Autonomous Lunar Lander: Training an AI Agent with Deep Reinforcement Learning

Alexander Cartledge
Machine Learning and AI report
May 2025

Abstract

Lunar Lander is a genre of video games, where the player is tasked with landing a spacecraft in a simulated 2D lunar environment. Achieving a soft, accurate landing is surprisingly challenging, which sparked my curiosity as to whether a computer can be trained to do just that reliably. In my paper, I will attempt to train an algorithm using deep reinforcement learning to play a Lunar Lander game. Due to the simplicity of the game, with elementary graphics and inputs only for thrust and rotation, I believe that it is a perfect candidate for implementing such an algorithm. Simultaneously, due to the complexity of managing slightly unresponsive controls and a randomly generated landscape on which a landing must be achieved, it is still a problem where a hard-coded solution would be difficult. I will also attempt to verify the robustness of the algorithm by modifying the game environment and look for ways in which different performance aspects (accuracy and fuel consumption) can be prioritized by modifying the rewards during training. Success of the algorithm in such a simple environment would also serve as a good indicator that such training methods can be effective when dealing with more complicated problems.

Introduction

Autonomous spacecraft landing is a classical challenge in control and reinforcement learning, in which the agent must contend with continuous state dynamics, specific rewards, like landing in the same space every time, and safety-bound objectives, like not landing too hard, and not landing on one's side or head. This is all done using only the main thruster and attitude controls. OpenAI Gym's LunarLander-v3 environment manages to capture these difficulties in a 2D simulation, where the craft must land softly and upright on the designated pad. Even though the state and action spaces are low dimensional in this case, the combination of issues like the horizontal drift, constant vertical descent, and rotational instability makes rule based, or 'hand-crafted' controllers labor-intensive to tune.

Proximal Policy Optimization (PPO) is a modern policy-gradient algorithm reinforcement learning method that allows for both discrete and continuous action spaces, by maximizing a clipped surrogate objective to ensure stable, incremental policy updates (I don't directly maximize the true expected reward, but instead, stand-in objective that is easier to compute from my data, and I prevent any single training step from changing my policy too much). It is a staple in continuous-control tasks since it balances sample efficiency and robustness to hyperparameter settings. In this project, I leverage Stable-Baselines3's PPO implementation to learn landing policies from raw observations. Then compare its success under different conditions, like adding a harsh fuel penalty, or randomising the starting x coordinate of the Lander relative to the goal. Some of the questions I ask in pursuit of modelling these modifications would be: How robust is PPO to domain shifts compared to a hand-coded baseline? Can reward shaping encourage fuel efficiency without sacrificing landing success? Which hyperparameters most influence stability under variation? Through exploration into these models, I find that baseline PPO is far more rewarding than agentic modelling, and can withstand modifications in the environment with lots of resilience.

Methods

Notebook 2

I first established a learned-policy baseline by training my PPO model on the fixed-pad Lunar Lander v3 environment (found in notebook 2). This is the default version of the simulator, without modifications. There are two things I did that are worth noting. First, I wrote up a custom wrapper (named LunarWrapper) to track fuel usage by measuring how long the lander uses the main thruster, as well as landing error by seeing how far the craft landed from the pad's centre. After that, you can see that I implemented the following. DummyVecEnv which is turning my single simulator into a kind of "vector" of one simulator, a basic requirement for Stable-Baselines3, VecMonitor which watches each episode and outputs basic results into a csv file that can be used for visualizations, and VecNormalize which scales my observations and rewards such that they stay in an appropriate range, clipping any observation values larger than 10 (meaning I guarantee the multi-layer perceptron doesn't see extreme inputs.) I decided to use the default policy in Stable-Baselines3, that being a small multi-layer network with two hidden layers of 64 neurons each. My key parameters are set as:

- n_steps → 2048: I collected 2048 steps of gameplay before each learning update
- batch_size → 64: breaking the n_steps into chunks of 64 for each gradient step
- learning_rate → 0.0003: how large each update step is (constant)
- clip_range → 0.2: to ensure that no single update can move the policy more than 20%, ensuring stability
- ent_coef → 0.005: a kind of bonus incentive to be more exploratory in trying new actions

I let this PPO train for 500,000 timesteps, and I use MetricsCallback to write the usual total reward per episode and the value for the fuel used into TensorBoard so I can watch curves live, then at the end the model is saved so it can be evaluated or played back. With this setup, PPO consistently reaches a reward score above 240, and uses around 100 units of fuel (a model is valued as appropriately successful when reward reaches over 200, and for now, fuel use is arbitrary).

