Solving the LunarLanderContinuous-v2

Royi Rassin, Amit Amar

Submitted as final project report for the RL course, BIU, 2021

1 Introduction

Reinforcement Learning is one of the most promising fields in Computer Science, advancements will ultimately lead to autonomous cars, reveal strategies in games which are not natural to humans, and maybe, if we are lucky, to house-cleaning robots! In this project, we solve OpenAI-Gym's continuous version of Lunar-Lander-V2, compare the performance of a variety of Deep Q Networks, and provide a brief overview on the related topics on which our solution is based on. The goal is to land the spaceship between the two flags smoothly and we say we solved the environment if the model reaches an average reward of 200 over 100 episodes. The spaceship has three throttles; one points downward, and the other two point in the left and right direction.

1.1 Related Works

While working on this project we were inspired by other researchers who worked on similar problems. For instance, a researcher presented his work on DDQN in a discrete action-space [anh20], another researcher gave an in-depth explanation on D3QN [Bal20a] and cleared misunderstandings we had, others showed us novel ways to solve the environment and taught us about DDPG [Ver19]. Finally, we learned about reward-clipping [Rie13] and prioritized-experience-replay [Bal20b].

2 Solution

2.1 General approach

We first tackled the problem by implementing a DDQN and optimizing its parameters until it solved the environment. Specifically, we adopted the environment from continuous to discrete by modifying an existing DDQN [anh20]. The environment was quantized by mapping the network's predictions to a closed-set of actions. Originally, The main and secondary engines' are real-values in $[-1, 1]$. The main engine is off if it is in $[-1, 0)$ and gradually increases in $[0, 1]$, starting from 50% throttle. The secondary engine is firing to the left if it is in $[-1, -0.5]$, turned off if in $(-0.5, 0.5)$, and to the right between $[0.5, 1]$. Quantizing these values means we select two finite sets M and S respectively to represent the continuous action space. Because each action takes into consideration both engines, the action-space is $2D$ with MxS possible values. We also compared the performance of our DDQN to a D3QN and a DDPG. Surprisingly, DDQN, and D3QN significantly outperformed DDPG. Experimenting with a DQN was unnecessary since we know DDQN is an improved version of it. In particular, DQN suffers from over-estimating the Q-values and is encouraged to choose actions that lead to lower rewards. DDQN solves it by using the target-network's weights to compute the Q-values, but it selects the max value by using the policy-network's weights.

Define $a^{\max}(s';\theta) = \arg\max_{a'} Q(s', a'; \theta)$

$$y_j = \begin{cases} r & \text{if } s' \text{ is terminal} \\ r + \gamma Q(s', a^{\max}(s';\theta); \theta^-), & \text{otherwise.} \end{cases}$$

Do a gradient descent step with loss $\|y_j - Q(s, a; \theta)\|^2$

Replace target parameters $\theta^- \leftarrow \theta$ every N^- steps

Figure 1: Take the max based on Q-values, calculated using θ and evaluate the Q-values of a^{max} based on θ^-

In terms of fine-tuning and optimizations, we experimented with reward-clipping, prioritized experience replay, different hyper-parameters, different action-spaces and gradient clipping.

2.2 Design

Most of the code is written in Keras, we tested it in Google Colab and our local machines, and the all-time best to beat the environment is 287 episodes (with noise in effect). As seen in figure 2, our code is split to two main components: a network agent (D3QN/DDQN) and a train-script. The networks contain an ExperienceReplay class; can be prioritized or regular, and except their architecture, they are quite similar. Further generalization in the code is possible, nevertheless, the project's emphasis is on solving the environment and we did not find it necessary. Since we did not conduct in-depth research on DDPG and only tested its potential, it is not included in our design. The biggest challenge was quantizing the network: we had to invest a significant portion of our time towards understanding the inner-workings of the network, deciding where it makes sense to implement it, the finite values we want to attach to each engine, and the number of values. Our intuition was that the less combinations we have (MxS), the less the network will have to learn and thus, it will use the actions it does have efficiently. So, we ended up with $M = 3$ and $S = 3$; the main engine has an engine-off, low-throttle and high-throttle values, while the

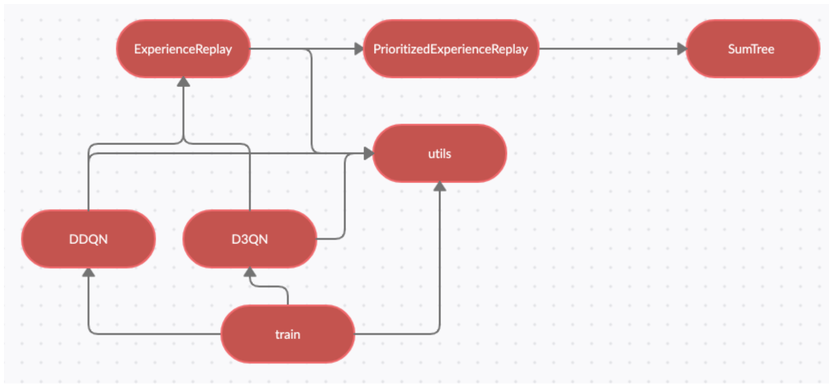

Figure 2: The design of our code

secondary has low-throttle-left, engine-off, low-throttle-right. We first wanted to solve the environment with any network, and only then experiment with the rest, for that reason, we focused on optimizing DDQN's hyper-parameters and other settings, and only then, comparing our results with other types of networks. Implementing D3QN was a little harder with Keras; in the beginning it was not clear how to create two flows from one network, combining them back, and back-propagating with the same loss, nonetheless, the work proved to be beneficial and as of now, D3QN is providing the best results. We also did some tests with DDPG, but since that part of the project was optional, we put less time in optimizing it. Yet, the literature indicates it holds better potential in solving these types of environments than DDQN and D3QN.

