Deep Reinforcement Learning

Marco Ballarin 1228022
marco.ballarin.6@studenti.unipd.it

Abstract

In this report we will review the deep reinforcement learning framework. We will see that it is a really tough task, due to its dynamical nature. In particular, we will adress the environment of the Cartpole, the Cartpole using the screen as input and the Lunar Landing game from the gym package. We will focus on the convergence speed and on tricks to improve the learning procedure.

1 Introduction

In this report we will tackle two different problems using the deep reinforcement learning framework. This means that we will have an environment where a agent can perform actions, which will change the state of the environment, and receive a reward dependent on the new state. The reinforcement learning problems are usually represented like a game, and so if the agent solves the environment we can say that he wins the game. We will use a neural network to control the actions of the agent, and optimize it to win the environment. We will use the environments provided by OpenAI in their python library, namely gym [1].

The report will be divided as follows:

1. Introduction, where we introduce our work;
2. Cartpole, where we try to balance a pole on a cart knowing its position, velocity and the pole's angle and angular velocity;
3. Cartpole with pixels, where we tackle the same problem as above, but using as input of the neural network only the rendered image of the cartpole. We will see that the learning will become really more difficult, since the network should infer the information about the state and then learn to control the agent;
4. Lunar Landing, where we tackle a different problem: manage to land with a spaceship using its engines. The description is a little more involved of the one above, and we will so go into details in Section 4;
5. Appendix, where we will report most of the images;

2 Cartpole

2.1 Methods

The cartpole environment from the gym library consists of a cart with a pole, moving on a 1-dimensional rail. The observation space of the environment O and its action space A are described as follows:

$$O = [\text{Cart position, Cart velocity, Pole angle, Pole angular velocity}] \quad A = [\text{ Move left, Move right }]$$

The aim of the agent is to keep the pole balanced for as long as possible, but the trial is considered solved at 500 time steps. For each time step that the pole is balanced the agent gets +1 to the reward. We show an example of the rendered environment in Figure 4.

In Q-learning the agent learns to associate a value to each state-action pair, which takes into account both the immediate reward for that action and the future rewards. In particular, we define an update step as:

$$Q(s_t, a_t) \rightarrow Q(s_t, a_t) + \lambda \left(r_t + \gamma \max_a Q(s_{t+1}, a_t) - Q(s_t, a_t) \right) \tag{1}$$

where s_t, a_t are the state and the action at time t, λ is the learning rate and γ the discount factor that takes into account how much future rewards matter. Notice that in the update we take into account which is the maximum Q value of the actions in the state at the next time step. Given the Q-values we then need a policy, i.e. a way of selecting which action to perform. We will use a softmax policy, i.e. select which action to perform based on their Q-values. It is however important to explore the environment at the initial stages of the learning: if we always select the best action when we do not know the full system we may be stuck in local minima. We so use a softmax with temperature, setting an exploration profile, i.e. an evolution of the temperature T such that at the initial time we select random actions ($T > 1$) and towards the end of the simulation we select the best action ($T \rightarrow 0$).[1] In particular, we will use an exponential decay for the exploration profile. We will also randomly insert gaussian addition to the profile, to see their effect.

We will use a neural network, shown in Figure 1, to calculate the Q-values. There are, however, some problems in this approach. The training dataset is built incrementally, the samples are not identically independently distributed and the target function is non-stationary. We so introduce two workarounds:

- A Replay memory. We store tuples of (state, action, reward, next state) in a memory buffer, such that in the training we can sample mini-batches from this buffer and "replay" those states. This is really important, because we are no longer forced to use subsequent time step in the training, decreasing the correlation between samples.
- A Target network. We will not use the same network for evaluating $Q(s_t, a_t)$ and $Q(s_{t+1}, a_t)$ in the update rule. We will instead use a target network for the latter, which weights are updated from the first network, called policy network, each $\tilde{n}$ episodes. This helps for the non-stationarity of the target function.

For the optimization of the network we will use as optimizer the stochastic gradient descent (SGD), but without momentum. This is because, differently from other deep learning frameworks, in reinforcement learning it is possible that the optimization direction changes quickly, due to the dynamical nature of the problem.

