Artificial Intelligence in Zero Sum Games

Auguste Lehuger, Jean-Marie Lemercier, Ludovic Theobald

***Abstract*—This paper aims at explaining how to train an agent to play a zero-sum game with complete information such as Tic-Tac-Toe, Othello or Go. Given the rules of the game, this agent will have to train only through self-play without any human knowledge. To do so, we will rely upon AlphaZero's algorithm, DeepMind's latest breakthrough in the field. This algorithm comes in three part: the settings of the Game, the Monte-Carlo Search Tree (MCTS) methods and the Neural Networks that deduces the action to choose from a state of the board. The first step of the project is to create the pipeline for a simple board game such as Tic-Tac-Toe. Then, we will adapt it to more complex games.**

I. Introduction

The world of Artificial Intelligence has known many breakthrough in recent years around the fields of Machine Learning and Reinforcement Learning. Thanks to these technologies, Deepmind has managed to defeat the world champion Go player in 2016. Hence, these technologies bear a great potential to answer specific tasks. Our goal is to understand the way it works and to implement it on other tasks or games. Our motivation stemmed from the fact that we could not play against good IA in Blokus, a strategic board game.

A. Building the Game

Our first idea was to create an IA for Blokus, a well-know strategic game where you have to reduce the opponent space by placing different tiles on a 14x14 board. We developed it with python with the PyGame library and created a random IA that could play by the rules. Since Blokus' board is 14 by 14: it means they are 3^{14*14} possible configurations and there is around 1000 actions possible for each configuration. The size of the state space and the action space involves a great amount of training which led us to consider a more basic game: Tic-Tac-Toe. Hence, we defined the rules: what were the state space, the valid actions and the reward the agent received; and a basic display.

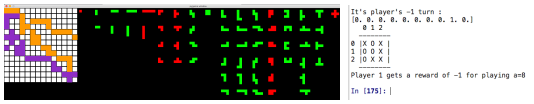

Fig. 1. Blokus environment vs. Tic-Tac-Toe environment.

B. Building the training pipeline

In order to train our agent TTTAI (Tic-Tac-Toe Artificial Intelligence) we followed DeepMind's [4] paper and the instruction gave by Surag Nair on his blog post [3]. The idea behind is to perform MCTS on the tree of possibilities to learn the value of various state of the board. Then, we use these newly learn data to fit a convolutional neural net. *Click here to find our source code: Run main.py to play our AI*

II. Background and Related Work

Two technical aspects need to be properly considered here : **Monte Carlo Tree Search** (MCTS) and **Convolutional Neural Networks** (CNN).

A. Monte Carlo Tree Search

We've broached in Chapter 7 of our 3third lecture [1] several tree search methods involving random choices and sampling. MCTS is one of those, and it's fairly recently used in algorithms as ours to compute the policies of actions e.g. the probability distributions $\pi(s)$ that will guide our AI's choice across the game. At each step, MCTS expands a node : if it's already been visited, it expands again, and marks the passage by raising a visit counter. Otherwise it adds the new node to the current tree. The principle in our actual implementation is further explained in IV-A

B. Convolutional Neural Network

Nothing new under the sun, we use a CNN for the reinforcement learning, which structure is summarized in IV-B. Those types of networks including convolutional layers grouped by 2 are particularly recommended when dealing with grid-based structures as our board. In each series of rollouts (we refer to them as *episodes*), we can extract training data of the form $(s, \pi(s), z)$. The neural network is a double headed network $\mathbf{f}_\theta$ such that $\mathbf{f}_\theta(s) = (\mathbf{P}_\theta(s), v_\theta(s))$. We train this neural network by minimizing the loss function

$$J(s,\theta) = ||z - v_\theta(s)|| + \pi(s)^T log(\mathbf{P}_\theta(s)) + \lambda||\theta||_2^2$$

At each step, we will train our network with the former training data (initialization is performed with uniform distributions on the action space for the policies $\pi(s)$) and compare its playing performances with a given threshold and update the former network if the criterion is met.

III. The Environment

Our objective is to create an environment that can be general for any zero-sum game with perfect information. Therefore, we created a general class called Game() that can generate any two-player, adversarial and turn-based game. This class contains several key methods *getNextState(self, state, player, action)* ; *getValidMoves(self, state, player)* or *getReward(self,player,state)* that creates Board object that deals with the specific game logic. Hence, the class Board() imposes rules

and the class Game() defined a standardized way to define the player, the state space and the action space for a parameter n.

