Houston, We Have a Solution: The Science of Not Crashing with Reinforcement Learning

Zachary W. Burgess

Abstract

Imagine trying to pilot and safely land an airplane, except none of your controls are labeled. Furthermore, you are not even sure if you are piloting an airplane. Similarly, you are not quite sure if the goal is to land, much less if you want to land safely. All you know is that when you use one of the controls, someone rates your action on a scale from 1 to 10. This would be nearly impossible to do! However, the subset of Machine Learning known as Reinforcement Learning does exactly that. This paper follows the inner workings of solving such a problem, from picking out an algorithm to the grueling task of deciding on hyper-parameters. Every step serves to bring the computer from an uninformed pilot to an expert astronaut.

1 The Lunar Lander

To begin, a classic Reinforcement Learning problem is posed, that of the Continuous Lunar Lander. In essence, the goal of this problem is to successfully have an agent navigate and land a Lunar Lander. The agent will be responsible for the velocity, angular speed, and a variety of other things surrounding the Lunar Lander, all with the goal of dropping the lunar landing safely between two marked flags as pictured below:

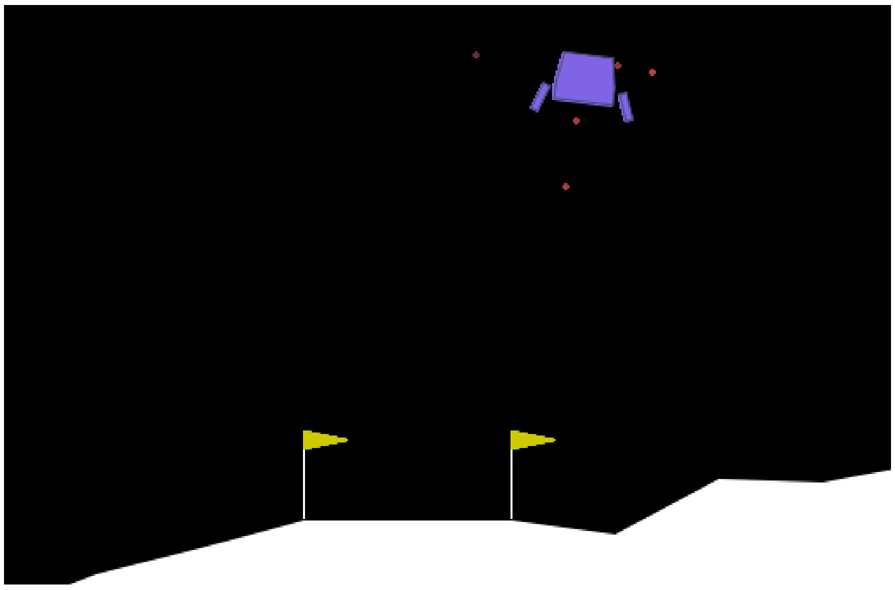

Figure 1: The Lunar Lander Environment

1.1 States

The state space of the Lunar Lander is fairly complex, requiring 8 separate attributes. These attributes give the agent a full understanding of the current "state" of the Lunar Lander. The goal is that the agent will look at these attributes and know precisely what it needs to do to move towards a more favorable circumstance. The attributes are as follows:

- The horizontal position position of the Lunar Lander.
- The vertical position of the Lunar Lander.
- The horizontal speed position of the Lunar Lander.
- The vertical speed position of the Lunar Lander.
- The angle of the Lunar Lander.
- The angular speed of the Lunar Lander.
- Whether the left leg is touching the ground.
- Whether the right leg is touching the ground.

With this space space in mind, it can then be said that a state is "terminal" or rather, the given episode has concluded and the environment is to be reset, in three cases:

- The Lunar Lander's body comes into contact with the moon (crashes).
- The Lunar Lander leaves the viewable space.
- The Lunar Lander is "asleep" or no longer moving(Project, 2024).

1.2 Actions

These states inform the actions of the agent by giving context of what is currently happening with the Lunar Lander. That being said, the agent at any given time-step can take an infinite number of actions. This is due to the "continuous" nature of the environment; rather than taking discrete actions like moving up, down, left or right, the agent controls two "sliders".

The first of these controls is the main engine throttle (the red dots coming from the bottom of the Lunar Lander in Figure 1). This controls the Lunar Lander's speed of descent. The main engine throttle's control is represented as a slider from -1 to 1, any number in the negative is considered "off" and any number from 0 to 1 adjusts the main engine from 50% to 100% power.

