Homework 3: Deep Reinforcement Learning

Alessandro Lambertini - 1242885
(Dated: August 18, 2022)

In this homework we have to implement and test neural networks in order to tackle some deep reinforcement learning tasks. As first task we have to work on the basis of what has been done in the 7^{th} laboratory session. We implement a deep Q-learning agent in order to learn to control the *Cartpole* environment. For the advanced task I have chosen to implement an agent that interact with the *MsPacman-v0* Gym environment using directly the screen pixels. In this way both the advanced tasks proposed are tackled at once.

I. INTRODUCTION

A. Cartpole

'CartPole-v1' is a simple environment provided by Gym to train and test reinforcement learning algorithms. The agent is the cart and the goal is to prevent the pole to fall over. In order to do so the agent can perform 2 actions, push the cart to the left and push the cart to the right (a numpy array with shape (1,) with values 0 or 1). The observation is a ndarray with shape (4,) with the values corresponding to the cart position and velocity, and to the pole angle and angular velocity. The agent is rewarded at each time step, if the pole is still upright, with a +1 reward. The episode ends when the pole is more than 12 degrees from vertical, the cart moves more than 2.4 units from the center or if the episode length is greater than 500 time steps. In order to tackle the task and to be consistent with the work done in the previous homeworks, some python classes and scripts are produced exploiting and modifying the code present in the Jupyter notebook of the 7^{th} laboratory session. Some hyperparameter optimization strategies are explored within the OPTUNA framework and different reward functions have been tried out.

B. Pac-Man

'MsPacman-v0' is a Gym environment that emulates the popular videogame. Although it is possible to customise the environment with some arguments that control things like the difficulty of the game and the action space, the environment has been settled up with the default settings. The action space is made of 9 possible actions (a numpy array with shape (1,) that takes integer values from 0 to 8 which stand for the actions: NOOP, UP, RIGHT, LEFT, DOWN, UPRIGHT, UPLEFT, DOWNRIGHT, DOWNLEFT). In order to fulfill both the advanced tasks proposed i have chosen the default type of observation, that is a 210×160 RGB image of the game. The images are preprocessed in order to lighten a bit the computational effort required. The agent is rewarded by default according to the game score. In this case of course the deep Q-network is chosen to be a convolutional neural network. Due to the heavy computational effort required it was decided to not perform any automatic hyperparameters optimization procedure. However, different tests have been made acting on the hyperparameters, policy, reward function and action space. A final long training has been performed and it constitutes the results of the task.

II. METHODS

A. Cartpole

The implementation of the agent that solves the Cartpole environment is based on the code in the lab notebook. However, as specified above, it has been reorganized and modified in different python classes and scripts:

- *replay_memory.py*: In this file we find the class 'ReplayMemory' that implements the memory of our agent. The past experience is stored as a deque object that is a collection of tuples of the kind $e_t = (s_t, a_t, r_{t+1}, s_{t+1})$, where s_t is the state at time t, a_t is the action taken from state s_t, r_{t+1} is the reward given to the agent at time $t+1$ as a result of the previous state-action pair (s_t, a_t) and s_{t+1} is the next state of the environment. Finally, have some methods useful during the training to 'push' new experience into the memory, to 'sample' a batch of experiences from it and to return the size of the memory.
- *policy_net.py*: Inside this file we find the class 'DQN' that defines the architecture for the policy and the target networks. It is a single hidden layer network that takes 128 features in input and returns 128 features in output

in the hidden layer. The activation function is chosen to be $tanh(x)$. Moreover, here is defined the function 'update_step()' which implement the optimization step in the following way:

  - a batch of experience data is sampled from the replay memory,
  - a mask that select only non-final states from the batch is applied,
  - the Q-values with the policy network are computed in train mode, the Q-values of the next states are computed with the target network in eval mode and we compute the expected (approximated) Q-values through the Bellman equation,
  - the loss between the Q-values obtained from the policy network and the expected Q-values is computed and the optimizer takes a step.

