UNIVERSITÀ DEGLI STUDI DI PADOVA

Reinforcement Deep Learning

Ziliotto Filippo

December 19, 2021

In this assignment we implement and test some neural networks to solve reinforcement learning problems. In particular, the *CartPole-v1* environment is solved training a Deep Q-Learning agent that accepts as input the state representation provided by the environment. Finally another Deep Q-Learning agent is trained to solve the *LunarLander-v2* (the discrete version with a state representation input).

1 INTRODUCTION

[1] [2]Reinforcement learning tasks mainly consist of training an agent A which interacts with an environment ε. Every action performed causes a change in the state of the environment, which might either penalize or favor the agent assigning a reward. The agent naturally seeks to maximize its reward, therefore needs to learn the optimal policy which results in taking the best actions to obtain the largest reward given it finds itself in a certain state. At every timestep t, the agent A performs an action a_t being in a given state s_t , receiving a reward r_t for it. The next state is denoted as s_{t+1}. Formally, the goal of the agent is to maximize a total return G_t , which is weighted according to some discount factor $\gamma \in (0,1)$, that quantifies how much A should care about future rewards:

$$G_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + ... = \sum_0^\infty \gamma^k r_{t+k} \quad (1.1)$$

1.1 Gym Environment

OpenAI Gym is a toolkit for developing and comparing reinforcement learning algorithms, providing several environments emulating classical problems or old-fashion games. In our work we will try to solve:

- **Cartpole-v1**: the goal is to balance a pole attached to a cart, which can move without any friction either left or right. The episode ends when the pole falls, i.e. reaches an angle of 12ř with respect to the vertical, or is considered "won" when 500 time steps have elapsed. We can see it in 3.1.
- **Lunarlander-v2**: the goal is to make a star ship land onto a landing pad located at the center of the screen (x,y coordinates equal to 0). The episode ends when either the starship crashes, comes at rest or exits the screen.

2 METHODS

In Q-learning the agent A learns to associate a value Q, known as long-term reward, to every

[1] All the code and the related files can be found at `https://github.com/ZiliottoFilippoDev/NNDL-21-22`

[2] All the code and the related files can be found at `https://gym.openai.com/envs/LunarLander-v2`

state-action pair $Q(s,a) = E[G_t|s,a]$. Actions performed are sampled according to a policy, denoted by (s), which needs to be learned in such way that the actions chosen will come with the optimal $Q^*(s,a) = maxQ(s,a)$. The network therefore needs to be able to learn Q^*, which obeys the Bellman Equation. However, since computations for it are too expensive and convergence might take too long, one may want to exploit a Deep neural network to directly approximate Q-values. The objective function, to be minimized through gradient descent algorithms, for this problem is:

$$L(\theta) = E_{s,a,r,s'}\left[\left(r + \gamma max_{a'} Q(s', a', \theta^T) - Q(s,a,\theta)\right)^2\right] \tag{2.1}$$

where r is the reward, the second term refers to a **target** network whose task is to approximate $Q(s,a,\theta)$ given the actual parameters and is trained at every iteration, and the third term refers to a policy network updated every n steps and used for action selection. The introduction of the target network allows for a more stable training, and breaks the correlation between the target function and the Q-network.
In addition, in order to make the convergence occurring faster thus improving efficiency, it has been used **experience replay**. It basically acts as a buffer, and allows the learning from past experience once the network has gained "enough experience". Practically, this has been implemented using deque objects.
The actions to perform can usually be chosen according to either one of the following policies:

- **ε-greedy policy**: a non-optimal action is chosen with probability ε, while the optimal with probability $1-\varepsilon$.
- **softmax policy**: the action is chosen according to a softmax distribution of the Q-values, at a certain temperature. Temperature usually decreases over time starting from a "large" value, hence allowing at the beginning less theoretically favorable scenarios, while later choosing most likely actions that are known to return the largest Q-values.

