MASTERING THE GAME OF GOMOKU WITHOUT HUMAN KNOWLEDGE

A Thesis

presented to

the Faculty of California Polytechnic State University,

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

by

Yuan Wang

June 2018

COMMITTEE MEMBERSHIP

TITLE:	Mastering the Game of Gomoku without Human Knowledge
AUTHOR:	Yuan Wang
DATE SUBMITTED:	June 2018
COMMITTEE CHAIR:	Franz Kurfess, Ph.D. Professor of Computer Science
COMMITTEE MEMBER:	Lubomir Stanchev, Ph.D. Associate Professor of Computer Science
COMMITTEE MEMBER:	Hisham Assal, Ph.D. Lecturer, Computer Science and Software Engineering Department

ABSTRACT

Mastering the Game of Gomoku without Human Knowledge

Yuan Wang

Gomoku, also called Five in a row, is one of the earliest checkerboard games invented by humans. For a long time, it has brought countless pleasures to us. We humans, as players, also created a lot of skills in playing it. Scientists normalize and enter these skills into the computer so that the computer knows how to play Gomoku. However, the computer just plays following the pre-entered skills, it doesn't know how to develop these skills by itself. Inspired by Google's AlphaGo Zero, in this thesis, by combining the technologies of Monte Carlo Tree Search, Deep Neural Networks, and Reinforcement Learning, we propose a system that trains machine Gomoku players without prior human skills. These are self-evolving players that no prior knowledge is given. They develop their own skills from scratch by themselves. We have run this system for a month and half, during which time 150 different players were generated. The later these players were generated, the stronger abilities they have. During the training, beginning with zero knowledge, these players developed a row-based bottom-up strategy, followed by a column-based bottom-up strategy, and finally, a more flexible and intelligible strategy with a preference to the surrounding squares. Although even the latest players do not have strong capacities and thus couldn't be regarded as strong AI agents, they still show the abilities to learn from the previous games. Therefore, this thesis proves that it is possible for the machine Gomoku player to evolve by itself without human knowledge.

Keyword: Monte-Carlo Tree Search, Neural Network, Reinforcement Learning, Game

ACKNOWLEDGMENTS

Thanks to:

1. My wife and parents for being with me for the past 4 years of school.

2. Dr. Kurfess for guiding me through this process.

3. Dr. Lo for helping me to check the grammar issues.

4. Google’s DeepMind for inspiring the idea and the name of this thesis.

TABLE OF CONTENTS

Page

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

Chapter 1

INTRODUCTION

This is the era of Big Data, where neural networks can make use of this huge amount of data by analyzing its inner relationships. However, in many cases, these inner relations cannot be built precisely. Even the most powerful network cannot exhaust the possibilities of a checkerboard game, the game of Chess involves 35^80 [12] possible sequences of moves while Go involves 250^150 [12] such moves, in comparison, the number of atoms of the universe is 10^90 which is significantly smaller than Go's possibilities. Without having a precise analysis of the data, a neural network player won't beat a professional human player. This fact forces researchers to consider what can be done to improve the neural network's performance before the explosive breakthrough of the technology which may happen in the far future.

A precise understanding of the data's inner relations, in terms of checkerboard games, means building policy function and value function precisely. The former outputs move probabilities and the latter outputs a position evaluation. According to previous research [4], these games may be solved by recursively computing the precise policy function and value function in a search tree containing possible sequences of moves, where b is the game's breadth (number of legal moves per position) and d is its depth (game length). However, since both of these two variables are relatively large, thus making the balanced tradeoff between the machine player's ability and time efficiency impossible if all the possibilities need to be searched.

In order to solve this problem, numerous efforts have been made on reducing the game's breadth and depth. The technology of Monte Carlo Tree Search (MCTS) [13] has been widely used to provide not optimal but still acceptable policy/value functions in many games and achieved superhuman performance in Backgammon [14] and Scrabble [15]. But in the field of Go, the MCTS approach has only achieved amateur level's play [4]. A plausible reason for this might be that linear analysis cannot handle the huge

search space of Go, so acceptable policy/values functions cannot be generated by this approach.

Recently, by creatively embedding the technology of deep neural network into the structure of MCTS, Google's Go player AlphaGo [4] made a breakthrough development in AI checkerboard game by beating human the player Ke Jie, the world-champion of Go. Moreover, the newest version of AlphaGo, AlphaGo Zero [7], gets an even better performance by totally discarding human's data and initializing the neural networks from scratch.

With these technology developments in hand, this thesis proposes an AI Gomoku player. It involves technologies like Convolutional Neural Network (CNN), Residual Neural Network (RNN), Monte Carlo Tree Search (MCTS) and Reinforcement Learning (RL). Inspired by AlphaGo Zero, the networks implemented by this thesis will not use experts' plays as its training data, instead, they will be initialized from random weights and been trained through self-play. This approach ensures that the generated policy/value functions will not follow human expert's habits and will develop a learning strategy that is qualitatively different to human players.

There are three major differences between this thesis and AlphaGo Zero. Firstly, the game is different. Secondly, AlphaGo Zero ran training and playing processes in parallel, while this thesis treats them separately and therefore has a different training strategy. Finally, this thesis tries to use a computing resource that is far worse than AlphaGo Zero's to achieve the same functionality as AlphaGo Zero.

The goal of this thesis is twofold. On the one hand, to design a system that is able to train machine Gomoku player. On the other hand, this training is based on self learning, without the help from prior human knowledge. This thesis tries to demonstrate that in the domain of Gomoku, a strategy which learns from scratch is possible.

Chapter 2

BACKGROUND

2.1 The Game of Gomoku

Gomoku, also known as Five in a Row, is one of the most popular checker games in the world. Two players, indicated by different stone colors, alternate turns placing stones on a checkerboard, usually the size of the board is 15*15 or 19*19. The winner is the first player who forms his stones in an unbroken line of five horizontally, vertically, or diagonally. Figure 2.1 shows an example of the winning board of Gomoku. The player with the black stones wins the game by connecting the stones No.35, No.33, No.27, No.29, and No.37 diagonally.

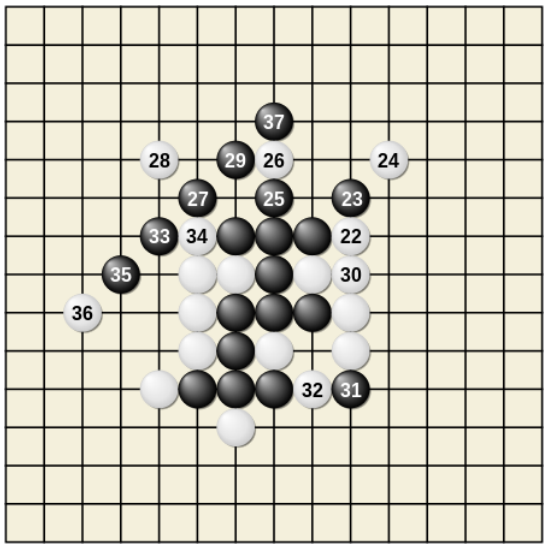

Figure 2.1. A winning board of Gomoku.

Different from other checker games, like the game of Chess (which contains rules that don't allow the players do some particular moves), the rules of Gomoku are loose. The player can place the stone to wherever he wants if that location is empty.

2.2 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) is a method for finding optimal decisions in a given domain by taking random samples in the decision space and building a search tree according to the result [1]. In the domain of checkerboard game, the purpose of MCTS is to approximate the potential actions that may be taken from the current board state by searching a partial tree. Due to the huge space complexity that is required by traditional checkerboard games, (e.g., Chess's space complexity is 35^80, the case for Go is 150^250). The MCTS has been widely used to reduce the excessive use of computing resources since it doesn't require to search the whole tree.

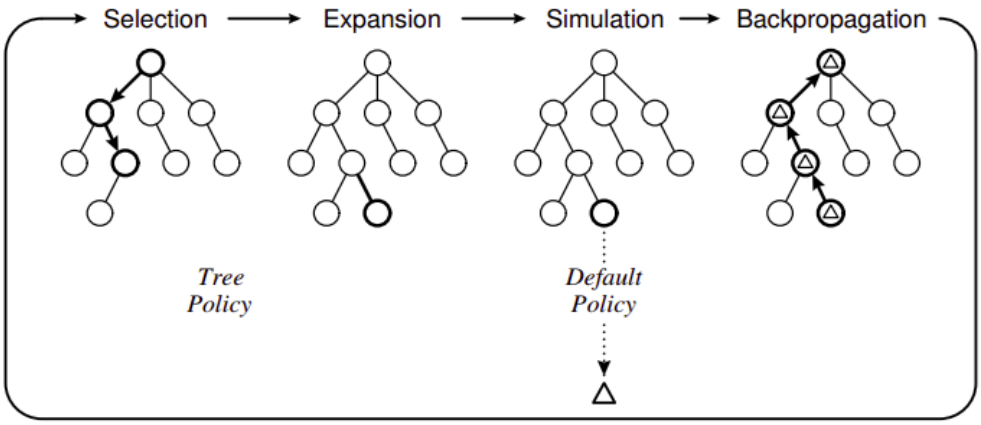

Figure 2.2. The iteration process of MCTS [1].

Each iteration of MCTS consists of four phases: Selection, Expansion, Simulation and Backpropagation; until the terminal state such as win or timeout is met, these four phases will be performed repeatedly. As shown in Figure 2.1, the process of the iteration can be explained as follow:

1. Selection: Starting from the current state, based on the statistic data stored in its child states, one with highest value will be chosen. Recursively apply the selection process until an edge state is met. An edge state means a terminal state, or it has unvisited child states. In the first case, the game ends. In the second case, since there is no value stored in unvisited state, the selection process stops.

2. Expansion: Once the edge state is met, an unvisited child state is selected randomly and is added to the tree. The searching policy during the selection and expansion phases is called Tree Policy, which is different from the policy of the next phase.

3. Simulation: From the newly added state, a simulation is then run based on a searching policy called Default Policy. The Default Policy selects the next action randomly, and is performed repeatedly until a terminal state is met.

4. Backpropagation: The outcome achieved by the simulation phase is transferred into a reward value, and is back propagated through selected states in the selection phase to update their statistic values. One point that needs to be mentioned is, the reward value will be only updated in these selection phase's states, and those states chosen by simulation phase are gone and won't be affected.

As mentioned before, MCTS performs two kinds of search policies; they have different usages and are the key point of this algorithm. When all of the potential next actions are visited states, based on the statistics data stored in those states, the Tree Policy performs a calculation to determine the next state. When the statistics data is not available, which means that current state has unvisited children, the Default Policy will pick the next state randomly without performing any calculation. These two policies are shown in Figure 2.2 and Figure 2.3.

function BESTCHILD(v, c)

return $\underset{v' \in \text{children of } v}{\arg\max} \frac{Q(v')}{N(v')} + c\sqrt{\frac{2\ln N(v)}{N(v')}}$

Figure 2.3. The calculation process of the Tree Policy [1].

Where the N(v) means the visit count of state v, Q(v) means the total rewards of node v for all players, and c is a constant value.

```
function DEFAULTPOLICY(s)
    while s is non-terminal do
        choose a ∈ A(s) uniformly at random
        s ← f(s, a)
    return reward for state s
```

Figure 2.4. The random selection process of the Default Policy [1].

MCTS has two advantages especially in the field of machine checkerboard game. Firstly, it improves the performance of the machine game player through searching only a partial tree instead of building a whole tree. The latter will be inefficient and in some cases impossible due to the game's space complexity. Secondly, its estimated value will become more and more accurate with the increase of the simulation times and nodes accessed [2], which means the MCTS has potential ability to improve itself.

MCTS has its own limitation, it relies on the randomness too much and is not efficient enough. In order to solve this problem, those current machine checkerboard players based on MCTS use a policy abstracted from human expert moves to replace the original randomly Default Policy. This approach achieves good results [3]. However, according to [4], this approach has been limited to shallow policies based on a linear combination of input features. Without the help of technologies like neural networks, MCTS cannot exhaust its maximum potential power.

