Deep Reinforcement Learning

Tommaso Faorlin (2021857)

Università degli Studi di Padova
tommaso.faorlin@studenti.unipd.it

I. Introduction

In this laboratory report we solve the third homework of the course in Neural Networks and Deep Learning at the University of Padova. The topic discussed is deep reinforcement learning, and we are going to solve the `CartPole-v1` and the `LunarLander-v2` environments with Deep Q-Learning algorithm.

II. Deep Q-Learning (DQL)

In this section we briefly illustrate the RL technique used to solve the tasks. In the final part of the course we have dealt with the branch of machine learning concerning reinforcement learning. Under this framework (that personally I found to be the most intriguing one) at each time step t, an agent $\mathcal{A}$ is in a state $s_t \in \mathbb{S}$, interacts with the environment performing actions $a_t \in \mathbb{A}$ and moves to a new state $s_{t+1} \in \mathbb{S}$ obtaining a reward r_t. The final goal of the player is to choose the correct actions in order to maximize a cumulative reward $\mathcal{R}_t$ defined as

$$\mathcal{R}_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots = \sum_{i=0}^{T} \gamma^i r_{t+i}. \tag{1}$$

where T is the last instant of the game and $\gamma \in [0,1]$ is defined as *discount-rate*: when $\gamma \sim 1$, the agent is more interested in events in the distant future. We used the word *player* because the problems in this branch of machine learning are often represented in the form of games, and the player wins the game when he accumulates the maximum reward. In general, the agent take actions following a given policy π, that is a probability distribution over the set of actions. In our implementation, we are going to try two kind of policies:

- **ε-greedy policy.** The agent chooses with probablity ε a non-optimal action or the best action otherwise. The value of ε is tuned in a custom way.

- **Softmax policy** The agent samples actions from a temperature dependent softmax distribution, weghted on the values of the successive state-action pairs. The initial value of T is sampled between 2 and 7, and then decays exponentially. For $T \to 0$ the agent always selects the best action.

In a Q-Learning algorithm, $\mathcal{A}$ learns to associate the expected cumulative reward to each possible state-action pair $Q_\pi(s_t, a_t) = \mathbb{E}(\mathcal{R}_t | s_t, a_t, \pi)$ under a policy π. From this concept, we define the optimal Q-value function as the maximum return achievable after seeing some sequence s_t and then taking some action a_t: $Q^*(s_t, a_t) = \max_\pi \mathbb{E}(\mathcal{R}_t | s_t, a_t, \pi)$. This is what the agent needs to learn, and in particular this action-value function obeys the Bellman equation:

$$Q^*(s_t, a_t) = \mathbb{E}_{s'}(r_t + \gamma \max_{a'} Q^*(s', a') | s_t, a_t) \tag{2}$$

that is exactly what we aim to approximate with a Deep Neural Network (DNN).

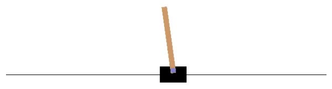

Figure 1: *CartPole-v1 environment.*

III. CartPole

I. Description of the game

A pole is attached by an un-actuated joint to a cart (the agent), which moves along a frictionless track (snapshot in Fig. 1. The actions space is bidimensional: we can either push a cart to the left (0) or to the right (1). A state of the environment is a four dimensional vector containing the cart position, velocity, the pole angle from the vertical and its angular velocity. So: $\mathcal{A} = [0, 1]$, $\quad \mathbb{S} = (x, \dot{x}, \theta, \dot{\theta})$. The pendulum starts upright, and the goal is to prevent it from falling over. A reward of $+1$ is provided for every time step that the pole remains upright, and the game is considered solved after 500 time steps. The episode ends when the pole is more than 15 degrees from vertical, or the cart moves more than 2.4 units from the center.

