

**Department of Computer Science and Engineering**  
**The Chinese University of Hong Kong**

**Final Year Project Report**

**2016 – 2017**

**LYU1601**

**Intelligent Non-Player Character**  
**with Deep Learning**

Prepared by Meng Zhixiang

Supervised by Prof. Michael Lyu

**April, 2017**

***This page is left blank intentionally***

## **Abstract**

Deep Learning has been experiencing a burst of evolution recently, and been used to solve different kinds of problems. Existing game engines of Chinese chess, however, are mostly based on traditional searching approach and highly rely on hard-coded libraries of game records. In our project, we design and build a game AI of Chinese chess with Deep Learning that uses Policy Network to predict the probabilities of moves, Evaluation Network to evaluate chessboard statuses and Minimax Searching to select moves. Policy Network is further divided into Piece Selector to predict probabilities of selecting a piece, and Move Selector to predict probabilities of destination for that selected piece. Policy Network is trained by Supervised Learning based on the game records of master human players, and Reinforcement Learning by competing with itself. Evaluation Network is trained by Supervised Learning with the help of an open-source API. Minimax Searching is to combine the outputs of Policy Network and Evaluation Network to make a move selection. At last, our Game AI achieved 76% winning rate against amateur human players.

# Table of Contents

<b>Table of Contents .....</b>	<b>- 4 -</b>
<b>1. Introduction .....</b>	<b>- 7 -</b>
1.1. Motivation .....	- 7 -
1.2. Background .....	- 9 -
1.2.1. Development of AI in Go .....	- 9 -
1.2.2. Development of AI in Chess .....	- 11 -
1.2.3. Development of AI in Chinese Chess .....	- 12 -
1.3. Difference among Chinese Chess, Chess and Go .....	- 13 -
1.4. Objective .....	- 14 -
1.5. Definition of Terms .....	- 15 -
1.5.1. PGN .....	- 15 -
1.5.2. FEN .....	- 15 -
<b>2. Literature Review .....</b>	<b>- 17 -</b>
2.1. AlphaGo .....	- 17 -
2.2. Predicting Moves in Chess using Convolutional Neural Networks .....	- 20 -
2.3. Giraffe: Using Deep Reinforcement Learning to Play Chess .....	- 21 -
<b>3. Methodology .....</b>	<b>- 23 -</b>
3.1. Supervised Learning .....	- 23 -
3.2. Convolutional Neural Network .....	- 25 -
3.3. Softmax .....	- 28 -
3.4. Minimax Searching .....	- 29 -
3.5. Reinforcement Learning .....	- 31 -
3.6. TensorFlow .....	- 33 -
3.7. Aliyun .....	- 34 -
<b>4. Design .....</b>	<b>- 35 -</b>
4.1. Project Workflow .....	- 35 -
4.2. Game Engine Overview .....	- 37 -
4.3. User Interface .....	- 39 -
4.3.1. Game-Related Functions .....	- 40 -
4.3.2. Communication Functions .....	- 41 -
4.4. Game AI .....	- 42 -
4.4.1. Structure Overview .....	- 42 -
4.4.2. Piece Selector and Move Selector .....	- 45 -
4.4.2.1. Model Structure Design .....	- 47 -
4.4.2.2. Features Extraction .....	- 48 -
4.4.3. Evaluation Model .....	- 53 -
4.4.3.1. Model Structure Design .....	- 53 -
4.4.3.2. Feature Extraction .....	- 54 -

4.4.4.	<i>Decision Maker</i> .....	- 56 -
4.5.	Connection between Frontend and Backend.....	- 57 -
<b>5.</b>	<b>Implementation and Development Process .....</b>	<b>- 58 -</b>
5.1.	Project Schedule .....	- 58 -
5.1.1.	<i>Schedule in First Term</i> .....	- 58 -
5.1.2.	<i>Schedule in Second Term</i> .....	- 59 -
5.2.	Summary of Different AI Models .....	- 60 -
5.3.	Summary of Different Selection Strategies .....	- 61 -
5.4.	Simple AI with Monte Carlo .....	- 62 -
5.5.	Policy Network Model.....	- 67 -
5.5.1.	<i>Previous Design</i> .....	- 67 -
5.5.2.	<i>Model Structure</i> .....	- 71 -
5.5.3.	<i>Selection Strategy</i> .....	- 76 -
5.6.	AI Model Version 018.....	- 84 -
5.7.	AI Model Version 018.1.....	- 86 -
5.7.1.	<i>Evaluation Model</i> .....	- 88 -
5.7.1.1.	Neural Network Structure .....	- 88 -
5.7.2.	<i>Selection Strategy</i> .....	- 90 -
5.8.	AI Model Version 018.2.....	- 93 -
5.8.1.	<i>Selection Strategy</i> .....	- 95 -
5.9.	Other Improvements .....	- 99 -
5.9.1.	<i>Deploy Web Server</i> .....	- 99 -
5.9.2.	<i>Multiple Login</i> .....	- 100 -
5.9.3.	<i>Reducing Responding Time</i> .....	- 101 -
5.9.4.	<i>Auto Training</i> .....	- 102 -
<b>6.</b>	<b>Training Process.....</b>	<b>- 103 -</b>
6.1.	Supervised Learning of Policy Network.....	- 103 -
6.1.1.	<i>Training Dataset</i> .....	- 103 -
6.1.2.	<i>Preprocessing</i> .....	- 104 -
6.1.3.	<i>Chessboard Flipping</i> .....	- 106 -
6.1.4.	<i>Training Strategy</i> .....	- 107 -
6.2.	Reinforcement Learning of Policy Network .....	- 108 -
6.2.1.	<i>Training Dataset</i> .....	- 108 -
6.2.2.	<i>Training Strategy</i> .....	- 108 -
6.3.	Supervised Learning of Evaluation Network .....	- 110 -
6.3.1.	<i>Training Dataset</i> .....	- 110 -
6.3.2.	<i>Training Strategy</i> .....	- 110 -
<b>7.</b>	<b>Results.....</b>	<b>- 112 -</b>
7.1.	Results of Supervised Learning .....	- 112 -
7.1.1.	<i>Accuracy Testing</i> .....	- 112 -
7.1.1.1.	Piece Selector .....	- 112 -
7.1.1.2.	Move Selector .....	- 116 -

7.1.2.	<i>Real Performance Testing</i> .....	- 117 -
7.1.2.1.	Game-Playing Case 1 .....	- 118 -
7.1.2.2.	Game-Playing Case 2 .....	- 122 -
7.1.2.3.	Game-Playing Case 3 .....	- 126 -
7.1.2.4.	Game-Playing Case 4 .....	- 131 -
7.1.2.5.	Game-Playing Case 5 .....	- 133 -
7.1.2.6.	Game-Playing Case 6 .....	- 142 -
7.2.	Results of Reinforcement Learning .....	- 146 -
7.3.	Results of Final Model .....	- 148 -
7.3.1.	<i>Data Collected via Web Server</i> .....	- 148 -
7.3.2.	<i>Testing Case 1</i> .....	- 149 -
7.3.3.	<i>Testing Case 2</i> .....	- 152 -
<b>8.</b>	<b>Contribution</b> .....	<b>- 156 -</b>
<b>9.</b>	<b>Discussion</b> .....	<b>- 158 -</b>
9.1.	The Difficulties in Our Project .....	- 158 -
9.2.	The Reason to Use Neural Network .....	- 162 -
<b>10.</b>	<b>Conclusion</b> .....	<b>- 163 -</b>
<b>11.</b>	<b>Future Work</b> .....	<b>- 164 -</b>
<b>12.</b>	<b>Reference</b> .....	<b>- 165 -</b>

# 1. Introduction

## 1.1. Motivation

Artificial Intelligence (AI), especially machine learning has been experiencing a burst of evolution in recent years, as the computing capability of computers has increased a lot so that the computations required by machine learning approaches are achievable using much shorter time. Among all, Google's AlphaGo is a good example. AlphaGo is a game AI that plays GO, and it beat Lee Sedol, one of the top-class professional players, in a five-game match with the score of 4-1 in March 2016. It's a surprising but expected result, which shows the powerfulness of AlphaGo, and more importantly, the great potential of machine learning. It uses a method called deep learning, which uses Neural Network (NN) to search for the best option for current situation.



*Figure 1.1. AlphaGo Playing against Lee Sedol*

Currently, many people are trying to use deep learning to solve different kinds of problems, including building game AI for different games. While most of them focus on Go or chess, none has ever applied the approach of deep learning to Chinese chess. Chinese chess is one traditional strategy game, which is still very popular nowadays. Nowadays, the existing game engines of Chinese chess are all based on searching

approach without using machine learning and primarily rely on hard-coded libraries of the initial phases and the final phases of games to make move choice. As the approach of deep learning has been used on many fields, like GO and chess, however, the field of Chinese chess remains blank. Therefore, we tried to build a game AI for Chinese chess, using deep learning.



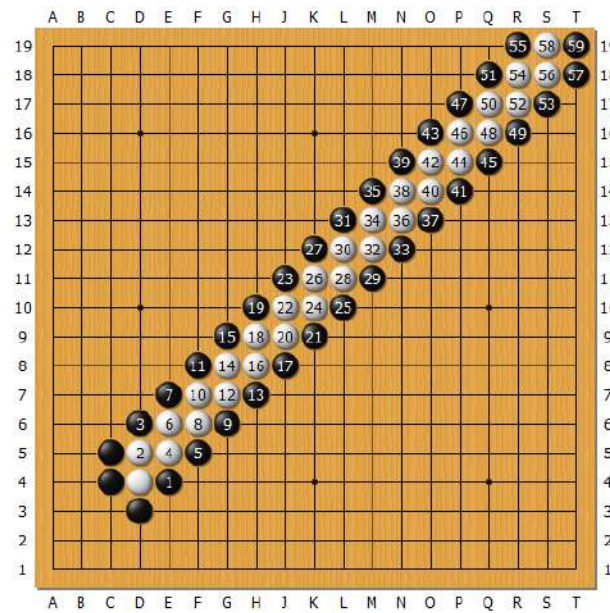
## 1.2. Background

### 1.2.1. Development of AI in Go

In Go, the size of game board is  $19 \times 19 = 361$ , which is much larger than other games. For example, there are  $8 \times 8 = 64$  available positions in chess, and there are  $9 \times 10 = 90$  available positions in Chinese chess. In a recent research, the number of legal positions on a game board of Go is  $2.801682 \times 10^{170}$ . [18] The number is 1090 times larger than the number of atoms in the universe. It means the complexity is much larger than other chess games, like chess and Chinese chess. So, direct searching approach is not applicable for Go because searching is slow and limited in global area where the depth of searching may be too large.

In the beginning process of development history of AI, however, there were no better algorithms to search and evaluate the game board. All they could do was to modify the evaluation function and pruning condition. As Go uses a really big game board, players are required to have the ability to judge the current situation (the difference of areas controlled by players). But in a game, the ownership of one place may be fuzzy and hard to decide even for human players, and for computer program at that time, it made lots of mistakes and couldn't be used for a high-level AI. So, for the scientist that time, building an AI to overcome top-class human players seems impossible.

To solve the problem, Monte Carlo Tree Search (MCTS) was introduced in this field. To explain what MCTS is, imagine that a person who is absolutely a beginner and knows nothing about Go, and let him choose a place randomly. Then repeat the process and calculate the winning rate of every possible move. However, simple randomization is not suitable for complex board games. For instance, in Go, there exist situations where there may exist many legal moves but only few among them are reasonable.



*Figure 1.2. An Example of "Ladder" In Go*

As shown in Figure 1.2. above, what the black side does is called “ladder”. The black side forces those white stones to move like zigzag, and finally can capture them all. During this period, the black side must put its stones in correct points as indicated in Figure 1.2., or the white side can escape, which is a common sense for Go players. This is easy for human players but not for a computer program.

In 2006, the invention of UCT (Upper Confidence Bound 1 applied to trees), an improved version of MCTS, changed this status. UCT would prefer a known better move with higher winning rate other than select them completely randomly. By this improvement, the efficiency of searching had been growing fast. In 2006, the level of the best AI that time had only k level, below the average level of amateurs. But in 2012, Zen, a Go engine using MCTS, beat top-class professional player at four stones handicap, which means it could win against nearly half of amateurs.

However, MCTS also has its own limit in global view though it is good at local battle. The level of program hardly improved until 27 January 2016, the day when the paper about AlphaGo was published on Nature. AlphaGo had beat Fan Hui, a 2-dan pro, in 5-0 complete victory, which is the first victory between Go program and professional Go

players in equal condition. This revolutionary improvement could attribute to the use of neural network. The detail of the algorithm of AlphaGo can be found in the literature review. To be brief, neural network provides a faster way to evaluate the situation on board and to generate a quick predicted move, like what human players will do. Combining with the accurate calculation, distributed AlphaGo running on Google's cloud service wins all 500 games against "old" AI. And in March 2016, AlphaGo beat Lee Sedol, one of the top class professional players, in a five-game match with the score of 4-1 in March 2016. In an ELO-ranking website GoRatings, it is the second-best player in the world.

The success of AlphaGo has proved that it is possible for computer programs to beat human players in Go.

### 1.2.2. Development of AI in Chess

In chess, Deep Blue has done it long before. In May 1997, it beat Garry Kasparov with  $3\frac{1}{2}-2\frac{1}{2}$ . Nowadays, even top-level professional players have little possibility to win against AI running on normal computers as current personal computers have higher computation ability than Deep Blue.

But the AI of Deep Blue is different from AlphaGo. In fact, the hardware of Deep Blue consists of 30 paralleled CPUs and 480 specially made VLSI chips, meaning that the computer could only run chess program. But nowadays, the newest chess engine, Stockfish, can run on Windows machines, and beat any other players or AIs on personal computer with 4-cores CPU.

Nowadays, a normal game engine of chess or Chinese chess will contain searching part and libraries for opening and ending. In fact, though usage of these libraries is not necessary, it can improve the performance of AI greatly. This is because that the number

of branches in searching will increase greatly as the number of available moves increases. So, using hard-coding libraries will be a good choice. But if we use Neural Network, this should not be a problem. The best neural network AI, Giraffe, can reach the level of an FIDE International Master though Stockfish is still stronger.

### 1.2.3. Development of AI in Chinese Chess

However, those game engines in Chinese chess are still using traditional methods. In National Computer Games Tournament of 2016, Chess Nade (“象棋名手” in Chinese) won its fifth consecutive champion. And it is recognized as the best Chinese engine in China. The detail algorithm of it remains secret as it is commercial software. But we can infer that it still uses traditional method, including searching and pruning. Now, there is no any software using Neural Network in Chinese Chess. So, it is a blank field for us.

### 1.3. Difference among Chinese Chess, Chess and Go

The main difference between Chinese chess and Go is the way to make a move. In Go, players should put one stone into an empty position every turn, while in chess and Chinese chess, players should move a piece on the board following a set of rules depending on the type of the piece selected. And in chess and Chinese chess, pieces can be captured so that the number of pieces on the board will become less and less, leaving the possible moves of those remaining pieces become more and more. While in Go, the number of stones on the board will generally become more and more and the available positions to place a stone become less and less.

Besides, compared with chess, there are mainly two different points in Chinese chess. First, there are two fortresses and one river on the chessboard, restricting the move of certain types of pieces, like King, Bishop and Advisor. Second, there is a special type of pieces, called Cannon, which can capture only with exactly one piece in the middle but move like Rock. Therefore, certain considerations are necessary for these difference, compared with the methodologies used in previous researches about GO and chess.

## 1.4. Objective

Our objective is to implement a game AI which can play Chinese chess with human users, and the whole game system should have following components.

A user interface lets human players to play Chinese chess against our AI. It should be able to communicate with our AI, like sending information describing the chessboard status to the server and receiving move choice of our AI from the server. It should be able to judge whether every move is legal or not and decide if a player is checkmated.

A game AI makes moves against the opposite player based on the output of pre-trained NN model. It should be able to receive the message sent from frontend, preprocess it, then feed it into NN model to get a move choice, and at last send the choice back to frontend. It would be better if it is able to play Chinese chess with multiple users simultaneously and record game histories for further training usage.

A program trains NN model ahead to be used in our AI and saves the trained model. For our AI, it only restores the previously saved model to do calculation.

## 1.5. Definition of Terms

### 1.5.1. PGN

PGN is short for Portable Game Notation, which is one popular string format to record the game history for chess games. [1] The basic format for recording moves is simple: [one character to represent the type of selected piece] [the coordinates of the destination of this move]. Obviously, only a complete sequence of PGN starting from an initial chessboard status will make sense, as the original position of the selected piece in each move is not recorded.

Besides, there is a Chinese version of PGN to record Chinese chess games. The basic rationale is quite similar, with little difference. There are also other formats in English or Chinese to record games. The common problem of them, however, is as the same as stated above, which is that only the whole sequence together will make sense.

### 1.5.2. FEN

Chinese Name	帅/将	仕/士	相/象	傜/馬	炮/砲	俥/車	兵/卒
English Name	King	Advisor	Bishop	Knight	Cannon	Rock	Pawn
Symbolic Representation	K/k	A/a	B/b	N/n	C/c	R/r	P/p

*Figure 1.3. Symbolic Representation for Different Pieces*

FEN is short for Forsyth–Edwards Notation. Similarly, it is one standard string format representation of the chessboard status, using one letter to represent each type of chess pieces as shown in Figure 1.3. above.

We also made certain modifications for simplicity, like using ‘1’ to denote an empty position. Lowercase letters are to represent the pieces of upper-side player while

uppercase letters are to represent the pieces of lower-side player. FEN represents the whole chessboard row by row, with '/' as delimiter. At last, the player to make the next move is also declared in FEN, with 'b' for black side and 'r' for red side. The move is recorded using four digits, by combining the coordinates of both the original position and the new position of that piece. Clearly, FEN is much better than PGN for our NN training usage, as it contains complete information for every intermediate game status.

