Design and Implementation of GomokuAI Program

Chen Wei

Abstract

With the traditional reinforcement learning method, it is difficult to solve complex tasks with a large number of action sets and state sets. In recent years, a new reinforcement learning method based on deep learning becomes increasingly popular. Deep reinforcement learning combining reinforcement learning with neural networks provide solutions to many difficult problems that tend to have high-dimensional tasks. For example, AlphaGo, developed by DeepMind a few years ago, was the first program ever to beat professional players. This amazing performance has made many enterprises and researchers try to apply deep reinforcement learning in different industries. This paper presents the algorithm design in reinforcement learning and the network training process in deep learning through making a program reproducing AlphaGo Zero [1] but adjusting the algorithm based on the characteristics of Gomoku.

Keywords: Machine learning, Deep learning, Reinforcement Learning, Deep Reinforcement Learning, Neural Network, Self-play, Monte Carlo Tree Search, AlphaGo Zero

1. INTRODUCTION AND LITERATURE REVIEW

1.1 Research Background

In recent years, artificial intelligence technology has been developing rapidly and becoming increasingly popular in China. Deep reinforcement learning (deep RL), as a subfield of AI, has been applied in many domains such as healthcare, robotics, industrial automation, advertising, education, finance, and many more [2]. In addition, as a vital direction of China's "New Infrastructure" policy, AI will require more researchers to participate in the research of related technologies in the future. There is an urgent need for more advanced AI technologies and methods to bring about better changes in entertainment life and health, especially in the gaming and medical industries.

1.2 Research Content

AlphaGo Zero is a computer program based on the game of Go that uses a Monte Carlo tree search algorithm with a neural network to play Go. It does not require any data from human experts but rather plays Go against itself to improve its intelligence from scratch and surpasses all its previous versions in just 3 days. Inspired by AlphaGo Zero, in the paper, the implementation of GomokuAI also combines reinforcement learning and deep learning approaches.

Gomoku (also called Gobang or Renju) is a two-player board game similar to Go, which normally plays on a 15x15 board. The first player who gets five consecutive stones horizontally, vertically, and diagonally wins the game.

Two-player board games, especially those with small game trees such as chess or tic-tac-toe, use Minimax Algorithm [3] with Alpha-beta pruning [4] in the past to make decisions for selecting moves. However, for board games with large search trees, such as Go, the Monte Carlo tree search algorithm (MCTS) with deep learning approaches tend to have more advantages than the other two algorithms.

MCTS is a form of the Monte Carlo method applied to the game tree search [5]. It is a heuristic search algorithm, named by Rémi Coulom in 2006 [6], which is mainly used for zero-sum board games with complete information but can also be used for games with imperfect information such as bridge and poker. Therefore, I chose MCTS as the basis for making the decisions to place every stone.

My program, GomokuAI, uses MCTS to optimize the parameters of the neural network, which in turn guides the MCTS search process. At the end of each search process, a stone is placed until the game end and the data for that game will be generated. After a period of time, a large amount of data will be fed into the neural network for training, and the trained neural network will allow MCTS to make a more reasonable choice for its next move. Pytorch (a machine learning library) is used for its ease of use, debugging and relatively short learning curve. Also, I created a GUI based on Tkinter, because it is more useful for analysis and testing. To facilitate adjustment of parameters and hyperparameters, Tensorboard is used to track and visualize metrics such as loss and accuracy. Finally, due to the limitations of my hardware environment, GomokuAI only trained on a 9x9 board and got a powerful AI.

1.3 Research Objectives

Deep reinforcement learning (deep RL) has been applied with great success in the games industry. Game developers can use deep RL to help them better design their games, such as rules and patterns. Besides, deep RL can help gamers or human experts to better understand the game they play. MCTS in combination with neural networks is an important component for the success of AlphaGo Zero. This paper aims to apply such methods to Gomoku and to investigate more possibilities and challenges regarding these methods.