Notebook 1

However, a classical baseline was needed to measure how much more successful this model was against a heuristic agent, issuing commands based on the lander's state instead of relying on Machine Learning. The environment found in notebook 01 shows how this would work, and it gives a "control" against which we can compare my actual ML models against. I opened the standard Lunar Lander v3 environment, wrapped using LunarWrapper to track fuel, and wrote a function rule_action(obs) to follow 3 simple rules:

- If the lander is descending too fast, or is about to crash, fire the main engine to slow the descent.
- If drifting right, fire left thruster, and if drifting left, fire right thruster
- If tilting to the right, fire right thruster, and if tilting left, fire left thruster.

Over 10,000 episodes, the reward was exceptionally low, only succeeding 5% of the time, and using about 5 times more fuel than the PPO model. These results clearly show how fragile the models shaped by hand-made rules are, as the PPO model uses far less fuel, and is far more successful over many episodes, highlighting the advantage of learning a policy automatically.

Continuing from my rule based vs. PPO comparison, I decided to modify the PPO agent to see if it can generalize to more situations.

Notebooks 3-6

First, I introduced a fuel-use penalty in notebook 3. There would be a per-step penalty for each main engine fire, such that each time the main thruster is fired (per step), the agent lost an extra 0.3 points. The

only changes in other variables that I made from notebook 2 were that I trained the model in notebook 3 for 1,000,000 steps instead of 500,000, a learning rate that decayed from 5×10^{-4} to 1×10^{-5}, and a clip range of 0.15 (These changes were made in an attempt to stabilize the model). However, peculiarly, the mean reward of the model seemed to only reach a reward of around 100, and the mean fuel used reached around 180, both values being less preferred to the baseline model in notebook 2. Some reasons for why the model might have performed worse could be that: The fuel penalty was too small compared to the bonus that the lander would receive in reward, so landing using more fuel was still preferred over not landing using less fuel. Also, I theorized that the extra penalty increased variance in the rewards, meaning advantage estimates were less reliable, so PPO updates became unstable, and learning would slow or often even collapse. In short, my naive fuel penalty actually introduced more difficulty for the model rather than incentive for less fuel usage, and a more effective tactic might have been introducing shaping variables that would more carefully align with the agent's objectives.

Afterwards, in notebook 4, I tested whether a PPO model would be able to handle the pad being shifted each episode, without losing on performance. To test this, I wrote an entirely new wrapper, named RandomPadWrapper, in which the helipad would be moved to the left or right by a random offset (not too much distance such that the lander would still be able to properly land on flat ground), then I shifted the lander's observed x-coordinate, such that from the agent's point of reference, the pad always appears at x = 0, like in the baseline model. I found that, given enough time to train, the model can eventually reach a reward of 200, just reaching the threshold for a successful model, showing that the model can gradually adapt and recover performance lost from the gap created by the domain shift.

In notebook 5, I decided to do something similar than in notebook 4, but now, to build upon the pad-randomization, I tested how well a PPO model handles the actual lander starting in different horizontal positions. To do this, a new wrapper named RandomStartWrapper basically mimics the wrapper from notebook 4, but now applies it to the lander instead of the pad. Interestingly, even when tested over 1 million timesteps, instead of 1.5 million like notebook 4, it still reaches a performance that is, on average, better than notebook for, reaching a reward of 220 and using around 150 units of fuel. This demonstrates that there is a meaningful distinction in whether the pad is shifted by some amount, or if the lander itself is. One reason as to why this might have been the case was because of the fact that, in rare occasions, the pad would be shifted in such a way that part of it would be on sloped terrain, thus reducing the amount of space that can be used by the lander, and making the challenge of landing properly even more difficult, a problem that the model in notebook 5 didn't have to contend with.

