Trying to master the game of Tablut with Alpha Zero and REINFORCE

Michele Faedi [1*]
[1]University of Bologna
michele.faedi@studio.unibo.it

Abstract

Tablut is an unbalanced table game with two player and two distinct role and little work on. In this paper we will present an implementation of two different algorithms: Alpha Zero, inspired by the groundbreaking work of Silver et al. in "Mastering the game of Go without human knowledge.", and REINFORCE, a well known reinforcement learning algorithm. Due to limitations in computational power, a two-step approach is investigated. Initially, the REINFORCE algorithm will be used as a pretraining step to provide a baseline for the subsequent Alpha Zero implementation. This approach should allow for the exploration of game dynamics and initial learning of state-value and policy approximation. Alpha Zero is a well known algorithm with slow convergence, in some case requiring weeks of training with self play. We will compare 4 players: REINFORCE, Alpha Zero, random player and a MinMax with alpha beta cuts and iterative deepening.

1 Introduction

The game of Tablut, with its complex rules and strategic elements, poses a significant challenge for artificial intelligence (AI) agents due to its unbalanced nature and a different victory probability for the two players. The groundbreaking work of Silver et al.[2] demonstrated the success of the AlphaGo Zero algorithm in achieving superhuman performance in the game of Go. Inspired by this work, we aim to develop an AI agent capable of mastering Tablut using the AlphaGoZero algorithm adapted to this game.

However, our research faces two significant challenges: limitations in computational power and time required for Alpha Zero to converge. To address the challenges, we adopt a two-step approach. In the first step, we employ the REINFORCE with baseline algorithm as a pre-training phase. This initial implementation allows the neural network to learn a basic policy and a state value approximation.

After this initial phase the model is finetuned by applying the AlphaGo Zero framework. We investigate the hypothesis that the neural network trained with the REINFORCE algorithm can be directly applied to the Alpha Tablut Zero without extensive fine-tuning. This approach aims to leverage the knowledge acquired during the REINFORCE pre-training phase and enhance the performance of the Alpha Tablut Zero algorithm.

The rest of the paper is structured as follows: Section 2 provides an overview of the related work in AI game-playing algorithms and highlights the unique challenges presented by Tablut. Section 3 describes the source code wrote in python. Section 4 presents the experimental results and analyzes the performance of our approach compared to the baseline. Section 5 discusses the implications of our findings and explores potential future directions for improving AI gameplay in Tablut. Finally, Section 6 concludes the paper with a summary of our contributions and the significance of this research in advancing AI game-playing algorithms.

Preprint. Under review.

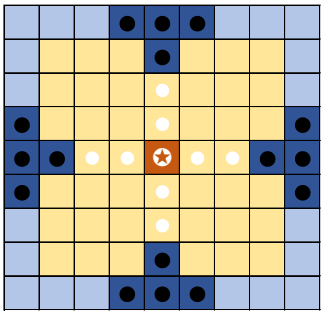


Figure 1: Initial configuration of the board

Tablut game posed a remarkable challenge for the models to learn an effective strategy, for this reason the same approach has been applied to another table game: tic tac toe large. This game have a much simpler action and state space, no loops, no status quo, game last at maximum 81 steps. The rule of the game are simple: in a board nine by nine the players have to position the pieces in order to have at least four pieces aligned and consecutive.

2 Related work

2.1 Challenges of tablut

The game of tablut is a little known table game. There is a study by Galassi [1] that calculate an upper bound of the state space of tablut: his results shows that tablut is comparable to the game of Draughts.

Tablut is a strategic board game played on a 9x9 grid. The game involves two players, one controlling the Swedes(the white) and the other the Muscovites(the black). The objective of the Swedes is to escort their King from the castle to one of the edges of the board, while the Muscovites aim to capture the King. Pieces move orthogonally (like the towers in chess) and capture by surrounding an opponent's piece on two opposing sides, only one in the case there is a wall on the opposing side. The King move only by one cell and if in the castle requires four captures to be captured, only 3 if is touching the castle. The game ends when the King is captured or the Swedes successfully escort the King to a corner. In the Figure 1 you can see the scenario.

