Game Theory Optimization Based on Python: Evidence from Board Game "Go-Moku"

Zijing Li *

Department of Software Engineering, LUT University, Lahti, Finland

* Corresponding Author Email: Zijing.Li@student.lut.fi

Abstract. Game theory has been widely used in multiple fields such as economics, computer science, and political science to study the rational behavior of multiple decision makers in different scenarios. In recent years, with the rapid development of artificial intelligence, AI trained using game theory has shown outstanding performance in board games such as Go, chess and backgammon. This study explores the application of game theory optimization in the board game "Go-Moku" using a learning algorithm combining Monte Carlo tree search algorithm and reinforcement learning. The paper discusses recent developments in game theory in Go-Moku and explains the Monte Carlo tree search algorithm in detail. The performance of the algorithm is evaluated through experimental results of the application of Alpha Zero in the Go-Moku domain, demonstrating its effectiveness in improving the gaming capabilities of artificial intelligence. According to the analysis, after 1,100 training sessions, the algorithm combining reinforcement learning and MCTS had a 9:1 win rate compared to a pure MCTS approach with 2,000 self-simulations per step. And playing against the 800-times pure MCTS method, it reached a 10:0 win rate for the first time in the 300th game. In addition, the paper discusses potential applications of game-theoretic optimization in other dynamic games. Overall, these results shed light on further exploration of game theory in the area of complete information games and reinforcement learning.

Keywords: Game theory; MCTS; Go-moku; dynamic games; Alpha Zero.

1. Introduction

Game theory, as one of the important branches of mathematics, has received increasing attention from the fields of economics, computer science and psychology since the theory was proposed [1]. Game theory is a kind of game theory used to analyze strategic decisions between rational subjects [2], from the original two-person zero-sum game master key refinement supplement, has several types of cooperative/non-cooperative, complete information and incomplete information games, etc., is widely used in more than the existence of cooperative and competitive relations of behavior, and in the field of games is a huge impact. In areas such as chess, go, backgammon and tic-tac-toe, AI using game theoretic methods have demonstrated powerful decision-making abilities, beating many professional players in the field [3]. Algorithms such as Monte Carlo tree search, and Alpha-Beta pruning are used to evaluate the position of the board and select the best move strategy based on the expected outcome of future moves [4]. It is worth mentioning that one of the developing areas of game theory is the study of behavioral game theory, which attempts to explain the behavior of players in situations where individuals may act irrationally or may be subject to cognitive biases. This has led to a better understanding of human decision making and the development of more accurate models for predicting human behavior. This STUDY focuses on python-based game optimization methods in complete information games.

Browne published a review of the Monte Carlo Tree Search (MCTS) method, discussing its great success in terms of accuracy and generalizability for chess puzzles such as Go [5]. Liu et al. proposed a self-playing Go-Moku algorithm designed using deep reinforcement learning and Monte Carlo tree search to find win-balance points that can effectively achieve tempo balance and reached a world-leading level [4]. Zhang et al. proposed a new neural virtual self-gaming (NFSP) algorithm, which gives new possibilities for solving optimization problems by approximating Nash equilibrium points in large-scale deep search games with the combined Monte Carlo tree search and NFSP algorithms [6]. Srivastava and Yang also used a combination of Monte Carlo tree search and neural networks

when replicating the underlying logic of Alpha-Go in the Go-moku game, which turned out to be a good solution for games in large state spaces [7]. Fu discussed in detail the performance of AI such as AlphaGo in games such as Go, and elaborated that traditional game tree search becomes impractical in Go, as traversing the game tree for the entire game is simply unrealistic and infeasible. [3]. Han and Wang proposed an auxiliary mechanism to address the problem that backward movers in Go-Moku have a low win rate and always receive negative rewards by adding two principles, and this mechanism significantly enhances the training level of the model [8].

