Solving the Continuous Lunar Lander Problem

Final Project Report in Reinforcement Learning Course

Submitted by Ran Algiser, 322768805 & David Guriel, 211481130

Bar-Ilan University

March 2021

1 Introduction

In this project we solve the *Continuous Lunar-Lander Problem*, which was published in OpenAI Gym (1). In this problem, a spaceship is trained to land on the moon, using its engines and observing the spaceship's current state. We solve the task using a Reinforcement-Learning system, which is a system that learns how to act in response for given observations about the spaceship's state.

In this task, as well as in the other OpenAI Gym tasks, the environment is already implemented for us. This means that we should not bother ourselves with how the physics of the situation works, and how the spaceship is influenced by it. Instead, there exists an API for us to easily get the spaceship's states and make it act upon our instructions.

Every observation the spaceship produces consists of 8 parameters:

1. Its location, with x, y coordinates.
2. Velocity - in x and y axes - v_x, v_y.
3. Angle and angular velocity - θ, v_θ.
4. Two boolean variables for the spaceship's legs, which indicate whether each leg touches the moon's ground. They also tell whether the landing ends successfully.

In addition, there are 2 continuous variables that indicate the amount of power the main (horizontal) and vertical engines should apply. The observation space of the environment, as well as the action space of the agent, is described in table 1. The 8 parameters of the spaceship's observations, along with the 2 action variables are visualized in figure 1.

Variable Name	Data type	Range
x coordinate	Continuous	$[-2, 2]$
y coordinate	Continuous	$[0, \infty]$
v_x, horizontal velocity	Continuous	$[-\infty, \infty]$
v_y, vertical velocity	Continuous	$[-\infty, \infty]$
θ orientation	Continuous	$[-\infty, \infty]$
v_θ angular velocity	Continuous	$[-\infty, \infty]$
Left leg touching the ground	Boolean	$\{0, 1\}$
Right leg touching the ground	Boolean	$\{0, 1\}$
e_v vertical engine	Continuous	$[-1, 1]$
e_h horizontal engines	Continuous	$[-1, 1]$

Table 1: The ranges of every parameter in the parameter space

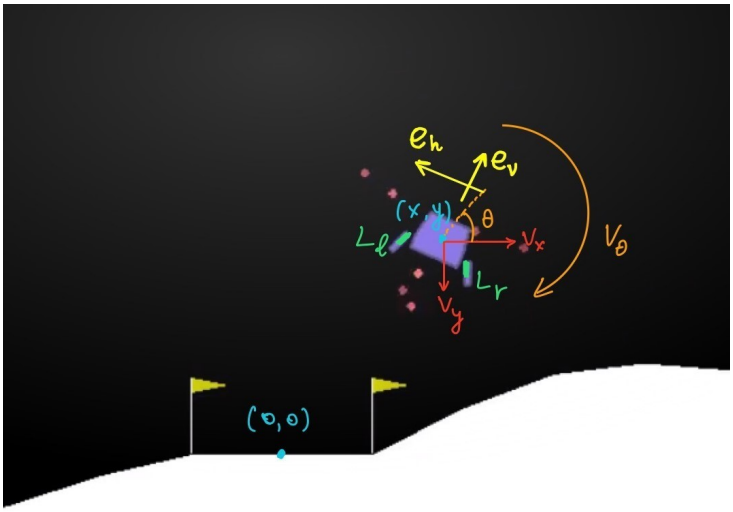

Figure 1: The 10 variables involved in the calculations

1.1 Related Works

1.1.1 SARSA

SARSA, which stands for State–Action–Reward–State–Action, is a model-free, tabular reinforcement learning algorithm based on Bellman equations. It was first proposed by Rummery and Niranjan in (2) with the name "Modified Connectionist Q-Learning" (MCQ-L).

1.1.2 DQN

DQN is a model free reinforcement learning algorithm that is based on the classic Q-learning algorithm. This algorithm learns to approximate the state-action value function $Q(s, a)$ using a Neural Network. It was introduced in (3), which made the algorithm work on a series of Atari games.

1.1.3 Double DQN

Double DQN (DDQN) is an improvement over DQN, with the goal of reducing overestimation by decomposing the max operation of our agent into two separate operations, action selection and action evaluation. It was first introduced in (4) and has been heavily utilized since.

1.1.4 Dueling DQN

Dueling DQN is another improvement over the classic DQN, with the goal of improving the speed and stability of convergence of DQN by separating our Q-network into two separate heads. One head will calculate a state-value function that is not affected by some concrete action while the other will be in charge of how each action affects the current state. This change allows the network to change the value of some state-action pair without changing the value of the state itself, which should bring a faster convergence. This idea was first presented in (5), which showed significant improvements in RL benchmarks.

