Stability of Deep Q-Learning in a confrontational environment

Sacha Braun, Lucas Gascon, Hugo Malafosse

***Abstract*—The goal of this project is to code a pygame environment where several agents will confront each other: on a discrete mesh of space, an agent of type "Cat" pursues an agent of type "Mouse". As soon as the cat catches the mouse, the game stops. The objective of this project is to successfully train adversary agents on complex boards, with walls and speed bumps for example, and then to study the stability of Deep Q Learning algorithms by testing our agents on boards where we have rearranged the location of these special boxes.**

I. Introduction

This project involves the study of the adaptation of Deep Q-Learning networks in an interchangeable environment. What happens if we teach a mouse to escape from a cat on a board, and how does the cat adapt to the modification of the board, by moving walls or boxes that slow down the agents for example?

This aspect is interesting insofar as Deep Q-Learning, compared to classical Q-Learning, is supposed to react better to situations that it has never seen. Thus, we will study the reaction of two types of agents who discover new situations while they have been trained on similar but not identical situations.

The challenges of this project are first to create a modular environment from scratch with PyGame [6], on which the players will be able to play with the board as they wish, and then to succeed in creating two agents who will compete on this board, and who will be able to progress with the appropriate methods.

This project is similar to the game "Frozen-Lake" in that an agent (the cat) must move to eat the mouse. The uncertainty of the movements of the agent of "Frozen-Lake" is replaced by the uncertainty of the movement of the objective (the mouse) and it is not possible for the cat to die if it goes on a bad square. Moreover, we want to train our agents with Deep-Q Learning compared to the Q-Learning methods usually used on "Frozen-Lake" to be able to test our agents on new boards.

So we recreated from scratch a game environment on which we trained our agents. Since this game is a winning game for the mouse given the implementation features, we can expect the mouse agent to constantly beat the cat. This is indeed what we observe on simple boards, and it remains true despite a slightly lower win rate on more complex boards. We also managed to create a cat agent that optimize all its moves on simple boards as it catches the mouse in a minimum number of moves when the mouse is immobile. Concerning the stability of the Deep Q-Learning methods, we draw two conclusions: on the one hand, training on a complex terrain still leads to a victory of the cat when we test our agents on simple terrains. On the other hand, on complex terrain most of the actions performed by the agents are good, but some observations still lead to sub-optimal decisions.

If we had more time, we would have tried to model several agents. For example, start with a single cat and several mice, and as soon as a mouse is touched it becomes a cat with it. This would have allowed us to observe possible group behavior, where several cats would have helped each other to block a mouse for example. Finally, we could have pushed the study of stability by performing transfer learning on new game boards.

All our code can be found here : https://github.com/ElSacho/World_Chase_Tag_project.git

II. Background

Before working on the agents, we explain the environment we have created. First of all, we create a board, composed of elements of type 'box' which correspond to the boxes of our board. Thus, each square can be initialized with properties like 'home' for the mouse, or like a square that slows down the agents.

Then we have two agents, these two agents can move left, right, up, down, when possible. The observations of the two agents are roughly the same: each agent has a 'vision' parameter. The agent then sees the state of all the squares included in a square of center 'agent's position' and length '2*vision + 1'. The value of each square is changed if ever a cat or a mouse is on it, if ever the square is a wall, a speed bump or a house for the mouse for example. You can get the state of one agent thanks to the get_state() function. The get_reward() functions then differ according to the type of agent studied. For the cat, the get_reward() function returns 100 / self.step * distance_min where distance_min represents the distance in norm 1 (the distance used on the board) between the initialization positions of the cat and the mouse if the last action of the cat allowed it to eat the mouse, -1 every x time steps, and 0 otherwise.

For the mouse, we tried several types of rewards (the method can be changed by modifying the string parameter 'method' in the function get_reward(method='the method used'). The one that worked best is the one that studies the impact of the last mouse movement on the position with the cat. If the mouse has moved away from the cat, it receives a reward of 2, and a reward of -1 otherwise.

