Università degli Studi di Padova

Homework #3 Deep Reinforcement Learning

Student: *Beatrice Segalini* – 1234430

Course: *Neural Networks and Deep Learning* – Professor: *A. Testolin*

1. Introduction

In this homework, the implementation and testing of neural network models for solving Reinforcement Learning (RL) problems is displayed.

A neural network will be trained as a Deep Q-learning (DQ) Agent in three different environments, provided by the OpenAI library `gym` .

At first, the Cart-Pole environment is considered. The task consist of balancing a pole attached to a cart, which moves along a frictionless track, preventing it to fall.

Secondly, the same environment is treated, but using the screen pixels as inputs, rather than the state representation (cart position, cart velocity, pole angle, pole angular velocity) provided by the environment.

Finally, the same agent is used to study the MountainCar environment, in which the task is to push a car on a one-dimensional track over a hill by building up momentum.

2. Cart-Pole

This `gym` environment consists of a cart that can move, without friction, on a 1-dimensional rail. At its centre, a pole is pinned and is left free to oscillate. The pendulum starts in a vertical position, and the goal is to avoid its fall by moving the cart along the track.

The library provides a **observation** space of dimension 4 that includes the cart position (ranging between -2.4 and 2.4); the cart velocity ($\in [-\infty, +\infty]$), the pole angle ($\in [-15°, 15°]$) and pole angular velocity ($\in [-\infty, +\infty]$).

The agent can choose between two possible **actions** that the cart can do: either moving to the left or to the right, with the velocity depending on the pole angle.

Reward is 1 for every step taken, including the termination step, and an episode ends when the pole falls (i.e., its angle overcomes $\pm 12°$), the cart exits from the observation screen, or the episode lasts more than 500 steps.

2.1. Methods

First of all, the `ReplayMemory` class is defined, for implementing the so-called *experience replay*, used especially to improve the convergence properties of the algorithm and to reduce data correlation. The Replay Memory uses the `deque` python function to store the agent's last experiences, forming a queue of fixed length `capacity` .

The class `DQN` includes the architecture of the networks used for this task, i.e. the *policy* and the *target* network. The policy network takes a state as input, and provides the Q-value for each of the possible actions. The target network is employed for a more stable training process and to reduce the eventual correlation between the target function and the Q-network. It is updated only every `target_net_update_steps` .

Both the networks are composed of 3 fully connected layers with $[4, 128, 128, 2]$ units, activated by the hyperbolic tangent function, with SGD optimiser and Huber loss (`SmoothL1Loss`).

Starting from the estimated Q-values, the agent chooses then the action to take, according to the *exploration policy*. This is a criteria according to which the agent picks the action that provides the highest long term reward (exploitation), or a different option, that might be not convenient at the moment but could lead to finding a better policy (exploration).

Two different exploration strategies are defined: $\epsilon-$greedy, according to which a non-optimal action is chosen with probability ϵ, and the Softmax, in which the action is decided based on a distribution obtained applying the Softmax function (with temperature τ) to the estimated Q-values. The Softmax strategy is adopted in this section.

Moreover, it is important to define how ϵ and τ change during the learning process. To do so, two different *exploration profiles* are defined: one in which the temperature decays exponentially (`expl_profile_softmax`), and one in which it decreases linearly (`expl_profile_linear`). Besides, a function that adds gaussian fluctuations (`gauss_fluctuation`) to the exploration profile is coded, in order to see their effect on the learning process. Up to two fluctuations will be added in the profile, respectively with probability 0.75 and $0.75^2 = 0.5625$. Examples of exploration profiles can be found in Figure (6). 15 different exploration profiles are generated and randomly applied during the search for optimum parameters.

