AlphaOthelloZero

Amine CHERIF HAOUAT, Sami JALLOULI, and Tom LABIAUSSE

CentraleSupélec - MDS: Reinforcement Learning

Abstract. Board games like Connect4, Othello, Chess or Go have long served as a testing ground for artificial intelligence research. Between 2015 and 2018, a research team at *DeepMind* developed a series of computer programs based on Reinforcement Learning and reach human professional level at the game of Go for the first time in history. We built on this foundational work to create an agent that can learn to play Othello at a high level through self-play only. This agent relies on a Policy-Value network combined with tree search for policy improvement and we compared it to random, greedy and Monte Carlo Tree Search baselines. We also included and tested simpler games like TicTacToe and Connect4 in our framework.

1 Introduction

Board games like Connect4 [1], Othello [2], Chess or Go have long served as a testing ground for artificial intelligence research. In 1997, *DeepBlue* [5] was the first computer system to beat a Chess world champion in regular conditions and relied on a IBM supercomputer for its computations. As impressing as it may sounds, Chess is a relatively simple game compared to Go where the branching factor is way larger and it is much harder to evaluate a non-terminal state. For these reasons in particular, it was estimated that the game of Go will not be solved for several decades.

However, between 2015 and 2016, a research team at *DeepMind* developped *AlphaGo* [10]: the first computer program able to play Go at a human professional level. The algorithm used a combination of Monte Carlo Tree Search (MCTS) with deep neural networks trained both on human and computer play. They next version of *AlphaGo* called *AlphaGoZero* [11] was able to beat *AlphaGo* by training in a self-play manner only. In 2018, *DeepMind* presented *AlphaZero* [12], a generalized version of *AlphaGoZero* for the games of Chess, Go and Shogi. This exciting Reinforcement Learning (RL) research paved the way for applications of the same techniques to other games but also to a wide variety of problems like proteins folding with *AlphaFold* [13] or optimal matrix operations with *AlphaTensor* [14].

In our work, we chose the game of Othello to implement an *Alpha(Go)Zero* approach. It is indeed a much simpler problem than Go or even Chess and therefore requires less computing power to obtain interesting performance. We started by building the game environment as well as simple players with random and greedy strategies serving as baselines. We then implemented the MCTS procedure which is reviewed in [6] and tested it against the previous players. At last, we implemented the *Alpha(Go)Zero* training pipeline to train *AlphaZero* agents on different board games. In order to speed up our development phase, we also created the simpler game environments of TicTacToe and Connect4. We ran contests between players on different games and analyzed the results.

2 Methodology

2.1 Monte Carlo Tree Search

The *Alpha(Go)Zero* approach described in the context of the game of Go in [11] builds on a previous well-known technique used for combinatorial games called MCTS. In this section, we give a high-level overview of this procedure.

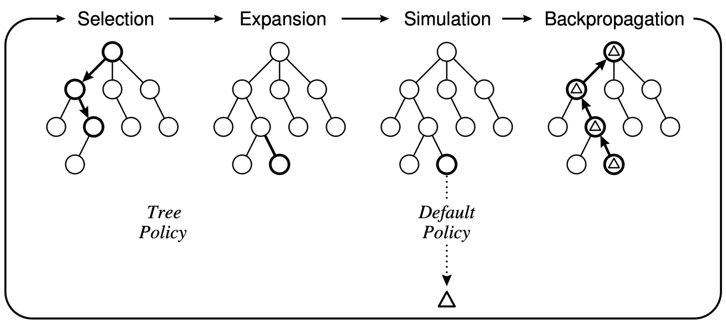


Fig. 1: Monte Carlo Tree Search (MCTS) algorithm from [6]

In a classic RL framework, MCTS is a method that can be used to identify the best actions to take given the state of an environment. As suggested by the expression Monte Carlo, it is a model-free method as it relies on simulations to build estimates of action values. It is also a tree search algorithm where each node corresponds to a state of the environment with edges being the actions allowing to transit from a state to another. As explained in the section 3 of [6]: MCTS builds a game tree by expanding the most promising nodes following a *Tree Policy* and evaluating them using a random policy called *Default Policy*. At each round of a game and given a fixed computational budget, MCTS loops over the fours steps of **figure 1** in order to grow the search tree and updates the average reward value of the nodes. In a zero-exploration mode, the action corresponding to the child with the highest visits count is ultimately chosen.