1.1.5 Rainbow DQN

Rainbow DQN is an extension to the DQN algorithm that combines several DQN improvements into a single module. It includes:

1. Double Q-Learning
2. Prioritized Experience Replay
3. Dueling networks
4. Multi-step learning
5. Distributional reinforcement learning (Categorial DQN)
6. Noisy linear layers

The idea was presented in (6) and showed incredible results.

1.2 Goal

In the Continuous Lunar-Lander, the goal of the agent, who is controlling a spaceship, is to land on the moon. Yet, it is important for the spaceship to land with its two legs downwards, and to have minimal velocity. In the environment, these features are expressed through the requirement to achieve an average reward of 200 over 100 consecutive trials (episodes).

2 Solution

2.1 General Approach

As mentioned, in the project we decided to first examine a Tabular method. Then, we implemented neural methods.

In our work, we implemented (and did not take existing code) several methods and architectures for solving the problem. We first decided to examine a Tabular method, which turned out to be unsuccessful. Then, we implemented neural methods, which performed relatively well. In total, the algorithms we tested are:

1. Q-Learning with TD(0) (2)
2. Double DQN (4)
3. Dueling DQN (5)
4. Categorical DQN with Noisy Linear layers, inspired by (6)

In all the algorithms, we added an Epsilon-greedy technique, and also added it decay, so the model can explore new paths in the first episodes and be exploitative in the advanced steps of the learning. In TD(0), we used the formula

$$\varepsilon \leftarrow \varepsilon \cdot 0.94^n, \tag{1}$$

where n is a function of the current episode index. In the neural methods, the formula we used is

$$\varepsilon \leftarrow max(\varepsilon \cdot eps\_decay, 0.01), \tag{2}$$

and we took $eps\_decay = 0.996$. We also added an exponential learning-rate decay scheduling (similar to eq. 1).

The neural network methods utilized Experience replay, with a buffer of size 10000 and a batch size of 64.

2.2 Discretization

First of all, in order to be able to perform discrete calculations on the environment, we implemented auxiliary functions that *discretize* the continuous environment of the game. That is, they divide the observation and action spaces into discrete bins, and assign to every predicted coordinate its corresponding bin index. In TD(0) we discretized both the states and the actions spaces, so we can create the $Q(s, a)$ table. In the neural methods, since we did not need such table, we only discretized the actions space. The main auxiliary function we used was `numpy.digitize`, which splits the continuous ranges into discrete bins (or buckets).

For every element in the observation space, we set two variables, *range* and *number of buckets*. We then proceed to parse the *range* of every variable into buckets according to the number of buckets for each variable. We also add two

separate buckets to symbolize values that are outside the range (one to the left of the range and one to the right of the range). We illustrate an example in figure 2.

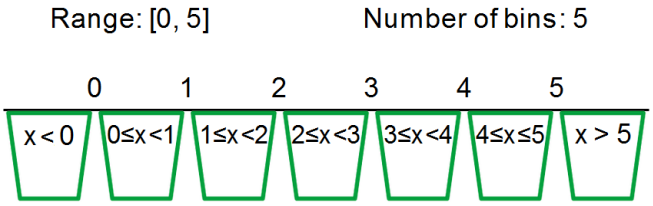

Figure 2: Example for a discretization of x. We first define our variables *range* and *num_of_bins*. Using them, we then calculate the bin limits. Given some observation of this variable, we associate it to the correct bin using the formulates that are written inside each bin.

A major technical challenge we experienced is the gigantic amount of memory the TD(0) method required: our observation space is made of 6 continuous numbers, as well as 2 boolean variables. The action space is comprised of two other continuous numbers. Let $number\_of\_bins = n$ for all variables (assuming it is the same). The produced Q-table then has $(n+2)^8 \cdot 2 \cdot 2$ cells. If we only assume $n = 8$, and given that the data type is Floating Point, then we get a table of size 1.6 gigabytes, which is obviously unfeasible to train. The solution we found for this problem is to use coarse discretization, which turned out to be not enough for the TD(0) to succeed.

2.3 Architecture

For every algorithm, We defined an interface for training the agent. First, we defined a class that represents the agent (`agent`) - it included a `predict` method, whose role is (as the name implies) to predict the best action, which results in the highest expected Return value. That is also where we used the ε-greedy technique. In TD(0), this method used explicitly Bellman equation, while at the deep methods it performed the forward calculation of networks.

