Solving the Lunar Lander Problem Using Reinforcement Learning

Yiming Liu
Department of Computer Science
University of Bath
yl4167@bath.ac.uk

John Georgousis
Department of Computer Science
University of Bath
ig441@bath.ac.uk

Emtiaz Samad
Department of Computer Science
University of Bath
ems88@bath.ac.uk

Hang Yin
Department of Computer Science
University of Bath
hy672@bath.ac.uk

1 Problem Definition

The Reinforcement Learning problem chosen for this project is OpenAI Gym's *LunarLander-v2* [6]. The environment consists of a lander (the agent) whose goal is to optimally land on a designated area on a lunar surface. This is done through the use of thrusters under low-gravity conditions.

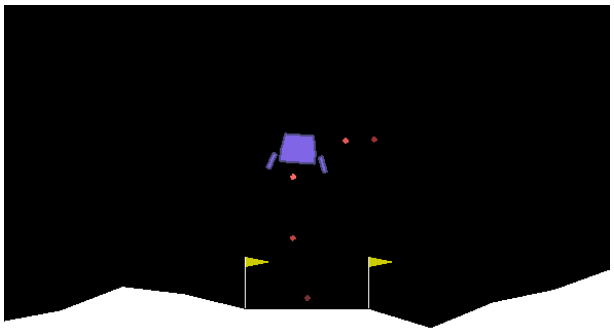

Figure 1: Visualisation of the Lunar Lander problem [3]

States, Actions, and Transition Dynamics
The state space of the lander has eight dimensions, with six continuous state variables and two discrete (boolean) ones [6]. The state vector consists of the horizontal position, vertical position, horizontal velocity, vertical velocity, angle, angular velocity, left leg contact , right leg contact:

$$(x,\ y,\ v_x,\ v_y,\ \theta,\ \omega,\ R,\ L). \tag{1}$$

The starting state of the lander is at the top centre with a random initial force applied to its centre of mass (i.e. random v_x and v_y). Then, the agent can change its state through a set of actions.

The action space has the following 4 discrete dimensions:

1. Do nothing.
2. Fire the main engine (constant increase to v_y).
3. Fire the left engine (constant increase to v_x).
4. Fire the right engine (constant decrease to v_x).

The landing pad is always at coordinates $(0, 0)$ and the fuel is infinite.

Rewards
The agent is rewarded for a soft, fuel-efficient landing, and an episode ends when the lander crashes, comes to rest, or gets outside of the viewport [6]. The exact rewards are the following:

- Lander crashes: -100.
- Lander comes to rest: $+100$.
- Each leg makes ground contact: $+10$.
- Each leg loses ground contact: -10.
- Firing main engine: -0.3 per frame.
- Firing side engine: -0.03 per frame.
- Comes to rest between flags: $100 - 140$ (depending on the positioning on the pad.)

Note that the reward for landing between the flags will be lost if the agent later leaves the area. The problem is considered optimally solved when a total reward of +200 is consistently obtained.

Remarks
The theoretical maximum reward based on official documentation is 260 without the use of any engines. However, our agent was observed to sometimes inexplicably obtain a reward of 280+ in a single episode. This inconsistency was also noted in [10] and [11]. Default parameters for v2 were used. Finally, the surface around the landing pad varies with each episode, while the landing pad surface remains flat.

Lunar Lander Parameters

"This environment is a classic rocket trajectory optimization problem. According to Pontryagin's maximum principle, it is optimal to fire the engine at full throttle or turn it off. This is the reason why this environment has discrete actions: engine on or off." [6]

2 Background

There are numerous reinforcement learning methods that are applied through different algorithms and each one can lead to drastically different performances on a given problem. In this section we will cover the theory behind the most impactful ones and discuss how their strengths and weaknesses.

Tabular vs Non-Tabular Methods
Tabular methods do not work because of the six continuous state variables. These result in a practically infinite number of states, making a tabular representation of state values $v_\pi(s)$ impossible. Therefore, we instead need to use function approximation methods where $v_\pi(s)$ is represented as a real-valued function with parameter vector $\boldsymbol{\theta}$. This allows us to use what we know about one state to make inferences about another, a concept known as *generalisation*. However, function approximation methods (especially in the domain of *deep reinforcement learning*) can suffer from *instability* [4].

