Neural Networks and Deep Learning - Homework #3

Chioccarello Stefano - 2011656
stefano.chioccarello.1@studenti.unipd.it

January 14, 2022

This report describes the third homework of the Neural Networks and Deep Learning course about Deep Reinforcement Learning (RL). The goals, methods and results achieved in this activity are described in detail in the following.

1 Introduction

The purpose of this homework was to implement and test deep architectures for solving RL problems. In order to make a good analysis of the work done, it can be useful to point out some of the most important aspects of this subfield of Machine Learning.
The main elements that define a RL problem are the following:

- *Agent*: the entity that is trying to solve a given task
- *Environment*, the domain in which the agent finds itself and with which interacts
- *State*: a set of information about where the agent finds itself inside the environment (it could be the spatial position, or some parameters that describe its current situation etc.)
- *Action*: the action taken by the agent given a certain state
- *Reward*: the outcome of the interaction between the agent and the environment; it can be both a positive or a negative reward.

The goal of the agent is to maximize the cumulative reward over time, and the training is done over repeating episodes in which the agent sees rewards coming from the interaction with the environment so that it adjusts its acting decision process in order to maximize an objective function.
One important difference with respect to other learning problems such as Supervised and Unsupervised learning is that, in this case, there are no a priori data available to the learner: the data that the agent uses to learn come directly from its interaction with the environment. From this fact it is obvious that it will take a lot of iterations to learnb because most of the times the agent will fail at reaching the goal, but overtime it will learn which is the best set of actions to take in each state. This introduces the concept of *policy*, usually indicated with π, which is substantially a function that describes the behaviour of the agent in terms

of actions and states.
The best policy for the agent would be to choose for each state the best action that maximizes the immediate reward. However this may lead to sub-optimal solutions due to the so called *Exploration-Exploitation Dilemma*. Briefly, the problem is that, if the agent always chooses the best action that until now gave him the best reward (*greedy action*), which is the exploitation part, it may happen that he will skip some actions that may lead him to better future rewards (exploration), but also to worse rewards.
This trade-off is regulated by introducing some stochasticity into the policy, in order to not always act greedly. One usual choice is the ϵ-*greedy* policy, which selects the action in the following way:

$$\begin{cases} \text{greedy action} & \text{with probability} \quad 1-\epsilon \\ \text{non-greedy action} & \text{with probability} \quad \epsilon \end{cases} \tag{1}$$

with small ϵ. This allows to ensure a bit of exploration in the range of possible actions of the agent. The higher is ϵ, the more frequent will be the non-greedy action.
Another relevant policy is the *Softmax* policy in which, instead of picking a random action with probability ϵ, the action is chosen according to a softmax probability distribution of the Q-values with temperature τ:

$$p(a_t|s) = \frac{e^{-\frac{Q(s,a_t)}{\tau}}}{\sum_a e^{-\frac{Q(s,a_t)}{\tau}}} \tag{2}$$

Note that, as $\epsilon \longrightarrow 1$ and $\tau \longrightarrow \infty$, the action becomes totally random, while as $\epsilon \longrightarrow 0$ and $\tau \longrightarrow 0$ the action gets greedy.
However, to consider a RL problem solved, it is necessary to find the best approximation of the so called *Q-value* function, which is a function $Q(s, a)$ that is a prediction of the future rewards starting from a given *(state, action)* pair and according to the policy π.
There exist many approaches to do such thing, but the focus is pointed to Deep Reinforcement Learning algorithms, namely the use of deep networks to approximate the Q-function. In doing that, in this homework the *Experience Replay* mechanism is exploited. The key idea of this approach is that the experience gaining phase is separated from the training phase. Instead of approximating the Q-function after each *(state, action)* pair as they occur during simulation or actual experience, the system stores the data in tuples composed as $[state, action, reward, next\_state]$. This allows to store previous experiences that can be used to learn multiple times, and it has better convergence properties and benefits in terms of computational cost.

The goal of this homework was to implement and test a Deep Network for solving an environment from the *GymAI* library, that is a collection of RL enviroments among which there are some Control theory problems from the classic RL literature.
Moreover, it was requested to analyze the exploration profiles with the used policy to see how the choice of that variable would change the learning curve.