1. SYSTEM ANALYSIS

2.1 Program Functions

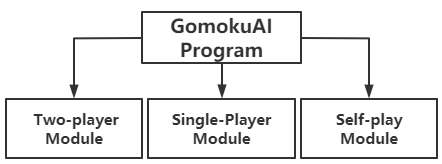


Figure 1. Functions of GomokuAI Program

1. Two-player Module: Two human players play against each other.
2. Single-Player Module: A human player play against GomokuAI.
3. Self-play Module: Using MCTS with a deep neural net to play against itself to improve the strength of GomokuAI.

2.2 Dictionary Structure

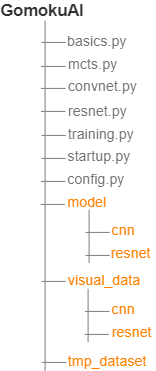


Figure 2. Directory Structure of Program

1. basics.py

The basics.py module defines the basic functions of the game, such as the size of the board, the number of stones to win, rules of the game, and GUI.

1. mcts.py

The mcts.py module implements the Monte Carlo tree search algorithm.

1. convnet.py

The convnet.py module is a CNN model written in Pytorch with a shallow network layer. This module is used for comparison with the performance of the resnet.py module.

1. resnet.py

The resnet.py module is a residual neural network model written in Pytorch. The number of convolutional kernels and residual blocks can be freely defined. The filter size is 3x3 and the stride and padding are both 1, plus the BatchNorm [7] layer.

1. training.py

The training.py module trains the neural network. It collects the game data by calling the self-play method, transforms the game data, and feeds it into the neural network for training.

1. config.py

The config.py module defines the hyper-parameters used for training and the parameter values used to adjust the algorithms.

1. startup.py

The startup.py module is used to start the game interface.

1. visual\_data

The visual\_data folder is used to store the data files that visualize the training process.

1. tmp\_dataset

The tmp\_dataset folder is used to store the game data from self-play.

1. model

The model folder is used to store the model files after training.

2.3 Model Comparison

|  |  |  |
| --- | --- | --- |
| Parameter | GomokuAI | AlphaGo Zero |
| Number of searches/playouts (MCTS) | 600 | 1600 |
| Learning Rate | 0.002-0.001 | Annealed learning rates |
| Alpha (Dirichlet noise) | 0.3-0.15 | 0.03 |
| Weight (Dirichlet noise) | 0.25 | 0.25 |
| First n moves of each game | (n=6-14,8x8 board), (n=10-18,9x9 board) | (n=30,19x19 board) |
| Buffer size | 10,000-50,000 states | 500,000 games |
| Batch size | 512 states | 2048 states |
| Optimizer | Adam | SGD with momentum |
| Number of blocks in Resnet | 4 | 19 & 39 |
| C\_PUCT | 5 | 5 |
| Neural network type | CNN & ResNet | ResNet |

Table 1. Model Comparison

1. IMPLEMENTATION OF ALGORITHM

3.1 Monte Carlo tree search

In this program, the implementation of the MCTS algorithm consists of three steps, but with a few changes according to the Gomoku rule.

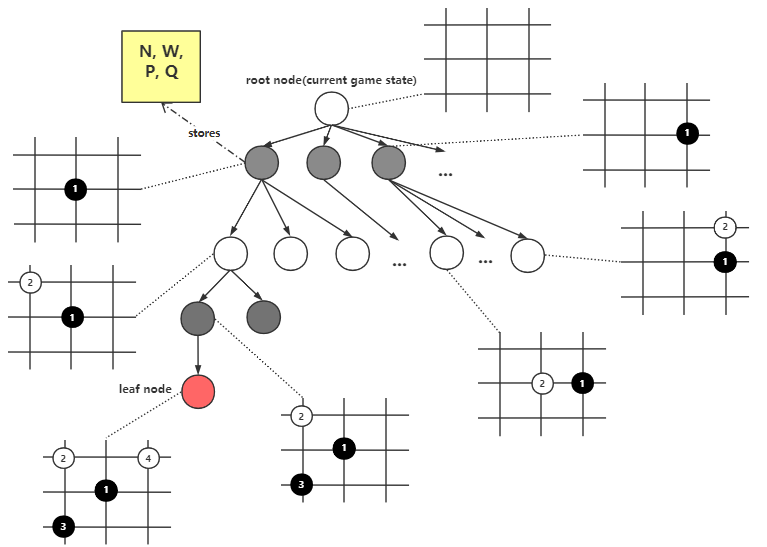


Figure 3. Selection Step

1. Selection: The root node represents the current position of the game (as shown in Figure 3). The process of this step is to select from the root node until the leaf node is reached, and select nodes that maximize the value of upper confidence bounds for trees plus the value of Q. In my program, each node in the search tree stores the information of each position of the game, and points to its parent and child nodes, except for the root and leaf nodes.

