Reinforcement Learning to control an agent in LunarLander-v2 - an OpenAI gym environment

Akhras Emad (u3566296@hku.hk), Jain Uday (u3555276@hku.hk),
Lohia Suyash (suyash@hku.hk), Ma Chun Lai Jonathan (clma@hku.hk)

COMP3340 Group 7

1. Introduction

Inspired by the learning behaviours in biological species, reinforcement learning is considered a fundamental paradigm in machine learning, alongside supervised learning and unsupervised learning, that aims to maximize the notion of cumulative reward in a task. From a high level, a reinforcement learning problem contains an actor, an environment and a reward function, with the goal being that the agent should learn to take the actions that maximize the expected reward in the given environment, by repeatedly interacting with the environment [3]. In this project, the topic concerned is to learn the optimal steps in controlling the lander in a way that maximizes the reward. The reward system of LunarLander-v2 has been defined in the problem definition section below.

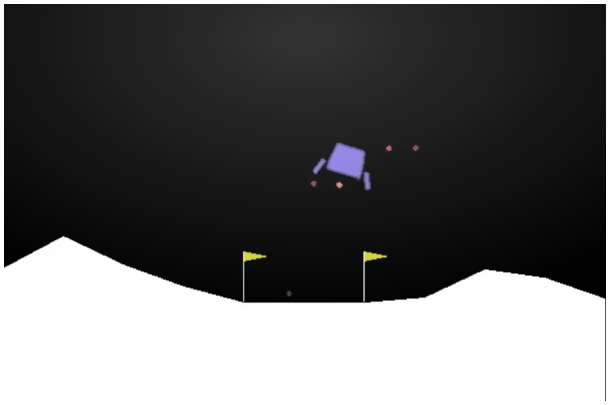
2. Problem Definition

In this project we aim to apply reinforcement learning to train an agent in an OpenAI Gym environment. We adopt "LunarLander-v2" as the simulator environment to run the algorithm implementation, train the agent, and test its performance. The agent will be trained to make a safe landing on the moon surface in between two yellow flags with some given constraints. In the context of this paper, the term environment may capture the agent, the yellow flags, the landing surface, etc [10]. The state space contains 8 state variables where each variable describes a specific attribute about the agent.

- (x, y) : coordinate of the agent in the space
- (v_x, v_y) : velocity vector of the agent in the space
- θ : orientation of the agent in the space
- v_θ : angular velocity of the agent
- $leftLeg$: is agent's left leg on the ground
- $rightLeg$: is agent's right leg on the ground

It follows from the state space and attributes that the agent can choose from four discrete actions at a time: activate left engine, activate right engine, activate main engine, and do nothing.

Figure 1. Visualisation of the environment and the agent [11]



3. Related Work

We have identified a number of unique strategies in existing literature aimed at solving the lunar lander environment. Some works attempt to solve the environment using a particular algorithm. Lu, Chai, and Squillante implemented an unconventional model-based technique where the algorithm avoids learning the system's parameters in favour of learning the optimal control parameters [5]. Banerjee, Ghosh, and Das improved upon a simple policy gradient by evolving neural network topologies through incorporating Evolutionary Algorithms (JADE) and Metaheuristic Algorithms (PSO) [1]. Peters, Stewart, West, and Esfandiari' algorithm relied on spiking neural networks to dynamically select actions [12]. However, other works attempt to evaluate some of the reinforcement learning algorithms used in solving the lunar lander environment in the presence of environment and agent uncertainties [15].

4. DQN

In this subsection, we present the DQN model. The Deep Q-Network is used to estimate the Q-values where the input to the network is the a vector representing the 8 state variables, i.e. current state, and the outputs are the Q-values for all state-action permutations of that state. Furthermore, the rule for updating Q-values:

$$Q(s,a) = Q(s,a) + \alpha[r - Q(s,a) + \gamma \max_{a'} Q(s^{'}, a^{'})]$$

The full algorithm for training the agent in a deep Q-network is shown in Algorithm 1 [8]. Defining an appropriate reward model is crucial to the success of training the agent. The Gym Lunar-Lander environment provides access to *step* function that returns *reward* (*float*) among other variables. With the goal being to maximise total reward, the reward model is defined below [13]:

- Reward for moving from the top of the screen to the landing pad and coming to rest is about 100-140 points.
- If the lander crashes, it receives an additional -100 points.
- If it comes to rest, it receives an additional +100 points.
- Each leg with ground contact is +10 points.
- Firing the main engine is -0.3 points each frame.
- Firing the side engine is -0.03 points each frame.
- Solved is 200 points.

