Playing Atari with Deep Reinforcement Learning

Benjamin BOURBON

MS HPC-AI Mines ParisTech

I. Overview

Atari 2600 games is a challenging Reinforcement Learning test bed that presents high dimensional visual input (210×160 RGB video at 60Hz) and tasks that were designed to be difficult for human players. We apply Deep Q-Network (DQN) on the game *Ms-Pacman* implemented in the Arcade Learning Environment (ALE).

We consider the sequence of actions, observations and rewards, the tasks in which an agent interacts with an environment. At each time-step, the agent selects an action from 0 to 9 for our game. They represent movements allowed by the joystick on a real machine. The action is applied to the emulator and modifies the state of the game. The agent observes an image which is a vector of raw pixel values representing the current screen. As well, it receives a reward r_t representing the change in the game score.

The goal of the agent is to maximise future rewards by interacting with the emulator. Future rewards are discounted by a factor of γ per time-step. We can define the future discounted return at the time t as :

$$R_t = \sum_{t'=t}^{T} \gamma^{t'-t} \, r_t$$

where T is the time-step at which the game terminates.

Bellman, who worked on Markov decision process (MDP), found a way to estimate the optimal action-value $Q^*(s, a)$ which is the maximum expected return achievable by following any strategy, after seeing some sequence s and then taking some action a :

$$Q^*(s, a) = \max_{\pi} \mathbb{E}[R_t | s_t = s, a_t = a, \pi]$$

where π is a policy mapping sequences to actions.

The optimal action-value function obeys the *Bellman equation*. If the optimal value $Q^*(s', a')$ of the sequence s' at the time-step was known for all possible actions a', then the optimal strategy is to select the action a' maximising the expected value of $r + \gamma \, Q^*(s', a')$:

$$Q^*(s, a) = \mathbb{E}_{s' \sim \mathcal{E}}[r + \gamma \max_{a'} Q^*(s', a') | s, a]$$

where $\mathcal{E}$ is the environment of the game.

After getting the optimal Q values, it is simple to define the optimal policy $\pi^*(s)$. When the agent is in the state s, he has to choose the action which has the highest Q value in this state, in other words $\pi^*(s) = \underset{a}{\text{argmax}} \, Q^*(s, a)$.

We can use the Bellman equation as an iterative update $Q_{i+1}(s, a) = \mathbb{E}[r + \gamma Q_i(s', a') | s, a]$ in order to converge to Q^* when $i \to +\infty$. But as we also know in reinforcement learning context, it is

common to use a function approximator to estimate the action-value function $Q(s, a; \theta) \approx Q^*(s, a)$. Neural networks can be trained by minimising a sequence of loss functions $L_i(\theta_i)$ that change at each iteration i,

$$L_i(\theta_i) = \mathbb{E}_{s,a\sim p(s,a)} \left[(y_i - Q(s, a; \theta_i))^2\right]$$

where $y_i = \mathbb{E}_{s'\sim\mathcal{E}}[r+\gamma max_{a'} Q(s', a'; \theta_{i-1})|s, a]$ is the target for iteration i and $p(s, a)$ is a probability distribution over sequences s and actions a that refer to as the *behaviour distribution*.

II. Application of Reinforcement Learning

i. Algorithm

Deep Q-network is an algorithm which was designed to work on high dimensional value input. There are more than 250 dots to eat and because a dot can be eaten or absent, the number of possibilities is more than 2^{250} which is impossible to save an estimation for each Q-value. The solution as we explained, is to find an approximation of Q values such as find the $Q(s, a; \theta)$ function. Using convolution neural network, it can avoid the hard work to find good features. Moreover, CNN produce better results, especially on complex problems. In 2013, DeepMind introduces DQN with two specific points : the algorithm has a replay memory (as we will explain it after) and the algorithm has two DQN. The policy DQN is the one which plays and learns at each training iteration. The target DQN is used only to calculate target Q-values. At each regular steps, the parameters of the policy DQN are copied to the target DQN.