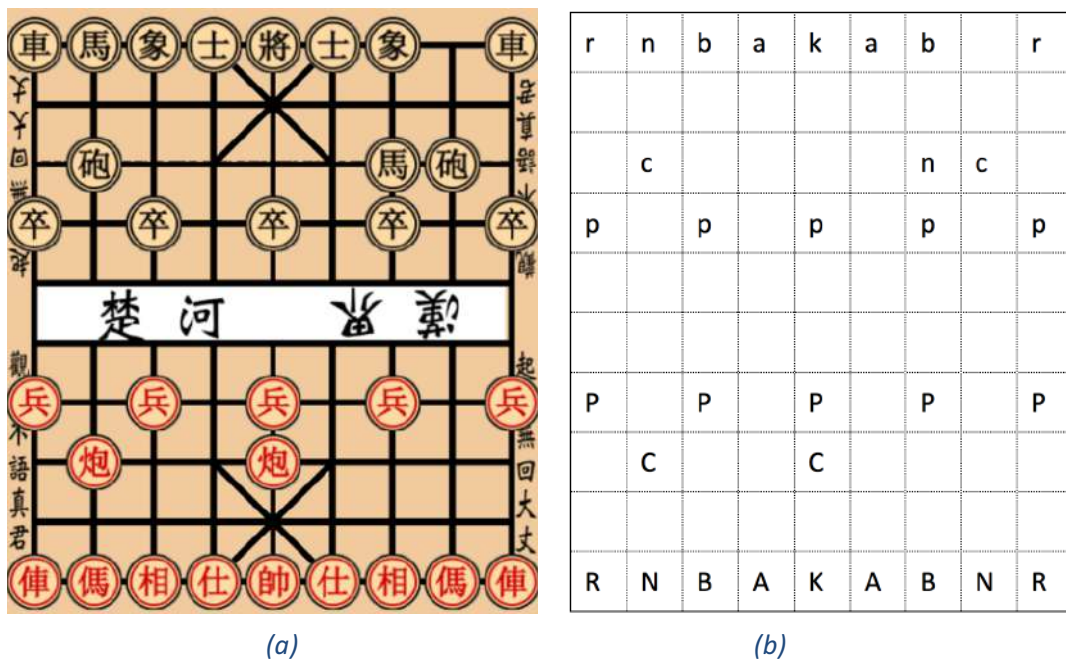


Figure 1.4. An Example of FEN Format

Here is an example of FEN. Picture (a) in Figure 1.4. is a chessboard status and Picture (b) is the chessboard after replacing real pieces with symbols. And the next move is the turn of the red side. The corresponding FEN representation of this chessboard status is:

“rnbakablr/11111111/1c1111nc1/p1p1p1p1p/11111111/11111111/P1P1P1P1P/1C11C1111/1111111111/RNBAKABNR, r”



## 2. Literature Review

### 2.1. AlphaGo

AlphaGo mainly contains four Neural Networks.

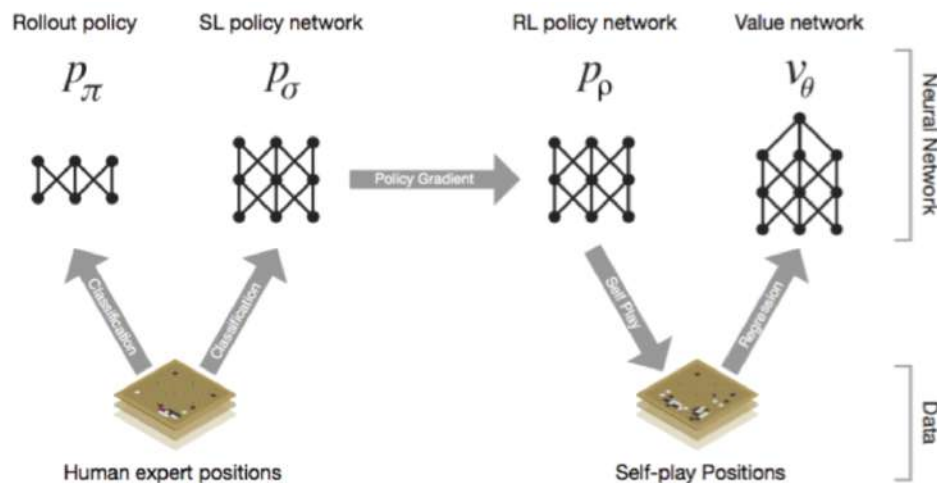


Figure 2.1. Neural Network of AlphaGo

In Figure 2.1., the left two networks learned from human experts, which use supervised learning to train.

Rollout Policy Network is a simple network that can deal with chessboard. It is similar as first impression of human players. It has relatively low accuracy about 24.2% in predicting human players' moves, mainly used for reducing the nearly impossible moves of searching tree. The mainly advantage of this neural network is that it can run faster which needs only 2 nanoseconds to select a move while SL Policy Network needs 3 ms to do that.

SL Policy Network also is used to predict the human player's move. However, as it is more complex in structure which have 13 layer and well-trained which uses 30 million positions to train, it has higher correct rate. For normal chessboard, the accuracy can reach 57.0%. However, it is not enough as it has only approximately 10% winning rate again traditional AI using MCTS.

The two networks on the right side are using Reinforcement Learning. RL Policy Network has same structure as SL Policy Network, but it continuously plays with itself, and makes improvement based on the result. After Reinforcement Learning, it has 80% winning ratio against previous version using supervised learning. Even if it does nothing search at all, it performs better than any other AI.

The last part is value network. Though the structure of this network is not very different from Policy Network, it only output a value representing the prediction of winning rate for one side. It uses the positions sampling from the self-playing game from RL Policy Network in order to prevent overfitting. Because if it uses normal games as training dataset, it would trace every move in a specific game and then record the result of the game instead of the stone distribution.

Besides Neural Networks, it also uses MCTS. Different from normal AI, with the help of Neural Networks, the single searching used by AlphaGo will start with using SL Policy Network predicting a chain of moves, and using the result from value network and rollout to improve it. The result will be the score of next move. After repeating this procedure for enough time, it will have a map of score for all possible next move and put next move according to it.

Figure 2.2. shows an example for how AlphaGo makes next move. The position is taken from the game with Fan Hui, and AlphaGo is on black side. In all of these subgraphs, the point with red circle is the best move according to the method it uses.

Figure a is representing the evaluation after next move using valuation network. Figure b is representing the result from searching where it uses only value network without rollout network. Figure c is representing the result from searching where it uses only rollout network without value network. We can notice that the result from MCTS would be different if the ratio between them are changed. In their practice, they discovered that a mixed version would have best level. Figure d is the result from SL Policy Network

directly. The first move chose by it is a move of middle level. Figure e shows the results from its search tree, and it will choose the move with the highest value. Figure (f) shows principal variation from search tree of AlphaGo. The number of sequence number means a most possible prediction about process of the game. Though Fan Hui's move is not the same as the prediction of AlphaGo, he admitted that moves suggested by AlphaGo would be better.

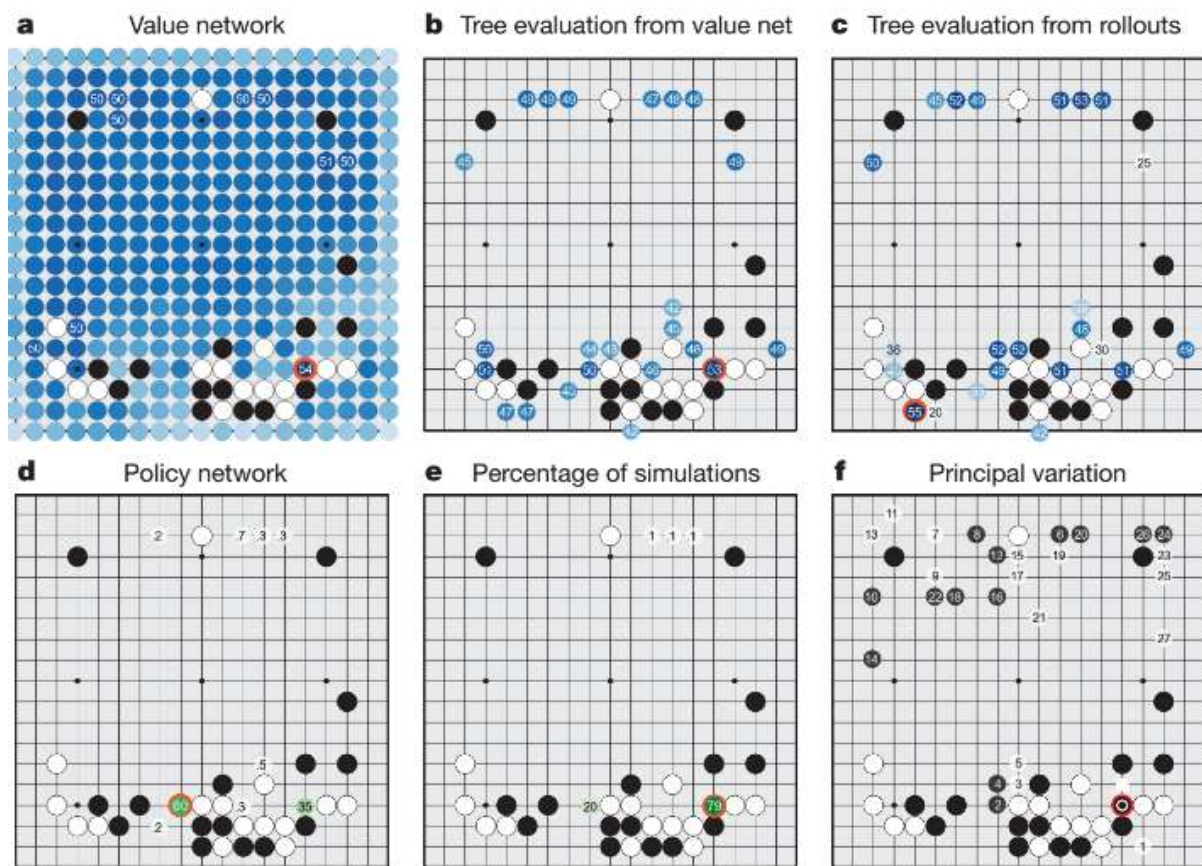


Figure 2.2. An Example of AlphaGo Making a Move

Due to the improvement above, AlphaGo has become the strongest AI in the world. Consisting of 1,202 CPUs and 176 GPUs, the distributed version of AlphaGo beat any other while a normal version using 48 CPUs, and 8 GPUs only lose one game in 495 games in total. Even with handicaps, it still had high winning rate against others.

Though the rules of Go and Chinese Chess are different, we can still learn from the method and ideas of building AlphaGo.

## 2.2. Predicting Moves in Chess using Convolutional Neural Networks

The work from Oshri, B., & Khandwala, N also uses convolutional neural network. As it is design for chess, we think it more helpful for our project because chess is a lot more similar from Chinese chess compared with paper about AlphaGo.

In their work, they mainly build policy neural network. And in predicting next move from human players, it reached the accuracy of 44.4%, which is pretty high. The success of their work proves that it is possible to use CNN to train an AI for playing chess.

In their thesis, the recognize reasoning of chess as kind of pattern recognition while traditional method only consists of searching and evaluation. And the way for the neural network to select a move is to separate a move into two parts: select a piece and move it to other places. And use piece selector and move selectors to solve the part respectively. This is different from AlphaGo, because of the difference between moving a piece in chess and putting a stone in Go.

However, the high accuracy in predicting next move doesn't mean that the program has high level of playing chess. In the 100 games with Sunfish, a famous chess engine, it loses 74 games and draws in the rest of game. In those draw games, this program play well in the middle game, and force opponents to make a draw. However, in the sparse ending game, it faces many troubles because patterns can't be found in that kind of position.

## 2.3. Giraffe: Using Deep Reinforcement Learning to Play Chess

In Matthew Lai's work, he implemented evaluation function of game engine based on neural network. We use his paper as reference on our Evaluation Network.

The feature of the inputs of neural network has following features.

- a) Side to Move – It is turn for black or for white.
- b) Castling Rights - Presence or absence of castling rights. Castling is a special rule for chess. In Chinese Chess, we don't need to consider it
- c) Material Configuration – Amount of each kind of pieces
- d) Piece Lists – for every piece, note their position coordinate, existence
- e) Sliding Pieces Mobility – for sliding piece, note how far they can move along a direction, and liberty of them.
- f) Attack and Defend Maps – for each square, note the attacker and defender with lowest value.

After determining these features, the author did not mix them directly because the connection between two features with long distance logically would have no benefits to the results. As a result, the last 2 layers are fully connected, while the prior one was trained separately.

For their training dataset, instead of using that collected on Internet. They added a random legal move to the board and used the processed one as training data. The reason of this process is to increase the variety of dataset, in order to help the neural network to evaluate the unseen situation.

Then the author used Reinforcement Learning to the neural network, and use TD-leaf algorithm. In each time of iteration, they use the network to move 12 moves, and trace on the move to see when the score of board will change, weighted by the distance from the beginning position.

Search Score	Score Change	Discount Factor	Contribution to Total Error
10	0	$0.7^0$	0
20	10	$0.7^1$	7
20	0	$0.7^2$	0
20	0	$0.7^3$	0
-10	-30	$0.7^4$	-7.2
-10	0	$0.7^5$	0
-10	0	$0.7^6$	0
40	50	$0.7^7$	4.12
40	0	$0.7^8$	0
40	0	$0.7^9$	0
40	0	$0.7^{10}$	0
40	0	$0.7^{11}$	0

Total Error: 4.10

*Figure 2.3. An Example of TD-leaf Searching Results*

In the sample graph as shown in Figure 2.3., the network used a discount parameter 0.7. The second move changed the score by 10, then its effect on Total Error is  $10 * 0.7^1 = 7$ . We can see that in this algorithm, if a move which will change the board is far away from now, it would have lower contribution. The algorithm is consistent with our common senses about chess.

The result of their Neural Network is remarkable. Their program, named Giraffe, have an evaluation function comparable to those of best chess engines worldwide, though evaluation functions of those engines are all designed and tuned by human over many years.

## 3. Methodology

### 3.1. Supervised Learning

Supervised learning is one of deep learning approaches, through which the NN model is trained by dataset with target labels. In supervised learning, examples in the training dataset are composed of inputs, usually representing features of objects to be learned, and target outputs. Generally speaking, the goal of supervised learning is to learn a function, classifying objects into different labels depending on the values of certain features, out of the training data. An acceptable function should be able to deal with unseen instances correctly, which requires the function to classify the data in a learned reasonable way. In supervised learning, there are several tradeoff issues, which would affect the training results, as stated hereinafter. [2]

*Bias-variance Tradeoff:* The tradeoff between bias and variance is the first issue to be considered. [3] An algorithm with high bias will ignore the relevant relations between features and expected outputs and give incorrect answers. And an algorithm with high variance will record the random noise rather than expected labels and perform bad in unseen inputs, which is also called overfitting. An algorithm should have flexibility to retain low bias. If we try to increase its flexibility, the variance of the algorithm would increase as well. Also, the similar tradeoff issue happens between the complexity of regression function and the size of training data. [4] A complex function will require a large amount of data for the model to learn correctly and the function may have low bias and high variance. On the opposite, a simple function only needs a small size of data, but it may become inflexible, and have high bias and low variance. To handle the tradeoff issues, a good model should adjust between bias and variance and make a balance.

*Dimensionality of Input Space:* If there are lots of features in inputs, it will be difficult for the model to learn, because redundant unrelated features will confuse the model. To

solve this problem and increase the accuracy, a reduction of features should be done.

*Noise in Output Values:* In reality, the desired outputs in a dataset may not be always correct or optimal due to many reasons, such as human errors. For instance, in our project, human players may make faults and choose a bad move sometime. It's also possible that different players may apply different strategies based on personal reasoning, and choose different moves in one same situation. If the learning algorithm wants to make perfect matches, it will overly fit into a specific training dataset and perform quite bad for other datasets.



## 3.2. Convolutional Neural Network

In machine learning, a Convolutional Neural Network (CNN) is a special type of NN and its connection pattern between neurons imitates the structure of cat's visual system. The main difference of CNN and normal Neural Network is that CNN makes assumption that inputs are pictures. And it has following features. [5]

*Local Receptive Fields:* In a fully-connected Neural Network, the input is connected to every hidden neuron. In CNN, however, neurons in the first hidden layer will only be connected to small region of inputs. The values of the first layer will be the results of a convolution between the input layer and filters. [5]

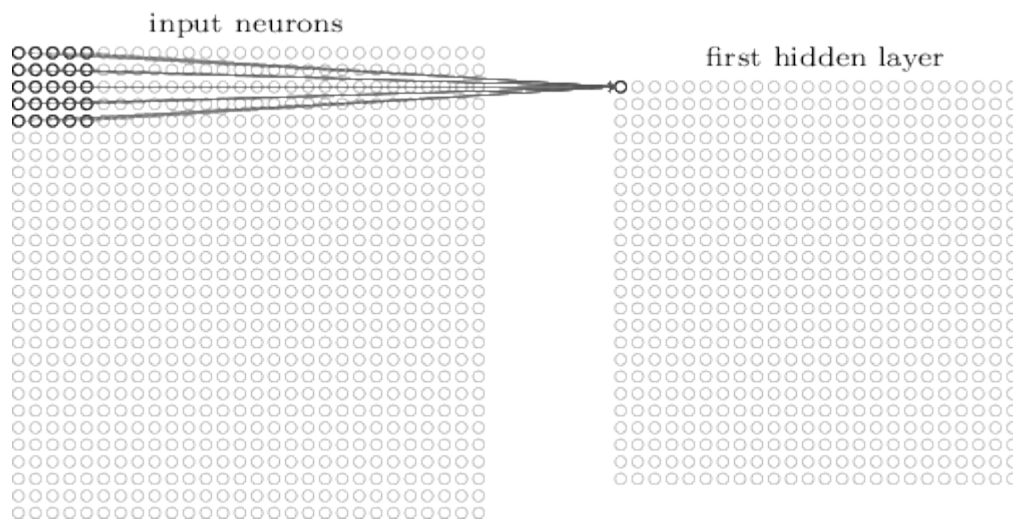


Figure 3.1. Local Receptive Fields

As shown in Figure 3.1., it applies a 5\*5 filter to a 28\*28 input image, and will get a 24\*24 hidden layer. Usually the filter is moved for one pixel at a time, but sometime a larger stride will be used. For instance, sometimes we may use a stride of 2, which means that each time we move the filter by 2 pixels to the right or down.