The parameters to be optimised in the random search, in which 50 different combinations of exploration profiles and hyperparameters are tested, are:

- the discount rate `gamma` , taken in $[0.9, 0.99]$ at 0.01 steps;
- the SGD `learning_rate` , chosen in the log-spaced interval $[10^{-3}, 10^{-1}]$;
- the number of episodes to wait before updating the target network `target_net_update_steps`, picked between 5, 10, 15 and 20;
- the `batch_size` $= [32, 64, 128, 256]$, i.e. the number of samples to take from the replay memory for each update;

The capacity of the replay memory `replay_memory_capacity` , the minimum number of samples in the replay memory to enable the training process `min_sampling_for_training` and the extra penalty factor for discouraging "bad" states `bad_state_penalty` are set respectively to $[10000, 1000, 0]$.

The reward during the training is slightly tweaked, by adding a penalty if the cart gets further from the centre of the screen.

2.2. Results

The better performing agent's parameters are reported in Table (1), while the results of the random search are reported in Figure (7).

gamma	learning_rate	batch_size	target_net_update_steps
0.95	0.1	64	5

Table (1) – Optimised hyperparameters for the DQN, found via a random search over 50 different parameter-exploration profile combinations.

This agent is able to win the game and reach the maximum score of 500 after 370 episodes, and then maintains its performances, as shown in Figure (1). Other graphs similar to Figure (1) can be found in the Jupyter notebook, for each set of parameters/exploration profile considered.

A very interesting observation can be done concerning the exploration profiles. During the various trials performed during the optimisation of the parameters, it emerges that the exponential decay of the temperature is the profile which suits best this task. The linear decay leads generally to lower

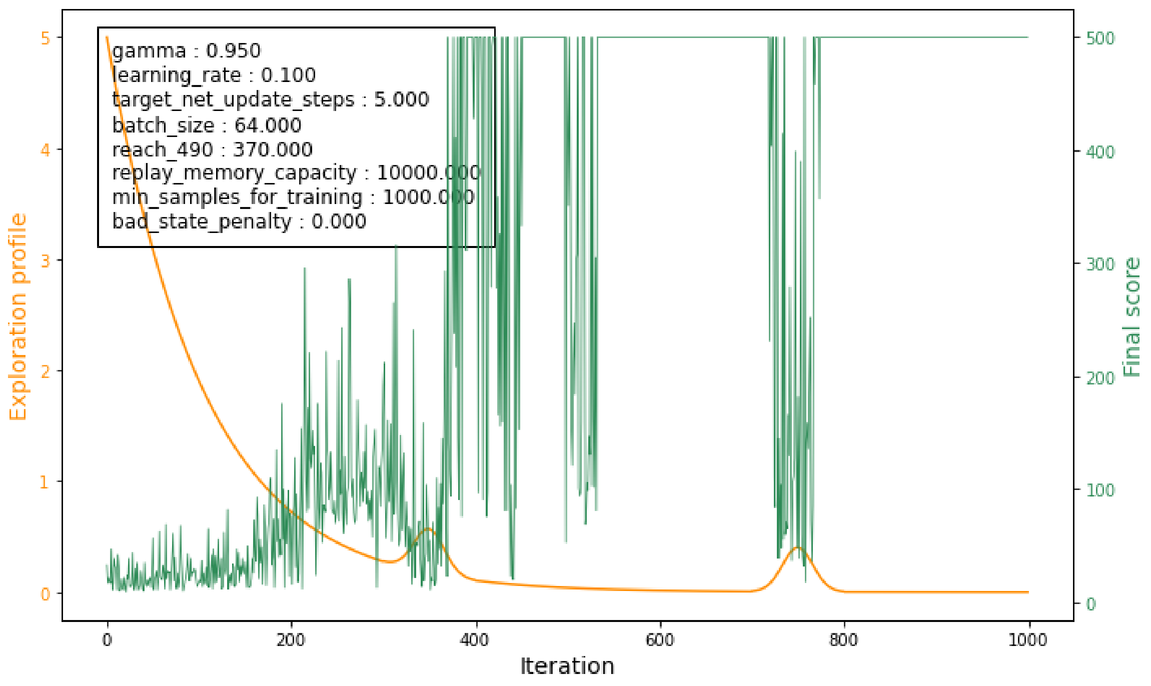


Figure (1) – Plot of episode score vs. episode number and of the exploration profile, for the best performing agent. In the graph, also the hyperparameters are reported.

scores or is slower to learn. The gaussian fluctuations, instead, have a disrupting effect in the learning process, which can also be seen with the better performing network: around the two gaussian peaks, the final score of the episodes drastically drops. Therefore, they are not employed anymore in the next sections.