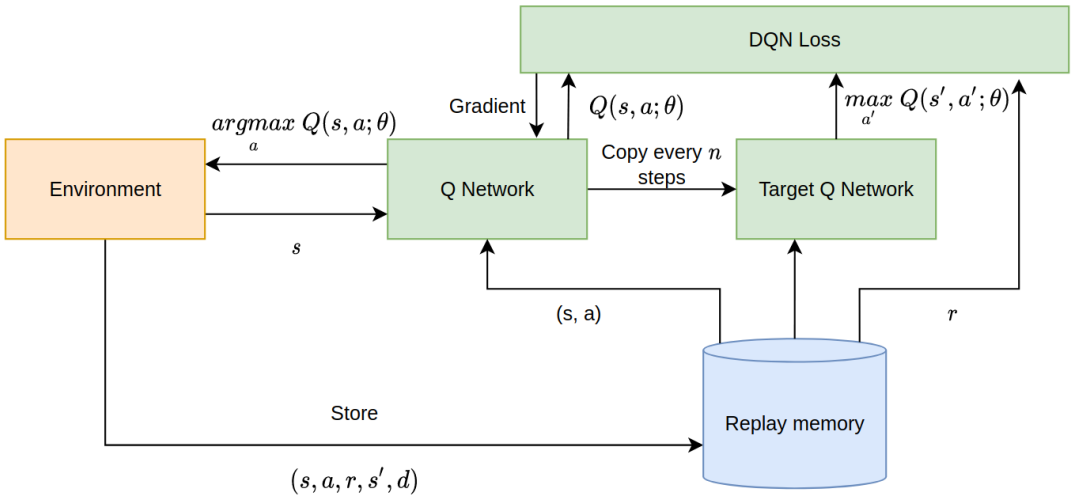

Figure 1: *Deep Q Network algorithm with replay memory, (policy) Q-network and the target Q-network*

ii. Preprocessing

The game is defined on an input which is 210×160 RGB images as we said. These images are overly large. To reduce the size, we chose to take one column in two and one line in two and to crop the final result. In addition, there are 3 channels for colors, we simplified them in only one channel (black and white) and change the color of Ms Pacman to constrast the character. The color can be obtained with any software that has the *pipette tool*. After, we normalize the values to have a range between -1 and 1. Note that we stack four observations : the current one and the previous observations. At the end, we have a state described by $88 \times 88 \times 4$ values.

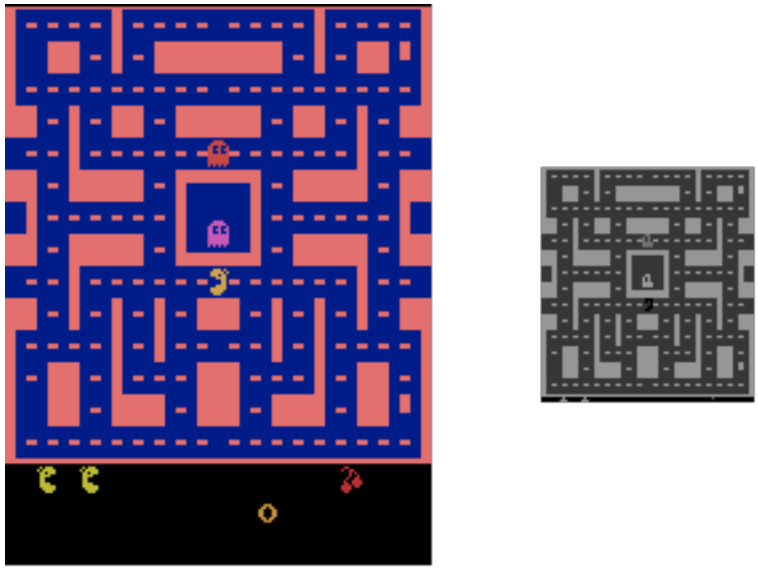

Figure 2: *Visualization of preprocessing : on left, the original image, on right, the preprocessed observation without normalization*

iii. DQN architecture

As we said, DQN is a neural network and because the game offers high dimensional visual input, we implemented a Convolution Neural Network (CNN). It has three convolution layers along with a batch normalization for accelerating deep network training and the activation function. We chose the rectified linear unit function (ReLU on figure 3).

As it's shown on figure 4, after convolution layers, there is a hidden layer with 512 neurons and finally the output for the 9 discrete actions of the game. We recall that the input is 4 observations stacked, in other words, the input size is $88 \times 80 \times 4$. To get a whole output size between convolution layers, we ajusted padding to respect the following formula :

$$W_o = \frac{W_i - K + 2\ P}{S} + 1$$

where W_o is the output size and should be an integer, W_i is the input size, K is the kernel size, P is the padding and S represents the stride.