In addition to a basic DQN algorithm, there are additional features that our model implements. These are detailed in the following subsections. To conclude this section, we also detail some findings upon running the model on the given problem.

4.1. Experience Relay

Experience Replay [14] is used as a method to reduce correlation between successive data points. In this method, the experiences of an agent are saved at every time step. From this pool of experiences, a sample is chosen and the model is trained on the chosen sample. Thus, by reducing the variances of the updates, the model is able to achieve better convergence behaviour.

4.2. Exploration v/s Exploitation

When training a reinforcement learning model, it is possible for the agent to favor exploiting the promising areas found where rewards have been high [7]. Doing this can possibly ignore a non explored local minimum that could yield better and more consistent rewards. In order to balance this trade-off, we use a hyper-parameter to define a small probability of choosing a random action as opposed to following the model's previous prediction. This process is called Exploration [7].

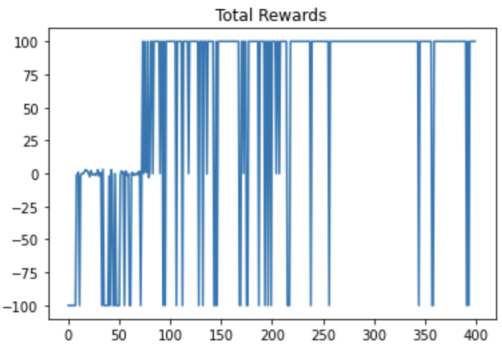
Algorithm 1 Deep Q-Learning with experience replay

initialise Q-network parameters
initialise replay memory **D**
for $episode = 1 \rightarrow$ **M do**
  initialise state $\phi(\mathrm{s}_1)$
  finished $\leftarrow$ *false*. ▷ check for terminal states
  for $t = 1 \rightarrow$ **T do**
    take random action a; $P(a) = \epsilon$
    otherwise:
      Forward propagate $\phi(\mathrm{s}_t)$ in the Q-network
      Execute $argmax_a Q(\phi(\mathrm{s}_t), a)$
    Observe rewards r and next state s_{t+1}
    Derive $\phi(\mathrm{s}_{t+1})$ from s_{t+1}
    $(\phi(\mathrm{s}_t), a, r, \phi(\mathrm{s}_{t+1})) \rightarrow$ replay memory **D**
    sample random mini-batch of transitions from **D**
    if *finished* **then**
      set targets **y** *by forward propagating* $\phi(s_{t+1})$
      then compute loss
    update parameters with gradient descent

4.3. Results

Using Experience Replay 4.1 and Exploration v/s Exploitation 4.2, our model has been able to achieve excellent results. These results are presented in Figure 2.

Figure 2. Total Rewards for DQN after 399 episodes.



The model was allowed to run for 390+ episodes. For the first 100 episodes, the average reward is negative because the agent is exploring the environment and collecting information to store in *D*. After the first 100 episodes, our agent rapidly increases consistency in being able to achieve a perfect score. In all the models tested until now, DQN's performance has been the best although it took $\sim$ 11 hours

to train the DQN agent. In the next steps, we aim to explore additional ways of increasing consistency and to analyse the effect of removing the Experience Relay and Exploration v/s Exploitation features from our current model.

To further improve results obtained using DQN, we tried setting environment seed values as hyper-parameters for the Model. However, we were not able to find a way to find a reliable and fast way to this in an automated manner. Manually setting seed values to 42, 1339 and 1995, we did not observe any notable improvements over the provided result (seed value set to 21).

5. Policy Gradient

Policy gradient aims to maximize the expected reward of an episode. The actor is represented in the form of a neural network, which is trained and improved incrementally based on the gradient of the expected reward given the network. This process is commonly known as gradient ascent. Among the variants of Policy Gradient algorithms, this section focuses on applying REINFORCE (Monte Carlo Policy Gradient) and Proximal Policy Optimization (PPO) to "LunarLander-v2".