Lastly, in notebook 6, my model retained the x-offset of the lander in notebook 5, but also changed the entropy coefficient of the PPO model. The entropy coefficient is a kind of "curiosity dial" for the agent. It measures how random the policy's choices are, high entropy meaning more randomness, and low entropy meaning less. The coefficient is simply the weight on that randomness bonus in the loss function. With low entropy, a model is less exploratory, and therefore is likely not to find the best minimum, but is very consistent (training the same model over and over will produce consistent results and a smoother learning curve). A model with higher entropy is more "courageous" in its exploration, but often it tries methods that are too weird for the model to be successful, so a sweet spot must be struck between entropy that is too high and too low. In notebook 06, I loop over multiple entropy coefficients to find which give the best results, illustrated in *figure 3*. Although the PPO's recommended coefficient is 0.005, I found that a coefficient of 0.01 yielded better results more frequently, as shown by the consistent reward of 250 and mean fuel usage of 95.

Results

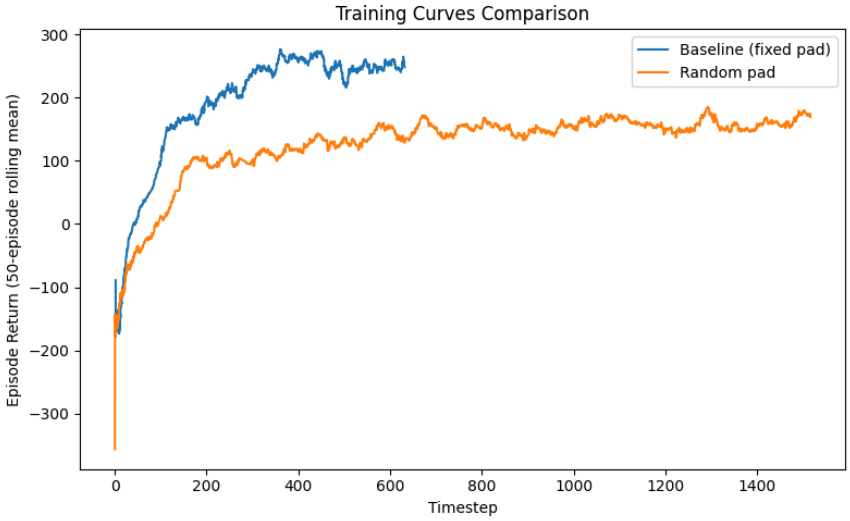

Figure 1 plots how the averaged score (over 50 episodes) evolves as two versions of PPO learn to land the rocket. The blue line represents the normal agent found in notebook 2. It rises to a reward of over 200 in 300K steps and hangs around 250, showing how well it manages to play the game. The orange line represents the agent for which the pad changes position, in notebook 4. We can see how it follows a similar trend to the baseline model, however it consistently remains below it in performance, even when given 3 times more steps to train.