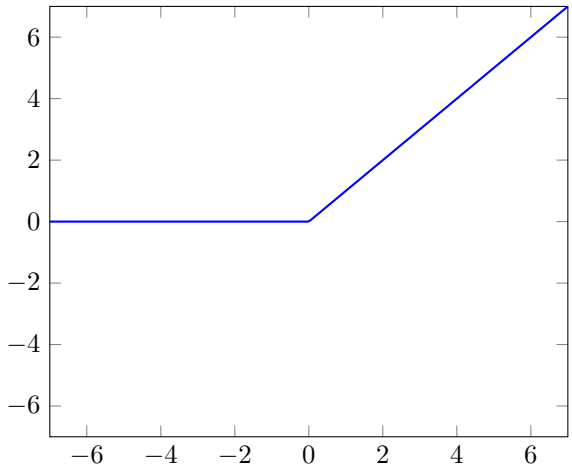

Figure 3: *Rectified linear unit function*

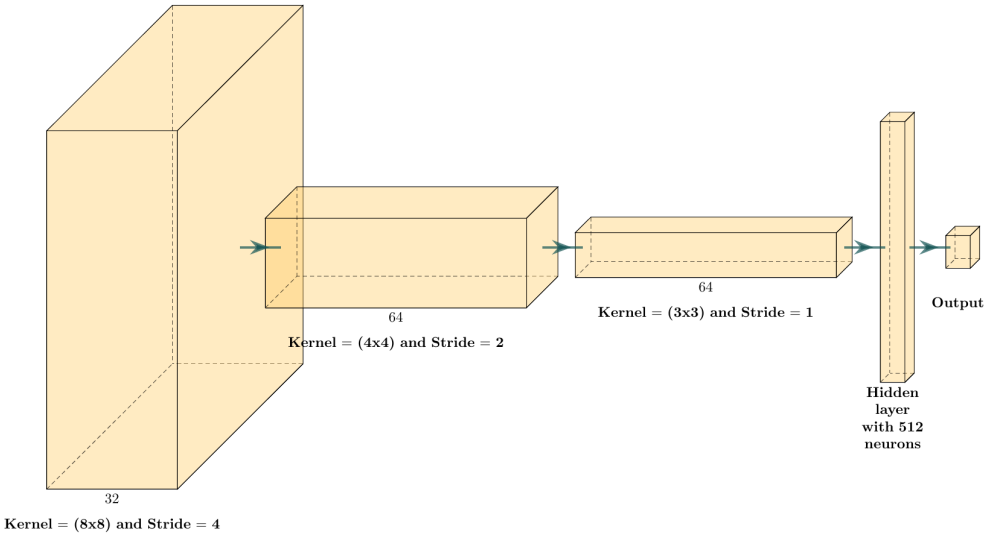

Figure 4: *Architecture of the Convolution Neural Network*

iv. ϵ-greedy strategy for exploration

The ϵ-greedy strategy is known to let the agent to explore the environment. We started with $\epsilon = 1$ at the beginning of the game and finished with $\epsilon = 0.1$ at the end on one million time-steps. In other words, the agent chooses randomly actions when the game starts and chronically and linearly, the probability to choose a random action decreases to 0.1 and thus the agent plays to maximise rewards 90% of the time. We compared two approaches to implement the ϵ-greedy strategy. The first one is using a linear progression and the second one decreases exponentially.

As we can see on the Figure 5, with same parameters, the exponential function is clearly slower than the linear function. Even if we change parameters to approach the linear behaviour, we don't think it has better advantages. That is why we chose linear function.

