CentraleSupélec

Connect4 with Reinforcement Learning

Clemot Lucie

Nicolas Perruchot

Vicaire Thomas

Blervacque Tanguy

Paris-Saclay,

Avril 2023

Contents

1 Introduction

Reinforcement learning is a subfield of machine learning that involves training an agent to make decisions in an environment based on feedback in the form of rewards. Connect Four is a popular two-player board game that has been used as a benchmark problem in the field of artificial intelligence. In this project, we will explore how to train an agent to play Connect Four using reinforcement learning techniques, specifically Q learning and deep Q learning.

We will be using the *pettingzoo.classic.connect_four_v3 module*, which provides a Python interface to the Connect Four environment. Q learning is a classic reinforcement learning algorithm that learns an optimal policy by iteratively updating a table of state-action values. Deep Q learning is a more advanced variant of Q learning that uses a neural network to approximate the state-action values, allowing for more complex and accurate function approximation.

By implementing and comparing these two reinforcement learning algorithms, we aim to demonstrate their effectiveness in learning to play Connect Four and gain insights into the strengths and weaknesses of each approach.

2 General structure of the project

As mentioned above, we decided to focus on implementing two Qlearning techniques in order to compare them. Our general folder is divided into three types of files.

First we have *src* (for source), in which all the source code is stored and where you can find 4 scripts. In *agent.py* we define the agent, which is very simple here as he is only defined by the action that he takes in any given situation (a number between 0 and 6 corresponding to the column where he wants to play his move). In fact, we do not directly train this agent, but rather train the agent defined implicitly in the *pettingzoo.classic.connect_four_v3 module* environment, and use the **Player** class in *agent.py* to fetch this trained agent in a notebook if needed. Then, our main structure for the Qlearning model is defined in *base_model.py*. To explain briefly, everything that is common to both Qlearning and deepQleanring models is defined in the class **BaseQLearningModel** (playing a game, collecting training stats, ...). We are then left with training, updating policy and fetching an action which are different for both models. For this reason, those functionalities are implemented separately in *qlearning.py* and *deepqlearning.py*. The idea is that in order to train say the deepQlearning model, you actually call the **DeepQlearning** class from *deepqlearning.py*, which inherits all the methods and attributes from **BaseQLearningModel** in *base_model.py*.

The second subfolder in our project is *tools*. In this folder you can find all our unit tests in *unit_test.py*, the script to launch in order to train a model and save it (*train.py*) as well as a script called *win_checks.py* which implements all the functions that enabled us to understand automatically the characteristics of any

given game situation (state of the world). To give you an example, the function **is_direct_win(state)** tells you if your agent can win with his next move. The aim of theses functions was essentially to evaluate the player using statistics, but I will talk about this later.

Finally, in the folder *notebooks* you can find notebooks to train and test our models, and generally understand how they work.

3 My work

The entire group was pretty present on the project, so we all coded bits of everything. Nevertheless, I did spend more time on 2 particular tasks, which I will describe here.

3.1 BaseQLearningModel

First of all, while Nicolas focused on how to implement the training of the deepQleanring model, and Lucie and Thomas did the same with the Qlearning model, I spent most of the start, of the projects' time on implementing the **BaseQLearningModel** class (also with the help of Nicolas). The idea was to implement a base class with all the features common to both Q and DeepQ learning methods. In particular, they had some common parameters such as *initial_exploration_factor*, *final_exploration_factor*, *discount_factor* or *learning_rate*. Of course, theses parameters can also be model specific if you change them when calling your model. This class is also where the environment and the agents are initialized, and where training statistics are collected across training, but I will come back to this after. An important thing to note is that we implemented an exponential decay of the *exploration_factor* thanks to the function *get_exploration_factor* which modifies the *exploration_factor* at each epoch of training. This enabled us to better control the convergence of our training.

3.2 Training stats

Speaking of training, my biggest contribution to the project was related to better controlling and understanding how our agent performed during training. For this, I implemented the *self.stats* object which collects data across training in order to be able to plot the training stats afterwards. For this, I implemented 7 stats which we found interesting.

3.2.1 Win percentage (***Fig 1***)

Win percentage is the percentage of games won by our agent (agent 0). This is smoothed across 100 training games in order to make the figure more readable. We would expect this to go up with training, but keep in

mind that the agent is training **against itself**, so his opponent is actually getting better as well! Because of this, this statistic actually gives an idea of the "first mover's advantage" rather then the true win percentage, which should be evaluated after training against a test agent (random player for example, but other rules can be used, like playing against the agent yourself). This testing phase is going to be discussed later.

