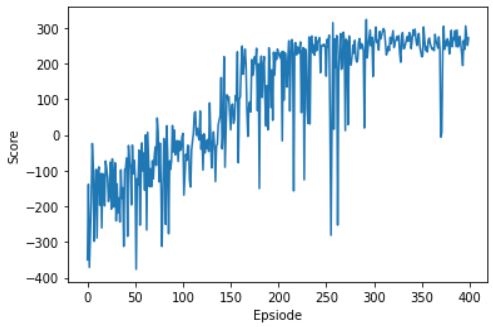

Figure 7: Double DQN Scores by Episodes

5.5. Double DQN with Prioritised Experience Replay

Another improvement that could be made to the DQN model was to choose more important experiences for the network model to learn from instead of uniform sampling.

Using Prioritised Experience Replay (PER), each experience is allotted a priority proportional to the loss obtained. Therefore, experiences which led to more rewards and allowed the network to learn a lot are given higher priority. The replay buffer then samples these experiences more often than others, allowing the model to further improve its accuracy or learn to solve the environment faster than a DQN model with experience replay [14].

We implemented PER with our Double DQN Model. However, over a training period of 400 episodes, the model gave slightly poorer results than the general Double DQN. This may be due to the fact that PER favors transitions with non-zero rewards and is more suitable for environments with sparse rewards, i.e., most transitions having reward = 0.

Below is the run of the Double DQN with Prioritised Experience Replay with the same parameters as Double DQN.

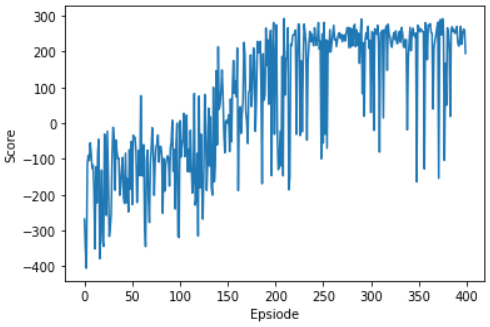

Figure 8: Double DQN with PER Scores by Episodes

5.6. Dueling DQN

In some environments, actions may not always impact meaningfully, and it is not important to learn how actions impact states at each timestep. Dueling DQNs learn more important features in such environments like which states are desired and undesired.

The Bellman equation for Dueling DQN:

$$Q(s,a;\theta,\alpha,\beta) = V(s;\theta,\beta) + A(s,a;\theta,\alpha) - 1|A\,\Sigma A(s,a;\theta,\alpha)$$

The Q value is the sum of $V(s;\theta,\beta)$ representing the value of a state independent of the action taken and $A(s,a;\theta,\alpha)$ representing the advantage of taking the particular action at that state relative to others. To ensure we can identify the contributing factor V and A uniquely we subtract the mean of actions A. By maximizing Q value for action to equal V, we ensure proper model performance [15].

Below is the run of the Dueling DQN with the same parameters as Double DQN.

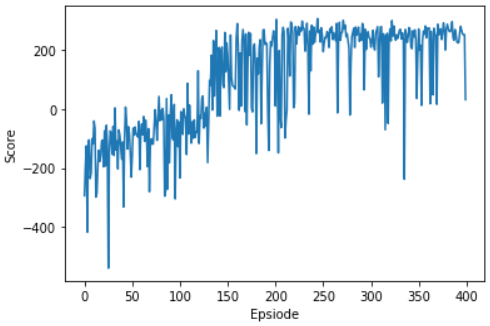

Figure 9: Dueling DQN Scores by Episodes

5.7. Result Comparison

On comparing the results of the above-mentioned DQN models, the Double DQN model dominates in training accuracy achieving an average score of 254.83 in episode 400. However, it is worth noting that each of the models achieved an average score above 200 at the end of their training periods, thus yielding very high results and performance on the environment. The below figure graphs the average scores/performance of each of the DQN models over their respective training episodes.

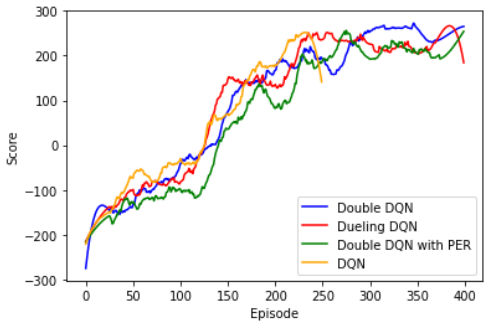

Figure 10: Comparison of DQN models' training results

6. Deep SARSA

Deep SARSA is the combination of SARSA on-policy reinforcement learning algorithm with deep

learning. Its objective is to estimate state action values and to get optimal policy for the agent. The SARSA algorithm is implemented according to this equation:

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma Q(s', a') - Q(s,a)]$$

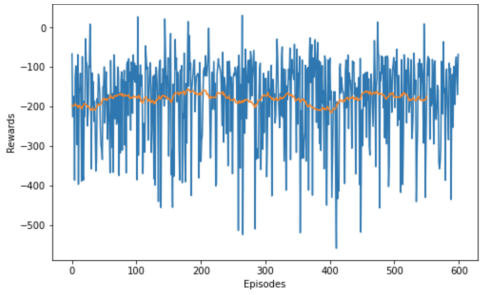

Figure 11: Rewards vs Episodes - Deep SARSA with random action

In Deep SARSA, the agents will control the state action pair values for the Q network based on experience. Initially, we used SARSA and let the agent choose random moves. The result is shown in Figure 11. It is clear that further optimization is needed since the agent still cannot land properly and the final score is below 0.