The first key element of MCTS is the *Tree Policy* used during *Selection*. Indeed this policy needs to maintain some balance between the exploitation of promising actions (i.e. nodes with high average reward Q) and the exploration of actions that were not tested enough and may lead to good outcomes (i.e. nodes with low visits count N). The *Upper Confidendce bound applied to Tree* (UCT) formula introduced in [8] is a common criteria ensuring a reasonable balance as it is based on the well-known *Upper Confidence Bound* (UCB) formula derived in [7]. During *Selection*, the associated *Tree Policy* selects the child with the highest UCT value computed with the following formula:

$$UCT = Q + c_{uct}\sqrt{\frac{log(N_{parent})}{N}}$$

N_{parent} is the visits count of the parent node and c_{uct} is a hyperparameter balancing exploitation and exploration set to $\sqrt{2}$ for theoretical reasons but usually adjusted empirically. When $N = 0$ for a given node, it is common to consider that the associated UCT value is ∞ so that all children of a given node are considered at least once before any grandchild is created.

The second key element of MCTS is the *Simulation* phase where a given node is evaluated using the outcome resulting from a *Default Policy* applied until a terminal state is reached. This gives an estimation of the value of the state which is then backpropagated to the root node by updating Q values on its path.

2.2 MCTS with Policy-Value Network

The brilliant idea of the *Alpha(Go)Zero* authors is to use a neural network as an approximation function to obtain prior probabilities for the MCTS *Selection* as well as a better evaluation during the *Simulation* phase. This neural network f_θ parametrized by θ has two heads: a *Policy* head and a *Value* head. It takes as input the state of the environment which is simply a raw representation of the board s and outputs a vector of prior action probabilities p through its *Policy* head and a scalar estimation of the value of the state v for the current player through its *Value* head. In the following, we will consider that $f_\theta(s) = (p, v)$.

Every type of *AlphaZero* agent still relies on the MCTS procedure to select actions. However, it uses a new criteria in the *Selection* phase built as a variant of PUCT [9] given by the following formula:

$$PUCT = Q + c_{puct}P(s,a)\frac{\sqrt{N_{parent}}}{1+N}$$

$P(s, a)$ is the prior probability of taking action a from state s given by f_θ. More formally, $P(s, a) = p_a$. Intuitively, when the visits count N of a node is low, then $P(s, a)$ has an important influence in the action selection. As the visits counts of the nodes increase, the importance of the exploration term (second term) in the PUCT decreases and the selection starts to rely more on the Q values. The neural network also replaces the MC rollouts of the *Simulation* phase as the predictions v of its *Value* head are used to update the Q values of the nodes during the *Backpropagation* phase.

2.3 Self-play and Policy Iteration

In order to use the neural network f_θ as part of the neural tree search procedure built from MCTS, one must first find an appropriate set of parameters θ. As explained in [11] and [12], this can be done by only using self-play as part of a policy iteration procedure. In simpler terms, the training of the neural network doesn't require data collected from human games but can learn by itself the best *a priori* actions to take given a specific state as well as an evaluation of that state.

Given a neural network f_θ, it is possible to play an episode of the game using the procedure of **section 2.2**. From each visited state s, one can store the MCTS action probabilities vector $\pi(s)$ obtained from the visit counts of the children of the root node at each step. The true winner of this self-play game is denoted by z. It then becomes possible to improve the predictions (p, v) of the neural network through gradient descent on θ using the following loss function:

$$loss = (z - v)^2 - \pi(s)^T log(p)$$

As explained in [11], MCTS can be viewed as a policy improvement operator as the probabilities $\pi(s)$ generally indicate much stronger moves than the raw prior p given by network. The loss function is indeed built to improve the moves selected by the network using the second cross-entropy term between distributions. The self-play procedure leading to an actual winner z can be viewed as a policy evaluation operator. Therefore, by alternating self-play games producing data and network optimization, it follows a form of policy iteration.