After selecting the best model, the policy net was tested making the agent always choose the best action by setting the temperature to 0. In this conditions, the agent manage to always win the game, obtaining the perfect score of 500, proving that the strategy he learned is indeed the winning one. Example videos of a winning runs can be found at this link: Cart-Pole - best model - video.

3. Cart-Pole with pixels

In this section, the same `gym` environment will be studied but changing the observation space from the one provided by the library to the screen pixels.

3.1. Methods

Inferring the required information from the screen pixels is indeed a complex task. The built-in render `rgb_array` mode returns $3 \times 400 \times 600$ RGB-images, that are indeed too big to handle. Therefore, the first step consists of reducing the data dimensions by pre-processing the inputs. In particular, the user-defined `get_frame` function will:

- convert the RGB pictures to black and white ones ($3 \times 400 \times 600 \longrightarrow 1 \times 400 \times 600$);
- resize them, interpolating with the `cv2` library, reducing the dimension to 160×240;
- crop the non-relevant areas, and centre each window on the middle of the cart, but keeping the original proportions ($160 \times 240 \longrightarrow 100 \times 150$).

To derive information about the velocity of the cart, 4 consecutive frames will be considered in the learning process. This directly translates into the shape of the input layer of the policy/target network.

In fact, since the inputs are 4 pictures, the main architecture coded is a Convolutional network, with ReLU-activated layers, composed of:

- a convolutional layer with 4 input channels, 16 output channels, kernel size of 8, padding $= 0$, stride $= 4$, that reduces the image sizes to 24×36;
- a convolutional layer with 16 input channels, 32 output channels, kernel size $= 4$, padding$= 0$, stride$= 2$, that decrease the picture sizes to 11×17;
- a fully connected linear layer (after the required flattening) with $11 \cdot 17 \cdot 32 = 5984$ inputs, 256 outputs;
- a fully connected linear layer with 256 inputs, 128 outputs;
- a final linear layer with 128 inputs, and `action_space_dim` $= 2$ outputs.

Other architectures were also considered and trained, more specifically by changing the stride from 2 to 4 of the second convolutional layer or removing the second linear layer (i.e. making the output layer with 256 input units insted of 128), or also by changing the input size without cropping (leaving it 160×240), or cropping to a 100×100 square image and not a rectangular one (taking inspiration from [1]).
Both the $\epsilon-$greedy and the Softmax strategies are implemented.

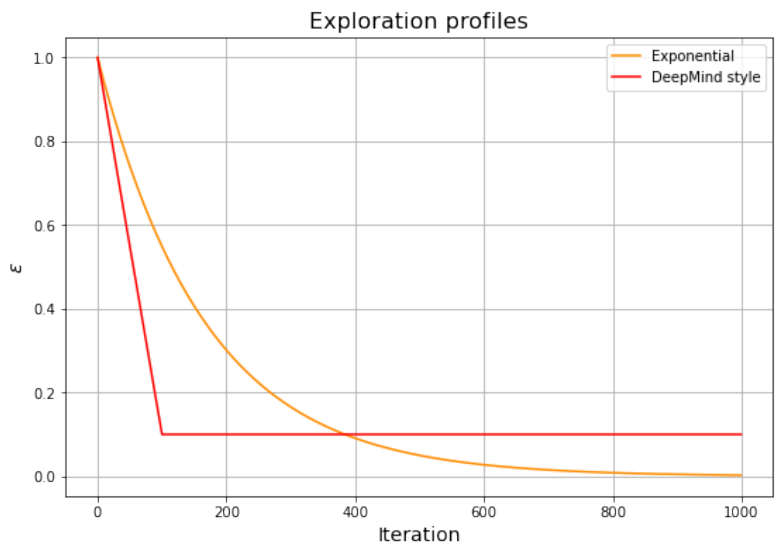


Figure (2) – Cart-Pole with pixels: two exploration profiles.

