Crazy Taxi - Reinforcement learning

Axel de Lavaissiere de Lavergne, Mohamed Salah Zaiem, Vincent Mallet

***Abstract*—The aim of our project is to implement an agent and an environment that reproduce the game Crazy Taxi, a free game that was released on Facebook. A description and a link to play this game is available here : Playing Crazy Taxi. To do so, we create our environment and agent from scratch, and then use different reinforcement algorithms to improve our agent. In this report, we present the implementation of our model, the performance of our agent.**

I. Introduction

Reinforcement learning is widely used to automatize the performance on different games, and achieve results better than humans : chess, go, Snake, Pacman... In our project, we implemented reinforcement learning on the **Crazy Taxi game**, as it is a simple game that allows to fully understand how reinforcement learning can work on gaming tasks. Our work is based on an agent and an environment we created.

Our code is available on the follow link : https://www.dropbox.com/sh/t303d7ld3o7d6tl/AADUhar3E4fQWAUp_-hJ67w0a?dl=0. It is coded in **Python 3.6**, and is in a **Jupyter Notebook** file. This code only uses basic libraries, as we created our environment and agent from scratch.

We also use the library tkinter that is natively installed with Python, in order to visualize the moves of the taxi during the episodes. We provide two different Jupyter Notebooks in our code : without and with the interface. If you run the code with the interface, do not close the windows that appear with tkinter, just wait until the cell in the Jupyter Notebook finishes, and launch the following cell. Finally, you have to leave the pictures for the interface in the same folder as the codes.

II. Background and Related Work

Firstly, let us explain how to play the Crazy Taxi game, on its original Facebook version. Here is a picture of the game :

The main goal of the game is to **survive the longest time possible on the road**. You control the taxi, and you have to avoid the different cars that are arriving, either by moving to another road, either by jumping above the car. The game ends as soon as you touch another car.

In our implementation of the Crazy Taxi, we only consider **the movements of the road, the taxi cannot jump**. For this reason, the car positions such as the one illustrated in the above figure are removed from the environment in our model, or our agent would automatically lose.

We created our agent and our environment, using Python and basic libraries. In the following parts, we will explain in more detail how they are implemented.

Fig. 1. **Crazy Taxi game interface**

The reinforcement learning part uses different temporal difference learning algorithms, present in [1], which are the following :

- **Q-learning algorithm**
- **Sarsa algorithm**
- **Forward/Backward eligibility trace algorithm**

All these algorithms are based on the same principle. If we suppose a system receives as input a time sequence of vectors (x_t, y_t) , $t = 0, 1, 2, ,$ where each x_t is an arbitrary signal and y_t is a real number, TD learning applies to the problem of producing at each discrete time step t , an estimate, or prediction, p_t , of the following quantity:

$$Y_t = y_{t+1} + \gamma y_{t+2} + \gamma^2 y_{t+3} + \ldots = \sum_{i}^{\infty} \gamma^{i-1} y_{t+i}$$

where γ is a discount factor, with $0 <= \gamma < 1$. Each estimate is a prediction because it involves future values of y . The signal x_t is the information available to the system at time t to enable it to make the prediction p_t.

Concerning eligibility traces, TD learning can often be accelerated by the addition of these. When a TD algorithm receives input (y_{t+1}, x_{t+1}) , it updates the prediction only for the immediately preceding signal x_t . That is, it modifies only the immediately preceding prediction. But since y_{t+1} provides useful information for learning earlier predictions as well, we can extend TD learning so it updates a collection of many earlier predictions at each step. Eligibility traces do this by providing a short-term memory of many previous input signals so that each new observation can update the parameters related to these signals. Eligibility traces are usually implemented by an exponentially-decaying memory trace, with decay parameter λ. More information can be found on [2].

III. The Environment

The environment we created is based on the previous picture of Crazy Taxi. We have **three tracks on the road, and we consider that the taxi can see up to three rows ahead** to avoid the future cars. The above picture shows our environment, that we visualize with the library tkinter.

Fig. 2. **Environment using tkinter library**

We also add a reward in our environment that is not present in the original game of Crazy Taxi : **coins that appear and increase the score** of the taxi if it takes it.