3 Experiments

3.1 Implementation and AlphaZero training

We implemented everything from scratch in Python using only main librairies such as *numpy* and *pytorch*. We trained *AlphaZero* agents for the games of TicTacToe, Connect4 and Othello and reported the best results we were able to obtain. As Othello can be played with various board sizes, we chose a 6×6 board as it is faster to train and still reasonably complex. We implemented most of the features described in [11] such as temperature annealing, dirichlet noise and data augmentation through board symmetries and rotations. All configurations used for training and evaluatuin are listed in **table 1**. Most values were taken from [11] but we changed some of them to adjust to each game. In particular, we took inspiration from [12] as the neural network is automatically replaced by its newer version at each iteration instead of competing against its previous version.

We chose common architecture for the neural networks. For TicTacToe, we used a simple fully connected network with 2 hidden layers of 9 units each with batch normalization layers. For Connect4 and Othello, we used the same architecture which is a succession of 4 convolutional layers with 64 filters of size 3x3 each followed by two fully connected layers with hidden dimensions of 1024 and 512. More details are given in **table 2**.

3.2 Baselines

In order to evaluate our *AlphaZero* agents, we used three baseline players which are: *RandomPlayer*, *GreedyPlayer* and *MCTSPlayer*. We ran fair contests (each player starts 50% of the games) between these players on Othello 6×6 and obtained the results presented in **table 3**. As expected, MCTS outperforms the naive baselines while the greedy approach is slightly better than pure random.

3.3 Results and discussion

We first trained an *AlphaTicTacToeZero* agent as it is faster and it allowed us to debug our implementation. **Figure 2** shows the evolution of the loss during training. As one can see, the neural network obtained at the end of the training doesn't necessarily have the lowest error but we still chose it to play against the baselines. **Table 4** shows the results of the contests between *AlphaTicTacToeZero* and our three baselines. As expected, the *AlphaZero* agent outperforms the random and greedy players and is able to beat the MCTS player in most games. As it is the case with good TicTacToe players, a lot of games end in a draw but the *AlphaZero* agent never loses when starting the game. **Figure 4** showcases some actions performed by the *AlphaZero* agent during games.

We then trained an $AlphaOthello_{6\times6}Zero$ agent and obtained similar results but with much more regular loss curves as it can be seen in **figure 5**. The *AlphaZero* agent quickly outperformed our strongest baseline MCTS during training as shown in **figure 6**. As with TicTacToe, we selected the final neural network to perform evaluation against all baselines in **table 5** demonstrating the impressive superiority of the agent. **Figure 7** presents some stategies used by the *AlphaZero* agent during games which is mainly focused on targeting edges and corners as expected.

We were not able to train a strong *AlphaConnect4Zero* agent using the same number of self-play games as Othello. Indeed, by exploiting the symmetries in Othello, we were able to perform efficient data augmentation compared to Connect4 where we couldn't use rotations. Training efficiently for Connect4 would then require more self-play games and therefore take longer.

4 Conclusion

In this project, we implemented a complete framework to train *AlphaZero* agents on board games such as TicTacToe, Connect4 and Othello. We were able to obtain agents which significantly outperform random, greedy and MCTS baselines on TicTacToe and Othello. Further work could focus on evaluating against new baselines such as an *AlphaBetaPlayer* or improving the code efficiency with parallel evaluation of positions by the neural network during self-play as it was of course done by the original research team at *DeepMind*.

References

1. *Connect Four* - Wikipedia.

2. *Othello (Reversi)* - Wikipedia.

3. *Chess* - Wikipedia.

4. *Go* - Wikipedia.