*Shared Weights and Biases:* For a given feature, the weight and bias of every neuron are same, resulting in the identical feature being detected by all neurons. The advantage of using this method is that it can reduce the total number of parameters and computations in the network. [5]

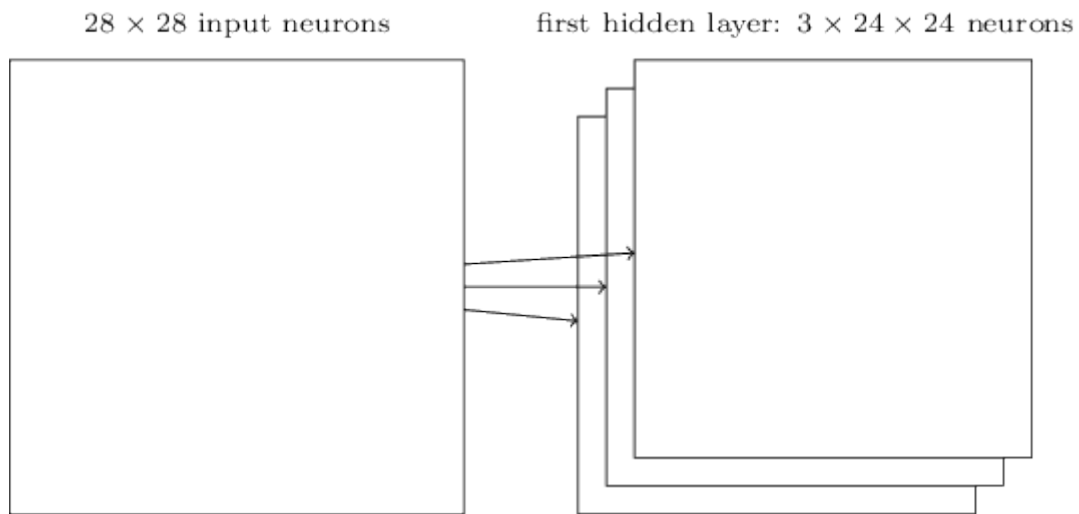


Figure 3.2. Shared Weights and Biases

In Figure 3.2., there exist 3 feature maps in the network. In every feature map, a  $5 \times 5$  filter is used, and the whole image shares the identical weights and bias. This network can detect 3 different kinds of features across the whole image.

*Pooling Layers:* Pooling layers are used to condense the output from convolutional layers and simplify the information. For example, max-pooling, a most-used method for pooling will pick the maximum value in a region of specific size, and then the number of neurons in the output of pooling layer will decrease greatly. [5]

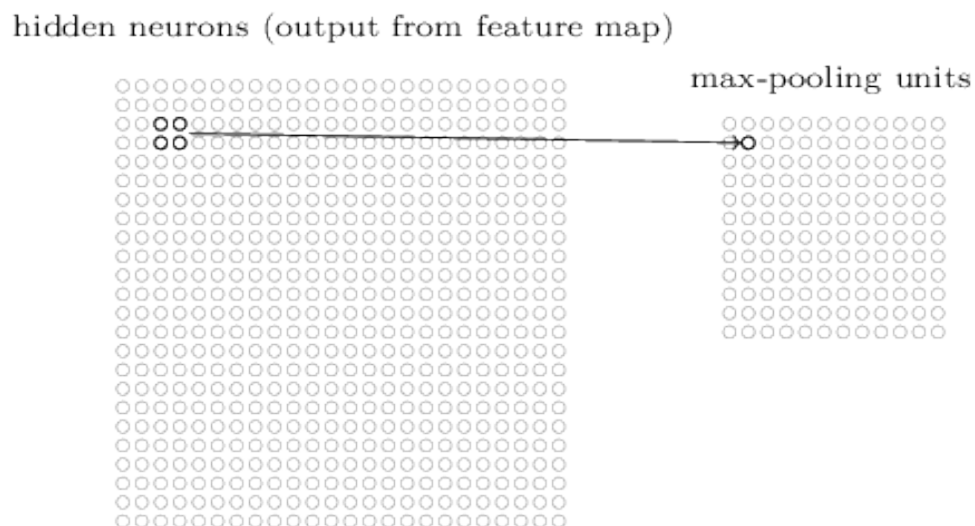


Figure 3.3. Pooling Layer

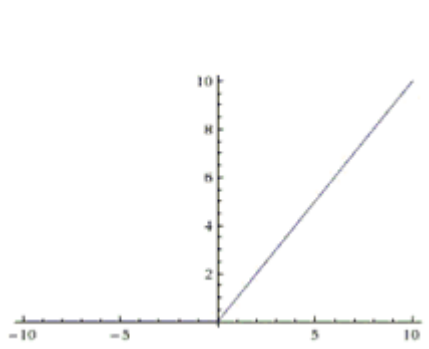
In Figure 3.3., a  $2 \times 2$  max-pooling is used. In every  $2 \times 2$  region, the pooling unit will find

the maximum value in the region and use it as the output. After the pooling process, the size of the output layer will become half of the hidden layer.

The final layer of CNN is usually a fully connected layer, connecting every neuron in its previous layer to every neuron in this layer, and uses a logistic function to output result.

With all these features, CNN will have better performance in some appropriate problems than traditional NN. The reason that we choose CNN will be mentioned afterwards.

In our project, we choose the rectifier as the activation function used in CNN.



*Figure 3.4. Plot of ReLU Function*

The graph above is the plot of the rectifier function. In NN, any unit employing the rectifier is called a rectified linear unit (ReLU). Compared with normal logistic functions like sigmoid function, it has higher efficiency in computation because it only contains comparison and addition, and avoids the problem of vanishing or exploding gradient. [6]

### 3.3. Softmax

The softmax function is a generalization of logistic regression when we need to classify among multiple classes. [7] After softmax, the highest input value will have highest probability and other values will be depressed. Every element in the output vector has a value in  $[0,1]$  represents the probability of the label is correct. For a  $K$ -dimension vector  $\mathbf{z}$ , the softmax function can be represented as:

$$\sigma(\mathbf{z})_j = \frac{e^{z_j}}{\sum_{k=1}^K e^{z_k}} \quad \text{for } j = 1, \dots, K$$

$\sigma$  is the output vector with sum equal to 1.

### 3.4. Minimax Searching

Minimax is a search method that can be used in different kinds of chess game that it can minimize the maximum loss. And it works well in two-man zero-sum games, including our topic Chinese Chess.

To explain how minimax works, firstly, we define max-min value as the maximum reward a player can win if he doesn't know the others' action. Equivalently, it is the minimum value his opponents can reduce the reward to if they know his action. Or in the formula below:

$$\underline{v}_i = \max_{a_i} \min_{a_{-i}} v_i(a_i, a_{-i})$$

In the formula,  $i$  represent the current play while  $-i$  represent the others.  $a$  is the action taken and  $v$  is the reward gain or the evaluation.

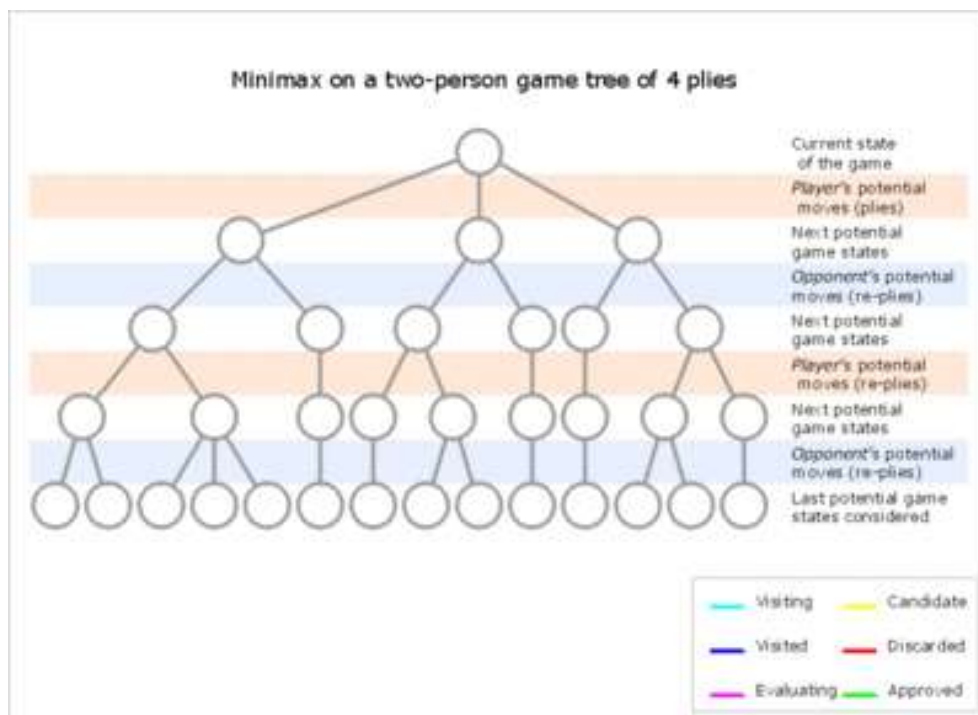


Figure 3.5. Minimax Searching

To calculate the value, every time we search a move, we always suppose the opponent will make the strongest move, or in other way the move which make the evaluation the

lowest. And this kind of searching can be iterated and be recognized as search tree as Figure 3.5..

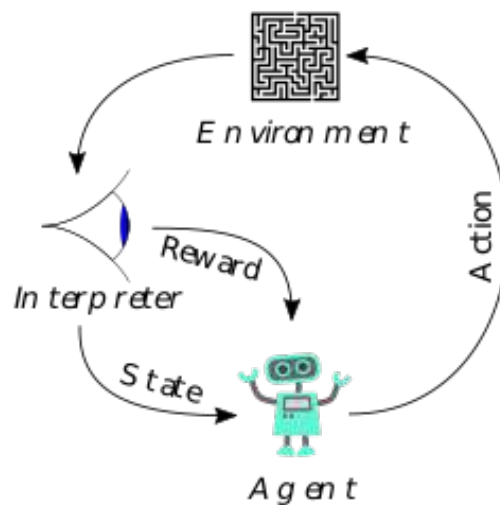
In the zero-sum games like Chinese Chess, the result of minimax will reach Nash equilibrium. And the action they make will ensure that they can get maximum regardless the action made by the opponent.

Using this kind of strategy, the engine can look further and prevent some short-sighted move. For example, the old engine may use the cannon to capture the opponent's knight directly, which will soon be captured by the rock nearby and is absolutely a bad move. With minimax, it can predict the counter and make a better move.

### 3.5. Reinforcement Learning

Reinforcement Learning also belongs to Machine Learning. It mainly concerned with how the model (or agent) should react to the input in a specialized situation to maximize the cumulative reward. Usually, we can use supervised learning to train if we have enough size of dataset. However, if we want to improve the performance of the model, the original dataset may not be enough. Reinforcement learning, as one of unsupervised learning, can be a solution.

The main difference between Reinforcement Learning and standard supervised learning is that in the process of Reinforcement Learning, standard correct answers to inputs are never given. So, the model will try to find a balance between exploitation (of learnt knowledge) and exploration (of unknown territory).



*Figure 3.6. Reinforcement Learning*

In Reinforcement Learning, a basic model should consist of the following parts: a set of statuses about the environment and agent, a set of action to transit between states, rewards given according to the transition and the action, observations that the agents can see (in our case, Chinese chess is fully observable, so the whole chess board is always seen by the network).

At the process of Reinforcement Learning, the model should give response to the

current states in discrete time steps. At each time  $t$ , the agent will see its observation of the environment and chooses a movement from all the available moves, and sent to the environment. Then the environment will transit to a new state and calculate the reward according to the transition. The objective of the learning process is to learn how to get the maximum amount of rewards. In the chess game, the greatest reward is the win of a game.

Google's AlphaGo also uses Reinforcement Learning in the Policy Network. In their approach, they use a Reinforcement Learning network whose structure and values are identical to those in their previous supervised learning network. And they stochastically select a version among the iterations they made to avoid overfitting. After the Reinforcement Learning, the Policy Network has 80% winning rate against the iteration before Reinforcement Learning. So, we can see that Reinforcement Learning can be a powerful tool to improve the performance of neural network model.



### 3.6. TensorFlow

As an open-source project, TensorFlow is a software library designed to do numerical computation. It can support different platforms, including desktop, server and mobile platform, and can run on both CPU and GPU. TensorFlow provides developers using deep learning with an easy way to handle underlying layer computation. They just need to define the architecture of their own Neural Network model, select the objective function they want to use, and then feed the training data into the model. TensorFlow makes their work much easier and clearer. As TensorFlow is built to support threads, queues, and asynchronous computation, it can make the best of the computation ability of hardware including both CPU and GPU. [8]

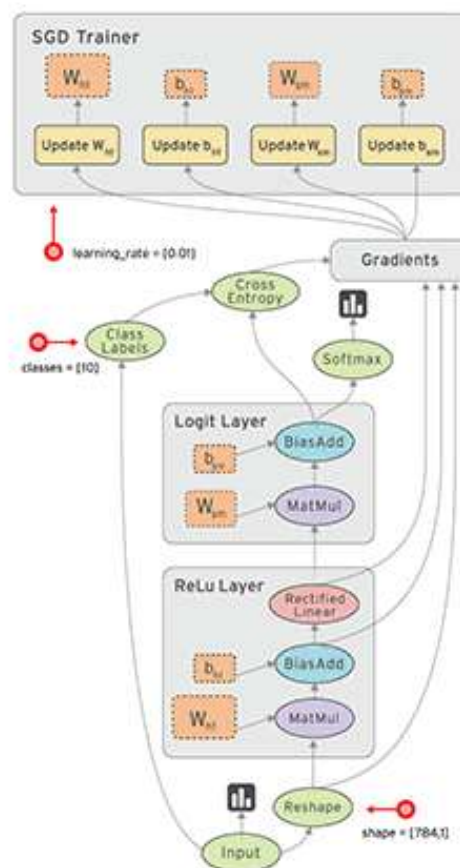


Figure 3.7. Data Flow Graph

### 3.7. Aliyun

Cloud computing is a new concept that service provider distributes their processing resources to users on demand. And people can deploy their program or server on it without worrying about setting up the environment. The use of cloud computing releases the workload of engineers from trivial problems and help them focus on their own work.

Aliyun is a cloud computing service provider own by Alibaba, and it is the leading provider in mainland china. Due to 2015 IDC report, it is one of the top five providers in this field. Also, there are many well-known companies using Aliyun's service, including Nestle, Philips and Alipay.



*Figure 3.8. Data Centers in Aliyun*

## 4. Design

### 4.1. Project Workflow

Our project development process could be roughly divided into the following four steps: Model Design, Model Building, Model Training and Model Testing. These four steps were repeated until we found that the final performance of the game AI was reasonable, or satisfying to certain extent.

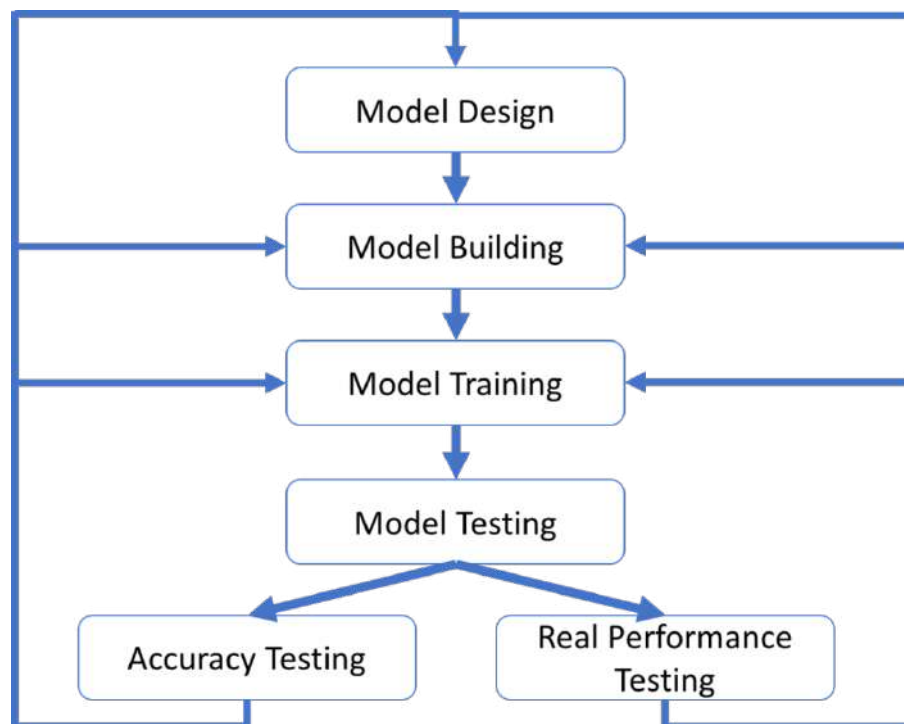


Figure 4.1. Project Workflow

First, based on previous works of other game engines in Chinese chess, chess and Go, our own model was designed, such as the structure details of Neural Network, the components of our AI model, the algorithms to make move choices, and so on, considering the special aspects of Chinese chess.

Secondly, the AI model was built based on our previous design, and functions were implemented accordingly. Also, the training data were collected and processed for future usage.

Next, the AI model was to be trained, using training datasets collected previously, by certain training strategies.

At last, the trained AI model was to be tested so that its performance could be quantified or directly demonstrated. Here, it was firstly tested by a testing dataset, which was in the same format with the training dataset, and its prediction accuracy was calculated, which was a quantitative measurement of the AI model. Besides that, the AI model was also tested against real human players, to see whether its performance appeared to be reasonable in actual games. Basically, only after a satisfying accuracy was achieved in that simple Accuracy Testing, the AI model would be tested in real games.

After testing, the results were analyzed to find the reasons behind, and then certain modifications would be made.

If the design was determined to be ineffective, a new model would be designed, after more researches, analysis and reasoning. But if errors were found in the procedures of Model Building or Model Training, or those steps could be changed to improve the training results, modifications would also be made accordingly and we would train the modified model and test it again.