5.1. REINFORCE

REINFORCE (Monte Carlo Policy Gradient) is the algorithm was implemented in the notebook of the tutorial. Currently, REINFORCE is able to create positive results, with the use of two different network architectures. Figure 3 shows the rewards for REINFORCE using 2 hidden layers with 16 features while figure 4 uses 2 hidden layers with 128 features, in hopes of boosting the performance of the network. Experiments were also done on changing the model's forward mechanism from tanh to ReLU, but the results were worse off. As seen below, while the performance for tanh on 128 features appears to perform better on average for reaching a score slightly over 100 after 1000 batches, the performance diverged a lot more, and it can be observed that the curve tends to drop dramatically occasionally. Having slow convergence has shown to be one of the issues for vanilla policy gradient, which brings one to explore off-policy methods such as Proximal Policy Optimization.

Figure 3. Total Rewards for REINFORCE with 16 features

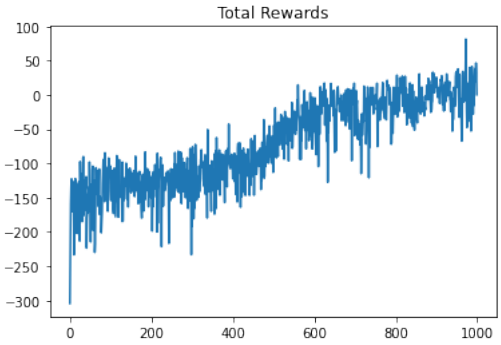
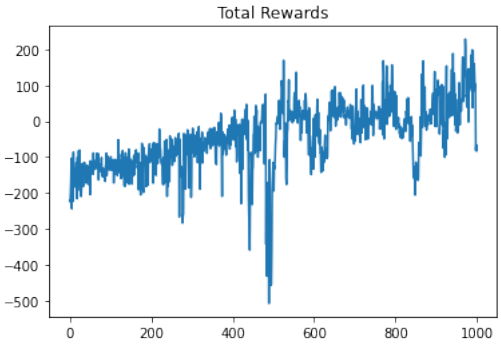


Figure 4. Total Rewards for REINFORCE with 128 features



5.2. PPO

Alternatively, Proximal Policy Optimization (PPO), which is the default algorithm for performing Reinforcement Learning tasks due to its promising performance, is a popular method in solving reinforcement learning problems [9]. Normal policy gradient methods suffer from high gradient variance and low convergence, which could impose conflicting descent directions, hindering the training process of the agent. As opposed to on-policy approaches, which require frequent sampling of state-action pairs, PPO adopts the off-policy method, which evaluates and improves a policy using state-action pairs from a different agent and performing importance sampling, speeding up convergence and the learning process.

The objective function of PPO implements a trust region update compatible with stochastic gradient descent to solve the problem [4]. The objective is defined as follows:

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

where $r_t(\theta) = \dfrac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$.

An existing implementation of applying PPO to LunarLander-v2 has been studied and referenced when

implementing the algorithm [6]. 200 updates were run, with 20 episodes in each update. The total rewards and final rewards for the algorithm is shown in figures 5 and 6 below.

Figure 5. Total Rewards for PPO in 200 updates

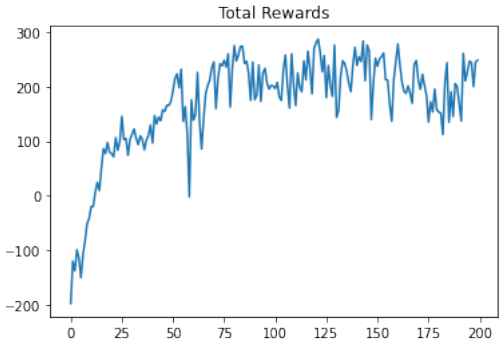
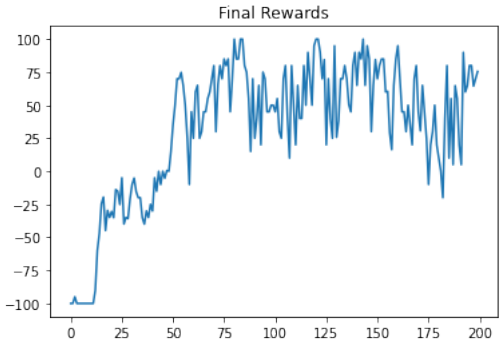


