NIE-MVI Computional Intelligent Methods

12th Jan 2024

Semestral Project

Theo Hernandez

academic year 2023-2024

1 Introduction

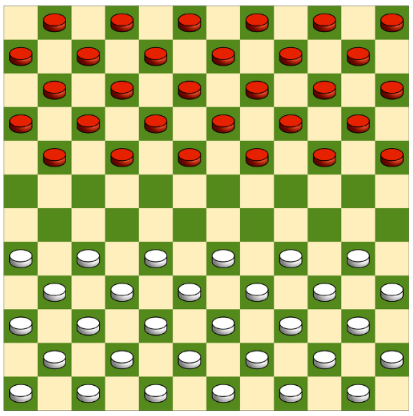
This project is made in order to compare several techniques to create a checkers bot. I used the Deep-Q learning algorithm, an algorithm of Reinforcement learning which doesn't require any knowledge to train.

The purpose of this work is to compare how a model reacts to different players in front of it during the training phase. To do so I implemented a bot which is trained with 3 different ways.
Here is the different players the model is going to play against to train itself how to play.

- Random player : plays random moves
- Minmax player : computes a minmax tree and plays the best moves based on a score computed by myself.
- Itself : (playing against itself)

2 Checkers

Many implementation of Checkers were proposed on github but it was always with simplified rules. I wasn't able to find a project with the real rules of checkers. So I decided to implement the game myself. Here is the game with the rules I implemented :
The board is a 10x10 square and the pieces are placed this way :



- The pieces are only allowed to move in diagonal and only for a distance of 1
- Pieces can eat an opponent piece buy jumping over the opponent piece and end after it. Eating can be done forward and backward
- If a piece arrive at the end of the board, the piece become a King
- A king is allowed to move forward, backward and to any distance (but it has to be in diagonal)
- Eating can be done multiple time in a row.
- The player has to eat if he can. He also has to eat as many pieces as he can.
- A player win if the opponent has no more pieces or cannot play anymore.

For more details please checked here

3 Deep-Q Learning Algorithm

I used the algorithm this way : First the player plays against himself. During the game the state of the board are copy and add to a list named *boards*. If the game ends successfully I add give a reward of 10, if the game end with a lose I give a reward of -10 and if the game ends with a draw I don't give anything.

Every x games, I trained the model again with the boards that were saved memory and the new values. It is called a "generation".

The new values are based on the rewards and here is how it is calculated :

$$\begin{aligned} ones &= 1 \text{ in case of a win and -1 in case of lose} \\ new\_value &= old\_prediction + 0.5 * (reward + 0.95 * ones - old\_prediction) \end{aligned}$$

4 Results

4.1 Random and Minmax bot

To compare the result, I create another bot but efficient based on the minmax (simplified) algorithm. To do so, I made a minmax tree of two layers and here is a quick view of the result with a random and a minmax bot :

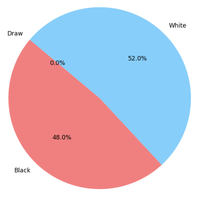
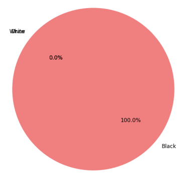
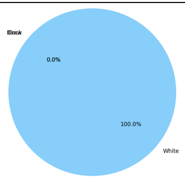
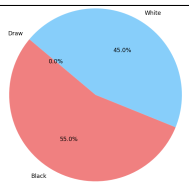
	random bot with black	**minmax bot whit black**
random bot with white		
minmax bot with white		