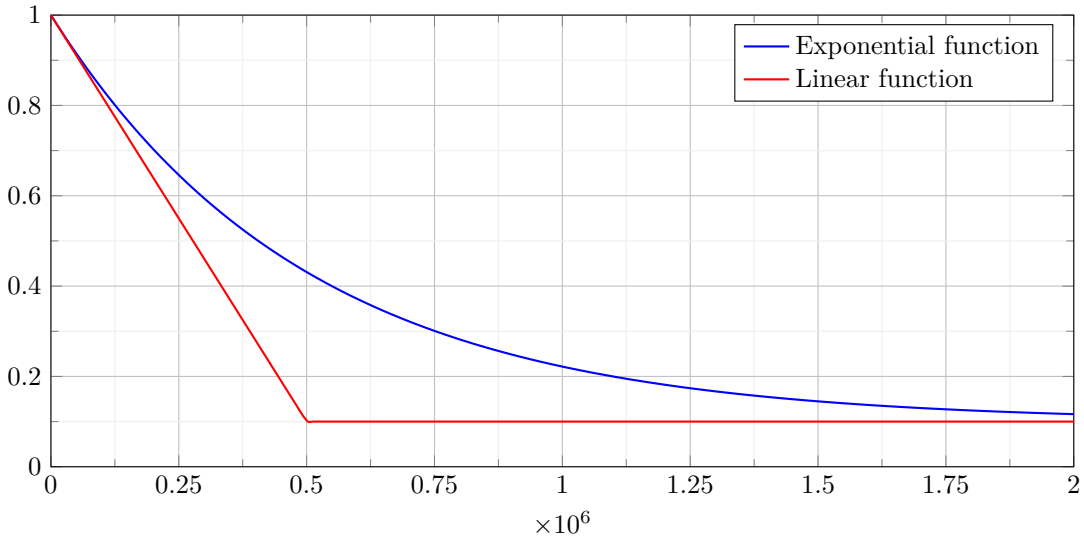

Figure 5: *Comparaison between two functions for ϵ-greedy strategy with $\epsilon_{min} = 0.1$, $\epsilon_{max} = 1$ and $\epsilon_{decay} = 500,000$*

v. Replay Memory

The algorithm DQN stores the experiences in a large replay memory and then for each iteration of training, it selects randomly a sample of this memory. This allows to reduce the correlation between experiences in a training batch and it helps greatly the training along. We recall the size of a state : $88 \times 80 \times 4$ and because we store the state, the action, the reward, the next state and if the game was done. That means we have at least around 225 Ko for one save. With a capacity of $25,000$ saves, it represents at least 5.6 Go. So we choose $25,000$ for the capacity and 32 for the batch size when we sample the replay memory.

vi. Rewards

The environment (ALE) offers default rewards but they represent only the additional points gotten by the agent during the game. In other words, when the agent eats a pill, it receives 10 points; with a boosted pill, 50 points and when it eats a ghost, it gets 200 points. There is no negative reward by default. Sometimes, it could be some combo points but we don't want to go more in details because it is not enough frequent.

We suggest different approaches :

- let all rewards like it is currently
- add negative rewards when it dies
- normalize rewards
- add negative rewards when the agent doesn't get a pill enough quickly

We think there are two difficulties with rewards : the quantification and the proportion. Adding the last point can quickly leads to only negative behaviour because the agent is not able to find a good behaviour to avoid negative reward. Not adding negative rewards means that the agent doesn't take account the effect of death and then the agent doesn't try to avoid ghosts or to find a way to collect quickly pills.

III. Results

Before explaining our results, we want to notice an important point. We have tried a lot of configurations which will not be covered here. And because we got bad results (no improvement of rewards) we have implemented the *Dueling Network Architecture* which is basically the same architecture as we currently have but with small modifications. The output is different : we maximise the following Q-value :

$$Q(s,a;\theta,\alpha,\beta) = V(s;\theta,\beta) + \left(\mathcal{A}(s,a;\theta,\alpha) - \frac{1}{|\mathcal{A}|}\sum_{a'} A(s,a';\theta,\alpha)\right)$$

In the figure 6, the different is more explicite.