## 4.2. Game Engine Overview

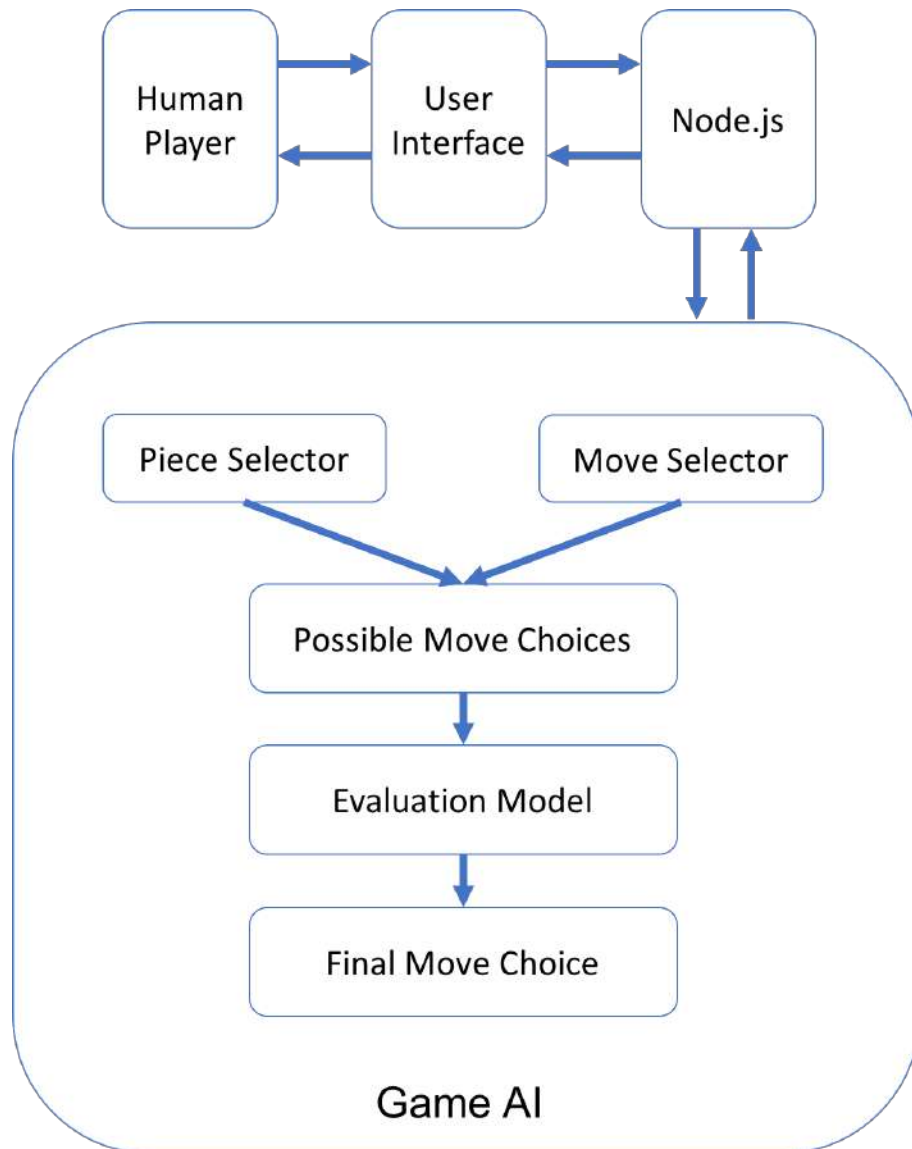


Figure 4.2. Game Engine Structure

Here is the general structure design of our final version AI model. Our game engine mainly consists of three parts: frontend, backend and the connection between them.

The frontend is the User Interface, which self-evidently serves as the interface for players to play Chinese chess against our game AI. The backend is basically the AI model, with several minor functions. This is the most important and difficult part of our whole project.

At last, a connection between frontend and backend is necessary, considering the fact that our model cannot directly run on the browser, as the library provided by TensorFlow is required but may not be supported by the frontend. This is also a tricky part, as our frontend is written in JavaScript while our backend is written in Python. Conventionally, JavaScript programs and work well together with PHP programs. TensorFlow, however, does not provide libraries for PHP. Therefore, this connection needs to be established by special techniques.

### 4.3. User Interface

Our game User Interface (UI) was written in JavaScript, using the cocos2d-html5 engine, so that it can support different types of platforms, like PC, iOS and Android. This UI was primarily based on an open-source project in GitHub [9], and certain modifications were made per the special requirements of our project.

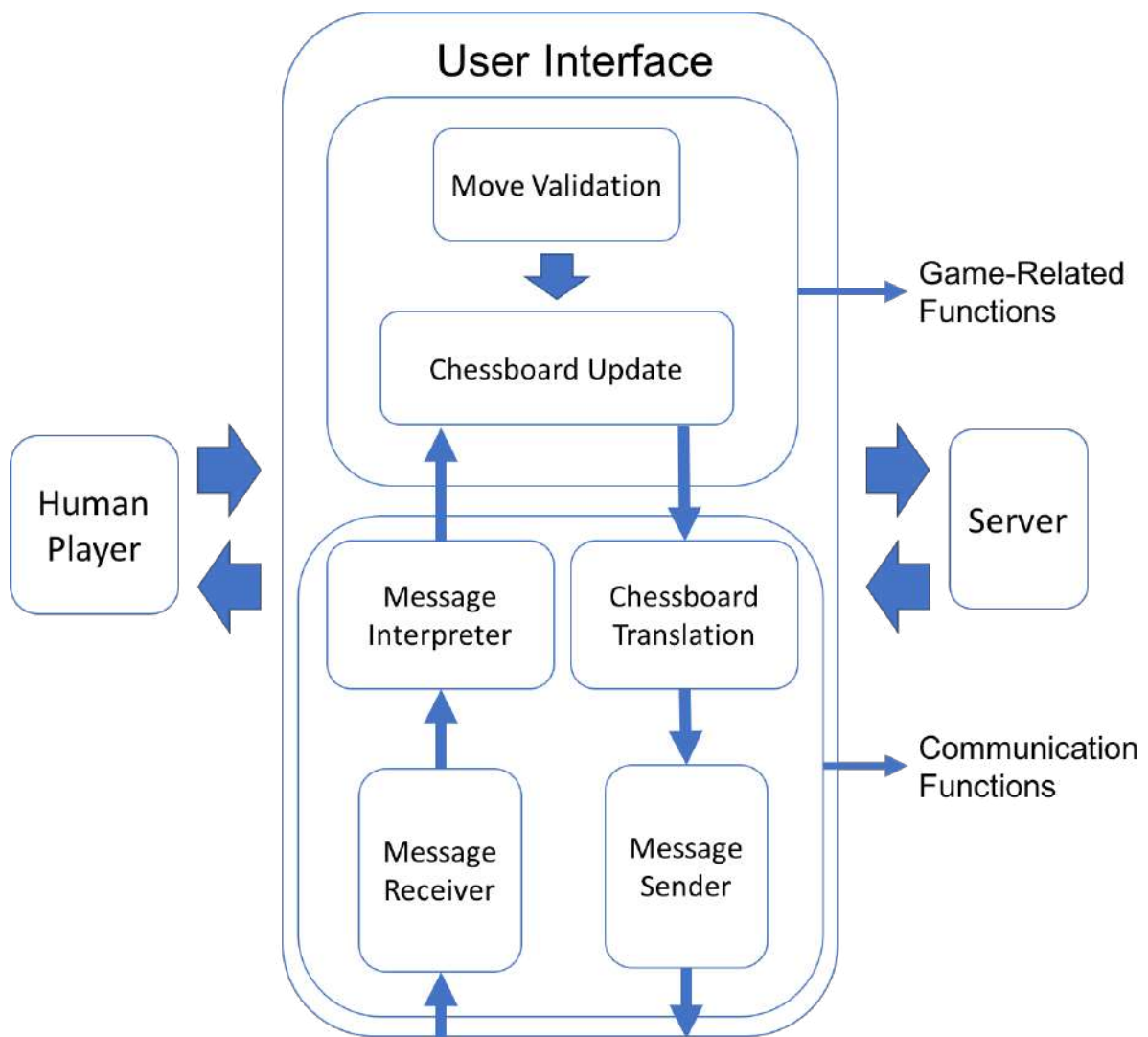


Figure 4.3. User Interface Structure

Self-evidently, the main function of UI is to convey messages between human players and backend programs. But it also needs to ensure the rules of the game, i.e. Chinese chess, to be obeyed and the game can continue smoothly. Therefore, our UI can be

divided roughly into two parts: Game-Related Functions and Communication Functions.

#### 4.3.1. Game-Related Functions

Following basic game-related functions were implemented:

- Move Choosing – to let players make moves alternately
- Move Validation – to ensure only valid moves per rules of Chinese chess can be made
- Move Execution – to make the move per players' choice and update the chess board status accordingly
- Checkmate Checking – to check whether one of the players is in check and whether one side is winning

These basic functions ensured our game engine could function correctly and legally.

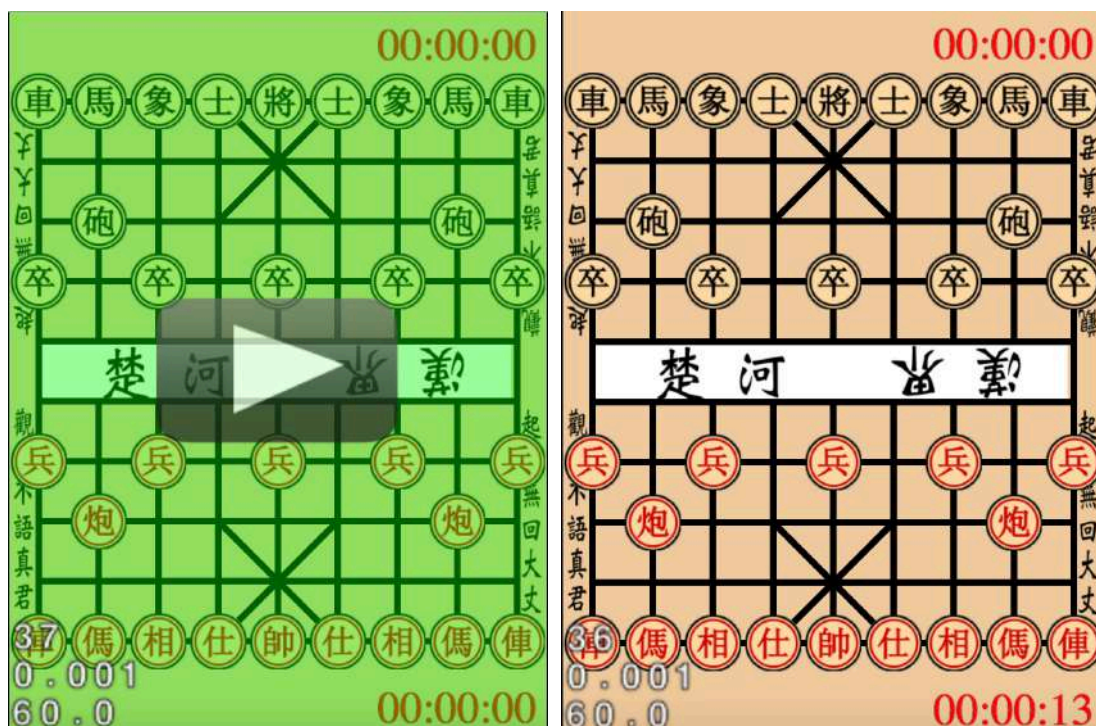


Figure 4.4. (a) Examples of User Interface



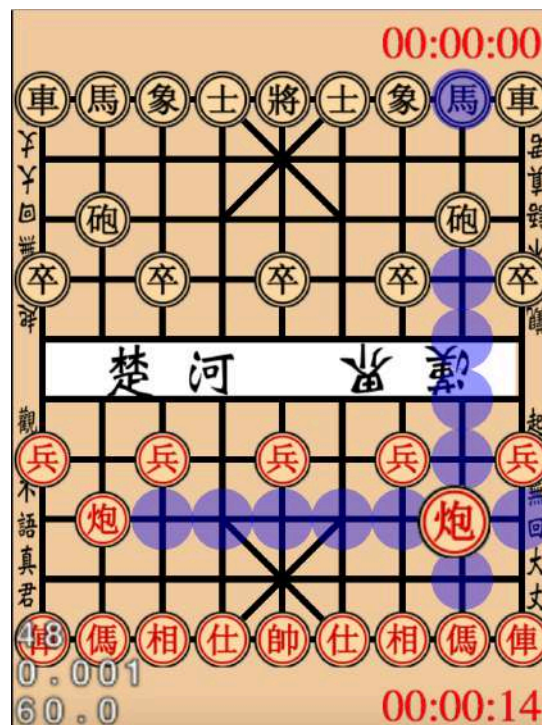


Figure 4.4. (b) Examples of User Interface

In Figure 4.4., the left picture is the beginning of UI and users need to click the button to start the game. The middle picture shows the initial chessboard. The right picture shows when the user is trying to make a move for the red Cannon and the purple cycles indicates the legal moves for it.

#### 4.3.2. Communication Functions

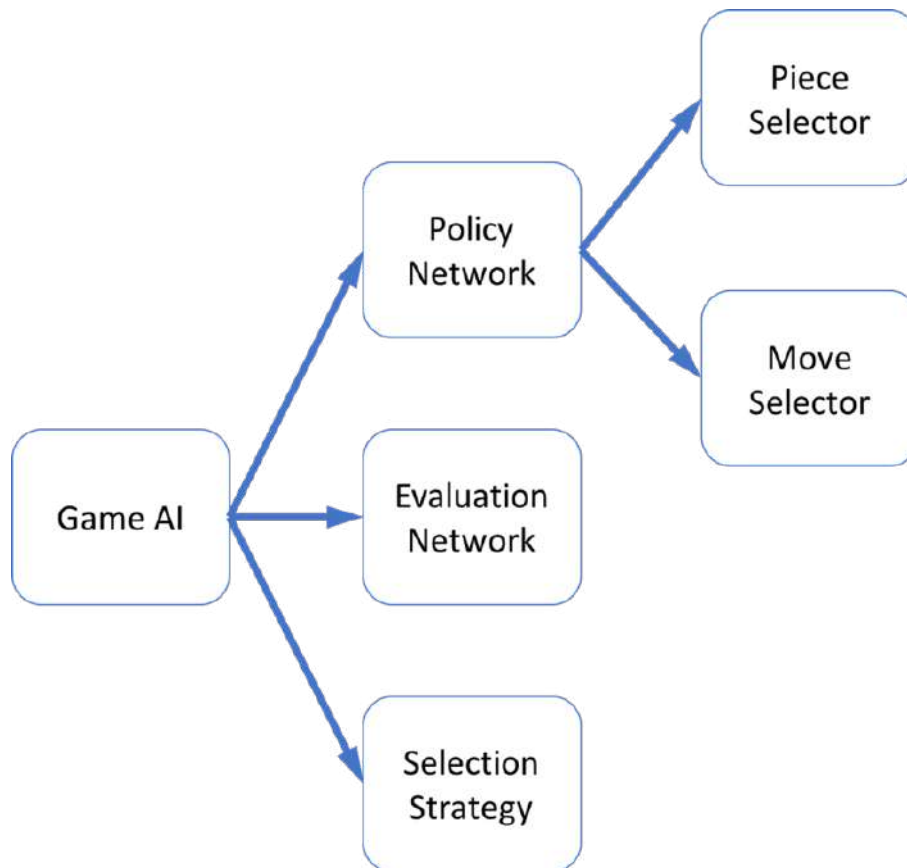
Apart from the basic functions mentioned above, several more functions were implemented as well, allowing our UI to communicate with our AI.

- Chessboard Translation - to represent the chessboard status in FEN format
- Message Sender - to send the FEN of chessboard status to the server via socket
- Message Receiver - to receive the message of move choice of our AI from the server via socket
- Message Interpreter - to interpret the received message and allow our UI to update the chessboard correctly

## 4.4. Game AI

### 4.4.1. Structure Overview

Our game AI consists of several Neural Network models, written in Python, with some other minor functions.



*Figure 4.5. Major Components of Game AI*

Generally speaking, there are mainly three important components inside the AI, as shown in Figure 4.5., two Neural Network models - Policy Network to predict the most possible next move and Evaluation Network to evaluate the winning rate given certain chessboard status, and Selection Strategy, an algorithm to make move choices based on the output of Policy Network and Evaluation Network, which to be specific mainly applied Minimax Searching.

For Policy Network, it can be further divided into two Neural Network models, Piece

Selector and Move Selector, which will be explained in detail later. Roughly speaking, Piece Selector decides which piece to be moved and Move Selector decides where that piece to be moved to. For both the two NN models, a probability distribution over the all 90 positions of a chessboard will be output, indicating the possibility to choose each position.

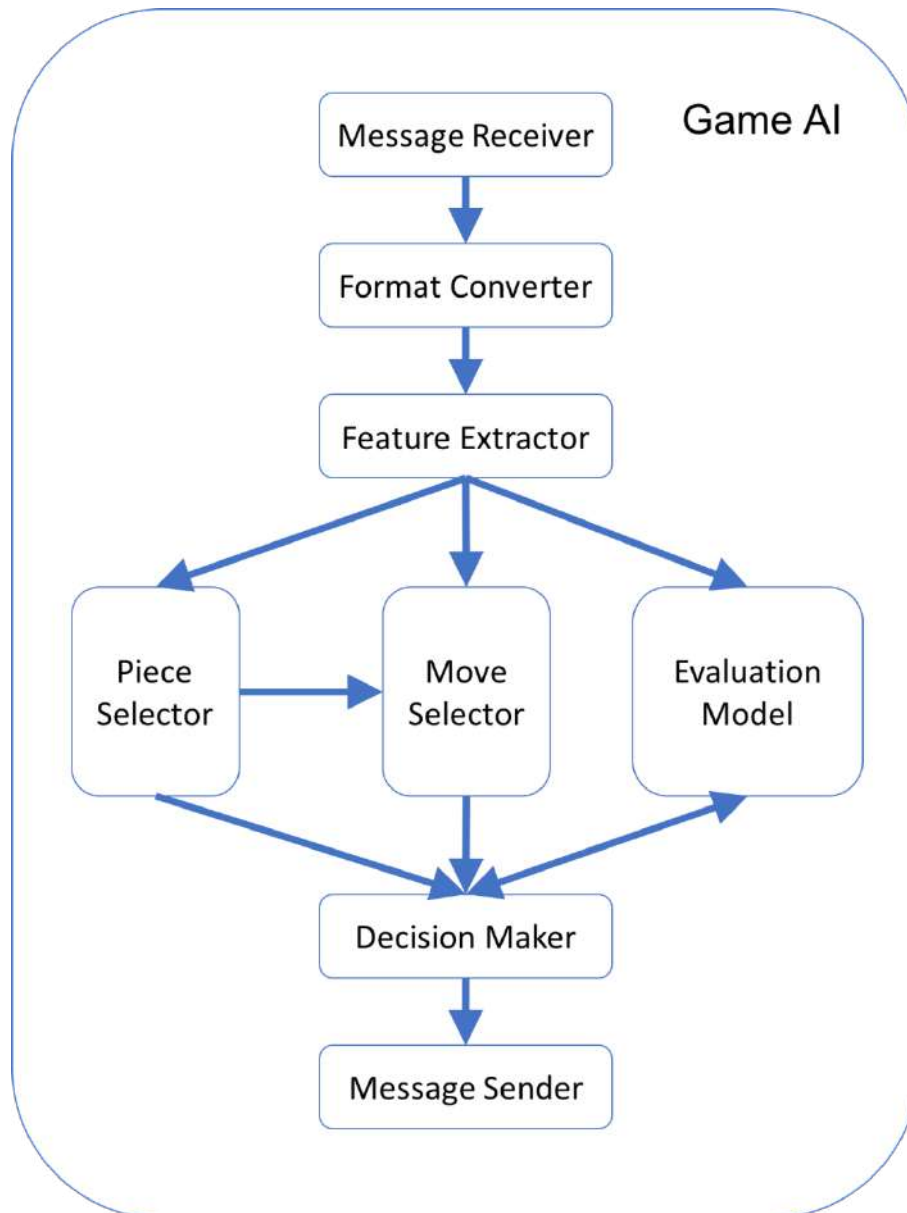


Figure 4.6. General Structure of Game AI

Other minor functions include processing input and output, calling some validation functions to validation the move choices, managing those Neural Network models and

so on.