2 Methods

Starting from the experience replay mechanism, the implementation was done by means of the *deque* object form the Python *collections* library. In fact, this method replaces older elements in the experience buffer with the newer ones.
The function approximator employed in this homework is a rather simple multi-layer feed-forward network composed as follows:

- Fully connected layer with *in_features* = *state_space_dim* and *out_features* = 128
- Fully connected layer with *in_features* = 128 and *out_features* = 128
- Fully connected layer with *in_features* = 128 and *out_features* = *action_space_dim*

The variables *state_space_dim* and *action_space_dim* indicate respectively the number of parameters of the state in which the agent is and the number of actions that he can take. The specific values for each environment are reported below.
Regarding exploration, the *Softmax* policy was used in order to choose the action given the state. The exploration profile of such policy decays exponentially, as one can observe in Figure (1).

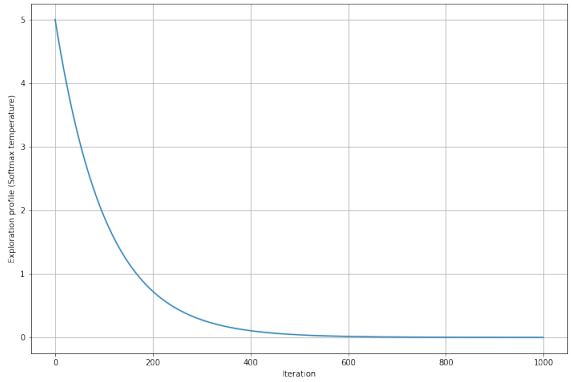

Figure 1: Softmax Exploration Profile

2.1 *CartPole-v1* Environment

In this *GymAI* environment, there is a pole attached to a cart with an un-actuated joint, and the cart can move along a frictionless track. The goal is to prevent the pole from falling over, and this can be done by applying a force of +1 or -1 to the cart. For every time step that the pole remains upright in equilibrium, a reward +1 is given to the agent. 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. In this environment, the *state_space_dim* is 4, in fact the state is composed by *[Cart Position, Cart Velocity, Pole Angle, Pole Angular Velocity]*. The *action_space_dim*, instead, is only of 2 because the possible actions are either to push the cart to the left or to the right.
For the training the *SmoothL1Loss* or *Huber function* was used. This loss criterion uses a squared term if the absolute error is smaller than one, and an L1 term otherwise. This loss function is less sensitive to outliers and it can prevent the issue of exploding gradients. The

optimizer for the Gradient Descent, instead, was the Stochastic Gradient Descent. Moreover, a set of parameters for the network update was defined:

- *gamma*: the discount factor for the penalizing the further rewards and giving more important to the more recent ones
- *replay_memory_capacity*: the length of the memory buffer
- *lr*: the learning rate of the optimizer
- *target_net_update_steps*: the minimum number of episodes to wait before updating the target network.
- *batch_size*: the number of samples to take from the replay memory for each update
- *bad_state_penalty*: a (possible) penalty to the reward if the agent is in a bad state (for example when the pole falls down)
- *min_samples_for_training*: the minimum samples in the replay memory to enable the training.

These parameters were manually tuned in order to find a satisfactory combination, without using any automatic technique such as random search or grid search. At first, an untrained version of the agent was deployed on the environment for 10 episodes, in order to understand the critical aspects and to develop a training strategy to maximize the cumulative reward, i.e. the length of the game.
After that, the training started. Backpropagation was used to update the network weights that would be used to estimate the best Q-value function. The training was done over 1000 iterations. In a first basic version of the training loop, the agent is simply given a 0 reward when the pole has fallen down. However, the reward function was tweaked in order to make the convergence faster. In particular, in a more advanced version of the loop, not only the null reward is given to the agent when the pole falls but, in addition to that, a penalty is given to it proportionally to the distance from the center. In this way, the agent learns that moving from the center leads to a negative reward. In the Results section, it can be observed how this modification impacted the learning curve.
To conclude, a final test over 10 episodes was performed to see the results of the training.

2.1.1 Results

After tuning a bit the parameters, the configuration in Table (1) gave pretty good results. From the first try with the agent picking random actions, it was clear how the pole was constantly falling down, and hence a reward function had to be implemented. After the first basic training loop mentioned in the previous section, the agent learned to keep the pole upstraight for a longer period of time, however most of the times the cart tended to move only in one direction, hence "exiting" the environment. In Figure (2) is shown the learning curve with respect to the exploration profile. As one can observe, the red line, which represents the score after each iteration is almost never reaching 500 which is the target to consider the

gamma	0.96
replay_memory_capacity	10000
learning_rate	0.1
target_net_update_steps	7
batch_size	64
bad_state_penalty	0
min_samples_for_training	1000

Table 1: Best set of parameters

game beaten, not even as the temperature of the softmax approaches to zero.
In the advanced version of the training loop, namely the one with an extra penalty for the cart position, the results were more satisfactory. As it can be observed from Figure (3), convergence to the maximum score was reached in about 800 episodes. The tweaking of the reward function successfully helped both the convergence and the learning speed. The learning curve was highly effected by this change, in fact the convergence to the maximum score was reached even before the end of the 1000 iterations.
In the final test it was possible to appreciate how, over 10 episodes, in all of them the highest score of 500 was reached.