N stands for the number of times the node is visited (also called visit count).

P stands for the prior probability of selecting this node.

Q stands for the mean action-value.

W stands for the total action-value.

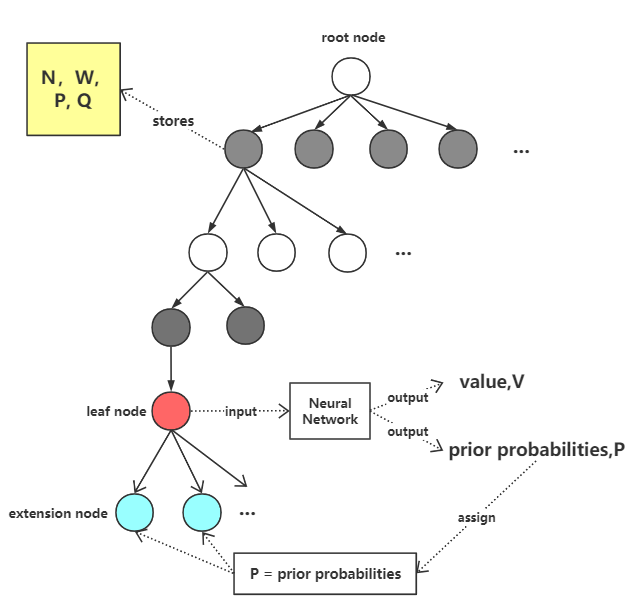


Figure 4. Expansion and Simulation Step

1. Expansion and Simulation: When a leaf node is reached, more (or one) child node are created unless the game has ended. The leaf node is expanded and then evaluated by a neural network to produce both prior probabilities and the value of the position. These extended nodes become the new leaf nodes in the tree and each new leaf node stores the prior probability P produced by the neural network and points to the previous leaf node as its parent node, with other information initialized to 0 or None (as shown in Figure 4).

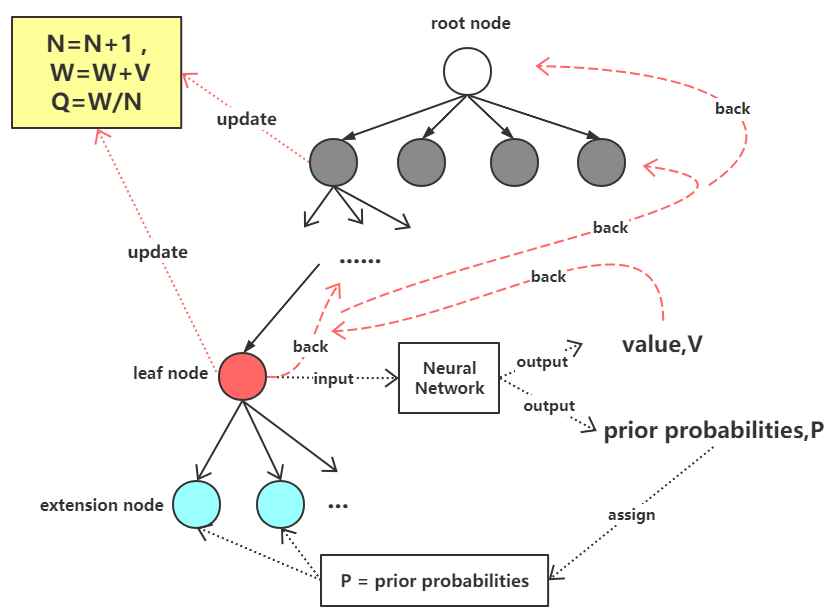


Figure 5. Backpropagation Step

1. Backpropagation: MCTS uses the result of expansion and simulation to update information stored in the node from leaf node to root node (as shown in Figure 5), where the Q of each node is derived from the number of visits to that node and the score V given by the neural network.