2.3 Neural Network

Neural Networks are a kind of machine learning, and have been widely used in domains like Natural Language Processing, Image Classification, Reinforcement Learning, etc. It can analyze the input features in a non-linear way, and approximates the transformation function between the input features and the target values with high confidence. It is especially helpful in the domain of machine checkerboard games. If we want to use other policy strategies to reduce MCTS's randomness, neural networks perform much better than traditional methods in gathering those policies.

There are two types of neural networks need to be mentioned, Convolutional Neural Networks (CNNs), and Residual Neural Networks (Resnets). The former helps to extract the most useful information for the task at hand, while the latter makes it possible to enlarge the network model, and to reduce the synchronization problem in the case where different network models need to learn the same thing.

2.3.1 Convolutional Neural Network

The key operation of a CNN is convolution. Each convolutional layer has multiple small sizes of kernels where each of them is related to a pattern. As shown in Figure 2.4, convolving this kind of kernel to the input features means to extract that specific pattern from the input. This is done by moving the kernel through the whole input and performing the convolution operation continuously. As a result, the original picture is transformed into a set of pictures consisting of different patterns, meaning it has been abstracted in different ways. Compared with the traditional fully connected layer which converts the input picture into a 1-dimensional vector, the convolutional layer converts it into a more hierarchical and clear format, and makes the future analysis easier.

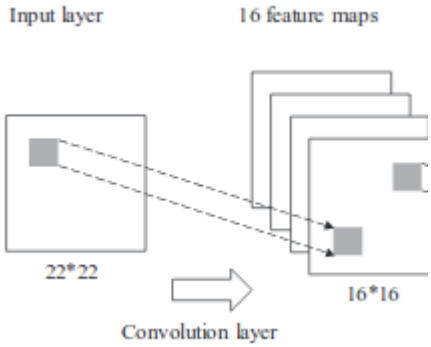

Figure 2.5. The convolution operation of CNN [5].

CNNs are especially suitable where the input value is large and has depth. This makes it a good candidate for machine checkerboard game players, since the board is always large, and each square needs to preserve different information such as a player's number, turn to play, etc.

2.3.2 Residual Neural Network

CNN focus on training the network model in a more efficient way, while the Resnet is trying to fully use CNN's potential ability, which means it enables a network model to contain as many convolutional layers as possible.

It is well known that due to some unsolved technique problems, such as gradient vanishing, the network model can't be expanded to a large number of layers, otherwise relevant information will be lost during the long back propagation process. Although the exact cause of the information vanishing has not been located yet, researchers developed Resnet to bypass it. A Resnet consists of residual blocks, as shown in Figure 2.5. The key point of this block is that it keeps adding shortcuts around at least each two convolutional layers. This means that before the actual convolution operations, it preserves a copy of the input value, and this copy is then added back to the output after the convolution operations are finished. By doing this, it ensures that even if the useful information is lost during the calculations, that information will be added back at the end of the calculation. As a result, since most of the useful information will be preserved during the training, the network model can be designed much deeper. And with more convolutional layers involved, such a network model will have better performance.

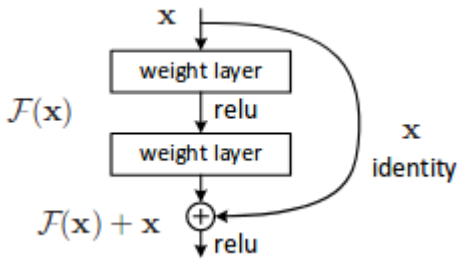

Figure 2.6. Residual block [6].

The Resnet is useful in two cases, and thus is suitable for machine checkerboard game field. The first case is where the training is complicated enough and couldn't be done by a shallower network model, such as Checkers. The second one is when there are multiple models that need to learn the same features during some periods, in this case those models can be incorporated into just one Resnet model. They learn the same

features since they are in one model, and when they need to learn different things, the Resnet model gives each of these “models” a shortcut and leads them to different outputs. The usage of the second case is not so clear here, but it will be explained further when the concept of Reinforcement Learning is introduced.

With the combination of neural networks and MCTS, a powerful checkerboard game engine will be built, where the neural network can help MCTS to reduce its randomness, and MCTS can provide the network model a platform which can further enhance its performance. However, according to [7], this approach can be even further improved with Reinforcement Learning.

2.4 Deep Q-Learning Framework

The reinforcement learning framework helps computers learn to play games by starting off afresh and learning by making errors and slowly getting better after seeing the rewards [8]. It gives the combination of MCTS and neural network model the ability to continuously improve itself. The specific framework that is used in the domain of machine checkerboard game is called Deep Q-Learning (DQN) framework.

The core of a Q-Learning framework is a table Q(S, A). It represents a function that outputs the reward value for a state S if it takes the action A. For each state S, only the action with the highest Q value will be chosen. The statistic data stored in states is updated after each play, so the Q value continuously becomes more accurate, and gives the player a better prediction. Instead of the table Q(S, A), DQN uses a neural network to approximate the reward function. According to [8], this approach works well.

Specific to machine checkerboard games, the reward Q function is divided into two network functions, a policy function and a value function [4]. The policy function returns a potential next action based on a given state, while based on the same state, the value function returns the probability score that indicates if this state will win. The outputs from these two functions are then balanced by a constant λ to output the final

reward Q value. By doing this, the DQN ensures that the machine player balances the local game state and the sense of the bigger picture.

To summarize these three techniques introduced by this section: For a machine checkerboard game player, the role of MCTS is a search structure which helps to reduce the size of the search tree and to enhance the searching efficiency. The neural network model will be used to replace MCTS's simulation Default Policy to avoid the original method's randomness. Furthermore, in order to let the machine player to have the ability to improve its quality continuously, the DQN arranges the network models into the format of policy model and value model, and changes MCTS's Tree Policy accordingly.

Chapter 3

RELATED WORK

3.1 Fuego

Fuego is an open-source software framework for developing game engines for full-information two-player board game, with a focus on the game of Go [9]. It enhances MCTS with a playout policy and prior knowledge trained to predict human expert moves. Alongside with a lock-free multithreaded environment, Fuego significantly improves MCTS's performance, while limits its randomness. In 2010, it was the top program in 9 * 9 Go, and was respectable in 19 * 19 Go [4].

To summarize the modifications of MCTS made by Fuego, the key point is, Fuego wants to reduce MCTS's search tree further by taking human experts experiences into account. The most interesting part of Fuego is that it holds a small set of hand-selected 3 * 3 patterns built by human experts. During the simulation process, a move will be chosen if it matches one of these patterns and is adjacent to the previous move; this is called Playout Policy. Only in the case where no adjacent pattern is found, Fuego lets the traditional MCTS Default Policy to choose the moves randomly. Since the Default Policy is less efficient than Playout Policy, in order to increase the latter's proportion, Fuego developed a Replacement Policy which attempts to move tactically bad moves to an adjacent point [9], which indicates that a move based on the pattern match is more likely to happen.

Alongside with the Replacement Policy, two more strategies, the prior knowledge and the move filter, are used to narrow the search to a set of moves with high probabilities.

The prior knowledge enables the game engine to reward and punish certain moves based on its features. After finishing every pattern match of the Playout Policy, based on the current selected move, a rewarding credit will be added to all of its neighbors up to a

certain distance. Meanwhile, human experts determine a set of dangerous and useless moves. These moves will be penalized during the simulation process. This strategy is important because it balances the human experts' knowledge with the randomness of MCTS. With different prior knowledges the game engine will have different performances.

While the prior knowledge uses the reward and penalty to affect the game indirectly, the move filter weeds out bad moves directly, which immediately reduces the size of the search tree. According to [9], based on human experiences, moves not visible within the search tree will be pruned, such as unsuccessful ladder-defense moves in the domain of Go. Also, moves will be pruned if they lead to edge states with no neighbors up to a certain distance.

In 2008, Fuego achieved the 1st place in the KGS competition and 4th place in the 13th International Computer Games Championship. However, according to [4], the biggest drawback of Fuego, alongside with many other pure MCTS based game engines like Crazy Stone and MoGo, is that it is trying to analyze the input features in a handcrafted way, which means all of the additional human knowledge is combined linearly. Compared with the nonlinearity provided by neural networks, the linearity limits the performance of pure MCTS based game engines to make strong amateur play.

3.2 AlphaGo

In 2016, AlphaGo won against Lee Sedol, a 18-time Go world champion, in a five-game match, with the score of 4 : 1, which was the highest achievement of a computer game engine in the longtime human-computer Go challenges. Considering all of the game engines that before AlphaGo could only play Go at a strong amateur level, beating the highest professional human player made AlphaGo a state of the art Go engine at its time.

Different from the pure MCTS based game engines like Fuego, AlphaGo introduces a nonlinear way by using a neural network to make better use of human experts knowledge. During the MCTS simulation process, unlike Fuego, AlphaGo is not

looking for the matching of the handcrafted patterns, but it uses convolutional neural network models to perform the MCTS Default Policy. This approach has two advantages. Firstly, the network models are trained from human plays, thus they enhance the original Default Policy by significantly reducing its randomness. Secondly, the useful information from human plays is extracted by network model with a high degree of nonlinearity, which is much more efficient than doing this by hand.

Specifically, AlphaGo trains a policy network and a value network to perform the Default Policy. According to [4], it trains a 13-layer policy network with 30 million positions from the KGS Go Server with an accuracy of 57.0%. In order to further improve the performance of the policy network, a reinforcement learning framework is applied. This framework lets different policy networks selected from different training iterations play with each other until a terminal state is reached with a reward value of +1 for winning and -1 for losing. Based on the reinforcement learning plays, a value network with the same structure as the policy network is trained. With policy and value networks in hand, a final approach then combines them into the structure of MCTS to actually play Go at AlphaGo's level.

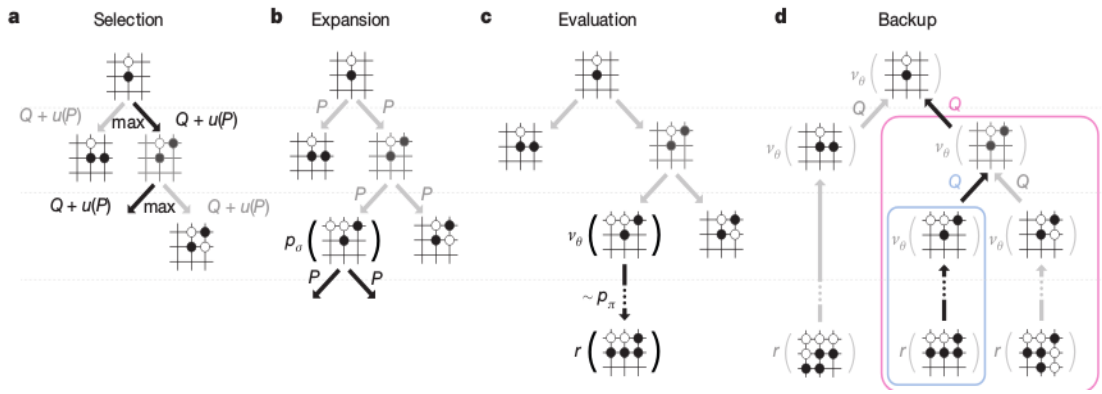

Figure 3.1. The MCTS structure in AlphaGo [4].

As shown in Figure 3.1, the process of AlphaGo's MCTS has two improvements compared with the pure MCTS algorithm. Firstly, inside of the evaluation (simulation) process, instead of using the previous Default Policy to choose next move randomly, the

AlphaGo uses a pre-trained policy network to perform the select. Secondly, inside of the selection process, instead of using the traditional UCB method, AlphaGo uses a new function shown in Figure 3.2 to take policy and value networks into account.

$$\frac{Q(v)}{N(v)} + \frac{P(v|v_0)}{1+N(v)}$$

Figure 3.2. The selection function of AlphaGo Selection process [4].