2.2 MinMax baseline

In a previous iteration of this project we developed a MinMax player: an AI agent implemented using the MinMax algorithm with iterative deepening and alpha beta cuts. That agent implementation will not be discussed in this paper but will be used as a comparison.

3 Methodologies

3.1 State and action representation

The state is a matrix of by $9 \times 9 \times 9$ composed by: 7 channels for the boards history, plus 1 for the current board and one channel for the current player (1 for the white and -1 for the black).

The action space is a $9 \times 9 \times 18$ matrix. Two channels are for the piece selection, and the last one for the next position along the orthogonal axis.

3.2 REINFORCE Algorithm as Pretraining

The Alpha Zero algorithm assigned 5 second for the Monte Carlo Tree Search phase, this means that for a game of 500 actions the time for a single match is more than 40 minutes. This would require too much time for the model to converge, in order to avoid this problem we decided to pretrain the model using REINFORCE algorithm. REINFORCE on the other hand suffer from having a high variance

and a slow convergence, to overcome this problem a second neural network is used as a baseline that weight the policy loss with the difference between the expected and true reward; this variant is called REINFORCE with baseline. In 3.2 you can see the loss function: **w** are the weights for the state value network and θ are the weights for the policy network

During the training it was observed that the agents tends to repeat the same moves entering in a loop. The convergence of REINFORCE is not granted anymore and this problem make the pretraining mostly useless. We tried to solve this problem by adding a penalty to the actions that leads to a repeated state, called $\pi(A_r|S_t, \theta)$. This helped to reduce the average episode length.

$\delta \leftarrow G_t - \hat{\mathrm{v}}(S_t, \mathbf{w})$

$\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}} \delta \nabla \hat{\mathrm{v}}(S_t, \mathbf{w})$

$\theta \leftarrow \theta + \alpha^{\theta}(G_t - \hat{\mathrm{v}}(S_t, \mathbf{w})) \nabla \ln \pi(A_t|S_t, \theta)$

- $\alpha^{\theta} \nabla \ln \pi(A_r|S_t, \theta)$(1)

3.3 Alpha zero

The paper by Silver et al. [2] demonstrates that by selecting the best action directly, the agent in AlphaGo Zero performs reasonably well. However, when the same model is enhanced using the Monte Carlo Tree Search (MCTS), the agent achieves a state-of-the-art elo rating. Building on this insight, we aimed to improve the REINFORCE model using the MCTS approach. Algorithm 1 shows the pseudo code for the Monte Carlo Tree Search of AlphaZero. The data structure for the states tree is tree, this became a compulsory choice if the tree is reusable in the subsequent steps to discard dead branches.

3.4 Hyperparameters

The development of this project encountered several challenges, primarily from the difficulty of assessing whether the model was genuinely learning a good policy or merely avoiding repetitive actions. Consequently, no hyperparameters were tuned; however, it is still relevant to outline their role and significance. (1) neural architecture: is one of the most important and complex hyperparameter to tune. Due to computational power constraint we used a simple neural network with 1.3M parameters. (2) step size: REINFORCE use two distinct neural networks and so they may require two different step size to learn, also one can converge faster than the other, but without any evidence of the effectiveness of this parameter we used $1e-4$. Since the loss is not a measure of convergence the step size can be tuned only by trial and errors over long episodes of train. (3) temperature and $C_{\texttt{puct}}$: these are hyperparmeters of the MCTS, the temperature is used to indicate how much to explore, while the $C_{\texttt{puct}}$ is used to give a weight to the policy with respect to the state value. We used the default values, 1 for the temperature and 5 for the $C_{\texttt{puct}}$. (4) episode max length: the game rules does not mention the maximum moves for a player, and the matches generally don't last more than 300 moves, so the max length was initially 2000, as we will discuss later this value turned out to be a wrong choice.