3 Experimental results

As mentioned previously, we experimented with three networks: DDQN, D3QN, and DDPG. D3QN net the best overall results. We tinkered in any way we could imagine: batch-normalization, Huber-loss vs. MSE, different number of layers, different number of actions, batch sizes, small and large number of nodes per layer, reward-clipping, prioritized experience replay, different γ values. Our base settings are as follows: DDQN, Adam(LR=0.0001), Batch-size=64, γ=0.99, ϵ-decay=0.995, min-ϵ=0.1, 2-layers: 256, 128, loss=MSE, update target-network per episode, no limit on steps per episode. During our first experiments, the network did not converge often, and we noticed it suffers from catastrophic-forgetting. With that in mind, we increased ϵ-decay to encourage exploration further into the learning-phase, however, it only slowed down the learning. We also tested prioritized experience replay, which did not change the results by much, we hypothesize that it is because it introduces bias towards non-zero rewards, nonetheless, in our environment most rewards are already non-zero, and thus, the effect is not significant. Eventually, hard-updating the target-network every $C = 100$ steps (soft-updating with τ=0.1 and 0.01 was tested too) and limiting each episode to a maximum of 2000 steps made a big difference. First, updating the target-network more often immediately made a difference. Second, limiting the steps creates a sense of urgency in the model, and avoids times where the spaceship endlessly hovers around the flags only to crash somewhere else, and in the process accumulates a very negative reward. As a result, the network finally solved the environment in around 700 episodes. At this point, we engaged in general tinkering; Huber-loss makes sense for Q-learning because of how it deals with edge values, and despite that, MSE still outperformed, lowering γ to 0.6 or even 0.9 resulted in a network that quickly forgets what it learned, even if during the first episodes the average-reward was much higher. In theory, reward-clipping seemed promising, and by clipping the reward to $\sim$ -1 for negative rewards, and $\sim$ 1 for positive rewards, we normalize the reward, and as a result, training the agent in one game may transfer well to another game [Rie13]. Despite that this use case does not apply to our project, we decided to test it. On its face, the reward is immediately higher with clipping, starting from $\sim$ -70 instead of $\sim$ -250, however, the model soon reaches a plateau, and never solves the environment. One explanation is that by clipping the rewards in this manner, the model cannot tell between rewards of different magnitude. So, a catastrophic error, like endlessly hovering over a random location, yields an equivalent reward to a rough landing between the two flags. Adding batch-normalization, different batch-sizes and exchanging Adam with RMSProp, was not helpful. The learning rate we had seemed to work well (0.0001), 0.001, 0.0003 and 0.01 were tested too. Adding a third layer with a descending number of nodes per layer (256, 256, 64 respectively), setting ϵ-decay=0.99, and min-ϵ=0.01 improved performance, coupled with a modest action-space; more finite values took the network longer time to master, one can argue that for bigger tasks it may be worthwhile since the actions will be more refined. After reaching a respectable number of episodes with DDQN, we added D3QN and DDPG. D3QN immediately outperformed DDQN with similar parameters, while DDPG yielded average results, however, it was clear that with some tuning DDPG can outperform the other networks. Despite the substantial improvement that was made, the network is not consistent 100% of the time (especially D3QN), and the results do vary from experiment to experiment. It is most probable to assume that the random weights it is initialized with, coupled with the random actions which are taken at the beginning make a significant difference on the rest of the training phase. The best results are recorded in the table below.

Table 1

Network	Episodes
DDQN	420
D3QN	287
DDPG	1178

Please keep in mind, these are the best overall results.