3.2.2 Number of moves needed to win (*Fig 2*)

The number of moves needed to win follows the same types of rules as the win percentage and is also smoothed across 100 epochs. That is, we would expect it to go down with time, but because the agent is playing against itself, this is not that obvious. Here, we can say that this metric is more about understanding if the agent is getting better at **defense** or **offense**. Indeed, if games are getting shorter, this means that the agent is getting better at winning (aka offense). On the other hand, if games are getting longer, this means that the agent is getting better at not losing (aka defense).

3.2.3 Direct win percentage (*Fig 3*)

A direct win situation is defined by a situation where the player can win immediately if he plays the correct move (for example, there are already three in a row, he just needs to play the fourth and it is a doable move). So the direct win percentage is the percentage of times where the player does make this "easy" offensive move. If training is going well, this metric should increase with time. In addition, because there is only one successful direct win situation per game (the game ends when this happens), there is no need to follow direct win situation quantities, the percentage holds all the valuable information.

3.2.4 Direct defense percentage (*Fig 4*)

A direct defense situation is defined by a situation where, if the player had not played his previous chip where he did, then the other player (the one playing now) would have a direct win situation presented to him. So Direct defense percentage is simply a measure of how often does the agent perform "easy" defenses. I f training goes well, we would expect this metric to go up.

3.2.5 Direct defense quantities (*Fig 5*)

Contrary to direct win, you can have many successful direct defenses in a single game, so it is important to follow this metric in terms of absolute quantities as well.

3.2.6 Exploration factor (*Fig 6*)

Finally, we plot the evolution of the exploration factor across training in order to make sure that this parameter is evolving as we would like it to.

4 Training

The training of our models was done in 2 different ways. First, we started by training small models on notebooks, looking at the results and the training stats and trying to understand how they worked, how they were different from one another and how we could make them better. This is in fact how we got most of our metric ideas.

Once we were pretty confident on the method, we decided to train larger models using *tools/train.py*. This script trains the model you choose ("q" or "deepq") and saves both the final agent and the training stats. Thee models you can test were trained on the parameters that you can see in *tools/train.py* right now, on 100000 epochs each (approximately 1h30 of training each). We then call the agents in the *test_ agents.ipynb* notebook in order to analyze the training and test them.

5 Results and testing

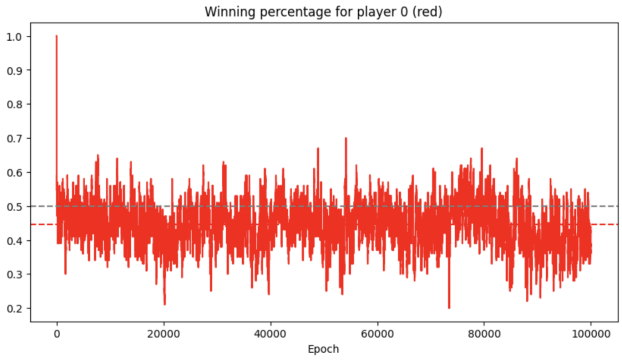
Most of the comparisons and interpretations of both models are in the notebooks, but I will recall the main results here.

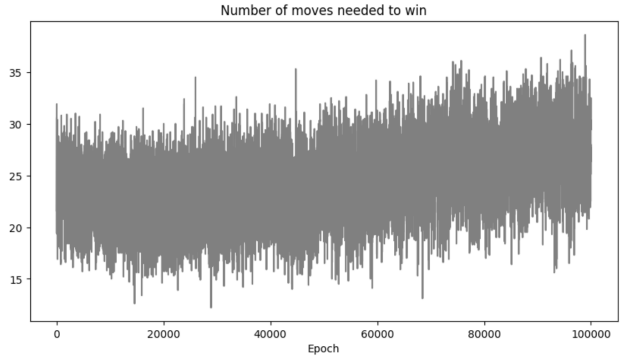
We were happy to see that both models did learn something, as all of the metrics either got better or stagnated during training. When it comes to comparing both models during their training, the most interesting thing we realized was that while the agent trained with **Qlearning got better in defense** (direct defense percentage went up and games tended to get longer and longer), the agent trained with **deepQlearning got better in offense** (direct win percentage went up and games tended to get shorter and shorter). You can also test the models yourself and play against them in *test_ agents.ipynb*.