- *policies.py*: In this file we find the two functions that implements the ϵ-greedy policy and the $softmax$ policy. The first one chooses a non-optimal action with probability ϵ, in the second case a non optimal action is chosen following a $softmax$ distribution of the Q-values with a certain temperature T. Both the probability ϵ and the temperature T decrease over time following a properly defined exploration profile. The policy chosen for the optimization and the training in this task is the $softmax$ policy. In this case the exploration profile is chosen to be exponentially decaying in such a way that its shape does not depend on the number of iterations.

- *Hype_search_two_objectives.py*: In this file we find a script that performs an OPTUNA multi-objective optimization over the hyperparameters in listing 1. The optimization procedure takes into account two values, the first one is the average score over 1000 episodes that deals with the stability of the learning, the second one is the 'convergence' episode that is defined as the episode after which the average score is greater than 490. As a reward function for the agent we have that a fixed penalty of -1 is applied whenever the pole falls, and a linear penalty term is added when the cart is far from the central position. The multiobjective optimization cannot choose between the two values therefore it is unable to select a best trial. For this reason the configuration and the results of each trial is stored in a txt file named 'study_params.txt'. Finally, a plot with the pareto front of the study is produced. The results of this process can be found in the folder 'results optimization two objectives'.

- *Hype_search_one_objective.py*: In this file we find a script that performs an OPTUNA study that try to maximize only the average score over 1000 episodes. The OPTUNA objective function remains the one implemented in the multi-objective study, but this time OPTUNA performs a real optimization through the TPE sampler and pruning is made possible through intermediate values of the average score. The convergence episode is manually added as a trial attribute and returned as well. A txt file with the best parameters found is produced with the name 'study_bestparams.txt' and the plots that help to visualize the optimization process are stored in a folder named 'visualization'. The results of this process can be found in the folder 'results optimization one objective'.

- *training_loop.py*: Here the training loop with the optimized parameter is implemented and the plots with the exploration profile and the score history, the average score and the convergence episode are produced. Moreover, every 100 episodes a video of the game is stored in a folder named video.

In first place the optimization strategy chosen has been the multi-objective one. However, after a 50 trial study was performed and the pareto front plot produced, it has been noted that the two quantities to be optimized were strongly correlated. Therefore, the one-objective optimization procedure has been implemented and a 50 trial study has been performed. With the resulting hyperparameters the training procedure has been performed. Both the optimization and the training are performed over 1000 episodes with a batch size of 128, a total replay memory capacity of 10000, and a minimum number of sample in the replay memory to enable the training of 1000. The optimizer chosen for the optimization and the training is SGD without momentum and the loss function is nn.SmoothL1Loss() that creates a criterion that uses a squared term if the absolute element-wise error falls below 1 and an L1 term otherwise. All the scripts described above have been run in local, however in order to make it easier to test their functioning a Colab notebook named *CARTPOLE_environment.ipynb* has been produced. To run the code it is sufficient to open the notebook in Colab, upload the files in the folder 'Code' and run the cells. The results of the optimizations and the training are stored in the folder 'results' inside the folder 'cartpole_v1'.

```
28  #INITIAL TEMPERATURE OF THE SOFTMAX DISTRIVUTION
29  initial_temp = trial.suggest_int("max_temp", 4, 8)
30
31  #DISCOUNT FACTOR GAMMA
32  gamma = trial.suggest_float("gamma", 0.96, 0.99)
33
```

```
#NUMBER OF TIME STEPS BETWEEN TWO UPDATES OF THE TARGET NETWORK
target_net_update_steps = trial.suggest_categorical("target_net_update_steps", [3,5,10,15])

#LEARNING RATE
learning_rate = trial.suggest_float('learning_rate',0.001,0.1)
```

Listing 1: Hyperparameters to be optimized.