5. M.Campbell , A. Joseph Hoane Jr., F.Hsu: *Deep Blue* , CMU & IBM (1997)

6. C.B.Browne et al.: *A Survey of Monte Carlo Tree Search Methods* , IEEE (2012)

7. P.Auer, N.Cesa-Bianchi, P.Fischer: *Finite-time Analysis of the Multiarmed Bandit Problem* , Machine Learning (2002)

8. L.Kocsis, C.Szepesvari: *Bandit based Monte-Carlo Planning* , ECML (2006)

9. C.D.Rosin: *Multi-armed bandits with episode context* , Ann Math Artif Intell (2011)

10. D.Silver et al.: *Mastering the game of Go with deep neural networks and tree search* , Nature (2016)

11. D.Silver et al.: *Mastering the game of Go without human knowledge* , Nature (2017)

12. D.Silver et al.: *Mastering Chess and Shogi by Self-Play with a General RL Algorithm* , Science (2018)

13. J.Jumper, R.Evans, A.Pritzel et al.: *Highly accurate protein structure prediction with AlphaFold* , Nature (2021)

14. A.Fawzi, M.Balog, A.Huang et al.: *Discovering faster matrix multiplication algorithms with reinforcement learning* , Nature (2022)

Appendix

AlphaZero training configurations			
	TicTacToe	Connect4	Othello 6x6
board width	3	7	6
board height	3	6	6
simulations	100		
dirichlet alpha	0.03		
dirichlet epsilon	0.25		
temperature scheduler type	linear		
temperature max step	2	3	4
temperature min step	2	3	4
iterations	30	10	10
epochs	10		
episodes	100	>200	200
batch size	64		
learning rate scheduler	linear ($\gamma = 0.9$)		
start learning rate	0.01		
data augmentation	sym + rot (×8)	sym (×2)	sym + rot (×8)

Table 1: *AlphaZero* training configurations (see subclasses of *Config* in the code)

Neural networks			
	TicTacToe	Connect4	Othello 6x6
input size	9	6×7	6×6
policy output size	9	$6 \times 7 = 42$	$6 \times 6 + 1 = 37$
# trainable parameters	316	43,208	707,782

Table 2: Main characteristics of the neural networks used for each game. The number of parameters is much higher for Othello as it has a larger action space and thus more connections between the last fully connected layers. Moreover, players are allowed to pass in Othello which is why the network outputs an extra value for the pass action compared to the number of cells in the board.

MCTS vs Random			
win	loss	draw	total
99	1	2	100

MCTS vs Greedy			
win	loss	draw	total
100	0	0	100

Greedy vs Random			
win	loss	draw	total
5,995	3,530	475	10K

Table 3: Comparison of baselines with Othello 6×6 (winner in bold)

AlphaZero vs Random				
first to play	win	loss	draw	total
AlphaZero	462	0	38	500
Random	423	2	75	500
total	885	2	113	1K

AlphaZero vs Greedy				
first to play	win	loss	draw	total
AlphaZero	241	0	259	500
Greedy	78	10	412	500
total	319	10	671	1K

AlphaZero vs MCTS				
first to play	win	loss	draw	total
AlphaZero	124	0	376	500
MCTS	7	13	480	500
total	131	13	856	1K

Table 4: Evaluation of *AlphaTicTacToeZero* against baselines (winner in bold)

AlphaZero vs Random				
first to play	win	loss	draw	total
AlphaZero	50	0	0	50
Random	50	0	0	50
total	100	0	0	100

AlphaZero vs Greedy				
first to play	win	loss	draw	total
AlphaZero	50	0	0	50
Random	50	0	0	50
total	100	0	0	100

AlphaZero vs MCTS				
first to play	win	loss	draw	total
AlphaZero	44	6	0	50
MCTS	46	2	2	50
total	90	8	2	100

Table 5: Evaluation of $AlphaOthello_{6\times6}Zero$ against baselines (winner in bold)