3.2 Self-play

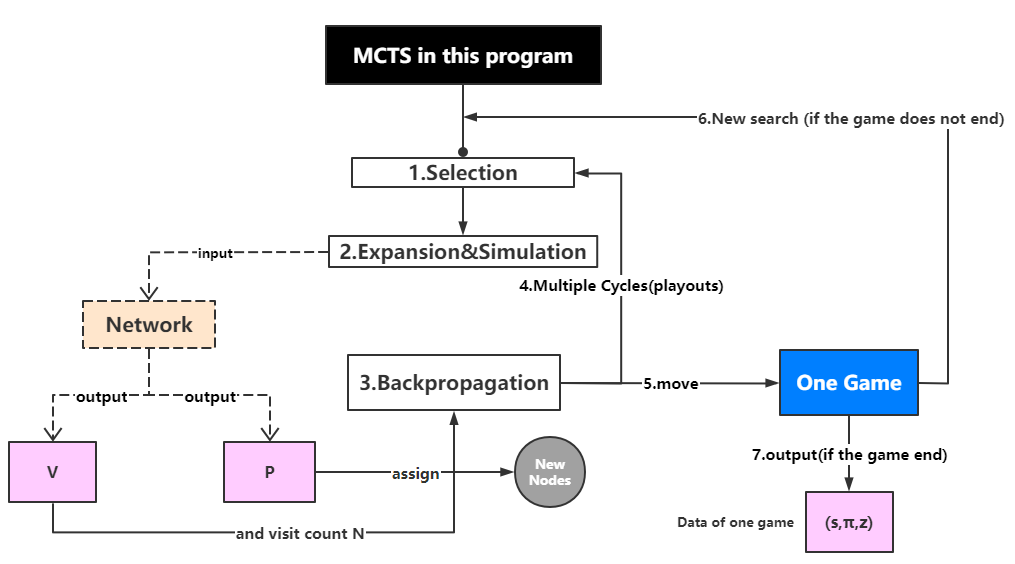


Figure 6. Process of Self-Play

Figure 6 shows the complete process of this program using MCTS and neural networks to play against itself and generate the data of a game (s, π, z). S, π and Z represent the set of all positions, the search probabilities, and the self-play winner in a game.

Considering the rule and the game tree of Gomoku, the number of searches (also called playout) of MCTS are generally defined as 400-600 in my program. I also augmented the game data by rotating and flipping each position [8].

3.3 Other Algorithms and Parameters

In the process of GomokuAI self-play, MCTS should both consider all possibilities and discard unreliable moves in order to prevent repetitive moves and to avoid ignoring valuable positions. The classic problem of reinforcement learning, Multi-armed Bandits [9], shows a dilemma: should we explore more to try new possibilities, or should we choose to make use of the best options known so far?

The application of MCTS to GomokuAI must also consider this dilemma.

3.3.1 Upper confidence bounds applied to Trees (UCT)

L. Kocsis and Cs. Szepesvári proposed UCT in 2006 [10], an algorithm that guides MCTS in selecting nodes in the search process of this program, which is calculated by

U (s, a) = Cpuct \* P (s, a) \*

Cpuct is a coefficient that determines the level of exploration, s represents the current game position, a is one of the nodes at a layer in the tree, b is the parent node of a, and P (s, a) is the prior probability of node a.

In the training process, Cpuct is initially defined as 5, sometimes changed to 4 to improve the accuracy of the decision.

3.3.2 Temperature parameter

The temperature parameter τ is used to control the level of exploration and exploitation of MCTS. After several MCTS searches, GomokuAI places a stone according to the following equation.

π (a | s) =

For a 9x9 board, I set τ to 1 in the first 1000 games to add more move options and to increase the diversity of the game data. After 1000 games, τ is set to 1 for the first 6 to 14 steps of each game, and then to 0.0001 starting at step 15. The reason for this was that I wanted GomokuAI to look for different possibilities in the early stages of the game and to adopt to a conservative style in the later stages of the game.

