Deep Reinforcement Learning for Connect 4

Lina Mezghani & Sarah Perrin

***Abstract*—This article presents our project for the INF581 course : Advanced Topics in Artificial Intelligence. We have chosen to code an intelligent agent for the famous game Connect 4 (Puissance 4 in French). To do so, we have used Deep Reinforcement Learning techniques. We have designed a neural network with multiple layers, following the same strategy than the AlphaZero agent. Our agent gave very good results, beating random player and human players in most of the cases.**

I. Introduction

A. Access to our code

You can find our code here.

B. About Connect 4

Connect 4 is a two-player connection game in which the players first choose a color and then take turns dropping colored discs from the top into a seven-column, six-row vertically suspended grid. The pieces fall straight down, occupying the next available space within the column. The objective of the game is to be the first to form a horizontal, vertical, or diagonal line of four of one's own discs.

Connect 4 has been solved perfectly since 1988 by Victor Allis in his masters thesis "A Knowledge-based Approach of Connect-Four" [1]. He has demonstrated that it is possible to win if you begin the game, and if you play in the middle column. Thus, the challenge is to design a perfect intelligent agent, which will win if it begins. The other difficulty is to deal with the memory issue, as there exists 4,531,985,219,092 different states in Connect Four, with more than 2 trillions states of victory. Then it will not possible to create a Q matrix ($states \times actions$) to store the policy we should take in each state, and this is why we have created a neural network to estimate these values.

II. Background and Related Work

A. Minimax algorithm

There exists a lot of algorithms to create an intelligent agent to play Connect 4. The most famous one is Minimax algorithm [2], often improved with Alpha-Beta pruning to reduce the search space. Minimax algorithm explores the tree of different states that the board of the game can reach. It then chooses the best move that can be played, according to an evaluation function (for example, the number of same-colored discs in a row, column, or diagonals). The algorithm first maximizes the gain of the current player, and during the search minimizes the gain of the other player.

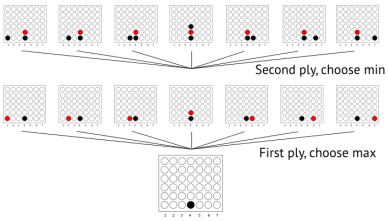

Figure 1. Principle of the Minimax algorithm

However, the number of states that can be reached at each turn is limited, because of exponential growth when the tree is visited (7, 49, 434, 2401 ...), even if it can be enhanced with clever tricks. The other problem is that the heuristic that calculates evaluation function needs to be perfect, for example by giving the final result of the game according to which move the player will make, but this information is not reachable at this step (if the minimax algorithm can not explore fully the tree).

We have implemented a simple version of minimax, for test purpose only, because Minimax algorithm is not considered as Reinforcement Learning.

B. Deep reinforcement learning model : AlphaZero

We have chosen to train our agent with a neural network inspired of the famous AlphaZero intelligent agent, which uses Deep Reinforcement Learning [3]. AlphaZero is the generic version of AlphaGo Zero, and can play every combinatorial game, including chess, go, shogi, ... and of course Connect 4. It has been developed by DeepMind a few months ago, and has for example beaten Stockfish, the best AI in chess in the world, after only 24 hours of learning. AlphaZero (and AlphaGo Zero in Go which has beaten AlphaGo), is able to do this by using a novel form of reinforcement learning, in which it becomes its own teacher. The system starts off with a neural network that knows nothing about the game. It then plays games against itself, by combining this neural network with a powerful search algorithm. As it plays, the neural network is tuned and updated to predict moves, as well as the eventual winner of the games.

This updated neural network is then recombined with the search algorithm to create a new, stronger version, and the process begins again.

This technique is more powerful than previous versions (AlphaGo for example) because it is no longer constrained by the limits of human knowledge. Instead, it is able to learn from scratch by playing against itself.

C. Specificities of AlphaZero

It only needs the board as input, whereas previous versions of AlphaGo included a small number of hand-engineered features.

It uses one neural network rather than two. Earlier versions of AlphaGo used a policy network to select the next move to play and a value network to predict the winner of the game from each position. These are combined in AlphaZero, allowing it to be trained and evaluated more efficiently.

It does not use rollouts - fast, random games used by other programs to predict which player will win from the current board position. Instead, it relies on its high quality neural networks to evaluate positions.

D. Monte-Carlo Tree Search

Monte-Carlo Tree Search [4] is an important part of our model. Given a state s, the neural network provides an estimate of the policy p, which is a probability vector over all possible actions.

