One Small Step for Man - One Giant Leap for Reinforcement Learning

An Investigation of Modern Deep Reinforcement Learning Algorithms Using the Lunar Lander Open AI Gym Environment

Linden Black, *School of Physics and Astronomy, University of Nottingham, Nottingham, NG7 2RD, UK*

***Abstract*—In this paper reinforcement learning algorithms were tested on the Lunar Lander V2 Open AI Gym environment. In particular PPO2, A2C and DQN methods were tested with the performance of the agents compared. The DQN was found to perform the best, with mean reward over 100 episodes of 179 on a test environment.**

I. Introduction

THE core aim of the project was to investigate a selection of reinforcement learning (RL) agents using the OpenAI Gym LunarLander-V2 environment to test them. A secondary aim was to develop an automatic optimisation algorithm that determined which parameter configuration was optimal. The question - *Which reinforcement learning algorithm performs best on the Lunar Lander game?*

II. Environment

The environment used was the Open AI Gym LunarLander-V2 game [1]. The evironment was selected as the action space can be discretised, which is a prerequisite for one the algorithms used in the paper and the state space can be described by a vector of 8 features. More complex environments often require computer vision to process the state space. The aim of the project was to investigate agent performance, as such computer vision was not included to reduce the complexity of the networks. Game based environments are well suited for testing RL agents, as a reward for the agent can easily be created from a score in the game. The aim of the Lunar Lander game is to control the descent of the lander and to direct it to land in the designated landing zone, marked by two flags, see figure 1. The episode finishes if the lander crashes or comes to rest, receiving an additional -100 or +100 reward points. If a landing leg makes ground contact 10 points are rewarded. Firing the main engine subtracts 0.3 points each frame. The action space A and state space S are described below:

$$S \longrightarrow \begin{cases} x\ coordinate \\ y\ coordinate \\ x\ velocity \\ y\ velocity \\ lander\ angle \\ lander\ angular\ velocity \\ leg\ 1\ ground\ contact \\ leg\ 2\ ground\ contact \end{cases}$$

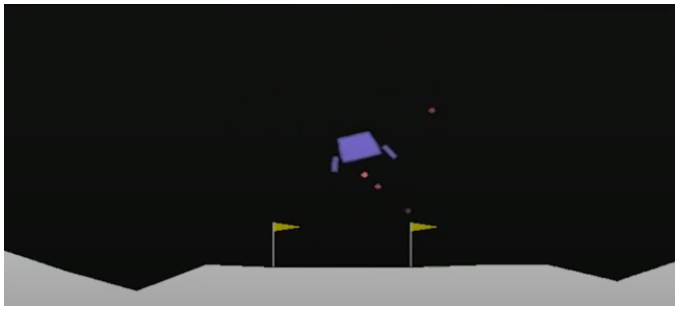

Fig. 1: The LunarLander-V2 display. Note the two flags that mark out the designated landing area.

$$A \longrightarrow \begin{cases} do\ nothing \\ fire\ left\ orientation\ engine \\ fire\ main\ engine \\ fire\ right\ orientation\ engine \end{cases}$$

III. Technologies and Agent Design

This paper will give a brief introduction to the methods used in the project. More detail on RL and the mathematics that underpins the deep RL algorithms discussed in this report can be found in the accompanying literature review.

A. Reinforcement Learning Introduction

RL considers the optimisation of an agents interaction with its environment. We define the state, s_t to be the observed environment at time step t and the action a_t, to be the action taken in that state at that time step. We also define a policy $\pi(s, a)$ that informs the agent to make a specific action based on its state. A reward, r_t, is given to the agent for making an action, if the action is good the reward will be greater than if the taken action was poor. The value of state can be defined using a discount factor, γ and summing over the future rewards from the optimal trajectory beyond that state.

$$v_\pi(s) = \mathbb{E}_\pi[R_t|S_t = s] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1}|S_t = s\right] \quad (1)$$