2.2 *MountainCar-v0* Environment

The second task of the howework required to train a deep RL agent on a different environment. Among the many different possibilities, one of them is the *MountainCar* environment. In this framework, a car can move on a one-dimensional track, positioned between two uphills. The goal is to reach the top of the mountain on the right; however, the car's engine is not strong enough to scale the mountain in a single pass. Therefore, the only way to succeed is to drive back and forth to build up momentum.
In this environment, the *state_space_dim* is 2, because the state keeps track of the car *position*, between $[-1.2, 0.6]$, and car *velocity* between $[-0.07, 0.07]$. On the other hand, the *action_space_dim* is composed of 3 possible actions: accelerate to the left, do not accelerate, accelerate to the right.
A reward of 0 is awarded if the agent reached the flag (position = 0.5) on top of the mountain, thus solving the task. On the contrary, a reward of -1 is awarded if the position of the agent is less than 0.5. Therefore, in this case, the agent has to learn to minimize the loss, namely to reach the top of the hill in less time steps as possible, differently with respect to the other environment where the goal was to maximize the cumulative reward.
The position of the car is assigned a uniform random value in $[-0.6, -0.4]$, while the starting velocity of the car is always assigned to 0. Finally, the episode terminates if the car position is more than 0.5 or the episode length is greater than 200.
The workflow for the training of this agent was similar to the one in the Cartpole environment. However, in this case, an advanced version of the training loop was immediately adopted without trying a basic reward function. A tweaked version of the reward function was implemented. In order to encourage the agent to exploit momentum without the knowledge of any physic law, the idea was to award the agent with a positive reward if the car was going right but accelerating left, or the other way around. In this way, the agent learned to

exploit momentum. Moreover, a positive reward of +1 is given to the agent when the car reaches position 0.5, instead of 0 as default. This allowed to reach a faster convergence.
To conclude, a final test over 10 episodes was done to observe the learning effect.

2.2.1 Results

The set of parameters used for this task is shown is Table (2). Before training, it was clear

gamma	0.97
replay_memory_capacity	10000
learning_rate	0.01
target_net_update_steps	10
batch_size	128
bad_state_penalty	0
min_samples_for_training	1000

Table 2: Best set of parameters

how the agent was picking random actions and was not able at all to exploit momentum. The training went over for 1000 iterations, and the results are shown in Figure (4). It is clear that the training was successful after only a bit more than 300 iterations. The left plot shows how, as the temperature decreases, the learning curve grows towards the convergence to 0, which is the highest score achievable. On the right, instead, the car positions are accordingly converging towards 0.5 which confirms that the goal was reached.
The final test went over 10 episodes, and in all of them the car reached the top of the mountain in 120 time steps on average. A *gif* of the result can be found attached to the report.