II. Method

We use a DNN to compute the Q-values for the actions given a state. The architecture is very simple and consists of two fully-connected layers of 128 neurons, both activated with a Tanh function. We are going to minimize the following objective function, that depends on the parameters $\boldsymbol{\theta}$ of the DNN used:

$$\mathcal{L}(\boldsymbol{\theta}) = \mathbb{E}[(r + \gamma \max_{a'} Q(s', a', \boldsymbol{\theta}) - Q(s, a, \boldsymbol{\theta}))^2]. \tag{3}$$

To have a more stable training and to facilitate the convergence to a solution we use two clever techniques: a replay memory and a target network. Let us briefly discuss them:

- **Replay memory.** The model is trained on tuples of (s_t, a_t, r_t, s_{t+1}): if we imagine to stay within a single episode and to train the network on consecutive states, the latter will be highly correlated and the model will thus specialize on each single episode, losing its generalization capabilities. To overcome this problem we store a total of 10000 tuples inside a replay memory, and we train the network on batches coming from this set.

- **Target network.** In Eq. 3, we may notice that the parameters $\boldsymbol{\theta}$ appear in two places at once. So, we detach the two dependencies by using two different (but with the same exact architecture) networks: the online network and the target network T. The former is updated at every step, while the second is *freezed* for an amount of `target_net_update_steps` and then synced with the online version. In the end $\max_{a'} Q(s', a', \boldsymbol{\theta}) \to \max_{a'} Q(s', a', \boldsymbol{\theta}^T)$ in Eq. 3.

To optimize the weights of the network $\boldsymbol{\theta}$ we use the Stochastic Gradient Descent (SGD) with null Nesterov momentum.

III. Results

After a random grid search over 10 models, where we vary $\gamma \in [0.9, 0.99]$ and `lr`$\in [0.01, 0.1]$ a reasonable set of parameters is the one reported in Tab. 1.

We then show in Fig. 2 the exploration profiles for the two different policies and the score relative to them. To clarify the trend of the score we will always report a moving average of the score computed over 30 iterations. When the score reaches 500 (+1 for 500 episodes) the game is solved.

`min_training_samples`	γ	`lr`	`opt`	`momentum`	`batch_size`	`target_net_update_steps`
1000	0.98	0.0278	`SGD`	0	128	5

Table 1: *Hyper-parameters for the best model.*

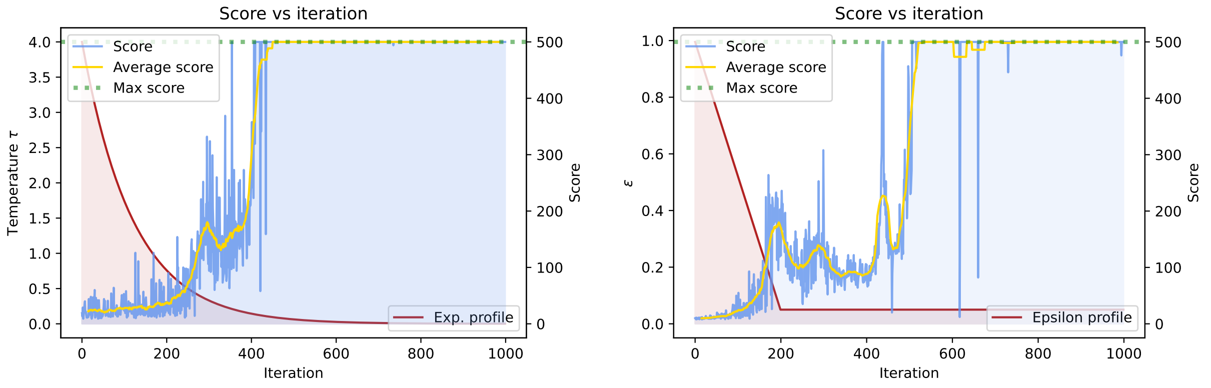

Figure 2: *(**Left**). We show the best result for a **softmax** policy. The game is solved in slightly more than* 400 *episodes (less than during the lab). (**Right**). The same task is addressed with an ε-**greedy** policy where the exploration phase consists of* 200 *iterations and then there is the exploitation phase with* $\varepsilon = 0.05$*. In this second case, we obtain a worse result.*

III.1 Gaussian kinks

If we train again the model with the same parameters, but we add to the exploration profile a gaussian kink when the game is solved we see that the score rapidly decreases (the agent is exploring and thus taking random actions). The maximum reward is regained right after the kink because the agent is robust.