B. PacMan

The implementation of the agent that solves the PacMan environment follows the same structure of the one just described for the Cartpole environment:

- *replay_memory.py*: The replay memory for the PacMan environment is implemented in the same exact way of the one implemented for Cartpole environment.
- *policy_net_cuda.py*: In this file we find the class 'DQN_PIL' which implements the convolutional neural network that will approximate our optimal action-value function. The policy and the target networks have the following structure:
  - First convolutional layer: it takes 1 channel in input and returns 32 channels in output. It has a kernel of 8×8 pixels with a stride of 4 pixels and a symmetric padding of 1 pixel.
  - Second convolutional layer: it takes 32 channel in input and returns 64 channels in output. It has a kernel of 4×4 pixels with a stride of 2 pixels and no padding.
  - Third convolutional layer: it takes 64 channel in input and returns 64 channels in output. It has a kernel of 3×3 pixels with a stride of 1 pixels and no padding.
  - First linear layer: it takes 3136 features in input and returns 512 features in output.
  - Second linear layer: it takes 512 features in input and returns 9 features in output.

  The activation function is chosen to be the 'ReLU' function.
- *policies_PIL.py*: In this file we find several fundamental functions:
  - 'preprocess_observation' takes care of the pre-processing of the images returned by the GYM environment. It takes the $3 \times 210 \times 160$ numpy array that describe the image, transform it in a PIL image, convert it in grey-scale and resize it in a 84×84 image. Then, the PIL image is converted back in a numpy array with size $1 \times 84 \times 84$.
  - 'choose_action_epsilon_greedy' and 'choose_action_softmax' implements the ϵ-greedy and the $softmax$ policies.
  - 'update_step' implements the optimization state in the same way as it is described in the Cartpole section with the modifications required to handle the different input.
- *training_loop_pacman.py*: In this file is located the training loop that exploit the $softmax$ policy. It is implemented on top of the one used for the Cartpole environment, however some changes have been made. In particular, the reward from the game has been clipped to 1 so that it is easier to apply penalties and make tests. For the final training in particular a fixed penalty of 1 is applied whenever the value of the voice 'lives' in the next state informations changes. The optimizer chosen in this case is 'Adam' while the loss function remains the nn.SmoothL1Loss() exploited also for the Cartpole environment.

Different tests have been performed during the implementation, however due to the computational effort required it has been difficult to produce meaningful results out of them. $softmax$ and ϵ-greedy policy have been explored within the environment. Moreover, different penalties have been tried out, choosing at the end the one described above. For a limited number of episodes (< 1000) the agent seemed to not improve at all.

```
#HYPERPARAMETERS
gamma = 0.99                    #DISCOUNT FACTOR
replay_memory_capacity = 50000  #REPLAY MEMORY CAPACITY
lr = 0.00025                    #LEARNING RATE
target_net_update_steps = 5     #NUMBER OF EPISODES BETWEEN TWO UPDATES OF THE TARGET NETWORK
batch_size = 48                 #SIZE OF THE BATCH SAMPLED FROM THE REPLAY MEMORY
```

```
bad_state_penalty = -1.          #PENALTY TO ASSIGN WHEN THE PACMAN DIES
min_samples_for_training = 2000 #MINIMUM SAMPLES IN REPLAY MEMORY TO ENABLE THE TRAINING
num_iterations = 4000            #NUMBER OF EPISODES
initial_value = 8                #INITIAL TEMPERATURE
```

Listing 2: Hyperparameters used to train the agent over the PacMan environment.

The most significant strategy that has been implemented to improve the performance has been to modify the action space of the agent. Indeed, the game can be played also with the reduced action space containing only the actions: NOOP, UP, RIGHT, LEFT and DOWN. This test have been made in a training session of $\sim$ 1600 episodes resulting in no significant improvement with respect to the agent equipped with the complete action space. After this trials, and having reached the implementation that can be seen in *training_loop_pacman.py* a long training has been launched in order to produce final results. The training has been launched with the parameters in Listing 2 exploiting a $softmax$ policy. It took a week to complete the 4000 episodes locally on a intel laptop without GPU. As for the Cartpole environment, a Colab notebook named *PAC_MAN_environment.ipynb* is produced in order to simplify testing the code. To run the script it is sufficient to open the notebook in Colab, upload all the file in the folder 'Code' and run the cells. All the results, including the video recordings, are stored in the folder 'pacman_results' inside the folder 'pacman_env'.

III. RESULTS

A. Cartpole

In FIG.1 it can be seen the relation between the two objective values of the optimization procedure implemented in 'Hype_search_two_objectives.py'. As we can see it seems to be a strong relation and therefore just one of the variable is needed to find a good combination of hyperparameters.