Based on a review of the findings of the above scholars, the aim of this thesis is to discuss the development and use of game theory in recent years in games of complete information gaming such as Go and Go. A major development in Go has been the use of Monte Carlo Tree Search (MCTS) combined with Deep Neural Networks (DNNs) to create powerful artificial intelligence, such as AlphaGo and Alpha Zero, which beat AlphaGo Zero based entirely on reinforcement learning algorithms [9]. These programs achieve unprecedented levels of performance and push the boundaries of what is considered possible in the game. Similar techniques have been successfully applied in Go. This chess game has become a training ground for the practice of game theoretical decision making and continues to push it to new heights. Due to the limitations of physical factors, game optimization algorithms such as Monte Carlo tree search are discussed in this paper, using Go-Moku as an example.

2. Data & Method

Monte Carlo Tree Search (MCTS) is a decision-making algorithm for game theory and artificial intelligence. It works by constructing a tree where each node represents a state of the game and the edges represent the possible moves from that state. To use MCTS, first should create a root node with the start state of the game. Then, select the node to explore using TreePolicy, which determines which node to choose based on the game state and the number of times each node is visited. Then, use DefaultPolicy to simulate the game from that node, choosing to move randomly until the game ends. A code instruction is given in Table. 1.

Table 1. Monte Carlo Tree Search Pseudo Code Introduction

MCTS_{s_0} *root* ← new node (s_0) within time budget *v* ← TreePolicy*root*

$reward$ ← DefaultPolicy *v* Backup $v, reward$ Best$Child root, 0$

TreePolicy *v* is non-terminal v is not fully expanded Expand*v* *v* ← Best$Child v$, c_{puct} *v* v

Expand*v* *s* ← random unexplored state from *v* add new child node *v′* to *v* for state *s v′*

$$\text{BestChild}v, c_{\text{v}' \in \text{children(v)}} \left(Q(v') + c\sqrt{\frac{ln\, N(v)}{N(v')}} \right)$$

DefaultPolicy*v* *s* ← state associated with *v* *s* is non-terminal s ← random successor state of s Rewards

Backupv, reward v is not null $N(v) \leftarrow N(v) + 1$ $Q(v) \leftarrow Q(v) + \frac{reward - Q(v)}{N(v)}$

v ← parent of *v*

After the simulation, the statistics of the nodes in the tree are updated using the backup function, which calculates the rewards earned during the simulation and propagates them up the tree. Finally, the best child node of the root is selected using the BestChild function, which selects the node that maximizes a certain formula. If a node has not been fully explored, it is extended by adding a new child node with a random unexplored state. When selecting the best child of a given node, use the BestChild function with the variable "c" to control the amount of exploration and exploitation.

The main part of the model and code source code of this paper comes from github [10], which is used for the reproduction of the AlphaGo Zero algorithm and the training of the AI model, and based on this, the subsequent adjustment and optimization of the network layer and the adjustment of the MCTS self-simulation. As training in Go is much more complex than in Go-Moku, this paper chooses

Go-Moku as the main object of study for the discussion of the principles and the introduction of the optimization scheme. AlphaGo Zero is a deep learning-based Go AI from DeepMind, involving three main techniques and gaming methods: self-gaming, Monte Carlo Tree Search (MCTS) and neural networks. The following will note the principles of and relationships between these three. Unlike AlphaGo, AlphaGo Zero's training process is entirely based on self-gaming, i.e., without the supervision and guidance of human Go experts, and generates a large amount of game board position data through self-gaming, which is used as training data for the neural network. As shown in Fig. 1, the MCTS used by AlphaGo Zero can be divided into four steps: selection, extension, simulation and back-propagation, and the search strategy is continuously optimized through an iterative process to find the optimal solution within a specific time and limit [11]. It has similarities to the process of self-gaming, where the situation and possible moves are evaluated by simulating the game several times. Unlike traditional MCTS, AlphaGo Zero utilizes a neural network to aid the search process. The neural network provides a probability distribution of the position of the fallen discs as well as an assessment of the situation, thus improving the efficiency and accuracy of the search.