The Q-function is similar, however instead of considering the value of the state it considers the value of an action made in a specific state, with respect to the long term discounted

reward. Q-learning [2] uses an iterative procedure to solve the Q-function and through maximisation of the solution it finds the optimal action. Policy gradient methods parameterise the policy and attempt to learn an optimal policy through gradient ascent. One example of this is the advantage policy gradient method used in the advantage actor critic algorithm [3], the objective function is shown below,

$$L^{PG}(\theta) = \hat{\mathbb{E}}_t[log\pi_\theta(a_t|s_t)\hat{A}_t] \tag{2}$$

where $\hat{A}_t$ is an unbiased estimate of the advantage $A_t = Q_t(s,a) - V_\pi(s)$ which itself is a comparison of the long term reward for a specific action against the average long term reward for an action in that state.

B. Deep Reinforcement Learning

Deep-RL uses deep neural networks to parameterise reinforcement learning functions. Three deep-RL networks are investigated in this paper.

1) Deep Q-Network (DQN): For a small state space Q-learning is great, however when the state space increases in size an iterative procedure to find an exact solution is not computational efficient. A deep Q-network aims to parameterise and learn the Q function through gradient descent [4]. Deep Q-learning uses a target and online network. The target network generates training data which is stored in a replay buffer and the online network randomly uniformly samples from this buffer and updates its parameters using gradient descent. The use of this target network buffer system means that the network is not attempting to learn a moving target and it removes the correlation that would be present if the online network made immediate predictions for its next input. Without the replay buffer the training would be highly unstable and divergent. To encourage exploration an epsilon greedy policy is implemented, without this the model will exploit the best action under the current policy and might miss out on the optimal policy from being trapped in a local minima. The policy is stochastic and introduces a small probability of taking a random action instead of the optimal action as determined by maximising the output from the deep Q-network . An epsilon value of $\epsilon = 0.05$ was used. In the Atari and Deep Mind DQN papers the target network was 'hard updated' every 1000 steps, copying the online network and setting the target network as the copy. An alternative to this is to use a soft-update (Polyak Update) [5]. Instead of an occasional full update, a frequent small update is used instead. The size of this update is determined by hyper-parameter, τ, which scales the target and online network as shown below. $\theta^{target\prime}$ represents the new target network parameters and θ^{target} the current target network parameters, θ^{online} represents the online network parameters.

$$\theta^{target\prime} = \theta^{online} \cdot \tau + \theta^{target} \cdot (1 - \tau) \tag{3}$$

In this project soft updates were used, with $\tau = 0.001$.

2) Advantage actor critic (A2C): A2C is a deterministic synchronous version of the asynchronous A3C algorithm [6]. It implements an actor critic framework using the advantage function as the critic and the policy as the actor. Both actor and critic are parameterised by neural networks.

3) Proximal Policy Optimisation (PPO): Proximal Policy Optimisation [7], is a policy gradient method that uses a complex loss function to clip gradient scaling to within $[1-\epsilon, 1+\epsilon]$ helping to smooth training, where ϵ is the clip-range and is a hyper-parameter. PPO2 is optimised for multiprocessing using a GPU.

4) Stable Baselines: Stable Baselines (SB) is a python reinforcement learning library that has pre-built versions of many of the most popular algorithms in deep-RL. Pre-built versions of the A2C, PPO2 and DQN models were used. SB models can be easily integrated into OpenAI Gym environments so were the natural choice for pre-built models.

5) Keras: This paper presents a self-coded implementation of a DQN built using Tensorflow's Keras with no use of reinforcement learning libraries.

6) Agent Challenges: In deep reinforcement learning hyper parameter optimisation is challenging, the initial action predictions made by the networks are erroneous and only when the networks have been sufficiently trained do they begin to output the correct actions. As such optimising the hyper parameters on a small amount of data, as is common practice in supervised learning, is not as effective. The need to constantly generate new data and train for a high number of steps meant that training was expensive and time consuming. Due to local PC limitations cloud computing with GPUs access was necessary. For this project Google Colab Pro was used, it allows a job to run for 24 hours with a Tesla K80 GPU provided.