In addition, we defined a `trainer` class, in which the entire training loop happens. We wrote the procedures `train` and `train_episode`, which together construct a modular and convenient way to manage the training process.

Of course, instead of writing the procedures over and over again, we wrote base classes for `agent` and `trainer`, and then created inherited classes for the actual algorithms.

Finally, we used external code in the Rainbow method implementation, specifically the training on the Categorical DQN (see https://github.com/Curt-Park/rainbow-is-all-you-need).

3 Experimental Results

As explained, we implemented 4 different agents. The exact architectures of the deep algorithms we used appear in Figures 3, 4, 5. In all 3 architectures, the output size for the actions is 81, and in Rainbow DQN the atom size is 51.

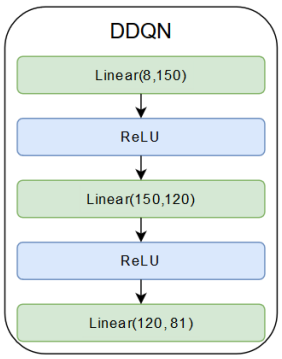

Figure 3: DDQN

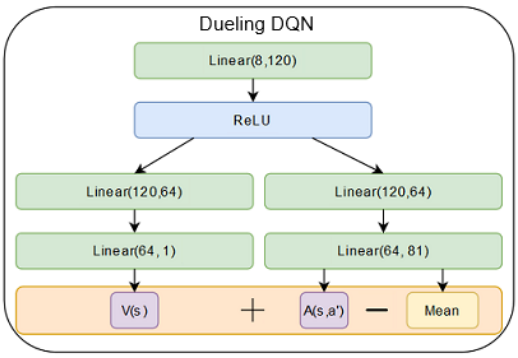

Figure 4: Dueling DQN

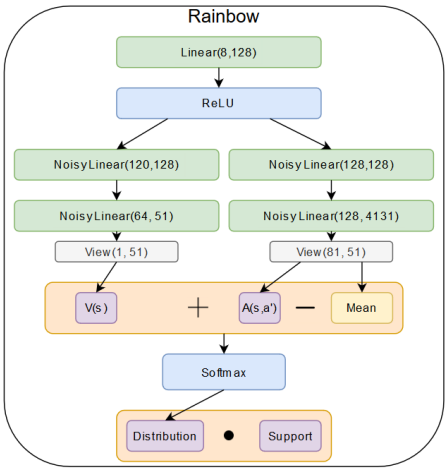

Figure 5: Rainbow DQN architecture

We then trained each one of the agents on our environment, as well as the environment with uncertainty (noise). We then evaluated them.

As explained in 1.2, the goal of the agent is to achieve an average reward of 200 over 100 episodes. This requirement can be tested in two different ways:

1. During training, track the accumulated rewards of the last 100 episodes. If at any point the average of these rewards is bigger or equal to 200, stop

training and announce that the agent has reached the requirement.

2. Every x episodes, run an *evaluation*, which is comprised of running 100 episodes of the environment with the agent using greedy (non-exploratory) policy, accumulating the rewards and calculating the average. If the agent reached the 200, stop training and announce that the agent reached the requirement. Note that we don't use these episodes for training (we don't insert them into the replay memory).

We chose to evaluate our agents using option 2 for the main reason that the recorded rewards of episodes in option 1 are usually lower than they should be since the policy is ε-greedy. Using option 2 in this case allows us to stop training earlier. Furthermore, option 1 updates the parameters of the agent during its evaluation, which adds variance to the rewards. This might make the agent diverge after its convergence, which might make us miss its success.

We chose to evaluate our algorithms with $x = 50$, so as to not slow our training too much. Table 2 features the final results of our agents.

The TD(0) algorithm completely failed in performing the task. We used the second option for evaluating - and the results are presented later in Figures 6 and 7.

Agent	Number of episodes	Figures
DDQN	700	8 9
Dueling DDQN	400	10 11
Rainbow	350	12 13

Table 2: Final results of our models

Table 3 features the final results of our agents when adding uncertainty to the environment.

Agent	Number of episodes	Figures
DDQN	950	14 15
Dueling DDQN	650	16 17
Rainbow	600	18 19

Table 3: Final results of our models when adding uncertainty to the environment

3.1 Hyper parameters

In TD(0), we took initially $\varepsilon = 0.8$ with $eps\_decay = 0.93$, as discount factor $\gamma = 0.99$ and learning_rate= 0.3. The agent trained for 1000 episodes.
All neural network agents used eps_decay = 0.996, experience replay of size 10000, batch size of size 64, 9 buckets for every action (which makes the output size 81). The DDQN and Rainbow agents use a learning rate of 0.0001 and Dueling DDQN uses 0.0002.

