Reinforcement Learning: Comparison of RL Algorithms

Gonzalo Recio (gonzalo.recio@fib.upc.edu) — June 2020

Abstract: *Several classical reinforcement learning algorithms are reviewed and put into practice within an OpenAI environment called* `LunarLander-v2`*. The convergence and characteristics of the algorithms are analyzed and discussed.*

1 Introduction

Autonomous drones and rockets have a spectrum of applications from urban transportation and reusable rockets to drug delivery and fugitive methane leak detection. Effective control systems are needed, especially for challenging tasks of landing and takeoff. This coursework applies AI to drone maneuvering using `LunarLander-v2` simulation environment from OpenAI Gym [1]. The goal is to land the vehicle on the target fast, safely, and efficiently. We are going to use reinforcement learning, and compare the behavior of several RL algorithms, to train an agent to achieve this goal.

2 Environment

`LunarLander-v2` environment consists in a continuous state representation of a lander in a 2-dimensional space with physic laws (gravity, velocity, acceleration, etc). The lunar lander has four discrete actions: do nothing, fire left orientation engine, fire main engine, fire right orientation engine.

The state consists of 8 different values: two first values for defining the coordinate (x, y) position of the lander, ant the rest for defining the horizontal speed, the vertical speed, the angle, the angular speed. The last 2 values are for each leg of the lander, indicating if the lander legs are in contact with the surface (binary valued). All state variables are continuous except for the last 2 ones.

In each episode, the landing pad (the target position to land on) is always at coordinates $(0, 0)$. The lander appears at the top with a random velocity, direction, angular velocity and... . Hence, one initial task is to stabilize the lander. The reward for moving from the top of the screen to landing pad and with zero speed is about 100...140 points. If lander moves away from landing pad it loses reward back. An episode finishes if the lander crashes or comes to rest, receiving additional -100 or $+100$ points, respectively. Each leg having ground contact is $+10$ of reward. Firing main engine is -0.3 points each frame. Solving the scenario is 200 points. Landing outside landing pad is possible and fuel is infinite, so an agent can learn to fly and then land on its first attempt.

3 Algorithms

We are going to test several classic reinforcement learning algorithms. In this case, we chose Q-Learning to estimate a baseline, Deep Q-Network (DQN), Double Deep Q-Network (DDQN) and Advantage Actor Critic (A2C).

3.1 Q-Learning

Q-learning consists in estimation the action value $Q(s, a)$. It can be seen as a memory table $Q[s, a]$ to store Q-values for all possible combinations of s and a. By initializing $Q_0(s, a)$ randomly, each time we sample an action a and take the reward and resulting state s', we can update Q-table as follows:

$$Q_{k+1}(s, a) = (1 - \alpha)Q_k(s, a) + \alpha(R(s, a, s') + \gamma \max_a Q_k(s', a'))$$

In order to motivate exploration during the learning process, we are going to use the ϵ-greedy method for choosing an action.

3.2 Deep Q-Network (DQN)

DQN is a variant of the Q-learning algorithm where a neural network is used to non-linearly approximate the Q-function.

In the literature, we can find a common technique for achieving better performance which is called experience replay [2]. Standard versions of experience replay in deep Q-learning consist of storing experience-tuples of the agent as it interacts with it's environment. These tuples generally include the state, the action the agent performed, the reward the agent received and the subsequent action. These tuples are generally stored in some kind of experience buffer of a certain finite capacity. During the training of the deep Q network, batches of prior experience are extracted from this memory. Importantly, the samples in these training batches are extracted randomly and uniformly across the experience history.

Also, we are going to make use of ϵ-greedy method for taking a new action.

3.3 Double Deep Q-Network (DDQN)

DQN often suffers from over-estimation on Q-values as the max operator is used to select an action as well as to evaluate it. Double DQN (DDQN) (DDQN), Van Hasselt et al. (2016) [3], is thus used to de-couple the action selection and Q-value estimation to achieve better performance.

The implementation is similar to DQN but now two neural networks (θ_A and θ_B) are used to estimate the Q-function. The idea is to switch from one network to an other after some episodes or to switch randomly, but never on the same time steps. Thus, the iteratively Q-function updating is the following:

$$Q_{\theta_A}(s, a) = r + \gamma Q_{\theta_B}(s, \operatorname*{argmax}_{a'} Q_{\theta_A}(s', a')$$

and vice-versa when using the other network,

$$Q_{\theta_B}(s, a) = r + \gamma Q_{\theta_A}(s, \operatorname*{argmax}_{a'} Q_{\theta_B}(s', a')$$

To improve exploration, again, we are going to use ϵ-greedy technique.

