Exercise 03 -Deep Reinforcement Learning

University of Padua

Name:	Daniele Mellino
Student Number:	2013373
Course of study :	Physics of Data
Date:	February 13, 2022

1 Introduction

In this homework the implementation and testing of neural network models for solving Reinforcement Learning(RL) tasks is discussed. The neural networks model will be implemented to be the Deep Q-Learning Agent in the Cart-Pole environment provided by the OpenAI library *gym*. The task in this environment is to balance a pole attached to a cart, without letting it fall. In the meantime the cart moves along a fractionless track. The report will be divided in two section, consisting of the two ways in which the environment is going to be considered:

- In the first procedure the input considered for the task are the position, the velocity and the pole's angle and angular-velocity(state representation).
- In the second procedure the input of the neural network will be the rendered image of the cart-pole.

2 Cart-Pole state input

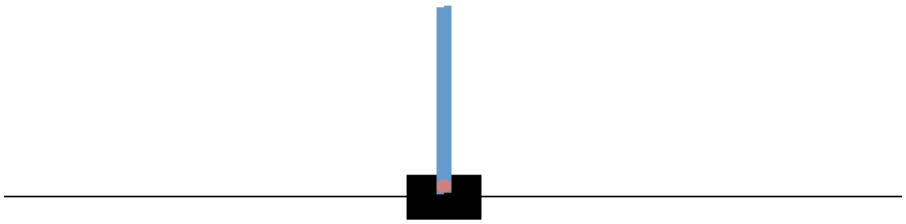


Figure 1: The gym environment consists of a cart moving on a one dimensional rail without friction. At its centre a pole is pinned and is left free to oscillate. It start vertical at the centre, the goal is not letting it down

An example of the environment provided is showed in Figure 1. The observation space in this first part consist of :

- **cart position**: it range between [-2.4 and 2.4];
- **cart velocity**: it range in $[-\infty, \infty]$;
- **pole angle**: it range in $[-15°\ , 15°]$
- **pole angular velocity**: it range in $[-\infty, \infty]$.

The agent has two available actions, moving the cart to the left or to the right. The pole is considered fall if its angle became bigger than $|12°|$. In that situation and in the case the cart goes out of the window the game is lost, otherwise if it is not the case for 500 iterations the game is won.

2.1 Methods

If we call respectively the action, the reward and the state at timestep t a_t, r_t and s_t, the goal of the agent is to maximize a return G_t, which is weighted through a discount factor $\gamma \in (0, 1)$, which quantifies how much the agent should care about the future: $G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k}$.

In Q-learning an agent learns to associate a value Q to each state-action pair such that it takes into account both the immediate and future rewards. In other words $Q_\pi(s, a) = E[G_t|s, a]$, where $\pi(s)$ is the policy by which the actions are performed. In principle the goal is to choose $\pi(s)$ in such a way to retrieve the optimal Q, $Q^*(s, a) = max_\pi Q(s, a)$. Obtaining Q^* is an heavy task in term of time and computation, that is why in DeepRL they are approximated by a deep neural network. Such a network needs to minimize the objective function:

$$L(\theta) = E_{s,a,r,s'}[(r + \gamma \cdot max_{a'} Q(s', a', \theta^T) - Q(s, a, \theta))^2] \tag{1}$$

In the above formula, the terms with γ refers to the *target* network, the last term instead is the *policy* network. The first is employed for a more stable training process and to reduce the possible correlation between target function and Q-network. It is updated only every "target_net_update_steps".

Finally experience replay is used to helps convergence and reduce the correlation between samples, in fact it consists in storing tuples of (state, action, reward, next state) in the Replay Memory, than in the training, batches from this buffer are sampled and replayed. The Replay Memory used consists of a *deque* object with fixed length("capacity").

The architecture used for the target and policy network is the same and consists of 3 fully connected layers. The first takes in input the state so it has 4 units and output 128 nodes, the following ends up with 128 nodes too, and finally the last layer ends in 2 nodes(coinciding with the left or right action). The activation functions of each layer is the Tanh.

