Landing on the moon with Deep Q-Learning

Alberto Jesu
Department of Computer Engineering
University of Bologna
Bologna BO, 40121
`alberto.jesu@tudio.unibo.it`

Abstract

The purpose of this project is to compare, both in terms of training stability and testing performances, one of the most well-known algorithms for Deep Reinforcement Learning, Deep Q-Learning, and its improvements. In particular, the experiments will train the base DQN, the DQN with fixed Q-targets, the Double DQN and the Dueling Double DQN, and test them on the Lunar Lander environment from OpenAI gym. The results of the experiments show that, while the extensions to the base algorithm improved its performances greatly, the simpleness of the environment prevents us from identifying a clear winner.

1 Introduction

Q-Learning is an *off-policy*, *temporal-difference* algorithm for which the update is defined as

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \Big(R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t) \Big)$$

This algorithm, if used in problems where the state space is limited, works through the creation of a matrix known as *Q table* in which all the predicted Q values are stored and updated for each possible state and, as such, is called a *tabular method.* [1]

This approach, as the number of states increases, quickly becomes intractable; thus, a *function approximation* is used instead to approximate the state-value or action-value functions. One such approximator are Artificial Neural Networks, leading to Deep Reinforcement Learning.

1.1 Deep Q-Learning

Deep Q-Learning is a specialisation of Q-Learning with an artificial neural network as function approximator. Instead of keeping a table with all the possible states and their predicted Q-values, the *action-value* function is approximated by a neural network with a number of parameters $\ll$ number of states. The network takes the current state of the environment as input and outputs the predicted Q-values for all the possible actions in said state, and is called Deep Q-Network (DQN). [2]

1.1.1 Experience Replay

The DQN algorithm relies on a limited-length buffer known as *replay memory*. At every timestep t, the environment is in a state S_t and the agent chooses an action A_t, which makes the environment shift in a new state S_{t+1} and give to the agent a reward R_{t+1}. This quadruple

$$E = (S_t, A_t, R_{t+1}, S_{t+1})$$

33rd Conference on Neural Information Processing Systems (NeurIPS 2019), Vancouver, Canada.

is called an *experience*. While the agent interacts with the environment, its experience is stored in the replay memory, from which a minibatch of experiences is drawn randomly to perform the Q-learning updates. [2]

1.2 Fixed Q-targets

While the original DQN implementation consisted in a single neural network, this improvement uses a clone (which will be referenced to as *target* network from now on) of the standard network uniquely to compute the Q-targets $Q(S_{t+1}, a)$ during the Q-learning update step. After every C updates, the network is cloned once again to refresh the weights. This is done to make the algorithm more stable: using a single network means that an update that increases $Q(S_t, A_t)$ also increases $Q(S_{t+1}, a)$. In other words, both the Q-values and the target values shift during the Q-learning update, leading to oscillations while learning. [3]

1.3 Double Deep Q-Learning (DDQN)

In some cases, especially those involving stochastic environments, DQNs tend to overestimate the Q-targets, because the action chosen for the future state is the one with the highest Q-value. This might lead to the agent being unstable or converging slowly. An alternative approach uses the neural network to get the best Q-value for the next states, and the target network then calculates the Q-value of taking that action in the next state, leading to a double estimate. [4][5]

1.4 Dueling Double DQN (Dueling DDQN)

The Dueling Double DQN combines the advantages of Double DQN with a modified architecture. The key idea behind this innovation is that for many states, it is unnecessary to estimate the value of each action. In order to do this, the advantage function

$$A(S_t, A_t) = Q(S_t, A_t) - V(S_t)$$

combines the value-function $V(S_t)$, that estimates how good is to be in a determined state S_t, with the Q function $Q(S_t, A_t)$ that measures the value of taking a given action in said state. The advantage function, then, by subtracting V from Q, obtains a measure of the relative importance of each action. [6] The neural network is modified into two streams that compute *values* and *advantages* for each state, that are then merged in a unifying layer.

2 Environment

The environment chosen for this task is OpenAI gym's Lunar Lander. [7] In this environment, the agent controls a spaceship in a bidimensional world with the task of landing it safely onto the below landing pad.

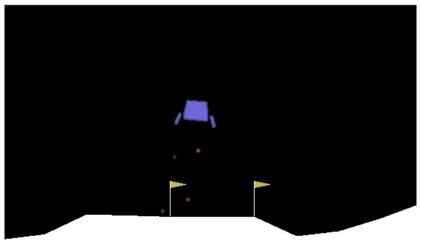

Figure 1: A screenshot of the Lunar Lander environment.

Such landing pad is always a flat region situated at coordinates $(0, 0)$ and is delimited by two yellow flags. The surrounding surface, instead, is randomly generated at every iteration.

2.1 State space

The state of the environment is represented by an eight-dimensional vector [7]

$$(x, y, v_x, v_y, \theta, \omega, ld_{left}, ld_{right})$$

where (x, y) are the lander's current coordinates, (v_x, v_y) are the horizontal and vertical velocities, θ is the angle of the lander, ω is the angular velocity,and (ld_{left}, ld_{right}) are two flags that are set to 1.0 if the corresponding leg of the lander made contact with the ground.

