# Second part for the project, Deep RL - MVA 2022/2023

Erwan Fagnou erwan.fagnou@telecom-paris.fr
Marc Pierre marcpierre1999@gmail.com
Denis Duval denis.duval13@gmail.com
Dorian Gailhard dorian.gailhard@telecom-paris.fr

June 27, 2023

## 1 Introduction

The objective of this project is to maximize the score of an agent in a custom FlappyBird environment. To do this we implemented DQN, a Deep RL algorithm, which is a way to extend classical Q-Learning with neural networks. We will see in the following the different hyperparameters and the choices we had to make to obtain the best possible score.

## 2 Method description

### 2.1 Free RL attempt

Initially, we tried to design an algorithm without any learning process (in particular without deep reinforcement learning). The idea was to find some simple rules, heuristics, that together could solve the problem. We started from the baseline and then extended the idea of a target altitude, giving the agent a variable target altitude at each time. The two difficulties were to find the right 'target' altitude at each moment, and a way to get there with a displacement that anticipates as well as possible the too abrupt changes of speed (according to the y axis). After some tuning of these rules, the results were quite bad even if they exceeded the baseline. We think that it would have been too much to tune all these rules and that reinforcement learning is a set of methods that can find these rules.

### 2.2 Learning method

We started by using a simple DQN algorithm. More specifically, we minimise the TD error between a target and an online network, while updating the parameters of the target network using a convex combination of these and the online parameters. Our final agent actually uses a variant, Double DQN, which improves the loss function to reduce the bias of the online network. We noticed it made the training more stable.

The exploration is done using the $\epsilon$-greedy method. The explored trajectories are stored in a replay buffer, as usual. We did try to use a prioritized replay buffer, and while it made the learning faster in the beginning, it slowed down the computation time by 2.5 times, so we decided not to use it in the final method.

We also tested an alternative equivalent reward system that rewarded the model for simply staying alive – which leads to the same optimal policy – and it did help the agent to learn better. The intuition was that this reward was simpler to predict than the one from the environment. We however reverted back to the original one, because the improvement was not worth it.

During the training, we regularly evaluate the agent, and keep the 3 best ones. At the end, we test each of them for 100 episodes to select the one with the best average reward as the final model.

### 2.3 Architecture detail for the prediction of the Q-function

Initially we tested a convolutional neural network in which we fed the "image" of the game map. Then we replaced it with a multilayer perceptron. We observed that centering the image around the bird increases the agent's performance.

We however totally changed the approach, and our final method directly gives the agent useful information. At first we simply gave the position and velocity of the bird, and the information of the two closest top and bottom obstacles. This was problematic for situations like Fig. 1, when multiple pillars are adjacent and the agent is unable to anticipate the second one, so we added a few additional information to the agent input that solved the issue:

- distance to the ground or to the bottom obstacle if there is one
- distance to the ceiling
- distance to the obstacle in front of the bird
- nature of the obstacle in front of the bird (bottom or top pillar)

This information is then fed into a multilayer perceptron with 3 hidden layers of 50 neurons each with ReLU activations.

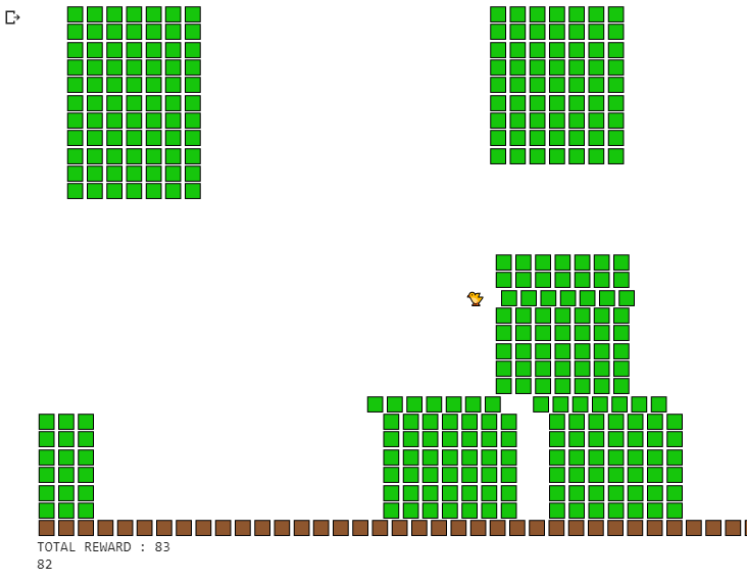



Figure 1: One example of the situations in which our model fails.