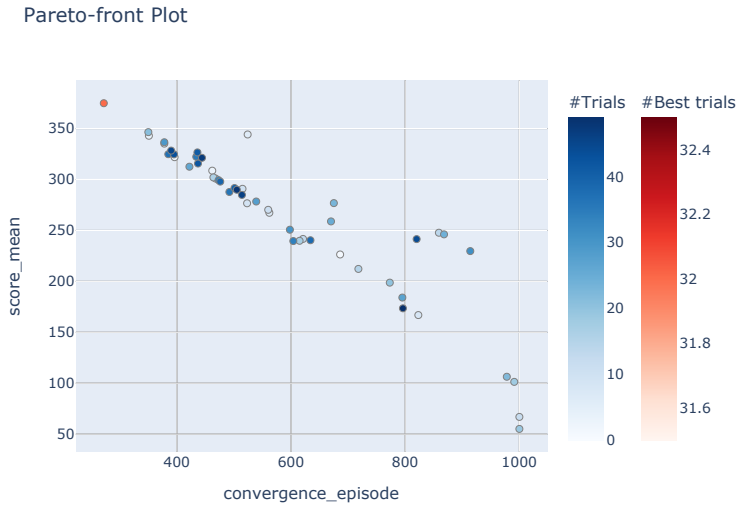

FIG. 1: Pareto front plot of the 2-objectives optimization.

The results of the optimization procedure that aims to maximize the average score are shown in TABLE I and in TABLE II. In FIG. 2 are reported the optimization hystory and the hyperparameters importance.

Study numbers
Number of finished trials: 50
Number of pruned trials: 41
Number of complete trials: 9
Best average score: 388.674
Convergence episode: 241

TABLE I

Hyperparameters found:
Initial temperature: 8
Discount factor: 0.969
steps before target network update: 3
learning rate: 0.0799

TABLE II

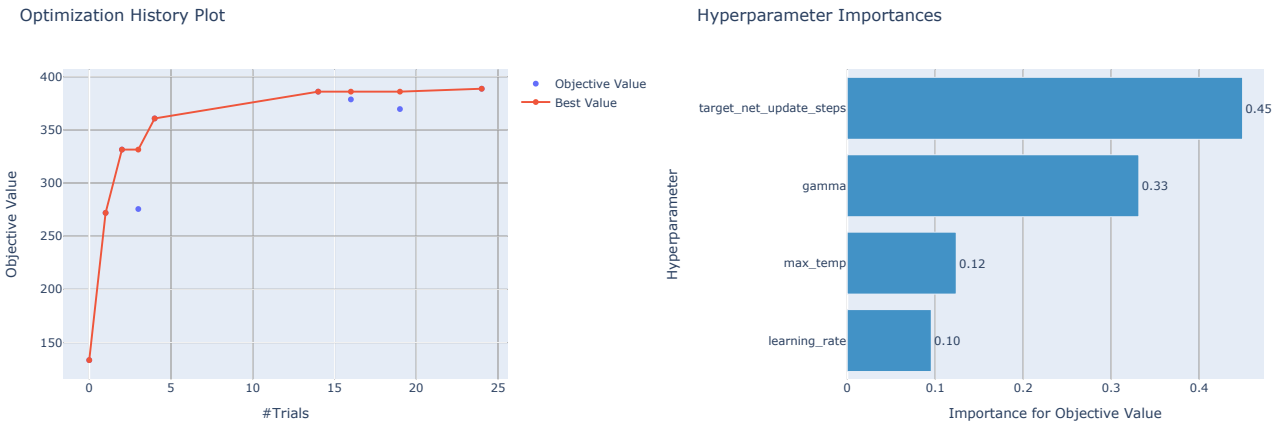

FIG. 2: Left: Optimization hystory; Right: hyperaparameters importance

In FIG. 3 are reported the results of the training with the best configuration of hyperparameters and the exploration profile used both for the optimization and the training. It can be seen how the agent begins to obtain the maximum score as the exploration reaches very low values. From the videos recorded during the training it can be seen that after 300 episodes the agent is already able to practically solve the game, however it keeps moving a lot in order to maintain the pole up. After 600 episodes the cart is able to remain at the central position for all game duration.