Instability: On-Policy vs Off-Policy Methods
Instability is when a small change in input (e.g. training data or experience) leads to a large change in the agent's performance. This affects the agent's ability to converge on a near-optimal policy. Figure 2 shows the instability of a deep reinforcement learning model (Advantage Actor Critic or A2C) compared to a more stable, modified version. Whether an algorithm uses On-Policy or Off-Policy methods is a strong determinant of stability. In On-Policy algorithms (e.g. SARSA, Policy Gradient, etc.) the agent optimises the policy that it is actually following (even if it is an ϵ-soft policy), which often cannot converge on the optimal policy. An effect of this is that old data is not

used, making On-Policy methods less sample efficient [5]. In Off-Policy algorithms (e.g. Q-Learning, Deep Q-Network) the agent's behaviour policy is different to the agent's target policy (which can converge on the optimal policy). While Off-Policy agents pursue the optimal policy and reuse old data efficiently, this comes with mathematical complications [10] leading to instability. In our problem, this means that the Lunar Lander is more likely to converge on a near-optimal policy using On-Policy algorithms such as SARSA or Proximal Policy Optimisation.

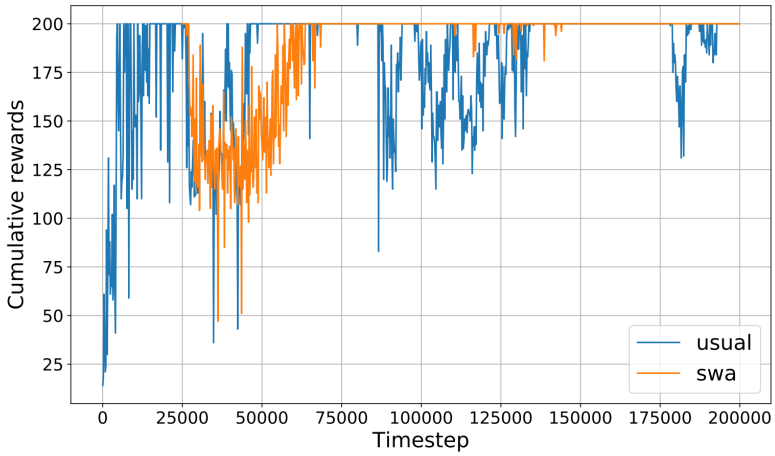

Figure 2: Stability of default A2C (usual) in the Cart-Pole environment versus with a stability increasing technique (swa) [10].

Action-Value Methods vs Policy Gradient Methods
The last class of RL methods that we will be exploring is Policy Gradient Methods and Action-Value Methods. In action-value methods, during learning, the agent optimises value estimates for all state-action pairs it observes to then make decisions based on them using its policy. However, in policy gradient methods, the agent learns a *parameterised policy* $\pi_{\boldsymbol{\theta}}(\boldsymbol{a} \mid \boldsymbol{s}) = \boldsymbol{P}[\boldsymbol{a} \mid \boldsymbol{s}]$ which outputs a probability distribution of actions making it possible to select actions without the need of a value function [18]. The parameter vector $\boldsymbol{\theta}$ is updated by maximising an objective function through gradient ascent:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \boldsymbol{\alpha}\widehat{\boldsymbol{\nabla J}\left(\boldsymbol{\theta}_t\right)} \tag{2}$$

Note that as a result, Policy Gradient Methods lead to a stochastic policy (however a deterministic one can often be approximated). This has been shown to be disadvantageous in our environment since the Lunar Lander is likely to be deterministic rather than stochastic [1]. However, it has been shown that for some environments a stochastic policy is easier to approximate than an action-value function [18], leading to shorter training times and superior convergence. For our problem, we chose to compare two Policy Gradient Methods and one Action-Value Method.

2.1 Previous Work

With the lunar lander being one of the most popular RL problems, there has been considerable previous work done in solving it. In [10], the authors solved it using Sarsa and DQN, achieving 170+ and 200+ rewards respectively. They also introduced uncertainty into the environment and achieved 100+ rewards with both agents. In [10], Q-Learning, DQN, and DDQN agents were compared and it was found that DDQN and DQN failed to consistently solve the problem while vanilla Q-Learning managed to converge to a solution. In [12], DQN with experience learning was implemented and managed to solve the problem with an average score of 200+. In [1], DQN and REINFORCE were compared but they both failed to solve the problem.