Moreover, two different exploration profiles are adopted: a linear one (inspired by [1]) and an exponential one, as shown in Figure (2).

The optimiser chosen is RMSProp, with learning rate varying between 0.1, 0.01, 0.001 and 0.0001. The discount rate `gamma` is fixed to 0.99, and also the batch size and the number of steps between each target network update are set respectively to 32 and 5.

Broadly speaking, the training is very difficult and time consuming, so, although several tests are carried out with different combinations of exploration profiles, parameters or architectures, they are tweaked by hand and not through a proper random search.

To speed up the process, for the most promising network, the training was performed twice, using the network parameters found after the first 1000 iterations to initialise the new network a second time. The learning procedure is stopped manually when it reaches acceptable performances, by looking at the average score of the last 50 epochs.

3.2. Results

The most effective model is the one aforementioned in the list above, with a RMSProp learning rate of 0.0001, and weight initialised from a previous training (`cartpole_pix_crop` file, results in Figure (8)). The action strategy adopted is the $\epsilon-$greedy one, with an exponential exploration profile. The results of the training are displayed in Figure (3). It is worth noticing that the learning process might still not be finished, but stopping it was needed due to the limited computational resources (Google Colab GPUs are available for free for a limited amount of time).

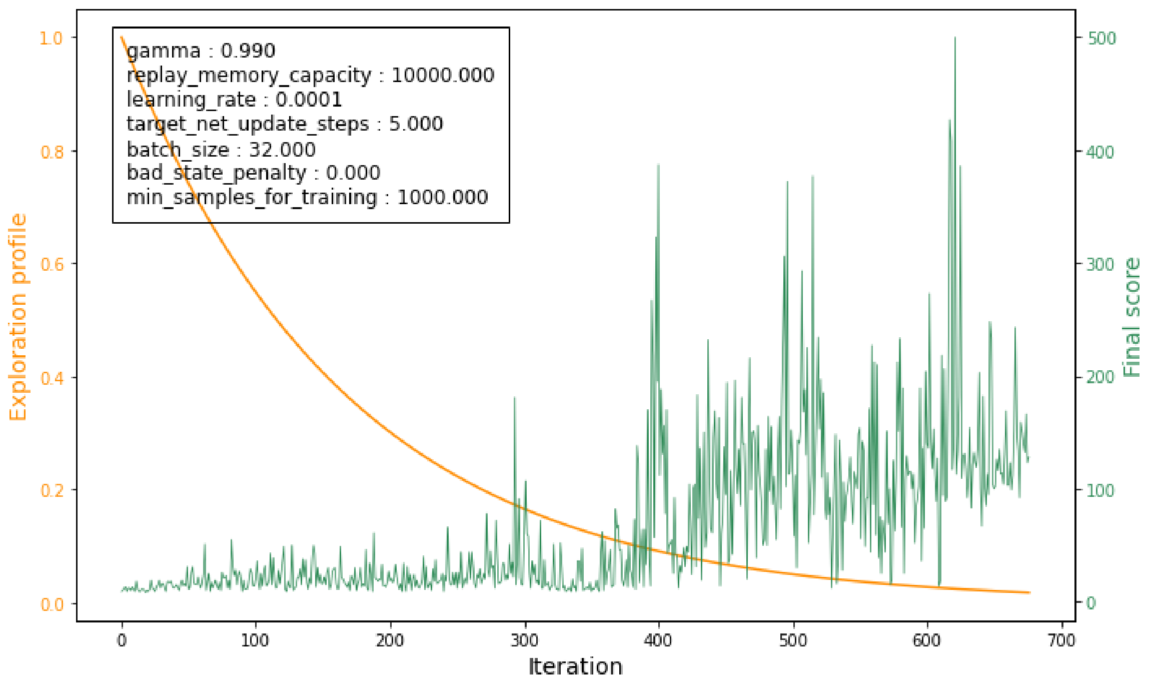


Figure (3) – Exploration profile-Final Score vs. episode number, for the final model of the Cart-Pole+pixels task. The agent managed to win once, at episode 622.