# 3 Choice for the hyperparameters

Most of the hyperparameters were chosen empirically by running multiple tests. A few of these parameters are worth mentionning:

## 3.1 Replay buffer size

The size of the replay buffer controls the maximum number of game experiences that can be stored. We noticed that the larger the size, the more stable the learning was, so we increased it to 50,000. We hypothesize this is because Q-learning is an offline method, so the more data the better.

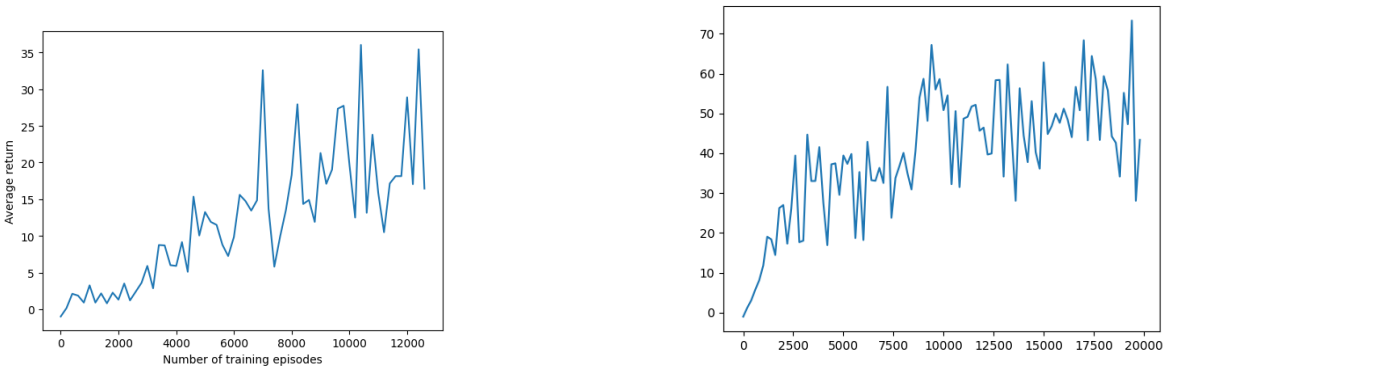
## 3.2 $\epsilon$-greedy exploration

Different values of $\epsilon$ did not change much the performance of the method, so we kept it very high to 30%, thus adding a lot of varied trajectories in the replay buffer.

## 3.3 Target network update rate (*ema*)

The higher this parameter is, the longer it can take to learn the environment. In our first attempts, we set it to zero so that the agent would learn quickly, but the learning was too unstable. However we noticed that with a high *ema* DDQN works better, and allows us to have better results than in the classical case.

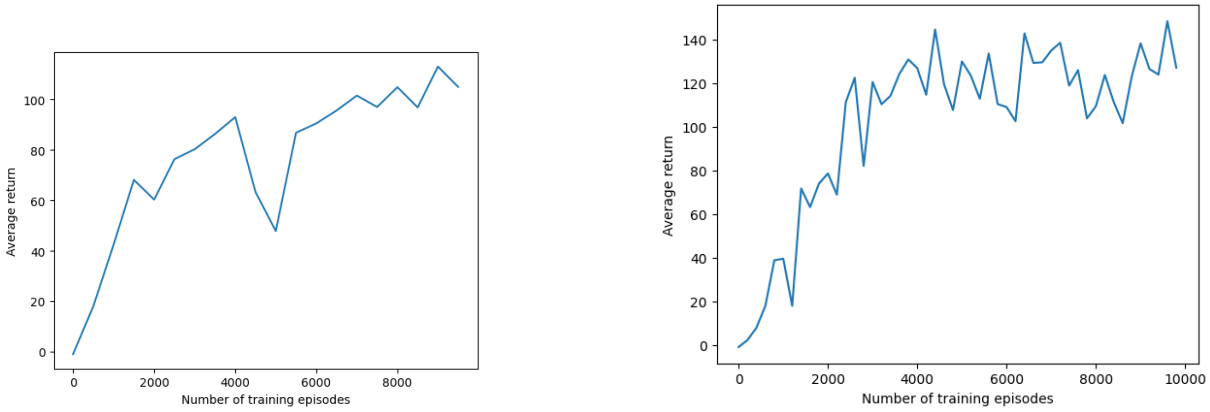
# 4 Results




First results, with a CNN  Centering the agent, simplifying the image and using a common NN