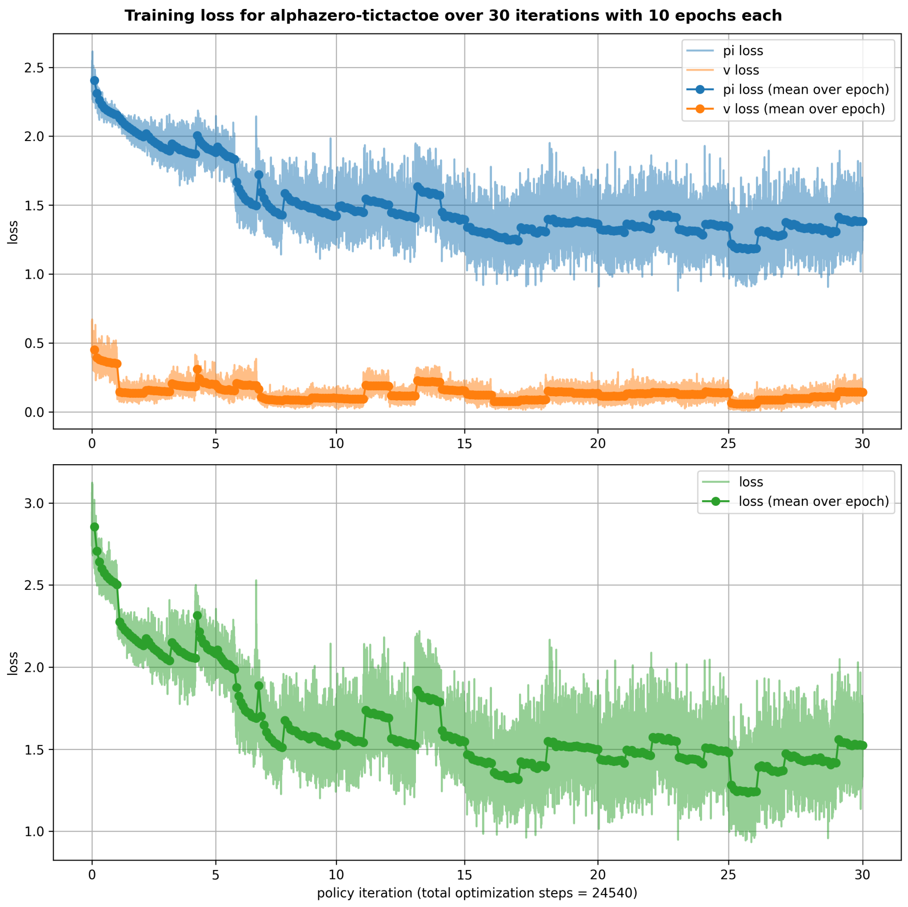


Fig. 2: Training loss of *AlphaTicTacToeZero*. The blue (resp. orange) curve is the policy (resp. value) component of the loss computed for each batch during the whole training. The green curve is the total loss (sum of both components) over the training. Jumps in the curves happen at the beginning of a new optimization phase during policy iteration and are caused by the apparition of new policy targets π computed during the previous self-play phase.

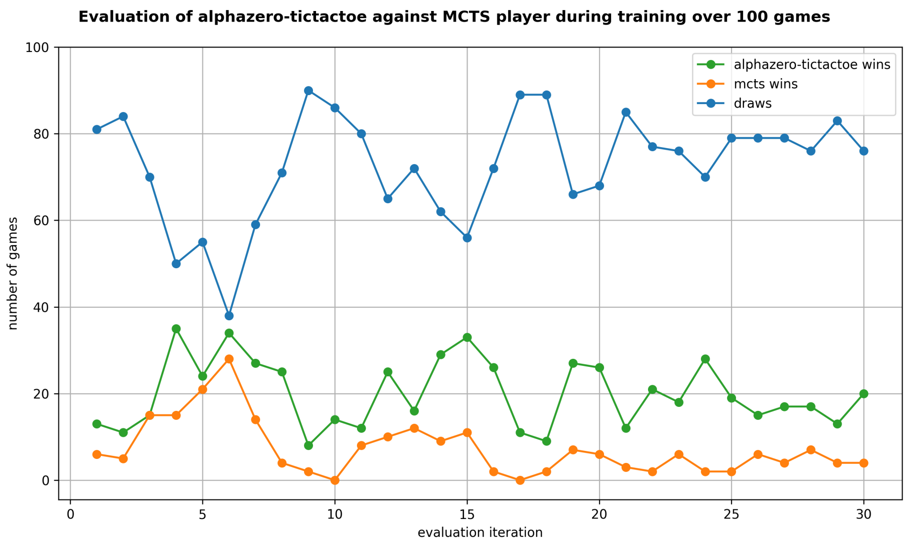
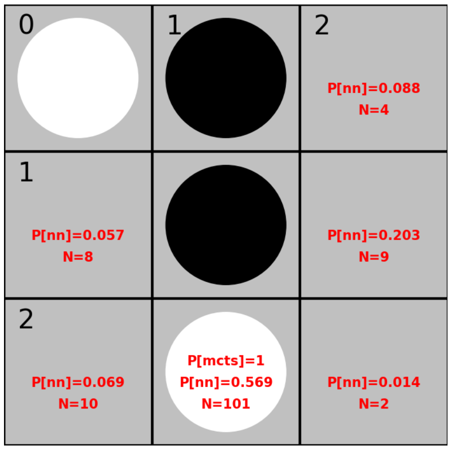
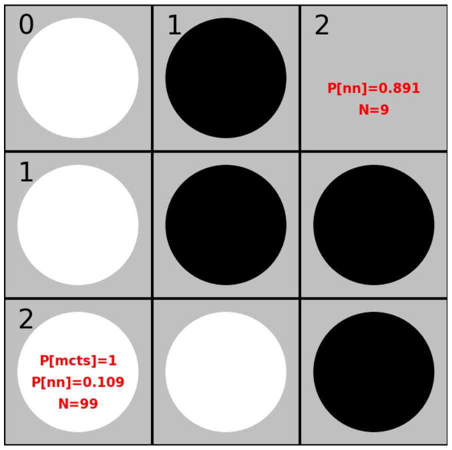