During the training phase, we want to improve the predictions of the actions. This is accomplished using a Monte-Carlo Tree Search. In the search tree, each node represents a board configuration. A directed edge exists between two nodes i and j if a valid action can cause state transition from state i to j. Starting with an empty search tree, we expand the search tree one node (state) at a time. When a new node is encountered, instead of performing a rollout, the value of the new node is obtained from the neural network itself. In fact, given a state s, the neural network provides an estimate of the value v. This value is then propagated up the search path.

III. The Environment

The environment we designed is a classic environment for Connect 4.

A. Description of the environment

You can run the environment by executing the `test.py` file.

The display is really simple : R for a red chip, J for a yellow, and . for an empty slot. A human player can play against the computer. At each turn, the user is asked to enter a column (a number between 0 and 6) to put its chip in. We improved a very basic code found on the internet in order to be adaptable to different strategies, by adding a lot of classes and methods. Indeed, we wanted to test our AI against different policies : human, random and Minimax strategies. We thus created a class PLAYER that contains the information of which policy it has adopted. The class NEWGAME contains all the functions required for a game to be played. Particularly, the function getMove() describes the strategy of a player depending on its strategy.

```
0 1 2 3 4 5 6
. . . R . . .
. . . J . . .
. . J R . . .
. . R J . . .
. . J R J R .
. J R J J R .
```

Figure 2. Screen-shot of our current playing environment

Several tests file are available to simulate many parties.

IV. The Agent

As mentioned in the second part, we have chosen to design an agent inspired of AlphaZero. We could create and train our agent thanks to a well document article[5]. As the AlphaZero neural network is very complex, we have not designed ourselves the network. However, we put a lot of efforts in understanding exactly how the whole algorithm works, and the following section explains it.

A. Description of the neural network

The input of the neural network is just a state, coded as an array of length 42, 0 is for an empty spot, 1 for the player spots and -1 for the opponent spots. The neural network is composed of a convolutional layer followed by several residual layers. A residual layer is itself a combination of several convolutional layers. After the residual layers, the networks divides in two parts : the value head and the policy head, each one will output respectively the value and policy given the state input. You can find a scheme of the neural network below.

B. How the whole algorithm works

There are three phases in the learning : Self play, retrain network and evaluate network.

1) Self play: During this phase, the agent plays a lot of games against itself. At each move, the vector of search probabilities is returned with the game state and the winner. The winner defines the reward of the model.

To choose its next move, the algorithm performs a Monte-Carlo Tree Search described in the algorithm below, and updates the weights of the neural network to improve the predictions.

Here are some precisions about the algorithm. When the Monte-Carlo Tree Search begins, a new tree is built. For each state that is visited, we first initialize and then updates :

- N = number of times action
- Q = mean value of the next state.
- W = total value of next state.
- P = prior probability of selecting action a.

Q represents the exploitation part and U represents the exploration part. U is a function of P and N that increases if a node has not been explored a lot. After all

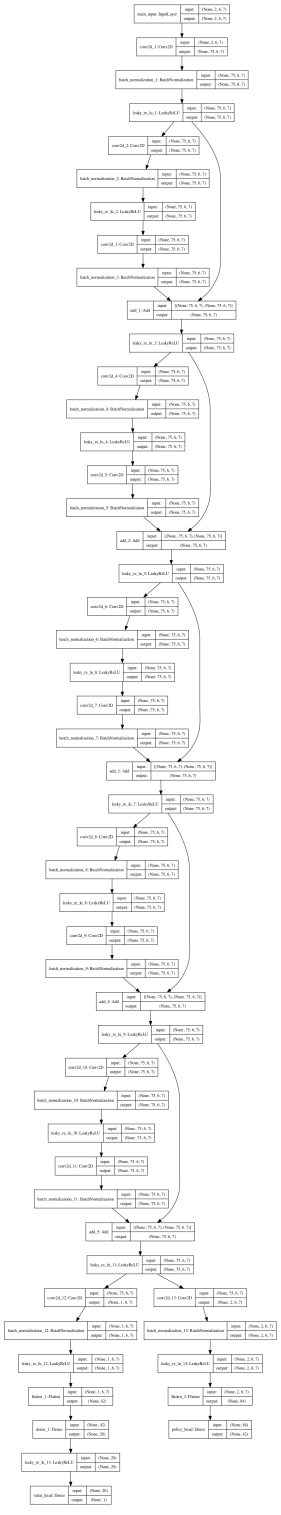

Figure 3. the Neural Network used for our best agent

Algorithm 1 Monte Carlo Tree Search Algorithm

for i in range($nb\_iterations$) **do**
  $node$ = root
  while $node$ is not a Leaf **do**
    $node$ = node corresponding to the action that maximazises $Q + U$
  end while
  Pass the state of $node$ in the NN
  compute the probabilities of moves for that state
  compute the value v for that state (for the current player)