3 Appendix

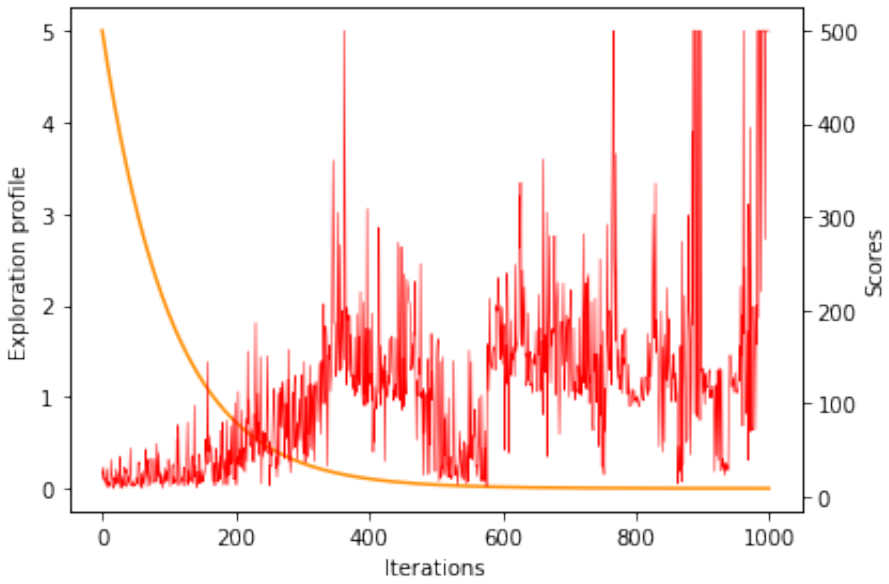

Figure 2: Exploration profile vs. learning curve in the basic training loop

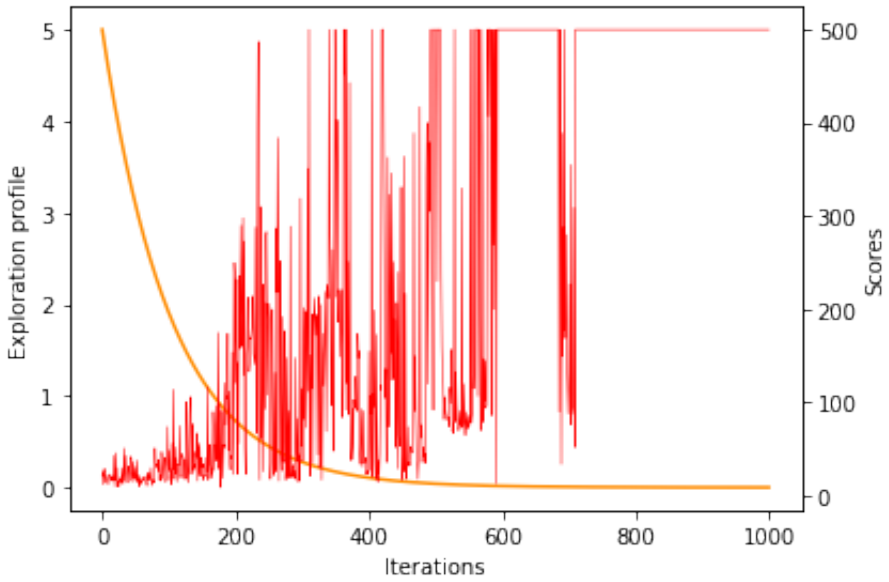

Figure 3: Exploration profile vs. learning curve in the advanced training loop

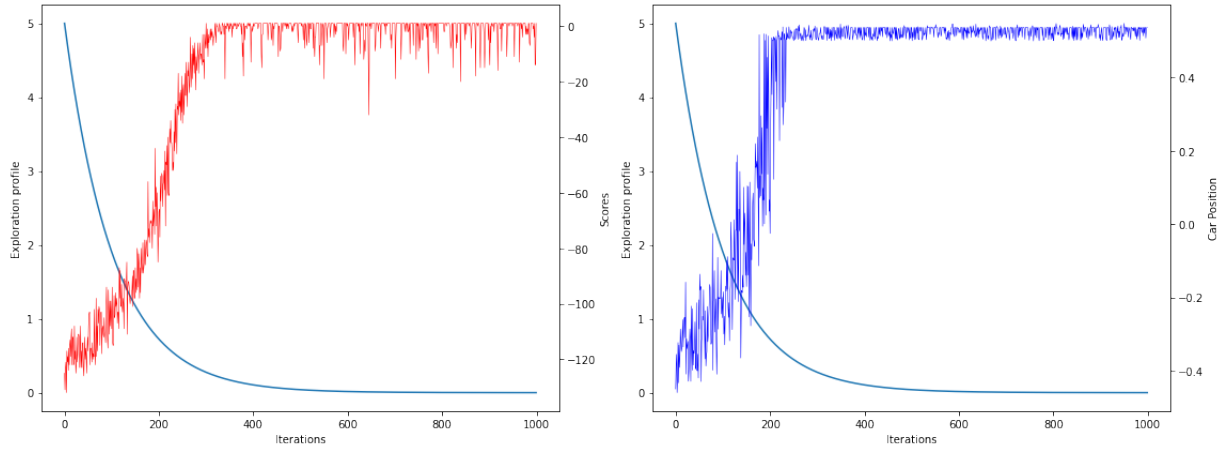

Figure 4: Exploration profile and learning curve with both the score and the car position

.