The Rainbow agent uses 51 atoms and $v_{max} = 100$, $v_{min} = 100$. These parameters are defined in the original paper (6).

4 Discussion

In this project we tried several methods for solving the continuous Lunar-Lander problem, by OpenAI Gym. We used one Tabular method - TD(0), and 3 neural methods - DDQN, Dueling DQN and Rainbow DQN.

The Tabular method, despite its strong theoretical basis, fails to solve the task. We conjecture the reason is technical issues - in order for it to had worked, it should have had the variables spaces split into much more cells, and 8 grids per variable were probably far from enough. Moreover, the running time was extremely high, and we decided to not keep on trying to make it work, and proceed to the neural methods.

The neural methods gave much better results. The Double DQN method was quite stable in the training process, and approached 0 in the average rewards (compared to much lower numbers in TD(0)). Dueling DQN & Rainbow DQN gave even better results, and pretty much solved the task after less than 1000 episodes.

Another insight we found is that the discretization size did not matter very much to the neural methods (unlike TD(0)), as they succeeded in approximating the $Q(s, a)$ function even with few bins for every variable.

Also, we noticed that despite the fact that theoretically the neural algorithms we used are expected to be robust, the networks are very sensitive to modification of the hyper-parameters. We had to perform grid-search for appropriate hyper-parameters in order for the networks to achieve good results.

5 Code

Here is the link for the private Github repository: https://github.com/ranran9991/RL.git.

6 Figures

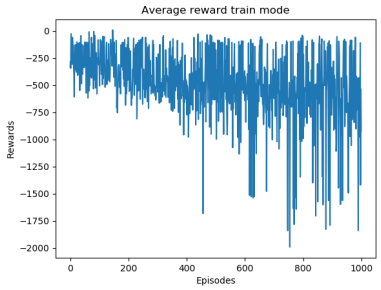

Figure 6: Average rewards of TD(0) in train mode

Figure 7: Average rewards of TD(0) in test mode

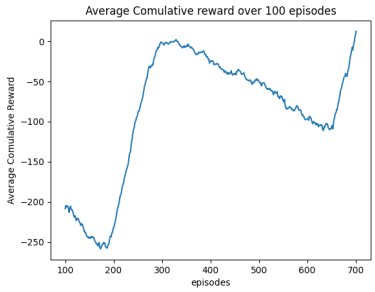

Figure 8: Average cumulative rewards of last 100 episodes for the DDQN agent over the environment without uncertainty

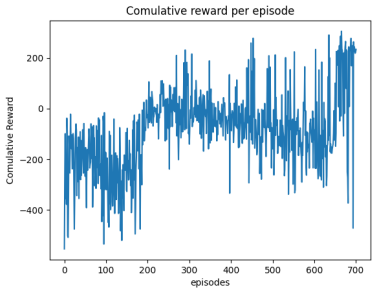

Figure 9: cumulative rewards of episodes for the DDQN agent over the environment without uncertainty

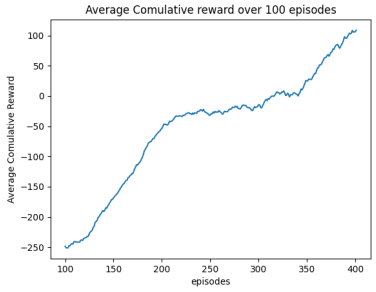

Figure 10: Average cumulative rewards of last 100 episodes for the Dueling DDQN agent over the environment without uncertainty

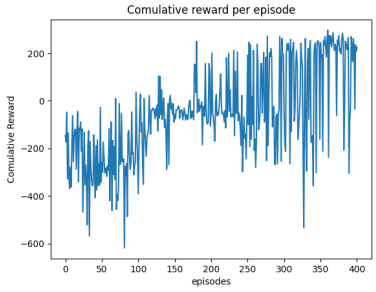

Figure 11: cumulative rewards of episodes for the Dueling DDQN agent over the environment without uncertainty

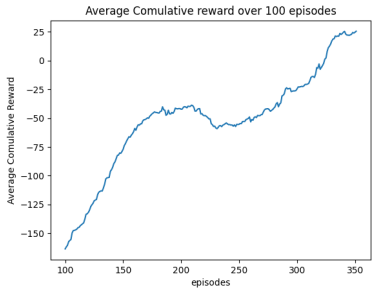

Figure 12: Average cumulative rewards of last 100 episodes for the Rainbow agent over the environment without uncertainty