Combining MCTS with deep neural networks, AlphaGo reached the professional level of Go. Although AlphaGo outperforms Fuego a lot, they still share the same idea which is to bring the human knowledge into the domain of MCTS. However, the newest version of AlphaGo, AlphaGo Zero, reveals that, without the help of human knowledge, computer game engines can perform even better.

3.3 AlphaGo Zero

Recently, AlphaGo Zero became the state of the art computer Go engine that outperforms any other engines [7]. Similar to its ancestors, AlphaGo Zero combines MCTS with neural networks. However, instead of training the network models from human plays, AlphaGo Zero trains them from scratch. This means that at the beginning, the policy and value networks know nothing about the game. The engine then puts these models into a reinforcement learning (RL) framework, and trains them based on their own results. This approach achieved huge success. According to [7] and Figure 3.3, AlphaGo Zero outperforms the highest human professional level in only 70 hours; after that, it won against the original version of AlphaGo in 4 days, and exceeded the best version of AlphaGo in 32 days. Based on this information, the reinforcement learning framework without human knowledge turns out to be the best available choice to implement the computer checkerboard game engine.

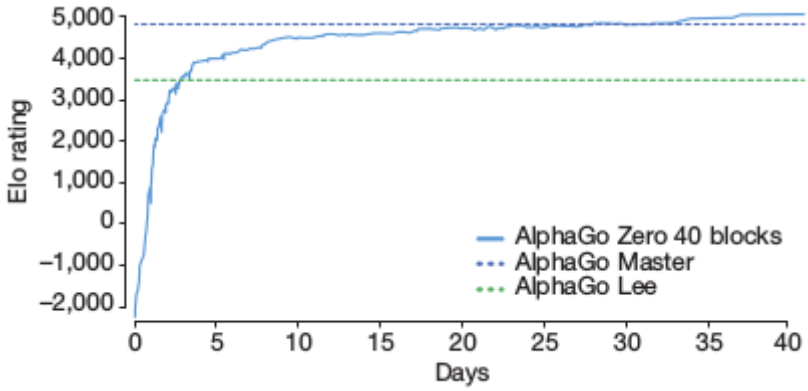

Figure 3.3. The performance of AlphaGo Zero [7].

The improvements of AlphaGo Zero, compared with its original versions, can be summarized in two parts. Firstly, it uses a different way to train the policy and value networks. Secondly, it modifies the structure of the MCTS algorithm so that the new kind of network models can be added in.

The training strategy of AlphaGo Zero can be described as follows. The policy and value networks are initialized randomly, a MCTS structure is then using these models to play against itself. Moves are selected based on the search probability π, which is the output value of the policy network, and the scalar value v, which is the output value of the value network. Once the terminal state s and the winner z are recorded, a training process will be held in parallel. The purpose of this training is to take s as input and output its own search probabilities p and scalar value r, and to maximize the similarity between π and p, while minimizing the error between r and the real winner z. After several such training iterations, the RL framework will start a competition between this newly trained network with the current MCTS network, and the winner's weights will be updated into the MCTS's network.

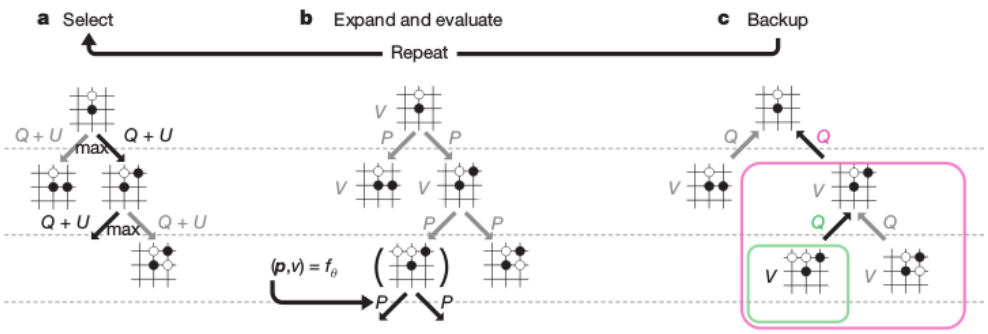

Figure 3.4. The MCTS Structure of AlphaGo Zero [7].

In order to adapt the training changes, AlphaGo Zero also modifies the MCTS structure. It combines the Tree Policy and Default Policy into one, which means for MCTS's four stages, the difference between the expand stage and the evaluation (simulation) stage is now disappeared. As shown in Figure 3.4, the randomly Default Policy is discarded, so instead of letting the previous simulation stage complete the whole play as often as possible, now the four stages are performed repeatedly to finish just one play.

Besides these two differences, AlphaGo Zero combines the policy and value networks into one residual network model with two branches. According to [7], this architecture outperforms the other approaches like separate residuals, dual convolutional, and separate convolutionals.

3.4 Gomoku with ADP and MCTS

In 2016, researchers designed a machine Gomoku engine [11]. They used Adaptive Dynamic Programming (ADP) to train a neural network, combined it with MCTS, and finally found it outperformed the Gomoku engine with a single ADP algorithm.

The workflow of this engine can be described as follow.

1. A shallow neural network with 3 layers which is trained by ADP algorithm is provided.

2. From any given checkerboard state, using the pre-trained network to output 5 candidate moves with their winning probabilities.

3. Using these 5 moves with their associated states to be the root state of MCTS respectively, the MCTS then performs these 5 choices, and outputs 5 MCTS based winning probabilities.

4. Using a weight value λ to balance the network based winning probabilities and MCTS based winning probabilities, the highest one will be chosen as the next move.

Compared with AlphaGo's algorithm, there are two differences with this Gomoku engine. Firstly, the network model is not embedded into the structure of MCTS. They perform separately, and the combination of them is based on their outputs. Secondly, since it is MCTS that actually performs the simulation and plays the game, the policy network is not needed any more, only a shallow value network which outputs the winning probabilities is trained.

Chapter 4

ALGORITHM

4.1 Algorithm Overview

This project designs a machine Gomoku engine by combining the technologies of MCTS, deep neural network, and deep Q reinforcement learning framework. Inspired by AlphaGo Zero, the game engine in this project is trained without the help of any prior human knowledge.

The whole process can be summarized as follows.

1. Designing a residual neural network model (Resnet) with a policy head and a value head. The input value of that residual model is the current board state. Based on the same state, the policy head returns the possibility scores for all of the possible moves, sorted by the scores, while the value head returns a scalar value indicating the winning probability. This network is initialized randomly, no human expert plays will be trained into the model.

2. Designing a search structure which combines the MCTS and the Resnet. Different from the traditional MCTS, this structure incorporates the selection, expansion, and backup stages into one simulation stage, combines the Tree Policy and the Default Policy, and implements a new framework to store the statistical values to make use of the Resnet. For the simulation stage, in the case where all of the potential moves have complete statistical values, the engine chooses the move with the highest value. This simulation process is performed repeatedly until a terminal or edge state is met. The engine then uses the values from the Resnet to directly predict the next move without starting a simulation stage, which means it won't play a simulation game to against itself. Once all of these values are backed up, the actual next move will be chosen.

3. A different training process is performed along with the playing process. Each time a Resnet model is incorporated into MCTS to perform the prediction, the engine preserves a copy of that model outside of the playing process. That copy is then trained with the values from the ongoing playing process. In order to check if the new model is better than the current one, for every such iterations, n times simulation plays will be held between the combination of the MCTS and the original Resnet model, and the combination of the MCTS with the newly trained Resnet model. The Resnet model's weights from the winner side will be used to continue the playing process.

4. Repeating 2 and 3. On the one hand, the predictions made by the Resnet model will be more and more precise. On the other hand, the combination of the Resnet model and MCTS will have better and better performance. Theoretically, this repeating process has no end.

This chapter will focus on Items 2 and 3, including the algorithm part of the game engine which is related to the MCTS's playing process and the network's training process. The next chapter will discuss Items 1 and 4, which is more related to the implementation part.

Although there are multiple other AI checkerboard game engines, this project has its own value. Different to the pure MCTS based game engines like Fuego, this project introduces the technology of neural networks to extract the features of the checkerboard state in a nonlinear way, and this extraction needs no prior human knowledge. Different from AlphaGo, this project doesn't train the network model before incorporating it into the MCTS, but trains the model alongside with it. Also unlike from AlphaGo Zero, this project doesn't arrange the training process and the playing process in a parallel way; the details will be discussed in the next chapter. Different from the Gomoku engine with ADP and MCTS, this project has full implementations of policy and value networks. And the network model is embedded into MCTS, they are performed together, not separately.

4.2 The Playing Process of MCTS

4.2.1 Data Structure

This project uses a tree to represent the Gomoku game. As shown in Figure 4.1, each node in this tree represents a board state. A board state contains a raw board and pieces on that board. The state is called child state when it is derived from the previous state. Starting from a root state, in three cases a state becomes a leaf state. Firstly, the first time the state is searched, none of its child states have been searched yet. Secondly, the board is full and the state couldn't have any subsequent states. Thirdly, it wins the game and the state becomes a terminal state.

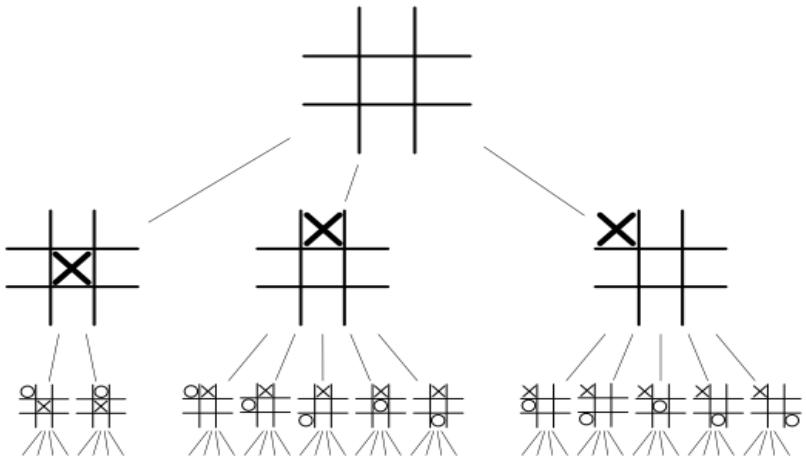

Figure 4.1. The tree of the board game.

The tree contains an enormous number of different top-down routes, and the target of the searching process is to find the best one that leads to the winning state. It is impractical to build the whole tree before playing game since for the game like Gomoku due to the tree size. Instead, the tree is initialized with an empty state, and as the game unfolds, the search process determines a set of subsequent states to be expanded. This ensures that in most cases the tree is only partially built, thus saving the search time.

In order to increase the efficiency of the partial tree which means only useful states would be expanded, each traversed state preserves statistical values to keep track of its usefulness. So based on these pre-saved values, the search process selects moves.

Specifically, for each traversed state, there are four values that need to be saved. They are N(s, a), W(s, a), Q(s, a), and P(s, a), where (s, a) means based on state s makes an action a, N(s, a) is the visit count, W(s, a) is the total action value, Q(s, a) is the mean action value, and P(s, a) is the prior probability of putting a onto s.

The way to calculate these 4 values can be explained as follows.

N(s, a) is the visit count. Each time when a search engine tries to simulation a specific action a on a specific state s, it increments the N value for that specific state s by 1. Without this value any calculation couldn't be accurate, since we cannot safely compare with a 1-traverse state and a 100-traverse state.

W(s, a) is the total action value. $W(s, a) = \sum(s'|s,a \rightarrow s')*V(s')$ where $s, a \rightarrow s'$ indicates that a simulation eventually reaches s' after taking action a from state s [1], and V(s') means the scalar value produced by the value network which indicates the winning probability score of the state s'. In other words, the value W indicates that, if it takes action *a* from state s, returns the winning probability of the new state s'.

Q(s, a) is the mean action value. $Q(s, a) = W(s, a) / N(s, a)$. This function takes the value N into account to make the action value more accurate. And it is this Q that will be used in the actual searching process.