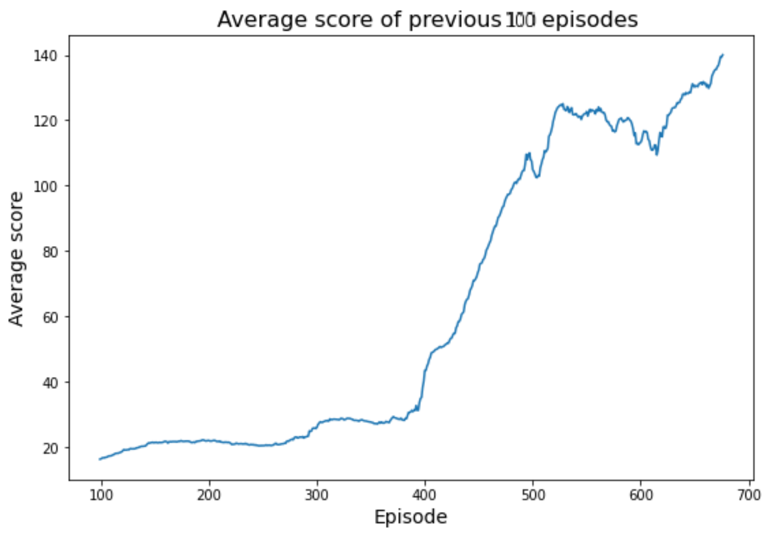


Figure (4) – Average score along the previous 100 epochs.

To check the properties of the learning process, the graph in Figure (4) is produced, in which the average score of 100 epochs is plotted, as a function of the episode number.
It can be clearly seen that after 400 iterations the score significantly increase, going up at a steady rate for about 200 episodes, and than stabilises around $120 - 150$.
In the last part of the curve, it is noticeable how the average score is still increasing. Thus, one can deduce that more iterations could have improve further the network performances.

However, to avoid the possibility of catastrophic forgetting or to lose the already computed data due to Colab runtime disconnection, the training was stopped after 677 episodes.

The final network is tested with 100 trials and achieves a final average score of 106.67, with an episode reaching 156.

From the videos obtained (Cart-Pole - pixels), it can be noticed that the pole seems always to be falling because the cart moves to the left too much, and more or less all the episodes ends with the cart in the left part of the screen and the pole falling to the right. The strategy the agent learned is not the best one, but it is indeed proof that learning occurred, and nevertheless assure scores over 100.

With more training iterations, it is likely that the agent could have learned how to completely beat the game.

4. MountainCar

In this section, the same DQN model presented for the Cart-Pole game is used to solve another `gym` enviroment: MountainCar.

In this setting, a car is on a one-dimensional track, positioned between two "mountains". The goal is to drive up 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. The possible actions for the agent are 3 (and not 2 like the Cart-Pole): accelerate to the left, to the right or cease any acceleration. The observation space has two components: the car position, ranging from $[-1.2, 0.6]$ and velocity in the $[-0.07, 0.07]$ interval.

The agent receives a 0 reward if it reaches positions > 0.5 (the flag on the mountain), or -1 for every step it does not. Episodes end when the car position is more than 0.5 or if their length is greater than 200 steps.

4.1. Methods

The functions applied are the same already explained for the standard Cart-Pole problem. The only difference is the dimension of the action space (3, and not 2) and of the observation space (2, not 4). The class `DQN` was however already flexible to these changes, therefore it is chosen as model for the policy/target networks. The parameters chosen are the same reported in Table (1). The exploration profile is an exponential one, starting from $\tau = 1$.

