Solving the Lunar Lander Continuous (v2)

Yuval Mor, Roey Fuchs

Submitted as final project report for Reinforcement Learning course (896873),
Bar-ilan University, 2021

1 Introduction

This project explores the application of different reinforcement learning methods and extensions, on the continuously lunar lander problem provided through the OpenAI gym framework [1]. In this environment, the lander starts at the top of the screen and should land between two flags with zero speed, as shown in figure 1.

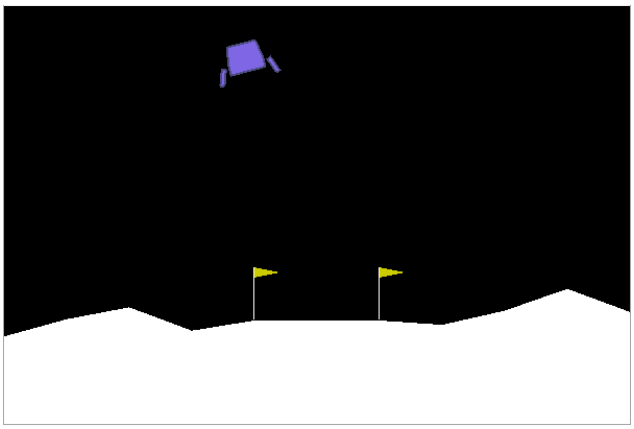

Figure 1: Moment after the environment starts. The lander fall from the top middle of the screen, and should land between the flags.

For each frame, the lander sends its current state in the environment, with the following data:

$$State = \begin{cases} x \text{ coordinate of the lander} \\ y \text{ coordinate of the lander} \\ v_x \text{ the horizontal velocity} \\ v_y \text{ the vertical velocity} \\ \theta \text{ the orientation in space} \\ v_\theta \text{ the angular velocity} \\ \text{Left leg touching the ground} \\ \text{Right leg touching the ground} \end{cases}$$

All the values in the state are continuous, besides the last two which are booleans.

For each state of the environment, the agent takes an action based on its current state. The action is composed of two float numbers – the first effects on the main engine, and the second on the sides engines. The main engine active when its value in the range $[0, 1]$. The left engine active for values in range $[-1, -0.5]$ and right engine for a values in range $[0.5, 1]$. The agent can choose to take action from 15 discrete possible actions.

Our goal is to create an agent which will solve the environment as fast as possible. To do so, it will need to get an average reward of 200 points over 100 episode.

1.1 Related Works

As we mentioned before, the Lunar lander is a popular environment, so over the internet, many attempts to solve it can be found. Most of them were of the discrete version due to its simpleness and lower computing resources, especially in table methods like SARSA.

Since the discrete environment and the continuous one are similar but not identical, we tried to get from related works specific improvements, especially those related to reduce the number of states.

An interesting example for that can be found in comparison [2] between SARSA with bins optimizations and without. Bins optimization stands for reducing the number of states – a bin of values will be interpreted as the same value. Example in figure 2. Another usage of bin optimizations is in the greedy-epsilon method, such that every bin has their epsilon for epsilon–greedy method, so the decay is usually more robust. Another

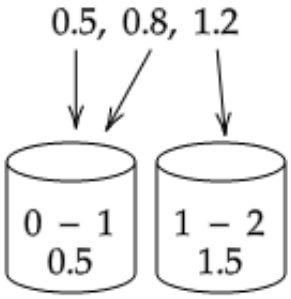

Figure 2: Example of bins optimize. Here we have 2 bins. values in range of 0 to 1 will convert to 0.5, and values from 1 to 2 will convert to 1.5. 0.5 and 0.8 enter the first bin, and 1.2 to the second.

important technique we use is experience replay [4], which reached the front of the stage with DeepMind Atari project [5]. An improvement to experience replay, we also tried to use prioritized experience replay [6], which use prioritized select instead of random select in regular experience replay. We will discuss this later.

2 Solution

2.1 General approach

We decided to start with a table method and then move forward to deep Q learning. The first implantation was the SARSA algorithm, and afterward DQN, DQN with extensions, and double DQN. We assumed that the SARSA agent won't converge due to the huge possible states, and deep learning should overcome this difficulty.