P(s, a) is the prior probability value. It is a value returned from the policy network. Taking state s as input, the policy network returns a vector of values indicating the probability score for each of the potential next moves, P(s, a) indicates the specific probability score for the action a.

4.2.2 The Simulation Stage

Before the game engine picks the next move, it starts a simulation mode to play against itself and to gather statistical information. The next move is then decided based on this information. Different from the traditional MCTS method where selection, expansion, simulation, and backpropagation are separate stages, this project follows the strategy of

AlphaGo Zero that puts three different functionalities, selection, expansion, and backpropagation, into one simulation stage. The reason of AlphaGo Zero and this project can use this different approach is that, instead of doing the simulation randomly, the new strategy stores information from prior searches in each state it traversed, which means the current simulation can make use of prior simulations. By taking this strategy, there is no need to start a brand new simulation every time, since most of the information has been preserved priory, and the searching process needs only to update or to expand it.

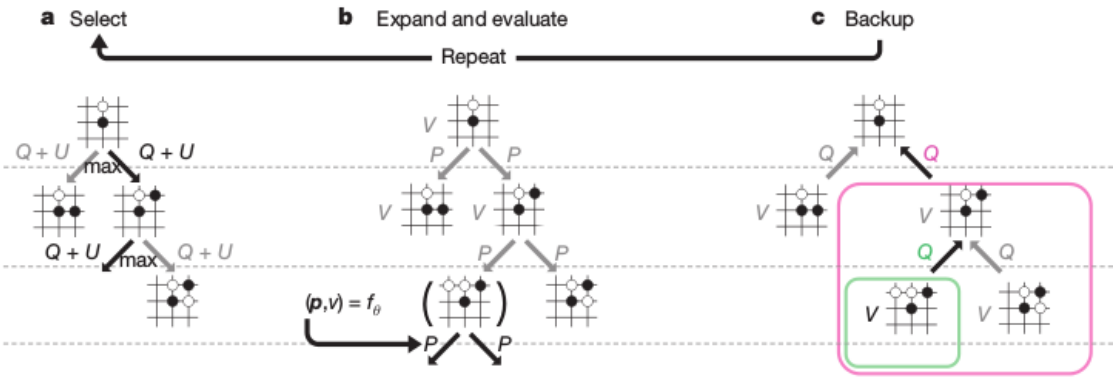

Figure 4.2. The simulation stage in this project [7].

Figure 4.2 shows the simulation process implemented by this project. This process can be generally explained as follow. Starting from state s, in Case 1 where its child states' values can be calculated, choose one state with the highest value as its next state. Case 2 where those values couldn't be calculated, it then assigns its child states new values, and backs up these new values. Then repeat Case 1. The simulation mode is a repetition of this whole process, until the game reaches a terminal state.

More specifically, starting from a state s, there are two cases that might happen.

The first case is that the state has been traversed, which means its child states preserve the necessary information. In this case, the next action a is calculated by $a = argmax(Q(s,a) + U(s,a))$ where U(s, a) is an upper confidence bound value; its calculation function is shown in Figure 4.3. As shown in Figure 4.3, the U(s, a) is mainly related to P(s, a) which is the output value from the policy network. Therefore, the

function to calculate the next action a can be regarded as a balanced choice between the value network (Q(s, a)) and the policy network (U(s, a)).

$$U(s,a) = c_{\text{puct}} P(s,a) \frac{\sqrt{\sum_b N(s,b)}}{1 + N(s,a)}$$

Figure 4.3. The function of U(s, a) [7]

The second case is that this state s has never been traversed, which means its child states lack the necessary information. In this case, based on the state s, the policy network produces a vector of probability scores, for each of the child state of s, the correspondence probability score is then assigned. At the meantime, for each of the child states, the value network returns a scalar winning probability score. Base on those scores, the N(s, a), W(s, a) and Q(s, a) are calculated and assigned to the child states, respectively.

In the second case, since the Q values of its child states are newly calculated and haven't been considered by the current state s, a backup process is performed. The purpose of this process is to back-propagate the newly calculated V(s) to the root state, and during this process, all of its parent states' Q values and W values will be updated to reflect the newest situation. The updating functions for W and Q are

$W(s, a) = W(s, a) + v$, and

$Q(s, a) = W(s, a) / N(s, a)$, where the

N(s, a) = N(s, a) + 1.

By combining these two cases, the simulation mode is then able to play against itself, it ends when a terminal state is met. Then starts the playing stage.

4.2.3 The Playing Stage

Once finishing the simulation stage, the final search probability π is returned. These probability scores are Q(s, a) + U(s, a) for each of the child states from the current root state, with Q values updated during the simulation stage. In order to take the visit count of the root state into account, the final calculation function for π is

$$\pi = [N^{1/\tau} * (Q(s, a) + U(s, a)) \mid s \in \textit{child states of the root state}],$$

Where N is the visit count and τ is a parameter controlling temperature. The game engine picks the next real move a = argmax(π). This next move a is then becomes the new root state in the next simulation mode.

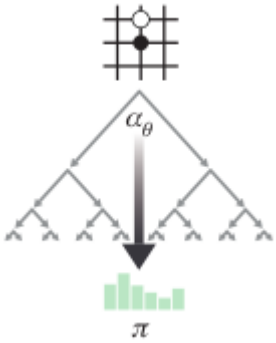

Figure 4.4. The playing stage of this project [7].

To summarize the simulation stage and the playing stage, each time the game engine tries to pick the next move, it starts a simulation mode to play against itself and generates statistical values, based on those values, and it then chooses the next action. In order to finish a whole play, suppose such a play contains n moves, it needs to process the combination of the simulation stage and playing stage n times.

4.3 The Training Process of Resnet

While the engine is playing the game with the combination of the MCTS and the current Resnet model, it keeps producing additional data for the future training of the new Resnet

model. The process of training a new Resnet model with the playing data is called a single training cycle, while the strategy of replacing the original Resnet with the new one is called reinforcement learning.

4.3.1 Single Training Cycle

The model to be trained is a Resnet copy of the current MCTS's Resnet model with the same weights, so this training can be regarded as a self-improvement of the single Resnet model. The reason to train the copy instead of training the original model is that it is not sure if the new training will lead to a better performance.

Each time the engine chooses an action, at set of data (s_t, π_t, z_t) is preserved for the new Resnet model's training, where s_t is the current state at time t, π_t indicates the search probability scores returned by the simulation mode, and z_t indicates if s_t will lead to a final win. The values of s_t and π_t can be gathered at the time the next action is chosen, but until the game engine finishes the whole play, the value of z_t is always empty. Once a terminal state is reached, the winner value is then updated through all of the (s_t, π_t, z_t) sets, with +1 indicating win, -1 indicating lost and 0 mean tie.

After collecting the complete (s_t, π_t, z_t) sets from a whole play, these data are passed to a separate training process. The new model takes s_t as the input value, it then passes s_t through many residual blocks which consists of convolutional layers. When the Resnet is divided into two heads, one policy and one value, the same calculated value will be passed to them respectively. The policy head uses a softmax function to process the passed values and returns a vector of values π' that indicates the probability score for each of the potential moves; it then implements a cross-entropy optimization function to maximize the similarity between the target value π_t and the output value π'. The value head returns a scalar value z' indicating the winning probability of the input s_t; it then uses a mean-square error (MSE) function to minimize the error between the target value z_t and the output value z'.

The purpose of this training is to adjust the copy's weights to learn from the data produced by the playing process. However, the original Resnet model's weights are not affected by the training. In order to determine whether or not the original weights need to be replaced, this project implements a reinforcement structure to handle this problem.

4.3.2 Reinforcement Learning

Each time when finishing the training of the copy model, this project starts 20 plays between the new model and the original one. If the winning time of the new model is 1 times more than the original one's, the new weights will be used in the next playing process and the old one is dropped, otherwise the old weights won't be affected and will be continuously used. In the latter case, the new Resnet model won't be discarded, it keeps training itself with the training data from new plays, until finally it beats the original Resnet model and replaces it. This strategy ensures that the MCTS always lets the best network model to make the predictions. The more plays the game engine performs, the more versions of the network model will be produced. And the performance of the current network model is improved continuously.

Chapter 5

IMPLEMENTATION

This chapter describes the project's implementation side. Specifically, it covers three aspects. First, the data structure that represents the Gomoku game. Second, the implementation details of the Resnet model. Last, the system design of this project which incorporates all of these components.

5.1 Settings of Game Rule

Gomoku is a checkerboard game with a winning rule of connecting 5 pieces in a row. This section explains the data structure this project uses to represent the game of Gomoku.

5.1.1 Board, State and Symmetric State

Checkerboard games can have different sizes of their boards, for the purpose of practice, this project chooses 8*8 as the board size which is initialized with an empty python list with length 64. Inside of the list, '0' indicates spots haven't been occupied yet, '1' indicates spots occupied by the player No. 1, and '-1' indicates spots occupied by the player No. 2. In order to make the board more intuitive, when showing the board, '0' will be translated into '-', '1' to 'X', and '-1' to 'O', as shown in Figure 5.1. The board with (0, 1, -1) is the data that will be actually processed inside of the engine, while the board with (-, X, O) will be only used for display purpose.

```
0,  0,  0,  0,  0,  0,  0,  0,          -,  -,  -,  -,  -,  -,  -,  -,  -,
0,  0,  1,  0,  0,  0,  0,  0,          -,  -,  X,  -,  -,  -,  -,  -,  -,
0,  0,  0, -1,  0,  0,  0,  0,          -,  -,  -,  O,  -,  -,  -,  -,  -,
0,  0,  0, -1,  0,  0,  0,  0,          -,  -,  -,  O,  -,  -,  -,  -,  -,
0,  0,  0, -1,  1,  0,  0,  0,          -,  -,  -,  O,  X,  -,  -,  -,  -,
0,  0,  0,  1,  0,  0,  0,  0,          -,  -,  -,  X,  -,  -,  -,  -,  -,
0,  0,  0,  0,  0,  0,  0,  0,          -,  -,  -,  -,  -,  -,  -,  -,  -,
0,  0,  0,  0,  0,  0,  0,  0,          -,  -,  -,  -,  -,  -,  -,  -,  -,
```

Figure 5.1. Board with (0, 1, -1) and Board with (-, X, O).

The system incorporates the Gomoku board into the structure of MCTS by using the board as the nodes of the search tree. The translation from the board to the node can be explained as follow. Firstly, compress the board list from an integer list to a string, combine this string with four empty values indicates the N, W, Q and P which are mentioned last chapter. A node is an object that includes those five values.

However, for the purpose of training, the board with (0, 1, -1) has only 1 dimension, it lacks depth which is not efficient to be trained directly since the Resnet model prefers inputs with multiple dimensions. To increase the board's depth, each 1-dimensional board is translated into a 3-dimensional board, as shown in Figure 5.2. The board of 3 dimensions is then passed to the Resnet model as the input value. In order to distinguish these two boards, the board with 3 dimensions is given a new name 'State'.

```
(0,0,0), (0,0,0), (0,0,0), (0,0,0), (0,0,0), (0,0,0), (0,0,0), (0,0,0),
(0,0,0), (0,0,0), (0,1,0), (0,0,0), (0,0,0), (0,0,0), (0,0,0), (0,0,0),
(0,0,0), (0,0,0), (0,0,0), (0,0,1), (0,0,0), (0,0,0), (0,0,0), (0,0,0),
(0,0,0), (0,0,0), (0,0,0), (0,0,1), (0,0,0), (0,0,0), (0,0,0), (0,0,0),
(0,0,0), (0,0,0), (0,0,0), (0,0,1), (0,1,0), (0,0,0), (0,0,0), (0,0,0),
(0,0,0), (0,0,0), (0,0,0), (0,1,0), (0,0,0), (0,0,0), (0,0,0), (0,0,0),
(0,0,0), (0,0,0), (0,0,0), (0,0,0), (0,0,0), (0,0,0), (0,0,0), (0,0,0),
(0,0,0), (0,0,0), (0,0,0), (0,0,0), (0,0,0), (0,0,0), (0,0,0), (0,0,0),
```

Figure 5.2. State with 3 dimensions. (0,0,0) indicates empty spot, (0,1,0) indicates player No. 1, (0,0,1) indicates Player No. 2.