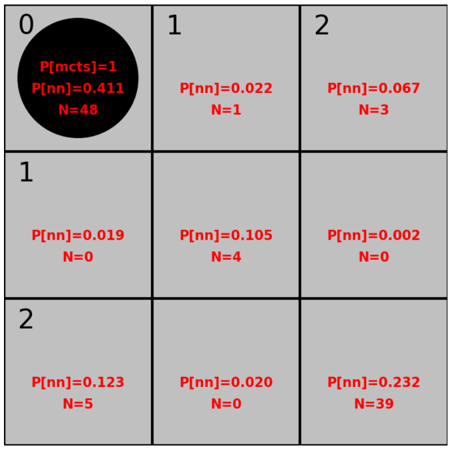
Fig. 3: Evaluation of *AlphaTicTacToeZero* during training.



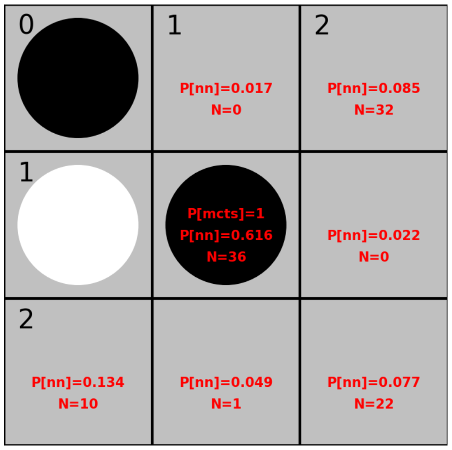
(a) *AlphaZero* plays (row=2,col=1)



(b) *AlphaZero* plays (row=2,col=0)



(c) *AlphaZero* plays (row=0,col=0)



(d) *AlphaZero* plays (row=1,col=1)

Fig. 4: Examples of action evaluation and selection by *AlphaTicTacToeZero*. Boards represent the state of the game after a move of the *AlphaZero* agent in different games. Annotations give the prior probabilities $P[nn]$ given by the neural network over the legal moves, the visits count N of the nodes in the search tree and $P[mcts] = 1$ indicates the action selected by the *AlphaZero* agent corresponding to the position with the highest visits count as temperature is set to 0 (i.e. the strongest action is chosen). *AlphaZero* agent plays as white in (a) and (b) and as black in (c) and (d).