3 Methods

3.1 REINFORCE with Baseline Algorithm

REINFORCE is a policy gradient algorithm that optimises a policy using gradient ascent. It can be considered a Monte Carlo method for learning the policy parameter, θ. For baseline, it seems natural to also use a Monte Carlo method to learn the state-value weights, w.

This algorithm holds two step sizes, denoted α^{θ} and $\alpha^{\mathbf{w}}$. Choosing the step size for values (here w) is relatively straightforward; in the linear case, there are rules of thumb for setting it, such as $\alpha^{\mathbf{w}} = 0.1/\mathbb{E}\left[\left\|\nabla\hat{v}\left(S_t, \mathbf{w}\right)\right\|_{\mu}^2\right]$ It is much less clear how to set the step size for the policy parameters, α^{θ}, whose optimal value relies on the range of variation of the rewards and on the policy parameterization [18].

3.2 Proximal Policy Optimisation (PPO)

PPO is a popular policy-optimisation algorithm [7] that has is the evolution of the Vanilla Policy Gradient (VPG) algorithm [9] and variation of the Trust Region Policy Optimisation (TRPO) algorithm [8]. Its methods are similar to those of the REINFORCE algorithm.

These algorithms are *on-policy* algorithms, where stability is favoured over sample efficiency by neglecting old data while learning the behaviour policy rather than the optimal policy. However, PPO has built-in techniques for fixing sample efficiency issues.

PPO works by optimising a stochastic policy $\pi_{\theta}(a \mid s)$ that outputs a probability distribution of actions. The update rule for parameters $\boldsymbol{\theta}$ aims to maximise the objective function:

$$\theta_{k+1} = \arg\max_{\theta} \mathrm{E}\left[L\left(s, a, \theta_k, \theta\right)\right], \tag{3}$$

where L is given by,

$$L\left(s, a, \theta_k, \theta\right) = \min\left(\frac{\pi_{\theta}(a \mid s)}{\pi_{\theta_k}(a \mid s)} A^{\pi_{\theta_k}}(s, a), \quad clip\left(\frac{\pi_{\theta}(a \mid s)}{\pi_{\theta_k}(a \mid s)}, 1-\epsilon, 1+\epsilon\right) A^{\pi_{\theta_k}}(s, a)\right) \tag{4}$$

.

For this project, the PPO-Clip variation of the algorithm was chosen. PPO aims to increase policy improvement by updating the policy in small steps to avoid performance collapse. This is achieved via clipping the objective function, making it unfavourable for the new policy to significantly diverge from the old policy. The PPO-Clip pseudocode is shown in Appendix B.

PPO was chosen because it is one of the most popular RL algorithms with the unique features of being a policy optimisation, on-policy algorithm, allowing us explore how these features perform on the Lunar Lander problem.

3.3 Double Deep Q-Networks (DDQN)

DDQN is an extension of DQN which is a non-tabular Q-learning algorithm using neural networks to approximate to approximate policy instead of a Q-table. In DQN, there is a neural network which takes the state of the agent as input and outputs the estimated value of each action the agent can take. A visualisation of this is provided in Figure 4.

(This system needs a few fixes to be effective, namely experience replay, target networks, and reward clipping.)

The issue with DQN is that its neural network approximates values that are interrelated (leading to high bias) and it tends to be overoptimistic. Double DQN addresses these issues by separating the selection and evaluation of actions so that a different Q-Network can be used for each step. This is achieved by implementing a target Q-Network which evaluates the actions selected by the online Q-Network.