For the purpose of increasing the training data set, once a state is achieved, the system automatically generates its symmetric state, as shown in Figure 5.3. Suppose one game is finished in 20 steps, each step is simulated 50 times, there will be total of 20*(50+1)*2=2040 states generated by the game.

```
(0,0,0), (0,0,0), (0,0,0), (0,0,0), (0,0,0), (0,0,0), (0,0,0), (0,0,0),
(0,0,0), (0,0,0), (0,0,0), (0,0,0), (0,0,0), (0,1,0), (0,0,0), (0,0,0),
(0,0,0), (0,0,0), (0,0,0), (0,0,0), (0,0,1), (0,0,0), (0,0,0), (0,0,0),
(0,0,0), (0,0,0), (0,0,0), (0,0,0), (0,0,1), (0,0,0), (0,0,0), (0,0,0),
(0,0,0), (0,0,0), (0,0,0), (0,1,0), (0,0,1), (0,0,0), (0,0,0), (0,0,0),
(0,0,0), (0,0,0), (0,0,0), (0,0,0), (0,1,0), (0,0,0), (0,0,0), (0,0,0),
(0,0,0), (0,0,0), (0,0,0), (0,0,0), (0,0,0), (0,0,0), (0,0,0), (0,0,0),
(0,0,0), (0,0,0), (0,0,0), (0,0,0), (0,0,0), (0,0,0), (0,0,0), (0,0,0),
```

Figure 5.3. Symmetric state of the state shown in Figure 5.2.

5.1.2 Winning Rules

The winning rule of Gomoku is having 5 pieces with same color connected in a line; this line can be vertically, horizontally, or diagonally. All 86 possible winning lines are typed into the system manually before the game start, such as [0,1,2,3,4] for the horizontal line, [0,8,16,24,32] for the vertical line, and [4,11,18,25,32] for the diagonal line. The values of each line come from their index number of the 64-length board list.

Each time an action is taken and a new board is achieved, the system checks if this state is a winning state for any of the players. The checking process can be summarized as follows. For each potential winning line, the system gets 5 pieces locations and their corresponding values in the new board, adds these values together and check if the result is +5, -5, or others. Since each spot in the board takes value from the set (0, 1, -1) where 1 means player No. 1 and -1 means player No. 2, thus +5 indicates all of the 5 pieces are taken by player No.1, -5 means all of the 5 pieces are taken by player No. 2, the other results indicates there is no winning for the specific line.

If the system finds there is a winning state, it records the winner, stops the current game, and prepares for the next one. Otherwise, if no winning state is found, before the engine playing the next action, it will check if the current board is full, which means all of the spots are occupied by either ‘1’ or ‘-1’. In this case, if the board is full, the engine stops the current game, records this game as a tie, and prepares for the next game.

5.1.3 Steps of Taking Action

An action is taken in 4 steps.

1. Getting the current board, and translating it into a 3-dimensional state, MCTS starts a simulation mode to update the statistical values of the child states of the current board.

2. From the child states, it chooses the state with the highest (Q+U) value. In the case where multiple child states have the same (Q+U) value, the system randomly

chooses one. In the case where the highest state is not available which means it has been occupied before, the system chooses the next highest state.

3. Based on the move determined by the system, it then updates the current board to a new board, translates this new board into 3 dimensions alongside with its symmetric state, and saves these two states for the future's training.

4. Move the root to the new board, and then repeat this process from 1 to 4.

5.2 Structure of the Resnet Model

This project chooses Resnet since it offers the network model an ability to become deeper. It consists of five residual blocks, with a policy head and a value head.

5.2.1 Residual Block

As mentioned before, the Resnet model consists of different residual blocks, the purpose of these blocks is to ensure that useful information is not lost during this training. There are lots of residual block templates used today [6] [16], this project chooses the one shown in Figure 5.4.

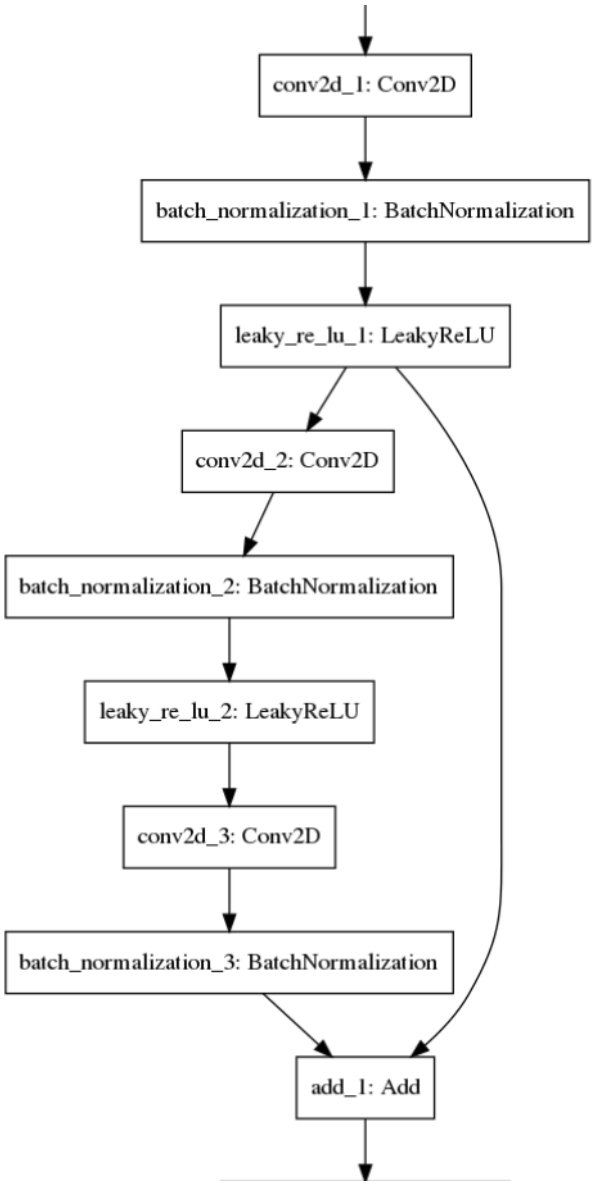

Figure 5.4. The Residual block used in this project, along with the first several layers before the first residual block.

Fig.5.4 shows the first residual block used in this project, along with the first several layers before it. The first convolutional layer outside the residual block contains a kernel size (4, 4); this kernel size is chosen because it is divisible by the board size (8, 8). Then the value is passed to the residual block. As shown in Fig.5.4, the block consists of two convolutional layers, the first one is followed by one batch normalization layer and a Leaky ReLU activation function, the second one is only followed by a batch

normalization layer. The reason that the second convolutional layer lacks the activation function is that the activation function is pushed into the next block, thus it won't be used in the current block.

After the second batch normalization layer, the result value is then added to the value saved before entering this block. Then the new result is passed to the next residual block as its input value. This project's Resnet model consists of 5 such blocks, each with the same structure. The original input value will be passed through those blocks until it hits a point where the network model is divided into two parts and leads to different heads.

5.2.2 Policy Head

After the original input board has passed the 5 residual blocks, the Resnet model already grasps enough features from it, the model then passes the value into a branch called policy head.

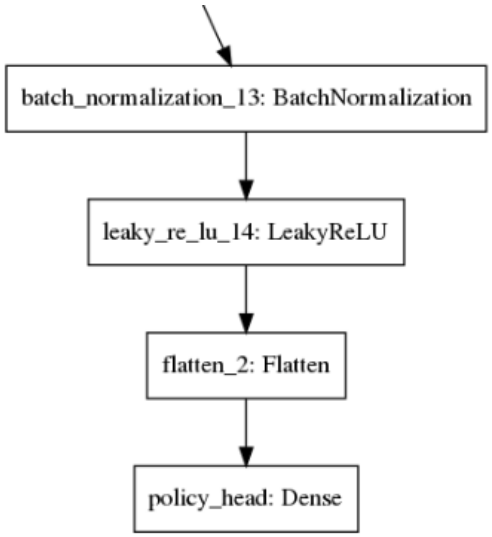

Figure 5.5. The structure of the policy head.

As shown in Figure 5.5, the policy head starts with a batch normalization layer followed by a Leaky ReLU activation function. It then performs a flatten layer to transfer

the value from 3 dimensions into 1 dimension, finally, a fully connected layer with a softmax function converts the passed value into 64 possibility scores, each one indicating the possibility of the correspondent potential move.

5.2.3 Value Head

The value head receives the same value as the policy head, and returns the winning probability of the input board. As shown in Figure 5.6, the value head passes the input value through a batch normalization layer and a Leaky ReLU activation function, and then downgrades its dimension, which is the same as the policy head. The difference is that the value head then performs two fully connected layers with an activation function between them, and the last fully connected layer doesn't have a softmax function with it. The reason that the value head needs two fully connected layers while the policy head only needs one is that the output value's size of the value head is 1, which is much smaller than the size of the policy head's output value, thus more fully connected layers are needed to reduce its size.

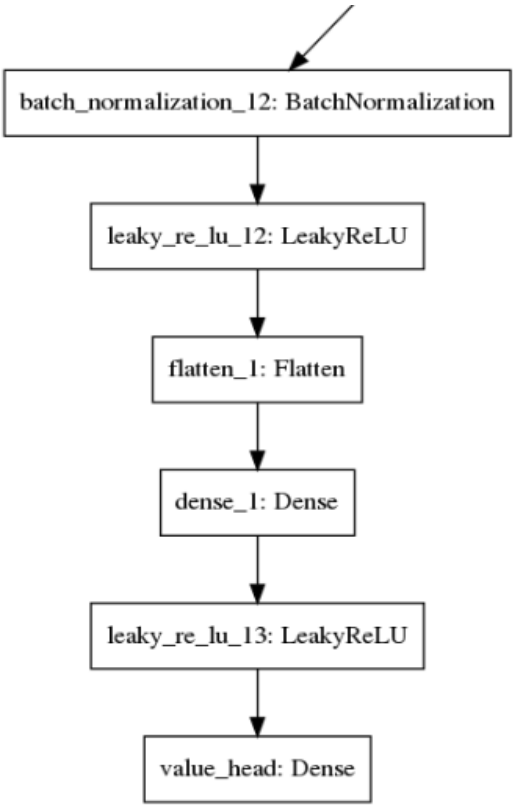

Figure 5.6. The structure of the value head.

The result value of the value head is a scalar value from [-1, 1] inclusive. The closer the output value is to 1, the more likely the input board is to win. Conversely, the closer the result value is to -1, the more likely the input board is to lose.

5.3 Structure of the Driver

This project defines several objects to incorporate all of these functionalities. These objects include Agent, Model, MCTS, Memory, and Archive. This section will explain how they are designed, and how they work with each other.

5.3.1 Agent

Agent is a class object that represents a game player; the system creates two agents for each game. Each agent is initialized with its own Monte-Carlo Search Tree and Residual neural network. The Agent class contains 3 important methods, Simulation, Act, and Replay.

The **Simulation** method performs the Monte-Carlo Tree Search process. Starting from the root node, it first moves to the leaf node where no child state has statistical values. Based on the calculation of the residual neural network, it then evaluates that leaf node, assigning (N, W, Q, P) to each of its child states. Finally, it backfills the values of N, W and Q from the child states up through the tree.

The **Act** method runs the simulation method and makes the final movement decision. It first updates its own Monte-Carlo Search Tree, selecting the current board as the root node of the tree. Based on the root node, multiple simulations are performed to ensure the statistical values of the root's child states are updated multiple times for more accuracy. The Act method then calculates the action value U for each of the root's child states, and chooses one with highest (Q+U). Finally, it updates the board.

The **Replay** method is responsible for retraining the residual neural network. When several games are finished and sufficient training data has been produced, one of the two players will run this method, training its network model with the data so far. This method won't be triggered simultaneously by both players. Also, it is not necessary to run this method for each game.

