Playing Lunar Lander with Reinforcement Learning

Andrea Virgillito
Department of Computer Science and Engineering (DISI)
University of Bologna, Italy
andrea.virgillito@studio.unibo.it

Abstract

In this report I will employ 2 RL algorithms (DQN and Double DQN), to implement an agent capable of solving the Lunar Lander game, included in the OpenAI Gym toolkit, which is an interface dedicated to the study of RL algorithms. Performance of both algorithms applied to this particular environment will be compared, pointing out pros and cons of each one.

1 Introduction

Reinforcement Learning (RL) is about learning the optimal behavior in an environment to obtain the highest possible reward. To achieve this goal, the agent learns how to act, basing its behavior on interactions and observations, taking into account how the environment responds. Due to the fact that our agent has to learn the right sequence of actions without the presence of a supervisor, RL is undoubtedly an excellent solution to deal with partially observable environments. For these reason, RL is applicable to a wide range of problems that can not be tackled with classic machine learning algorithms. This project will provide an example of implementation of these RL algorithms.

2 Background

2.1 OpenAI Gym

As previously mentioned, I have used the OpenAI Gym toolkit to apply and study RL algorithms. In fact, OpenAI has been created to encounter the lack of standardization in this topic, allowing the researchers to benchmark their algorithms in various scenarios and easily sharing their results with other colleagues. Gym toolkit is also compatible with the Tensorflow framework (`gym` is a python library) and provides a wide choice of test environments like Algorithmic, Atari, Robotics and Box2D.

2.1.1 Lunar Lander

Lunar Lander environment is part of Box2D. It is characterized by a continuous 8-dimensional state space, and a (discrete) 4-dimensional action space: do nothing, fire right engine, fire down, engine, left fire engine. Coordinates are the first two numbers in state vector. The reward for moving from the top of the screen to landing pad and zero speed ranges between 100 and 140 points. By the way, landing outside the landing pad is always possible but the lander will lose reward back. One episode finishes if the lander crashes or comes to rest, receiving additional -100 or +100 points. Each leg ground contact is +10. Firing main engine is -0.3 points for each frame. The problem is considered *"solved"* when 200 points (of mean reward) have been achieved during the last 100 episodes. The fuel is infinite, so an agent can learn to fly and then land on its first attempt without any time constrains.[1]

[1]https://gym.openai.com/envs/LunarLander-v2/

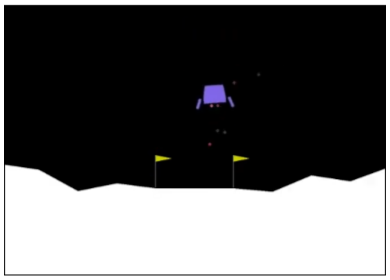
Figure 1: Lunar Lander environment.

2.2 Deep Q-Learning (DQN)

Deep Q-Learning is an *off-policy* algorithm, that updates its *Q-values* using the *Q-value* of the next state *s'* and the greedy action *a'*. It combines the traditional Q-Learning with a convolutional NN. *Experience replay* was introduced to deal with high correlation of successive samples: agent's experience is stored into a *replay memory*, that is accessed randomly to perform the weight updates.[1]

Algorithm 1: Deep Q-Learning with Experience Replay.

```
Initialize replay memory $\mathcal{D}$ to capacity $N$
Initialize action-value function $Q$ with random weights
foreach episode do
    Initialize sequence $s_1$ = $\{x_1\}$
    foreach step of episode do
        With probability $\epsilon$ select a random action $a_t$
        otherwise select $a_t = \max_a Q(s_t, a; \theta)$
        Execute action $a_t$ in emulator and observe reward $r_t$ and next state $x_{t+1}$
        Set $s_t{+}1 = s_t, a_t, x_t{+}_1$
        Store transition $(s_t, a_t, r_t, s_{t+1})$ in $D$
        Sample random minibatch of transitions $(s_j, a_j, r_j, s_{j+1})$ from $D$
        if $s_{j+1}$ is terminal state then
            $y_j = r_j$
        Else
            $y_j = r_j + \gamma \max_a Q(s_t, a; \theta)$
        Perform a gradient descent step on $(y_j - Q(s_j, a_j; \theta))^2$
```