3.3.3 Dirichlet Noise

AlphaGo Zero adds Dirichlet noise to the prior probability P of the node expanded in the expansion steps to ensure that all positions are taken into account. In this thesis, however, better results are obtained by adding Dirichlet noise to the stage which places a stone at the end of the MCTS search. It is calculated as:

P (s, a) = (1 - ε)\* π (a | s) + εηa

where ε = 0.25 and η = 0.1 to 0.3.

1. PROCESS OF TRAINING

4.1 Model selection

1. CNN

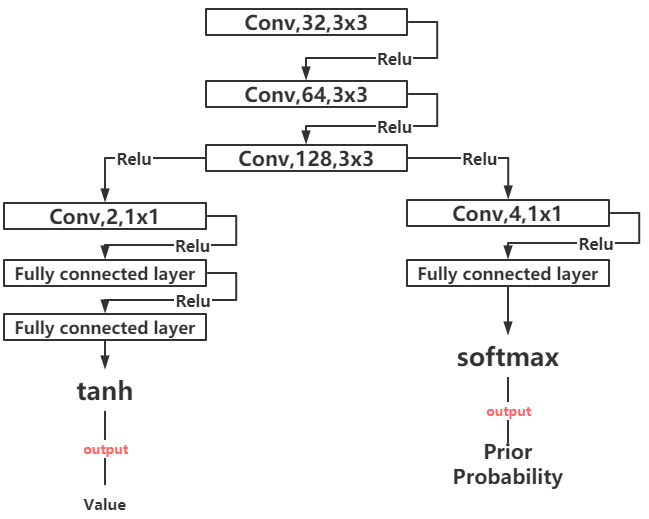


Figure 7. Structure of CNN

Convolutional neural networks [11] are often used to process time-series data or image data. Gobang also has the characteristics of the grid structure. Therefore, my program uses CNN to train and takes the training results as a reference. (as shown in Figure 7)

1. ResNet

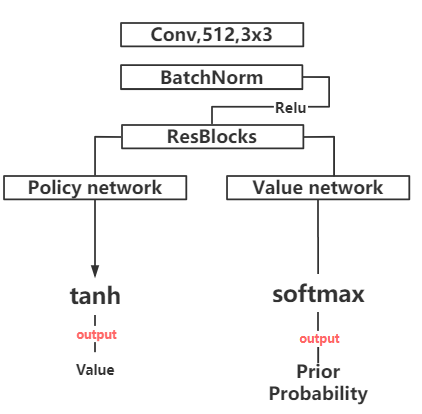


Figure 8. Structure of ResNet

Residual Networks (ResNet) was proposed by Dr. Kaiming He's team at Microsoft Research in 2015 and won the first place in the 2015 ILSVRC classification task [12]. By adding shortcut connections, it allows deeper neural networks to be optimized more easily.

The structure of the neural network in this program uses the approach of Alpha Zero [13]: the policy and value network are combined into one, with one end outputting the probability and the other end outputting the value of a position. In addition, considering the limitation of hardware and time, only 1 to 4 ResBlocks are used in this program. In this paper, due to the simplicity of the residual network construction used by GomokuAI, using a residual network does not achieve better performance than ordinary convolutional neural networks.