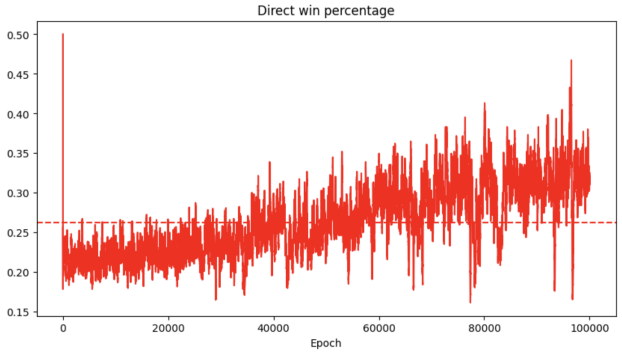
6 Conclusion

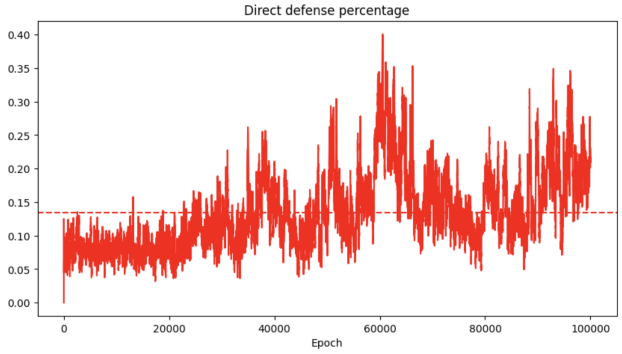
To conclude, I believe we have succeeded in building a solid framework to train, test and compare the performances of an agent trained at connect using Qlearning or DeepQlearning. In addition, the associated metrics in *self.stats* give important insights in what is actually happening during training and at the end. All in all, I was pretty motivated by this project and hope to have the opportunity to use the knowledge acquired in future projects.

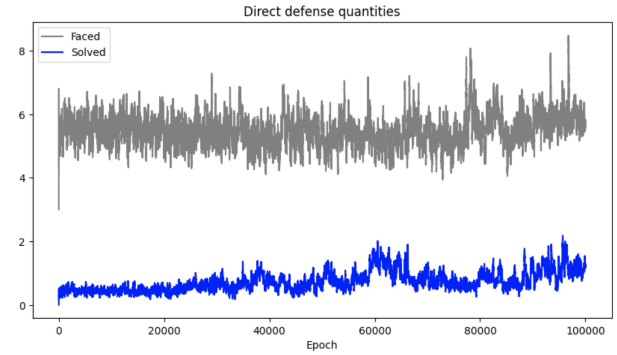
7 Annexe

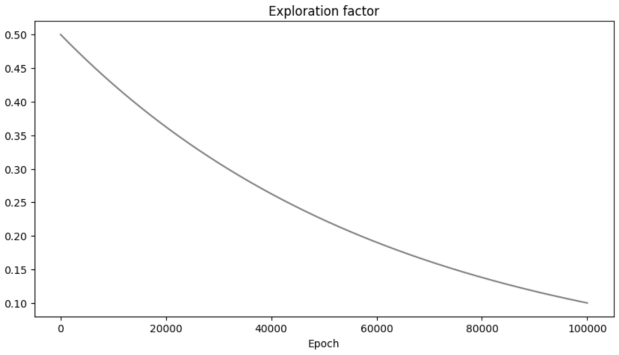

Winning percentage for player 0 (red)
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0
20000
40000
60000
80000
100000
Epoch


Number of moves needed to win
35
30
25
20
15
0
20000
40000
60000
80000
100000
Epoch


Direct win percentage
0.50
0.45
0.40
0.35
0.30
0.25
0.20
0.15
0
20000
40000
60000
80000
100000
Epoch


Direct defense percentage
0.40
0.35
0.30
0.25
0.20
0.15
0.10
0.05
0.00
0
20000
40000
60000
80000
100000
Epoch


Direct defense quantities
Faced
Solved
8
6
4
2
0
0
20000
40000
60000
80000
100000
Epoch


Exploration factor
0.50
0.45
0.40
0.35
0.30
0.25
0.20
0.15
0.10
0
20000
40000
60000
80000
100000
Epoch