6.1. Model Optimization

To further optimize and strengthen the model, we implement ε-greedy policy that will help the agent choose the best action for the next step. With an increasing number of episodes, the agent starts to learn the environment and become better in choosing the best action [16]. We also set the discount factor to 1 to let the model focus on the rewards.

As the agent exploring the environment, we program ε to decay by 0.995 of its previous value and finally settle at 0.01 (the optimum number after multiple trials) to limit the exploration and make the reward more predictable. Thus, in the end, the agent will know how to choose the best action. The Deep SARSA results match those of DQN but over 1000 training episodes.

6.2. Result

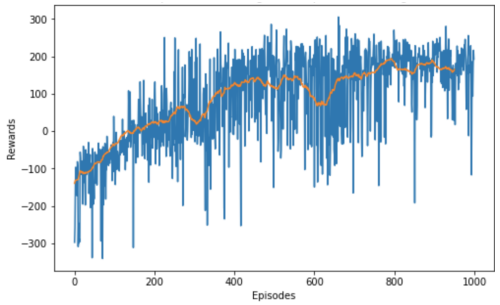

Figure 12: Rewards vs Episodes - Deep SARSA with optimization

Figure 12 shows us that the model has reached its peak. We then tested the model for 100 trial runs which resulted in an average reward of around 202 with 99% of it reaching the score of above 100 as shown in Figure 9. Since it is above the benchmark of 200 [2], thus we can consider that the model has solved the Lunar Lander.

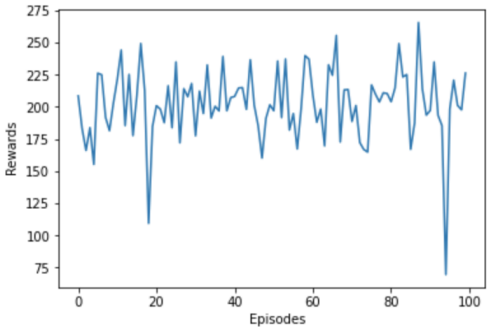

Figure 13: Rewards vs Episodes - Deep SARSA model test

7. Conclusion

After comparing all the models, the Double DQN with Epsilon Decay and Prioritized Experience Replay had the best score. All three classes of Deep Networks were able to get a score >200 (considered completed), but the Double DQN did it in the fewest episodes and with the lowest variance towards the final episodes.

References

[1] Richard Sutton and Andrew Barto. Reinforcement learning: An introduction. *MIT press*, 2018.

[2] Oleg Klimov. Lunar Lander - Gym Documentation. Available: https://www.gymlibrary.dev/environments/box2d/lunar_lander/?highlight=lunar.

[3] Shiva Verma. Train Your Lunar-Lander | Reinforcement Learning. *Medium*, 2021. Available: https://shiva-verma.medium.com/solving-lunar-lander-openaigym-reinforcement-learning-785675066197

[4] Youness Mansar. Learning to Play CartPole and LunarLander with Proximal Policy Optimization. *Medium*, 2020. Available: https://towardsdatascience.com/learning-to-play-cartpole-and-lunarlander-with-proximal-policy-optimization-dacbd6045417

[5] Tingwu Wang. Learning Reinforcement Learning by Learning REINFORCE. University of Toronto, 2020.

[6] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[7] Sergey Levine. Actor-critic algorithms. 2022. Available: https://rail.eecs.berkeley.edu/deeprlcourse-fa17/f17docs/lecture_5_actor_critic_pdf.pdf.

[8] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*, 2015.

[9] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Hasselt, Marc Lanctot, and Nando Freitas. Dueling network architectures for deep reinforcement learning. In *International conference on machine learning*, pages 1995-2003, 2016.

[10] Shangtong Zhang, and Richard Sutton. A deeper look at experience replay. *arXiv preprint arXiv:1712.01275,* 2017.

[11] Alexandre dos Santos Mignon and Ricardo Luis de Azevedo da Rocha. An adaptive implementation of ε-greedy in reinforcement learning. In *Procedia Computer Science*, pages 1146-1151, 2017

[12] Ketan Doshi. Reinforcement Learning Explained Visually (Part 5): Deep Q Networks, step-by-step. In *Medium*, 2021. Available: https://towardsdatascience.com/reinforcement-learning-explained-visually-part-5-deep-q-networks-step-by-step-5a5317197f4b

[13] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Proceedings of the AAAI conference on artificial intelligence, vol. 30, no. 1*, 2016.

[14] Guillaume Crabé. How to implement prioritized experience replay for a deep Q-Network. In *Medium*, 2020. Available: https://towardsdatascience.com/how-to-implement-prioritized-experience-replay-for-a-deep-q-network-a710beecd77b.

[15] Chris Yoon. Dueling Deep Q Networks. In *Medium*, 2019. Available: https://towardsdatascience.com/dueling-deep-q-networks-81ffab672751. [Accessed: 22-Nov-2022].

[16] Dongbin Zhao, Haitao Wang, Kun Shao, and Yuanheng Zhu. Deep reinforcement learning with experience replay based on SARSA. In *2016 IEEE symposium series on computational intelligence (SSCI)*, pages 1-6, 2016.

[17] JohDonald. Applying deep reinforcement learning algorithms, Deep Sarsa and deep Q-learning, to Openai Gym's Lunarlander-V2. In *GitHub*, 2020. Available: https://github.com/JohDonald/Deep-Q-Learning-Deep-SARSA-LunarLander-v2.