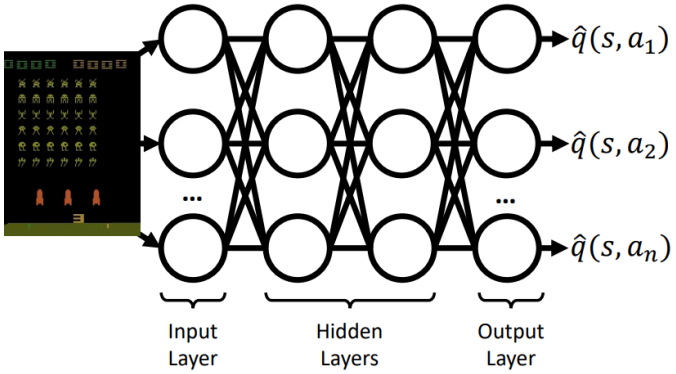

Figure 3: DQN Neural Network taking an image (state) as input and value estimates for each action [2]

Essentially, the Double DQN algorithm is identical to DQN with a minor, albeit important modification; Double DQN replaces the target of DQN with,

$$Y_t^{DoubleDQN} \equiv R_{t+1} + \gamma Q\left(S_{t+1}, \arg\max_{\alpha} Q\left(S_{t+1}, a; \theta_t\right), \theta_t^-\right) \tag{5}$$

This change builds upon the original Double Q-Learning algorithm, which produces two sets of weights, $\boldsymbol{\theta}$ and $\boldsymbol{\theta}'$, from two value functions which are learned by randomly by assigning experiences to randomly update either of the value functions. One set of weights is used to determine the greedy policy while the other determines its value for each episode. However, Double DQN replaces the weights of the second network $\boldsymbol{\theta_t^-}$ with the weights of the target network $\boldsymbol{\theta_t'}$. Evidently, Double DQN retains the benefits of Double Q-learning, while maintaining the structure of the DQN algorithm.

Double DQN was chosen as one of the contenders since it provides stability and ensures reliable learning through reducing over-estimations.

4 Results

After the implementation of the aforementioned algorithms, we let them train on the Lunar Lander problem over 2000 episodes. Then, their average rewards along with their fluctuations were plotted as shown in Figure 5. The performance of a random agent is shown for comparison. Note that the Lunar Lander problem is considered solved when an average reward of 200+ is consistently obtained (e.g. over the previous 100 episodes).

REINFORCE
The REINFORCE algorithm yielded the worst performance (average reward) over this episode range. Its final version did not manage to solve the problem as its maximum average reward was around 100. In terms of the environment, the behaviour that it had ended up following in the end was to safely land on a surface (not necessarily on the landing pad) and then hover for a long period of time. It did not seem to care to pursue the landing pad too frequently and it spent too much fuel on hovering even after having reached the surface. Note that if given more time we could see a convergence on a more rewarding behaviour since even through 2000 episodes there was still an uptrend in reward. Total training time for REINFORCE was approximately 28 mins for 2000 episodes.

PPO
The PPO algorithm converged faster than any other algorithm but it was incredibly unstable. This was a surprise since most research suggested more stability than DDQN, though it could have been due to an unfavourable set of hyperparameters. PPO also failed to solve the problem as its average reward could not reach 200 consistently. Its behaviour in each episode would vary significantly, sometimes

landing in the pad efficiently while other times flying around and crashing. Total training time for PPO was 58 minnutes for 2000 episodes.

DDQN

Finally, DDQN was our best-performing agent and it managed to solve the problem after the first 400 episodes. Its stability was significantly better than PPO and REINFORCE. After analysing several episodes it was observed that DDQN would land on the pad safely, efficiently, and consistently. Its total training time was 73 minutes for 2000 episodes, though it converged long before that.