2.2 Design

We used Python 3.6 and few popular Python libraries – NumPy (effective math compute), TensorFlow and Keras (deep learning), and Matplotlib (graphs).

As we mentioned in the introduction, we needed to convert infinity continuous action space to discrete. We wanted to balance between the number of actions and the training process time – more actions the lander can take will cause longer training time. On the other hand, more actions can achieve better control when landing. That's why we decided to set 15 different actions, almost 4 times than the discrete version of the lunar lander.

The actions that our lander can take are any combinations of the values shown in table 1.

Main engine	0		0.5		1
Left\Right engine	-1	-0.75	0	0.75	1

Table 1: Lander actions

2.2.1 SARSA

SARSA is a table method, so we implemented it mostly with Python lists and NumPy arrays. We used code from a tutorial [3] as a skeleton.

Since SARSA is a table method, we had to quantizing the state space, namely – to choose bins, as we already discribe in figure 2. A state defined as 6 continuous values and 2 boolean values, and we have 15 possible actions. First, we divided each feature to 20 bins from -1 to 1 with steps of 0.1. This led to huge table size:

$$\overset{con.vals}{20^6} * \overset{bool.vals}{2*2} * \overset{actions}{15} = 3.84 * 10^9 \quad (|S| * |A|)$$

Even when we doubled the steps to 0.2, we had $60 * 10^6$ cells. We decided to find more specific values, separately for each feature (bins optimization) [2], by playing the game manually – we checked what values we need to separate, as we show in a figure 3.

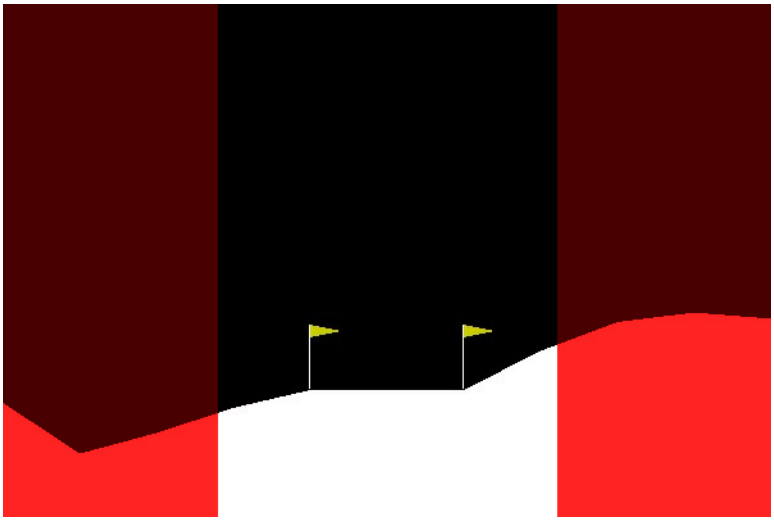

Figure 3: Example of relevant x coordinate area select. The red area (from each side), can be defined as the same area, because if the lander is there, its too far from the center, and the action will be the same – full power to the left or full power to the right.

After this process, we decided to use bins as shown in table 2, a total of $14 * 10^6$ cells. Any smaller separate led to unwanted behavior of the lander.

feature	**start**	**end**	**values**
x	-0.2	0.2	9
y	0	1.1	9
v_x	-0.3	0.3	6
v_y	-0.5	0.1	10
θ	-0.3	0.3	8
v_θ	-0.2	0.2	6
left leg	0	1	2
right leg	0	1	2

Table 2: SARSA features bins. The values column shows how many bins each feature has.

2.2.2 DQN

We saw that producing and updating Q-table is not sufficient in this huge state space environment, so instead of using a Q-table, we used Deep Q Neural Network. DQN uses a neural network to approximates Q-values for each possible action at each step. Our code is based on this repo. We examine different DQN agents, with a variety of extensions, and choose the one with the best performance. One common difficulty for all the models was to reproduce the results, for tuning the hyper-parameters. This problem happens because there's an aspect of randomness in the epsilon greedy policy and the random batch that is chosen each step from the replay memory. To overcome this problem, we ran every tuning option number of times, and average the results. One more difficulty was long-running times (6–8 hours).