IV. Experimental Setup

1) Experiment I - Hyper-Parameter Optimisation: To optimise the hyper-parameters of the SB models a function was created to iterate through parameter dictionaries for each model, with the aim of finding the optimal configuration. The agent was initialised with the parameters and trained for 1000 steps, a smaller number than desirable but due to hardware limitations a reduced number had to be used. The models were then evaluated for 100 episodes in a test environment and the results saved. The configuration that demonstrated the highest mean reward over the 100 test episodes was considered optimal and used for final training. The Keras model was optimised empirically without the use of a function.

2) Experiment II - Environment Completion: The maximum number of training steps for each model was set at 100000. OpenAI Gym describes an environment as being solved when a certain mean reward is reached, for the Lunar Lander environment this threshold is 200. To incorporate this, validation testing was run every 500 steps. 5 episodes were run on a new initialisation of the environment using the trained model as according to the OpenAI Gym documentation. If the mean reward was over 200 then training was halted and the model saved. This helped to reduce over fitting and also saved on computation time.

3) Experiment III - Optimised Agent Comparison: An *Evaluator* class was created to test and compare the final models. The trained models were loaded as class attributes. The models were each evaluated for 100 episodes using a *testModel* function that recorded the mean rewards and the

mean time taken to make a prediction. Prediction time is important for agents that run within a game environment as they need to react to changes in the state of the environment quickly.

V. Results and Discussion

The results of the hyper-parameter testing are shown in the appendix. Three out of the four models reached the completion score of 200 before the training ended. On training the Keras DQN converged the fasted, most likely owing to its relative simplicity, with only a single network being trained. The PPO2 algorithm did not converge even after 100000 steps, a poor hyper-parameter initialisation could have caused this. The instability of initial training with the policy gradient A2C method can be seen in Fig.2, this is one of the largest drawbacks to policy gradient methods. The PPO2 method that was designed to stabilise training has a flatter curve , however it does not converge within the maximum step allocation. On the test environment of 179, close to the 200 that indicates completion. This was unsurprising as DQNs are highly effective in discrete actions space environments. The DQN created using Keras performed second best, the difference between the two can be explained by the more advanced features in the SB implementation. Duelling [8] and double DQNs [9] were used, alongside a prioritised experience replay buffer [10], these three features have all been shown to outperform the vanilla DQNs. PPO2 performed poorly, this was unsurprising as it was the only model that did not reach the 200 reward threshold during training. A2C demonstrated the fastest mean prediction time with the SB DQN coming second. The difference between the SB and Keras DQN can be explained again, hyper-parameter testing on the Keras model found a $128, 64, 32$ hidden layer configuration to perform the best, the SB DQN used $64, 64$ - a shallower network will result in a faster prediction time. The core aim of this project

Model	Mean Reward	Mean Prediction Time
A2C	-1.14	0.001259
PPO2	-21.65	0.001607
DQN	178.87	0.001896
Keras DQN	136.26	0.002883

TABLE I: Performance of the optimised models on the final test environment

was to investigate which RL algorithm performed the best on the Lunar Lander environment. Through hyper-parameter optimisation, model training and evaluation of the trained models it was found that the stable baseline DQN model was the most effective in the environment, completing the environment by achieving the greatest mean reward of 179 on the test environment.