Figure 2: Results using an image as the input

Above we can see the results that we obtained using an image of the environment, encoding each pixel with a value in -1, 0, 1. The right plot shows the best results we have got with the method of using an image (simplified) of the environment to the network. The right plot was obtained with a way bigger buffer : 50000 compared to 1000 on the left.




First results Adding inputs

Figure 3: Results using an array of information

Above we can see the results that we obtained using an array containing all the necessary information for the agent. The main difference in the algorithms that led to the two plots is that the plot to the right is obtained when we fed also the second obstacle incoming and not just the next.

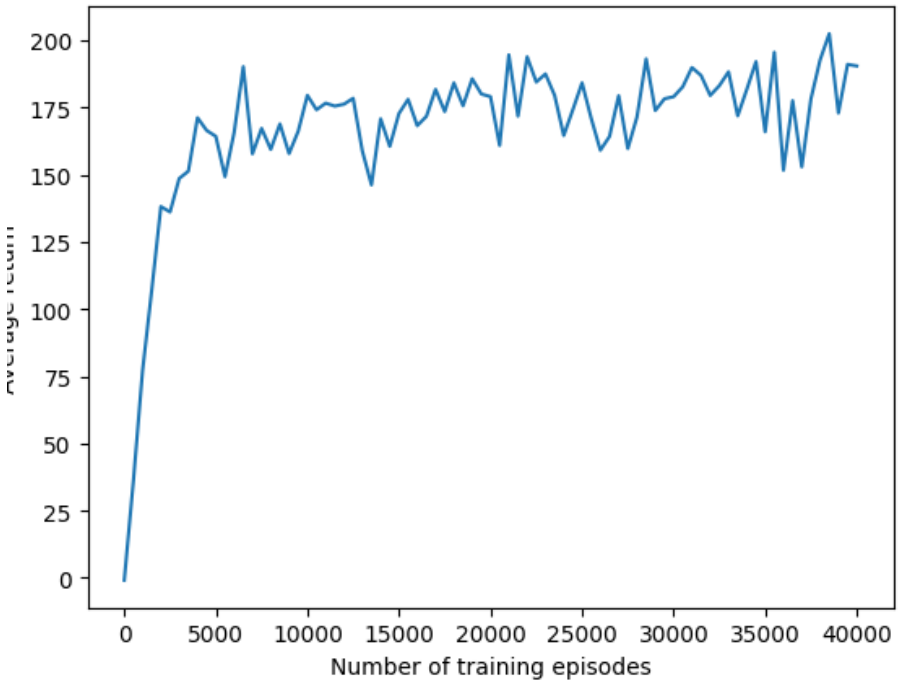



Figure 4: Evolution of the average reward over 100 episodes during training, with the final architecture and parameters.

Above we can see the final results that we obtained, with almost 200 of average reward after training.

## 5 Conclusion and acknowledgements

While the final performances are very promising, the agent barely improves after 20,000 episodes, and still makes a few mistakes. We envisioned modifying the environment to create a difficulty parameter such that, as the model becomes better, we could increase the prevalence of such structures to skip the easier ones that the model already knows, in order to save precious steps and training time, but modifying the environment was forbidden. We also briefly envisioned increasing $\epsilon$ (the exploration parameter) to increase the difficulty for the model as it then has much fewer steps to escape difficult situations and sometimes find itself in dire ones.

We also thought about programming an algorithmic model able to perfectly play the game, based on simple trajectory computations and a decision tree, so then the RL model could learn by imitation, but ultimately we found it tricky and abandoned the idea in favor to a simpler RL approach with worked really well.