Based on the figure above, **each state is a 7 digits array**. It stores if there is a car on each of the four rows and its position on the row, the position of the taxi on the first row, and if there is a coin and its position vertically/horizontally. Thus, the first four digits of the array store the positions of the furthest car to the car on the same row as the taxi, the fifth digit stores the position of the taxi on its row, the sixth the position of the coin vertically, the last one the position of the coin horizontally. In total, this represents **7680 possible states**, after removing the impossible states.

Concerning the values of the digits, the values for the position of the cars are :

- **0 : car on the left , 1 : car on the middle, 2 : car on the right**
- **3 : there is no car on the row**

The values for the position of the taxi are **0 : taxi on the left , 1 : taxi on the middle, 2 : taxi on the right.**
The values for the position of the coin are :

- **vertically (6th digit in the array) : going from 0 (furthest row) to 3 (row of the taxi)**
- **horizontally (7th digit in the array) : 0 : coin on the left , 1 : coin on the middle, 2 : coin on the right**
- if vertically and horizontally values are **both 4, there is no coin** present on the map

For instance the above picture would be represented by the state [2,0,0,1,0,4,4].

In the game, we chose to make the cars and the coins appear with probability 0.5 at each new state. The probabilities of their apparition can be changed in our model.

The challenging part of this environment is that we coded it from scratch and we had to define everything in order to have working reinforcement algorithms. For instance, we firstly created our environment with three $4 * 3$ matrix (one for the taxi, one for the cars and one for the coin), where each matrix is a representation of the 2D dimensions. The problem with this representation was that the reinforcement learning algorithms took way too much time. That is why we then decided to use the representation described above for our environment. Moreover, we also created the basic GUI in order to interactively visualize our environment, using tkinter. And finally, we had to remove all the states where the taxi is bound to die because of car positions, as our taxi cannot jump contrary to the real game Crazy Taxi.

The interesting part of this example is that it provides a concise and simple example of how a reinforcement learning IA can work on a game to achieve better performance. It also can be extended with different parameters : increasing the apparition of cars or coins, changing the number of tracks on the road, changing ”how far” the taxi can see above...

In the following part, we will describe how our agent (the taxi) is configured.

IV. The Agent

The **agent evolving in our environment is the taxi**. Its possible states are (0,1,2) corresponding to the left, middle and right position of the first row as described in the previous section, and is represented by the fifth digit of our seven digits array ”states”.
Based on the state it observes, our agent can then take three actions :

- **action ”l” equals to 0 : the taxi takes the action to go left**
- **action ”k” equals to 1 : the taxi takes the action to keep going on the same track**
- **action ”r” equals to 2 : the taxi takes the action to go right**

Before taking an action, based on the current state of the agent, we define the possible actions it can take in a dictionary with a key value of ”0” for the possible actions :

- taxi in state 0 (left) : the possible actions dictionary is ”1”:0, ”2”:0, meaning that actions ”k” and ”r” are allowed
- taxi in state 1 (middle) : the possible actions dictionary is ”0”:0, ”1”:0, ”2”:0, meaning that actions ”l”, ”k” and ”r” are allowed
- taxi in state 2 (right) : the possible actions dictionary is ”0”:0, ”1”:0, meaning that actions ”l” and ”k” are allowed

Finally, our agent takes its actions based on different rewards. At each new state, if **the taxi didn't bump into a car, it receives a reward of** $+1$, and as soon as it bumps into a car it gets a reward of -3 and the episode ends. And at **each coin collected, the agent gets a reward of** $+10$. We will show in the following part how our agent performs with different algorithms, and what type of actions it takes based on these rewards.

V. Results and Discussion

We tried three different learning algorithm. If the first two algorithms : Sarsa and Q-learning produce similar results, the third one, the customized **Q-learning (forward/backward eligibility trace) leads us to a far better policy**. To judge our algorithms, we will rely on three indexes : The **mean time of survival** in each episode, the **number of coins caught**, and the **proportion of coins caught**. We test the three algorithms with 20000 episodes (the plots below will show that the algorithms converge after this number of episodes.)
The results obtained with the customized Q-learning are very promising. The taxi never crashes and gets more than half of the coins. Since some coins can represent a trap leading to a crash, we can say that it gets a good majority of interesting coins.