As suggested in the paper from Deep Mind the optimiser used to train the networks is the SGD with learning rate "lr" and the loss used is the Huber loss.

Given the Q-values, it is important to wisely select which action to choose. Usually in the first part of the learning one wants to explore different states in order to not get stuck in a local minima, once enough experience is reached one want to exploit that knowledge. To do that the action to be taken is chosen according to this policies :

- ϵ-greedy policy : the action that maximise Q is chosen with $1 - \epsilon$ probability, a non optimal one vice versa.
- softmax policy: choose the action based on a distribution obtained applying a softmax (with temperature τ) to the estimated Q-values. The highest the temperature, the more the distribution will converge to a random uniform distribution. At zero temperature, instead, the policy will always choose the action with the highest Q-value.

In this home work the Softmax strategy is adopted. In order to define how the temperature τ change during the learning process two different exploration profiles is implemented. In the first the temperature decrease exponentially(type = 0) starting from a value T_{max}, in the second it decrease linearly to a certain value and then is kept constant(type = 1).

During the training, the agent gets a reward which is incremented by one for each step in which the game is not lost. In addition, in order to keep the cart in the middle(avoiding sub-optimal solution) a penalty is implemented. It is proportional to the distance from the centre powered by a factor "type_penalty".

Since the training is very time expensive, in order to save iterations, in addition to the original laboratory notebook, an early stop procedure is added. The training end if the maximal score is reached for "x" consecutively step. The optimal value of "x" is found by hand looking at the test scores, in fact too low value of it could stop the model in a sub-optimal minimum. In the opposite scenario computations would be wasted. The assignment of the homework is to improve the convergence velocity, this has been achieved through a multi-dimensional search through Optuna. Since training is slow, the default sampler algorithm is adopted(TPE algorithm) instead of a Grid/Random Search. The optimised metrics are the following :

- Maximum Score (without penalties): If the games convergence it is 500, otherwise the mean value of the last x epochs is considered.
- Velocity : the value at which the model has reached convergence. If the training ends at epoch R, if it is smaller than $1000 - x$ the velocity is considered $R - x$, otherwise if the training ends with maximum score after 1000 epochs the value of R is the epoch number at which the counter for convergence start the last time. Finally if the training ends without convergence, the value of R is 1000.

2.2 Results

The parameters used to optimize the model are :

- γ(see Eq. 1): it is sampled uniformly in $(0.93, 0.99)$;
- **learning rate** (lr): it is sampled uniformly between $(0.001, 0.069)$ with step 0.02;
- **type_penalty**: sampled uniformly between $(0.5, 2.5)$ with step 0.5;
- **type_profile**: could be 0 or 1, depending on the decay profile chosen, respectively exponential or linear;
- $\boldsymbol{T_{max}}$: sampled uniformly between $(3, 18)$;
- **target_net_update_steps**: sampled between $(1, 15)$

The search has been executed with 60 trials, the best model's parameters is in Table 1. In Figure 2 the the best model re-trained (notice that here a different random seed is used, however result is still better respect to the lab).

Velocity	γ	learning rate	type_penalty	type_profile	T_{max}	target_net_update_steps
324	0.954958	0.045	1	0	14	2

Table 1: Best parameter values from the optimization search, look at the notebook for the other values.