3 Cartpole

```
Observation:
    Type: Box(4)
    Num     Observation               Min                     Max
    0       Cart Position             -4.8                    4.8
    1       Cart Velocity             -Inf                    Inf
    2       Pole Angle                -0.418 rad (-24 deg)    0.418 rad (24 deg)
    3       Pole Angular Velocity     -Inf                    Inf

Actions:
    Type: Discrete(2)
    Num   Action
    0     Push cart to the left
    1     Push cart to the right
```

Figure 3.1: Image displaying the info about the representation space for the $CartPole-v0$ environment

Before proceeding with the description of the Target and Policy networks, which is the same, one should mention that the environment can be described using 4 quantities, namely $(x \in [2.4, +2.4], v \in \mathbb{R}, \theta \in [15degree, +15degree], \omega \in \mathbb{R})$. They are respectively cart position, velocity, pole angle and pole angular velocity.
The Target/Policy networks are implemented as it follows:

- **Input Layer**: 4 units, having the input state 4 elements.
- **First Hidden Layer**: 128 units, *Tanh* activation function
- **Second Hidden Layer**: 128 units, *Tanh* activation function
- **Third Hidden Layer**: 128 units, *Tanh* activation function
- **Output Layer**: 2 units, since the action can be only left or right

[3]Before discussing the training, we should mention that as optimizer it was chosen the *SGD* with no momentum to improve stability and with learning rate 10^2 . In addition, it has been selected the **softmax exploration policy**, with the Temperature parameter varying according to some function. In addition, some Gaussian perturbations is introduce after the game is solved to check how the exploration profile relates to the score. The loss chosen for the problem is the $HuberLoss$ function. During the training, the

[3]All the code and the related files can be found at `https://github.com/ZiliottoFilippoDev/NNDL-21-22`

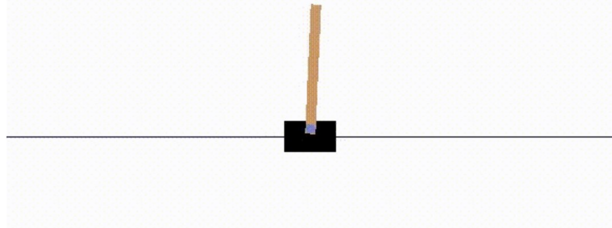

Figure 3.2: Final frame for the solved cartpole game.

agent receives a reward that is incremented by one for every steps in which the pole has not fallen, or the cart has not reached screen boundaries. Moreover, in order to keep the cart as much "centered" as possible, a **linear penalty** is applied when the cart moves away from the center. The assignment request is to improve the convergence speed to a solution for the $CartPole$ environment. Thus, two metrics are defined: the average score and the velocity. The first is used with the idea that, the larger the average score, the more stable should be the network in providing a solution thus being able to maintain it "high". On the other hand, the velocity is defined as the first episode at which the network is able to solve the game, i.e. to reach a score of 500. If a given implementation is never able to do it, this value is set to be equal to the total number of episodes devoted to training: namely 1000.[4]

. With some different trials we searched for the best parameters:

- $profiletype$: 'exponential' being this quantity the Temperature behavior in time
- $T_{initial}$: sampled uniformly between 2 and 7
- n_{update}: [5, 10] number of episodes every which to update the policy network.
- γ: sampled uniformly $\in [0.97, 0.99]$

Finally the best parameters are set in 3.1 and Some plots (3.3, 3.4) regarding this multi-parameter optimization can be inspected. It is interesting to see as the velocity parameter depends mainly on the type of profile chosen for the Temperature and only secondly on . Whereas, the average score gives more importance to nsteps and secondly on .

Indeed one can see as the game can be solved thus reaching 400 steps, with some stability of the score later on. This network has been tested and some videos of it are attached to this report. The network might be able to solve the game even faster, but not with a stable solution.

As said, the network has been optimized to make **convergence faster**. With this parameters the network converges in ~ 400 epochs which is quite fast w.r.t. to the normal implementation viewed in the lab lecture.

Parameters	**Optimal value**
Bad state$_{penalty}$	-20
T$_{Initial}$	4
n$_{update}$	5
Exp. Decay	$\frac{1}{12}$

Table 3.1: Optimal parameters for the Q-network for a ~ 400 epochs convergence.