As shown in Figure 4.6., the overall flow is: Message Receiver receives the FEN information from frontend via socket, and Format Converter preprocesses the information so that Feature Extractor can identify it and extract according features out. After that, Piece Selector and Move Selector together outputs the probability distributions of possible moves. Decision Maker will first pick up several move candidates, pass them to Evaluation Model and obtain scores indicating the relative advantage after making each move. At last, Decision Maker makes a move choice based on scores evaluated by Evaluation Model and Message Sender sends the choice back to frontend.

#### 4.4.2. Piece Selector and Move Selector

To make a move, the player needs to choose a self-side piece first and then choose a destination for that piece. Accordingly, our AI consists of two parts, Piece Selector and Move Selector, either of which is a NN model itself. [10]

Evidently, Piece Selector is to choose a piece per the chessboard information and Move Selector is to choose a destination for that piece chosen by Piece Selector. So, firstly Piece Selector will decide which piece to move and pass this information to Move Selector as well. Next, Move Selector will decide where that piece to be moved to. Combining the outputs of two NN models together, our game AI would output a four-element array to denote the move choice, decided by certain selection strategy, and send it back to frontend.

As different kinds of pieces should obey different rules when making moves, different Move Selectors were trained and used for each kind of pieces. So, Move Selector itself actually consists of 7 different NN models, and will use different ones to generate output accordingly, while Piece Selector consists of only one NN model.

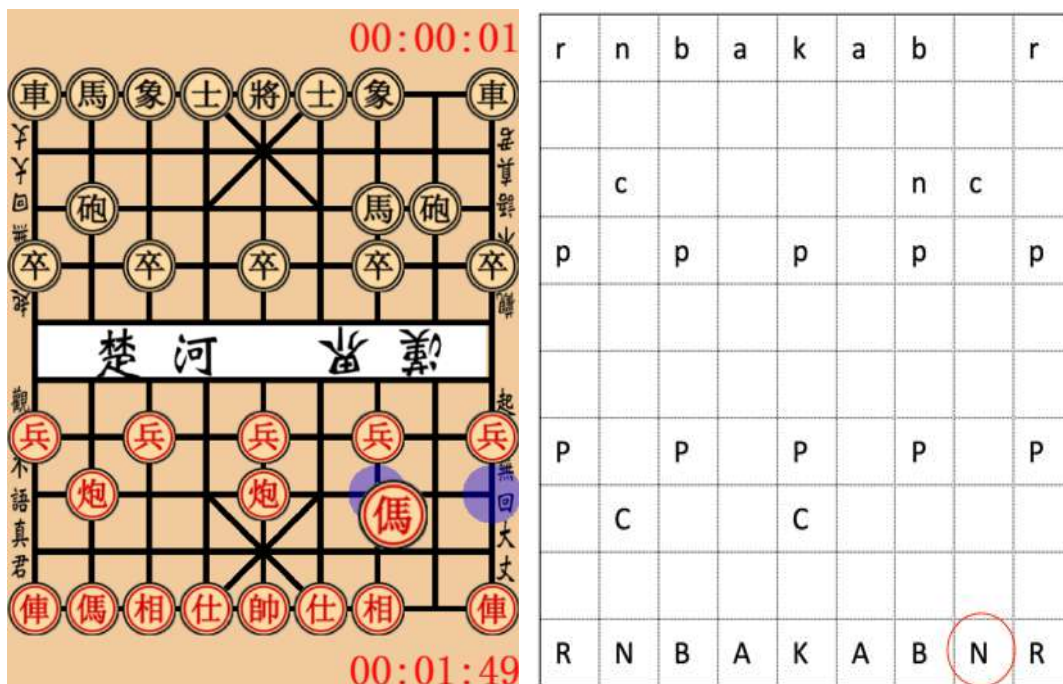


Figure 4.7. (a) An example of Piece Selector and Move Selector

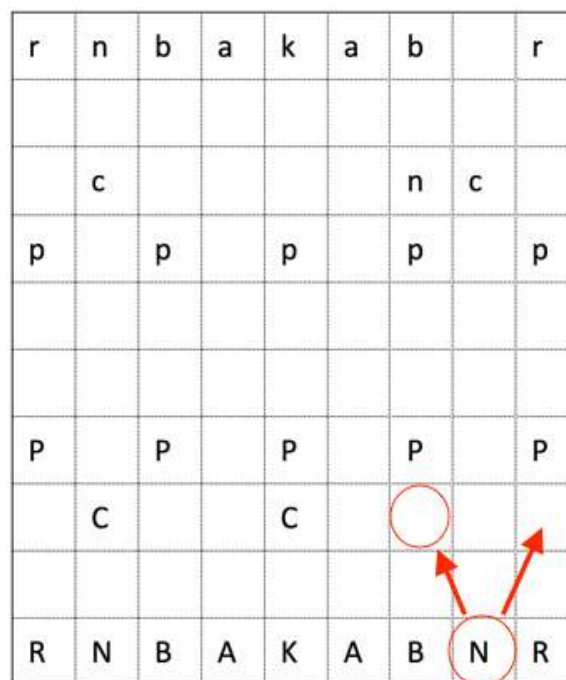


Figure 4.7. (b) An example of Piece Selector and Move Selector

As shown in Figure 4.7. above, the first picture is a real screen capture of our UI. The second and third ones represent the digital information that our AI receives. In the second one, the Knight piece in the red cycle indicates that our Piece Selector decides to choose this piece to move by certain selection strategy, like choosing the one with the

highest probability. Then with the chessboard information and the output of Piece Selector, Move Selector uses the NN model for Knight pieces, and decides a destination for that piece, i.e. the other red cycle in the third picture. And the two red arrows indicate the legal moves for that Knight piece.

#### 4.4.2.1. Model Structure Design

The general structure of Piece Selector and Move Selector is basically the same, as shown in Figure 4.8.. Both them accepts several same features of the chessboard status as input while Move Selector needs one more feature indicating valid moves for the piece selected by Piece Selector. Several convolutional layers were used first to do convolution among different feature channels. At last, one softmax layer would process the results of convolutional layers and output a probability distribution over all the 90 positions in a chessboard.

In our final model, pooling layer was not used because the information in the chessboard was already quite sparse and it would be better for all information to be preserved. Since the size of input is small and every value in the input represent a piece, a pooling layer may greatly influence the result. For the same reason, dropout was not used as well.

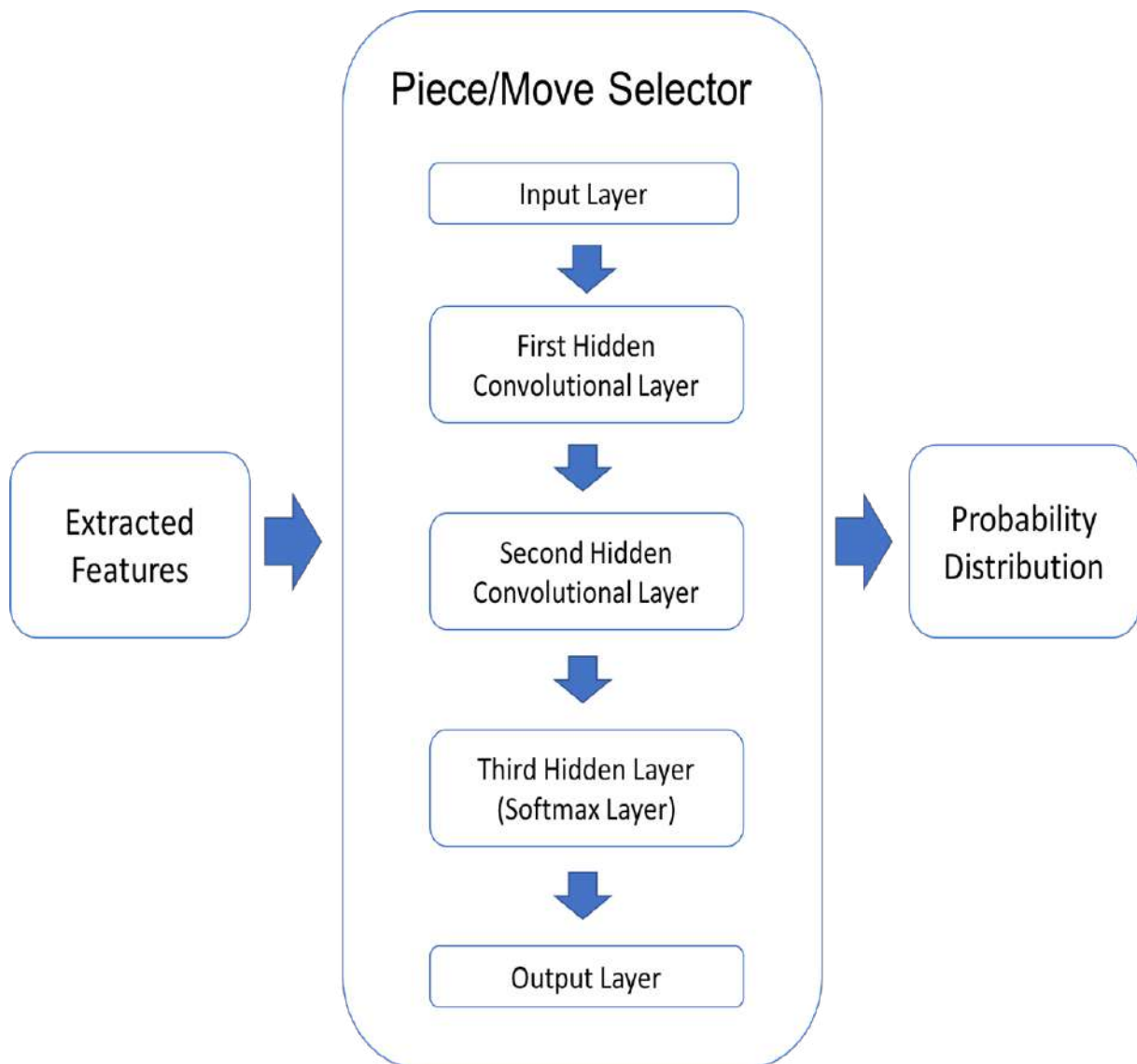


Figure 4.8. Structure of Piece/Move Selector

#### 4.4.2.2. Features Extraction

As mentioned above, several feature channels would be extracted as input feeding into the NN models, after converting the FEN string format into a 10\*9 matrix representing the chessboard and getting the current player, as shown in Figure 4.9..



Feature Channel 1	Pieces belonging to different sides
Feature Channel 2	Pieces of Advisor type
Feature Channel 3	Pieces of Bishop type
Feature Channel 4	Pieces of Cannon type
Feature Channel 5	Pieces of King type
Feature Channel 6	Pieces of Knight type
Feature Channel 7	Pieces of Pawn type
Feature Channel 8	Pieces of Rock type
Feature Channel 9	Valid moves for the selected piece (only for Move selector)

*Figure 4.9. Feature Channels*

First channel was to use ‘1’ and ‘-1’ to respectively denote the positions of self-side pieces and opponent-side pieces, and ‘0’ to denote empty positions.

For example, as shown in Figure 4.10. below, for the red side, all pieces represented by lowercase letters are of the opponent, so they are represented by ‘-1’ in this feature channel, while the other pieces, which belongs to the red side, are represented by ‘1’.

r	n	b	a	k	a	b	n	r
	c						c	
p		p		p		p		p
P		P		P		P		P
	C						C	
R	N	B	A	K	A	B	N	R

*Figure 4.10. (a) An Example of First Feature Channel*

-1	-1	-1	-1	-1	-1	-1	-1	-1
0	0	0	0	0	0	0	0	0
0	-1	0	0	0	0	0	-1	0
-1	0	-1	0	-1	0	-1	0	-1
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
1	0	1	0	1	0	1	0	1
0	1	0	0	0	0	0	1	0
0	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	1

Figure 4.10. (b) An Example of First Feature Channel

For each type of pieces, there was a feature channel to denote the positions of pieces of that type. Still, use ‘1’ and ‘-1’ to respectively denote self-side pieces and opponent-side pieces, and ‘0’ to denote empty positions. The reason for separating them into seven channels is that the values of different kinds of pieces are difficult to assign and they may vary in different situations, but we still need to find a way to tell our model that they belong to different categories, which is neither ordinal nor cardinal. Therefore, using 7 channels, one for each type, would be a good choice to distinguish different types of pieces.

For example, as shown in Figure 4.11. above, for the feature channel to represent pieces of Cannon type, we can find ‘1’ and ‘-1’ for Cannon pieces of two sides respectively in corresponding positions.

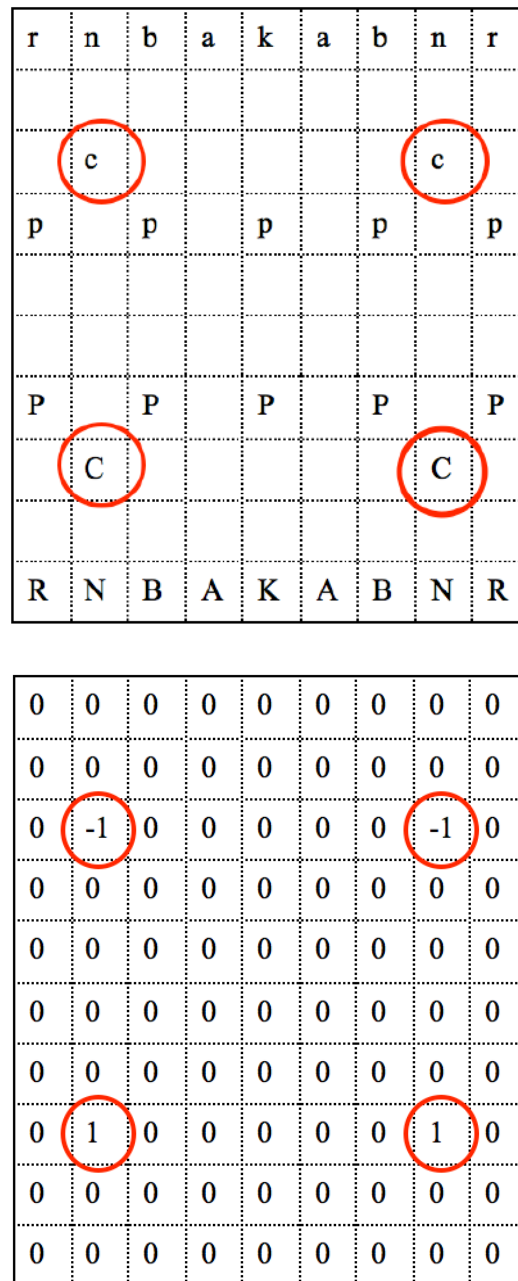


Figure 4.11. An Example of Feature Channels 2 ~ 8

For Move Selector, there was one more feature channel. In that feature channel, the position of chosen piece was denoted by '1', and all possible valid destinations for that piece were denoted by '2' while all possible invalid destinations were denoted by '-1'.

For example, as shown in Figure 4.12. below, assuming that the Cannon piece in red cycle is chosen, all the possible valid moves for it are represented by '1', while the

invalid moves are represented by '-1' and its own position is indicated by '2'.