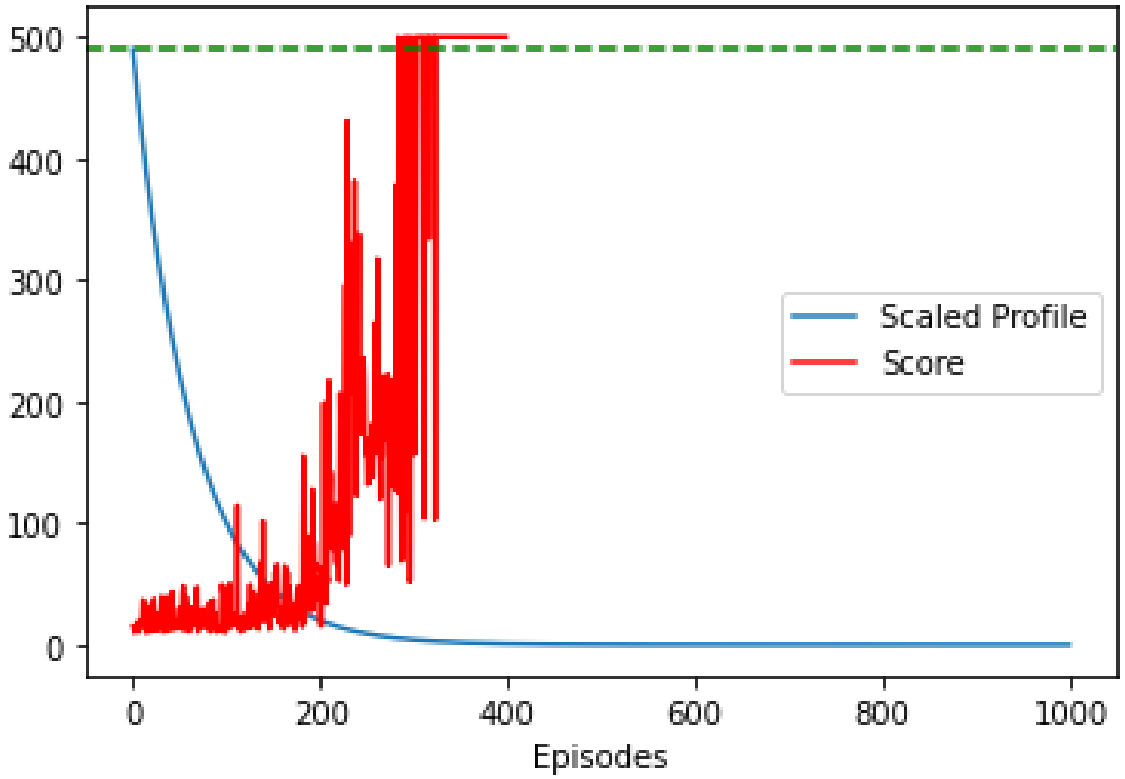
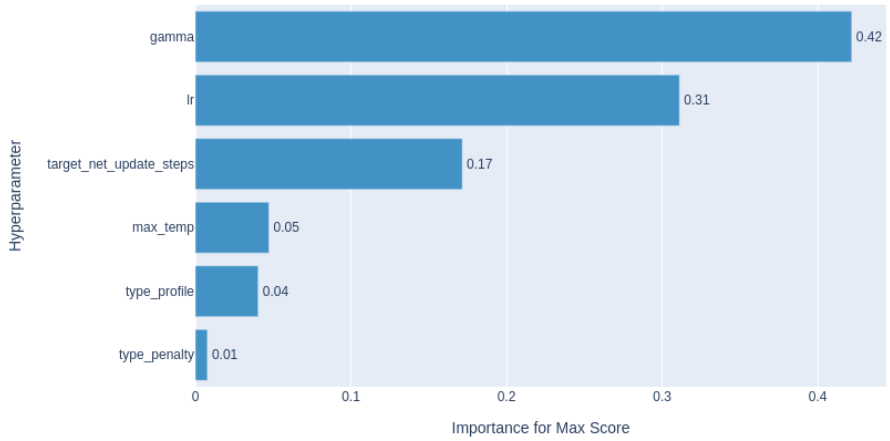


Figure 2: Re-trained best model, notice the early stopping. The green line is equal to score equal to 490

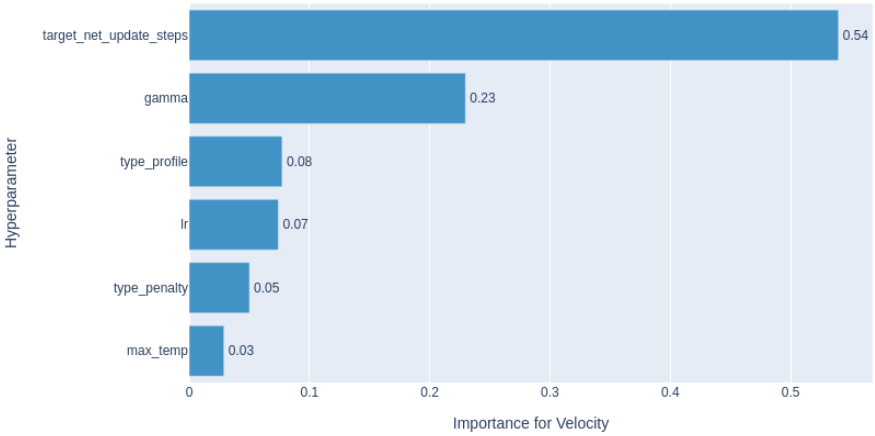
The result of the re-trained model is then tested for 100 different input obtaining perfect score.

