**Gomoku with AlphaZero**

E6885.2024Fall.Final Project.report.zl3477.jm5876

*Columbia University*

**Abstract**

*An abstract is written in a brief and factual form in one paragraph. It provides a clear description of the project, including the goal and objective(s), the motivation and the significance of the project. It describes the main technical challenge(s), and how the authors overcame the challenges. The abstract shows the key final result.*

*(For student projects: (i) the abstract need to state what the project does; (ii) the abstract needs to state if the original goal was accomplished; (iii) If the results did not meet the initial goals, that needs to be stated and reasoned.*

**1. Introduction**

In this project we use a reduced version of the AlphaZero algorithm for the game of Gomoku on a 6x6 board. AlphaZero —is a reinforcement learning algorithm that uses Monte Carlo Tree Search (MCTS) in combination with a deep Policy-Value Network to get superhuman results in a variety of games. Our goal is to develop an optimal implementation for Gomoku based on its success in Go, Chess and Shogi. Gomoku, or Five in a Row, is a two-player grid strategy game in which two people stand on a line, alternating black and white pieces. The game’s aim is to get five consecutive stones on the same line (vertically, horizontally or diagonally). The game is easy to play, but it is an especially difficult game for the AI due to the huge search space and optimizations needed to explore and exploit it. This simplified version of AlphaZero incorporates the following components:

* Policy-Value Network: A TensorFlow trained neural network that determines both the probability distribution for legal moves (policy) and the expected result of the game (value) for a given board state.
* Monte Carlo Tree Search (MCTS): A better tree search algorithm that employs network predictions to optimize the search, weighing exploration versus exploitation.
* Self-Play Mechanism: The AI improves by playing against itself, generating high-quality training data for the neural network.
* Pure MCTS for Baseline Comparison: To evaluate the effectiveness of AlphaZero, we compare its performance against a pure MCTS implementation that does not rely on neural network predictions.

Self-play and practice with repetition ensure that the model gradually learns to play Gomoku competitively. We experiment with playing multiple games using AlphaZero and pure MCTS to analyze performance, win rates, and improvement over training. The main goal of this project is to decipher and use the AlphaZero pipeline for a simple board game, learn new distribution algorithms (Dirichlet noise) to optimize MCTS exploration and explore how AlphaZero-Ai works better than traditional search methods. This paper explains the theoretical background, methodology, experiments and findings of the project and how AlphaZero can be used to solve games such as Gomoku.

**2. Background and Novel Contribution**

**2.1 Background and Related Work**

Reinforcement Learning has made many significant advances in solving large-scale decision problems — notably in the board games field. One of the most important RL experiments was DeepMind’s AlphaGo, which employed MCTS and deep neural networks to perform superhuman Go play [1]. It has since been expanded into AlphaZero, a computer that learns to play board games such as Chess, Shogi and Go autonomously, without needing any human instruction.

AlphaZero was adapted in many ways to other games, including Gomoku, a grid-based strategy game. AlphaZero is extremely fast, but existing implementations are usually highly computationally and resource-consuming. For example, a previous Gomoku AlphaZero implementation includes detailed parallelized MCTS, extensive replay buffers, and large-scale self-play for improved generalization [2]. And these projects typically feature large codebases, complex optimization pipelines, and extensive experimentations to fine-tune performance, making them less accessible for quick prototyping [3].

**2.2 Novel Contribution**

Our project simplifies and implements the AlphaZero algorithm for Gomoku on a smaller 6×6 grid, with the following key contributions:

1. We’re working on a smaller, modular AlphaZero implementation in contrast to other massive implementations. Its codebase keeps the core modules (MCTS, Policy-Value Network, and Self-Play Training) intact and removes unnecessary complexity like parallelization or replay buffers.
2. We incorporate Dirichlet noise at the root node of MCTS during self-play, enhancing exploration and ensuring the algorithm does not overly exploit early, suboptimal strategies.
3. We implemented a Pure MCTS baseline for performance evaluation. This allows us to compare AlphaZero’s learned policy with a traditional search-based approach and measure its effectiveness in decision-making.