- The player is either 1 or -1
- The state is a n*n numpy array
- The action is an integer between 0 and n*n +1

Since it is a perfect information game, the player has accessed to all the board before making a decision. His action consist of picking a tile to mark it with his color. The agent gets a reward only in the end of the game: +1 if he wins, -1 if he loses and 10^{-4} for a draw (in order to differentiate 'draw' and 'unfinished game'). Here, the agents choice impacts the board deterministically. The main challenge for the agent is to find a strategy without knowing in advance the behavior of his opponent and without searching in the entire tree of possibilities. Creating a powerful AI for Othello would already be a good real-world application as this game is played by thousand of players across the globe.

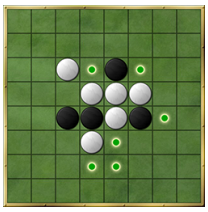

Fig. 2. A random board that can be generate from the Game class. Here n=8.

IV. The Agent

In this project, the agent is simply an algorithm simulating a player of the board game. The goal of the player is, of course, to win the game, that is selecting a sequence of actions a_i, leading to a final state s_T such that $reward(s_T) = 1$. To achieve this, the agent needs to evaluate a policy $\pi(s)$ giving the distribution of probability over the actions available from state s.

For this purpose, the algorithm can be broken down into two alternating parts detailed in II: a Monte-Carlo Tree Search (MCTS) and a neural network. The interaction between the entities and the superior algorithmic architecture is quickly summarized in Fig. 5

A. MCTS implementation

Our tree search is composed of several phases : the first is performing a *rollout* : we take for root a node **s** and perform a simulation e.g. we choose the action **a** that maximizes a utility function **U(s, a)** given by (1)

$$U(s,a) = Q(s,a) + c_{puct}.P(s,a).\frac{\sqrt{\sum_b N(s,b)}}{1+N(s,a)} \quad (1)$$

where

- ***N(s, a)*** is the visit counter : in a tree search, the action **a** has been taken ***N(s, a)*** times from node **s**

- ***Q(s, a)*** is the expected reward from playing action **a** at node (s)

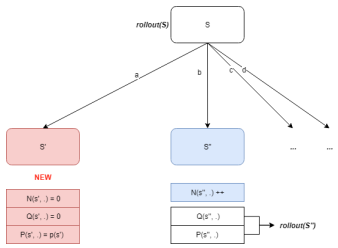

Fig. 3. Rollout system

- ***P(s, a)*** is the initial policy given by the neural network (see IV) and c_{puct} is a hyperparameter tuning the degree of exploration.

After playing action **a**, we update visit counter ***N(s, a)*** and we get to the new node **s'**. **s'** is either newly visited, in which case we simply add this node to the tree, and initialize ***N(s', b)*** and ***Q(s', b)*** to 0 for all actions b while calling the neural network for the policy ***P(s', .)***. In the other case, it has already been visited and then we call the function *rollout* recursively on **s'** until a new node or a terminal state is reached. In that case we back propagate to update all expected rewards $Q(s, a) in the path$.

The number of rollouts called at each node is a hyperparameter of our model. Once all rollouts have been performed and all values updated, we have produced examples e.g. samples which will be turned into policies $\pi(s)$ by just taking the empirical mean of the visit counter N(s, .) and normalize it. During a real game against a player, the agent will choose as an action $a^* = argmax_a(\pi(s))$.

B. CNN implementation

Our Convolutional Neural Network is composed of 2 hidden convolutional layers with batch normalization and same padding. We then add a softmax-activated dense layer to output the policies and actions.