5.3.2 Model

Model is a class object that represents the residual neural network. There is only one Model in each of the agents. This object is responsible for building the residual network model of the agents. It processes the input value, and makes the prediction in a (scalar value, (vector values)) format.

5.3.3 MCTS

The MCTS object represents the Monte-Carlo Search Tree, it is triggered by an Agent's simulation method. It has 2 important methods, MoveToLeaf, BackFill.

The **MoveToLeaf** method is responsible for finding the most promising leaf node. Starting from the root node, this method keeps calculating the action value U from its child states, choosing the child state with highest (Q + U), moving to that state, and repeating this process until reaching a node where no action value U can be calculated from its child state. This method then returns this node as the leaf node. One point that needs to be mentioned is, this method only returns one node; it is not responsible for returning all of the leaf nodes.

The **BackFill** method backfills the newly calculated values up through the tree. For every node it meets when it goes up, it increments the N value of that node by 1, increments the W value by the result of the value head, and based on the new N and W values, update the Q value using W/Q.

5.3.4 Memory

The Memory object is responsible for preserving the playing histories that the algorithm uses to retrain the residual neural network. Specifically, it stores the following values. A 3-dimension board, a correspondent action made by previous game, a scalar value indicates if that board finally won the game, and a value indicates the play turn.

In order to make the size of the Memory object controllable, and since the more recent training records are more useful than the earlier ones, this project uses the Python deque to design the Memory class. Suppose the system sets the upper bound of the Memory size to be 30000 records, then if the total size of the records is larger than 30000, it will only preserve the most recent 30000 records.

5.3.5 Archive

The Archive is a folder that stores the different versions of the machine player. Each time after a player has trained its network model, it then starts an n-time competition with another player that still uses the old network model. If the player with new model wins 1 time more than the player with old model, the new model's weights is then stored inside of the Archive folder. And the next game will start using the new network model. The purpose of storing different versions of the models is that, after collecting enough models, the system can let them play against each other to check if there is a real improvement of the performance during the training.

Chapter 6

EVALUATION

The goal of this project is to train a machine Gomoku player without help from prior human knowledge. After training it for one month, by letting different versions, produced from different training iterations, play against each other, a performance improvement is found. The later the versions are collected, the better their performance, and the more games they win. The purpose of this chapter is to explain the experimental process, analyze the results, and finally select a few examples for in-depth discussion.

6.1 Training

6.1.1 Environment and Configuration Setting

This project is trained on the Massively Parallel Accelerated Computing (MPAC) lab of the Cal Poly Computer Science and Software Engineering Department. The MPAC lab machines contain an NVIDIA GeForce GTX 980 GPU. Since this project uses Keras and Tensorflow to build the residual network model, a machine with GPU acceleration can significantly speed up the network model's training and prediction process.

The experiment's configuration setting is shown in Table 1. It contains three sections, self play, retraining, and evaluation.

Self play	
EPISODES	50
MCTS_SIMULATIONS	20
MEMORY_SIZE	20000
CPUCT	1
EPSILON	0.2
Retraining	
BATCH_SIZE	256
EPOCHS	1
REGRESSION_CONSTRAINT	0.0001
LEARNING_RATE	0.1
MOMENTUM	0.9
TRAINING_LOOPS	10
Evaluation	
EVALUATION_EPISODES	20
SCORING_THREHOLD	1.5

Table 1. The experiment configuration setting.

The self play section indicates in each playing iteration, the system holds n times of the game, to let two players play against each other, and to collect the training data.

1. EPISODES means the number of games per iteration.
2. MCTS_SIMULATIONS means the number of simulations per each movement decision.
3. MEMORT_SIZE means the largest number of records that can be stored for future training.
4. CPUCT and EPSILON are constraints in the action value U's calculation.

The retraining section indicates the residual network model's retraining process.

1. BATCH_SIZE means the total number of randomly selected records per epoch.
2. EPOCHS means the number of a complete pass through the training dataset; in this case, EPOCHS 1 means the total number of records passed through a complete training iteration is the number of BATCH_SZIE.
3. REGRESSION_CONSTRAINT, LEARNING_RATE, and MOMENTUM are values used in the SGD optimization method, whose purpose is to minimize the difference between the network models' output value and the target value.
4. TRAINING_LOOPS means the number of trainings performed by each retraining process.

The evaluation section indicates the process in which the system holds competitions between two versions to check which one is better.

1. EVALUATION_EPISODES is the number of competitions per evaluation.
2. SCORING_THRESHOLD represents the criteria that determine the better player. For example, if the SCORING_THRESHOLD is 1.5, then in order to determine if one version is better than the other, its winning times must be larger than or equal

to the other player's winning time multiplied by 1.5; otherwise, no decision can be made.

6.1.2 Training Strategy

To summarize the configuration setting, the whole training process can be explained as follows.

1. Randomly initializing two residual neural network models with the same architecture, creating two agents, and assign each of them a network model.

2. Letting the two agents play against each other with the help of MCTS. Each movement is based on 20 simulations. During the play, the records are stored, the maximum number of the records is 20000. The record flow control is based on first-in, first-out (FIFO).

3. After every 50 games, one network model will retrain itself from the collected data. The total number of records trained by the model is 10*256 = 2560. The system then starts an evaluation stage, holding 20 plays between the player with new model and the player with old model.

4. At the end of the evaluation, if the winning times of the new model are 1.5x greater than the winning times of the old model, the next self play stage will discard the old version and use the new model. The new model's weights are preserved in the Archive folder for future use. Otherwise, if no winning model is decided, the new model will be discarded and won't be preserved, the next self play will still use the old model.

5. Repeating the process from 2 to 4.

6. After having 150 versions of the player in the Archive folder, the system selects 30 of them, let's each of them to plays against others, and records the result for analysis.

6.2 Results

The following results are collected from a one-month training system. The results show that the training is promising. On the one hand, the residual model keeps refining its prediction, on the other hand, the players trained late show better performance.

6.2.1 Loss

As mentioned before, the target of the network model training is to minimize the error between the actual winner and the value head's predicted winner, and to maximize the similarity between the actual movement and the policy head's predicted movement. Figure 6.1 shows the loss curves of the value and policy heads during the 1-month training. The top line is the error in the value head, which is calculated by the mean square error (MSE), the bottom line is the error in the policy head, which is calculated by the cross entropy. These lines are drawn based on the statistical data from 150 iterations, each iteration consists of 50 games, and the value shown in Fig. 1 is the average value of those 50 games. The average line of the value and policy errors is shown in Figure 6.2. As shown in Figure 6.1 and Figure 6.2, after experiencing a short period of rapid decline in the first iteration, the loss enters a slow but steady descending channel. The portion around version No.120 is an exception where the policy loss increased, but according to the average loss, the global trend is still going down.

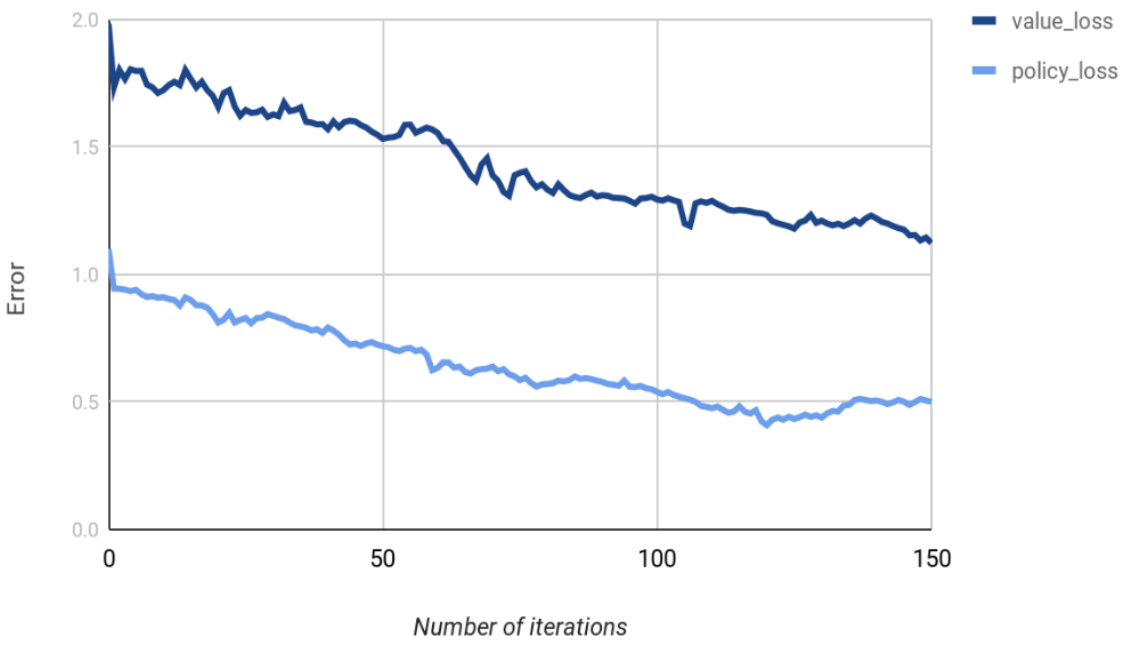

Figure 6.1. The value and policy losses

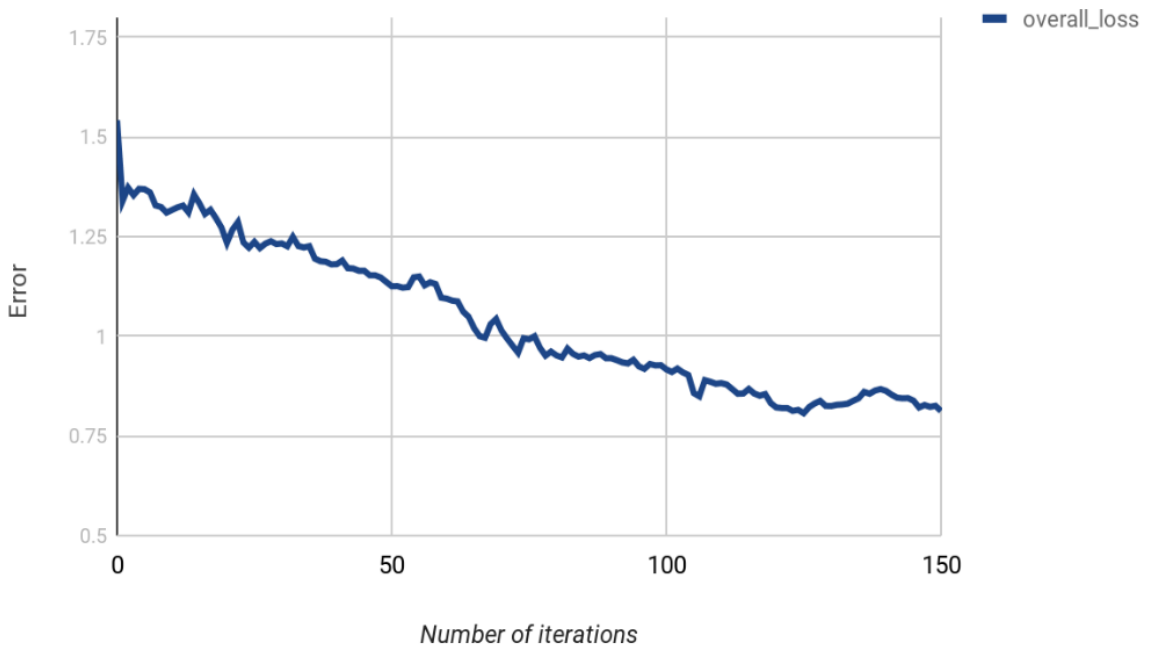

Figure 6.2. The overall loss

The policy and value losses have a double meaning. On the one hand, the large loss means the network model still cannot accurately simulate the process of the game, while the small loss means the model and the game process achieves some degree of synchronization. However, on the other hand, the large loss also means the game is open, which has too many possibilities to be predict accurately, while the small loss means the openness of the game is compressed to a smaller threshold, thus it is easier to be predicted.