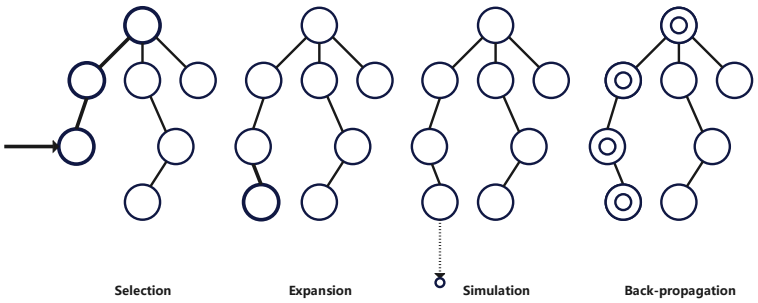

Fig 1. MCTS steps.

The neural network used by AlphaGo Zero is a deep residual convolutional neural network, which has multiple layers, each containing a set of convolutional layers, to extract local features on the game board. A binary feature plane representation of the backgammon board, representing black and white discs respectively, is used as input to capture all the information in the game state, with the strategy network and value network as output, representing the probability of taking a move and the prediction of the win rate for the ruling side respectively. The strategy and value predictions are progressively improved by playing against the previous version of oneself using the self-game.

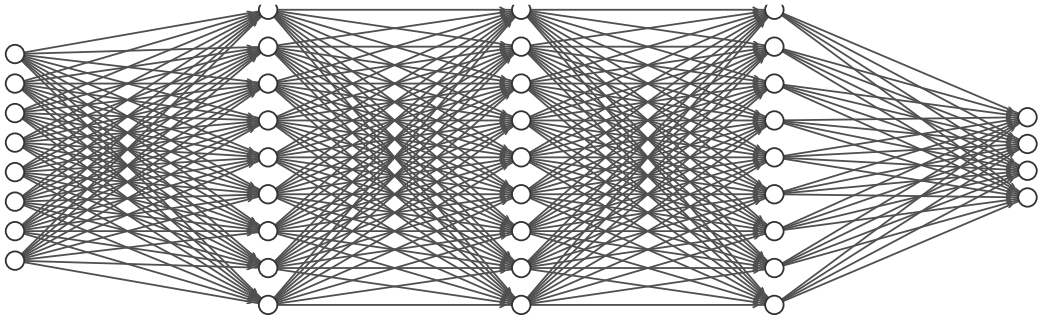

Fig 2. Schematic of a deep neural network

As shown in Fig.2 [7]. The input layer is a four-channel checkerboard state, and the neural network contains three convolutional layers with 32, 64 and 128 channels, where each convolutional layer contains a Batch Normalization layer for accelerated training. As an optimization, a fourth layer is added on top of the three convolutional layers to increase the depth of the neural network and improve the nonlinear feature extraction capability of the network. The convolutional layers are connected to two separate branches, one for predicting the strategy and the other for estimating the value. The strategy branch contains one convolutional layer and two fully connected layers. The value branch contains one convolutional layer and two fully connected layers. Because Go-Moku is a discrete strategy game, the output of the strategy network is a probability distribution representing the

probability of each landing position. The output of the value network is the evaluation value of the current chess game, which represents an estimate of the win rate of the current player. Overall, the structure of this neural network is relatively simple, containing two output layers for outputting policy vectors and state values, while the hidden layer uses Batch Normalization and ReLU activation functions to accelerate training and improve the robustness of the model.

3. Results & Discussion

3.1. Demo for Solution

Due to the limitations of computer performance and practical factors, this paper combines the training model and code of researchers in GitHub to properly adjust the size of the neural network and the chessboard, selects a four-layer convolutional network and a 10 X 10 chessboard, performs batch normalization processing, increases the number of channels in the convolutional layer, and other operations to optimize the calculation process and improve the training speed. Alpha Zero is a generalization of the AlphaGo Zero method to other chess games, and this section will deal with two AI models, pure MCTS and MCTS methods combined with neural network training, respectively. This method does not rely on a strategy value network and is relatively simple to implement, searching and evaluating through many random simulation methods. At the start of the run, a player object is created, an MCTS instance is initialized, then a root node is initialized and the strategy function, exploration parameters and number of playouts are passed in.