The first agent we implemented is a SimpleDQN, with a greedy policy for choosing an action, and experience replay mechanism. Greedy policy meaning selecting the action with the highest Q value among all the Q values for a specific state. Experience replay's purpose is to consider previous experiences and thereby smooths the training distribution over many past behaviors [5]. In each step, we are saving a tuple that contains: the current state, action chosen, reward, and the next state after performing the action $(e_t = (s_t, a_t, r_t, s_{t+1}))$. The agent train the model on a minibatch that is pulled randomly from the replay memory.

The second agent was DQN with ε-greedy. We changed our policy for choosing an action. Instead of always taking the one with the highest Q value given by the network output, the agent generates a random number between [0,1), and if the ε smaller than this value we will choose a random action. Since ε denotes the amount of randomness in the policy, we want to start with a fairly randomized policy and later slowly move towards a deterministic policy. Therefore, ε starts with the value 1, and exponential decay each step until it reached 0.01.

The third agent was DQN with a prioritized experience replay (PER). The key idea is that the agent can learn more effectively from some experiences than from others. Therefore, we will take more frequently replay experiences with high expected learning progress, as measured by the magnitude of their temporal-difference (TD) error for training [6]. To avoid a situation where the error is zero, we added constant epsilon to the calculation. Therefore, we save a new tuple in replay memory: $e_t = (p_t, s_t, a_t, r_t, s_{t+1}), p_t = |TD_{error}| + \varepsilon$.

The fourth agent was DQN with a target network. In DQN we using the same network to predict and estimate the target value, as a consequence, there is a big correlation between the TD target and the parameters we are changing [7]. We use a separate network (target network) for estimating the TD target, and after every $N = 50$ step, we are updating the target network wights with the prediction network parameters. Using a target network makes the learning more stable because the target network stays fixed for some time.

The last agent was Double DQN, which is almost identicle to the DQN with target network, except for the action choosing. For DQN with target network we use eqution 1, and for Double DQN we use eqution 2. Double DQN helps us reduce the overestimation of Q values and, as a consequence, we get more stable learning.

$$\triangle w = R + \gamma \max_a \hat{Q}(s', a) - Q(s, a) \tag{1}$$

$$\triangle w = R + \gamma \hat{Q}(s', \arg\max_a Q(s', a)) - Q(s, a) \tag{2}$$

3 Experimental results

All the algorithms below have several hyper-parameters, like learning-rate, decay-rate, batch size, etc. We used grid search to choose the bests ones. In addition, the neural network model structure in DQN can be with endless options – layers, number of neurons, optimizer, activation function. We checked a lot of possibilities for the network parameters, and choose the bests. The final result is described in table 4.

3.1 SARSA

Unfortunately, no matter which hyper-parameters we have chosen, the agent wasn't succeeded to solve the environment. We believe it happened due to the huge size of the Q table. The Q table was sparse, and that's why the agent failed to converge in a relevant number of epochs ($< 100,000$). In figure 4, we show the best result we achieved with this algorithm.

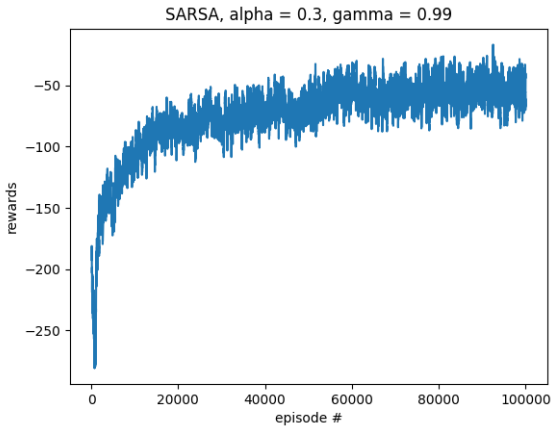

Figure 4: SARSA agent, $100,000$ epochs (average result over 100 episodes)

3.2 Simple DQN

The network anssmble from 2 hidden layers, the first one with 150 neruns, and the second with 120, with *relu* activation function, and for the output layers we used *linear* activtion function. We use MSE loss function, and $ADAM$ optimizer. The hyper-prametrs described in table 3. The best result shown in figure 5.