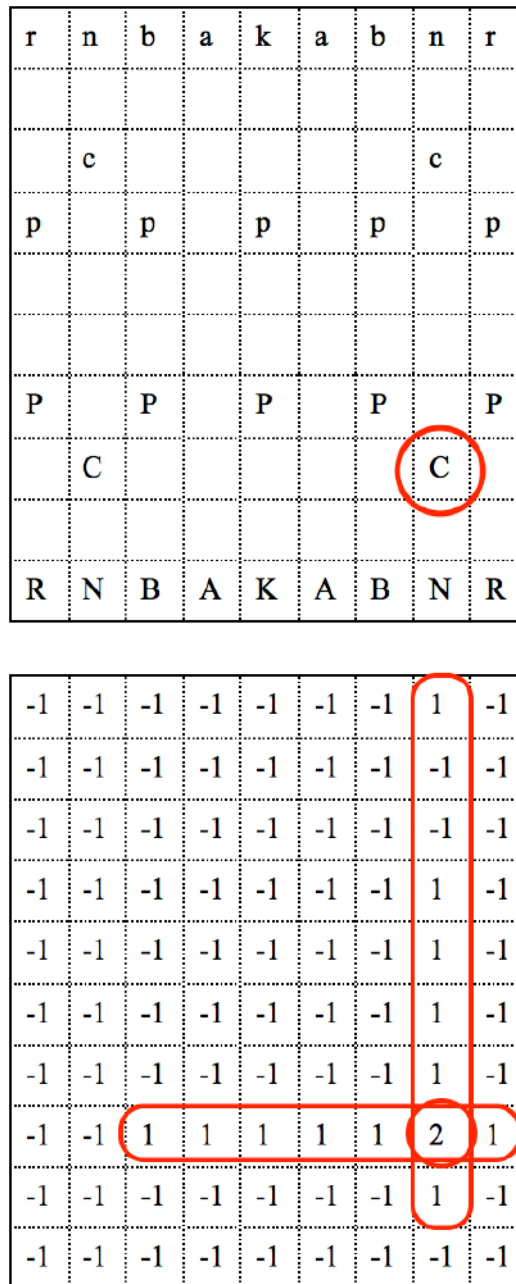


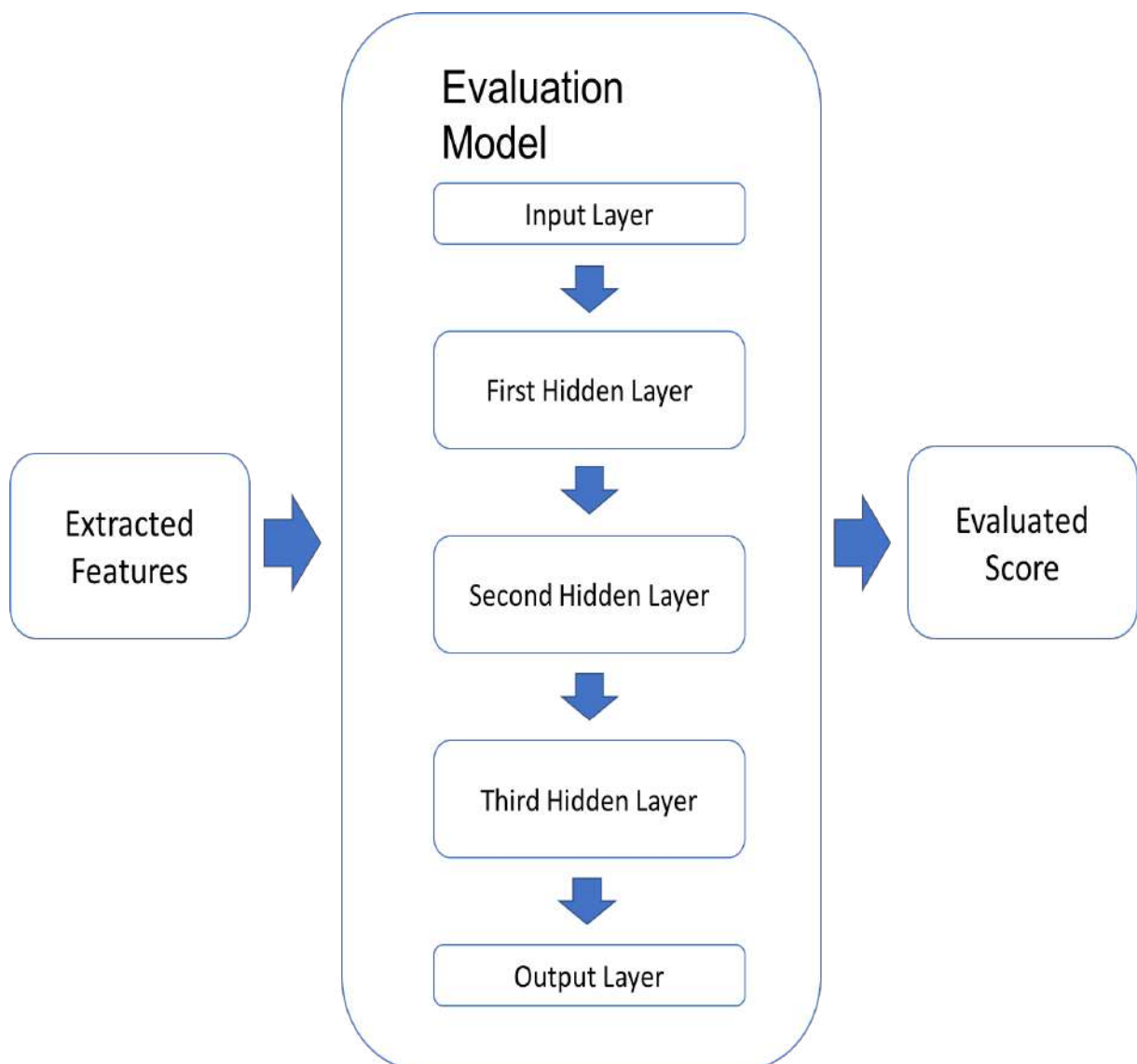
Figure 4.12. An Example of Ninth Feature Channel

In total, 8 feature channels for Piece Selector and 9 feature channels for Move Selector would be extracted accordingly.

#### 4.4.3. Evaluation Model

##### 4.4.3.1. Model Structure Design

The general structure of Evaluation Model is as shown in Figure 4.13.. It accepts some features describing the chessboard status as inputs. There are three hidden layers in the middle, which all are fully connected layers. At last, in the output layer, it outputs a number as the score of evaluating the chessboard status.



*Figure 4.13. Structure of Evaluation Model*

Here, the input features are different from Piece Selectors and Move Selectors as

Evaluation Model does not need information such as valid moves, piece types and so on. Information such as the number of pieces is more important when evaluating a chessboard status.

In Evaluation Model, we did not use Convolutional Neural Network layers. Instead, all the layers are fully connected. The reason is that for Piece Selector and Move Selector, using convolutional layers can help in recognizing patterns which is less important or helpful for evaluation. On the contrary, some summarized data may help evaluate whether a player has advantage.

#### 4.4.3.2. Feature Extraction

As to the input of evaluation model, we extract some features out of the chessboard state, whose length is 213 in total.

Feature	Length
Player Side	1
The Number of Pieces of Each Type	14
Pieces List (alive or not, xy-coordinates)	32 * 3
The number of valid moves for Rock, Cannon and Knight	12
Attack and Defend Map	90

*Figure 4.14. Features of Evaluation Model*

As shown in Figure 4.14., first feature is Player Side where we use 1 for red side and -1 for black side. Second feature is an array storing the number of each piece type, where all are non-negative numbers and the order is fixed – first red pieces then black pieces.

Next is a list storing two kinds of information of each piece. First is whether the piece is

still alive or not: if yes, put 1 here; else, put 0 here. Second is the position of that piece: if the piece is alive, put its current xy-coordinates here; else, put 0 in it.

Besides, we also calculate the number of possible moves for three types of pieces, Rock, Cannon and Knight. These pieces are quite important in Chinese chess as players usually use them to capture the opposite pieces and check the opponent. With more possible moves, these pieces can have more power to attack or defend.

At last, we calculate a so-called attack-and-defend map, which is a 90-long array. It restores whether the corresponding position in the chessboard is defended by the player or attacked by the opponent.

#### 4.4.4. Decision Maker

With Policy Network and Evaluation Network, the strategy to make the final selection is also very important in our project. In our Game AI, we tried several different Selection Strategies to make use of and combine the outputs from both Policy Network and Evaluation Network. The module Decision Maker is mainly responsible for this selection function as well as calling the models.

In our final AI model version, the selection strategy is as following.

First, it will choose several moves as candidates, mainly based on the outputs of Piece Selector and Move Selector. These moves have high predicted probabilities and hence can be regarded as good moves theoretically.

Of course, due to the training results of Piece Selector and Move Selector may not be perfect, these predicted values may not be very accurate. Therefore, in application, the AI will sort of relax the selection limit. Basically, we first eliminate those moves with extremely low predicted probabilities and set a maximum number as a changeable parameter so that the AI can select moves up to the limit.

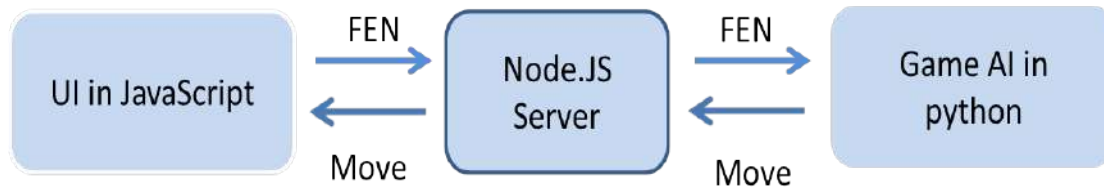
Afterwards, Decision Maker pass these candidates to Evaluation Model. Evaluation Model can evaluate the chessboard statuses after making those moves and return scores back. In our final version, we implemented Minimax Searching here and used that to do a little searching when evaluating those moves. The depth of searching is also a parameter that we can adjust. By Minimax Searching, the evaluated score can better reflect the winning rate of making a move and thus be more reliable.

In the end, one best move is selected based on the outputs of Evaluation Model and this will be the final decision of our Game AI.



## 4.5. Connection between Frontend and Backend

We used Node.js to build the connection between server and frontend, and used socket.io to support the communication between our UI in JavaScript and our AI in Python. The connection structure is as shown in Figure 4.15..



*Figure 4.15. The Connection between Frontend and Backend*

The game AI, running on the server and connecting to Node.js as a user, waits for the message from the frontend and sends move choice back, via socket supported by Node.js. To be more precise, after the server starts, every time a user opens the UI, it will connect to Node.js on the server through socket.io. After the player makes a move, the UI will generate the current FEN and send it to server via socket. The server receives FEN and transfers the message to the game AI. For the AI in python, we used the library socketIO-client to read the message because the socket in python cannot read message. After the AI generates the next move, it will send the coordinates in the form of four numbers back to frontend. And then UI will make a move to update the chessboard after receiving the coordinates.

The reason we choose socket.io is that the JavaScript program cannot invoke our python program directly. So, socket is our solution because it can communicate between server and client in real-time and support programs written in different programming language.

## 5. Implementation and Development Process

### 5.1. Project Schedule

#### 5.1.1. Schedule in First Term

Sep 14 ~ Sep 20	Implement translator from PGN
Sep 21 ~ Sep 27	Implement move generation and validation
Sep 28 ~ Oct 4	Build AI using Monte Carlo
Oct 5 ~ Oct 11	Trying to build evaluation network
Oct 12 ~ Oct 18	Trying to build evaluation network
Oct 19 ~ Oct 25	Build UI
Oct 26 ~ Nov 1	Connect frontend and backend
Nov 2 ~ Nov 8	Design policy network
Nov 9 ~ Nov 15	Build policy network
Nov 16 ~ Nov 29	Train and test policy network

*Figure 5.1. Project Schedule in Term 1*

## 5.1.2. Schedule in Second Term

Jan 9 – Jan 15	Make the schedule of term
Jan 16- Jan 22	Multiple login on the server
Jan 23- Jan 29	Chinese New Year
Jan 30- Feb 5	Chinese New Year
Feb 6 – Feb 12	Reinforcement learning design
Feb 13 – Feb 19	Reinforcement learning training
Feb 20 – Feb 26	Reinforcement learning training and test
Feb 27 – Mar 5	Testing the selection strategy
Mar 6 – Mar 12	Modify previous structure of evaluation model
Mar 13 –Mar 19	Building and train evaluation model
Mar 20 – Mar 26	Build minimax search
Mar 27- Apr 2	Combine the two parts
Apr 2 – Apr 8	Testing the performance of the game engine
Apr 9 – Apr 15	Writing report

*Figure 5.2. Project Schedule in Term 2*

## 5.2. Summary of Different AI Models

Model	Remark
001	Policy Network, Supervised Learning
006	Failures, Based on 001, Policy Network, Reinforcement Learning with positive/negative reward
018	Based on 001, Policy Network, Reinforcement Learning with positive reward only
018.1	Policy Network and Evaluation Network
018.2	Policy Network and Evaluation Network with Minimax Searching

Figure 5.3. Summary of Model Versions

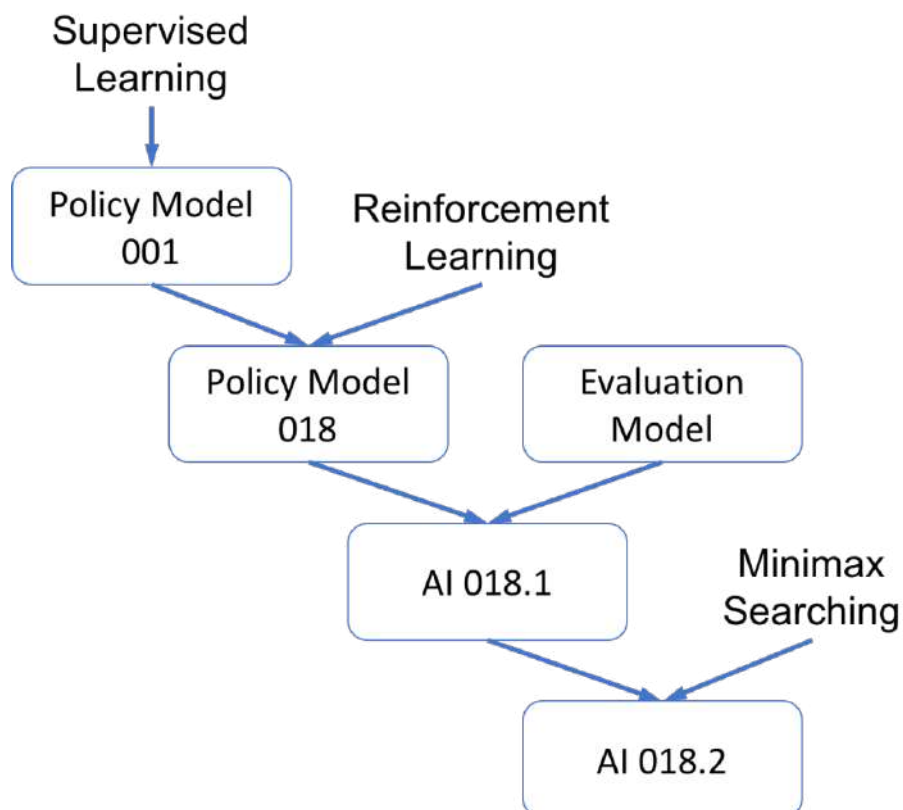


Figure 5.4. Relationship of Models

### 5.3. Summary of Different Selection Strategies

Selection Strategy	
1	Choose the greatest one from Piece Selector and then the greatest one from corresponding Move Selector
2	Choose the greatest three/five ones from Piece Selector and then the greatest one from corresponding Move Selectors; Multiply the possibilities correspondingly and choose the greatest combination
3	Randomly choose according to the predicted probabilities from Piece Selector and then the greatest one from corresponding Move Selector
4	Randomly choose according to the predicted probabilities from Piece Selector and then randomly choose according to the predicted probabilities from corresponding Move Selector
5	Choose the greatest three/five ones from Piece Selector and then the greatest three/five one from corresponding Move Selectors; Multiply the possibilities correspondingly and randomly choose one combination by the results
6	According to the order of predicted possibilities from Piece Selector, choose at most 3 moves for that piece by the predicted possibilities from its Move Selector; Choose the best one from all candidate moves by the outputs from Evaluation Model
7	According to the order of predicted possibilities from Piece Selector, choose at most 3 moves for that piece by the predicted possibilities from its Move Selector; For all these candidate moves, perform Minimax Searching by the outputs from Evaluation Model
8	With probability P, use Strategy 7; With probability (1-P), use Strategy 2

*Figure 5.5. Summary of Selection Strategies*

## 5.4. Simple AI with Monte Carlo

First of all, we planned to build a simple AI use MCTS. There two main reason for us to build these simple AI as stated following.

One reason is that, after we finish the AI model by using Neural Network method, we can compare the performance between two kinds of AI. If our AI based on Neural Network has better performance, our AI has obtained good capability. The other reason is that in our Evaluation Network, we may need to calculate the winning rate of a chessboard using our AI.

To build these simple AI, first we shall build chess board that can move the piece per our input instructions. However, we must make sure that the input is legal. So, we have implemented move generator and validator to examine the inputs.

The move generator and validator works similarly in some aspects. First, we should define what kind of movement is allowed. Among all kinds of pieces, cannons are hardest to implement. For cannons, they are only allowed to capture an enemy by jumping over exactly one piece in a straight line no matter how many empty blocks exist among the line. Also, the knights of Chinese chess are slightly different from them on chess. If there exists a piece adjacent, it can't move to that direction. After that, the move generator will apply these move patterns to the piece and use validator to determine whether the move is legal.

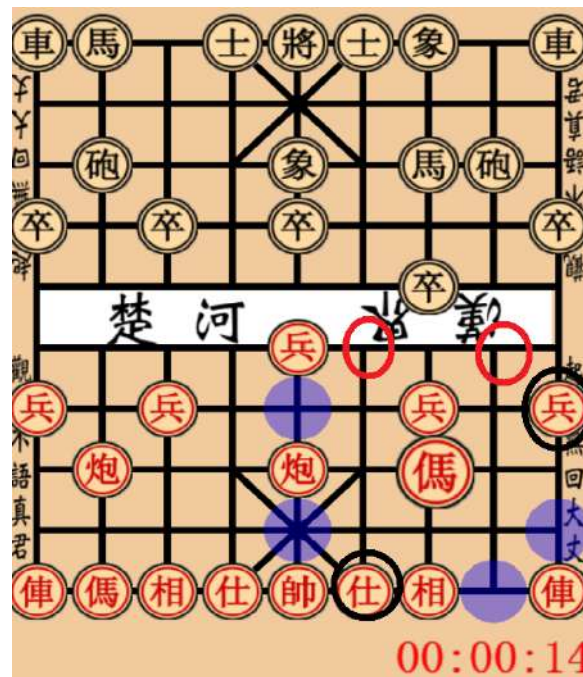


Figure 5.6. An Example of Move Generator and Validator

In Figure 5.6., all the circles are generated by move generator. And the validator will examine all these eight moves. The blue circles represent legal move of the knight. Black circles represent that there is piece of red side on the point. And red circles represent that a pawn blocks the way for the knight to move upside.

After we have these components, we should apply MCTS to Chinese chess by following steps below. For a given situation, first we find all the possible valid moves use our move generator. Then we shall traverse all of those moves. For every possible move, move forward and randomly select possible move until an end condition, usually when it reaches the maximum iteration depth and return the evaluation about the status. The evaluation of the chessboard mainly bases on the number of pieces exist, and different kinds of pieces have different scores, as shown in Figure 5.7..

Piece	Score
Pawns before crossing the river	1
Pawns after crossing the river	2
Advisor	2
Bishop	2
Knight	4
Cannon	4.5
Rock	9

*Figure 5.7. Values of Different Pieces*

Repeat the search for 10 times and use their average as the score of the move and choose the move with highest score. After this move, check if the current side is being checked. If the side to move will still be checked after the move, try to select a new move. Or we use this move as final output.

By this method, we have a basic AI which responses to easy game board. However, since the basic AI is very simple, it has some major disadvantages.

First, as the evaluation function of it is completely based on the value of weight. The highest priority of the AI is always trying to capture enemy's piece. For example, most probably, the first step of red side will be shown as Figure 5.8.. The red side uses its cannon to capture black knight.