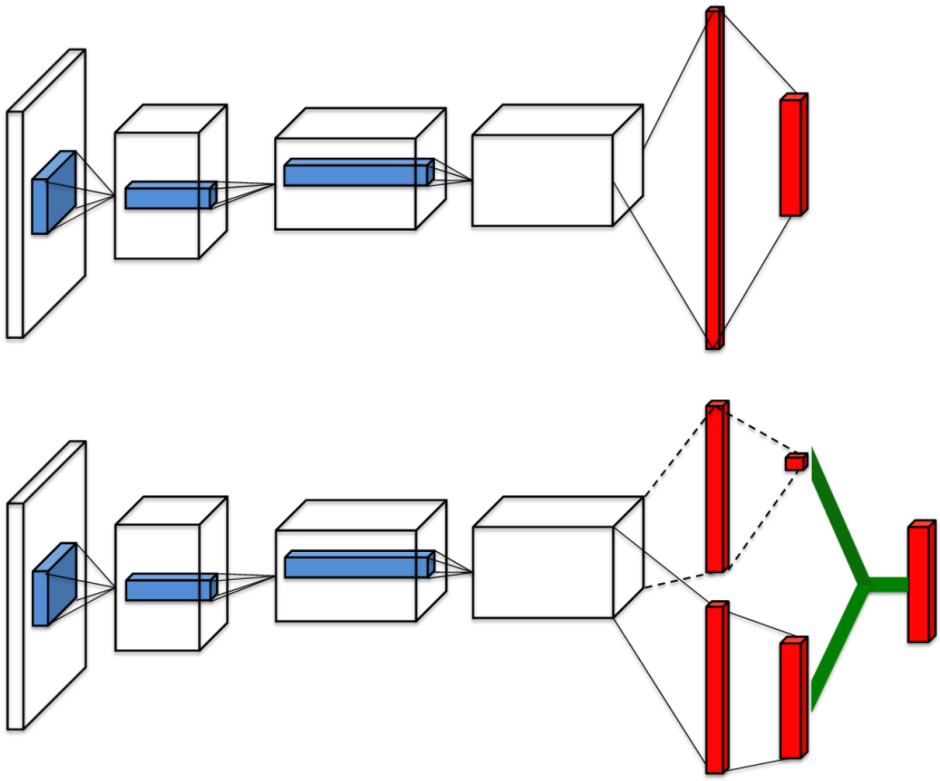

Figure 6: *Standard architecture (above) versus dueling architecture (below)*

We choose this architecture in addition with our current architecture because in literature, it promises good results and in the original paper, the game *MsPacman* has 48.92% of improvement. We want to highlight that *MsPacman* is a hard game for the algorithm because :

- the agent often is stuck in a corner
- ghosts don't appear on every frame due to the age of the game
- the character sometimes could disappear for no reason
- the game offers boosted pill which let the agent to eat ghosts whereas in most situations, it dies by touching ghosts

The best type of algorithm is the *Rainbow DQN* which outperforms other algorithms on score. Human play remains the highest score achieved for the moment.

We introduce our result to explain difficulties and our final state. For every simulation, we collected the loss of the current game, the rewards of the current game, the evolution of the total of rewards, the evolution of the mean of rewards, the evolution of the total maximum Q values, the evolution of the mean of Q values.