So here is an example of the simple game board, with the two agents Fig. 1:

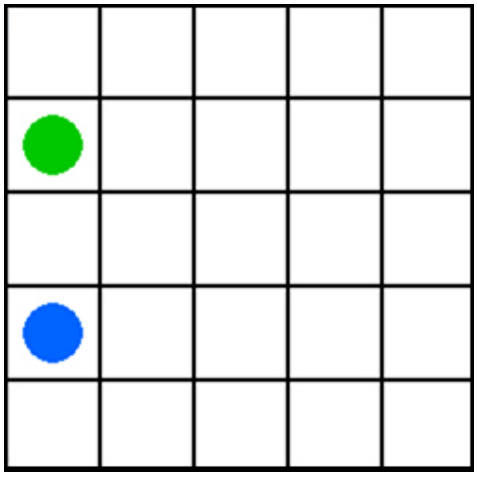

Fig. 1. 5x5 game board without special boxes, cat in green and mouse in blue

and here is an example of a more complex game board Fig. 2:

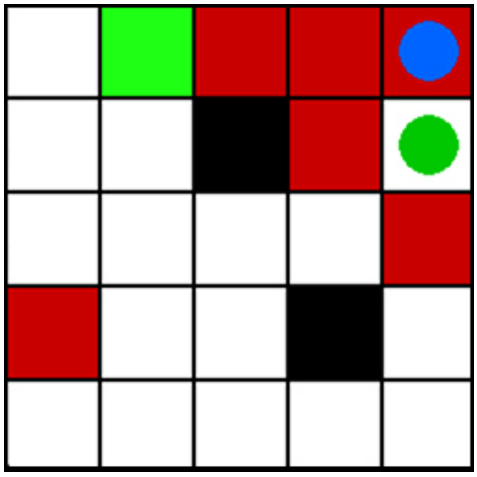

Fig. 2. 5x5 game board with special boxes, the red boxes slow down the agents, the black boxes are walls and the green boxes are houses for the mouse

Then comes the global environment of the game. We initialize the game and its parameters by creating an object of the class *gameEnv*. The mouse and the cat are then randomly placed on the board, and the parameters of the squares initialized (either randomly or manually by the user). The environment is then provided with functions for each agent: there is no 'get_state()' function, but two functions: 'get_state_mouse()' and 'get_state_cat()'; the same goes for the reset() function, or step. The environment has been coded to look as much as possible like a Gym environment, without using the Gym package. The function reset_cat() resets the position of the cat and the mouse and returns the get_state_cat(), the function mouse_step() returns 4 elements : next_state_mouse, reward, is_done, and {} where the dictionary is not used.

III. Methodology/Approach

A. Some tests to test our environment

Once the environment was coded, we implemented reinforcement learning methods on our agents. First, we trained our agents separately, on squares all initialized in the same way (all the squares of the board are classical): first, we trained the cat thanks to a cross-entropy method to go and look for the mouse which remained immobile. The training worked very well until the cat found the optimal path to fetch the mouse when the mouse was always initialized at the same location, but the training did not work as well as soon as the initial position of the mouse was randomized. So we changed our learning method. We were interested in Q-learning, but as the observation space is very large and variable, we preferred to use Deep Q-Learning [1], which will allow us to add features more easily on our get_state() functions, and to study the reactions of our agents to new situations. For the rest of the results, we set the agents' vision to $vision = 3$.

B. The implementation of Deep Q Learning

For Deep Q-Learning, we used the method described in the course [2] which uses a target neural network to evaluate the Q-Value of an action. We have studied this Reinforcement Learning method in more detail thanks to the book [5]. We opted for a neural network structure in three linear activated layers. We have adapted these methods to train our two agents simultaneously. At each step of our training loop, the cat gets its states, selects an action, then updates its neural network's weights according to the rewards of this action. Then, the mouse gets its states, performs an action and updates its neural network's weights.

C. First results

We obtain very satisfactory results. On a simple board, after studying about 200.000 actions ($\approx$ 5min on a CPU), our two agents are perfectly trained Fig. 3. Once we run our code to test the agents and display the board, the mouse always manages to escape the cat. This is not surprising: on a simple board, it is easy to keep the cat at bay if we move according to the same parameters as it.