VI. Conclusion

To conclude, this paper presents four deep-RL models for playing the LunarLander-V2 game within the OpenAI Gym. A function for the full optimisation and testing of the Stable Baselines deep-RL models was created. A second function that compared all the agents in a test environment

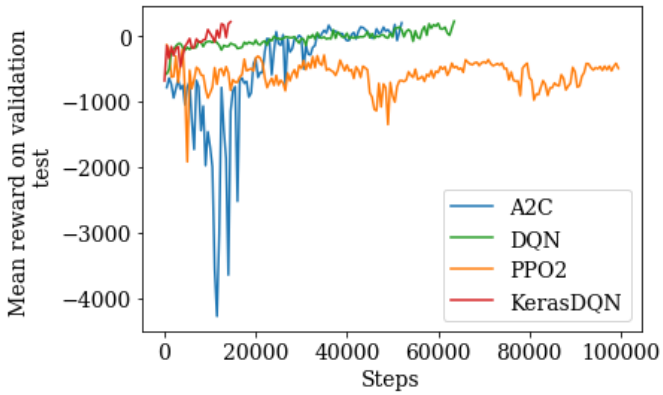

Fig. 2: Mean rewards on the validation test occurring every 500 training steps. Note that PPO2 is the only algorithm that did not converge within the 100000 step limit. Also note the initial instability of A2C.

that investigated mean reward and prediction time was also created. It was found that the Stable Baselines DQN using a double, duelling DQN with prioritised experienced replay performed best. Due to hardware limitations optimisation and training was not as extensive as desired, as such the hyper-parameter configurations used for final training may not have been optimal. Further work could include testing the Keras DQN in other environments that have a more complex action space. Further improvements of the Keras DQN such as adding a prioritized experience replay buffer as well as a double DQN could be explored. A new and highly effective branch of RL is that of 'learning from expert experience' where an agent is trained initially on games performed by a human expert and then it generates its own data and trains itself. This can help with speeding up the time to converge which is one of the greatest issues with RL. Given more time this would have been implemented into the code.

References

[1] Openai, "openai/gym," Apr 2020. [Online]. Available: https://github.com/openai/gym/blob/master/gym/envs/box2d/lunar_lander.py

[2] W. Christopher, "Technical note: Q-learning," *Machine Learning*, vol. 8, no. 3, pp. 279 – 292, 1992.

[3] R. S. Sutton, D. McAllester, S. Singh, and Y. Mansour, "Policy gradient methods for reinforcement learning with function approximation," in *Proceedings of the 12th International Conference on Neural Information Processing Systems*, ser. NIPS'99. Cambridge, MA, USA: MIT Press, 1999, p. 1057–1063.

[4] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," 2013.

[5] B. Polyak and A. Juditsky, "Acceleration of stochastic approximation by averaging," *SIAM Journal on Control and Optimization*, vol. 30, pp. 838–855, 07 1992.

[6] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," 2016.

[7] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," 2017.
[8] Z. Wang, T. Schaul, M. Hessel, H. van Hasselt, M. Lanctot, and N. de Freitas, "Dueling network architectures for deep reinforcement learning," 2016.
[9] H. van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," 2015.
[10] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay," 2016.

APPENDIX

Hyper-Parameter	**Value**
Gamma	0.95
Number of steps	5
Entropy coefficient	0.01
Learning rate	0.0007

TABLE II: A2C optimal hyper parameters

Hyper-Parameter	**Value**
Gamma	0.99
Number of steps	132
Entropy coefficient	0.02
Learning rate	0.00025

TABLE III: PPO2 optimal hyper parameters

Hyper-Parameter	**Value**
Gammas	0.99
Double Q-network	$True$
Batch size	32
Learning rate	0.0007
Hidden layers	$64, 64$

TABLE IV: Stable baselines DQN optimal hyper parameters

Hyper-Parameter	**Value**
γ	0.99
ϵ	0.05
τ	0.01
Learning rate	0.0005
Batch size	32
Hidden layers neurons	$128, 64, 32$
Kernel initialisation	$Glorot\ uniform$
Hidden layer activation	$ReLU$
Replay buffer length	10000

TABLE V: Keras DQN optimal hyper parameters

Model	**Number of Steps to Converge**
A2C	51500
PPO2	$Did\ not\ converge$
DQN	63000
Keras DQN	14500

TABLE VI: Number of training steps before model convergence on a mean validation reward of 200.