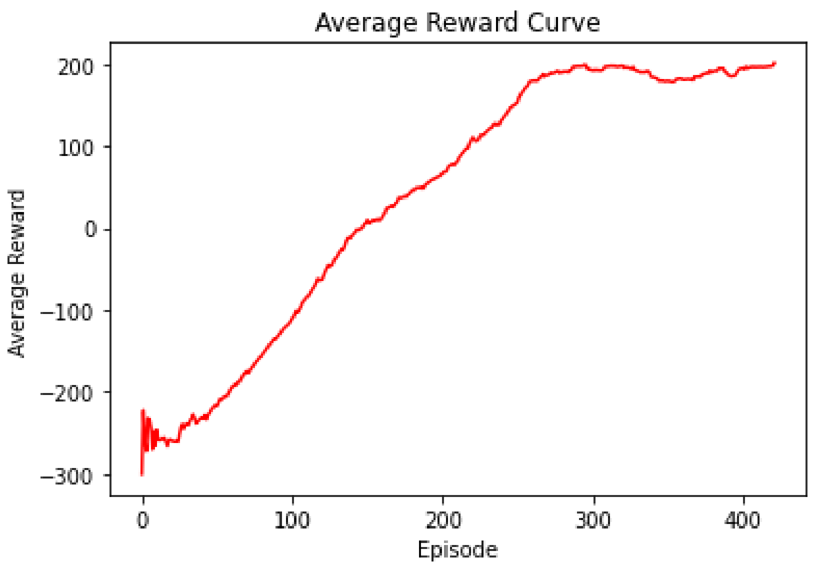

Figure 3: DDQN

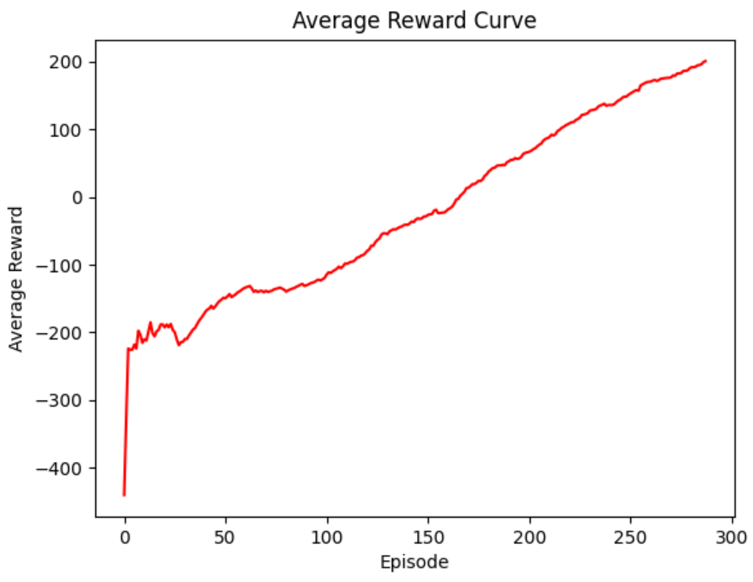

Figure 4: D3QN

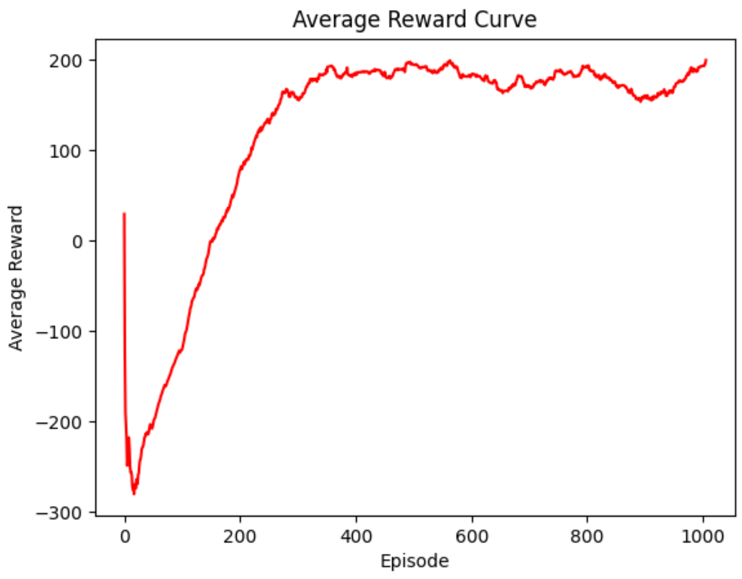

Figure 5: DDPG

4 Discussion

Ultimately, it does not only come down to what network one applies, but also to the training process. DDPG is the network that holds the most potential in this type of environment, yet, since we invested our time in fine-tuning D3QN and DDQN, they outperformed it. That is, the "biggest and baddest" network will not always do better than a fine-tuned less-optimal counterpart. Nevertheless, when all else is equal, it probably will; D3QN is more modern than DDQN, and since we attempted to fine-tune both, D3QN proved to be better as expected.

5 Code

All of our code with the relevant videos can be found here.

References

[Rie13] Volodymyr Mnih Koray Kavukcuoglu David Silver Alex Graves Ioannis Antonoglou Daan Wierstra Martin Riedmiller. “Playing Atari with Deep Reinforcement Learning”. In: *NIPS Deep Learning Workshop* (2013).

[Ver19] Shiva Verma. “Train Your Lunar-Lander | Reinforcement Learning | OpenAIGYM”. In: *Towards Data Science* (2019).

[anh20] anh-nn01. "Lunar-Lander-Double-Deep-Q-Networks". In: *GitHub* (2020).

[Bal20a] Rokas Balsys. "Introduction to Dueling Double Deep Q Network (D3QN)". In: *Medium* (2020).

[Bal20b] Rokas Balsys. "Reinforcement Learning (D3QN) Agent with Prioritized Experience Replay memory". In: *Medium* (2020).