However, we observe several things. The first is that our "mouse" agent only gets better and better at escaping the cat, while the cat's curve is increasing and then decreasing. The cat learns faster than the mouse that it must be eaten to be rewarded. But once the mouse begins to understand how to keep the cat at bay, it progresses very quickly to the point where the cat can no longer catch the mouse as easily as it did at first. Since there are winning strategies for the mouse, once the mouse starts using them, the cat doesn't progress.

D. Adaptation of some hyperparameters

To obtain such results, we essentially worked on the optimization of the $epsilon$-Greedy search. To do so, we used a ϵ parameter, which corresponds to the probability of performing a random action at each step. We chose to start at 1, so that all the actions are randomly realized at the beginning of the

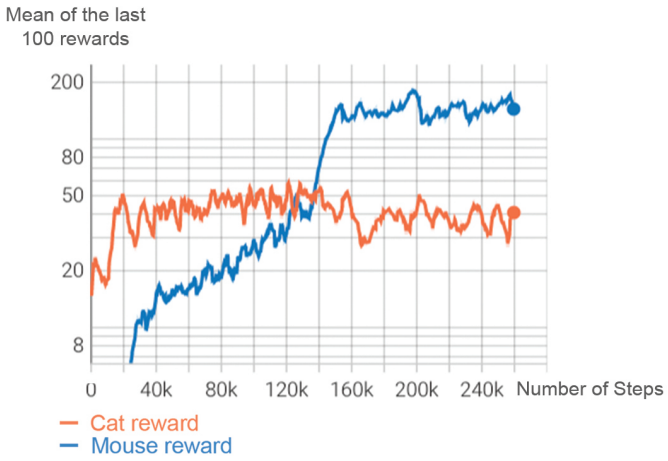

Fig. 3. Mean of the last 100 rewards for the cat and the mouse in a 5x5 game board without boxes

learning process and to mix the search for optimal actions, and to end at 0.03 to have about 3% of the actions randomly realized in steady state, following the structure of [5]. The parameter that had the most impact was the number of steps needed to go from a $\epsilon = 1$ to a $\epsilon = 0.03$. We started with 50,000 steps, but the results were not satisfactory enough as the agents were too quickly stuck in local maximum states without finding really optimal strategies. For a number of steps equal to 300.000, the learning was very long with many small cycles without improvements until the value of ϵ was lower. We agreed that the optimal number of steps was around 150.000 steps.

E. The choice of a reward for the mouse

Let's discuss a little about the choice of the reward used for the mouse. The method used for the reward of the cat is modifiable by changing the parameter "method" of the function "get_reward" in the file "mouseState.py". As a reminder, the method finally used is the one which rewards positively the mouse when it succeeds in moving away the cat, and negatively otherwise. The advantages of this method are that the mouse understands very quickly that it must move away from the cat to win. However, this reward has several disadvantages. First of all, the mouse regularly finds itself in situations where each of its actions will be rewarded negatively: when the mouse is in a corner for example. At that moment, it is very difficult for the mouse to choose an optimal action. Also, to get as far away from the cat as possible, the mouse tends to escape into the corners. With this method, the mouse moves away from the cat rather than avoiding it. Finally, with this method, the addition of certain squares such as the "house" squares for the mouse will not be used properly. If the mouse stays on a house square it will not be eaten, but the cat will be able to get very close to the mouse, so the mouse will not learn to use the house.

However, despite these disadvantages, this method works. The mouse manages to escape the cat. The problem in the corners is mitigated by the long-term reward, which encourages negative reward actions if the final reward is higher. The importance of the final reward is given by the γ parameter which we set to 0.99 to favor long term rewards. Indeed, we did not want to set this parameter to 1, taking into account the infinite loops in which the agents can find themselves, but we still wanted to privilege the long-term reward.