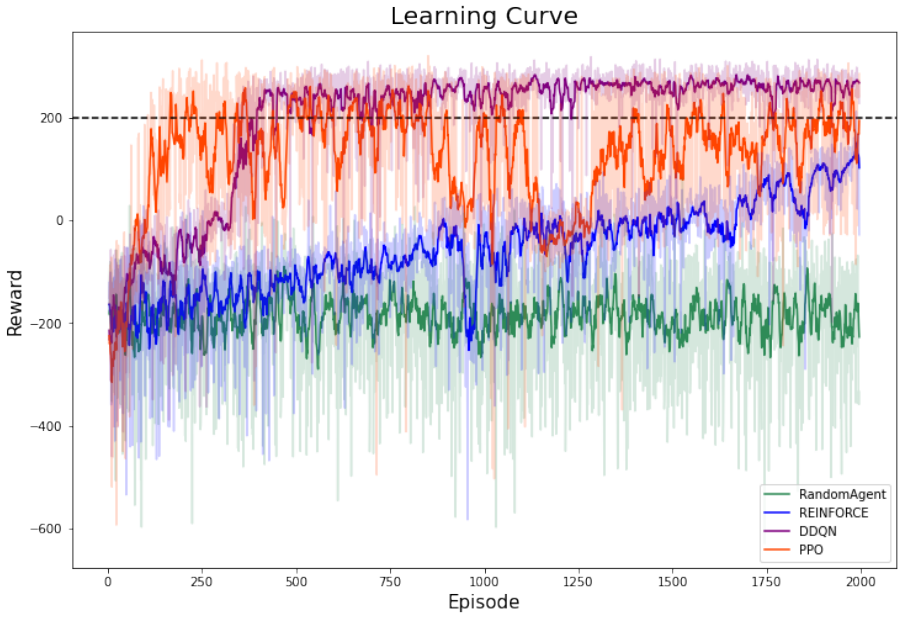

Figure 4: Learning Curve for DDQN, PPO and REINFORCE.

5 Discussion

In the end, the problem was solved and we gained a fair amount of insights into algorithm performance and metrics. REINFORCE failed to solve a problem but we were unable to find a REINFORCE agent that manages to solve it. PPO agents in literature have solved the problem though our agent also failed. However, DDQN solved it consistently and efficiently, achieving a score that other papers have failed to achieve consistently [10] [1].

The results we obtained were particularly interesting as they had not aligned with our initial predictions. Based on prior research, we had expected REINFORCE and PPO to outperform DDQN, however during the training process it became apparent that DDQN performed considerably better.

Upon realising the subpar performance of REINFORCE, we theorised reasons which could have contributed to this. We analysed video footage of episodes where the REINFORCE agent had learnt where the landing zone is located, and concluded that if the terminal conditions of leaving the frame or crashing were not satisfied, then the lander would continue to hover for 20 seconds. This meant that the probability assigned to the action of 'doing nothing' was minuscule so the action would rarely be taken. This significantly hindered the agents learning; the agent becomes 'trapped' in to achieving mediocre rewards because in the earlier episodes it learns to not take the action of 'doing nothing', but suffers in achieving optimal rewards as a result. As the action of 'doing nothing' is assigned a very small probability, the agent rarely achieves scores above 200, it doesn't learn how to achieve such scores and ultimately settles to achieve the mediocre rewards rather than negative rewards.

In regard to the time we had to complete this project, it was not feasible to produce a solution to this issue. Given more time, we would incorporate some randomness into the policy to allow any of the actions to be taken with a uniform probability distribution if like an epsilon greedy policy. This

would offset the low probability of some of the actions and ensure that such actions are played more frequently. We believe that this may help the agent learn reliably in the earlier episodes of training.

6 Future Work

Regardless of the sudden increase in popularity of reinforcement learning in the past years, there is still a substantial amount of research to be done.

Our Algorithms In the algorithms that we chose to apply to the Lunar Lander problem (DDQN, REINFROCE, PPO), there could be implementation and comparison of hyperparameter tuning over a wider range or parameters to understand how they affect an agent's performance. In addition, there could be further analysis on the long-term stability of these algorithms as we observed considerable instability after 2000 episodes.

Other Algorithms Moreover, given more time and resources, more state-of-the-art algorithms could be tested on the Lunar Lander problem and have their performances analysed and compared. Namely, we observed a lack of applications of A2C, A3C, Deep Deterministic Policy Gradient (DDPG), and Evolution Stategies (ES) algorithms to the Lunar Lander problem while they have the potential for significantly different results.

New Environments Regarding the environment that the agent learns in, there could be modifications to our chosen environment of Lunarlander-v2 (such as the addition of uncertainty or placing the landing pad on a steep slope), or the applications of these agents to entirely new environments to see how their performances change with environments while keeping their parameters constant.

7 Personal Experience

Initial Impressions Our personal experience was overall positive and challenging. Initially, discovering an RL problem to solve along with the algorithms we would use to solve it was simple due to the plethora of interesting articles on the topic. After unanimously agreeing on the Lundar Lander environment, we began researching algorithms that would be effective at solving the problem but would also provide useful novelties that had not been explored in the past (by us or others).