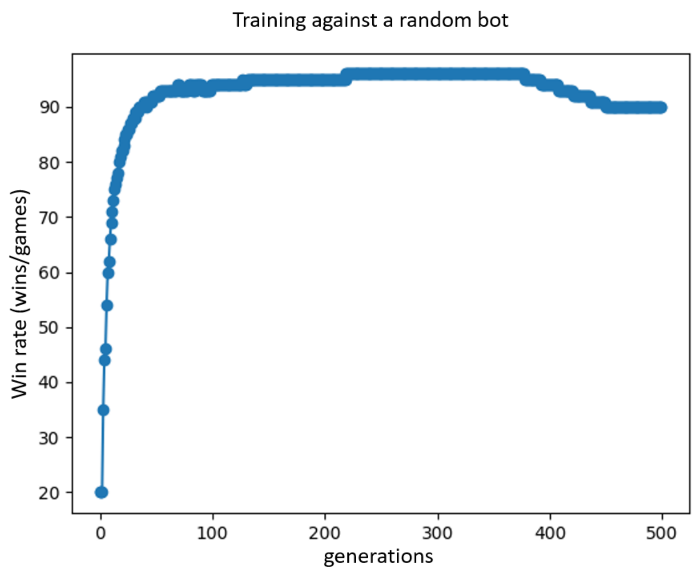
Table 1: blue when left (white) win and red when up (black) win

We can see that the minmax bot wins all the games against th random bot and the color of player doesn't change anything since it's quite equal between 2 random or 2 minmax bots. The evaluation was made with 100 games for each scenario.

4.2 Deep-Q learning bot

For every different situation, I will show a graph. It will show, generation by generation, the win rate of the Deep-Q learning bot.

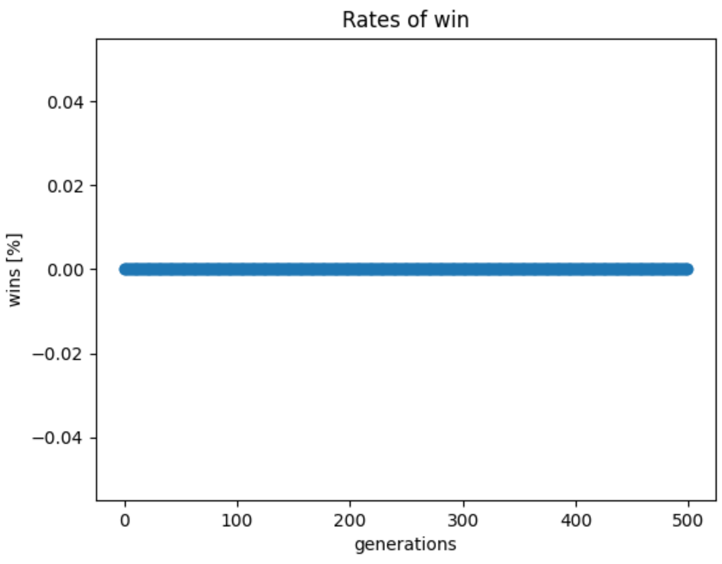
4.2.1 Trained against a random bot



We can see several things. First we can see that the model trained very well. It started as a win rates of 20% and ends over 95%.

Another thing we can notice, after the generation 390 the win rate is going down. We can see than training the model to much can be inefficient. For this example, 300 generations would be enough, only because the opponent (blacks) are not improving (always playing random moves).

4.2.2 Trained against a minmax bot



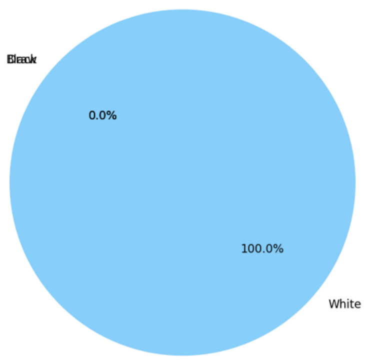
Training a model with another player which is already better than you is definitely not a good way to do it. The model is not able to train if it always lose, the model needs to win and lose to improve itself.

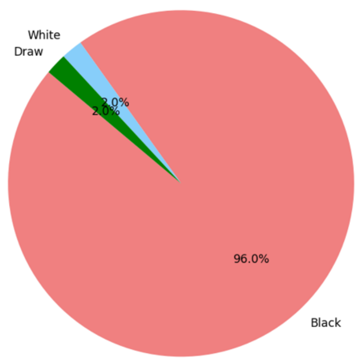
Given the graph, we are not expecting good results for the model against the other bots.