Layer (type)	Output Shape	Param #	Connected to
input_17 (InputLayer)	(None, 3, 3)	0	
reshape_17 (Reshape)	(None, 3, 3, 1)	0	input_17[0][0]
conv2d_33 (Conv2D)	(None, 3, 3, 32)	320	reshape_17[0][0]
batch_normalization_65 (BatchNo	(None, 3, 3, 32)	128	conv2d_33[0][0]
conv2d_34 (Conv2D)	(None, 3, 3, 64)	18496	batch_normalization_65[0][0]
batch_normalization_66 (BatchNo	(None, 3, 3, 64)	256	conv2d_34[0][0]
flatten_17 (Flatten)	(None, 576)	0	batch_normalization_66[0][0]
dense_65 (Dense)	(None, 64)	36928	flatten_17[0][0]
batch_normalization_67 (BatchNo	(None, 64)	256	dense_65[0][0]
dense_66 (Dense)	(None, 128)	8320	batch_normalization_67[0][0]
batch_normalization_68 (BatchNo	(None, 128)	512	dense_66[0][0]
dropout_34 (Dropout)	(None, 128)	0	batch_normalization_68[0][0]
dense_67 (Dense)	(None, 1)	129	dropout_34[0][0]
dense_68 (Dense)	(None, 10)	1290	dropout_34[0][0]

Total params: 66,635
Trainable params: 66,059
Non-trainable params: 576

Fig. 4. A summary of the layers of the CNN implemented.

C. Superior algorithm

The algorithm manipulates both this tree search and the neural network to deliver optimal results. On one hand, Monte Carlo Tree Search has a limited computing power to deliver a

somehow empirical optimal solution by performing rollouts: the neural network then helps increasing accuracy in the values for newly created nodes, thus converging faster toward an optimal strategy. On the other hand, the CNN is only efficiently trainable because the tree search allows to create consistent and accurate data with the rollouts.

In that way, structures help each other increasing their performances and that is the goal of our simple AlphaZero implementation. The interaction process is represented on Fig. 5.

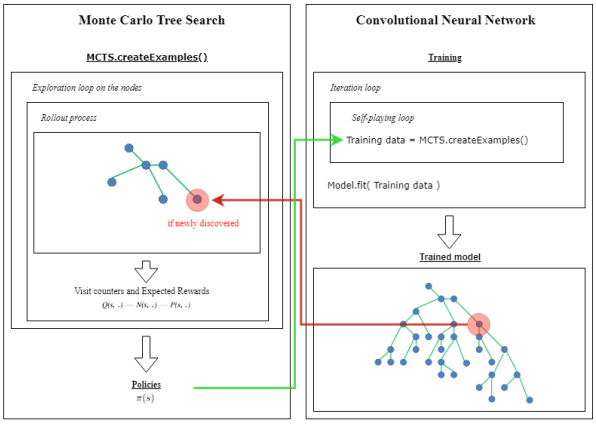

Fig. 5. Interaction between the superstructures of the algorithm

V. Results and Discussion

The results shown here are obtained for the game Tic Tac Toe, which is very simple.

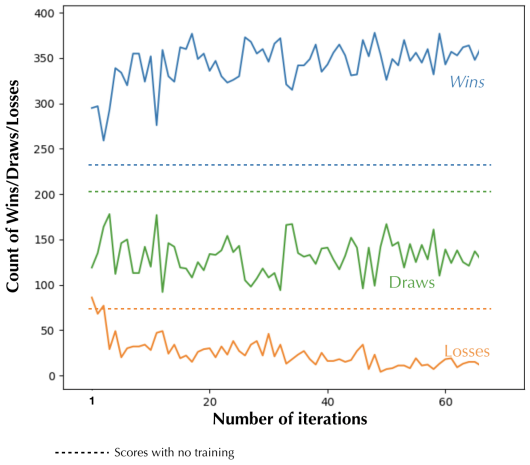

Fig. 6. Performances of the agent over iterations

1) Evolution of performances with the number of iterations: To assess the performances of our agent, we confronted it against a player acting randomly, and we counted the amount of games won, lost or which resulted in a draw. In Fig. 7, we plotted these counts - for 500 games - against the number of iterations of training the agent has been through, when the agent does not do the first move. The dotted lines are the performance of a non-trained agent, performing Monte-Carlo rollouts with no knowledge from the neural network.

This figure shows a significant improvement from even a single iteration of training, and a progressive improvement over time.

The last model obtained was the 10x20_shallow_50 trained on 50 iterations for 10 epochs each and with an MCTS performing 20 rollouts. This model stands out to be the most promising one according to the evolution graph. As a measure of performance, we made him play against a random player for 1000 games, the first time TTTAI starts; the second time Random starts.