Figure 6. Final Rewards for PPO in 200 updates



6. SARSA

SARSA or State–action–reward–state–action is a reinforcement learning method used for determining Markov decision process policies, On implementing a naive Sarsa model on the problem in which the Sarsa agent updates the state action pair values according to the rule:

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha[r_t + Q(s_{t+1}, a_{t+1})Q(s_t, a_t)]$$

and further comparing it with a random agent choosing actions randomly we found that the Sarsa agent has better results. However as seen in Figure 7 even after a high number of iterations the average rewards for the Sarsa agent was negative indicating that it further needs to be optimised. The full algorithm for training the agent using the SARSA reinforcement method is shown in Figure 8 [2].

Figure 7. Rewards vs Iteration - SARSA

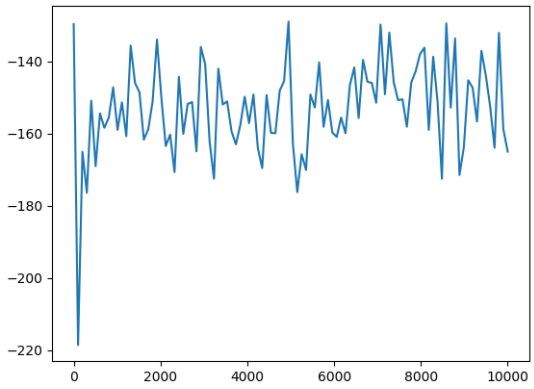


Figure 8. SARSA - Pseudo code

Sarsa (on-policy TD control) for estimating $Q \approx q_*$

Initialize $Q(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
  Initialize S
  Choose A from S using policy derived from Q (e.g., ϵ-greedy)
  Repeat (for each step of episode):
    Take action A, observe R, S'
    Choose A' from S' using policy derived from Q (e.g., ϵ-greedy)
    $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma Q(S',A') - Q(S,A)\big]$
    $S \leftarrow S'$; $A \leftarrow A'$;
  until S is terminal

6.1. State Discretization

Due to the large number of states and actions in the given environment, the state space becomes exponentially big due to which the Sarsa model isn't able to converge to the optimum solution within the given time frame and resources.To reduce the state space, there are some rules observed from the nature of the task which could be hard-coded into the agent. [15]

Some examples of observed rules include that the agent or the lunar-lander always wants to reach the horizontal centre of the space as soon as possible. For instance, if the agent starts at the left of the space, the rewards system and action pairs dictate it to move right and vice-versa.

After reaching the centre, the x-coordinate essentially gets fixed and the y-coordinate logically needs to only be decreased with the vertical velocity direction also always being negative.

With the end goal of determining the exact X-Y coordinates of the lunar-lander, by state discretization we can reduce the X-Y coordinates space from continuous variables to integral values. For example the co-ordinate be discretized into 7 states (-3,-2,1,0,1,2,3) & the Y-coordinate

be discretized into 3 states (-1,0,1). For the scope of this project, we shall be trying out different combinations of X-Y discretization to deduce a model providing satisfactory results.

6.2. Exploration Policy

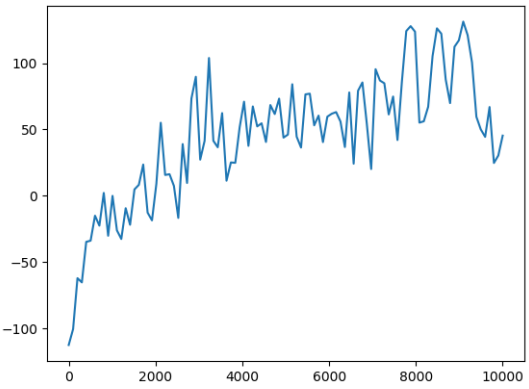
Even after state discretization, the number of state-action pairs is extremely large. To further optimize the agent, an exploration policy is used. We implemented an ϵ - greedy policy, where ϵ is the probability of agent acting sub-optimally/experimenting new state-action pair and $1 - \epsilon$ is the probability of agent acting optimally to max the rewards.