3.4 Advantage Actor Critic (A2C)

A2C is a policy gradient algorithm and it is part of the on-policy family. We are going to implement two networks, the *critic* and the *actor*. The *critic* estimates the value function. This could be the action-value (the Q value) or state-value (the V value). The actor updates the policy distribution in the direction suggested by the critic (such as with policy gradients). That means that we are

learning the value function for one policy while following it, or in other words, we can't learn the value function by following another policy. We will be using another policy if were using experience replay for example, because by learning from too old data, we use information generated by a policy (i.e. the network) slightly different to the current state. To update the network we collect experiences, and process them immediately:

- Do interaction with the environment and gather state transitions.
- After n-steps, calculate updates and apply them.
- Discard the collected data.

This works because the data we use is generated following the same policy we are updating. Additionally, an advantage function is included as a rescaler of the gradients ($A(s,a) = Q(s,a) - V(s)$). For this A2C algorithm, we are going to implement a n-steps approach and make use of ϵ-greedy approach to encourage exploration.

4 Experiments

As we have mentioned before, we are going to run the algorithms within the `LunarLander-v2` environment. For each algorithm we are going to perform 5 runs (due to the limited computational power) of at least 3000 episodes each. In case that we do not observe convergence, we are going to let the algorithms run for longer (e.g. Q-learning usually needs more iterations). Also, each episode is going to be limited to 1000 steps. For choosing the hyperparameters, we selected reasonable values found in the literature and tune them according to our specific case. Computational costs of this algorithms is expensive and the number of parameters is large, and performing exhaustive grid-search for optimum parameters was impracticable, due to lack of computational resources available (no GPU and run a laptop).

5 Result analysis

In this section we are going to discuss about the obtained results. First of all, table 1 shows the average reward of 100 executions with the learned Q-functions and policies. We can see that best results are achieved with DQN and DDQN. The poor performance of the Q-learning algorithm can be explained by the discretization process, which in our case we discretized continuous variables of the state representation into 8 bins. Furthermore, we can observe that A2C algorithm did not perform as it was expected in terms of quality. Later we will analyze what happened to its performance.

Besides, regarding the execution times, A2C seems to be faster than others per executed episode. This is because we implemented the n-steps. However, as a counter-effect, A2C needed more episodes to converge.

Algorithm	Avg. reward of 100 executions	Avg. training time per episode
Q-learning	-53.59	0.24 s
DQN	263.95	1.1 s
DDQN	235.05	2.1 s
A2C	62.74	0.01 s

Table 1: Average rewards of 100 executions of the trained models.

Focusing on convergence, figure 1 shows the average reward evolution of 5 training runs for each algorithm. We can see that Q-learning gets stuck converging to an average reward of under 0 points, and with a lot of variability. Observing DQN and DDQN, the performance and achieved results are quite similar, but DDQN seems to achieve convergence faster than DQN on average (DDQN gets over 100 points twice as faster than DQN). A2C, as we mentioned previously, takes more time to converge due to the n-steps implementation (networks are not constantly being updated). Even though, we notice that A2C converges, on average, around 100 points of reward, while DQN and DDQN achive +200 points, which is quite poor for A2C.