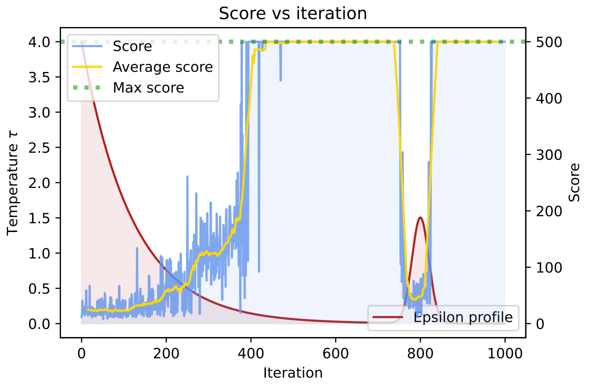

Figure 3: *Test for the robustness of the model.*

IV. Discrete Lunar lander

I. Description of the game

Compared to the game described above, this is slightly harder to solve since both states and actions are defined by more variables. A state here is composed of the coordinates of the vessel (the agent) in $\mathbb{R}^2$ space, the linear velocities in the two directions, the angle from the vertical, the angular velocity and two booleans corresponding to whether the two legs touched or not the ground. The actions are four: do

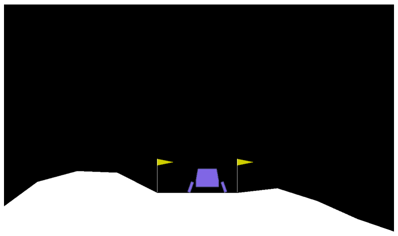

Figure 4: *LunarLander-v2 environment. Here we show the last frame of one valid episode.*

nothing, fire left orientation engine, fire main engine, fire right orientation engine. So, $\mathcal{A} = [0,1,2,3], \quad \mathbb{S} = (x, y, \dot{x}, \dot{y}, \theta, \dot{\theta}, l_1, l_2)$.

The aim of the game is to land a rover on a landing pad at coordinates $(x, y) = (0, 0)$. The game finishes when the lander crashes or lands safely at rest (with engines off). Each leg ground contact is a positive $+10$ reward, when the main engine is fired is -0.3, while is -0.03 for the side engines. If the lander moves away from the landing pad it loses reward. If the lander crashes, it receives an additional -100 points. If it comes to rest, it receives an additional $+100$ points.

II. Method

The method is the same as described above: we do not change the architecture of the network used for the Q-values but we choose `Adam` as optimizer.

III. Results

After a manual search of the hyper-parameters for a softmax policy, we report the best set in the following table. We notice that the best value for γ is almost 1, and this is motivated by the fact that the great part of

`min_training_samples`	γ	`lr`	`opt`	`batch_size`	`target_net_update_steps`
500	0.99	0.001	`Adam`	64	5

Table 2: *Hyper-parameters for the best model.*

the reward is achieved only at the end of the episode, when the vessel touches the ground. After fixing these parameters, we try to change policy and tweak the reward function.

III.1 Softmax policy

Looking at Fig. 5, the graph on the left is the best result obtained with a softmax policy, while the one on the right is what we obtain with the same policy but also adding a bonus if the vessel is descending. It happened that the spacecraft floated at half height, without landing and therefore consuming fuel. So, with the *bonus* we would like to discourage this behavior. Practically, knowing that `state[3]` is $\dot{y}$:

```python
next_state, reward, done, info = env.step(action)
descent_bonus = 1.5*np.abs(state[3]) if state[3] <0 else 0
reward = reward + descent_bonus
```

We have tried to further modify the reward function in our study, by adding *malus* terms proportional to the angle or distance from the center but we are not reporting them. In the end, we can consider the game solved because the score is fluctuating at around 200.