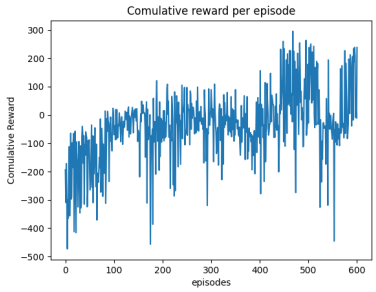

Figure 13: cumulative rewards of episodes for the Rainbow agent over the environment without uncertainty

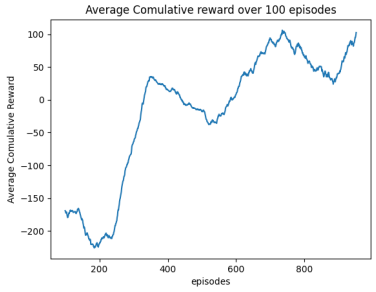

Figure 14: Average cumulative rewards of last 100 episodes for the DDQN agent over the environment with uncertainty

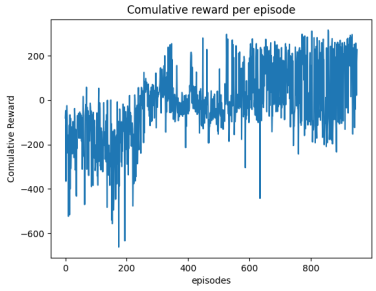

Figure 15: cumulative rewards of episodes for the DDQN agent over the environment with uncertainty

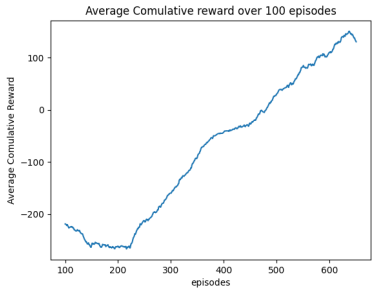

Figure 16: Average cumulative rewards of last 100 episodes for the Dueling DDQN agent over the environment with uncertainty

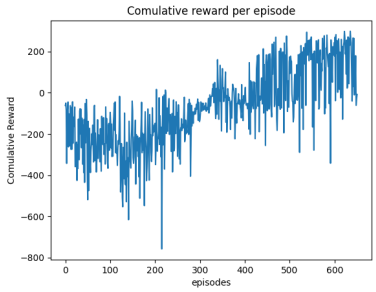

Figure 17: cumulative rewards of episodes for the Dueling DDQN agent over the environment with uncertainty

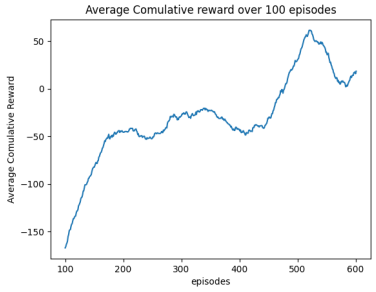

Figure 18: Average cumulative rewards of last 100 episodes for the Rainbow agent over the environment with uncertainty

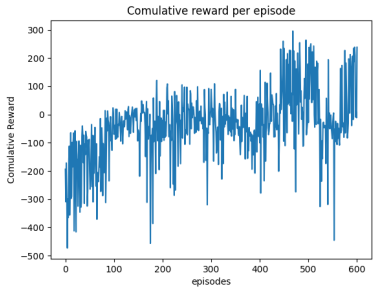

Figure 19: cumulative rewards of episodes for the Rainbow agent over the environment with uncertainty

References

[1] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," *arXiv preprint arXiv:1606.01540*, 2016.

[2] G. A. Rummery and M. Niranjan, "On-line Q-learning using connectionist systems," Tech. Rep. TR 166, Cambridge University Engineering Department, Cambridge, England, 1994.

[3] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," 2013.

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

[5] Z. Wang, T. Schaul, M. Hessel, H. van Hasselt, M. Lanctot, and N. de Freitas, "Dueling network architectures for deep reinforcement learning," 2016.

[6] M. Hessel, J. Modayil, H. van Hasselt, T. Schaul, G. Ostrovski, W. Dabney, D. Horgan, B. Piot, M. G. Azar, and D. Silver, "Rainbow: Combining improvements in deep reinforcement learning," *CoRR*, vol. abs/1710.02298, 2017.