  """Backup previous edge"""
  for all $edge$ that has been crossed to go to the leaf **do**
    $W = W + v$
    $Q = \frac{W}{N}$
  end for
end for

the iterations (nb_iterations times), the move can be selected deterministically or stochastically. In our algorithm, it is first selected stochastically with an epsilon greedy search to allow exploration, and at a point the model chooses it deterministically to test.

2) Retrain network: This step is to optimize the network weights.

It samples a mini-batch of 256 states in the previous games and retrain the network over these positions. It then compute the loss function and compares predictions from the neural network with the search probabilities and actual winner.

3) Evaluate network: Now we can test to see if the new network is stronger. We play 10 games between the latest neural network and the current neural network. If the current neural network is better than the latest, we exchange them.

V. Results and Discussion

A. Performance of our Agent in our Environment

We have thirty different versions of our agent, each one corresponds to an iteration of Self-play, retrain network and evaluate network if the current player has beaten the best player. Thus theoretically, the last version is supposed to be the best one. In that section we will try to evaluate the performance of the different versions and describe the limits of the agents.

1) Against a random player: We first tested our agent against an opponent that had a random strategy, which means that at each turn, the player choses a random (not full) column to put its chip into. Against such a naive strategy, we expected to have a winning rate close to 100%. We simulated 100 games against the random strategy for each of the 30 versions of our agent (1 being the less trained and 29 the most). The results are plotted in the graph below. We can see that for versions 10 and higher the winning is between 95% and 100%.

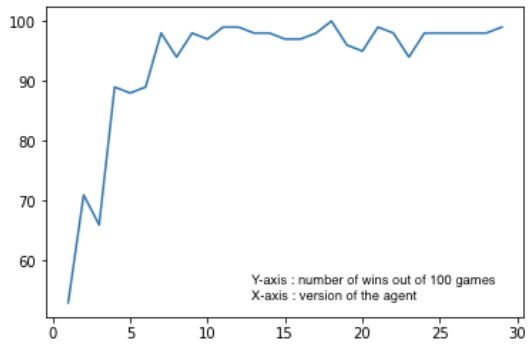

Figure 4. Percentage of win against random agent

Even though those results are quite satisfying, we would have expected a 100% win rate at one point. But we found some pathological cases that the agent was bad against (see example below). In that example, the random agent starts and put its first chip in the first row. Then our agent put its chip in the middle, and both of the agents continue to put it in their own column. After 7 turns, the random agent wins. Our interpretation for that mistake is that, since the agent is trained against himself, and that he gets better and better, he is not able anymore to face weak strategies. And starting in the first column is for sure a weak strategy because it is proven that the opponent can win in that situation. That being said, our final version of the agent did not fall into this trap.

```
0 1 2 3 4 5 6
. . . . . . .
. . . . . . .
. . . . . . J
. . . R . . J
. . . R . . J
. . . R . . J
```

Figure 5. Example of a pathological case for version 21. It was corrected by the agent in further versions.

2) Against a minimax player: We then tested our agent against a player that adopted the minimax strategy (see section II.A for a description of the minimax strategy). As for the random strategy, we tested each of the 30 versions of our agent against the minimax strategy, and plotted the result in the figure below. The results are not as satisfying as for random strategy. Even though the agent seems to get better and better against minimax, the maximum winning rate is 70%. It seems that this occurs because our agent is not trained for some states of the board. In fact, we have noticed that for theses states that the probabilities of actions (valid moves on columns) are very close. For example if only three moves are valid, they will be around 1/3 each. This suggests that our agent is not trained enough for some situations.

Thus, to improve our agent, we could try to train it with more simulations in the MCTS, and to increase the size of the memory.

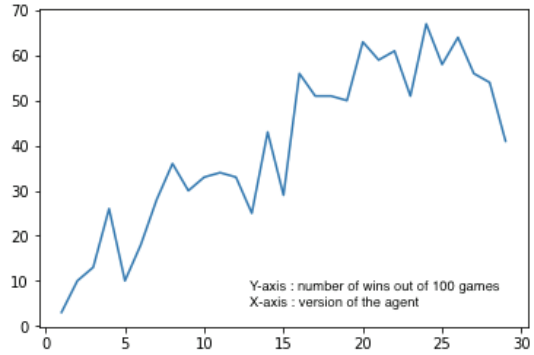

Figure 6. Percentage of win against minimax agent

3) Against a strong human player: We also played ourselves again our agent, and could not win. One of our friend, which is a very good player also tested it and could not neither win against it. If you want to try to beat it, just go in the test folder and run the `test.py` file.