As the number of iterations increases and the agent learns more about the environment, the ϵ value is decayed such that at the latter stages, there is less sub-optimal behaviour that pushes the agent to converge on a solution. By hard-coding the values of ϵ based on the iteration no. and after fine tuning we obtained better results as shown in 9 & 10 .

6.3. Results

Using naive state discretization wherein both X& Y coordinate were reduced to one state each and further complementing it with an exploration function the first set of results obtained is shown below in 9.

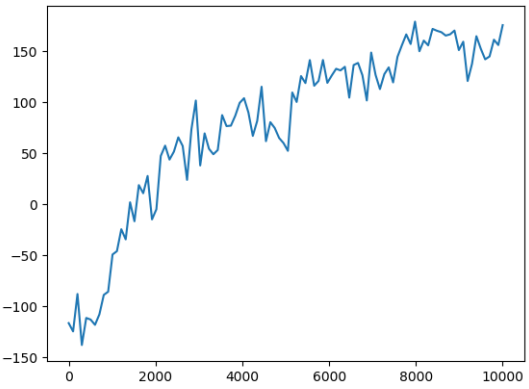
Figure 9. Rewards vs Iteration - SARSA with naive discretization & exploration function



After further trying out multiple combinations of X-Y discretizations, different exploration functions and extensively researching on methods to improve the reward function.We came across an exploration function and state-discretization combination which provided us with a satisfactory result. [15]

The discretizations include 5 states for X, 4 states for Y and the exploration function gradually reduces from 0.5 to 0 as the iterations increase. The result for this combination is shown below in 10.

Figure 10. Rewards vs Iteration - SARSA with best achieved optimizations



References

[1] Abhijit Banerjee, Dipendranath Ghosh, and Suvrojit Das. Evolving network topology in policy gradient reinforcement learning algorithms. In *2019 Second International Conference on Advanced Computational and Communication Paradigms (ICACCP)*, pages 1–5, 2019. 1

[2] Adesh Gautum. Introduction to reinforcement learning (coding sarsa) — part 4. In *https://medium.com/swlh/introduction-to-reinforcement-learning-coding-sarsa-part-4-2d64d6e37617*. 4

[3] HKUMMLab. Applied deep learning lecture notes (reinforcement learning). 1

[4] Jonathan Hui. Proximal policy optimization explained. In *https://jonathan-hui.medium.com/rl-proximal-policy-optimization-ppo-explained-77f014ec3f12*. 3

[5] Yingdong Lu, Mark Squillante, and Chai Wu. A control-model-based approach for reinforcement learning, 05 2019. 1

[6] Youness Mansar. Learning to play cartpole and lunarlander with proximal policy optimization. In *https://towardsdatascience.com/learning-to-play-cartpole-and-lunarlander-with-proximal-policy-optimization-dacbd6045417*. 4

[7] James G March. Exploration and exploitation in organizational learning. *Organization science*, 2(1):71–87, 1991. 2

[8] Google Deep Mind. Human-level control through deep reinforcement learning. In *https://storage.googleapis.com/deepmind-media/dqn/DQNNaturePaper.pdf*. 2

[9] OpenAI. Introduction to proximal policy optimization. In *https://openai.com/blog/openai-baselines-ppo/*. 3

[10] OpenAI. Openai gym documentation. In *https://gym.openai.com/docs/*. 1

[11] OpenAI. Openai lunarlander-v2 documentation. In *https://gym.openai.com/envs/LunarLander-v2/*. 1

[12] Chad Peters, Terrance Stewart, Robert West, and Babak Esfandiari4. Dynamic action selection in openai using spiking neural networks. In *https://www.aaai.org/ocs/index.php/20FLAIRS/FLAIRS19/paper//view/18312/17429*, 2019. 1

[13] Andrea Pierre. Openai gym - lunar-lander source code. In *https://github.com/openai/gym/blob/master/gym/envs/box2d/lunar*$_l$*ander.py*.2

[14] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*, 2015. 2

[15] Chengzhe Xu Yunfeng Xin, Soham Gadgil. Solving the lunar lander problem under uncertainty using reinforcement learning. In *https://web.stanford.edu/class/aa228/reports/2019/final8.pdf*. 1, 4, 5