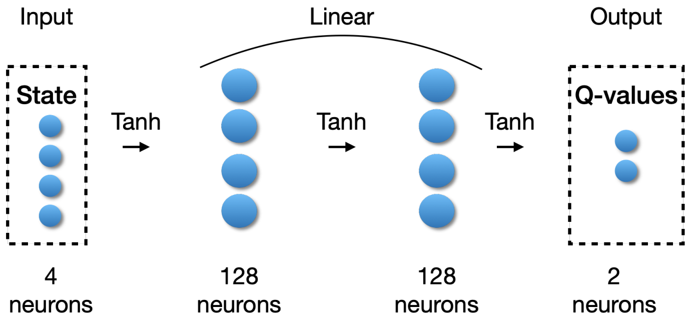


Figure 1: Neural network for the evaluation of the Q-values given the state.

In the reward function it is added a linear penalty proportional to the distance from the center of the screen, due to a problematic behavior of the agent that tried to "exit" from the screen.

We will try to speed up convergence as much as possible, in particular optimizing:

- the exploration profile, of which we will change the starting temperature. As cited before, we will add gaussian increment of temperature at random position in the profile with probability $p = 0.3$, to understand its effect on the optimization. The exploration profile will be made of 1000 points;

[1]This terminology for the temperature is borrowed from physics.

- The discount rate γ;
- The target update time $\tilde{n}$;
- The learning rate λ;
- The mini-batch size.

2.2 Results

We immediately understand that the performances of the algorithm are strongly dependent on the hyperparameters. We can see in Figure 5 the hyperparameter search, where the first_solved axis is defined as the first time at which the agent performs at least 490 points. If the agent is not able to solve the environment then we assign a first_solved value of 1000. We can see that the best performing agents have a high learning rate λ, a high discount rate γ and a frequent update of the target network. To better understand the evolution of the training and the effect of the exploration profile we plot the exploration profile and the training score, i.e. the score along all the points of the exploration. Obviously, they have different magnitudes, and so we will refer to the left axis for the exploration profile and to the right axis for the training score. The training score oscillates a lot along the evolution. We will so take an average each 20 time step, taking as relative time step the central one. We see in Figure 6 the best performing hyperparameters, in Figure 7 a good hyperparameter set and in Figure 8 a set which is not able to solve the environment. The figures show clearly that, when the hyperparameter set is optimal, the score abruptly increases as soon as the temperature approaches zero, i.e. when the agent starts to choose the best action. This means that the learning is really fast. A bad choice can instead damage the learning so much that the agent is never able to solve the environment.

3 Cartpole with pixels

3.1 Methods

The aim of this section is again to solve the environment of the Cartpole from the gym library. However, we will not use the observation space described in the previous section. We will instead use the cartpole displayed in a screen. This task is far from easy, since the agent now have to infer the previous observation space from images. We first decrease the size of the observation space, which is a 600×400 RGB image, as follows:

- We transform the image in grayscale, passing from 3 channels to 1;
- We crop the image such that we have a small amount of empty screen, while still leaving to the cart the possibility of moving and being observed. In particular, we crop 150 pixels from the bottom, 80 from the top, 200 from the left and 200 from the right.

We so end up with a 200×170 grayscale image, as shown in Figure 9. Still, we do not have any means of understanding information about the velocity from a single image. We so concatenate 4 subsequent images as input of the network.

Since we fed images to the agent, in this case we decided to use a convolutional neural network, which architecture is presented in Figure 2. It is important to notice that in the network we make plenty of use of the stride, which reduces the size of the image.

The firsts attempts were so difficult and low-performing that we thought of a trick to implement, divided into three main points:

- Run the normal training, as in the previous section, but using the new observation space. Store the couples $[obs, st]$, where obs is given by the 4 subsequent images defined above and st is the state defined in Section 2;
- Train a supervised convolutional neural network to predict st given the obs, with the same structure shown in Figure 2, but where the output layer has 4 neurons instead of 2;

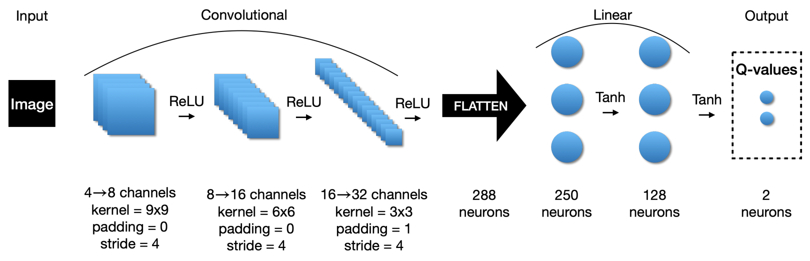


Figure 2: Architecture of the convolutional neural network used as agent.