4.2 Neural Network Training

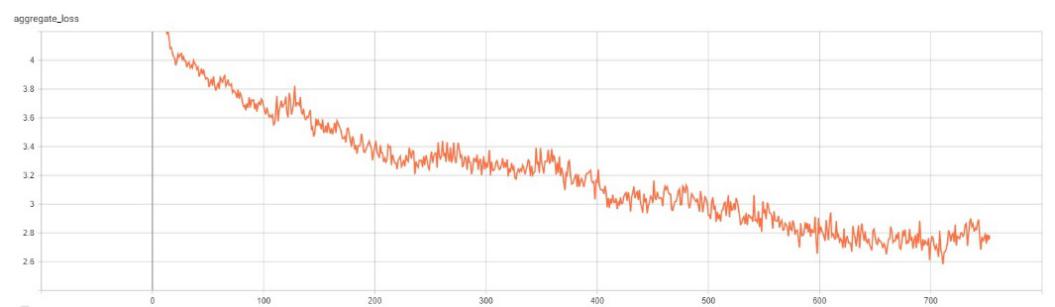


Figure 9. Training Loss

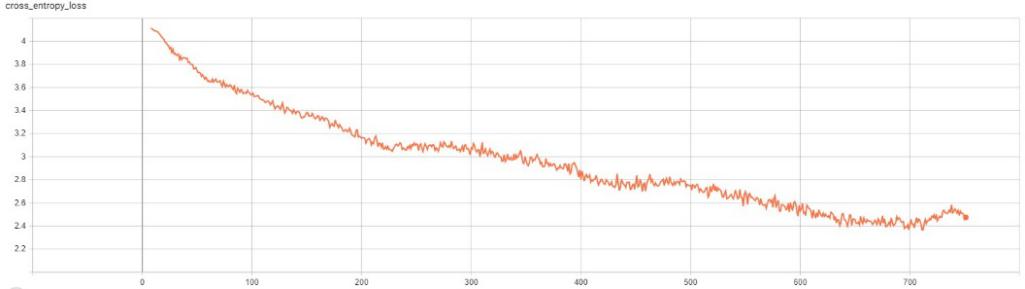


Figure 10. Cross-Entropy Loss

This program generates the data for each move n, stored as (S1, π1, Z1), ..., (Sn, πn, Zn) through self-play, where π and Z are used as the labels for training, and S is a set of all positions in a game. Each position S1, ..., Sn represented by the feature plane is input into the neural network, and then a probability P and a value v are output. This process is to minimize the error between value v and Z, and between π and probability P. The Loss function is as follows:

L = (z−v)2 − πTlogP + c||θ||2

where the first part of this loss function is mean squared error, the second part is the cross-entropy loss function [14], and the last part is the L2 norm [15] used to prevent overfitting.

Considering the rule of Gomoku and the size of the board, only four feature planes are used as input to the neural network. The first two feature planes indicate the current player's position and the opponent's position, respectively. The third plane shows whether the current player is black or white. The last plane represents the position of the current opponent's previous move. These four feature planes are represented by a binary matrix. The training loss decreases rapidly at the beginning of the training and then reaches a plateau where it is difficult to decrease, but then continues to decrease as the dataset is extended. During training, I also recorded the total number of steps for each game to track training and adjust the hyperparameters.

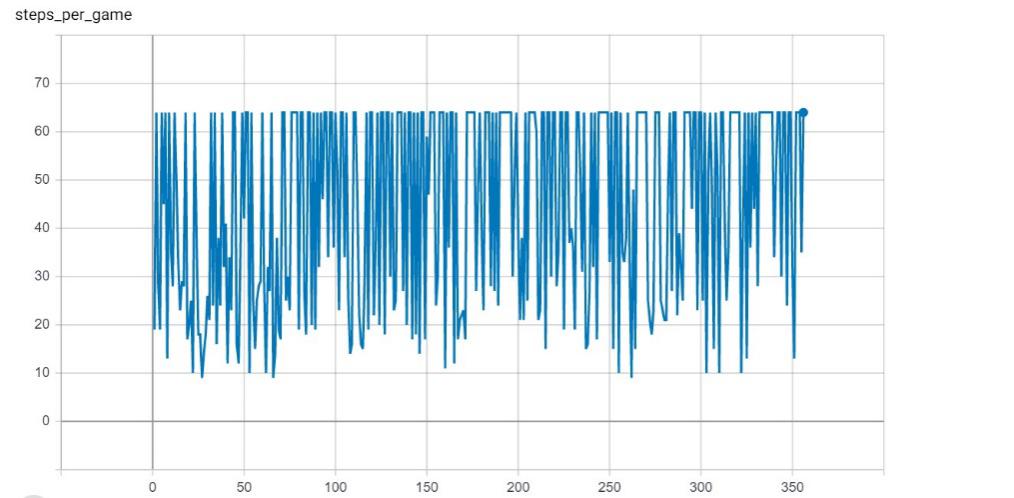


Figure 11. Record of game rounds

After training a certain number of games, the newest model and the optimal model are played against each other (10 games: 1 point for a win, 0.5 points for a draw). If the newest model scores 5.5 or over, it will be replaced by the optimal model.