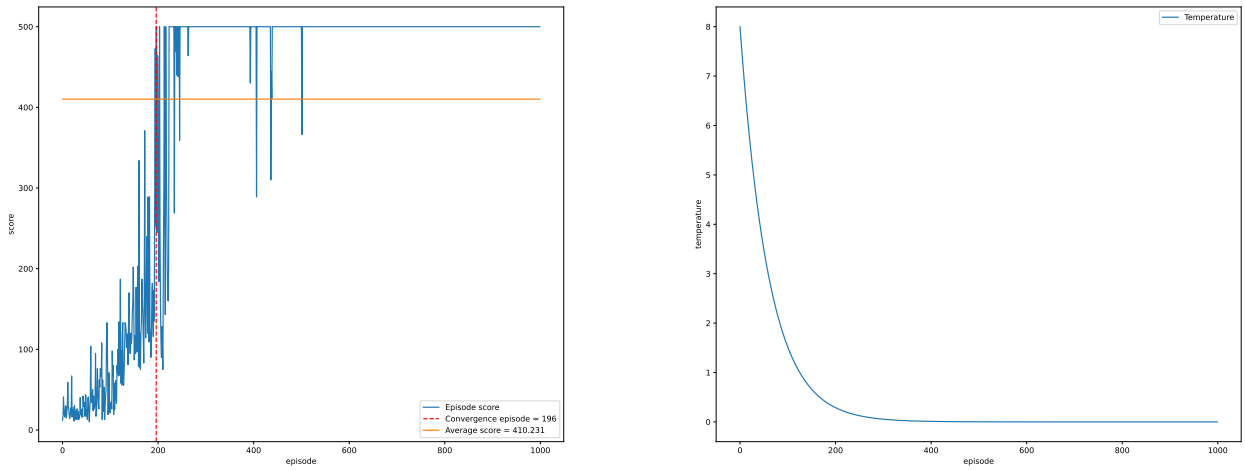

FIG. 3: Left: Scores history, average score and convergence episode; Right: exploration profile

B. Pac-Man

In FIG.4 Are reported the original output image of the environment and the image as it is when it is given in input to our agent. It can be noticed that all the main features of the image seems to be well preserved after the pre-processing. In FIG.5 are reported the results produced during the 4000 episodes training. It can be seen that the scores fluctuate a lot during all the training, however from the moving average plot it can be seen that the scores actually improves at least a little. The improvements are more evident if one looks at the videos. Indeed, in the first 1000/1500 episodes the agent goes from performing completely random actions to being a little more stable in his movements, this is due to the rapid fall of the exploration profile. In the remaining 2500 epochs however the agent seems to keep learning

gaining more fluency in his behavior. One of the most important features, that of escaping from ghosts, has not been properly learned by the agent. Only in the videos concerning the episodes from 3600 onwards we can perhaps notice some 'instinct' in this sense. Some videos have been deleted from the results in order to stick with the upload size limits in moodle.