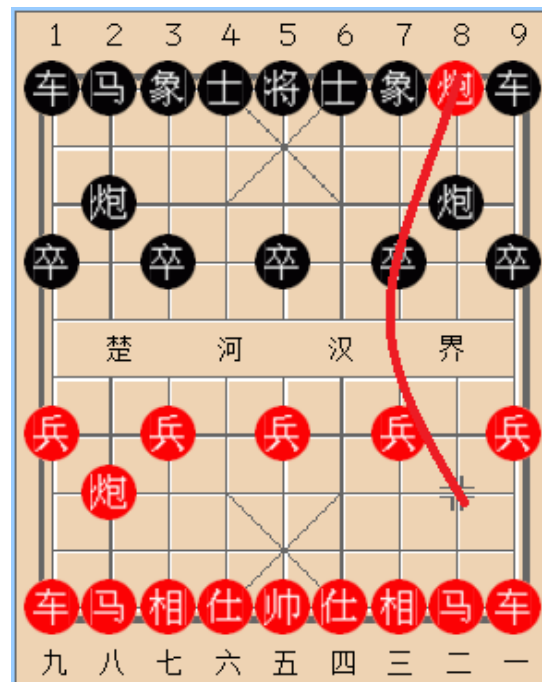


Figure 5.8. Choice of the Simple AI

And the response of black will be like in Figure 5.9., using its rock to capture red cannon.

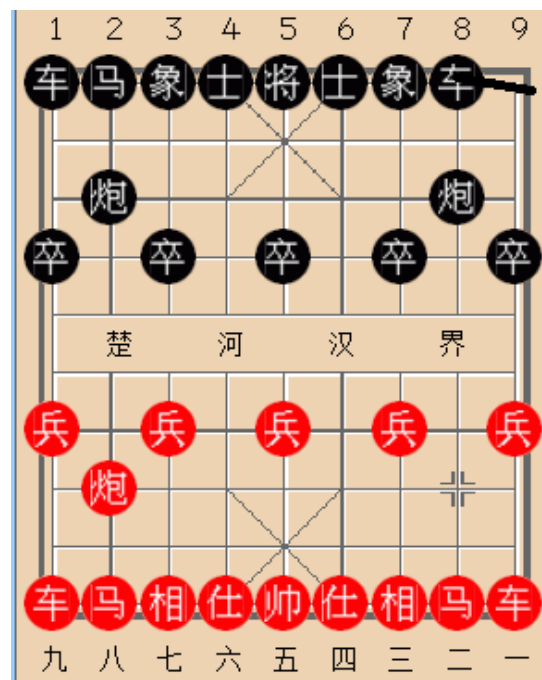


Figure 5.9. Choice of the Simple AI

The reason for this phenomenon is very simple. The algorithm only searches for one

layer, and find that it can capture enemy's piece. If it uses red cannon to capture the black knight and stay live, this move will have higher score. It doesn't see that the rock can capture red cannon as response because it is out of search boundary.

To solve this problem, we must add the amounts of layers of search tree before starting random search and use minimax to reduce possible search time. But this will reduce the speed of the program greatly.

## 5.5. Policy Network Model

### 5.5.1. Previous Design

At first, we did not come up with the idea to use Piece Selector and Move Selectors separately. Instead, we had some different designs.

One of the ideas is to use a vector to map all possible move for pieces. For example, a rock will have maximum of 17 moves in total (9 horizontal and 8 vertical) and a knight will have 8. And then serialize them according the order of relative moves. For example, for a knight, the move to front left would be k1, and the move to front right would be k2. Using this method, we would have 122 possible moves in total. And use it as the label. As the relative order of all moves would not change, we can restore the move from the vector. If the 75<sup>th</sup> label is correct, we can find the corresponding move. This method transforms the high-dimension move into a one-dimension vector suitable for neural network training.

However, this method is not very intuitive. The corresponding relations are hard to find even for humans. And for neural network, the training efficiency would be low for the same reasons. Another disadvantage is that the same relative move will have different value in distinct situation. Usually moving a pawn upside is a good move. However, moving it into the top line is not a good idea because it cannot return backward.

Because we had a better model later, this method has not been implemented.

Another model we used to implement is adding high-level information into the neural network. For example, the liberty of a piece would be considerable information for human chess player because a piece that has higher liberty would affect more space on the chessboard. We think that this kind of information can speed up the training process and make it faster to converge.

However, the result was not satisfying from our expectation. The accuracy even dropped

compared to the version without high-level information. The reason may be the meaning of this map is different from other channel, which confuses the neural network to make false prediction.

In fact, Neural Network can extract information internally and interferes of human are not always necessary. In our project, the feature channel of valid moves helped in training Piece Selector, because the meaning of the channel is clear and consistence with other channels. But a bad channel will cause overfitting or sheer drop in accuracy. So, we should take care of this kind of additional information.

For training evaluation model, we need to give them a label representing the current winning rate or the advantage. If we look at the status and score them one by one, that would be inaccurate and inefficient. So, we want to use MCTS to calculate the winning accuracy of the chessboard.

However, in our practice, the random game was really slow and nearly impossible to end. It is very obvious that a complete stochastic game has little chance to end. So, we tried different ending condition.

First, we tried to end the search by the time a check happened and calculate the amount of which side suffered a check. However, even if the number of pieces between two sizes was extremely large, the result was still neutral. Tracing the exactly branches of the search, we found that the side who had less piece would have less possible move, meaning the possibility of a suicide check would increase which never happened in a game between humans, but would cheat the evaluation progress. On the other hand, for the dominant side, as his piece was more than the opposite, a random choosing may select invariant bad move.

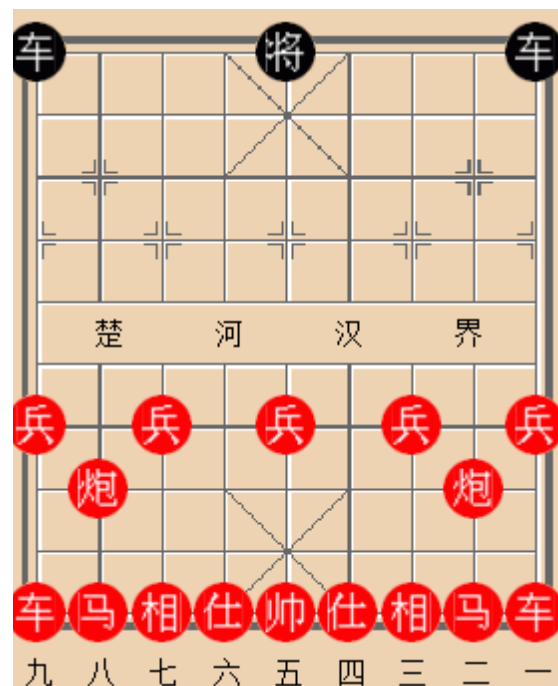


Figure 5.10. An Example of Extreme Cases

As shown in Figure 5.10., the black side has only two rocks remained. In this situation, the winning rate for red is nearly 100%. But in this method, the liberty of rocks means that they have freedom to move everywhere, where equally random selection make the move of red side awkward. In this case, the times of both sides are checked first are nearly equal. That's not good obviously.

Another way is not to use random selection, and use our MCTS AI instead. Though it may have good result, the speed of it is a disaster because the depth is too high and need too much instruction for next move. Dealing with a chessboard need more than 5 minutes. If we want to give the label to all data in the dataset, this may cost more than a year. So, this is also not a good way.

The last method is to regulate a maximum depth and return the final status after counting the live pieces of each side. However, besides problem from the first method, this evaluation method overemphasizes the importance of capturing enemy's pieces. The policy trained would be like simply calculating the live pieces with weight.

As long as the three methods are all unworkable, we decide to use result from Policy Network to replace random selection. As a result, the implementation of value network will be postponed until we finish Policy Network.

### 5.5.2. Model Structure

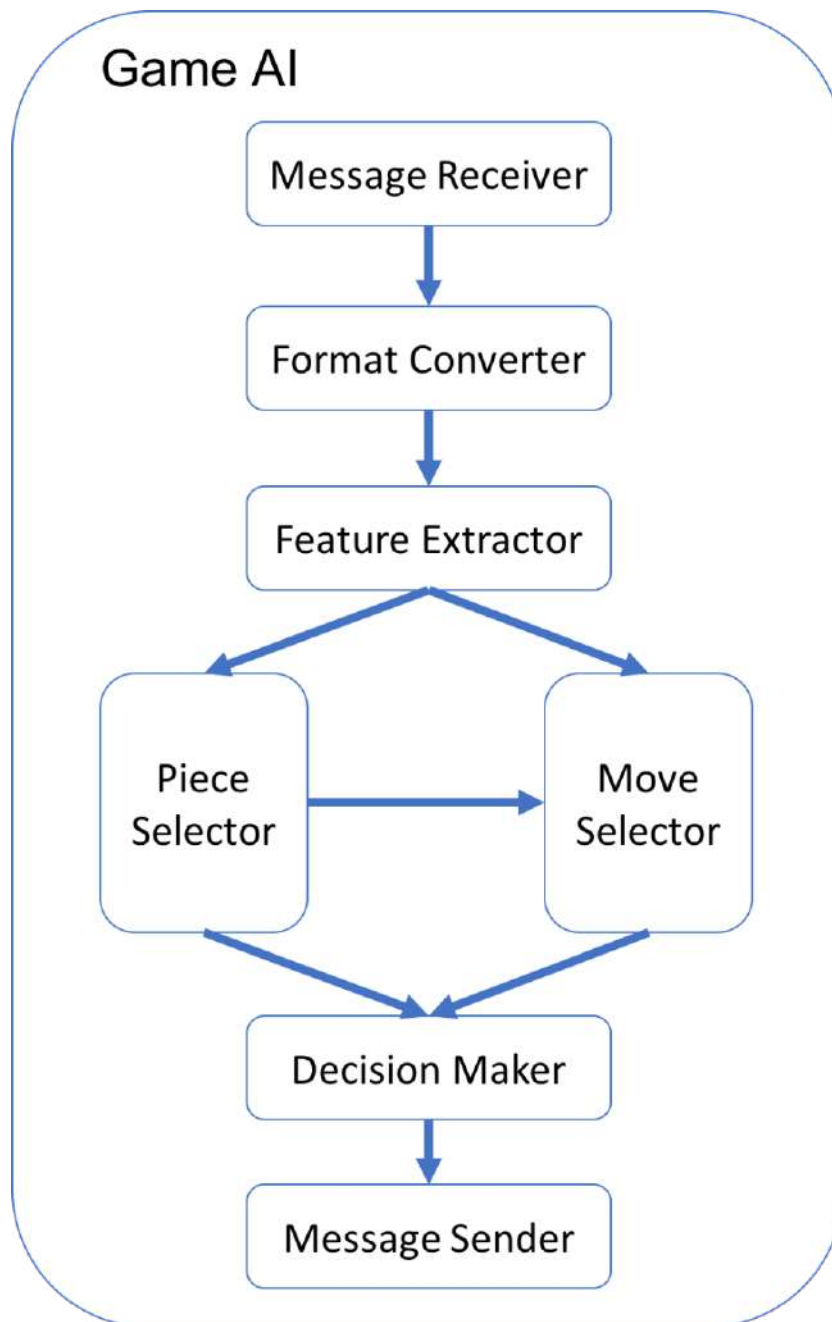


Figure 5.11. Structure of Game AI

As shown in Figure 5.11., the detail structure of our Game AI Version 001, it mainly consists of these components: Message Receiver, Format Converter, Feature Exactor,

Decision Maker, Message Sender, and most importantly, Piece Selector and Move Selector. The overall flow is: Message Receiver receives the FEN information from frontend via socket, and Format Converter preprocesses the information so that Feature Extractor can identify it and extract according features out. After that, Piece Selector and Move Selector together outputs the probability distributions of possible moves. At last, Decision Maker makes a move choice and Message Sender sends the choice back to frontend.

Both Piece Selector and Move Selector consist of five layers in Neural Network, as shown in Figure 5.12. and Figure 5.13..

The first layer is input layer. For Piece Selector, there are eight feature channels, so the size of its input layer is  $9 * 10 * 8$ ; while for Move Selector, there are nine feature channels instead, so the size of its input layer is  $9 * 10 * 9$ , which is the same for the Move Selector of different piece types.



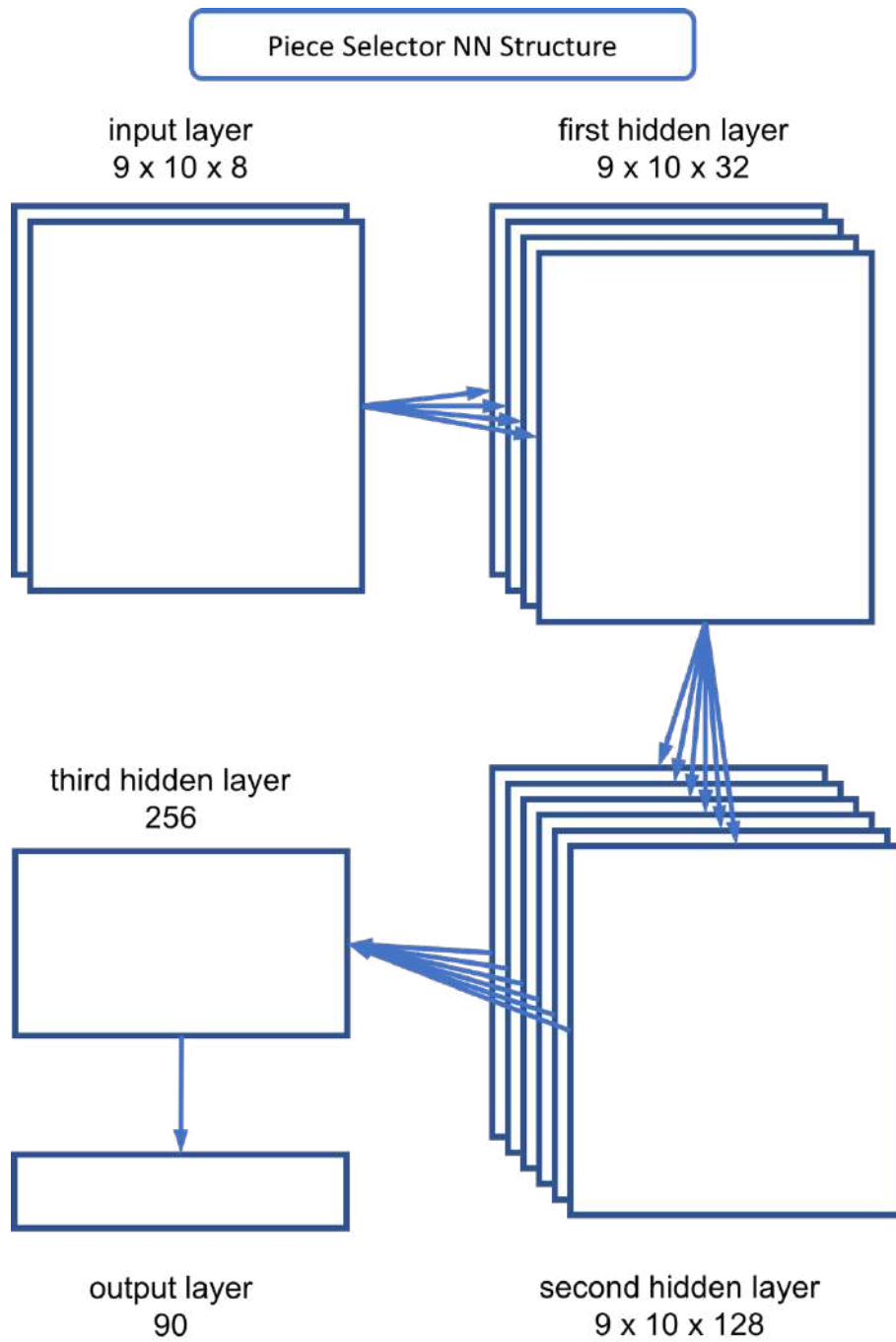


Figure 5.12. Piece Selector Structure

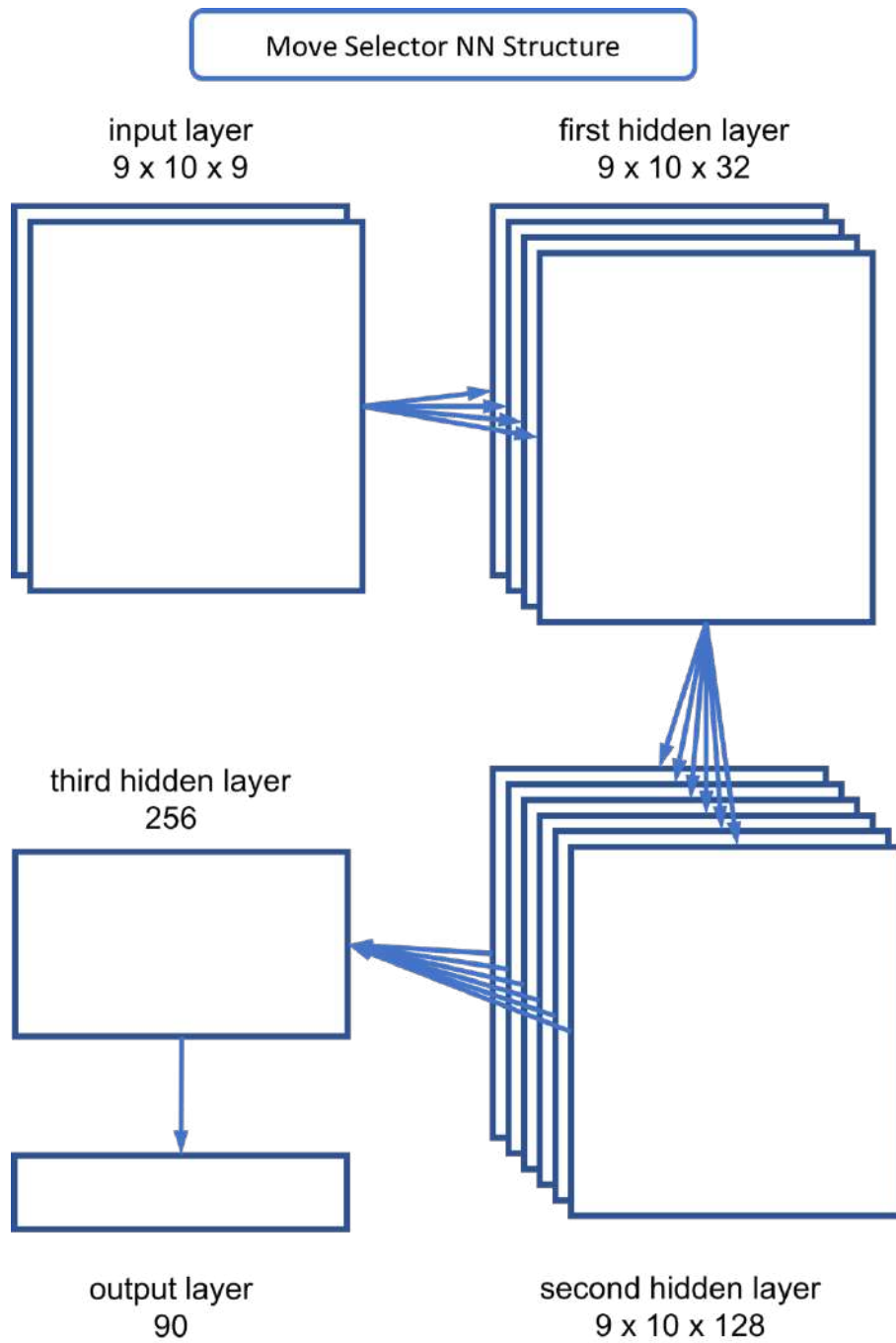


Figure 5.13. Move Selector Structure

There are three hidden layers in total, in both Piece Selector and Move Selector. The first two are convolutional layers and the last one is a fully connected layer.