To assure convergence and speed up significantly the learning process, after some trials and errors, the reward was tweaked in order to encourage actions that actually help the car in building up momentum. In particular, the reward is increased of 1 if the car previously moved to the left, and the action chosen is to accelerate on the right (and vice versa) and if it reaches positions ≥ 0.5. In this way, the agent will receive a reward of -1 for "wrong" moves and if its position is not the winning one, 0 if it chooses the right move, 1 if it wins the game.

4.2. Results

In this framework, the agent trained manage to perfectly solve the environment after less than 50 episodes, as it can be seen in Figure (5). In the graph on the left, i.e. the exploration profile-score vs. iteration number plot, the converging behaviour of the learning process can be clearly seen; on the right, the maximum position reached by the car is represented, and it can be observed that after only 34 episodes the car reaches for the first time the top of the mountain, then optimising its strategy to reduce the time required to achieve its goal. This is even more clear by watching the videos of the trained agent (MountainCar), in which it can be noticed that, after the training, the car learned to climb the mountain with just $2-3$ oscillations.

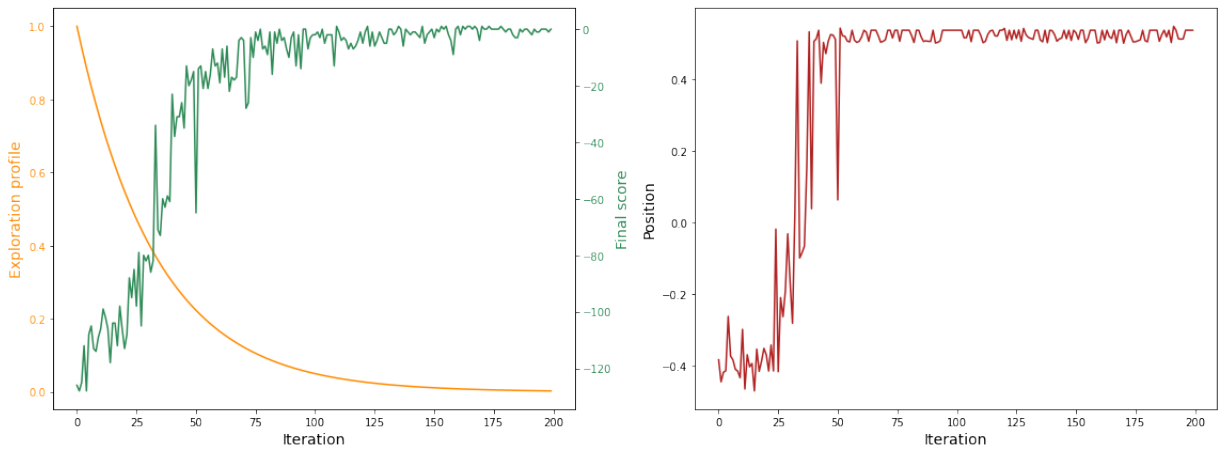


Figure (5) – Training process for the DQN in the MountainCar environment. On the left, exploration profile-score vs. iterations; on the right, maximum position reached by the car vs. episode number.