The current board action is obtained via the member method, and the specified number of playouts is executed, starting from the root node, and searching down the tree until a leaf node is found. The best move is then selected based on the sum of the node value Q and the a priori score u adjusted by the access count. If the search arrives at a leaf node, the node is extended by an extension method that calculates the probability of the action and creates a new child node based on the policy function. For the leaf node reached a simulation is performed and the game is completed globally according to the random strategy, returning the final win/loss result as a value of [-1,1], +1 (current player wins), -1 (opponent wins) or 0 (tie). The values of the leaf nodes are propagated backwards up the tree, updating the node value Q of each ancestor node on the path as well as the number of visits. After completing the specified number of playouts, a decision is made based on the number of visits to the node, executing the action and updating the root node of the tree to the corresponding child node. Subsequently, the steps are repeated until the game ends.

In another approach using a combination of MCTS and a strategy value network, with the intervention of a strategy value network function using the current board states as input, the probability p of each feasible action and the state value v are output. By using the (s, π, z) data recorded during the self-game as the training set, the strategy is made to work by continuously updating the parameters in the neural network so that probability vector p and the MCTS search probability to maximize the similarity as much as possible, while narrowing the error between the prediction and the game outcome z obtained from the self-game [9].

The following are the dueling wins and losses of the two methods for a playout of 2000. The model starts to show a better winning rate when the self-play batch reaches 150, after which a new optimal strategy emerges when the number of batches reaches 200 and 300. As batch amount increases, the average number of steps per self-play and the learning rate of the model remain stable, and the values of the loss function and the entropy of the strategy distribution show an overall decreasing trend, indicating a continuous improvement in performance. When the self-play batch reaches 550, the model again emerges as the best new strategy with a win rate of 90%. Subsequently, the performance of the model fluctuates widely, but it is still able to maintain a high win rate overall. Fig. 3 shows the change of entropy value in the 1100 bureau and the change of loss function value with batch amount. Fig. 4 presents the variation of win rate with batch.

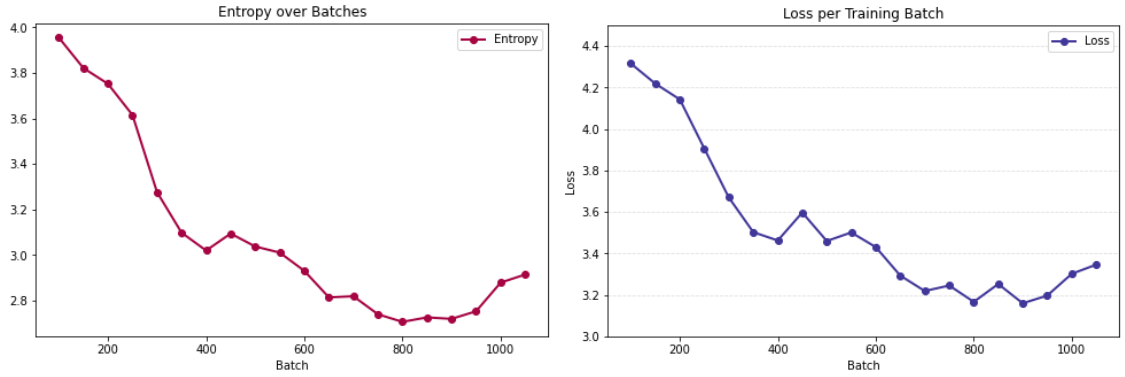

Fig 3. Entropy and loss value.