Algorithm Implementation After choosing REINFORCE, PPO, and DDQN, came one of the most challenging tasks of this project: Understanding the underlying theory of each algorithm and implementing it in a programming environment. A variety of mathematical equations had to be understood while even more coding errors were being resolved. This was by far the most difficult and time-consuming task of this project, but it was also the most intriguing as we collectively discussed and learnt numerous new ideas in the domain of reinforcement learning.
Finally getting the agents to train was incredibly satisfying and we quickly moved on the plotting of rewards and implementation of results. Another challenge was finding efficient ways to extract given experiences of our agent in mp4 format, but the product of this (the agent video) was gratifying in showing us how much improvement was made.

Authoring a Report & Recording a Presentation Putting the coding aside, the addition of a written report was beneficial and enjoyable to everyone. It helped consolidate our newly acquired knowledge and gave us a chance to share our work along with the video presentation. The presentation was a fun experience for everyone and helped us reflect on and appreciate the substantial work that we had all completed.

References

[1] URL: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9682970 (visited on 05/05/2022).

[2] URL: https://moodle.bath.ac.uk/pluginfile.php/1100399/course/section/174649/Lecture (visited on 05/05/2022).

[3] Soham Gadgil, Yunfeng Xin, and Chengzhe Xu. "Solving The Lunar Lander Problem under Uncertainty using Reinforcement Learning". In: (2020). DOI: 10.48550/ARXIV.2011.11850. URL: https://arxiv.org/abs/2011.11850.

[4] "Improving Stability in Deep Reinforcement Learning with Weight Averaging". In: (2018). URL: https://www.gatsby.ucl.ac.uk/~balaji/udl-camera-ready/UDL-24.pdf.

[5] OpenAI. *Algorithms - Spinning Up documentation*. URL: https://spinningup.openai.com/en/latest/user/algorithms.html (visited on 05/05/2022).

[6] OpenAI. *gym/lunar_lander.py at master openai/gym*. URL: https://github.com/openai/gym/blob/master/gym/envs/box2d/lunar_lander.py (visited on 05/05/2022).

[7] OpenAI. *Proximal Policy Optimization*. URL: https://spinningup.openai.com/en/latest/algorithms/ppo.html (visited on 05/05/2022).

[8] OpenAI. *Trust Region Policy Optimization*. URL: https://spinningup.openai.com/en/latest/algorithms/trpo.html (visited on 05/05/2022).

[9] OpenAI. *Vanilla Policy Gradient*. URL: https://spinningup.openai.com/en/latest/algorithms/vpg.html (visited on 05/05/2022).

[10] *Powered Landing Guidance Algorithms Using Reinforcement Learning Methods for Lunar Lander Case*. URL: http://repositori.lapan.go.id/964/ (visited on 05/05/2022).

[11] Xinli Yu. "Deep Q-Learning on Lunar Lander Game". In: (May 2019).

[12] Zhiqi Yu. *Solve LunarLander-v2 with Deep Q-learning*. URL: https://zhiqiyu.github.io/post/ll-dqn/ (visited on 05/05/2022).

[13] https://stanford-cs221.github.io/autumn2019-extra/posters/113.pdf

[14] Module Textbook

[15] Chapter 13, Reinforcement Learning (2nd Ed.), Sutton Barto 2018

[16] Deep Reinforcement Learning with Double Q-learning, Hasselt et al., 2015

[17] Proximal Policy Optimization Algorithms, Schulman et al. 2017

[18] http://www.incompleteideas.net/book/RLbook2020.pdf

Appendices

Appendix A: Parameters and hyperparameters

		REINFORCE		DDQN	PPO	
Discount factor		0.99		0.99	0.99	
Learning rate	Policy learning rate	0.0025		0.0001	0.0003	
	Value learning rate	0.005		None	None	
Epsilon	Intial value	None		1	None	
	Epsilon decay rate	None		0.995	None	
GAE(generalized advantage estimation)		None		None	0.95	
Policy clip		None		None	0.2	
Batch size of shuffling		None		None	5	
Num of episodes		2000		2000	2000	
Baseline		TRUE		FALSE	FALSE	
Neural Network Architecture		*Model*	*Model baseline*	*Model*	*ActorNetwork*	*CriticNetwork*
		Dense(16, 'relu')	Dense(16, 'relu')	Dense(128, 'relu')	Linear(4, 256, 'relu')	Linear(4, 256, 'relu')
		Dense(16, 'relu')	Dense(16, 'relu')	Dense(128, 'relu')	Linear(256, 256, 'relu')	Linear(256, 256, 'relu')
		Dense(16, 'relu')	Dense(16, 'relu')	Dense(4, 'linear')	Linear(256, 128, 'relu')	Linear(256, 128, 'relu')
		Dense(4, 'softmax')	Dense(1)		Linear(128, 4, 'relu')	Linear(128, 1, 'relu')
	Loss	categorical crossentropy	mse	mse	Linear(-1, 'softmax')	
	Optimizer	Adam	Adam	Adam	Adam	Adam
	Batch size of training	(default)	(default)	64	(default)	(default)