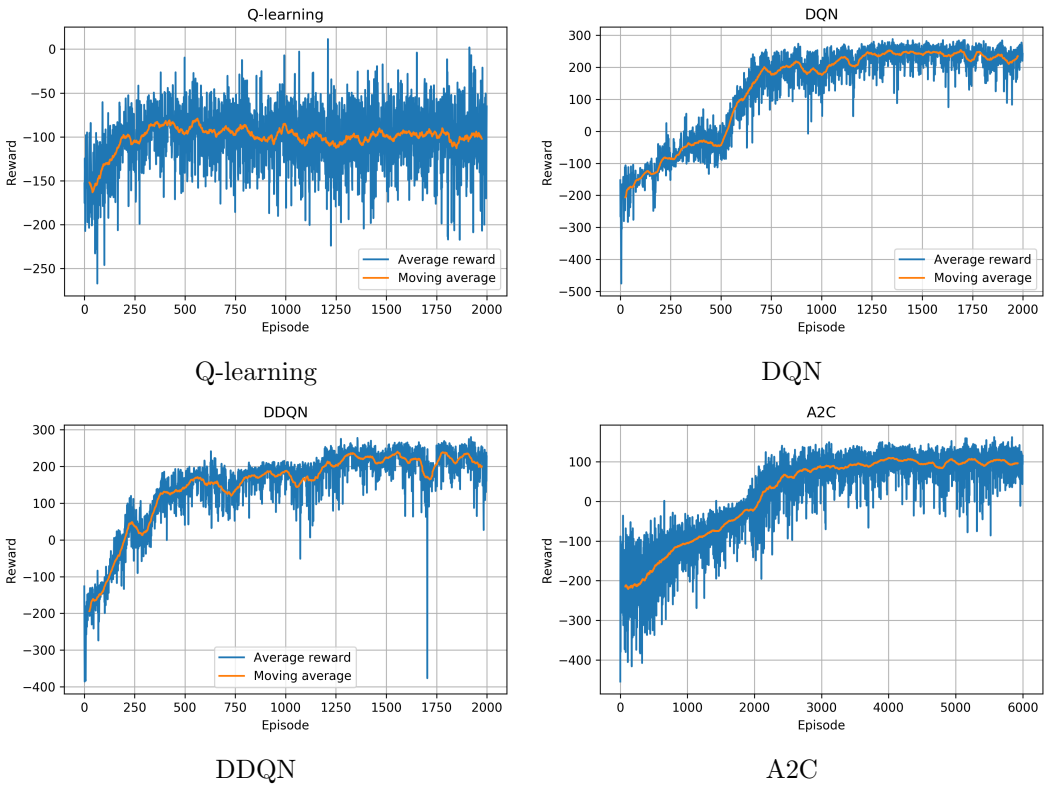
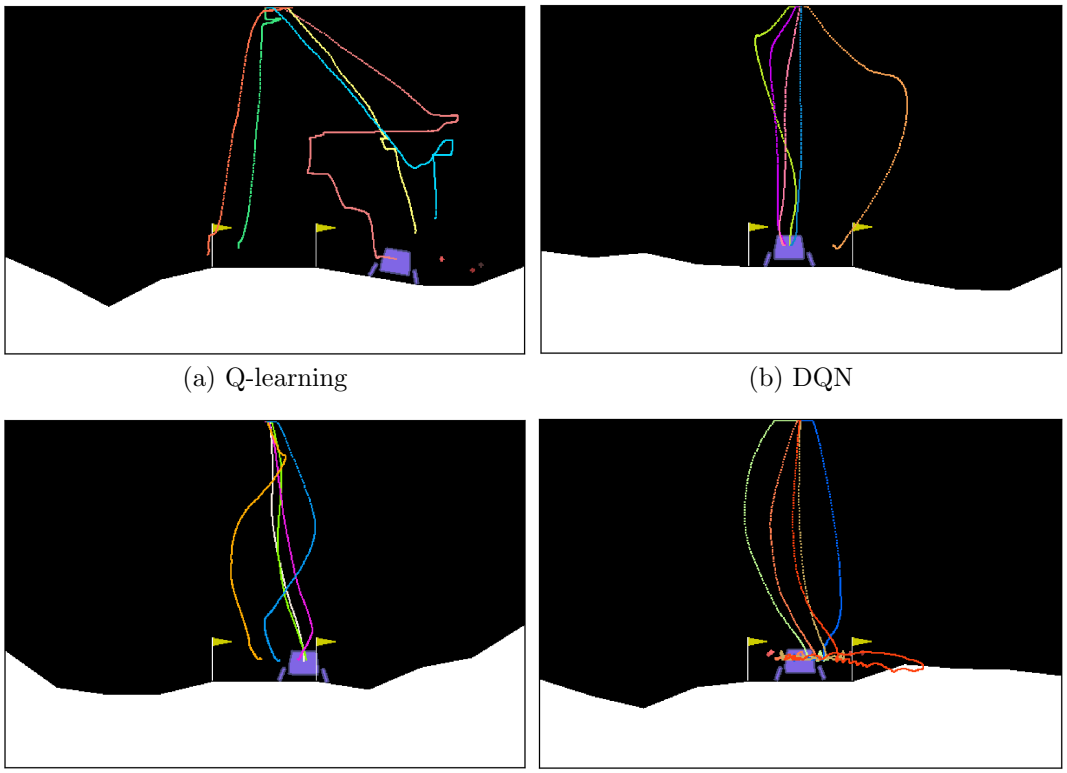


Figure 1: Average reward achieved along the episodes.