The number of rounds per game is the feedback for analyzing the training results (as shown in Figure 11). When it was observed that multiple consecutive games ended quickly, this proves that the intelligence of the Gomoku AI is still very low. When multiple games end quickly, this indicates that the GomokuAI's intelligence level is still low or that it is trying out new possibilities. Conversely, when a significant number of games result in a draw, this indicates that the GomokuAI is rarely trying to explore.

4.3 Other Adjustments

4.3.1 Learning rate

During training, the learning rate is initialized to 0.002, and if the loss function has difficulty converging, I will turn the learning rate down to 0.001.

4.3.2 Optimizer

Adam is used and the L2 regularization (weight decay) is set to 0.0001 to prevent overfitting.

4.3.3 Playout

The MCTS search process is also known as playout. Although the more playouts you have, the better the results, this value is set to 400 to 600 in this program due to hardware limitations.

4.3.4 Batch size

A relatively large amount of game data can be obtained by data augmentation, therefore Full Batch was not used. GomokuAI is trained based on a single process. If the batch size is high, it not only requires more memory space but also may cause the neural network to get stuck in local minima. If it is small, it is not conducive to convergence. Therefore, in this program, the batch size is typically set to 512 or 256.

1. PROGRAM RUNNING RESULT

This section shows the performance of AI in single-player mode.

5.1 Human versus machine

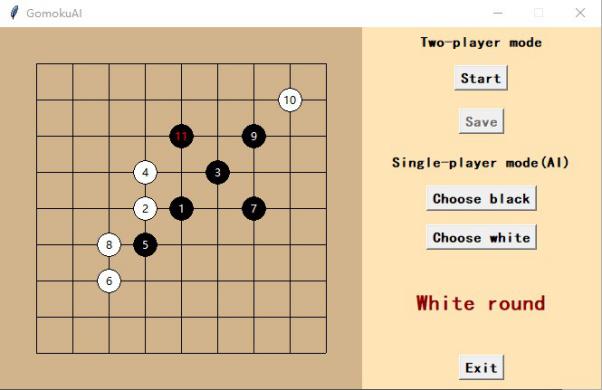


Figure 12a. Human Versus Machine (AI is Black)

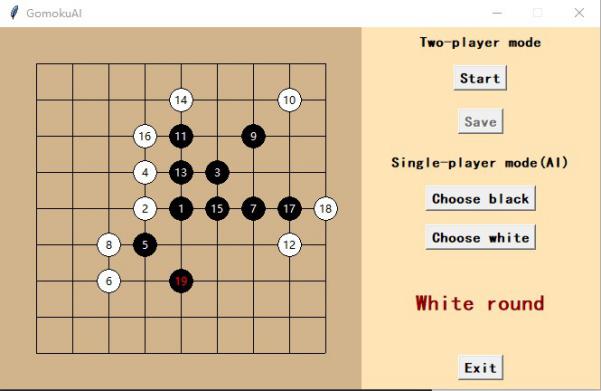


Figure 12b. Human Versus Machine (AI is black)

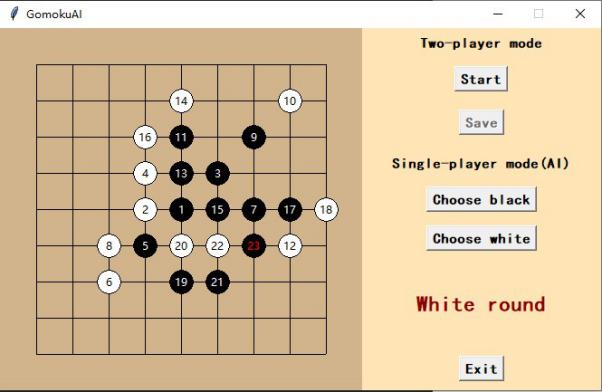


Figure 12c. Human Versus Machine (AI is black)