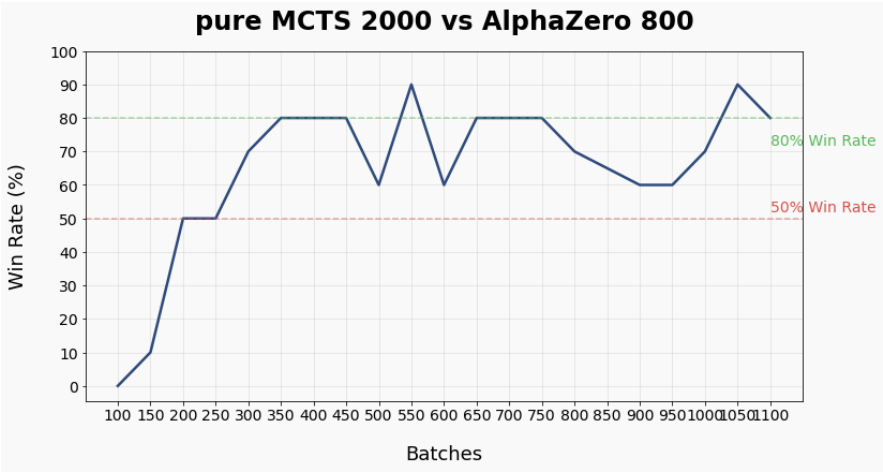

Fig 4. Win rate.

3.2. Optimization Analysis

According to the data comparison of experimental results, as each step of pure MCTS self-simulation increases, it is more difficult for the AI based on Alpha Zero method to win under the same number of innings of training. With a playout of 400 times, the first time there is 10:0 at 300 batches, and only after the number of playouts increases to 2000, there is a 9:1 win rate at 550 batches. It is worth noting that when the convolutional layers are increased beyond 4 layers, the AI's win rate starts to decrease and the entropy value fluctuates in a higher range of values, probably because the depth is too high and an overfitting situation occurs, causing the neural network to be unable to better understand unseen game situations and features. The specific parameters are shown in Table.2.

Table 2. Main parameter value

parameter	value
Board width	10
Board height	10
The number of consecutive pieces	5
learning rate	0.001
KL divergence adaptively adjusts the parameters of learning rate	1.5
training data buffer size	20000
The size of each small batch when training	1024
Target KL divergence	0.02
Check frequency of the model	50
self-play batch amount	1100
Playout of Alpha Zero AI	800
Coefficient of L2 regularization	0.0001
Pure MCTS playout amount	400/800/2000

Based on the analysis of the data above, pure MCTS does not rely on any neural network, which also means that it does not require a lot of data training and computation, and is suitable for simple and smaller search problems, saving costs such as time and computation, but the correctness rate in fives is quite poor. Alpha Zero's MCTS using a strategic value network can perform self-gaming reinforcement learning unmanned, requiring long and massive amounts of training sessions, and as the number of training games increases and accumulates, Alpha Zero will almost never lose against a pure MCTS. Additionally, there is a positive correlation between the number of simulations in each step of the training model and the win rate.

3.3. Suggestion

In Go, AlphaGo, using supervised learning guided by Go experts, lost to AlphaGo Zero, which was based entirely on self-gaming, but it is not strongly proven that this is also true in backgammon, and starting from scratch will certainly result in significant computational and time costs. This may help to improve performance and learning speed, for example by expanding the branches of the MCTS tree using the common patterns in futsal games first, or by using heuristic evaluation functions to improve the accuracy of local evaluations. In addition, the symmetry and rotation of the board can be exploited to reduce the number of training sessions and improve efficiency.