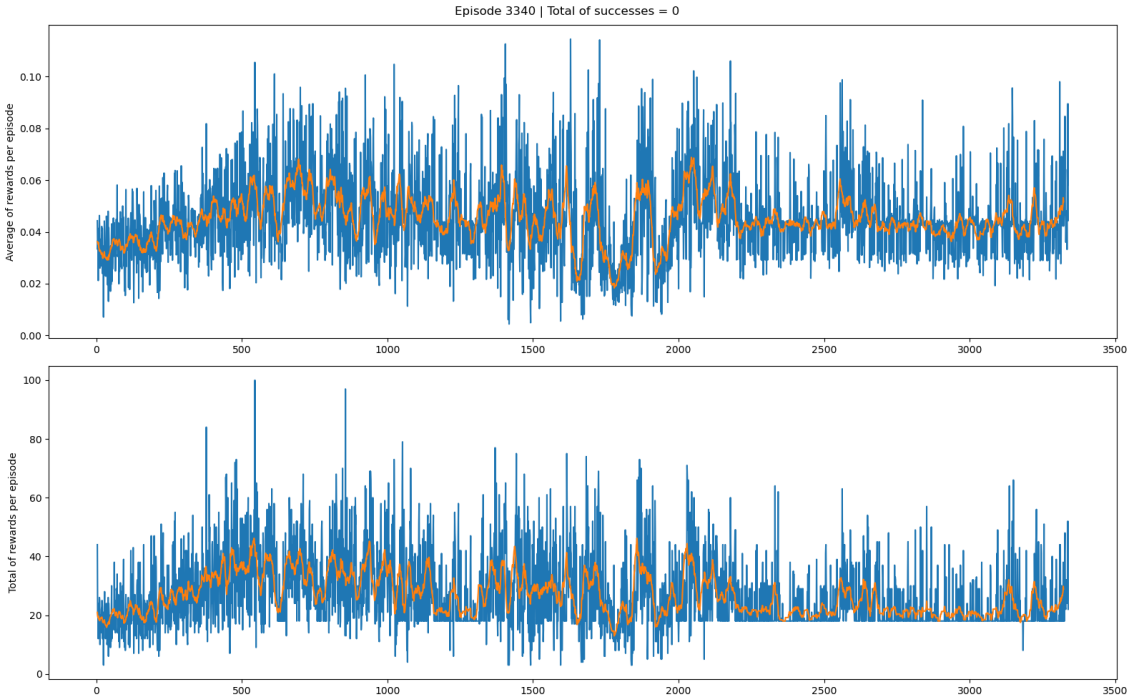

Figure 7: *Result of a simulation where* $\epsilon_{min} = 0.01$ *where the agent falls in a local minima*

On the figure 7, the last episodes, we can see a *"line"*. It shows that the agent is acting the same way on every episode. The total of rewards are not the same because randomly (10% of the time), the agent takes a random action and it "unlocks" his behaviour for a short moment. But it doens't learn enough to not repeat the same process. The only idea to avoid it we had, was to use SGD optimizer with momentum and Nesterov method. We encounter this behaviour again on other simulations.

On the figure 8, the decay of ϵ-greedy was $1,000,000$ from 1 to 0.1 with a discound rate 0.99. Around the 1400-th episode, the score doubles and varies. The trend of Q-values, on figure 9, is not smooth. The variance on rewards and Q values is high but we didn't find a way to reduce it.

When we analyse the behaviour through videos, the agent tends to go on the same way sometimes and not trying to collect other pills on other way. In addition, the boosted pills often attract it because it's the only way to eat ghosts. Nevertheless, after eating one, the agent don't chase ghosts.

On the figure 10 and on the figure 11, we wanted to see if the agent could learn something after a lot of episodes. But as the figures show, it stays on the same level.

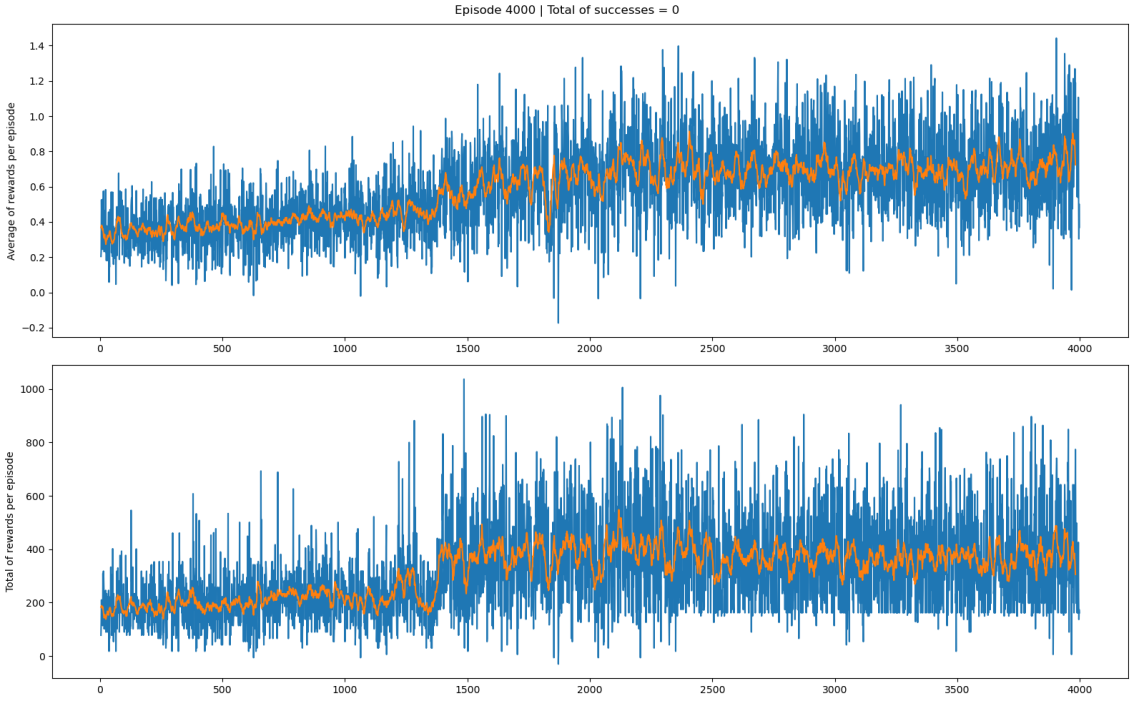

Figure 8: *Rewards on simulation with negative rewards and ajusted positive rewards*