Figure 2 illustrates some qualitative results of the trained models. By collecting the coordinates at each step, we managed to display the trajectory of the landers on each algorithm to have a visual idea of their performance. For proper visualizations of the executions, some GIF animations can be found in this github repository.

We note that with Q-learning, the lander manages to fly and control the spacecraft and sometimes even land it on the pad, but most of the times messes it up. We can also observe that DQN, DDQN and A2C algorithms manage to have great control of the lander and successfully drive it to the landing pad. However, in the case of A2C (figure 2(d)), we observe that the lander seems to continue flying although it has already arrived to the target position. This is strange learned behavior makes the agent to lose points on the long-term reward, and can be the explanation of the poor reward results presented in table 1.

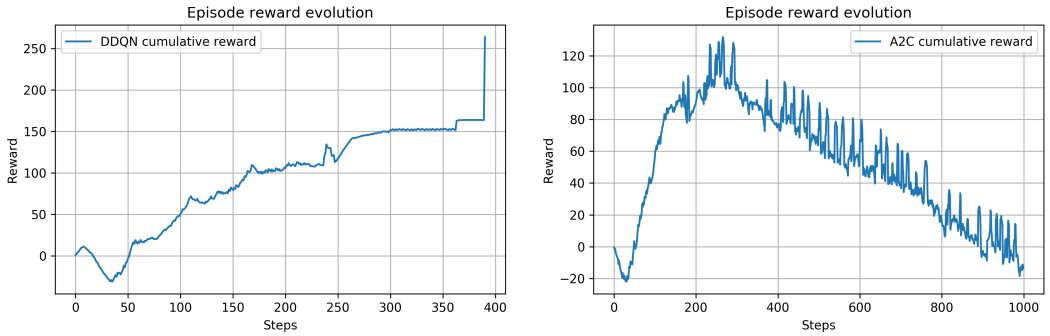


(a) Q-learning (b) DQN

(c) DDQN (d) A2C

Figure 2: Example of 5 runs with each RL algorithm.

If we get a closer look on A2C agent behavior within a single episode, we observe a strange evolution of the rewards (figure 3). Maybe, due to an implementation error, or because of the myopic algorithm behavior, the A2C agent discovers that hitting the ground with the legs of the lander gives +10 points of reward. In figure 3(b), we can see that in the early steps the A2C agent earns positive rewards for rapidly going down to ground, but then begins to perform small jumps for earning +10 points, which are represented by the mini-peaks in the plot. This makes the agent to never turn down the engine and finish the episode by landing, thus, it always obtains a poor long-term reward. This behavior of *small jumping* of can be better understood by watching the rendered animation here.



(a) DDQN (b) A2C

Figure 3: Comparison between DDQN and A2C reward evolution of a single episode.

6 Conclusions

Q-learning might work well for discrete small environments. For the case of `LunarLander-v2`, we had to discretize the states, and depending of the number of discrete bins. the performance might improve or not. In our case, setting too many discrete bins produced an sparse over-dimensioned matrix per representing the table $Q(s, a)$ that sometimes couldn't fit into memory. For this reason, DQN, DDQN and A2C are much more suitable methods for solving the continuous environment of `LunarLander-v2`.

The use of a Double Q-learning Network (DDQN) instead of a simpler DQN appears to increase learning convergence, although depending on how frequently we train the networks it could lead to significant speed improvements.

Surprisingly, these algorithms can find curious policies to get reward points such as the one obtained with A2C agent, which learned to perform little jumps to be rewarded each time the lander legs touched the ground.

In addition, we found that adjusting the appropriate parameters to achieve a stable and successful learning process for RL algorithms can be a tedious task. Some algorithms, e.g. A2C, has lots of hyperparameters: hyper parameters for each actor and critic networks, hidden layers on each network, the number of n-steps, adequate balance between exploration and exploitation, the ϵ-greedy decay factor, the discount factor, experience replay buffer size, etc.

This work has some interesting future work . For instance, training is currently done using a fixed learning rate, but decaying it over time may lead to better results, and the initial learning rates and, in general, many other hyperparaments parameters should benefit from a more exhaustive grid search. Also, for the implemented The current algorithm uses a ϵ-greedy strategy, but other approaches may lead to better results.

The implementation of the algorithm, the saved models, and code to reproduce the illustrations and plots and can be found in this github repository.

References

[1] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," 2016.

[2] S. Zhang and R. S. Sutton, "A deeper look at experience replay," 2017.

[3] H. van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," 2015.