- Perform transfer learning, by freezing the weights of the supervised network and attach to it a network with the same structure of Figure 1, and run the training again.

This approach is particularly challenging due to the size of the *obs*, which quickly filled the RAM. It is however an interesting idea, and so we decided to present it anyway.

3.2 Results

We used the best hyperparameter set found in the previous Section. As we can see in Figure 10 the network is able to learn a strategy to improve its score, thanks to all the tricks defined above. However, it is not able to solve the environment. We so decided to try to implement the procedure described above, adding a supervised part. The results of such a procedure are shown in Figure 11. As we can observe, the results are worse than in the previous case, even if there is a learning. These results can be due to various problems of this task, first of all the scarcity of training samples for the supervised part, on the order of 10 000, and not iid. Since the supervised part was not predicting the correct state the additional network was not able to produce meaningful q-values.

We finally try to apply the same trick as above, but without freezing the weights of the previously trained network. The outcome is shown in Figure 12. The learning is very unstable, and even if we manage to reach peaks of 200 it is not clear if the performances really improve, or it is just a fluctuation.

4 Lunar Landing

4.1 Methods

We choose to study the Lunar Lander environment from the gym library. It is slightly more difficult than the cartpole, since it has a larger observation space. The observation space is the following:

$$O = [x, y, v_x, v_y, \theta, \omega, l_l, l_r]$$

where x, y are the spatial coordinates, v_x, v_y the velocities, θ the angle of the lander and ω its angular velocity. l_l and l_r are boolean variables and are True if the left (right) leg is in contact with the ground. The action space is instead:

$$A = [null, l_e, c_e, r_e]$$

where *null* is “do nothing”, l_e is “fire the left engine” and the last two action are respectively fire the central and right engine. The fuel is infinite.

The Landing pad starts always at coordinates $(0, 0)$. The reward for moving from the top of the screen to landing pad with zero speed is about $100 - 140$ points. If the lander moves away from landing pad it loses reward back. An episode finishes if the lander crashes or comes to rest, receiving additional -100 or +100 points. Each leg which makes ground contact is +10 reward. Firing main engine is -0.3 points each frame. The environment is considered solved if the agent reaches 200 points. It is possible to land outside the pad. We show an example of the rendered environment in Figure 13.

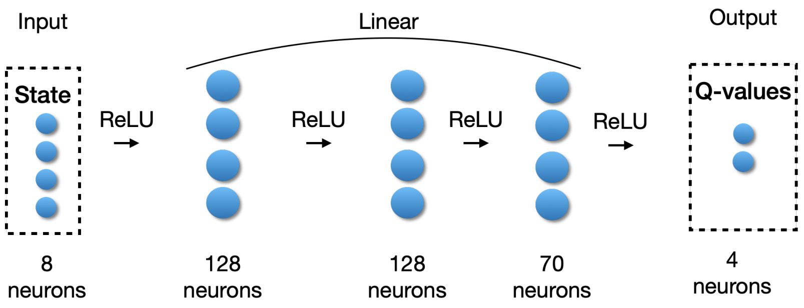


Figure 3: Architecture of the network used in the Lunar Lander environment.

We will use all the tricks explained in Section 2. Since the environment is slightly more difficult than the previous one, we will add another linear layer in the network, using the architecture presented in Figure 3.

4.2 Results

This environment was not trivial to solve. Indeed, to speed up convergence we tried to tweak the reward function. First, we gave some linear malus if the lander moved away from the center of the screen, i.e.

$$malus = -\eta|x|$$

Even though the lander remained in the center of the screen it learned to fly indefinitely. Instead, adding a linear bonus for going downward, i.e.

$$bonus = -\alpha v_y \theta(-v_y),$$

where $\theta(t)$ is the Heaviside function and *alpha* is a parameter set to 1.5, helped in the convergence. In particular, we observe the average score and the exploration path in Figure 14, and we can observe that it solves the environment at about 800 steps. We can also observe that the score increases significantly at 600 iterations after a plateau, in correspondence to the end of the gaussian addition. This suggests that the increase in the temperature helped to discover a new strategy that was the correct one to solve the environment. We can so state that increasing the temperature at later stages of the learning may be useful to improve the final score.