The second control is the side boosters (the red dots coming from the left and right of the Lunar Lander in Figure 1). This controls the Lunar Lander's angle and is the primary method of positional control, moving the Lunar Lander from left to right. The side boosters are controlled similarly to the main engine throttle, with a slider from -1 to 1. Any number from -.5 to .5 is considered "off" while -1 to -.5 adjusts the left booster from 100% to %50 power and 0.5 to 1 adjusts the right booster from 50% to 100% power.

It should be noted that neither the main engine throttle, nor either booster can operate below 50% power, which is why they shut off should they end up below that threshold.

1.3 Rewards

Finally, to completely understand the Lunar Lander environment, the rewards must be fully understood. These rewards are the incentive the agent is given to take certain actions, guiding it to the desired outcome. These rewards are constructed as follows:

- The agent is given a "potential" reward for moving closer to the designated zone, this reward is said to be "potential" because the agent loses the reward by moving further away from the zone. Eventually securing the positive or negative reward upon episode termination.

- The agent received a negative reward of -0.3 for each usage of the main engine throttle.
- The agent is given a positive or negative 100 depending on whether a given episode terminated inside or outside the designated zone.

The Lunar Lander problem is said to be "solved" by a model when the model is able to average above 200 reward across 100 episodes.

2 The Pilot

With the problem well defined, we turn our attention to the solution. In this case, the "solution" to the Lunar Lander is a pilot and that pilot is the agent. However, with the implementation of an agent comes a choice: the choice of algorithm for the agent to represent. Within the context of the Continuous Lunar Lander problem, there are a few things to consider when selecting an algorithm: the type of environment (discrete vs. continuous), the computational overhead, the environment's rewards, the flexibility of hyper-parameters, and more.

2.1 Possible Algorithms

Starting with the first, the type of environment, the Lunar Lander is "continuous" which means we have sliders of actions. This limits the top agent algorithms to a mere handful: TD3, DDPG, and SAC. All of the listed algorithms come with their own list of pros and cons. DDPG is efficient in a continuous space, but it can be unstable without proper tuning of hyper parameters. SAC on the other hand is extremely stable, but heavy on computational overhead. Lastly, TD3 is fairly stable, but more complex due to it's numerous neural networks.

For the purposes of this paper, flexibility of hyper-parameters and computational overhead are huge factors in the selection. This means that while TD3 and SAC are both viable alternatives to DDPG, perhaps even performing better in some case, DDPG will be piloting the Lunar Lander. DDPG offers a balance of stability, computational efficiency and sample reuse that are critical for this problem.

2.2 DDPG

The DDPG algorithm centers around the communication between two objects, that being the "actor" and the "critic". The actor looks at the given state of the environment and selects an action that it believes the critic will like. The critic then sees the actor's action and gives it a score. The hope is that the actor will learn to take actions that the critic rates highly and that the critic will learn to rate better actions more highly.

2.3 The Algorithm

The following section provides a high level overview for the DDPG algorithm in the form of pseudocode. The process of this algorithm is important to understand as the back half of this paper will discuss various tweaks to the "settings" of the algorithm in the form of hyper-parameters. The DDPG algorithm functions as follows:

```
# initialize environment
# initialize actor
# initialize critic
# initialize target_actor
# initialize target_critic
#
# for some amount of episodes
### start an episode
### until this episode is completed
##### actor selects an action
```

```
##### take that action in the environment, store: reward, next_state, termination
##### store the following "snapshot" into the replay buffer: current_state, action,
    reward, next_state, termination
##### if there are enough snapshots in the replay buffer
####### sample a batch_size amount of random snapshots from the replay buffer to "study"
####### have the target_actor map the next_states to next_actions
####### have the target_critic map the (next_state, next_action) pairs to q_values
####### nudge the output q_values towards their proper value using the bellman equations
####### have the critic map the (state, action) pairs to q-values
####### calculate the difference between the critic and the target critic
####### update the critic by nudging it closer to the target critic
####### calculate the difference between the actor and the critic
####### update the actor by nudging it closer to the critic
####### update the target_critic according to the critic, but slower
####### update the target_actor according to the actor, but slower
```

The goal of this algorithm is for the critic to "chase" it's target counterpart, constantly overshooting above and below until eventually it converges to the proper Q-value.