TABLE I
Competition over 1000 games: random IA against trained TTTAI

Environment config.	Wins	Losses	Draws
TTTAI plays P1	982	5	13
TTTAI plays P2	849	13	138

Therefore our trained algorithm outperforms an untrained algorithm that only performs rollouts. This is promising but still the opponent were rather naive. We can ask ourselves if the neural networks really learn something insightful ? Here are three different boards that describe different situation: an opening, a defense move and an attack. Below one can see the policy predicted by the network.

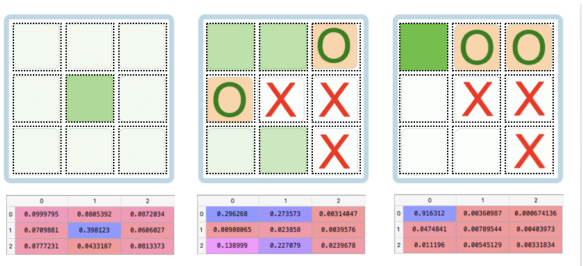

Fig. 7. Prediction of the agent on 3 boards. (opening, defense, attack

The efficiency of the neural network to choose the right action for a given board is what makes this algorithm so powerful. Indeed, we made compete TTTAI with only 2 rollouts against an untrained network with 150 rollouts, we realized TTTAI where winning in landslide (70% of the time even when it played as P2).

VI. Conclusion and Future Work

At first, we planned on creating an Artificial Intelligence that could play and beat a player at the game Blokus, using the same general method of AlphaZero. The environnement, logic, and GUI corresponding to this game has been fully implemented. However, this game leads to an action space and a space state that are huge, and that would take month to train with the AlphaZero framework. Hence, we here train a agent to play a much simpler game : TicTacToe. Although TicTicToe is solvable easily by a simple tree search - it has a very small state tree - using AlphaZero method enabled us to reimplement its different parts and to test them on a quick

example. Now that our training algorithm is shown to work, we can easily extend it to more complex games.

Therefore, the natural evolution of our project is to apply this training method, first to Othello, whose actions are really close from those of TicTacToe, and then to Abalone or Blokus. Moreover, some improvements can still be done on the training pipeline. In particular, training data can be augmented by taking into account the symmetries of the game.

A. Motivation for the Othello game

After demonstrating the efficiency of our methodology and algorithm of Toc Tac Toe, we decided to apply it on the game Othello. This game differs from Tic Tac Toe in two aspects : first, the board is bigger, and secondly, the rules are differents. However, the structure of the actions stays the same as for Tic Tac Toe. At each turn, each player only as to choose one position on the board. This similarity in the structure of the action space makes Othello a natural extension to Tic Tac Toe.

B. Results on Othello

TABLE II
COMPETITION OVER 100 GAMES: RANDOM IA AGAINST TRAINED OTHELLOAI

Environment config.	Wins	Losses	Draws
OthelloAI plays P1	37	50	13
OthelloAI plays P2	30	55	15

As TABLE II shows, the agent playing Othello is not good, and worst than a random agent. We propose two explanations to this inefficiency. First, some *"depth recursion error"* sometimes shows up while doing the training, which suggests that the rollouts are not performed correctly (they run indefinitely) for the Othello game. This error is intrinsically linked to our coding of the rules of Othello since it didn't happened for Tic Tac Toe. Secondly, the model was trained with only 5 iterations of 15 episodes each, which already took around 10 minutes, using a rather shallow neural network, and performing only 20 rollouts per node. Fully training the agent to play on a 8×8 board would take several days, since everything is longer in this case : the training and evaluation of the network, the state-action tree, and the time to converge to a good policy. Code for it can be found *here*.

REFERENCES

[1] J. Read. Lecture III - Search and Optimization. *INF581 Advanced Topics in Artificial Intelligence*, 2019.

[2] O. Pietquin. Lecture VIII - Scaling up Reinforcement Learning. *INF581 Advanced Topics in Artificial Intelligence*, 2019.

[3] S. Nair. A Simple Alpha(Go) Zero Tutorial https://web.stanford.edu/~surag/posts/alphazero.html, 2017.

[4] D. Silver, T. Hubert and J. Schrittwieser A general reinforcement learning algorithm that masters chess, shogi and Go through self-play https://deepmind.com/documents/260/alphazero_preprint.pdf, 2017.