Figure 5: Parameters and hyperparameters for REINFORCE, DDQN and PPO.

LunarLander-v2 parameters: Default [6].

Appendix B: Pseudocodes

REINFORCE with Baseline (episodic), for estimating $\pi_{\boldsymbol{\theta}} \approx \pi_*$

Input: a differentiable policy parameterization $\pi(a|s, \boldsymbol{\theta})$
Input: a differentiable state-value function parameterization $\hat{v}(s,\mathbf{w})$
Algorithm parameters: step sizes $\alpha^{\boldsymbol{\theta}} > 0$, $\alpha^{\mathbf{w}} > 0$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ and state-value weights $\mathbf{w} \in \mathbb{R}^d$ (e.g., to $\mathbf{0}$)

Loop forever (for each episode):
  Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot, \boldsymbol{\theta})$
  Loop for each step of the episode $t = 0, 1, \ldots, T-1$:
    $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$ $\qquad (G_t)$
    $\delta \leftarrow G - \hat{v}(S_t,\mathbf{w})$
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}} \delta \nabla \hat{v}(S_t,\mathbf{w})$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} \gamma^t \delta \nabla \ln \pi(A_t|S_t, \boldsymbol{\theta})$

Figure 6: Pseudocode for the REINFORCE with baseline algorithm.

Algorithm 1 : Double Q-learning (Hasselt et al., 2015)

Initialize primary network Q_θ, target network $Q_{\theta'}$, replay buffer $\mathcal{D}$, $\tau << 1$
for each iteration **do**
  for each environment step **do**
    Observe state s_t and select $a_t \sim \pi(a_t, s_t)$
    Execute a_t and observe next state s_{t+1} and reward $r_t = R(s_t, a_t)$
    Store (s_t, a_t, r_t, s_{t+1}) in replay buffer $\mathcal{D}$
  for each update step **do**
    sample $e_t = (s_t, a_t, r_t, s_{t+1}) \sim \mathcal{D}$
    Compute target Q value:
      $Q^*(s_t, a_t) \approx r_t + \gamma\ Q_\theta(s_{t+1}, argmax_{a'} Q_{\theta'}(s_{t+1}, a'))$
    Perform gradient descent step on $(Q^*(s_t, a_t) - Q_\theta(s_t, a_t))^2$
    Update target network parameters:
      $\theta' \leftarrow \tau * \theta + (1 - \tau) * \theta'$

Figure 7: Pseudocode for the Double Q-learning algorithm.

Algorithm 1 PPO-Clip

Input: initial policy parameters θ_0, initial value function parameters ϕ_0

for $k = 0, 1, 2, \ldots$ **do**

Collect set of trajectories $\mathcal{D}_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment.

Compute rewards-to-go $\hat{R}_t$.

Compute advantage estimates, $\hat{A}_t$ (using any method of advantage estimation) based on the current value function V_{ϕ_k}.

Update the policy by maximizing the PPO-Clip objective:

$$\theta_{k+1} = \arg\max_{\theta} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \min\left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)} A^{\pi_{\theta_k}}(s_t, a_t), \;\; g(\epsilon, A^{\pi_{\theta_k}}(s_t, a_t)) \right),$$

typically via stochastic gradient ascent with Adam.

Fit value function by regression on mean-squared error:

$$\phi_{k+1} = \arg\min_{\phi} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \left(V_\phi(s_t) - \hat{R}_t \right)^2,$$

typically via some gradient descent algorithm.

end for

Figure 8: Pseudocode for the PPO-Clip algorithm.