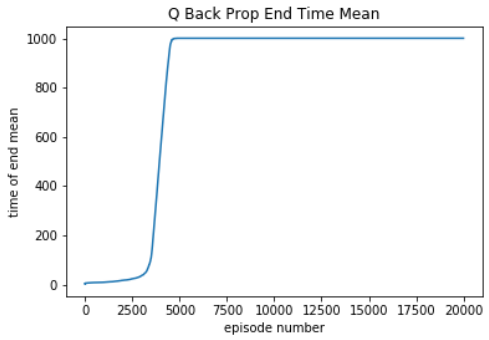

Fig. 3. **Q-Back Prop : End time mean**

The convergence of this algorithm is very interesting. Since the reward of getting a coin is a way higher than the survival one, it starts by increasing very rapidly the proportion of caught coins. This is the first phase. In this phase, the ending time increases slowly, because the attention is taken by the coins (figure 5) . The two phases are clearly visible in the Figure below (Figure 3). After the first phase, the ending mean time increases very quickly to reach its maximum (1001) after around 5000 episodes.
The performance of this learning algorithm is even more surprising when we see the other algorithms' performances. Sarsa and Q-learning never go above 100 as a mean ending time. And they do not reach 0.40 as a proportion of coins caught. (Figure 4)

A. Performance of our Agent in our Environment

In the table below we will present the results reached after convergence. 1000 is the maximal number of iterations. We assume that if the taxi reaches 1000 steps without crashing, then it will continue.

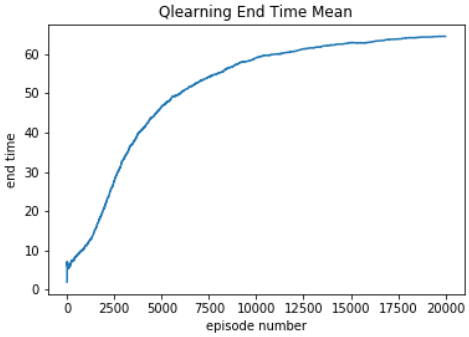

Fig. 4. **Q Learning : End time mean**

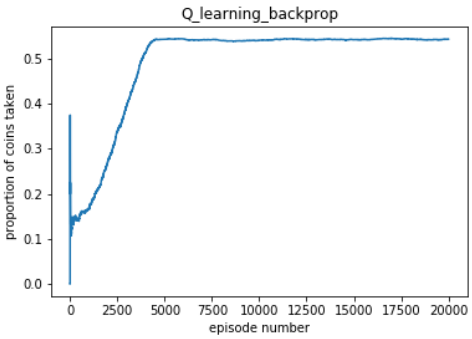

Fig. 5. **Q-Back Prop : proportion of coins caught**

TABLE I
Mean results table

Index	SARSA	Q Learning	Q Backprop
Mean Survival Time	80	70	1000
Coins caught	6	5.5	108
Proportion of coins caught	0.38	0.35	0.54

When we look at the episodes at the GUI, we quickly observe that our agent misses some easy coins. This is the main weakness of our Backprop algorithm. If the crash leads to a quick readjustment of the previous acts, we will next present the development of a backpropagation if we miss a coin, with a special negative reward for missing a coin.

B. Performance of our Agent with other settings

We tried different other settings to better understand how our agent takes its actions. First of all, as we mentioned above, we firstly added a negative reward (with different values) when our agent misses a coin, to see if it has an influence on the number of coins it takes and its survival time. Thus, we kept a reward of -3 when it bumps into a car, $+10$ when it takes a coin, and two cases when it misses a coin : negative rewards of -2, -5 and -10. Here we only focus on the results with the backpropagation algorithm as it yields the better results.

First of all, with a reward of -2, we can interestingly see that our agent, as expected, tries to get more coins (around

0.70 - Figure 6) and also solves the environment after around 5000 episodes. Therefore, a negative reward on our agent if it misses a coin forces it to focus more on catching the coins to get a better score, but does not affect its survival time. We have the same results with a negative reward of -5.

