Reinforcement Learning - Connect4 model

Lucie Clémot
CentraleSupélec
lucie.clemot@student-cs.fr

1 Introduction

The well-known game of Connect4 (Puissance 4 in French) is an interesting example to implement in Reinforcement Learning. Indeed, the rules are mastered by everyone, the game environment is a simple grid to model, and the game involves two players who act successively.

In the context of our project, we will use the PettingZoo library [1] and its API to visualize the game of Connect4 and the tokens placed by the players.

Our group, consisting of Tanguy Blervacque, Nicolas Péruchot, Thomas Vicaire, and myself, has chosen two more or less sophisticated learning models and worked on the quality of our learning by adding strategic aspects and quantifying the evolution of victories obtained by our agent. Moreover, we studied these elements during a game between two agents, between an agent and a random player, and between an agent and a human player. By agent, I mean an agent that has learned using one of the learning models.

2 Code structure

Our git repository[2] has the following structure :

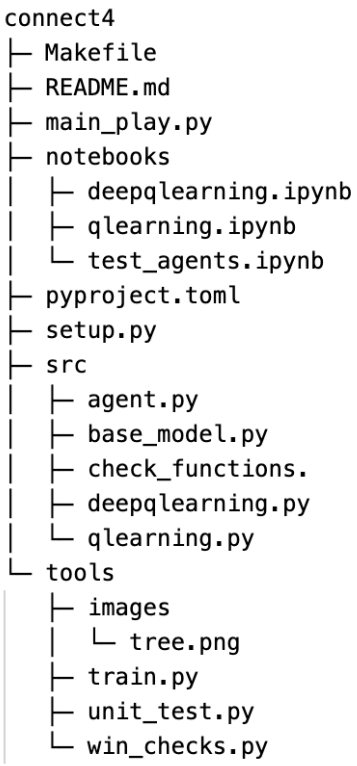

Figure 1: Repository tree structure

For the sake of this paper, we'll be focusing on the `qlearning.py` and `deepqlearning.py` file found in the src folder, the `unit_test.py` file found in the tools folder, as well as the `make_play.py` file.

Please note that all interpretations and most of the content is on the github repository and won't be discussed in this paper.

3 Personal Input

3.1 Q-Learning Model

The `qlearning.py` file and the model class in it define a Q-learning algorithm for connect4, where the agent learns to make decisions based on rewards received for different actions in different states.

The `update_policy` and `update_q_table` functions update the Q-table based on the rewards received and the state transitions, while the `get_action` function uses the Q-table to make decisions about which action to take in a given state.

We chose the following hyperparameters for the following reasons :

- `initial_exploration_factor = 0.5` because we wanted our agent to choose a random action 50% of the time initially.
- `final_exploration_factor = 0.7` because ultimately, we wanted the agent to choose a random action 10% of the time.
- `discount_factor = 0.7`
- `learning_rate = 0.1`

We overall used values we saw in literature.
Results of this training and main takeaways can be found on the github repository.

3.2 Unit tests

After a group member created the functions in `win_checks.py`, which analyze if a win is possible in different configurations (a column, row or diagonal win) as well as identify situations where a defense play is possible, and when this play is taken. It was now necessary to test these functions, in borderline cases as well as in basic cases. By running `pytest -s` from the tools folder, there are many unit tests run, and here are the more interesting ones :

- The gravity diagonal test In the configuration below, player 1 shouldn't be able to win since there are no tokens below the wining spot. Pytest confirms that in this situation, `win_diagonal` is False.

```
[[0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]
 [0 0 1 0 0 0 0]
 [0 1 2 0 0 0 0]
 [1 2 2 0 1 0 0]]
```

Figure 2: Gravity Diagonal Test

- The double defense test In the configuration below, player 1 did choose an optimal defense (column 6), but the defense wasn't successful since a defense was also needed column 2. Therefore, the `was_succesfull_direct_defense` is False.

```
[[0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]
 [1 0 0 0 0 0 0]
 [1 0 0 0 0 0 0]
 [1 0 2 2 2 1 0]]
```

Figure 3: Double Defense Test

Adding these unit tests enabled me to take the time to focus on the quality of the tools we were creating. If these tools were inexact, our training evolution would be as well.

3.3 Random play

Since we trained our agent by playing against a similar agent, we never got to test our play against a random player. I implemented this possibility as a method in `base_model.py`. It was very interesting to see how our agent interacted with a random player.

3.4 User play

We realized we never allowed the user to play against the machine. This led us to implement the `play_user` method in the `base_model.py` file.

Using the `main_play.py` file in the root folder, you have the capacity, as a user, to play against the agent using terminal commands (as below). This is accompanied by the usual PettingZoo graphic interface.

```
Choose a column to play in (between 0 and 6)
1
Choose a column to play in (between 0 and 6)
2
Choose a column to play in (between 0 and 6)
1
Choose a column to play in (between 0 and 6)
3
Sorry, you lost against our agent.
```

Figure 4: Terminal Interface - User play

4 Conclusion

This project was a great way to try and use reinforcement learning in a game setting. However, I believe that the nature of connect4 is suitable for such a task, whereas more complex games might not be. It enabled me to focus not only on the actual agent's performance, but on ways to quantify its performance and course-correct if the performance isn't satisfying.

References

[1] J Terry, Benjamin Black, Nathaniel Grammel, Mario Jayakumar, Ananth Hari, Ryan Sullivan, Luis S Santos, Clemens Dieffendahl, Caroline Horsch, Rodrigo Perez-Vicente, et al. Pettingzoo: Gym for multi-agent reinforcement learning. *Advances in Neural Information Processing Systems*, 34:15032–15043, 2021.

[2] Nicolas Péruchot Thomas Vicaire Tanguy Blervacque, Lucie Clémot. Connect4 rl project, 2023. https://github.com/NicolasPeruchot/connect4github.com.