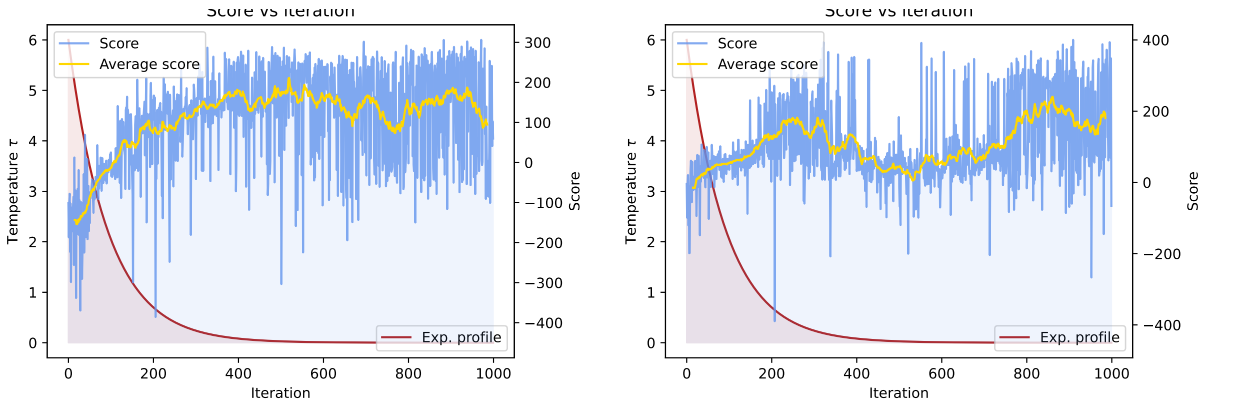

Figure 5: *(**Left**). Best result obtained with a Softmax policy (**Right**). Same, but with a slightly modified reward function.*

III.2 ε-greedy policy

We also study the model with the same parameters as before but with a greedy policy. We define two custom profiles, basically related to two different balances between exploration and exploitation and the results are shown in Fig. 6. On the left we have a profile for ε that gradually decrease, allowing the agent to explore random actions with low probabilities as it goes along. Instead, on the right, the exploration region ends in 100 iterations. The model on the left is the one that, at the end of the day, is giving the highest score on the test out of 10 episodes (results reported in Fig. 7).

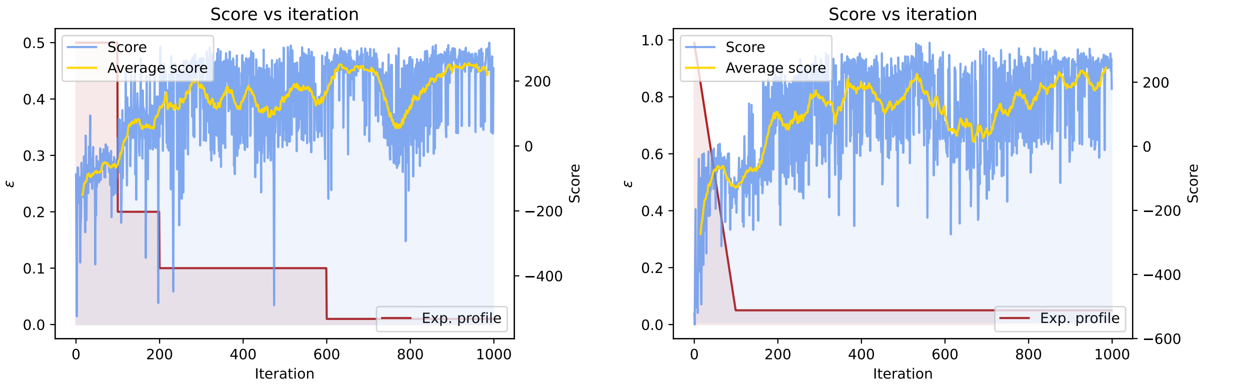

Figure 6: *Score with two different profile for ε in a greedy policy. The results are almost similar, we can notice how the score on the left is slightly more stable in the last iterations.*

```
EPISODE 1 - FINAL SCORE: 278.870965008212
EPISODE 2 - FINAL SCORE: 251.9096074879601
EPISODE 3 - FINAL SCORE: 169.26960768742705
EPISODE 4 - FINAL SCORE: 304.42948625379864
EPISODE 5 - FINAL SCORE: 285.2705633085526
EPISODE 6 - FINAL SCORE: 274.405052228632
EPISODE 7 - FINAL SCORE: 248.29120790289795
EPISODE 8 - FINAL SCORE: 253.24516193957584
EPISODE 9 - FINAL SCORE: 266.81426754518
EPISODE 10 - FINAL SCORE: 278.8202880584146
```

Figure 7: *Results of a test consisting of 10 episodes.*

V. Conclusions

This has been a challenging assignment but also the most interesting one. With more time and more computational power we can further improve the models and see how to make them converge even faster. We also produce a `.gif` (exploiting the code found here) for the LunarLander and the best model that solves it!