Based on this understanding, Figure 6.1 and Figure 6.2 cannot be understood intuitively, they need dialectical analysis. Although high loss is not acceptable, a very low loss is also not a promising phenomenon; a good loss value should be able fluctuate within a relatively low range. Specifically, this analysis can be described as follow. In the first several iterations of training, the ability of the game engine is weak, while the network model's predictive behavior is still largely dominated by randomness. This leads to an unpredictable game process and ends up with a high loss. Once the game engine learns some knowledge, its movement decisions are more likely to be predicted, thus lowering the loss value. However, this also limits the game's openness, and it is not

conductive to the improvement of the engine's ability in the long run. But as the engine learns more knowledge, the reinforcement learning framework will reopen the game's open interval and increase the loss. In this way, the improvement process of the engine's ability can be recognized as a competition between the openness of the system and the accuracy of the network model.

6.2.2 Result of Competitions

After one-month training, 150 different versions of the network model are collected, the system selects 15 of them, and lets them play against each other. Each pairing plays 10 times, with both players having an equal chance to play first. The result is shown in Table 1, the number of ties is not counted. Figure 6.3 shown the total winning times for each of these versions. Clearly, as the training continues, the later versions of network model achieves better performances than the earlier versions, winning most of their games. Also, as the line in Figure 6.3 doesn't show a downward trend, the training is still far from complete.

	1	10	20	30	40	50	60	70	80	90	100	110	120	130	140	150	total
1		3	0	2	1	1	2	0	1	0	0	0	0	0	0	0	11
10	7		3	2	2	0	2	2	4	1	0	0	1	1	0	0	25
20	8	5		3	3	2	0	5	4	2	3	0	1	0	1	1	38
30	5	5	2		0	0	2	0	0	0	0	0	0	0	0	0	14
40	7	8	5	2		4	0	0	2	0	0	1	0	0	0	0	29
50	9	6	8	3	6		2	3	0	1	0	0	1	2	0	0	41
60	5	6	5	3	9	7		0	2	2	0	2	1	0	0	0	42
70	7	6	3	4	4	4	9		2	2	2	1	0	0	0	0	42
80	8	5	4	8	7	7	4	1		0	0	1	2	2	2	0	51
90	10	8	6	5	6	6	4	0	7		5	3	2	1	1	1	65
100	10	10	4	6	10	8	7	5	6	0		1	1	1	0	1	65
110	10	10	9	5	8	5	5	7	7	4	9		2	3	2	1	87
120	10	8	9	9	9	6	8	7	7	8	6	5		3	0	1	96
130	10	7	9	10	10	6	10	5	8	8	6	2	6		2	2	101
140	10	10	8	9	8	7	9	9	7	9	8	6	7	3		3	105
150	10	10	9	9	10	9	8	8	8	9	9	7	6	3	5		110

Table 2. The results of the competitions

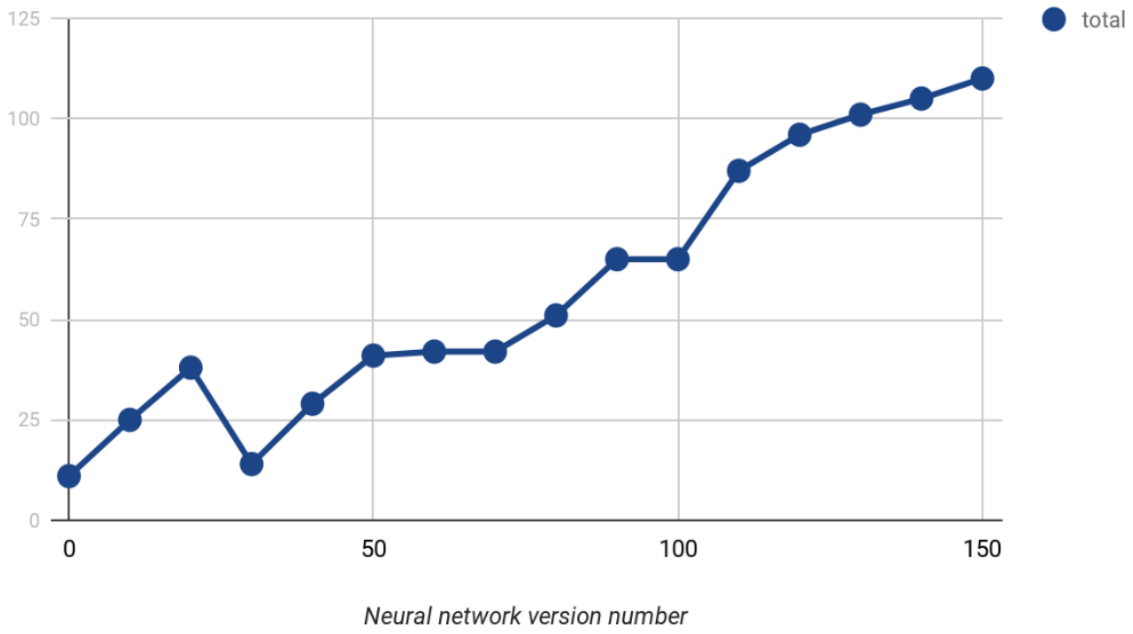

Figure 6.3. The number of wins of different versions

There are some points that need to be mentioned.

Firstly, although the rising trend of performance is continuous and stable, it doesn't mean that for two adjacent models, the latter must be stronger than the previous one. For example, as shown in Table 2, in the competition of version No.70 and version No.80, the later version wins once, while the earlier version wins twice. However, as the training continues, the later versions still show a better ability than version No.70.

Secondly, the rising trend of the performance is slow. Even if most of the time, the models are defeated by their immediate followers, the decisive performance superiority can only be shown after long training. For example, in Table 2, version No.20 is an early player with a very poor performance, and it is defeated by most versions. However, it wins once when competing with version No.150. This is surprisingly since version No.150 is the best player so far.

Thirdly, since the decisive superiority can only be shown in long term, it can be concluded that among these 150 versions, the basic moving strategy doesn't change much,

the progress of the performance is based on fine-tuning the details. This explains why version No.20 can still occasionally defeat No.150.

As a conclusion, this training achieves very good results. It shows that without help from prior human knowledge, the system still has the ability to improve itself, slowly but steadily learning how to play a good Gomoku game. However, this training is far from complete; it needs longer training times until the decisive superior performance emerges.

6.3 Examples

This section uses examples from three different versions of the player, No.10, No.40, and No.150, to demonstrate the learning process and the performance improvement in training, in an intuitive way.

6.3.1 Player with Network Model Version No.10

			o	x	o	x	o
x	o	x	o	x	o	x	

Figure 6.4. Board from the competition between two players with version No.10.

Figure 6.4 shows a board from the competition between two players with version No.10. Those players are not smart at all. It seems that their game strategy is to fill up the board from bottom to top. This strategy is clearly shown in Figure 6.5, which is the heat map of the statistical information before making the first step of the game.

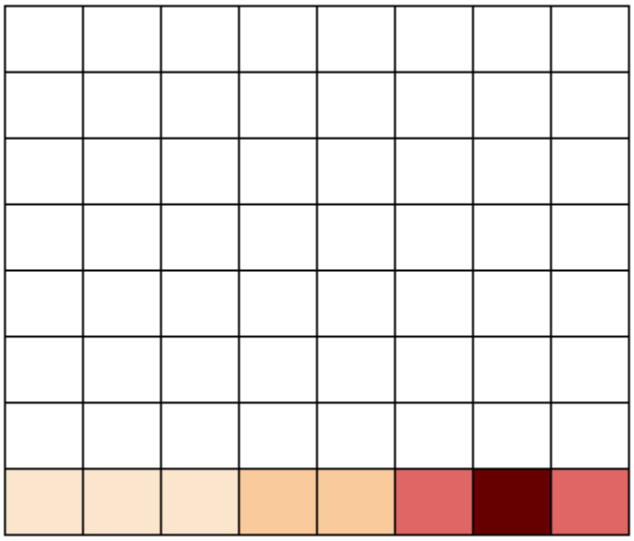

Figure 6.5. The heat map of the statistical information from the competition between two players with version No.10.

In Figure 6.5, the second square from the right in the last row has the highest (Q+U) value, therefore a piece is placed on it, as shown in Figure 6.4. The distribution of the statistical information shows that the network model of this version considers the squares in the current row have higher probabilities than squares in other rows. Meanwhile, in the same row, the model gives the squares close to the current action higher scores than the squares far away from it. This explains the bottom-up strategy where pieces will be placed near the current piece in the same line, once that line is full, it then moves up a row.

There is no intelligence in this strategy, even if one side wins in the end, it is only just a matter of chance, as shown in Figure 6.6. The player with piece X defeats the other one, just because it goes first, and has a better luck.

						X	O
X	O	X	O	X	O	X	O
X	O	X	O	X	O	X	
X	O	X	O	X	O	X	
X	O	X	O	X	O	X	

Figure 6.6. Player with piece ‘X’ defeats the other one. Both players have network model version No.10.

6.3.2 Player with Network Model Version No.40

As the training continues, the network No.40 considers the square one row up to the current piece should get higher scores. As shown in Figure 6.7, this leads to a column-based bottom-up strategy. The common point of version No.10 and No.40 is, there is no intelligence.

			X				
			O				
			X				

Figure 6.7. Board from the competition between two players with version No.40.

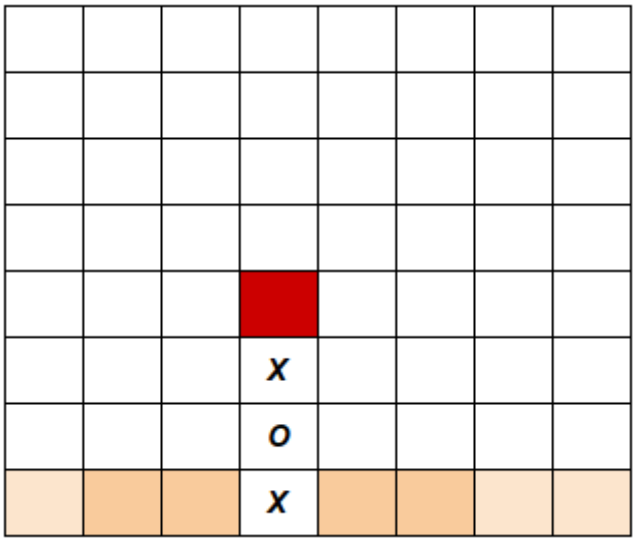

Figure 6.8. The heat map of the statistical information from the competition between two players with version No.40.

However, version No.40 is still important. It shows that during the reinforcement training, the system slowly finds that it is more likely to win the game by putting the piece up a row instead of placing it to the left or right. This strategy is completely derived from the old strategy. As Figure 6.8 shows that, except the highest square, the majority of the potential actions still locate at the bottom row.

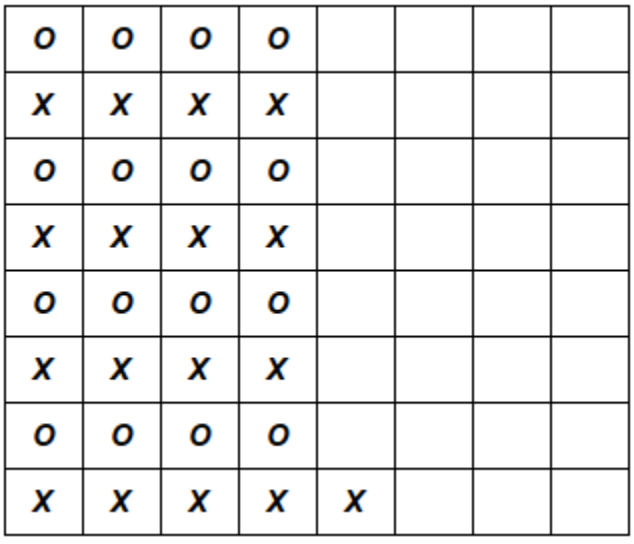

Figure 6.9. Player with piece ‘X’ defeats the other one. Both players have network model version No.40.