References

[1] Greg Brockman et al. *OpenAI Gym*. 2016. eprint: `arXiv:1606.01540`.

5 Appendix

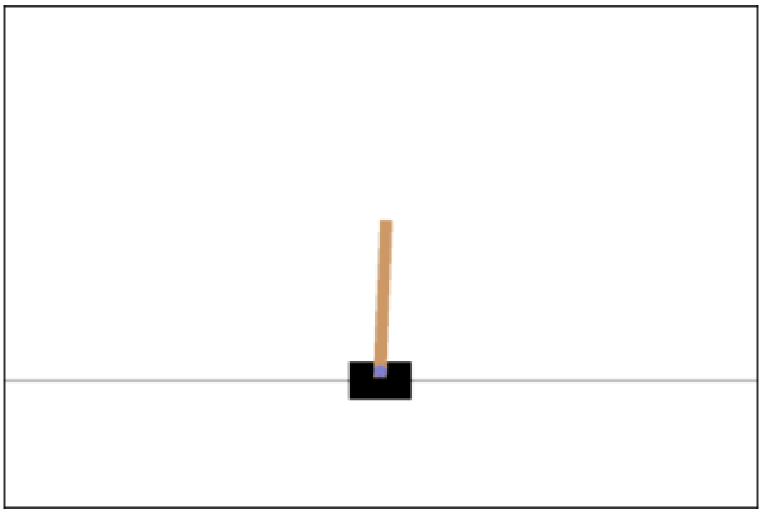


Figure 4: Example of the rendered environment for the cartpole. We can notice the cart in black, the pole in brown and the moving rail represented by black line.

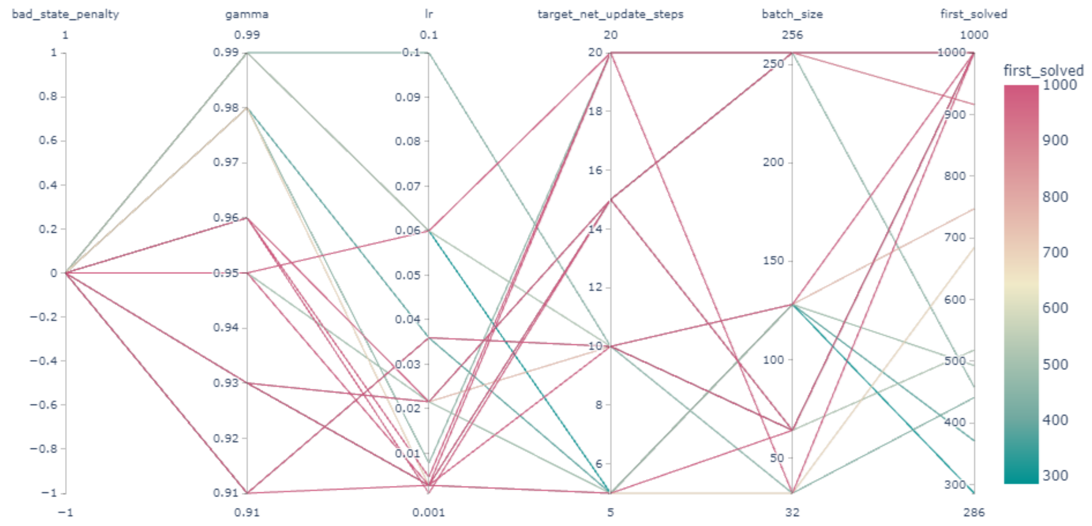


Figure 5: Hyperparameters search, where the color of the line is proportional to the first_solved field. We stress that the best set of trials are the one with lowest first_solved.