Appendix

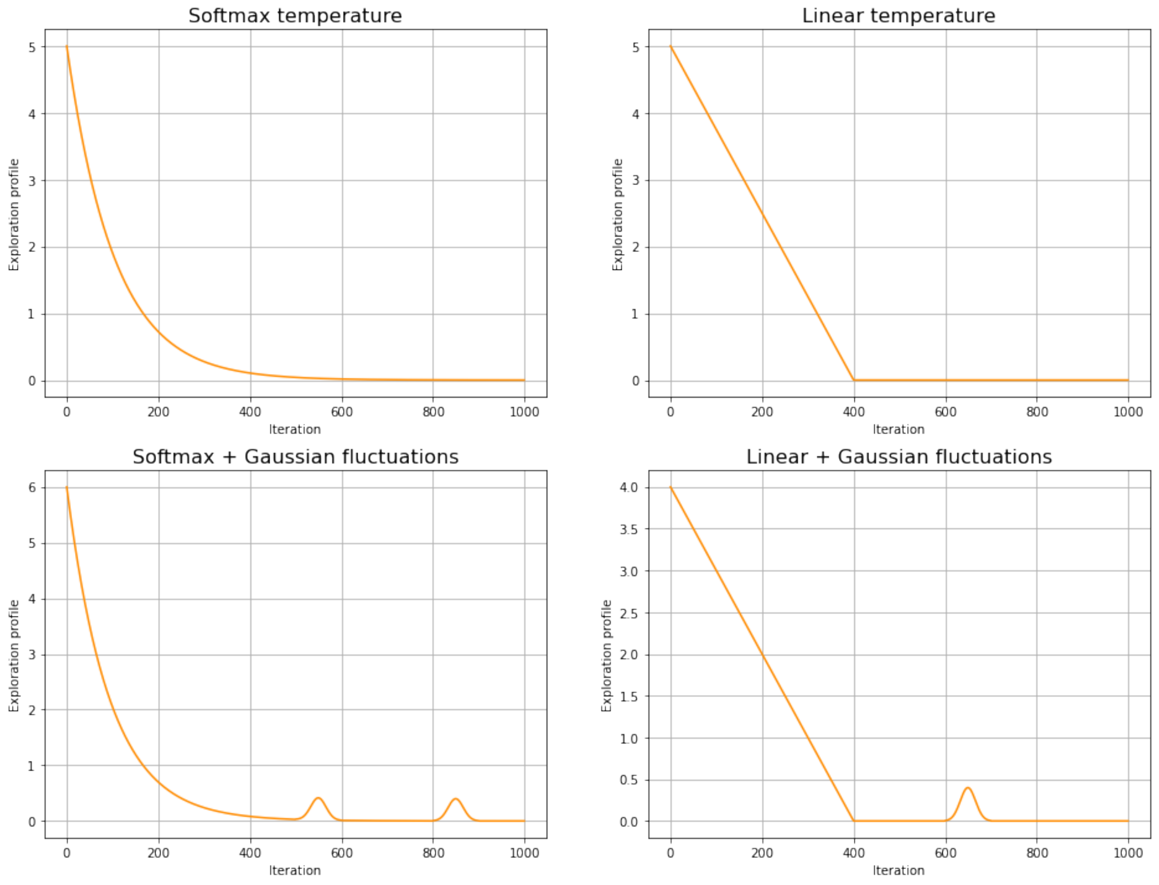


Figure (6) – Example of exploration profiles, with different starting points and with/without gaussian fluctuations.