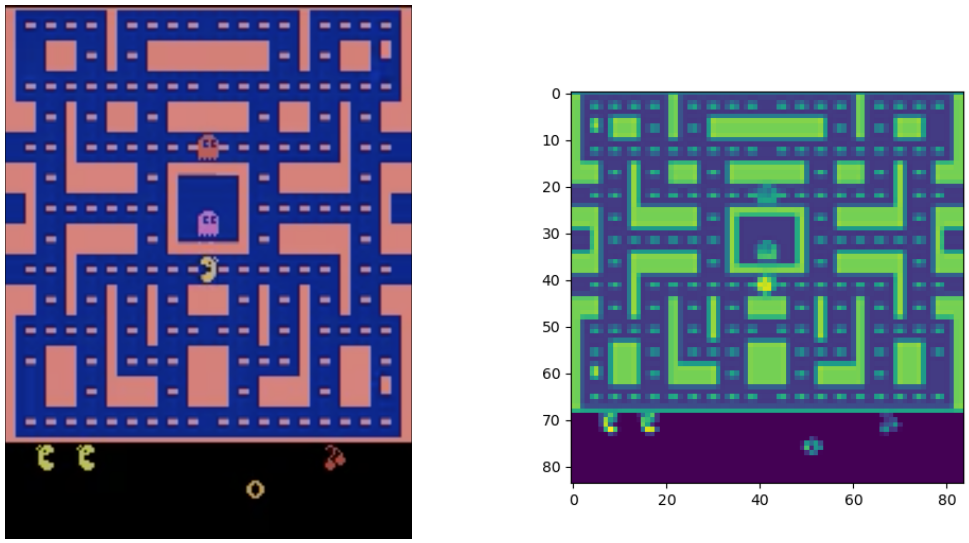

FIG. 4: Left: example of an RGB image returned by the environment; Right: example of a preprocessed image, observation of the agent.

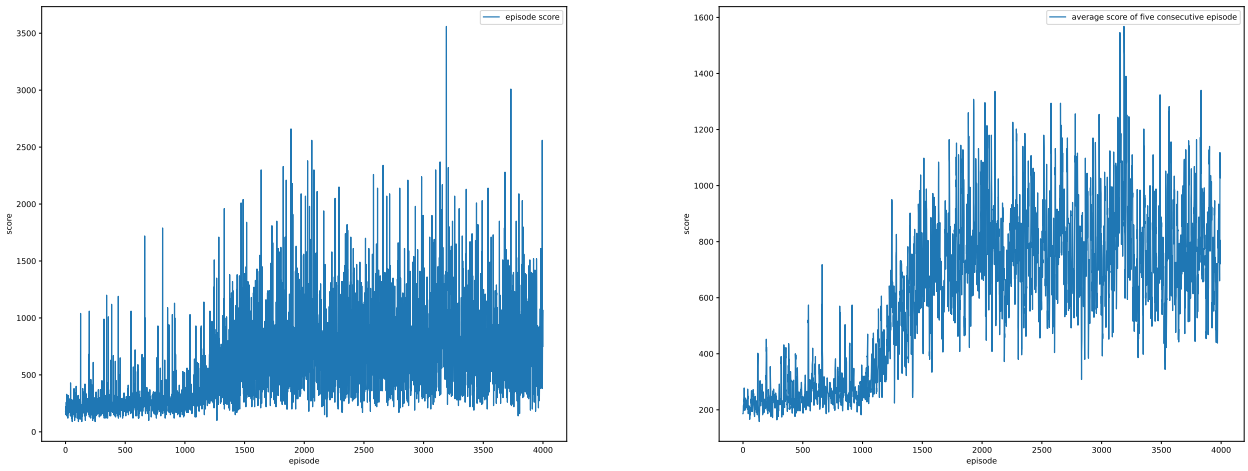

FIG. 5: Left: plot of the scores performed by the agent during training; Right: moving average plot of the scores performed by the agent, with a pass equal to the number of steps between two target network updates.

IV. CONCLUSIONS

To conclude, one can say that both the tasks proposed has been explored. While the game in the first task has been properly solved, for the PacMan environment one cannot say that. In fact the agent is not able to properly play the game but still something has been learned in the sense that it does not act completely random.