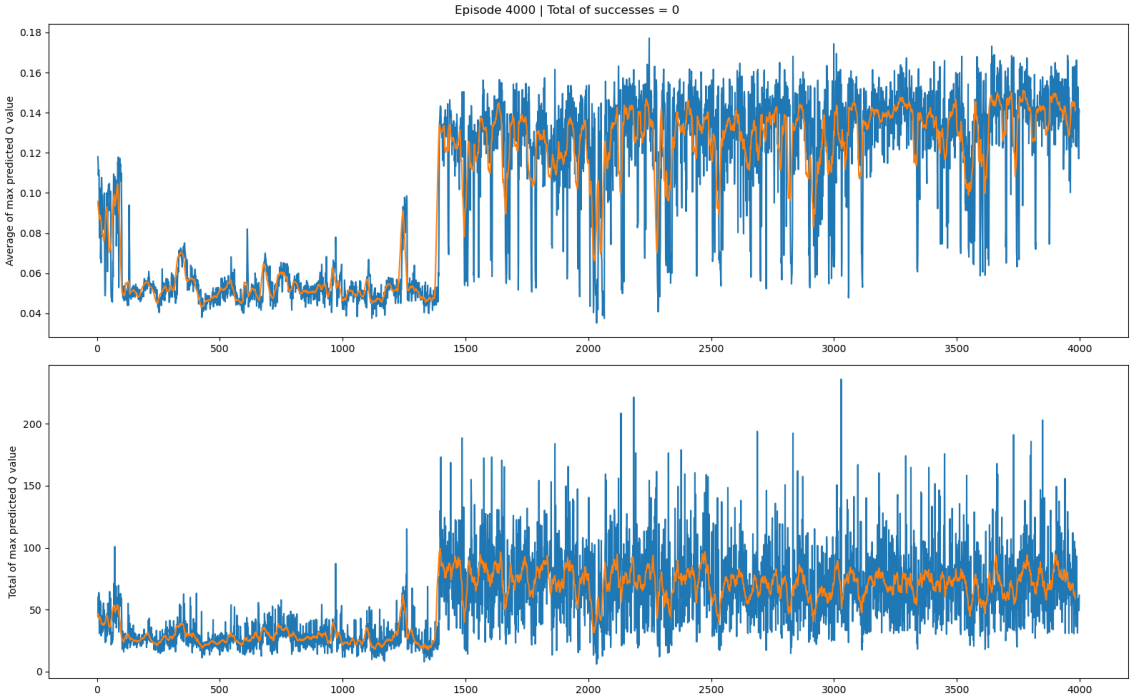

Figure 9: *Q-values on simulation with negative rewards and ajusted positive rewards*