2.2 Action space

The environment action space is discrete and consists of four possible actions [7], as described in Table 1.

Number	Action
0	Do nothing
1	Fire left engine
2	Fire central engine
3	Fire right engine

Table 1: Action numbers and meanings.

2.3 Reward

The reward for moving from the top of the screen to the landing pad is $[100, 140]$ points, depending on the distance of the former to the latter. If the lander crashes or lands, a bonus of -100 or $+100$ respectively is assigned and the episode is terminated. Firing the main engine incurs in a -0.3 points of penalisation per frame.
The task is considered solved if the agent reaches an average reward of at least 200 points over the last 100 episodes. [7]

3 Implementation

In the scope of this document, four different versions of the DQN algorithm will be implemented and their performances compared: DQN, DQN with fixed Q-targets, DDQN, and Dueling DDQN.

3.1 DQN, DQN with fixed Q-targets, DDQN

The network architectures for these 3 agents are identical, with the only difference that the DQN with fixed Q-targets and the Double DQN also feature a clone of the network used for computing the Q-learning update. The architectures are described in Table 2.

Layer	Nodes	Activation
Input	8	None
Linear	32	ReLU
Linear	32	ReLU
Output	4	None

Table 2: Neural Network architecture for the DQN Agent.

3.2 Dueling DDQN

The Dueling DDQN agent network architecture is different. After a common layer, it is split into 2 output streams: an **advantage** stream and a **value** stream as described in Table 3.

Common		
Layer	**Nodes**	**Activation**
Input	8	None
Linear	64	ReLU
Advantage		
Linear	32	ReLU
Output	4	None
Value		
Linear	32	ReLU
Output	1	None

Table 3: Neural Network architecture for the Double Dueling DQN Agent.

Parameter	**Value**
Number of episodes	$2,000$
Batch size	64
Replay Memory size	$100,000$
γ	0.99
ϵ max	1.0
ϵ min	0.01
ϵ decay	0.99 per episode
η	0.001
Update frequency	4
Target clone frequency*	500

Table 4: Hyperparameters chosen for the experiments.

4 Results

The agents were trained over $2,000$ episodes with the same parameters, as highlited in Table 4.

The target network update frequency was not used for the basic DQN implementation, since it does not have a separate network for the computation of the Q-targets in the Q-Learning update.

4.1 DQN

The basic DQN agent solved the environment after 293 episodes, and reached a maximum average over 100 episodes of around 248 points.

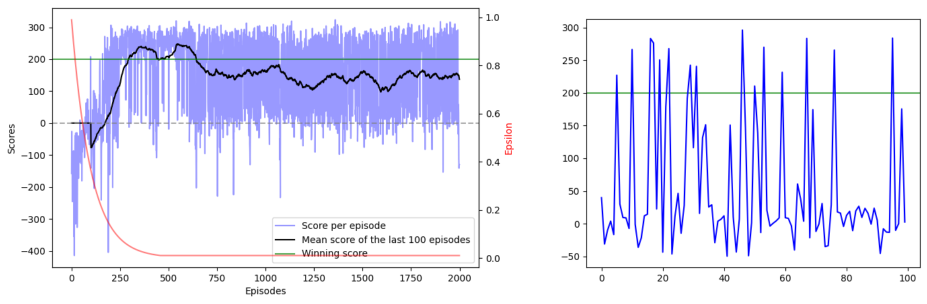

Figure 2: Performance of the basic DQN agent.

As shown in Figure 2, the agent performances, right after solving the environment, quickly worsens and the agent is not able to solve the task any more. Also, during testing, the agent was able to successfully land only on 15% of episodes.

4.2 DQN with fixed Q-targets

The Fixed-DQN Agent was able to solve the environment after 291 episodes, with the best average over 100 episodes of around 262 points.

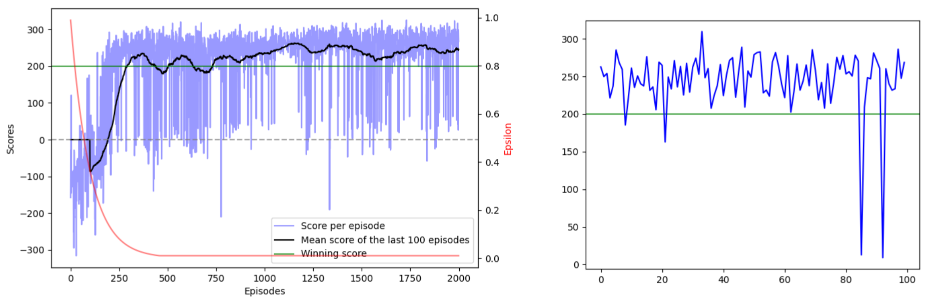

Figure 3: Performance of the DQN with fixed Q-targets.

As shown is Figure 3, it is able to keep completing the task even in later training stages. During testing, the agent lands successfully 97% of the times.

4.3 DDQN

The DDQN Agent was able to solve the environment after 459 episodes, with the best average over 100 episodes of around 261 points.