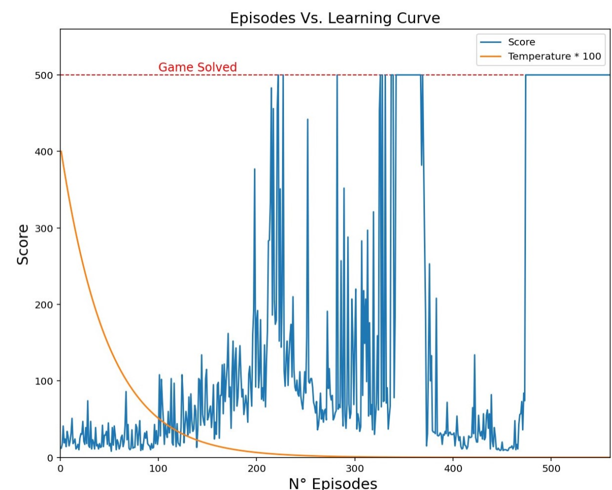

Figure 3.3: Score and temperature profile w.r.t. the number of played games.

In addition to check how the exploration profile changes, w.r.t. the temperature for the softmax exploration policy, when the game was already solved i added a gaussian noise. We can see that when the

[4]All the code and the related files can be found at `https://gym.openai.com/envs/LunarLander-v2`

temperature is high the score decreases and the turns to 500 (game solved) when the noise disappears.

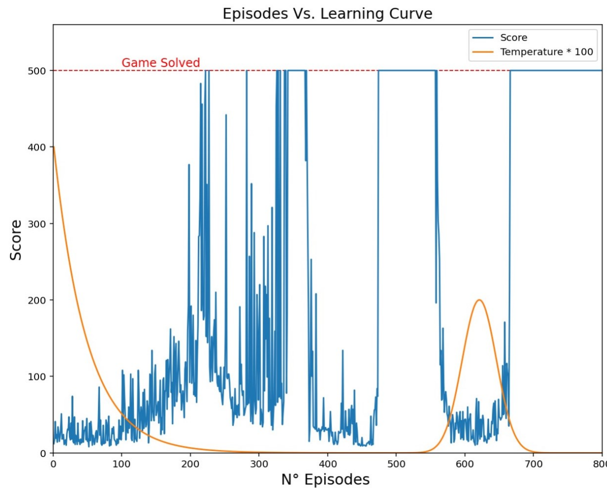

Figure 3.4: Adding a guassian noise at the end of the training to check how it changes the exploration profile.

4 LunarLander discrete

As introduced before, the goal is to make the lander land onto a landing pad that is always at coordinates (0, 0) (see 4.1). The reward for moving from the top of the screen to landing pad and zero speed is about 100, . . . , 140 points. If lander moves away from landing pad it loses reward back. 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 each frame. Solved is 200 points. Landing outside landing pad is possible. Fuel is infinite, so an agent can learn to fly and then land on its first attempt. Four discrete actions are available: do nothing, fire left orientation engine, fire main engine, fire right orientation engine. *This is taken from the summary of the GYM environments library.* [5][6]
The state returned by the environment consists of 8 variables, namely:

- X coordinate ∈ R
- Y coordinate ∈ R
- X velocity ∈ R
- Y velocity ∈ R
- Angle ∈ R
- Angular velocity ∈ R
- Left leg touching the ground (bool)
- Right leg touching the ground (bool)

Since dimensionality has changed, we double the number of units of the second hidden layer with respect to the implementation of the first network, and change input and output accordingly:

- **Input Layer**: 8 units, having the input state 8 elements.
- **First Hidden Layer**: 256 units, Tanh activation function
- **Second Hidden Layer**: 256 units, Tanh activation function
- **Third Hidden Layer**: 128 units, Tanh activation function
- **Output Layer**: 4 units, since the discrete action set has cardinality 4.

The training is performed with different penalties being added to the reward: we want to penalise states with large angle, since a "good" landing has null angle, X coordinate to make the lander stay in the center and finally Y coordinate to make it land in a faster way. Also we applyed a linear penalty to the angle state and the v_y velocity state. All of this penalty have different weights, the two state with the most weights are the X coordinate and the angle. This because we want the lunar lander taking the straight path from the top to the bootm also staying in the right angle position.
The result for such search are visible in 4.2. It has been decided to not proceed further, or to tune "better" the values of such penalties, or even to explore different Temperature profiles, due to the large computational demand even for such simple task. Some attempts were however performed introducing penalties on other variables or parametrizing the actual ones in a different way but they were not promising as the ones presented here. In 5.1 we can see a frame were the lunar lander has failed in the attempt to learn.