From the importance plots in Figure 3, one can see that in this simple game "target_net_update_steps" needs to be small enough. In addition it is the most relevant parameter in the Velocity search together with γ. In the convergence of the model instead also the learning rate became relevant.

Finally to see how deviation in the profile affect the learning, a profile is corrupted with random gaussian noise in a random position. In Figure 4 one can see that the distortion in the previous profile led to higher convergence time and a less stable behaviour.



(a) Importance of the parameters used in the search for the Maximum Score.



(b) Importance of the parameters used in the search for the Velocity.

Figure 3: Parameters Importance in the Optuna search

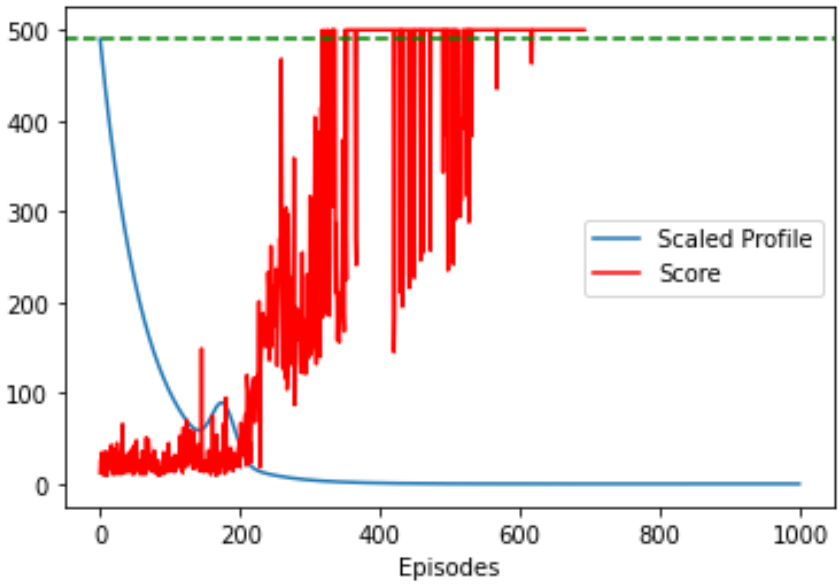


Figure 4: Re-trained best model with distorted profile. The green line is equal to score equal to 490

3 Cart-Pole pixels input

In this section, the same gym environment is used with a different observation space. The new input will consists of the screen pixel of the rendered image.

3.1 Methods

To tackle this task we need to overcame two main problem. The first is that the size of the input is too big to handle. The second is that infer the required information from the screen pixels is a complex task which require a lot of computational time.

Cartpole gym environment outputs 600x400 RGB arrays (600x400x3). That is way too many pixels for this task. That is why we preprocess the data.

First we convert it to grayscale(2 channels less) afterwards to a full black Cartpole. This is not enough yet. The image is still too large, therefore the part above the pole is cut. Since now no-relevant information has been deleted. However the images are still too big, therefore we cut vertically a small boundary of the pole and resize the images of the 50%. The final output of preprocessing is showed in Figure 5 and has size (76,42). Notice that here we "cheated" using the position of the pole obtained by the environment to center the cutting function.