As opposed to the massive AlphaZero-based Gomoku projects, our implementation is lightweight and easy to use for prototyping and learning. Moreover, we focus on the main algorithm while achieving high performance using small training rounds.

**3. Approach**

Gomoku is a two player turn-based game, played over a grid. It aims to place five identical stones vertically, horizontally, or diagonally. The state space expands exponentially with board size, so finding all possible moves by brute-force search is computationally impractical. This drives the adoption of AlphaZero, a machine learning-driven deep learning platform whose MCTS algorithm leverages the efficient use of self-play to train the best strategies.

**3.1. Algorithm Development**

The neural network predicts two outputs given a board state. First one is policy, a probability distribution over all possible legal moves. Second one is value, a scalar estimation of the probability of the current player winning the game. We designed a convolutional neural network using TensorFlow, structured as input, hidden layers and output layers. Input is a 6×6×2 tensor representing the board state (one channel for each player’s stones). Hidden layers are two convolutional layers with ReLU activation and a flattening layer. For the output layers, first one is about a dense layer with a SoftMax activation producing move probabilities. And then A dense layer with a Tanh activation outputting the win probability. The network is optimized using a combined loss function called network loss function shown below [4].

(1)

MCTS serves as the decision-making module, leveraging the neural network predictions to guide the search. At each decision point, the algorithm balances exploration and exploitation. For selection part, traverse the tree using the Upper Confidence Bound for Trees formula is [5]:

(2)

We also use Dirichlet Noise to improve exploration during self-play. This ensures the algorithm explores a broader range of moves early in training. Moreover, the model is trained using self-play to generate game trajectories, which means the agent plays against itself, using MCTS guided by the current policy network and each move generates a training sample. After this, the network is updated by minimizing the combined loss using collected samples and updated network is used in subsequent self-play games.

**3.2.** **Theoretic Results**

The AlphaZero algorithm leverages the theoretical strengths of MCTS and deep learning:

1. MCTS Convergence: MCTS, in combination with reliable value estimation, approximates the game’s minimax value.
2. Policing Enhancement: By combining the network’s predictions with MCTS-enabled policies, the algorithm optimizes the policy over time towards optimal play.
3. Self-Play Convergence: Self-play in MCTS ensures that the agent can learn from stronger opponents (themselves) in order to obtain a stable policy.

**3.3. Implementation Simplifications**

To make the project lightweight and efficient:

* We use a smaller 6×6 board to reduce computational complexity.
* The MCTS implementation is single-threaded, focusing on simplicity.
* Training is limited to 100 iterations for quick prototyping.
* Evaluation is performed against a Pure MCTS baseline, which does not require neural network predictions.

**4. Experiment results**

Describe the organization of this section, then provide detailed discussion about your implementation.

**4.1 Data**

Details about dataset(s)

**4.1 Deep Learning Network**

Provide the following descriptions and discussions:

1. Architectural block diagram(s).
2. Training algorithm details
3. Flowchart(s)
4. Data used

**4.2 Software Design**

Provide the description and discussion:

1. Flow chart(s) - provide one top level flow chart, then additional flow charts for detailed lower level implementations,
2. Algorithm, e.g., description of the step by step implementation,
3. Pseudo code for each section of the implementation,
4. Links to code in your project github need to be included through the sections above.

**5. Conclusion**

**5.1 Project Results**

Detailed:

1. Description of results,
2. Figures, plots,
3. Testing, verification.

**5.2 Comparison of the Results Between the Original Paper and Students’ Project**

Provide detailed comparison between the results of the original paper and the students’ project:

1. Figure, plots showing differences in things such as training length/time, training/verification error, test error, other.

**5.3 Discussion / Insights Gained**

Provide detailed discussion, regardless of the actual results. Why are your results different from the results of the original paper. Examples: did you use a smaller dataset, did you use different hyper-parameters, number of epochs, ...).

**6. References**

TODO

**7. Contributions of each member of the team**

|  |  |  |
| --- | --- | --- |
|  | UNI1 | UNI2 |
| Last Name |  |  |
| Fraction of (useful) total contribution | 1/3 | 1/3 |
| What I did 1 |  |  |
| What I did 2 |  |  |
| What I did 3 |  |  |