[5] All the code and the related files can be found at `https://github.com/ZiliottoFilippoDev/NNDL-21-22`
[6] All the code and the related files can be found at `https://gym.openai.com/envs/LunarLander-v2`

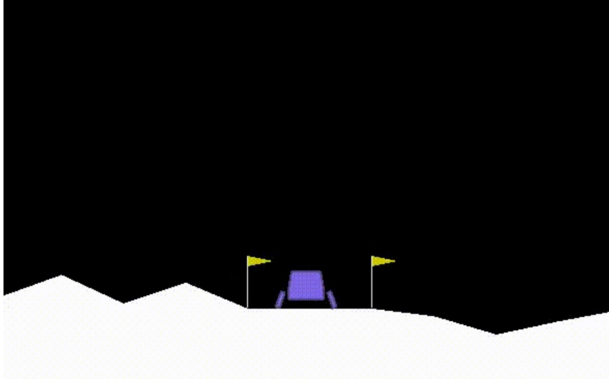

Figure 4.1: Final frame of the lunar lander network when it has been solved.

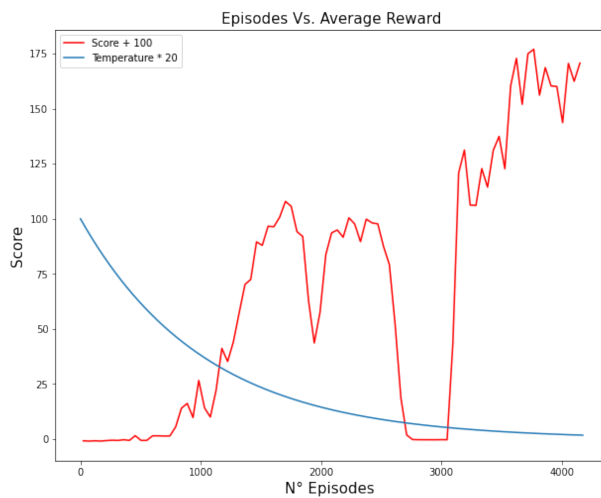

Figure 4.2: This is the score w.r.t. the number of epochs. We see that the solving takes time but after a while the score and solving is consistent.

Further inspecting at the videos attached, it seems like the ones with low score do not end up with the lander crashing, rather than with the latter wasting fuel to reach the central point. This might suggest some more optimization is needed, though the training has been successfully performed: the lander is now able to land onto the landing pad in most of the cases without crashing. [7]

5 Appendix

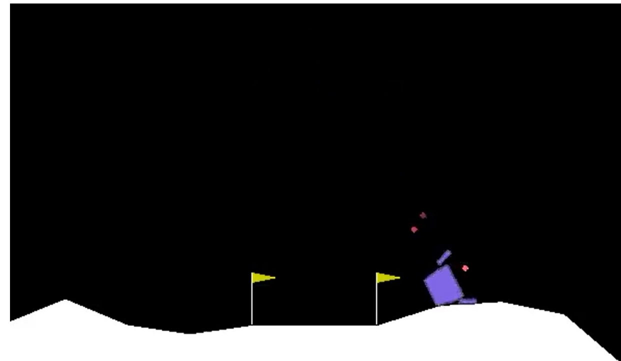

Figure 5.1: Funny example of a failing for the lunar lander environment.

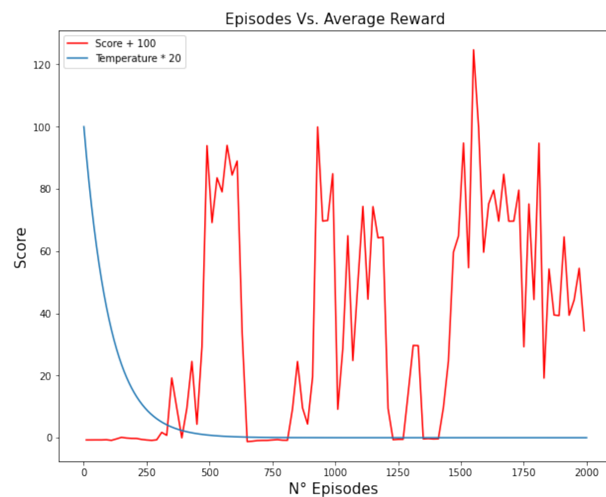

Figure 5.2: Another exploration of the score for the lunar network. We see that with 200 epochs the network does not converge yet.

[7] All the code and the related files can be found at `https://github.com/ZiliottoFilippoDev/NNDL-21-22`