Appendix A: Cartpole

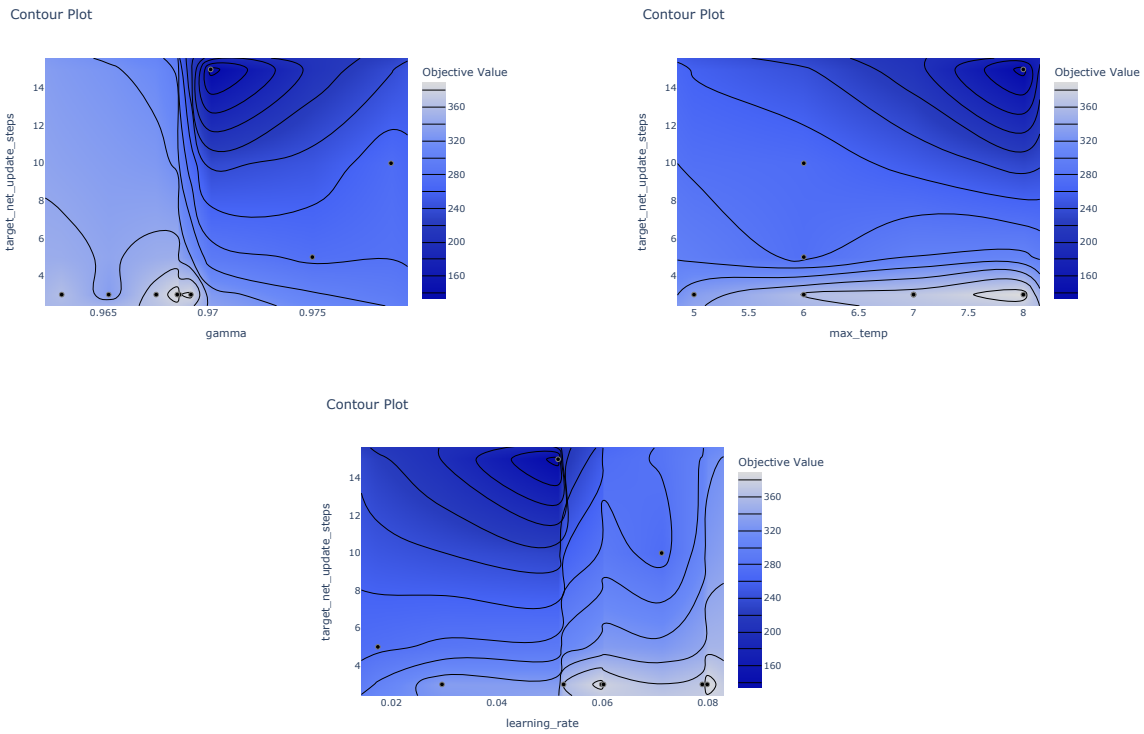

FIG. 6: OPTUNA countur plot of the most important parameter, the number of episodes to wait to update the target network, with respect to the others three parameters, the discount factor γ, the initial value of the temperature in the $softmax$ distribution and the learning rate

Appendix B: PacMan

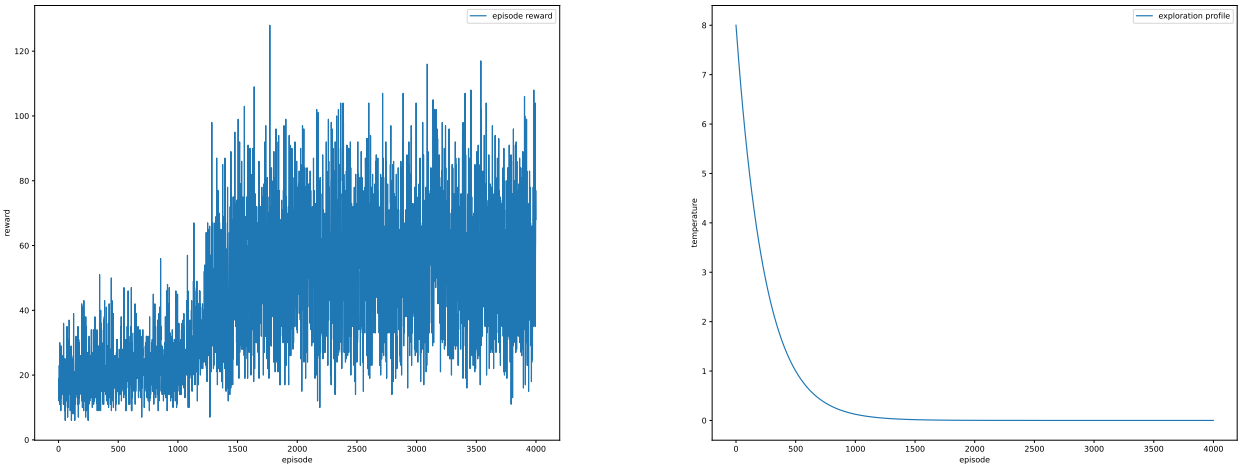

FIG. 7: Left: Reward history; Right: exploration profile.