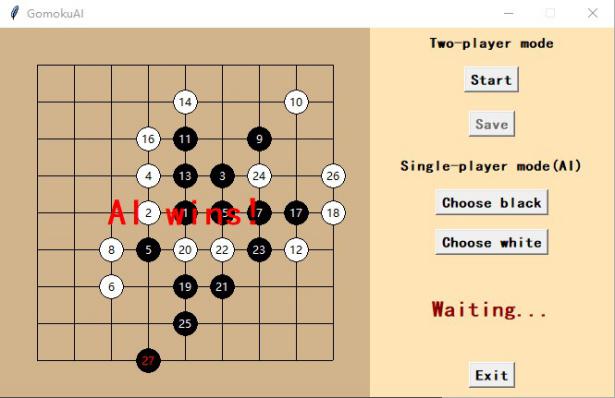


Figure 12d. Human Versus Machine (AI is black)

When GomokuAI played black, it was extremely aggressive. It not only understood the rules of Gomoku but also knew how to increase its advantage. It defended against all my attacks early in the game, and then waited for an opportunity to counterattack (see Figure 12).

1. FUTURE WORK

The core of this program is the training module, in which the process of generating data by self-play alternates with the process of training. MCTS combines with the neural network to generate data and store it in the dataset, and then randomly selects a certain amount of data from the dataset to input to the neural network for training.

I made several changes and trade-offs in the algorithm and hyperparameters of this program according to the size of the Gomoku board and its rules, but there is still room for improvement. First of all, the program is single-threaded, so it is less efficient for training (only 9x9 board is trained in this program). In future work, the training efficiency of GomokuAI can be improved by rewriting the MCTS part of the self-play module by using C++ and using MPI. Secondly, the program only implements the basic rules of Gomoku and the size of the board is not standardized, so this can also be improved in future work. Finally, the feature planes are defined in a simple way and the number of feature planes is small, so more feature planes can be added in the future to improve the performance of GomukuAI.

REFERENCES

1. Silver David et al. Mastering the game of Go with deep neural networks and tree search. Nature, 2016, 529(7587):484-489
2. François-Lavet, V., Henderson, P., Islam, R., Bellemare, M. G., & Pineau, J. (2018). An introduction to deep reinforcement learning. arXiv preprint arXiv:1811.12560.
3. Vardi, Avi. "New minimax algorithm." Journal of Optimization Theory and Applications 75, no. 3 (1992): 613-634.
4. Knuth, Donald E., and Ronald W. Moore. "An analysis of alpha-beta pruning." Artificial intelligence 6, no. 4 (1975): 293-326.
5. Browne, C. B., Powley, E., Whitehouse, D., Lucas, S. M., Cowling, P. I., Rohlfshagen, P., ... & Colton, S. (2012). A survey of monte carlo tree search methods. IEEE Transactions on Computational Intelligence and AI in games, 4(1), 1-43.
6. Coulom, R. (2006, May). Efficient selectivity and backup operators in Monte-Carlo tree search. In International conference on computers and games (pp. 72-83). Springer, Berlin, Heidelberg.
7. Ioffe, S., & Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. arXiv preprint arXiv:1502.03167.
8. Tanner, M. A., & Wong, W. H. (1987). The calculation of posterior distributions by data augmentation. Journal of the American statistical Association, 82(398), 528-540.
9. Audibert, J. Y., & Bubeck, S. (2010, June). Best arm identification in multi-armed bandits.
10. Kocsis, L., & Szepesvári, C. (2006, September). Bandit based monte-carlo planning. In European conference on machine learning (pp. 282-293). Springer, Berlin, Heidelberg.
11. LeCun, Y., & Bengio, Y. (1995). Convolutional networks for images, speech, and time series. The handbook of brain theory and neural networks, 3361(10), 1995.
12. He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 770-778).
13. Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., ... & Lillicrap, T. (2017). Mastering chess and shogi by self-play with a general reinforcement learning algorithm. arXiv preprint arXiv:1712.01815.
14. Golik, P., Doetsch, P., & Ney, H. (2013, August). Cross-entropy vs. squared error training: a theoretical and experimental comparison. In Interspeech (Vol. 13, pp. 1756-1760).
15. Ng, A. Y. (2004, July). Feature selection, L 1 vs. L 2 regularization, and rotational invariance. In Proceedings of the twenty-first international conference on Machine learning (p. 78).