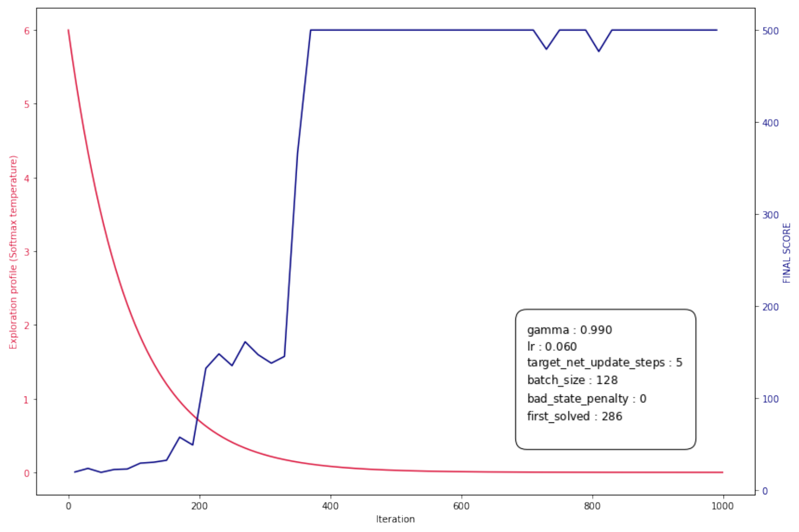


Figure 6: In red the exploration profile and in blue the training score for the best performing hyperparameter set. The training score increases immediately after the temperature approaches 0.

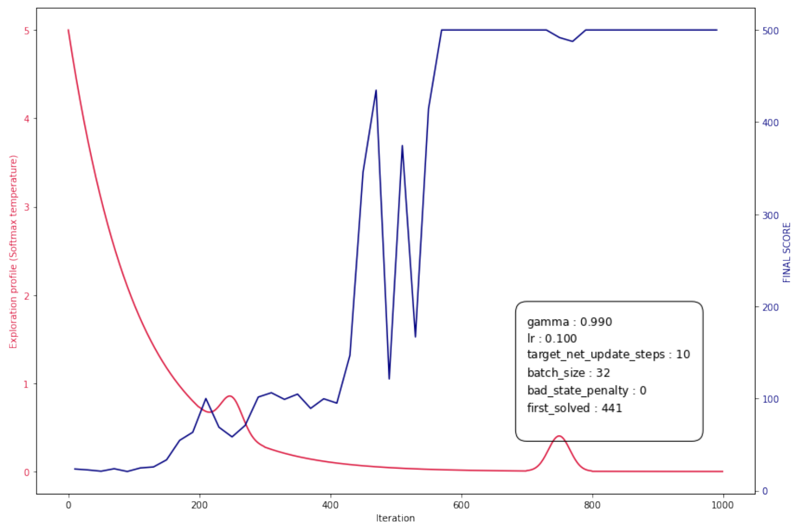
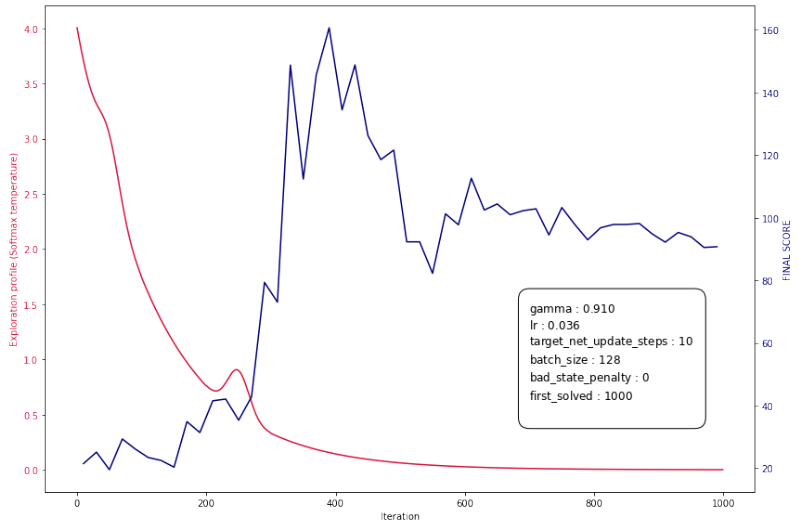


Figure 7: In red the exploration profile and in blue the training score for a well performing hyperparameter set. We see that, even if we have an increase in the temperature towards the end, the agent still performs a very good score.



gamma : 0.910
lr : 0.036
target_net_update_steps : 10
batch_size : 128
bad_state_penalty : 0
first_solved : 1000

Figure 8: In red the exploration profile and in blue the training score for a hyperparameter set not able to solve the environment. The training score stabilizes around 100, which is far under the maximum score.