We have also tried other reward methods. The most interesting one seems to be the method that rewards the mouse with 1 at each step, as soon as it has survived a minimal number of times. This minimum time is set by the initial positions of the mouse and the cat, more precisely, as soon as the cat has moved more times than the minimum number of movements needed for the cat to catch the mouse under the initial conditions, the mouse is rewarded. The advantage of delaying the reward is that the cat and the mouse are not abnormally rewarded in situations where the cat and the mouse would have very different initial positions. However, this method does not work directly: when the cat does not yet know how to play, the mouse is rewarded a lot, not because it avoids the cat well, but because the cat cannot catch it. To solve this problem, we coded in the file "dqn_with_simple-_rewards.py" the fact that the mouse starts to progress only when the cat has finished its initial learning. The results obtained are however very unsatisfactory: at the end of its learning, the cat catches the mouse by constantly making the shortest path, but when we start training the mouse, this one does not manage to delay its death of more than 10 movements Fig. 4.

Fig. 4. Mean of the last 100 rewards for the cat and the mouse in a 5x5 game board without boxes, with the simple reward described above

IV. Results and Discussion

We first present our results in the case of a simple game on a 5x5 size board to be able to compare its results to board perturbations.

In this particular case, we observe that the mouse wins almost all the time Fig. 5. The cat's reward (which is the most easily analyzed) shows that the cat either never catches the mouse, or more than $25 * distance\_initiale\_cat\_mouse$. Thus, the cat, even if we can consider that its learning is working (Fig. 4), is not up to the level of a trained mouse.

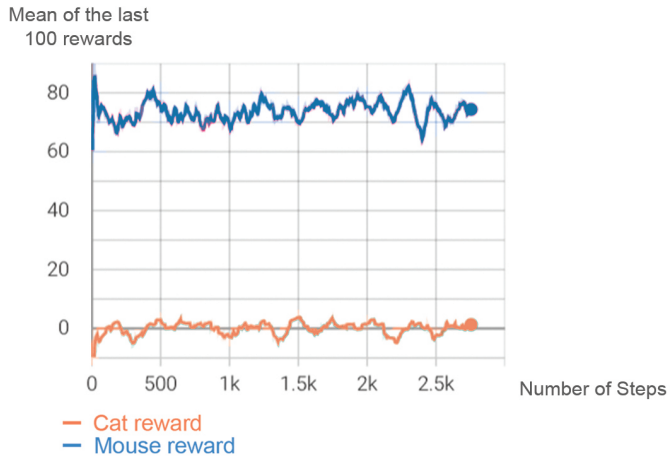

Fig. 5. Mean of the last 100 rewards for the cat and the mouse in a 5x5 game board without boxes, after the training

Now, let's look at the results obtained on more complex trays. After having trained our agents on a 5x5 board identical to the one presented above, we obtain very different results concerning the performances of the different agents. Indeed, we observe empirically that the mouse, being slowed down by the red squares, takes more time to leave them, which favors the cat which thus wins more often than it. Compared to a simple board, the cat performs only twice the number of minimal actions required by the initial conditions to catch the mouse.

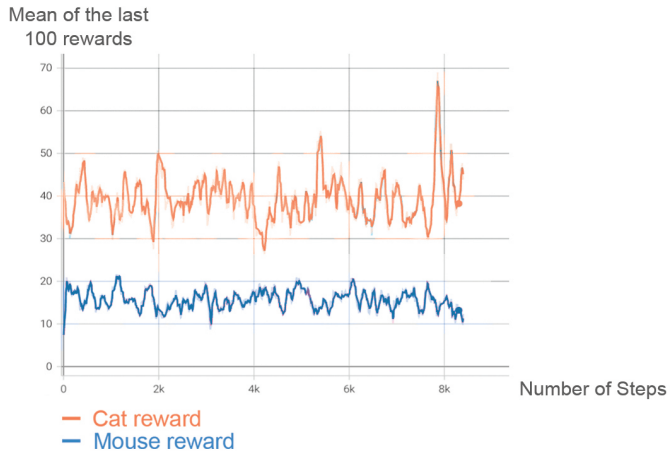

Fig. 6. Mean of the last 100 rewards for the cat and the mouse in a 5x5 game board with all types of boxes, after the training on this game board

Finally, let's observe how our agents react to changes in the plateau, and respond to situations they have never observed before Fig. 7. In this case, we observe a much higher variance of rewards. The disruption of the set leads to poor decision making, both by the cat, but also by the mouse more frequently. However, the use of Deep Q-Learning allows us to stabilize the rewards expectation of both agents at the same level as on the tray on which the agents were trained. The implications of these results show that Deep Q-Learning allows an adaptation of the agents' reactions to completely new situations that remain on average faithful to the actions that would have been taken by the agent if it had been trained to react to this type of situation.