γ	0.99
α	0.0005
batch size	64

Table 3: Hyper-prameters for the simple DQN model

3.3 DQN with ε-greedy

This network is base on the Simple-DQN, with the same parameters and hyper-prameters. We start with $\varepsilon = 1$ and decay in every step as shown in equation 3, after the model saves 64 samples.

$$\varepsilon = \max(0.01, \varepsilon * 0.996) \tag{3}$$

The result shown in figure 6.

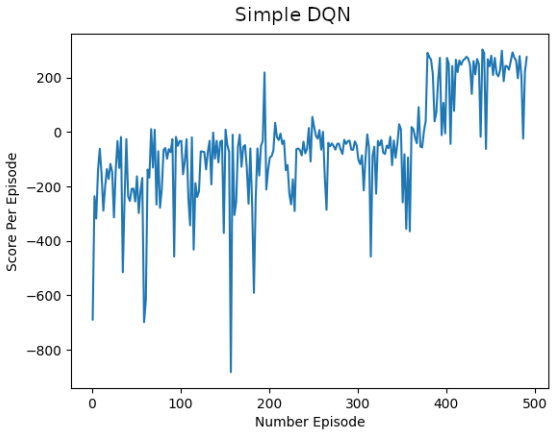

Figure 5: Simple-DQN network

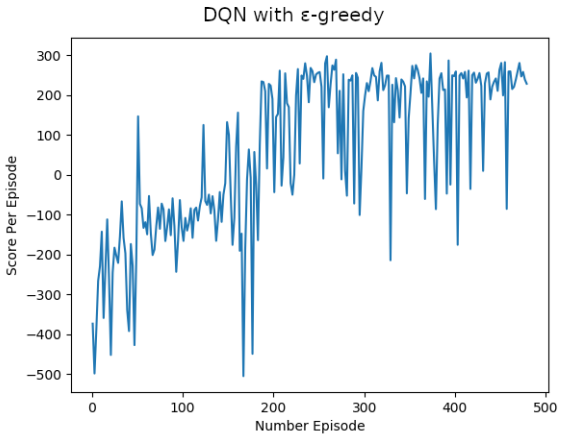

Figure 6: DQN with ε-greedy network

3.4 DQN with PER

This network is base on the DQN with ε–greedy. The preformend was worse than the DQN with ε–greedy model and not converge after 1000 epsidos. the result shown in figure 7.

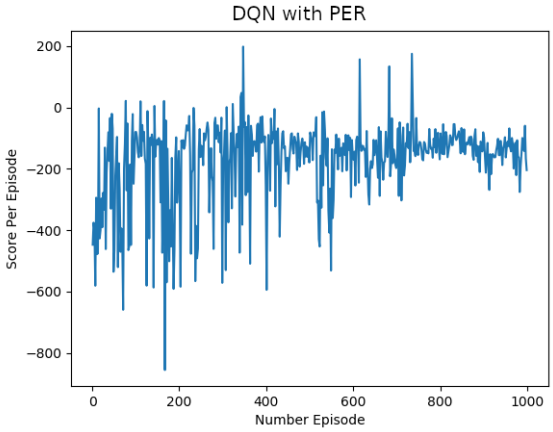

Figure 7: DQN with PER

3.5 DQN with target network

This model base on DQN with ε-greedy model. The addition hyper-parameter in this method is N, which presents the number of steps between updates of the target network wights. After tuning we use $N = 50$. The result is shown in figure 8.

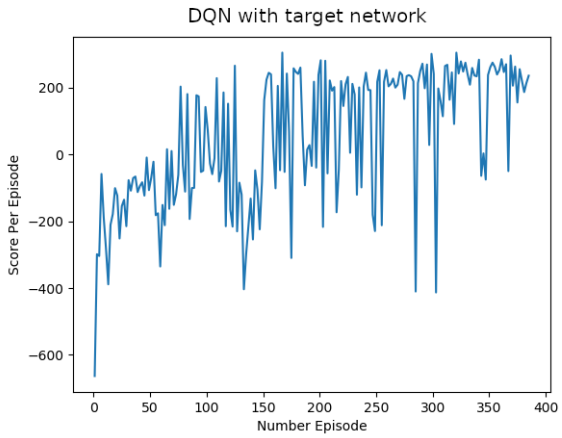

Figure 8: DQN with target network

3.6 Double DQN

This model base on DQN with a target network. The result was very similar to the DQN with the target network model and shown in figure 9.