4.2.3 Trained against itself

The problem here is that both are deterministic. Then a choose to use a *exploration*. I set it to 0.95, it means that on average one move on 20 ($1 - 0.95 = 1/20$). This will able to have different games when the model are not trained.

A good question would be "Why would it be efficient to train against itself ?". As the model is always playing against itself, the winning rates should be always constant.
But no. The model were trained only with the white pieces. Here is the comparison of the first bot (trained against a random bot) and a random bot :



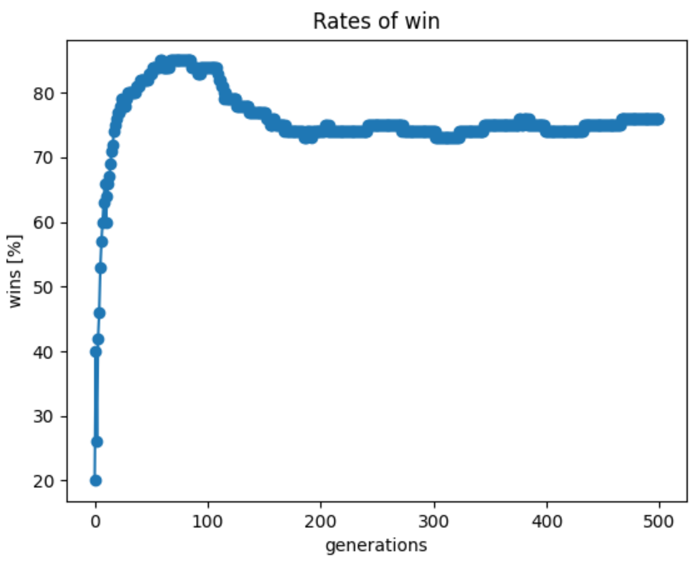


Trained bot (blue) - random bot (red) Random bot (blue) - trained bot (red)

We can see that the model is better with the white pieces. We have seen that the model is trained based on features and the features from the black or the white pieces are extracted the same way, so we should not have any difference if the model is playing with the white or black pieces. Yet we can see a difference.

I'll put forward the hypothesis that the model corrects its moves based on the boards it saw when it was its turn to play. So it never entered the model when a single move had been made for example (the whites always plays first).

Based on that, we can trained the model against itself. Only thing is that the model will train only for the white moves. Let see the result.



This is an interesting graph for many reasons. First of all the model is training very well, the shape of the graph is quite good, and close from what we expect from a good training.
Second, The white are having a win rate close to 90% and than going down. What would it be the reason ? I found two possible good reasons for that.