The winning board from the new strategy looks not smart at all, as shown in Figure 6.9. However, it is more likely that the new strategy will win the game if it plays against the old strategy player, as shown in Table 2. Thus it can be regarded as a better strategy.

6.3.3 Player with Network Model Version No.150

Version No.150 is a huge leap, compared to the previous models. There is no row-based or column-based bottom-up strategy anymore; as shown in Figure 6.10, it looks like it is played by human. Compared with the boring games played by versions No.10 and No.40, the version No.150 seems to have some kind of intelligence.

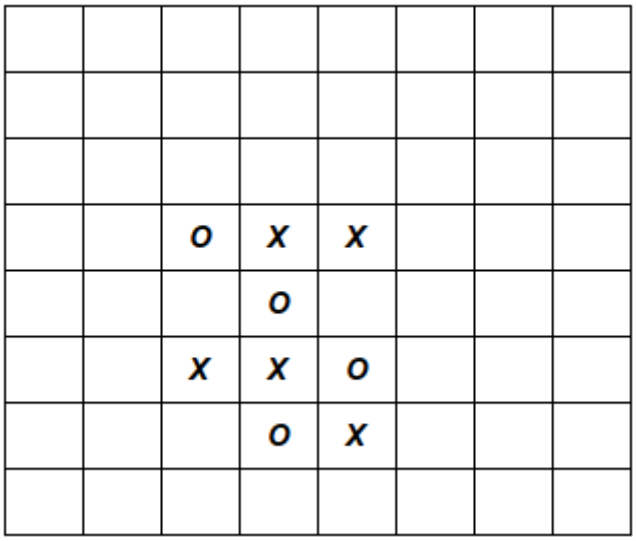

Figure 6.10. A board from the competition between two players with version No.150.

The reason of the intelligence is shown in Figure 6.11, which shows the most promising actions by the player O. Remember that the potential actions returned by the versions No.10 and No.40 were at the bottom of the board, with an exception for the No.40 where there is an potential action above the current piece. Figure 6.11 has a different story. Compared with Fig. 6.10, all of the potential actions are at the top, bottom, left, or right side of the current O pieces. This phenomenon shows the network model

finally gave up its preference the board's bottom, and gave preference to the squares around the current piece.

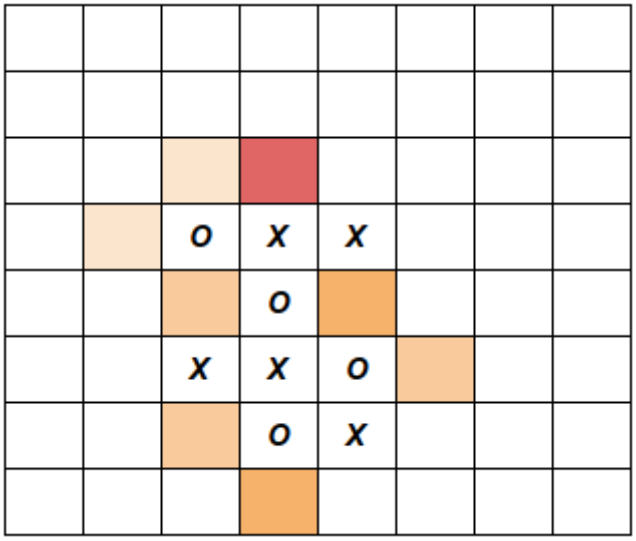

Figure 6.11. The heat map of the statistical information from the competition between two players with version No.150.

The version No.150 has its own limits. This model completely ignores the potential values of the squares at the top right, top left, bottom right and bottom left side of the current piece. For instance, in Figure 6.10, player O is only one step away from winning, since it already connected 3 pieces in a line. If it further connects the line into a 4-pieces line, nothing can stop it from winning. However, this won't happen, since this 3-piece line is a diagonal line, and the model doesn't consider the squares on the diagonal direction. As shown in Figure 6.12, it finally chose the square with the highest value. A human player will never make this move. Thus for player O, a great chance is lost.

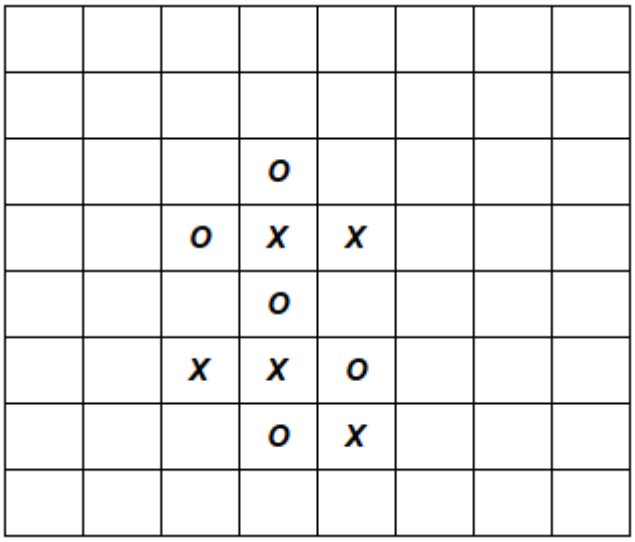

Figure 6.12. A successor board for the board shown in Figure 6.10.

Chapter 7

FUTURE WORK

Starting from scratch, this project eventually trained a machine player with some intelligence. This proves that, without the help of prior human knowledge, the machine player is still able to learn by itself, and finally get a certain level of intelligence. However, for the strategy used by this project, there is room for further improvement, and more methods can be used to better evaluate this project's result. Below are a few of things that can be done in the future.

7.1 Longer Training Time and Better Machine

With a longer training time and a machine with a stronger GPU, this project could train the machine player to become more intelligent. The current project used a machine with one GPU and trained for one month. Based on the result, these 150 versions of player show a slow, but steady increase of performance. According to those loss curves shown in Fig. 1, the losses will keep going down if the training continues. Also, with a machine with a stronger GPU support, the speed of the training process will be faster, and one month could result in a significantly more than 150 iterations, which means better players would be achieved during the limited training time.

7.2 Different Symmetric State

An important strategy of this project is to generate several similar boards automatically after each action is made, thus improving the variety of the training dataset. However, the details of this strategy can be improved. Currently, after creating a new board, only its horizontally symmetric board is generated, thus one board is expanded into two boards. The result of this is that the player picked a bottom-up fill up game strategy, which means

it only considers the squares located in the current row, either on the same side of the current piece, or on the horizontally symmetric side. Meanwhile, the other locations are ignored. Although the game strategy is improved through training, the latest version player still shows a favor for bottom-up.

Regarding this situation, after each action, more boards need to be generated. For example, the system should not only generate the horizontally symmetric board, but also the vertically symmetric board and diagonally symmetric board. It is likely that, based on this new symmetric strategy, when the system is choosing the next action, the number of potential squares would be significantly increased and more diverse game strategies would be learned. This may cause the intelligence to appear in a shorter time.

7.3 Different Hyper Parameters

This project consists of three sections: self play, retraining, and evaluation. The hyper-parameters of these three sections are shown in the last chapter. These parameters turned out to be useful, but it is also helpful to try other sets of the hyper-parameters. For example, currently, in the retraining process, the loss of the Resnet model is calculated by the policy head's result * 0.5 plus the value head's result * 0.5, which means both policy and value heads have the same weights in training. Those hyper parameters are set by default, and although they achieved good result, it is not guaranteed that they are the best ones.

To improve this, this project can be run with different sets of hyper-parameters, choosing the one with the best result. However, this will require lots of resources, since each try exhausts a powerful machine for several months.

7.4 Compared with the Machine Player Trained by Human Knowledge

According to [7], the training strategy without the help of human knowledge achieved much better performance than the strategy with human knowledge. For the Gomoku

game, we can also build a training system that is based on human expert data. Once the no-human-knowledge player is trained smart enough, we can compare the performances between these two players. However, this won't happen in the near future, according the current configuration, the machine player still needs a long training time to be able to have enough intelligence and performance.

7.5 Same Strategy, Different Games

Although the current machine player is only able to play Gomoku, with a few modifications of the training strategy, machine players for other games can also be trained. Specifically, since the current training strategy doesn't require human plays as the training data, the only difference between different checkerboard games is their rules. In other words, the differences are the rule that determines when the MCTS should regards the current board as a terminal state, and the rule that controls the movement of the pieces on the board. According to this, by just changing the winning rules, the Gomoku system can be easily converted into a Go system, a Chess system, and so on.

Chapter 8

CONCLUSION

This project demonstrates that, by starting from the scratch, it is possible to design an AI machine player that is able to keep refining itself without any prior human knowledge. For evaluation purposes, 3 of 150 versions of machine players produced by this project are analyzed. Based on this analysis, it is clear that the machine player evolves the game strategy from a row-based bottom-up, to a column-based bottom-up, and finally, to a strategy which favors its surrounding squares. Along with this game strategy evolution, the appearance of the game becomes more and more intelligent. Moreover, in the one month training process, the performance improvement of the machine player didn't stop, which means it is very likely that it can be further improved by continuing the training process.

BIBLIOGRAPHY

[1] C. B. Browne et al., *A Survey of Monte Carlo Tree Search Methods*. IEEE Transactions on Computational Intelligence and AI in Games, vol. 4, pp. 1-43, 2012

[2] Z. Tang, D. Zhao, K. Shao, L. Lv, *ADP with MCTS algorithm for Gomoku*. IEEE Symposium Series on Computational Intelligence, pp. 1-7, 2016

[3] M. Enzenberger, M. Muller, B. Arneson, R. Segal, *Fuego --- An Open-Source Framework for Board Games and Go Engine Based On Monte Carlo Tree Search*. IEEE Transactions on Computational Intelligence and AI in Games, vol. 2, pp. 259-270, 2010

[4] D. Silver et al. *Mastering the game of Go with deep neural networks and tree search*. Nature 529, pp. 484-489, 2016

[5] G. Chris Lee, K. Haung, C. Sun, Y. Liao, *Stem cell detection based on Convolutional Neural Network via third harmonic generation microscopy images*. IEEE Orange Technologies, pp. 45-48, 2017

[6] K. He, X. Zhang, S. Ren, J. Sun. *Deep residual learning for image recognition.* IEEE Computer Vision and Pattern Recognition, pp. 770-778, 2016

[7] D. Silver et al. *Mastering the game of Go without human knowledge*. Nature 550, pp. 354-358, 2017

[8] V. Mnih, et al., *Playing Atari with deep reinforcement learning*. arXiv preprint arXiv:1312.5602, 2013

[9] M. Enzenberger, M. Muller, *Fuego - An Open-source Framework for Board Game and Go Engine Based on Monte-Carlo Tree Search*. IEEE Trans. Comput. Intell. AI in Games 2, pp. 259-270, 2010.

[10] P. Baudis, J. Gailly, *Pachi: State of the Art Open Source Go Program*. ICGA Jan. 30, pp. 198-208, 2007.

[11] Z. Tang, D. Zhao, K. Shao, L. Lv, *ADP with MCTS algorithm for Gomoku*. IEEE Symposium Series on Computational Intelligence, pp. 1-7, 2016

[12] L. V. Allis *Searching for Solutions in Games and Artificial Intelligence.* PhD thesis, Univ. Limburg, Maastricht, The Netherlands (1994)

[13] Coulom, R. *Efficient selectivity and backup operators in Monte-Carlo tree search.* In 5th International Conference on Computers and Games, 72–83 (2006)

[14] Tesauro, G. & Galperin, G. *On-line policy improvement using Monte-Carlo search.* In Advances in Neural Information Processing, 1068–1074 (1996)

[15] Sheppard, B. *World-championship-caliber Scrabble*. Artif. Intell. 134, 241–275 (2002)

[16] He, K. el al. *Identity Mappings in Deep Residual Networks*. ECCV 2016, p 630-645.