3.5 Source code project structure

This project required to implement various classes and utility function. Here I will list the most salient part:

- **server.py**: This file contain a http server build on fastAPI that allow to use the model for evaluation of a state and for training. This is required because tensorflow and in general CUDA does not allow different process to use the GPU concurrently.
- **self_play.py**: This file allow to run a self play phase until a timeout occur. At the end each episode train the model with the new data. It allow to use the remote mode (using the http server) or run the model using the CPU.
- **games**: package that contains the game rule and board implemented.

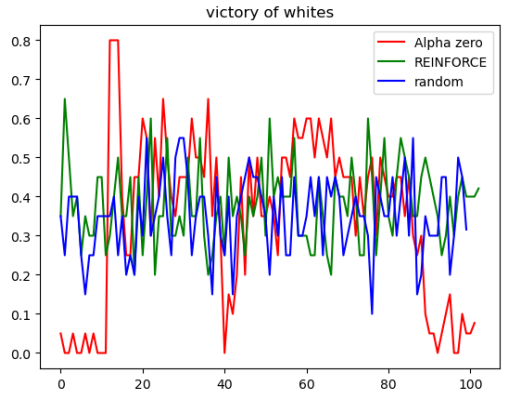
```
Function MCTS(s_0 : State, N : int) is
    N = ∅, Q = ∅, W = ∅, P = ∅;
    p, v = f_θ(s_0) Expand(s_0, p, N, Q, W, P) ;
    for n = 1 to N do
        s = s_0 ;
        while s is not leaf do
            a* = a arg max(Q(s, a) + U(s, a)) ;
            N(s, a*) = N(s, a*) + 1 ;
            s = s → a* ;
        end
        p, v = f_θ(s) ;
        Expand(s, p, N, Q, W, P);
        Backup(s, v, N, Q, W);
    end
    π(a | s_0) = N(s_0, a)^{1/τ} / Σ_b N(s_0, b)^{1/τ} ;
    return π;
end
Function Expand(s_L, p, N, Q, W, P) is
    for a ∈ Action(s_L) do
        N(s_L, a) = 0;
        W(s_L, a) = 0;
        Q(s_L, a) = 0;
        P(s_L, a) = p_a;
    end
end
Function Backup(s_L, v, N, Q, W) is
    s = s_L;
    while parent(s) != NIL do
        s, a = parent_action(s);
        W(s, a) = W(s, a) + v;
        Q(s, a) = W(s,a) / N(s,a);
    end
end
```

Algorithm 1: Monte Carlo Tree search pseudo code

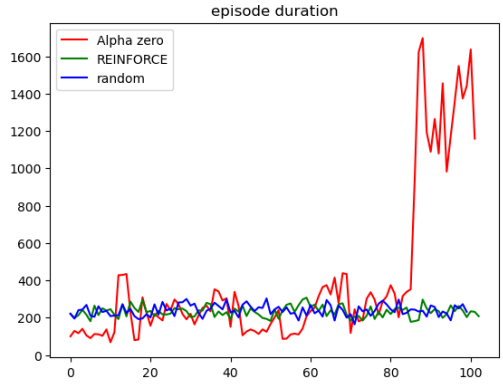
- **games/game.py**: abstract class with the function that a game must implement to be playable.
- **games/board.py**: class that implement a board of a generic game. It just update the internal representation of the board and manage the user turns.
- **games/player.py**: class that implement a player that play a random move
- **games/[tablut/tictactoe]/players/alpha_zero**: folder that contains the player that use Alpha Zero algorithm. It contains 3 main file: `alpha_tablut_zero.py`, `model.py` and `tree.py`. `alpha_tablut_zero.py` is the actual player, here is implemented the MCTS. `tree.py` is the implementation of a node for the state tree. It contains the values for the MCTS and the parent node and the children nodes. `model.py` contains the class that manage the keras model, load it, train it, save and load the previous stored games, evaluate the model with a game state and eventually send the request to the http server.
- **games/[tablut/tictactoe]/players/reinforce**: similarly to the Alpha Zero folder, it contains the implementation for the REINFORCE player.