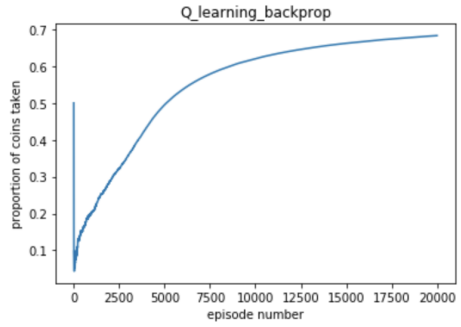

Fig. 6. **Q-Back Prop with negative reward -2 : proportion of coins caught**

But if we put a negative reward of -10, meaning that we greatly penalize our agent if it misses a coin, we can see that it affects its survival time compared to the baseline model (around 500 time steps - Figure 7). **But it will try to focus more on catching coins than surviving the longest time** (around 0.70 again - Figure 8).

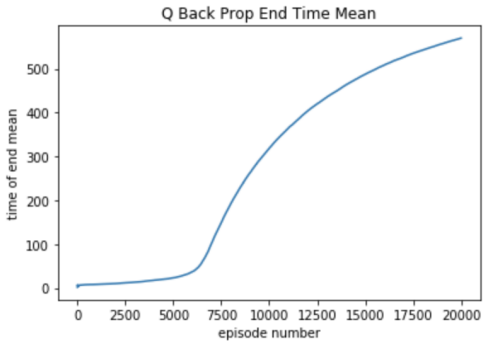

Fig. 7. **Q-Back Prop with negative reward -10 : End time mean**

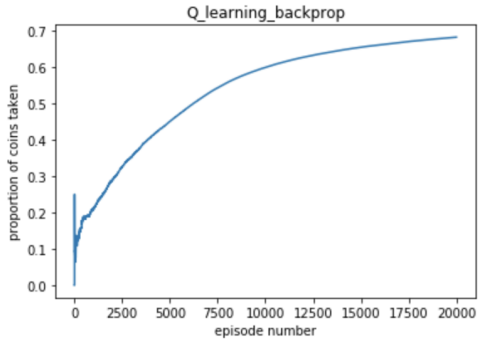

Fig. 8. **Q-Back Prop with negative reward -10 : proportion of coins caught**

But it we increase the negative reward to -10 instead of -3 when the taxi bumps into a car, meaning that the agent will try to survive for a longer time, and keep a -10 reward when it misses a coin, we notice that **our agent survives for 1000 time steps, and also retrieves around 0.80 coins** after only 500 episodes (Figure 9). Our agent thus learns that the best strategy is to survive the longest, and catch as many coins as possible to increase its score. **These different variants on the settings concerning the rewards for the coins and the bumps show that our agent effectively changes its strategy, based on the best advantage it gets from taking the coins or surviving the longest time.**

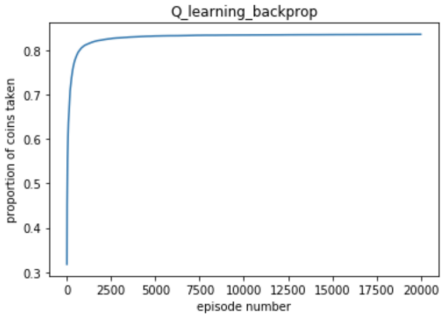

Fig. 9. **Q-Back Prop with -10 reward on bumps and coins missed : proportion of coins caught**

C. Performance of our Agent in the ALife Environment

We mostly focused our work on the creation of our own environment and agent, and the use of reinforcement algorithms on it. Therefore, we could not adapt it to the ALife Environment, as our environment and agent are working in a discrete way.

VI. Conclusion and Possible Improvements

In conclusion, we succeeded to implement an environment and an agent from scratch to play Crazy Taxi. Its strengths are that we created everything and that it performs very well with the forward/backward eligibility trace algorithm : we can consider that the environment is solved. Nevertheless, we can still see that it sometimes miss a coin that was not dangerous to catch for its later survival, so it can still be improved. The GUI interface could also be improved.

This project could be extended as follow : improving the GUI interface, or modifying our environment to create a more complex game.

References

[1] Nikolaos Tziortziotis. Lectures IV and V - Introduction to Reinforcement learning. *INF581 Advanced Topics in Artificial Intelligence*, 2018.

[2] Sutton, R. S., Tanner, B.,*Temporal-Difference Networks. Advances in Neural Information Processing Systems* 17, pp. 1377-1384, 2005