2.4 Hyper-Parameters

The DDPG algorithm employs a variety of hyper-parameters, these are parameters the model uses to fine tune various portions of it's learning. For the purposes of this implementation, the hyper-parameters used are: the batch size, the standard update rate, the slower targeted update rate, and the exploration noise. These hyper-parameters are initialized as follows:

```
batch_size = 64 # the amount of snapshots to be run through the critic at once
update_rate = 0.001 # the standard update rate of the q-values
target_update_rate = update_rate / 2 # the update rate rate of the target network
exploration_noise = 0.1 # the rate at which random actions occur
```

Classically, update rate and batch size have a positively correlated relationship, that is batch sizes require higher update rates for better performance(Anantha, 2024). Meaning that for large batch sizes, the update rate should be increased as well. This assertion will be tested in later sections alongside numerous other hyper parameter combinations.

Exploration noise is a wildcard, the effect it has on the other hyper-parameters chosen have many sorts of correlations. This makes it a suitably interesting choice for testing in upcoming sections, where all possible combinations of exploration noise with update rate and batch size will be thoroughly tested.

2.5 The Control

Before moving into the experimental portion, it is important to first establish a quality baseline. Using the hyper-parameters defined in the section above, the DDPG algorithm studied 1,000 episodes. This number was chosen somewhat arbitrarily, as throughout testing it properly demonstrated not only the learning curve, but the steady level of the model once it had successfully learned. The results of the baseline test are as follows:

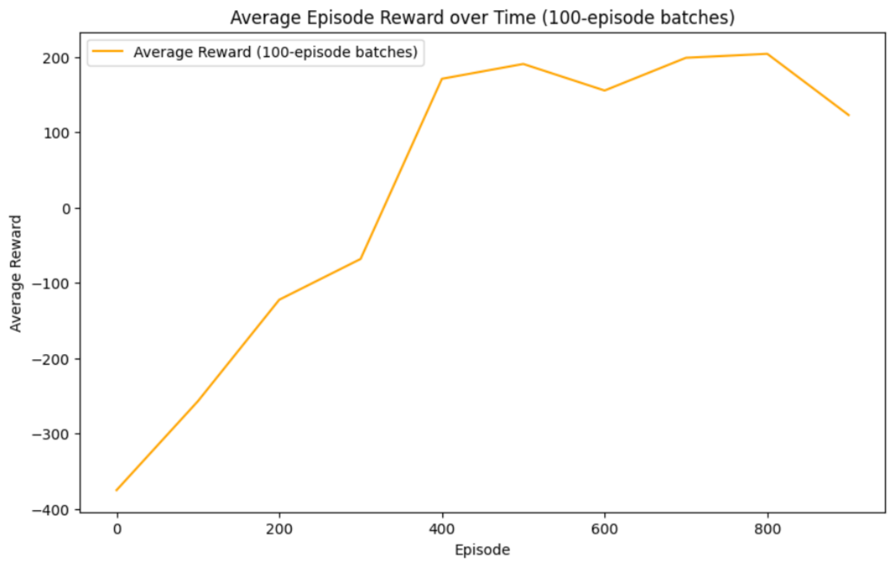

Figure 2: Control Test

Figure 2 demonstrates the outcome of the model on it's first run. The Y-axis of the graph is the numerical reward the model received at the end of each episode. This graph demonstrates that the agent come extremely close and may have (if you select the proper episode period) successfully "solved" the Lunar Lander problem around the 500 episode, averaging a numerical reward above 200 across 100 episodes. The table (with episodes abbreviated to "eps") below shows a more numerical representation of the graph, with the testing episodes split into sets of 200 and their average reward taken, as well as an additional 100 "test episodes" using the fully trained agent:

eps 0-200	eps 200-400	eps 400-600	eps 600-800	eps 800-1000	test eps
-315.97	-95.12	181.05	177.36	163.73	145.81

While the graph shows strength above the 200 reward mark, the resulting table and test episodes show this baseline model unable to retain that strength. There is certainly some work to be done as the agent seemed to falter towards the end, hopefully some tuning will see the agent reach that "solution" level of performance.

3 Simulation

With all of the background out of the way, the next step is to simulate the problem. This is with the purpose of achieving the best outcome from the agent. In each simulation, the agent will step through 1,000 episodes, just like it did in the control. The primary method of evaluation throughout all simulations will be the average episode reward, also like in the control.