4 Results and observation

The machine used have as CPU an Intel(R) Core(TM) i7-8750H CPU @ 2.20GHz, the GPU is a GeForce GTX 1060 Mobile. The self play and training lasted for one week, with a requirement of 200W, it consumed 33kwh.



(a) The mean victory for the white player. Is clearly visible that in Alpha Zero you can see various phase during the training: in the first phase the white alternate between winning and loosing momentum, then in the end it most of the times draws. The value is obtained with a sliding window of size 20 episodes result.



(b) Duration of self play matches during training except for the random player. As you can see alpha zero initially remained lower than reinforce but in the end it reached draw most of the times. The value is obtained with a sliding window of size 20 episodes result.

Given the limitations on computation resources and time required for the models convergence, the results for Tablut were not particularly exciting. The two AI players demonstrated slightly better performance compared to the random player while the MinMax agent won all the match he played. Consequently, it can be inferred that in scenarios where an agent with a handcrafted heuristic is feasible, it may outperform more sophisticated approaches. The development process encountered several challenges, primarily stemming from the difficulty of assessing whether the players were learning slowly or not learning at all.

During the training period of one week, the two models engaged in self-play, with the REINFORCE model participating in 43,000 matches and Alpha Zero in 2,000 matches. Since the game did not impose any rules regarding match duration, an initial limit of 2,000 was set. However, after 1,700 matches, Alpha Zero began to encounter numerous draws. Consequently, an investigation was initiated to identify the cause, which revealed that the model had entered into loops. One contributing factor was that the majority of the episodes consisted of loops, resulting in the models being trained predominantly on actions that leads to repeated states. Consequently, the models learned to remain trapped in loops.

This issue can be solved by appropriately setting the hyperparameter for the maximum length of an episode, such as choosing a value of 200. Initially, the problem was addressed by manually devaluing actions that led to repeated states. However, the model learned to maintain a sort of status quo in the game until a player exhausted their good moves and made a bad action that led to a defeat. The main challenge in solving problems is that to see an effect in the performance there is a time delay of many days making the development quite slow.

Still the focus of this project is to show that reinforce can be empowered with a Monte Carlo Tree Search without any model adaptation and in 3 you can see the results of tic tac toe after 3 hours of training and the same model with Alpha Zero MCTS.

5 Discussion

The developing of this project have some good results, as in case of simple games, and some bad results as in case of tablut. The challenges of complex games still pose many problems even when using state-of-the-art algorithm if some hyperparameters are not opportunely tuned. Anyway the approach was a success when applied to simple game.

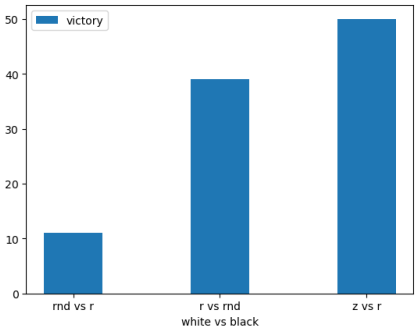


Figure 3: The victory rate for the white player with different players in a simple tournament of 50 matches

References

[1] Andrea Galassi. *An Upper Bound on the Complexity of Tablut*. 2021. arXiv: 2101.11934 [cs.CC].

[2] David Silver et al. "Mastering the game of Go without human knowledge". In: *Nature* 550.7676 (Oct. 2017), pp. 354–359. ISSN: 1476-4687. DOI: 10.1038/nature24270. URL: https://doi.org/10.1038/nature24270.