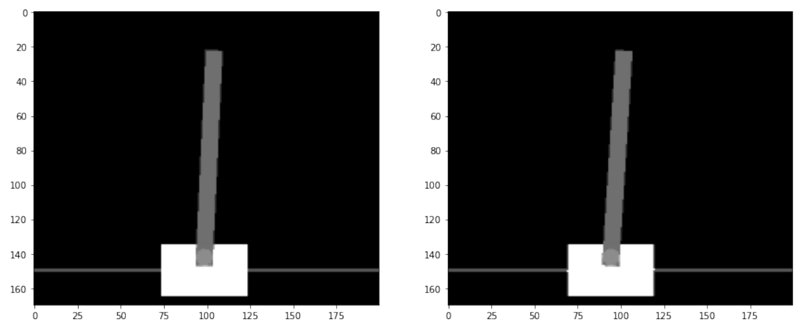


Figure 9: Samples of the image given as an input to the network. We observe that they are cropped and putted in a grayscale.

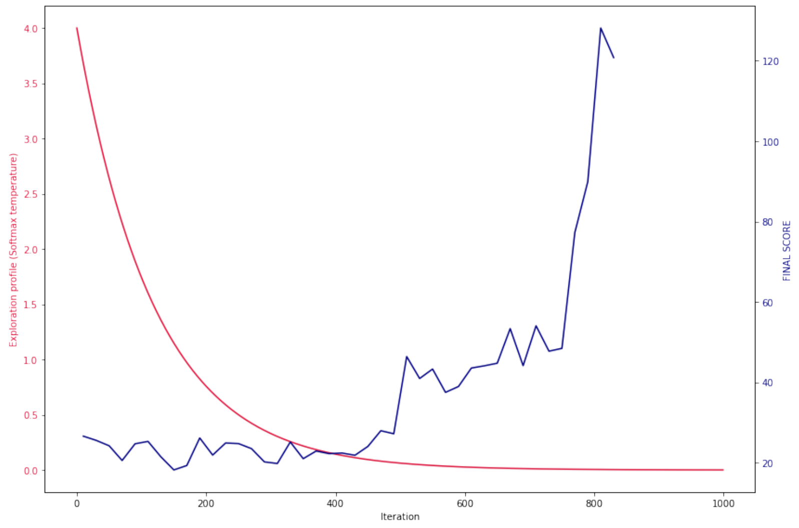


Figure 10: In red the exploration profile and in blue the training score before the application of the supervised network for the learning from pixels. The blue curve ends prematurely due to a memory leak. We can see that, even if the environment is not solved, the agent learns a strategy, since the average score increases significantly.

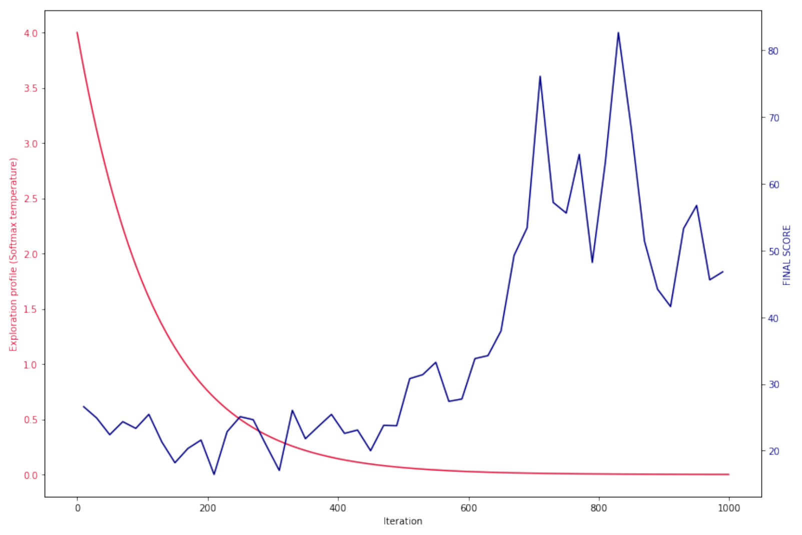


Figure 11: In red the exploration profile and in blue the training score before the application of the supervised network for the learning from pixels. The score is lower than the one in Figure 10. This is due to problems in the supervised learning task, such as the low number of training samples.