3.1 Hyper-Parameter Tuning

In section 3.1, the hyper-parameters for the DDPG agent were established, however in 3.2 that model, along with it's combination of hyper-parameters, fell short. The purpose of this test is to explore a variety of combinations of hyper-parameters with the goal of finding the "optimal" combination. The optimal combination here being the one that yields the highest average reward over a span of 100 episodes. The hyper-parameter ranges to be tested are as follows:

```
batch_sizes = [16, 32, 64, 128]
update_rates = [0.0001, 0.0005, 0.001, 0.005]
exploration_noises = [0.05, 0.1, 0.2, 0.3]
```

This test will run 64 separate DDPG models through 1,000 episodes, recording their performance during the training phase. Each model will be a unique combination of batch size, update rate (with the target rate being proportionally half), and exploration noise. Below are two tables, the first having the 10 highest performers and the second having the lowest 10 performers with the mediocre 44 omitted.

Batch Size	Update Rate	Exploration Noise	eps 0-333	eps 333-666	eps 666-1000
16	0.0005	0.3	-156.79	223.07	263.62
64	0.005	0.05	-48.15	215.93	243.01
32	0.001	0.3	-35.34	207.56	242.68
16	0.001	0.3	3.08	224.37	242.34
32	0.005	0.1	13.24	162.03	241.47
32	0.0005	0.3	-163.88	198.38	239.60
16	0.001	0.2	-81.01	222.61	234.68
16	0.001	0.05	-271.11	185.53	233.53
16	0.0005	0.1	-239.29	207.63	232.85
32	0.001	0.2	-172.01	139.31	232.83

Batch Size	Update Rate	Exploration Noise	eps 0-333	eps 333-666	eps 666-1000
16	0.005	0.2	-556.97	-529.70	-529.80
64	0.0005	0.2	-554.13	-534.74	-544.24
128	0.005	0.2	-540.17	-558.37	-553.35
64	0.005	0.1	-559.09	-554.49	-560.36
128	0.005	0.05	-554.46	-579.37	-570.67
128	0.005	0.3	-1009.37	-1052.06	-991.84
64	0.005	0.2	-1065.63	-1050.72	-1058.28
16	0.005	0.1	-1080.61	-1085.72	-1084.51
16	0.005	0.05	-1051.47	-1069.40	-1087.48
64	0.001	0.1	-224.43	-450.30	-2490.20

For convenience sake, the tests are referred to by the combination of their indices. For example, test 000 would be the hyper-parameter combination of the first item in each list (batch size of 16, update rate of 0.0001, and exploration noise of 0.05). Test 123 would then be batch size of 32, update rate of 0.001, and exploration noise of 0.3. Given this definition, the above table demonstrates that tests 013, 230, 123, 023, and 131 are the top performers.

3.2 Clash of The Agents

With the tuning out of the way, the 5 top performers will not be pit against each other in a "testing" phase. The agents will no longer learn anything, rather they will act on their learned behaviors over the course of 100 episodes. Below is a graph showing the result of 100 episodes of testing these five models:

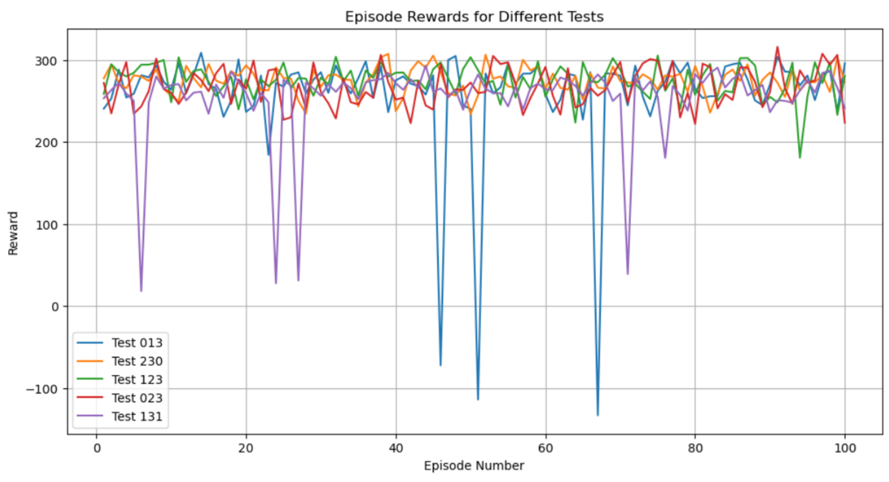

Figure 3: Hyper-Parameter Tuning Results

The above graph is useful for demonstrating that the agents are consistently maintaining what is considered a "successful" average. However, some of the agents (131, 013 and to a lesser extend, 123) have dangerous outliers within this small sample size, which makes them less desirable. Below is one last table summarizing these results as a single average over 100 episodes:

013	230	123	023	131
259.52	274.99	275.17	267.3	254.53

Using this table and graph a few things can be said about the agents. All of them perform exceptionally well, however 131 and 013 lose to the other agents in terms of consistency. From there, out of the remaining three, 023 performs marginally worse. This leaves the title of the "best" agent for either test 230 or 123. While 123 overtakes 230 in this small test by 0.18 more average reward, 123 also seems, according to the graph, to have a bit worse consistency than 230. This makes 230 the winner of the clash of the agents.

3.3 Hyper-Parameter Tuning: Continued

The results from the the above two sections led to an interesting outcome. Recall the way we defined the hyper-parameter tests, each number corresponds to an index in a list of four options. Test 230 then is the 3rd batch size, the 4th update rate, and the 1st exploration noise. Said another way, test 230 utilizes the largest update rate option and the smallest exploration noise option.

Therefore, more tests are necessitated, as the logical conclusion is that larger update rates and smaller exploration noises lead to a better trained agent. Demonstrated by the below code, tests containing larger update rates and smaller exploration noises are run to see if that trend continues. Every combination of the following hyper-parameters are tested:

```
batch_sizes = [16, 32, 64]
update_rates = [0.01, 0.05, 0.1]
exploration_noises = [0.001, 0.005, 0.01]
```

The following table displays the 10 highest performers from this new batch of tests:

Batch Size	Update Rate	Exploration Noise	eps 0-333	eps 333-666	eps 666-1000
16	0.10	0.010	-567.95	-592.33	-569.03
64	0.01	0.001	-573.14	-589.33	-575.37
32	0.01	0.005	-598.14	-574.52	-577.47
64	0.01	0.005	-576.33	-563.49	-577.59
64	0.10	0.005	-575.80	-578.76	-583.14
64	0.10	0.010	-582.70	-577.73	-587.45
16	0.05	0.001	-574.09	-558.20	-599.08
16	0.10	0.001	-1104.62	-1099.33	-1064.68
16	0.05	0.005	-1131.28	-1129.41	-1077.65
16	0.05	0.010	-1099.77	-1064.96	-1083.49

Unfortunately, the trend did not continue as hoped and these agents crashed and burned alongside their Lunar Landers. This was not entirely unexpected however, as the previous hyper-parameter tuning section did seem to demonstrate that a healthy balance between the batch size, exploration noise, and update rate was best for the agent's training.

4 Conclusion

It is evident from the above tests that the choice in hyper-parameters plays a crucial role in the development of an agent, particularly within the DDPG algorithm. Tests 123 and 230 demonstrate the importance of balance between the three tested hyper-parameters, and seem to suggest that aggressive update rates help the agent "learn" more quickly.

The poor performance of the second round of tests truly underscores the importance of this balance. While it did seem that intuitively higher update rates should encourage faster learning, it seems like taking that to the extreme destabilized the agent's ability to learn. Future experiments regarding "adaptive" learning rates and exploration noises (hyper-parameters that change mid training session) could be a fascinating test of the balance discussed here. However, that sort of experimentation is well outside the scope of this paper.

With a comprehensive analysis of the hyper-parameters controlling the DDPG algorithm, we identified several combinations which successfully managed to solve the continuous lunar landing problem. Overall, the journey from an uninformed pilot to an astronaut serves as a compelling demonstration of the things possible using reinforcement learning.

References

Sai Teja Anantha. How should the learning rate change as the batch size changes? *GeeksforGeeks*, 2024. URL https://www.geeksforgeeks.org/how-should-the-learning-rate-change-as-the-batch-size-changes/. Accessed: 2024-10-06.

Box2D Project. *Box2D Documentation: Simulation*, 2024. URL https://box2d.org/documentation/md_simulation.html. Accessed: 2024-10-06.