4) Against a perfect player: Some simulations of perfect players are available on the internet [6]. To check if our agent has a perfect strategy when playing against a perfect player, we have simulate a match between them. It appears that our agent has a perfect strategy in the beginning (approximately until the middle of the game), because it plays all the perfect moves, but at one point, it does not find the perfect move and then loses. Once again, this suggests that there is a lack of training and that we should be able to improve it with enough training.

5) Loss of our model: The loss function was calculated with a softmax cross entropy with logits, which measures the probability error in discrete classification tasks. The figure that plots the loss of our model underlines that our agent is improving during the process. However, we noticed that is was stagnating at one point (around 800 iterations), and the last agents did not seem to be better ones. This is why we could try another strategy and change the MCTS parameters and the memory size to allow the MCTS to explore more states at each iteration.

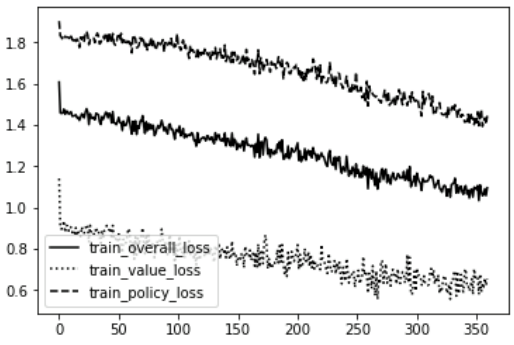

Figure 7. Loss of the model in the beginning of the process

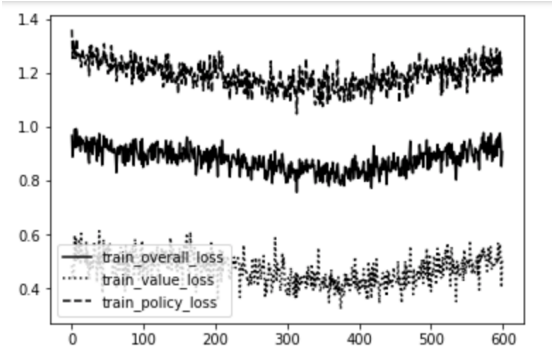

Figure 8. Loss of the model at the of the process

B. *Performance of your Agent in the ALife Environment*

We have not deployed our agent in the ALife[1] environment, because we think that it would not work well. In fact, our action space is discrete, whereas in ALife it is continuous.

VI. Conclusion and Future Work

To conclude, this project was an interesting and funny challenge: we have successfully created an environment and an intelligent agent to play Connect 4. We have fully designed a working environment for our agent, even if it could be improved even more to make it more intuitive and convenient for the human player. Furthermore, our agent, even if it not perfect, is still very satisfying and could also be improved thanks to more training. Our initial objective to build a perfect agent is not reached yet but it still has very good performances, even against a human player who can have a hard time to try to beat it. We will thus focus our energy into trying to improve our agent by tuning the parameters of the MCTS and allow it to make more tries.

Our future work, besides improving our agent and environment, could focus on adapting our agent to the ALife environment, for instance by discretizing the action space. It will be a challenging problem because the input of ALife (a picture) is really far from the input of Connect Four which is just a short list of numbers. However, we hope it will work since our neural network can in theory learn new games from scratch.

References

[1] http://www.informatik.uni-trier.de/~fernau/DSL0607/Masterthesis-Viergewinnt.pdf
[2] https://www.youtube.com/watch?v=6ELUvkSkCts&t=307s
[3] O. Pietquin. Deep Reinforcement Learning, https://moodle.polytechnique.fr/pluginfile.php/91215/mod_resource/content/1/DeepRL.pdf, February 2018.
[4] A Simple Alpha(Go) Zero Tutorial from Stanford.edu https://web.stanford.edu/~surag/posts/alphazero.html, December 2017.
[5] How to build your own AlphaZero AI using Python and Keras https://medium.com/applied-data-science/how-to-build-your-own-alphazero-ai-using-python-and-keras-7f664945c188, January 2018.
[6] Simulation of a perfect player http://connect4.gamesolver.org/?pos=

[1] https://github.com/jmread/alife