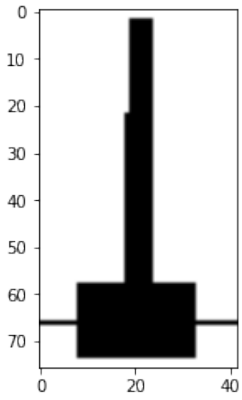


Figure 5: Rendered image preprocessed

Finally, in order to give the network the opportunity to learn from past frames, we dispose the network model to deal with different number of input frames,nf. In principle, to let the network learn all the information needed to solve the game, so the position, the velocity and the acceleration. To retrieve the position, one frame is enough, for the velocity two is needed, in conclusion the acceleration needs at least three frames.

The Deep-Q Convolutional Neural Network implemented architecture consists of:

- A First Convolutional Layer takes in input nf channels,output $4 \cdot nf$, with kernel size=(8,4), padding=(0,1) and stride=(4,3).
- A Second Convolutional Layer which double the input channels($4 \cdot nf$), kernel size=(6,2),no padding and stride=(3,3).
- After a Flattening layer a series of three Linear Layers of sizes: $(nf \cdot 25 \cdot 8, 250) \rightarrow (250, 128) \rightarrow (128, 2)$ follow. Here 2 is the dimension of the action space.

3.2 Results

Different trials to tune the model have been tried, the ones used in the following graph are memory capacity of 2000, the minimum samples in it to enable the training is set to 200. Then the batch size is set to 16, therefore we will have more noise result, in the trials emerged to work better the RMSprop optimiser with learning rate of 1e-03. In conclusion gamma is 0.99 and the softmax activation profile with $T_{max} = 4$ is used.

In Figure 6 the results obtained for different values of n_frames, one can notice that for 1 single frame(blue line) the network is not learning anything as expected. Then with 2 frames thing get better since the network is able to learn the velocity information. Looking at the average score for the last 300 episodes the best model seem to be the one with n_frames equals to 3(green line) achieving score 76.

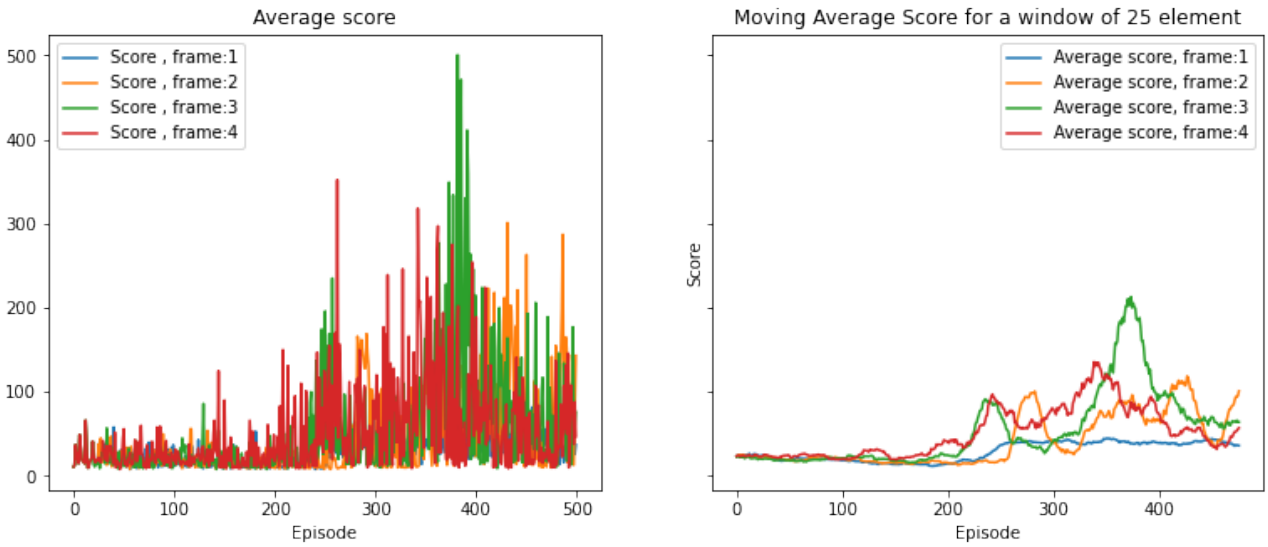


Figure 6: Training curves for the model trained with different n_frames.