2.3 Double DQN

Double DQN shares some principles with Deep Q-Learning, like the *experience replay* and the convolutional NN. Due to the fact that DQN uses the `max` operation, the selected *Q-value* tends to get very large over time (in a wrong way), compromising the performance of the Q-Network. This is the so called *overestimation of Q-values* problem. To solve this, Double DQN implements two networks that share the same architecture and the same weights, once initialized: a Q-Network and a Target Network. Q-Network chooses the best *Q-values* based on actions. The Target Network instead evaluates and selects the best ones. Once the *target update frequency* parameter has been defined, the weights of the Q-Network are copied to the Target Network every *n-steps*, to improve the stability of the training. This method is called *Hard update*.[2]

Algorithm 2: Double DQN Algorithm.

```
Initialize: $\mathcal{D}$ - empty replay buffer; $\theta$ - initial network parameters, $\theta^-$ - copy of $\theta$
Initialize: $N_r$ - replay buffer maximum size; $N_b$ - training batch size; $N^-$ - target network
  replacement freq.
for episode $e \in 1, 2, ..., M$ do
    Initialize frame sequence $\mathtt{x} \leftarrow ()$
    for $t \in 0, 1, ...$ do
        Set state $s \leftarrow \mathtt{x}$, sample action $a \sim \pi_{\mathcal{B}}$
        Sample next frame $x^t$ from environment $\mathcal{E}$ given $(s, a)$ and receive reward $r$, and append
          $x^t$ to $\mathtt{x}$
        if $|\mathtt{x}| > N_f$ then delete the oldest frame $x_{t_{min}}$ from $\mathtt{x}$ end
        Set $s' \leftarrow \mathtt{x}$, and add transition tuple $(s, a, r, s')$ to $\mathcal{D}$, replacing the oldest tuple if
          $|\mathcal{D}| \geq N_r$
        Sample a minibatch of $N_b$ tuples $(s, a, r, s') \sim$ Unif$(D)$
        Construct target values, one for each of the $N_b$ tuples:
        Define $a^{max}(s';\theta) = argmax_{a'} Q(s', a'; \theta)$
        $y_j = \begin{cases} \text{r} & \text{if } s' \text{ is terminal} \\ \text{r} + \gamma Q(s', a^{max}(s';\theta); \theta^-), & \text{otherwise.} \end{cases}$
        Do a gradient descent step with loss on $||y_j - Q(s, a; \theta)||^2$
        Replace target parameters $\theta^- \leftarrow \theta$ every $N^-$ steps
```

3 Project

In this section I have reported all the software prerequisites and the implementations that I have used in order to realize this project.

3.1 Setting the environment

For this purpose, I have used the latest official TF-Docker container, with Tensorflow 2.4.1 and Jupyter Notebook. Additional packages were needed to run the project code.

Python packages:

- `gym`
- `gym-notebook-wrapper`
- `matplotlib`
- `numpy`
- `keras`
- `Box2D`
- `PyVirtualDisplay`

Ubuntu `deb` package (in order to properly render the environment):

- `xvfb`

3.2 Neural Network architecture

I have adopted a similar architecture as shown in this report:[2]

- Input layer with dim. 8 (=*state space*)
- 2 fully connected hidden layers with 128 neurons each, with ReLU activation function
- Fully connected output layer with dim. 4 (=*action space*) with LINEAR activation function
- Target network shares the same NN architecture with Q-Network

[2]https://arxiv.org/pdf/2011.11850.pdf

3.3 Hyperparameters

Learning rate is maybe the most important parameter related to a NN, because it's in charge to change the model behavior in response to the estimated error, every time the model's weights are being updated. An inappropriate value could lead to very unfortunate situations: if it is too small training time could be very long (and the training task could miss its goal, never converging to the solution), and if it is too large, the training task could be too unstable and fast to get a valid result. **Minibatch** is the number of the random (s, a, r, s') tuples, sampled from the *experience replay*. **Epsilon** and **epsilon decay** are related to Exploration-Exploitation *trade-off*: epsilon should decay over the time as the learning task goes on. **Discount rate** is the one found in the Bellman's equation, describing the importance of the reward obtained in the current state. **Memory** is being chosen according to che capabilities of the machine. For what concerns Double DQN algorithm, **target update frequency** indicates in how many steps the model's weights have to been replaced with the target's ones (*hard update* method was implemented). 3 values have been selected by me for testing purposes: 100, 200, 400 (see Section 4.2).

These hyperparameters were used:[3]

- `learning rate = 0.001`
- `minibatch size = 64`
- `epsilon = 1.0`
- `epsilon decay = 0.996`
- `discount rate = 0.99`
- `memory = 1000000`
- `target update frequency = 100/200/400`

4 Results

After a sustained training session that lasted for days, I have found remarkable results, that have been properly represented in several plots.