4. Limitations & Propects

Considering the limitations of the author's computer CPU performance and other physical factors, the author was unable to conduct extensive neural network training. Consequently, the analysis relies on case studies of previous research results and corresponding codes to draw conclusions. This approach leads to potential biases in the experimental data and quality, resulting in a decrease in reliability. The size of the chessboard is also limited to 10X10, making it impossible to train on larger planes. Owing to the lack of experimental data sources, the experiments did not compare the self-play algorithm in the Alpha Zero method against a supervised learning method with expert supervision. This absence of comparison deprived the study of a reference target to visually analyze the performance, training costs, and win rate comparisons of the models at the data level. The data and models currently available are limited for various reasons to less than 3000 runs. As a result, comparisons between more fully trained models and pure MCTS methods remain at a more latent level. For future in-depth research, adaptive methods or hyperparameter optimization techniques will be used to restrict the combination of parameters, add experimental controls, and generate updated experimental results and data to compare with the previously concluded data. Efforts will also be made to increase the complexity of the neural network structure and the number of network layers, improving the model's adaptability to complex games. Moreover, more consideration is given to using restrictions such as regularization to reduce the parameters of the model and reduce the risk of overfitting. When equipment and other conditions permit, researchers will strive to increase the training time and the number of models as much as possible. This approach will allow for a more diverse range of comparisons and references to pure MCTS methods. Besides, researchers can explore novel data augmentation techniques to enhance the model's generalization capabilities further. The integration of advanced reinforcement learning algorithms and techniques could also improve the Alpha Zero method's performance in the game of Go-moku. Additionally, investigating the impact of different reward functions or exploration strategies could provide valuable insights into the model's performance and behavior. In the next in-depth study, some fusion of supervised learning can also be tried, using the master's games as part of the training set, which can be used to improve the training speed. To sum up, while the current implementation of the Alpha Zero method in Go-moku has some limitations, there is significant potential for future development and improvement. By addressing these limitations and exploring novel approaches, researchers can push the boundaries of the Alpha Zero method and its applications in Go-moku and other complex games, further contributing to the advancements in the field of artificial intelligence and game-playing algorithms.

5. Conclusion

In summary, this study discusses the potential of self-gaming in mastering complex games as demonstrated by the application of methods combining Monte Carlo Tree Search (MCTS) and neural network reinforcement learning in the game of Go-Moku. MCTS is used to explore game trees and to make optimal decisions in a competitive environment using the principles of game theory. Despite limitations such as shallow neural network structures, biased experimental data and a lack of comprehensive comparisons with other methods, the article illustrates the optimization and promise of the Alpha Zero approach to decision making in Go-moku. To continue to refine the experimental process, and optimize the model, future research directions should consider deeper and more advanced neural network structures, more suitable parameter settings as well as more efficient computational performance and smaller unit running costs. Through continued research and improvement of game optimization and Alpha Zero methods in pentomino, researchers will update the understanding of MCTS and game theory and its use in chess games and beyond.

References

[1] Shunhua T, Miao C. Search Algorithm in the Five-piece Chess. Journal of Emerging Trends in Computing and Information Sciences, 2012, 3: 4.

[2] Myerson R B. Game theory: analysis of conflict[M]. Harvard university press, 1997.

[3] Fu M C, MONTE CARLO TREE SEARCH: A TUTORIAL 2018 Winter Simulation Conference (WSC), Gothenburg, Sweden, 2018: 222-236.

[4] Liu P, Zhou J, Lv J. Exploring the first-move balance point of Go-Moku based on reinforcement learning and Monte Carlo tree search, Knowledge-Based Systems, 2023, 261: 110207.

[5] Browne C B, Powley E, Whitehouse D, et al. A survey of monte carlo tree search methods. IEEE Transactions on Computational Intelligence and AI in games, 2012, 4(1): 1-43.

[6] Zhang L, Chen Y, Wang W, et al. A Monte Carlo Neural Fictitious Self-Play approach to approximate Nash Equilibrium in imperfect-information dynamic games. Frontiers of Computer Science, 2021, 15: 1-14.

[7] Srivastava A, Yang L. A ReImplementation of AlphaGo-Zero on a game of Gomoku. Association for the Advancement of Artificial Intelligence, 2020.

[8] Han H, Wang X. Ancillary mechanism for autonomous decision-making process in asymmetric confrontation: a view from Gomoku, Journal of Experimental & Theoretical Artificial Intelligence. 2022.

[9] Silver D, Schrittwieser J, Simonyan K et al. Mastering the game of Go without human knowledge. Nature, 2017, 550: 354–359.

[10] Song J. AlphaZero-Gomoku. Retrieved from: https://github.com/junxiaosong/AlphaZero_Gomoku.git. 2018.

[11] Chaslot G, Bakkes S, Szita I, Spronck P. Monte-Carlo Tree Search: A New Framework for Game AI. Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment, 2021, 4(1): 216-217.