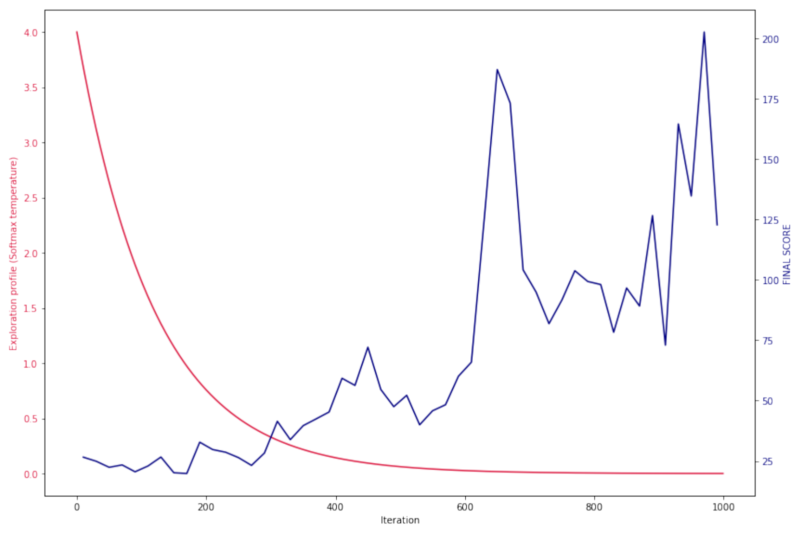


Figure 12: In red the exploration profile and in blue the training score, starting the learning from the supervised network without freezing the weights. The score is higher than the one in Figure 10, but it is really unstable and it so not a reliable improvement.

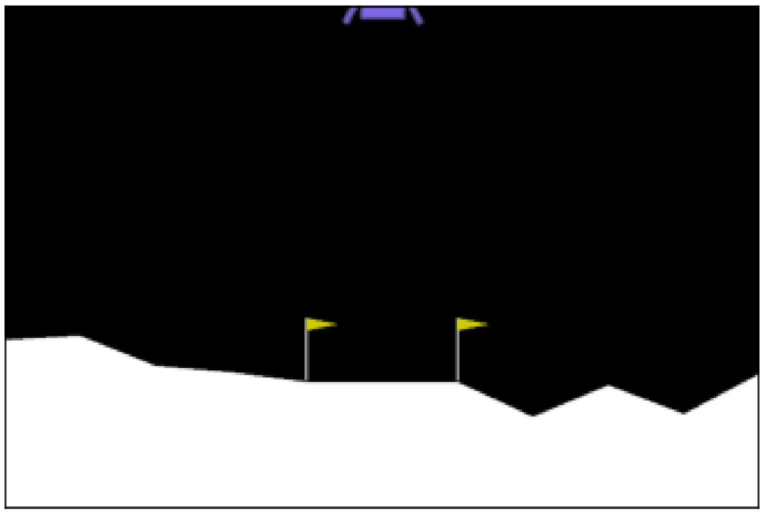


Figure 13: Example of the rendered environment for the lunar lander. We can notice the landing space delimited by the flags and the lander in violet on the top center.

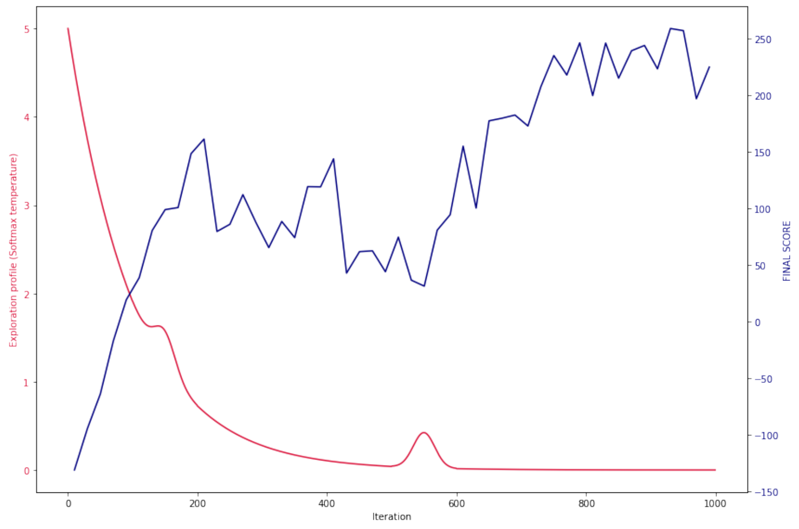


Figure 14: In red the exploration profile and in blue the training score for the LunarLander environment. The environment is considered solved with a score of 200, which is reached at around 800 iterations. We can also observe that the score increases significantly at 600 iterations after a plateau, in correspondence to the end of the gaussian addition.