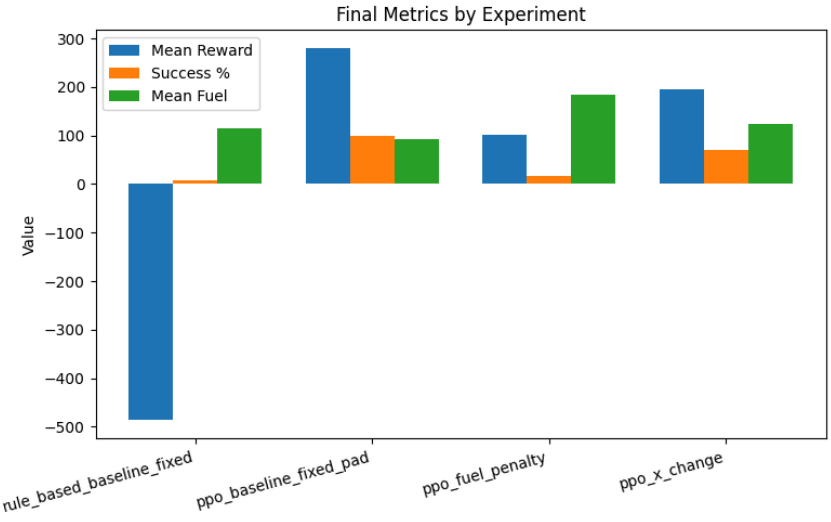

Figure 2 lines up the performance of the four models from notebooks 1-4, measured by average score, percentage of success, and fuel used. The rule based baseline shows by far the worst performance, only landing 5% of the time. The PPO fixed pad baseline model is far better, showing successful results 100% of the time, and even uses less fuel than the rule based model. The fuel penalty model has a far lower performance than the PPO baseline, only managing to land well about 10-15% of the time, while using even more fuel than each of the other models. Lastly, the x_change model where the pad changes position, but it still performs somewhat well, managing to land about 80% of the time and using less fuel than the fuel penalty model, but still more than the baseline model.

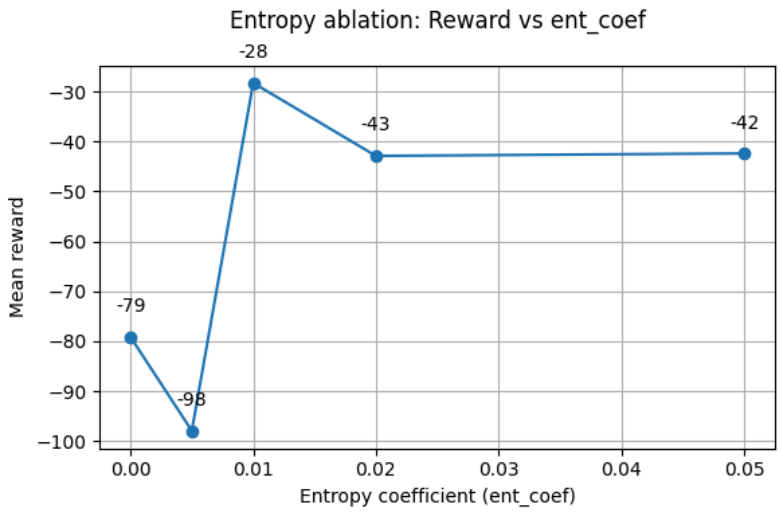

Figure 3 plots how the average score may depend heavily on the randomness bonus coming from the entropy_coef. We find that with no bonus, the agent barely learns, with a bonus of 0.005, the agent may actually perform worse and get stuck on bad results more often, for 0.01, the score immediately jumps up dramatically due to it having the freedom to explore more strategies, and the other two bonuses cause the rewards to decrease again due to too much randomness. Thus there's a kind of Goldilocks zone found at 0.01.

Conclusion, Discussion and Future work

To close off, despite PPO's strong performance on the default Lunar Lander task (reaching heights in reward up to 250 and landing successfully 100% of the time over 10,000 episodes), my models revealed a few challenges. Firstly, naive per-step fuel penalties hurt the model far more than it helped, and introduced destabilizing factors into the model's learning. Also, models trained on a fixed pad starting position can fail far more when the pad shifts, even by a very small amount, but this isn't the case if the lander itself changes position (possibly due to the pad's area tightening unintentionally). Lastly, changing shaping parameters, especially the entropy coefficient may yield greater reward at the cost of slightly less per-run consistency, but are still worth varying where possible.

For future work, training for more timesteps (3-5 million) may close remaining gaps in rewards and allow the models to achieve perfect success every time. As well as this, implementing a small per-step proximity bonus as well as an end-of-episode fuel penalty may yield greater results than what was seen in notebook 3 while preserving the original intent of fuel conservation. Lastly, varying the gravity and perhaps introducing an element of wind would allow the model to generalize more accurately, ensuring that it becomes truly robust.

References

PPO algorithm: Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). Proximal Policy Optimization Algorithms. arXiv:1707.06347.

Stable-Baselines3: Raffin, A., Hill, A., Gleave, A., Kanervisto, A., Ernestus, M., & Dormann, N. (2021). Stable-Baselines3: Reliable Reinforcement Learning Implementations. Journal of Machine Learning Research, 22(268), 1–8.

OpenAI Gym LunarLander: Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., & Zaremba, W. (2016). OpenAI Gym. arXiv:1606.01540.