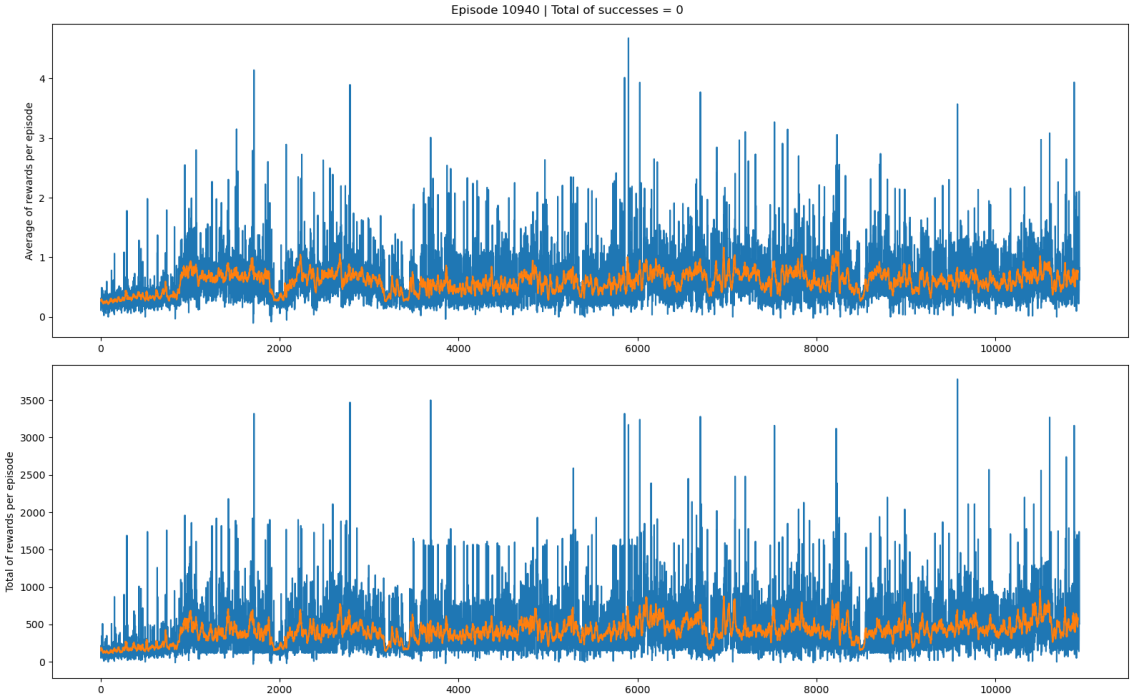

Figure 10: *Rewards on the longest simulation*

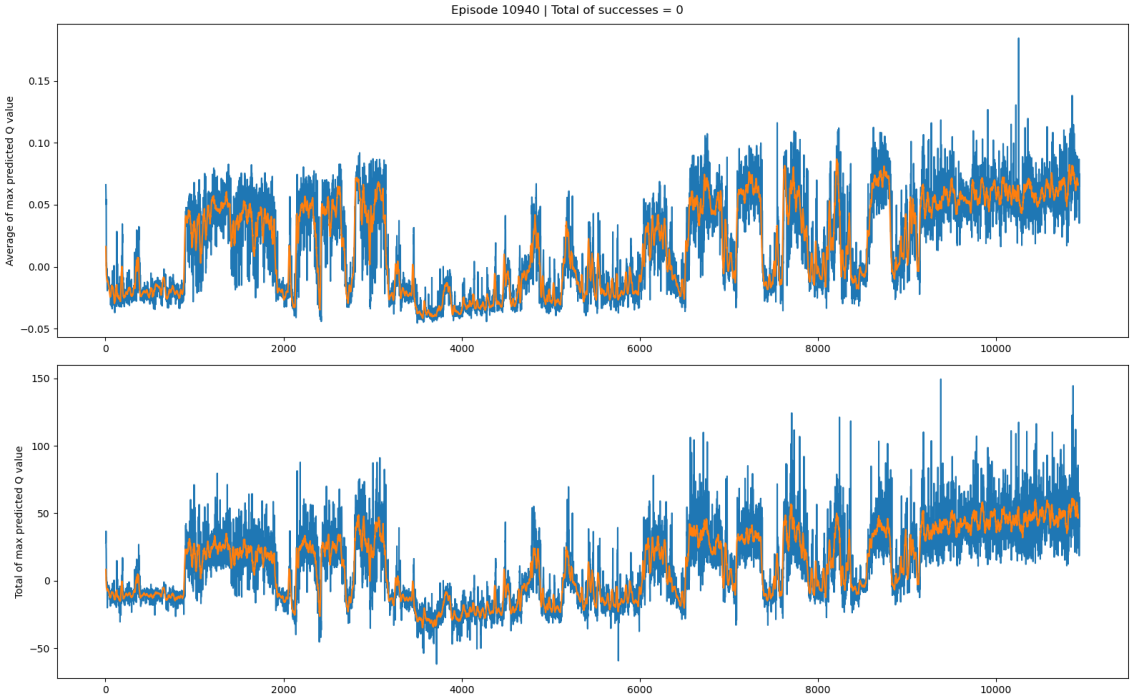

Figure 11: *Q-values on the longest simulation*