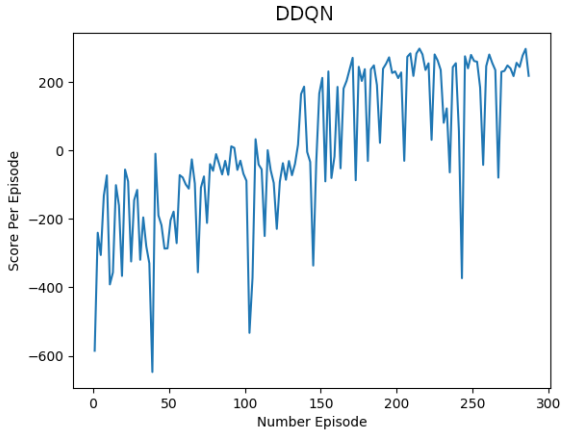

Figure 9: Double DQN

3.7 Double DQN with uncertainty enivronment

We choose the best model (DDQN) and added uncertainty to the location of the lander (using $ObservationWrapper$). The result shown in figure 10

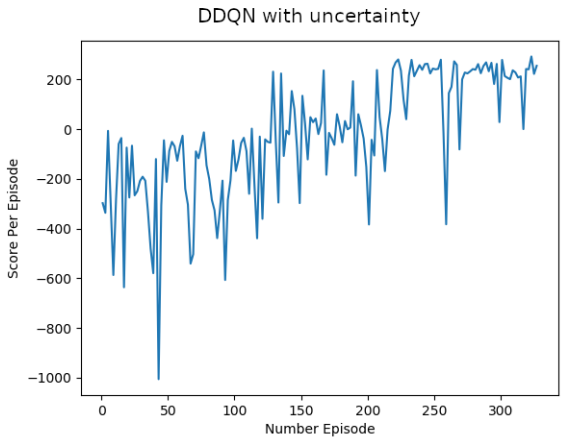

Figure 10: Double DQN with uncertainty enivronment

Agent	# Episodes
SARSA	∞
Simple DQN	491
DQN ε-greedy	478
DQN PER	∞
DQN target-network	385
DDQN	285
DDQN (uncertainty)	327

Table 4: Summrize of all agent and number of episodes until solving the environment. Note that the number of epsidoes including the 100 last episodes, that reach to average reward of 200.

4 Discussion

We started with a classic reinforcement learning method – SARSA. As we expected, this method isn't suitable for an environment with a lot of possible states. Afterward, we used more updated method – deep q learning. We started with basic DQN, and then examine extensions: ε-greedy, PER, target-network, Double DQN.

As we show in table 4 (note that the number of episodes including the 100 last episodes, that reach an average reward of 200), we can see improvement when using newer approaches. The exception is the PER model, which wasn't converged. The prioritization that we set, could lead to a loss of diversity and introduce bias [6].

The best model was DDQN, which solved the environment faster than the other models (285). We were surprised that the performance of the model was better than all other models even when we added uncertainty to the lander position.

5 Code

Our full implementation and running instruction can be found on GitHub.

References

[1] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.

[2] Soham Gadgil, Yunfeng Xin, and Chengzhe Xu. Solving the lunar lander problem under uncertainty using reinforcement learning. *arXiv preprint arXiv:2011.11850*, 2020.

[3] Alind Gupta. Sarsa reinforcement learning. https://geeksforgeeks.org/sarsa-reinforcement-learning, 2019.

[4] Long-Ji Lin. Reinforcement learning for robots using neural networks. Technical report, Carnegie-Mellon Univ Pittsburgh PA School of Computer Science, 1993.

[5] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[6] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*, 2015.

[7] Thomas Simonini. Improvements in deep q learning: Dueling double dqn, prioritized experience replay, and fixed. https://www.freecodecamp.org/news/improvements-in-deep-q-learning-dueling-double-dqn-prioritized-experience-replay-and-fixed-58b130cc5 2018.