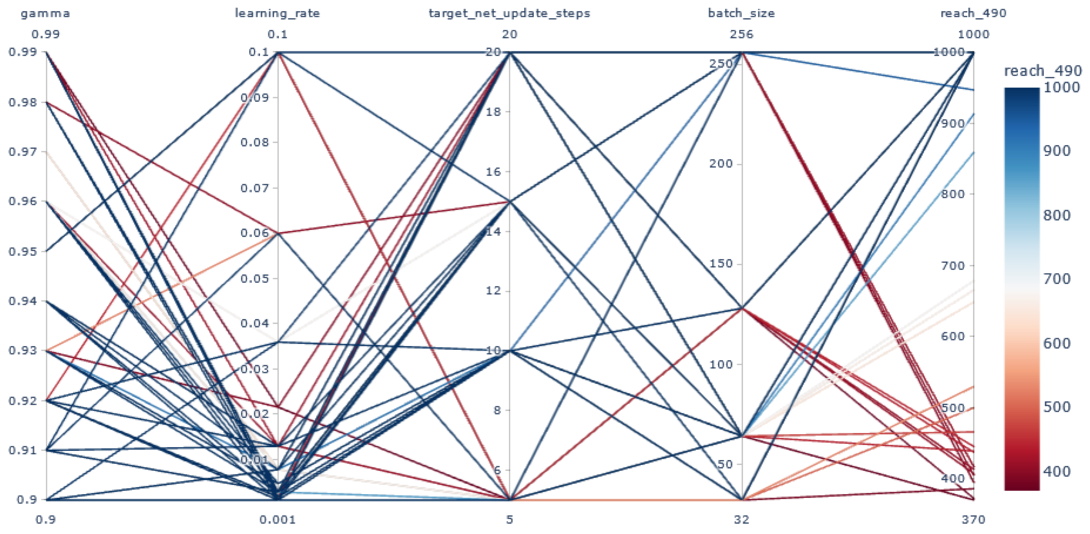
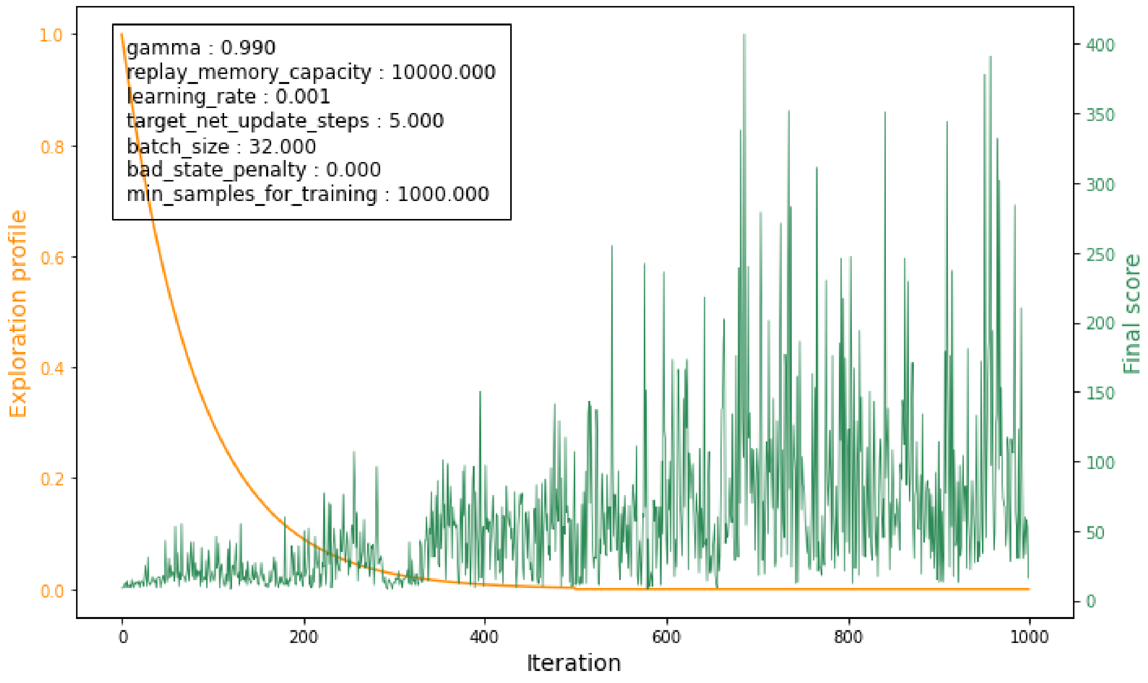


Figure (7) – Parallel coordinate plot, which highlights what set of parameters performed better during the random search. The last column is the one that shows at which iteration the network reached a score of 490.



gamma : 0.990
replay_memory_capacity : 10000.000
learning_rate : 0.001
target_net_update_steps : 5.000
batch_size : 32.000
bad_state_penalty : 0.000
min_samples_for_training : 1000.000

Figure (8) – Score-Exploration profile as a function of the episode number, for the CNN implemented as pre-network of the final training in the Cart-Pole+pixels task. The final score stabilise around 100, with some spikes towards higher scores, but never winning the game.

References

[1] Volodymyr Mnih et al. *Playing Atari with Deep Reinforcement Learning*. 2013. arXiv: `1312.5602 [cs.LG]`.