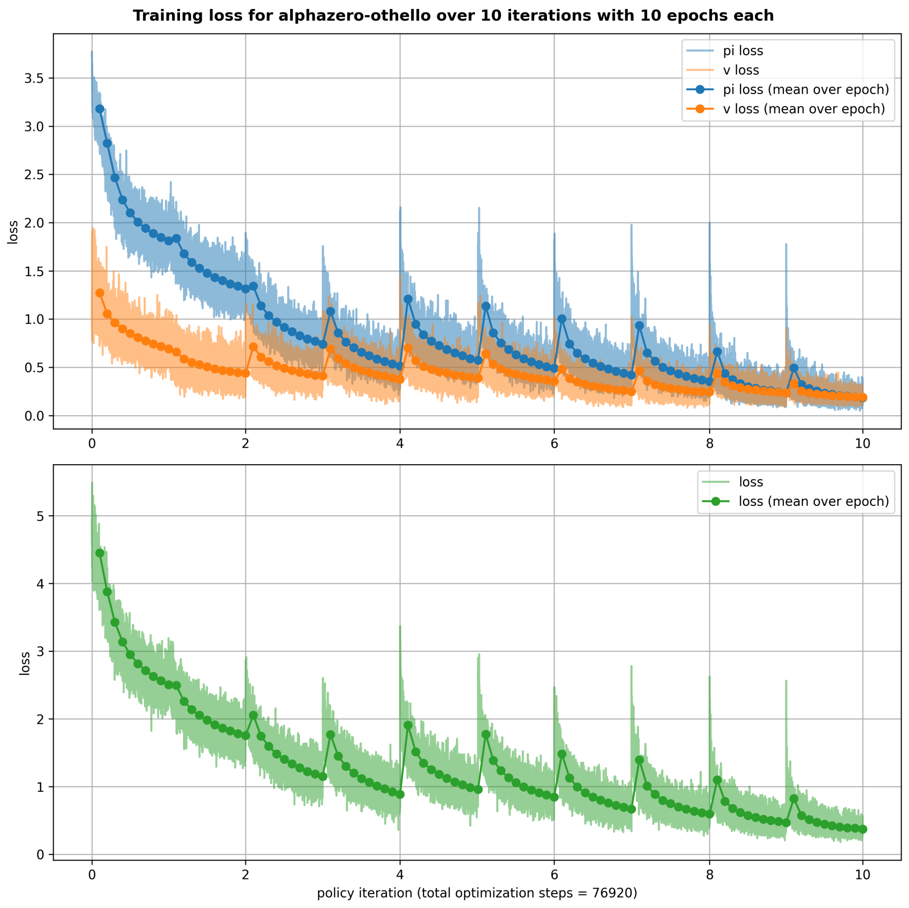


Fig. 5: Training loss of $AlphaOthello_{6\times6}Zero$. The blue (resp. orange) curve is the policy (resp. value) component of the loss computed for each batch during the whole training. The green curve is the total loss (sum of both components) over the training. Jumps in the curves happen at the beginning of a new optimization phase during policy iteration and are caused by the apparition of new policy targets π computed during the previous self-play phase.

