Connect 4 PERUCHOT Nicolas

BLERVACQUE Tanguy, CLEMOT Lucie, PERUCHOT Nicolas, VICAIRE Thomas

April 25, 2023

1 Introduction

This report presents the different choices we made to develop our Connect 4 agent and will mostly focus on the parts I addressed and implemented. The code can be found on this repository:
`https://github.com/NicolasPeruchot/connect4`

2 Structure of the code

The code is divided between:

- Models
- Demonstration notebook
- Various utils functions
- Metrics
- Tests

3 Models

I worked on implementing models, specifically the Deep Q-Learning (DQL) algorithm as well as the parent class, *BaseQLearningModel*, which enabled refactoring and sharing of a number of parameters and methods used by both the Q-Learning (QL) and DQL models.

3.1 BaseQLearningModel

The *BaseQLearningModel* class takes the following parameters:

- *discount_factor*: the discount factor
- *learning_rate*: the learning rate

- *initial_exploration_factor*: the initial exploration factor
- *final_exploration_factor*: the final exploration factor

The various methods implemented and shared by the QL and DQL models are:

- *get_action*: This method returns the action to be taken from a given state. Note that this function is not fully implemented in *BaseQLearningModel*, and will therefore throw an error if it is not implemented in child classes (similar to an interface in Java or C#).
- *update_policy*: Updates the policy. Similarly, this method must be implemented.
- *play_game*: This method starts the training procedure. It calls the two methods mentioned above, distributes rewards, and starts the games.
- *reset_agents*: Initializes the agents with a dictionary that stores their name, the last state, the current state, the current reward, and their last action.
- *get_exploration_factor*: Allows to obtain the exploration factor. During training, this factor decreases exponentially from the initial factor to the desired final value.
- *initialize_game*: Initializes a game.
- *play*: Allows to visualize a game after training.

The other methods, particularly the statistics that track the progress of the training and the agent's effectiveness, were mainly implemented by other members of the group. They allow tracking the number of wins of one of the agents.

3.2 DeepQlearning

The file *deepqlearning.py* implements the various classes and methods for implementing the learning of the strategy.

The *NeuralNetwork* class is the neural network used by the DQL model. It is a simple two-layer network.

The *DeepQlearning* class implements the complete model, including the methods defined above, as well as a *training* function for training the neural network.

It should be noted that both players learn at the same time during the training. We do not focus on only one player, except when calculating metrics. This is why we keep track of the last states and actions of each player. For example, if one agent wins, we will penalize the last action of the other agent, even if

when it played that action, it did not directly lead to the defeat.

The hyperparameters have been optimized based on various trials, except for the exploration factor, because we want the agent to explore different strategies at the beginning, and then focus on the best it found. With more time, we would have liked to use libraries such as Optuna to optimize the hyperparameters more precisely, based on the metrics we have implemented.

The advantage of DQL, compared to the simple QL model, is that the agent has an intuition for the action to take even in situations it has not encountered, which allows for better generalization compared to a simple q-table.

4 Evaluation

To evaluate our models, we implemented different metrics like:

- the win ratio
- the ability to detect moves that will lead to a direct victory: we check if the agent's next move can allow it to win the game in one move, and we measure if it actually played that move
- the ability to defend by avoiding such garanted-win situations for the opponent
- the number of moves needed to win

They are more details about our metrics in the report of my colleagues.

5 Conclusion

The other aspects of the code (unit tests, visualizations, agent interaction, metrics) were handled by other members of the group, as I was mostly focused on the models. Given more time, it would have been interesting to optimize the hyperparameters, particularly the neural network architecture used for the DQL model. With our project, we demonstrated that the QL agent was a better defender, and the DQL agent a better attacker. I believe the structure of the code and its organisation would allow to plug other models in our framework to test other approches.