4.1 Deep Q-Learning

Training times were the lowest ever tracked (between 8 and 10 hours). This particular task 2 was completed in 451 episodes. The *rewards curve* is characterized by a high variance over the time. The *average rewards curve* is also characterized by fast, albeit unstable, growth, struggling to achieve the 200 points on average, calculated on the last 100 episodes.

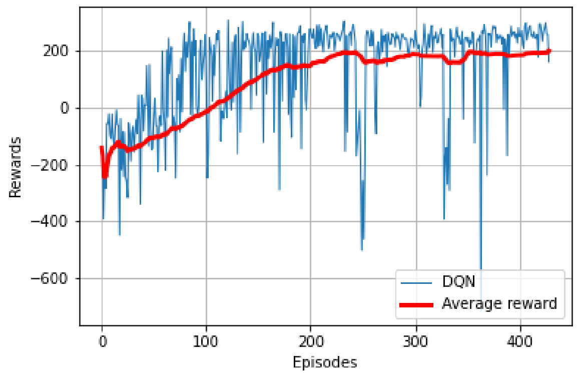

Figure 2: DQN

[3]See footnote 2

I have evaluated the performance of my model through several game sessions, each of them lasting 15 minutes. The trained agent was able to achieve 208 points of mean reward, on average, in line with the environment specifications.

4.2 Double DQN

Training times were very long (24 hours or even more). The task 3 was completed in 387 episodes. The *rewards curve* is characterized by a high variance over the time, but this time the *average rewards curve* is characterized by a smoother trend, converging to the 200 points threshold much faster.

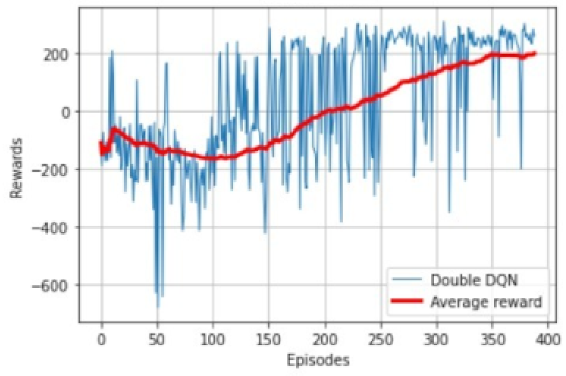

Figure 3: Double DQN with target update frequency = 100

The *target update frequency* that I choose definitely was **100**. Despite the fact the 200 case gave me a *rewards curve* with less variance (and an overall stable *average rewards curve*), the best performance was achieved in the first case: in the evaluation section I got **230 points on average**. Performance in 200 case was on par with the one achieved in the DQN case. Different speech for the 400 case: both the curves were characterized by a high variance, and the performance in the evaluation was terrible (only 120 points on average).

4.3 Comparison

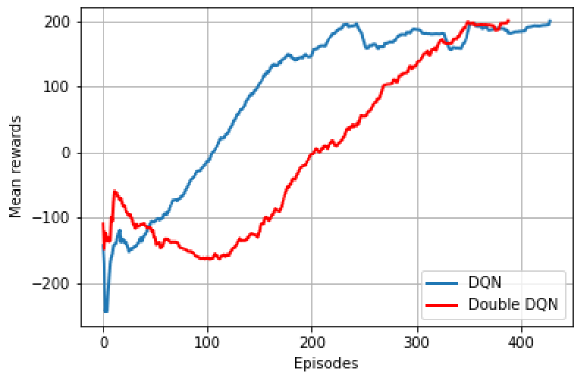

Figure 4: Comparison between the mean rewards curves

5 Conclusions

Both **DQN** and **Double DQN** algorithms were found to be up to the task to be performed, able to converging to the 200 points threshold. Thanks to Double DQN, I was able to achieve better results overall; however the long training times could be seen as a too high price to pay, due to an improvement compared to DQN in the order of 10-15%. In the future the project could be expanded further, implementing the *Soft Update* method in the Double DQN case, comparing the results at the end.

References

[1] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, & Martin Riedmiller. (2013). Playing Atari with Deep Reinforcement Learning.

[2] Ziyu Wang and Nando de Freitas and Marc Lanctot (2015). Dueling Network Architectures for Deep Reinforcement Learning. CoRR, abs/1511.06581.