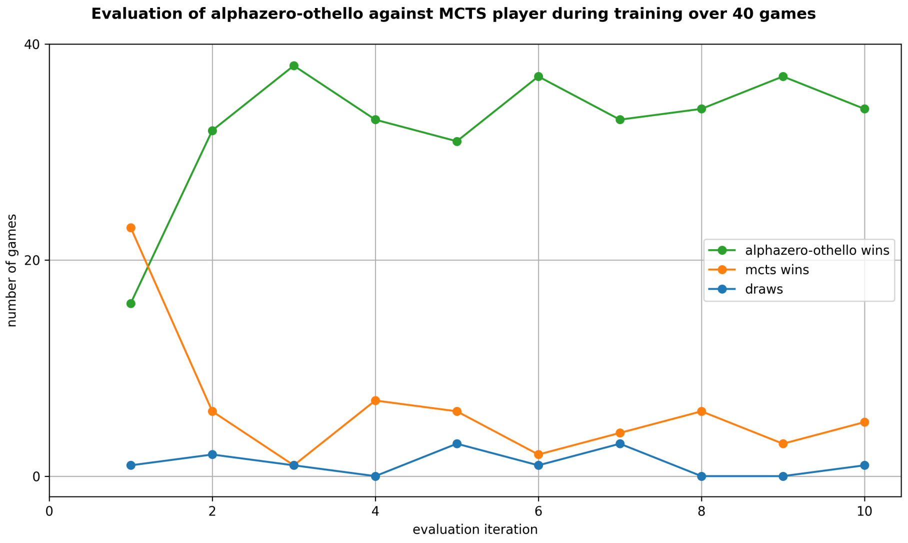
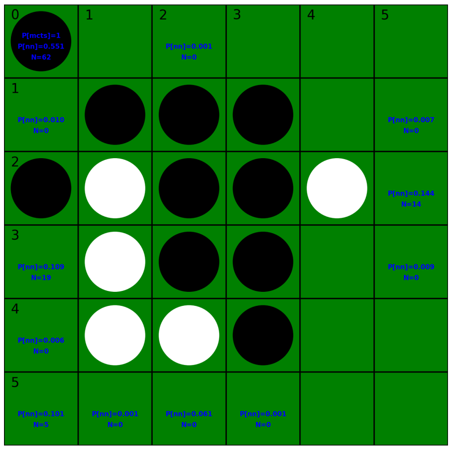
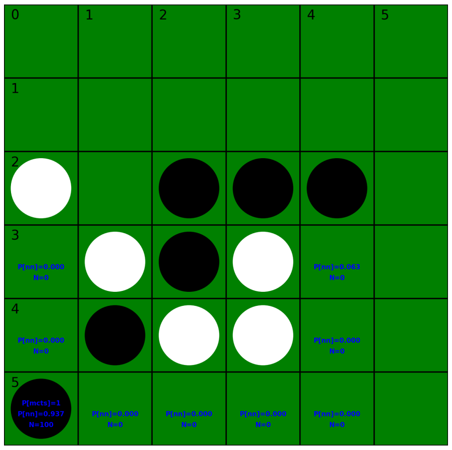


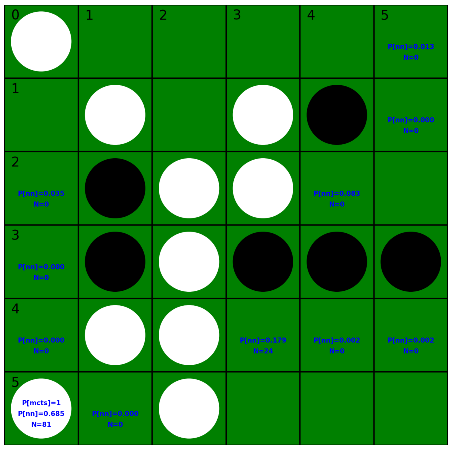
Fig. 6: Evaluation of $AlphaOthello_{6\times6}Zero$ during training.



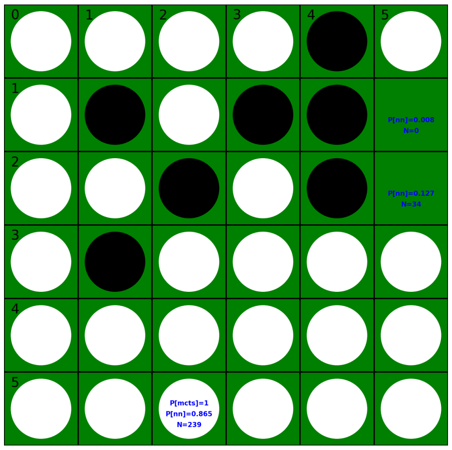
(a) *AlphaZero* plays (row=0,col=0)



(b) *AlphaZero* plays (row=5,col=0)



(c) *AlphaZero* plays (row=5,col=0)



(d) *AlphaZero* plays (row=5,col=2)

Fig. 7: Examples of action evaluation and selection by $AlphaOthello_{6\times6}Zero$. Boards represent the state of the game after a move of the *AlphaZero* agent in different games. Annotations give the prior probabilities $P[nn]$ given by the neural network over the legal moves, the visits count N of the nodes in the search tree and $P[mcts] = 1$ indicates the action selected by the *AlphaZero* agent corresponding to the position with the highest visits count as temperature is set to 0 (i.e. the strongest action is chosen). *AlphaZero* agent plays as black in (a) and (b) and as white in (c) and (d).