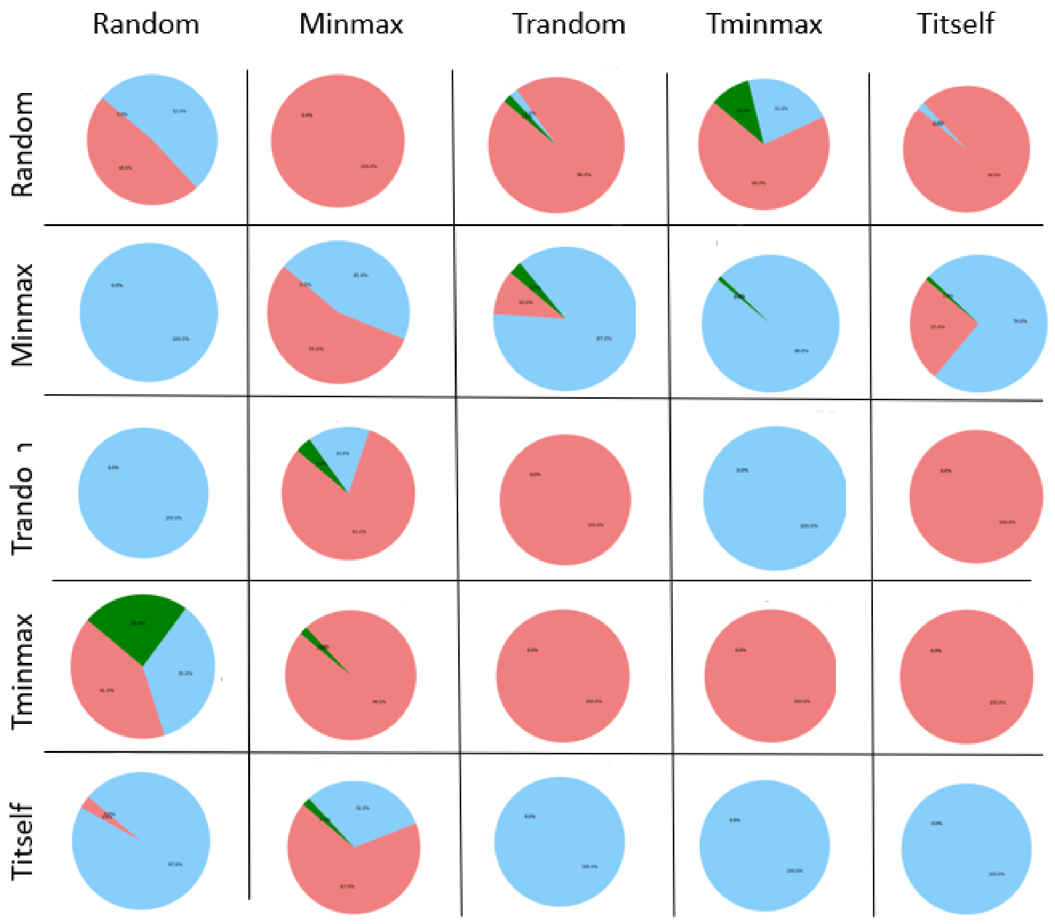
- The model is having a kind of over-fitting and the white don't play as well a it used to.
- The Black is getting better while the white stagnate.

5 Battle against all the bot

I made a little tournament between all the five bots

- Random bot (random)
- Minmax bot (Minmax)
- Trained against random bot (Trandom)
- Trained against minmax bot (Tminmax)
- Trained against itself bot (Titself)

And here is the result :



Blue when the left part wins, red when the upper part win et green for the draws.

We didn't expect any result from the bot which were trained against the minmax, and the result are actually right. This bot is losing every game (except against the random bot of course).

The bot trained against the random bot is actually winning games against the minmax bot. This means something : If a model is training with someone who plays always randomly, it is not inefficient against good player.

The bot trained against itself it very efficient. It is nearly the same level then the minmax bot and always win against other the other bots.

When the bot trained against the minmax bot plays against itself. The black always win. It means that even the bot is not good (it is not better than the random bot). The blacks learned more than the whites, even if the model when trained with the whites.

When can observe what happend when the bot are playing against itself. The random and minmax bot are quite equal between black and white as explained in a previous section.
The bot trained against itself is always winning (when he plays against itself) with the withe pieces. Which is what we expected since we train the model with the white pieces. What we didn't expected is that for the bot trained against the random and minmax bot, when the bot is playing against itself, the black always wins even though the white were more efficient against the random bot as we seen and explained in a previous section.

6 Conclusion

The conclusions in relation to the original question: "What is the most effective way of training a reinforcement learning model with the Deep Q learning algorithm ?" is the following : The best way to train the model is to train it against itself. Training the model against a random player also work but is not not the most efficient way. And training a model against a player who can already play doesn't work at all.

7 Improvement Area

Several things can be analyzed to create a more effective bot.

- The model : The number of hidden layers, the parameters, the number of neurons per layers... of the model can be optimize.
- The features we decided to analyse to train the model could also be a part of an optimization. With better features we can have a better knowledge of which player is more likely to win. This would be extremely helpful to improve the bot.
- We could try to first train a model against itself then against a better player and then back against itself and so one. We can try different kinds of combination and by comparing the results we could find another way to improve our bot.

There are of course other ways to improve the bot. If you want to improve it from my code please contact me first.