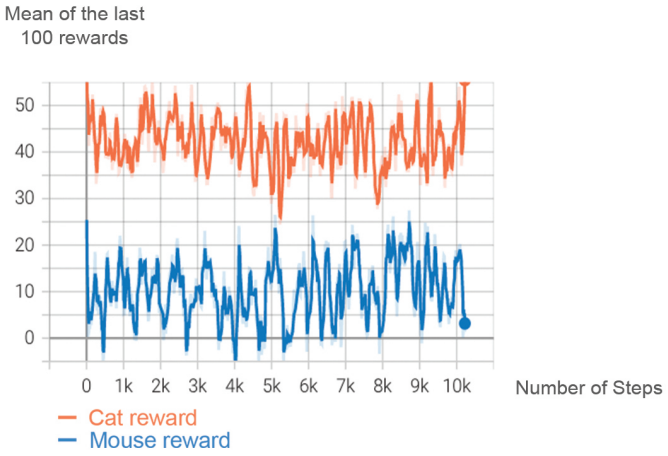

Fig. 7. Mean of the last 100 rewards for the cat and the mouse in a 5x5 game board with all types of boxes, after the training on an other game board

What happens if we play these agents on the classic 5x5 size board, without adding any special squares? We observe that the cat has much more difficulty in catching the mouse than before Fig. 8. The mouse manages to increase its reward considerably. This is due to the fact that the speed bumps also slow down the mouse's reward accumulation. Empirically, we observe that compared to the use of the agents on another board similar to the one in the training Fig.7, where the mouse was always caught by the cat because of the slowing squares, the mouse manages to escape from the cat most of the time (approximately 55% of the time), but that some cases related to the randomness of the initial positions lead to the cat catching the mouse directly. The implications of these results are multiple. On the one hand, this result shows that our "mouse" agent, despite its defeat on complex game boards, has still learned to play. This result also demonstrates the impact of the speed bumps on the cat's performance, and shows that these bumps alone are the cause of the cat's performance loss. Finally, we can conclude that the use of Deep Q-Learning allowed our agents to adapt more easily to new situations.

Finally, we wish to demonstrate the potential uses of special boxes to study the behavior of our agents. For this purpose, we create a training game board with walls and a hole in the center of it, Fig. 9. Training on this board is similar to training on a simple board. This means that the cat is able to find circuits that allow it to avoid the cat as much as possible. However,

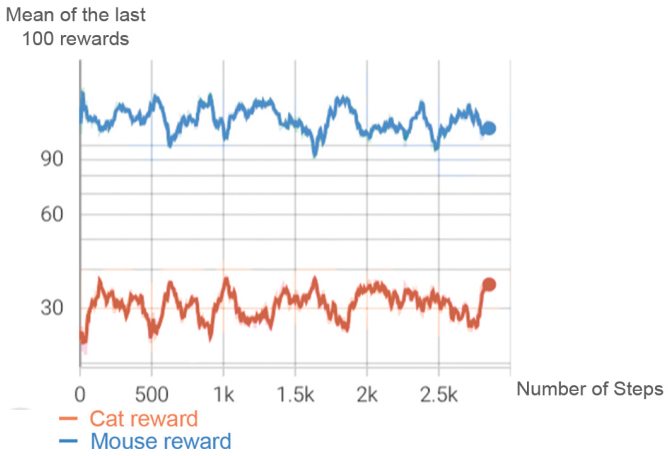

Fig. 8. Mean of the last 100 rewards for the cat and the mouse in a 5x5 game board without special boxes, after the training on an 5x5 game board with all special boxes