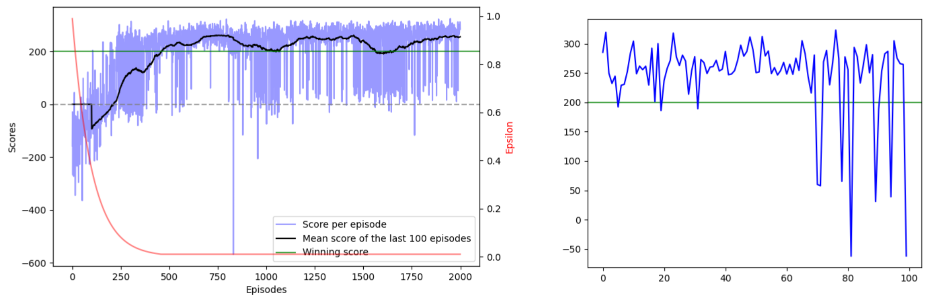

Figure 4: Performance of the DDQN agent.

While converging more slowly, the DDQN Agent showed a more stable trend throughout the 2,000 episodes training phase. However, it lands successfully during testing 89% of the times.

4.4 Dueling DDQN

Lastly, the Dueling DDQN Agent solved the environment after 326 episodes, reaching a maximum average reward over 100 episodes of around 274.

Despite the added complexity of the network, Figure 5 shows that this agent was able to solve the environment faster that the DDQN; it also holds the highest average reward, as well as the highest testing scores. During testing, the lander comes safely to rest 97% of the times.

5 Conclusion

As highlited in the previous section, and demonstrated in Figure 6, all the agents had a comparable performance during training time, with the base DQN performing the worst, being the one which yielded the lowest test score of all and being the least stable during training.

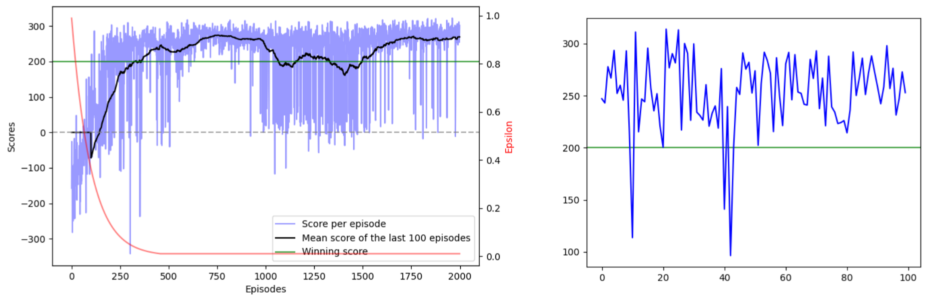

Figure 5: Performance of the Dueling DDQN agent.

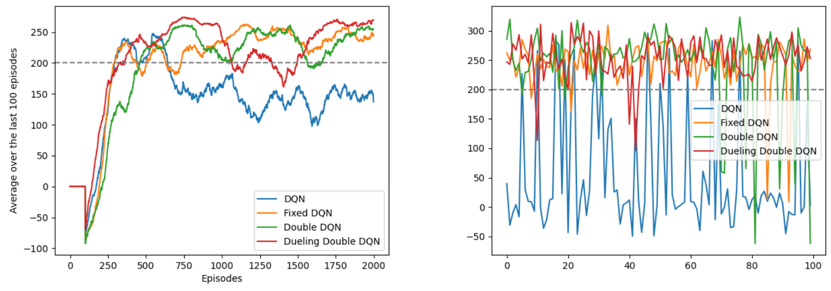

Figure 6: Comparison of the performances of the agents.

While, though, the theoretical improvements to the base algorithm actually delivered improved performances as well, it does not seem that there is a sizeable difference between them. It is possible to indentify which algorithm landed more times successfully and to compare the training behaviours of the agents, but these measurements are also subject to random fluctuations.

In conclusion, the DQN with Fixed Q-targets, the DDQN and the Dueling DDQN all improved on the base DQN algortihm in both training stability and testing results, with the Fixed Q-targets and the Dueling DDQN yielding the best results; however, there is not a clear winner and, in order to detect a more noticeable difference, it would be wise to repeat the experiments on more complex environments, which need more complex agents as well.

References

[1] Richard S. Sutton & Andrew G. Barto (2018) Reinforcement Learning: An Introduction. 2nd edition, MIT Press.

[2] Volodymyr Mnih et al. (2013) Playing Atari with Deep Reinforcement Learning

[3] Volodymyr Mnih et al. (2015) Human-Level Control Through Deep Reinforcement Learning

[4] Hado van Hasselt et al. (2016) Deep Reinforcement Learning with Double Q-Learning

[5] FreeCodeCamp - https://www.freecodecamp.org/news/improvements-in-deep-q-learning-dueling-double-dqn-prioritized-experience-replay-and-fixed-58b130cc5682/

[6] Ziyu Wang et al. (2016) Dueling Network Architectures for Deep Reinforcement Learning

[7] OpenAI Gym - https://gym.openai.com/envs/LunarLander-v2/