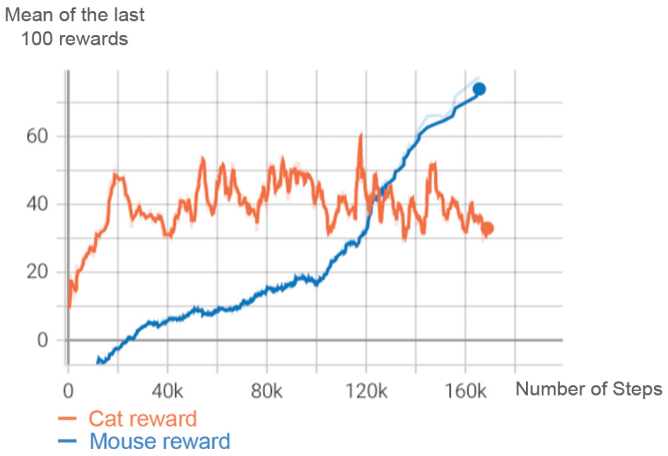

Fig. 10. Mean of the last 100 rewards for the cat and the mouse in the game board in Fig. 9, during the training

in this configuration, the cat and the mouse are regularly less than a square apart, which limits the mouse's reward. In this configuration, we are at about 47% of the mouse's victory after training, which is relatively balanced.

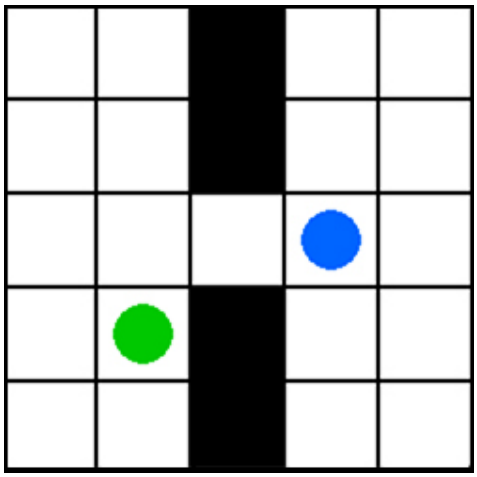

Fig. 9. 5x5 game board with walls

Finally, we wish to analyze the reactions of our agents to an increase in the size of the board. So we keep the idea of a wall with a hole, but on a 7x7 board according to which the agents have a field of view of 4 squares, but then the cat gains 76 times, because as soon as the cat's initial positions do not allow it to see where the mouse is, the cat and the mouse enter an infinite loop of series of movements. We could add parameters, such as the difference in positions between the cat and the mouse to alleviate this problem, but we preferred to keep our environment as it is, to favor and simplify the addition of multiple cats and multiple mice in a future improvement of our environment.

V. Conclusions

Thus, this study allowed us to create a new environment to be able to face several neural networks. We focused our study on the use of Deep Q-Learning to solve some of the problems of this environment, and then to demonstrate that Deep Q-Learning can anticipate actions in situations never encountered by an agent. We could also use this environment to face several Reinforcement Learning methods in order to compare them, or to observe the behavioral reactions of some agents in specific situations. More broadly, it is possible to use this environment to add agents, and observe crowd behavior when a hostile agent arrives in a room for example.

References

[1] Sutton and Barto. Reinforcement Learning, *MIT Press*, 2020.
[2] Read. Lecture VI - Reinforcement Learning III. In *INF581 Advanced Machine Learning and Autonomous Agents*, 2022.
[3] Gupta, Anmol and Roy, Partha Pratim and Dutt, Varun - *Evaluation of Instance-Based Learning and Q-Learning Algorithms in Dynamic Environments* - 2021
[4] M. Lapan. Packt Publishing - *Deep Reinforcement Learning Hands-On* - Chapter 4 - 2018
[5] M. Lapan. Packt Publishing - *Deep Reinforcement Learning Hands-On* - Chapter 6 - 2018
[6] How To Create Your Own Reinforcement Learning Environments, Machine Learning with Phil, https://www.youtube.com/watch?v=w1jd0Dpbc2o