The first and second convolutional layers were designed to have 32 and 128 feature channels respectively, so the sizes are  $9 * 10 * 32$  and  $9 * 10 * 128$  respectively. The size of filters was set as  $3 * 3 * \text{depth}$ . The third hidden layer is a fully connected layer with 256 nodes in it.

The reason for us to choose CNN is that there exist many common patterns in realistic Chinese chess games, similar with joseki in Go. Given a certain pattern, players will have relatively fixed solutions based on previous experience, which are usually considered as optimal. Undoubtedly, CNN works well in recognizing patterns seen from previous research results. Also, CNN can greatly reduce the number of parameters and accelerate training speed, compared with fully connected NN. That is why CNN is chosen to build our model, instead of fully connected NN.

The last layer is the output layer, which is a softmax layer. It will output an array of length 90 and the sum will be 1. After reshaping the output into  $9 * 10$ , we can get the probability for each corresponding position in the chessboard.

### 5.5.3. Selection Strategy

After getting outputs from Piece Selector and Move Selector, an algorithm is needed to make the final decision about move choice. After all, they only output probability distributions over all 90 positions. We designed several selection strategies and tested them respectively.

The simplest way to select a move is to select the piece with highest possibility given by Piece Selector and then select the destination of that piece with highest possibility given by Move Selector. This is Selection Strategy 1.

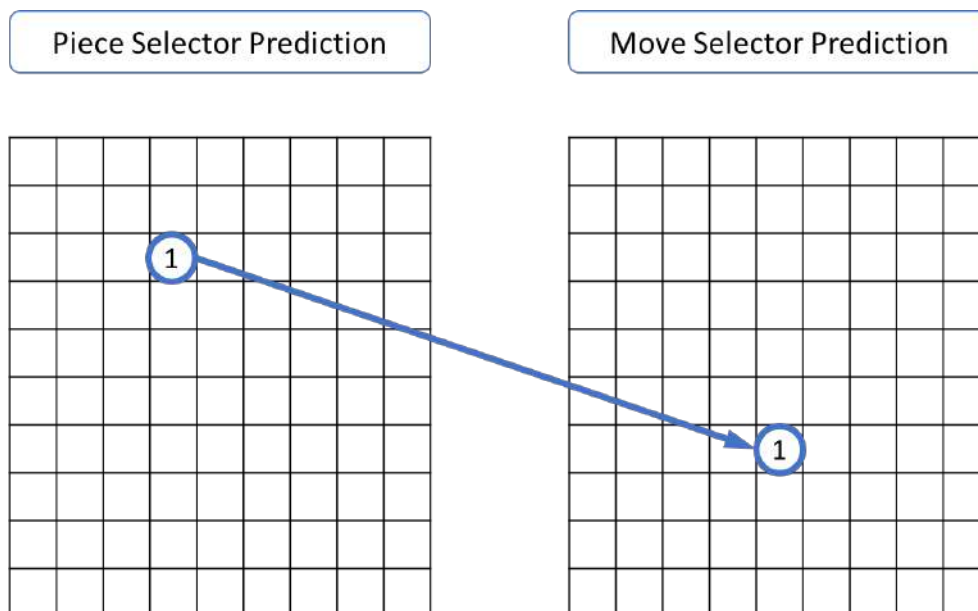


Figure 5.14. Selection Strategy 1

For Selection Strategy 1, it is the simplest method that we come up with at first. This strategy would guarantee the engine not to select some bad moves and perform well in normal cases. After all, the output is the predicted possibilities from the Policy Network after Supervised Learning, so a larger possibility means that it is more recommended by the model through learning experience from game records of master players.

It is a simple and straightforward strategy; however, the drawback is also very obvious.

First of all, it totally depends on the training results of Supervised Learning, which is

not flexible and may behave very bad when dealing with unseen moves. And the training results cannot be perfect, so not 100% reliable.

Also, it will consider the output of Piece Selector first and then the output of corresponding Move Selector, which may lead to a controversial problem when the probability predicted by Piece Selector is lower but the probability predicted by Move Selector is much higher.

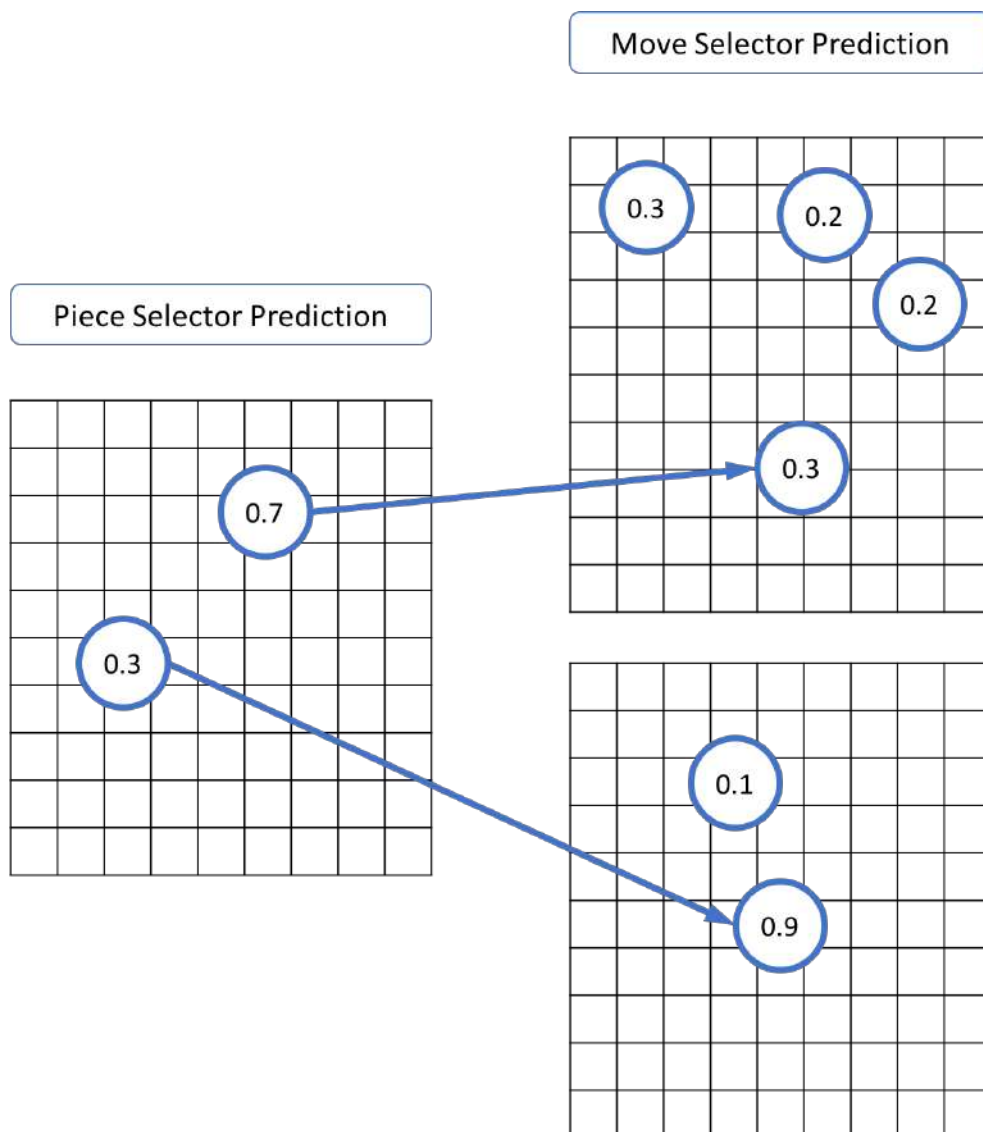


Figure 5.15. An Example of Drawback of Selection Strategy 1

For example, predicted by Piece Selector, 0.7 to choose Piece A, 0.3 to choose Piece B; but predicted by Move Selector of Piece A, around 0.25 to choose Move 1, 2, 3 or 4,

while predicted by Move Selector of Piece B, 0.9 to choose Move 5. In such a case, suggested by Move Selector, no move for Piece A is highly recommended while there is one move for Piece B is highly recommended, therefore maybe choose Piece B is a better choice, which will definitely be ignored by Selection Strategy 1.

To avoid the drawbacks as stated above, we need to use another strategy.

Considering the fact that the probabilities given by Move Selector are essentially conditional probabilities, we designed another selection strategy, maybe appearing to be more reasonable. Here, we don't separately consider the probabilities given by Piece Selector and Move Selector, but we multiply them respectively, i.e. the probability of moving a piece \* the probability of a destination of that piece, and then select the combination with highest probability.

In cases where there is one piece with relatively much higher probability given by Piece Selector, this strategy will much likely give the same result as the previous one. In other cases, however, where there are several pieces with all high but quite close probabilities given by Piece Selector, this strategy may perform better, as there is no clearly better piece to move and this strategy will consider more options.

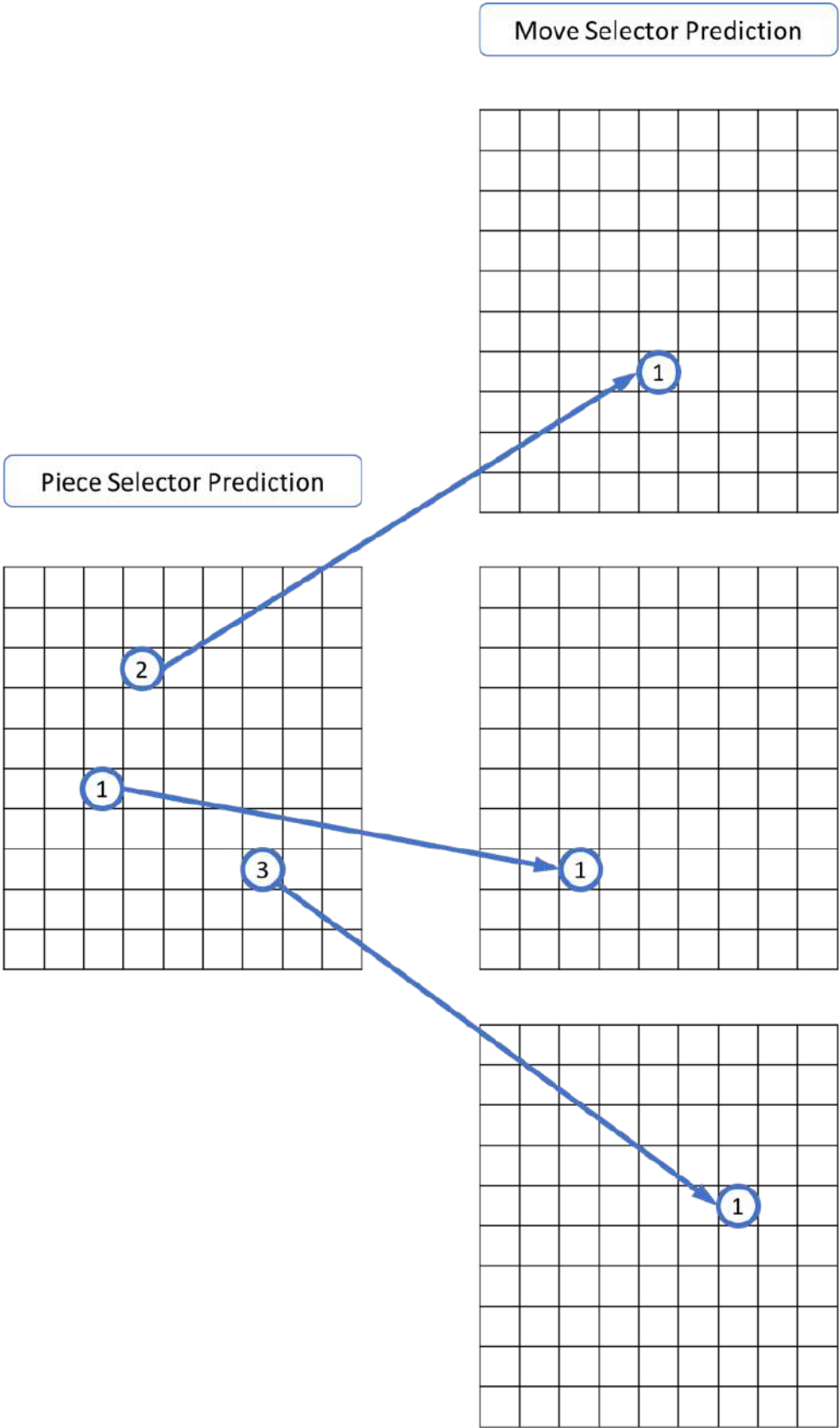


Figure 5.16. Selection Strategy 2

For Selection Strategy 2, it considers more pieces available. In fact, there exist many cases that two or more pieces have close predicted probabilities to be chosen. We need to further consider the move selector and where the pieces will move. As the highest move in one specific move selector will always have better possibility than other moves in the same move selector, we only need to select one from each move selector. And then give every move a chosen probability and select the one with greatest probability.

The advantage of this strategy is that it can make better choice when multiple choices with close probabilities are available. Also, it is more reasonable considering the essential of output of Move Selector.

One problem to be solved is that these strategies are not flexible and may behave bad dealing with unseen moves. To slightly solve this problem, these two selection strategies both can be modified by increasing the randomness. Previously, the decision is made by picking the one with the highest value, and every time met with the same situation, the AI model will make the same decision.

To encourage exploration, a random number between 0 and 1 will be generated, and the move will be chosen according to the probabilities predicted by Piece Selector and Move Selector. Also, this will help in Reinforcement Learning as it encourages exploring new moves instead of sticking with the output from Supervised Learning and then may help improve the performance and avoid becoming more and more converged to few choices.



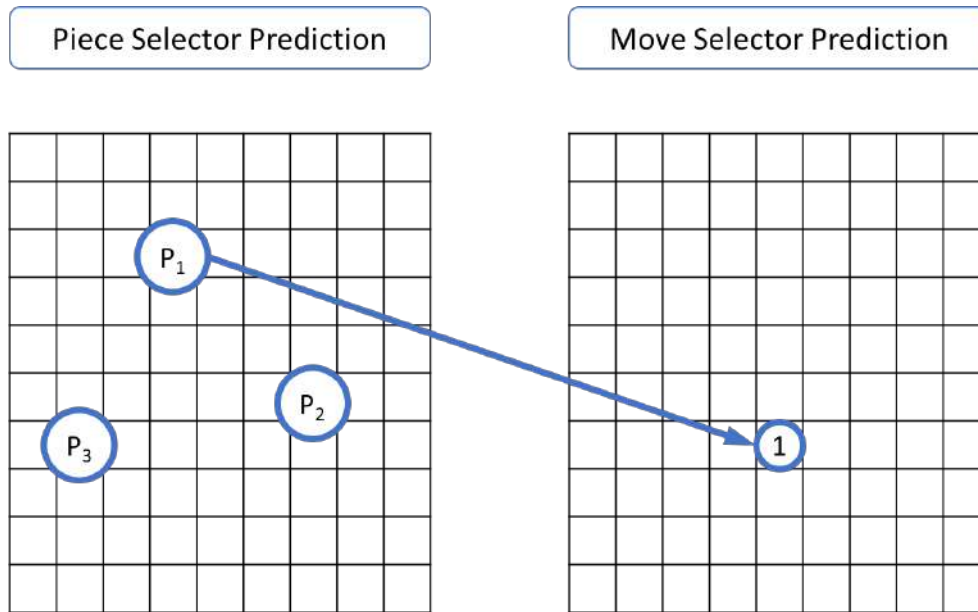


Figure 5.17. Selection Strategy 3

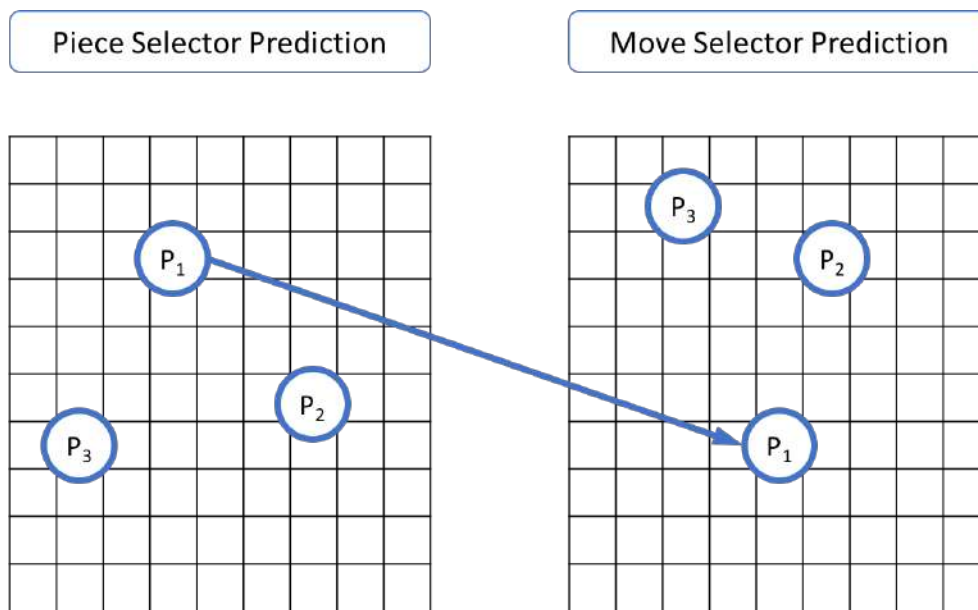


Figure 5.18. Selection Strategy 4

For Selection Strategy 3 and 4, they are similar and the only difference between them is to use the most possible move or randomly select one in the move selector. As these two strategy focus on making more variation in the chessboard, they perform well in training process because they can travel as much status on board as possible.

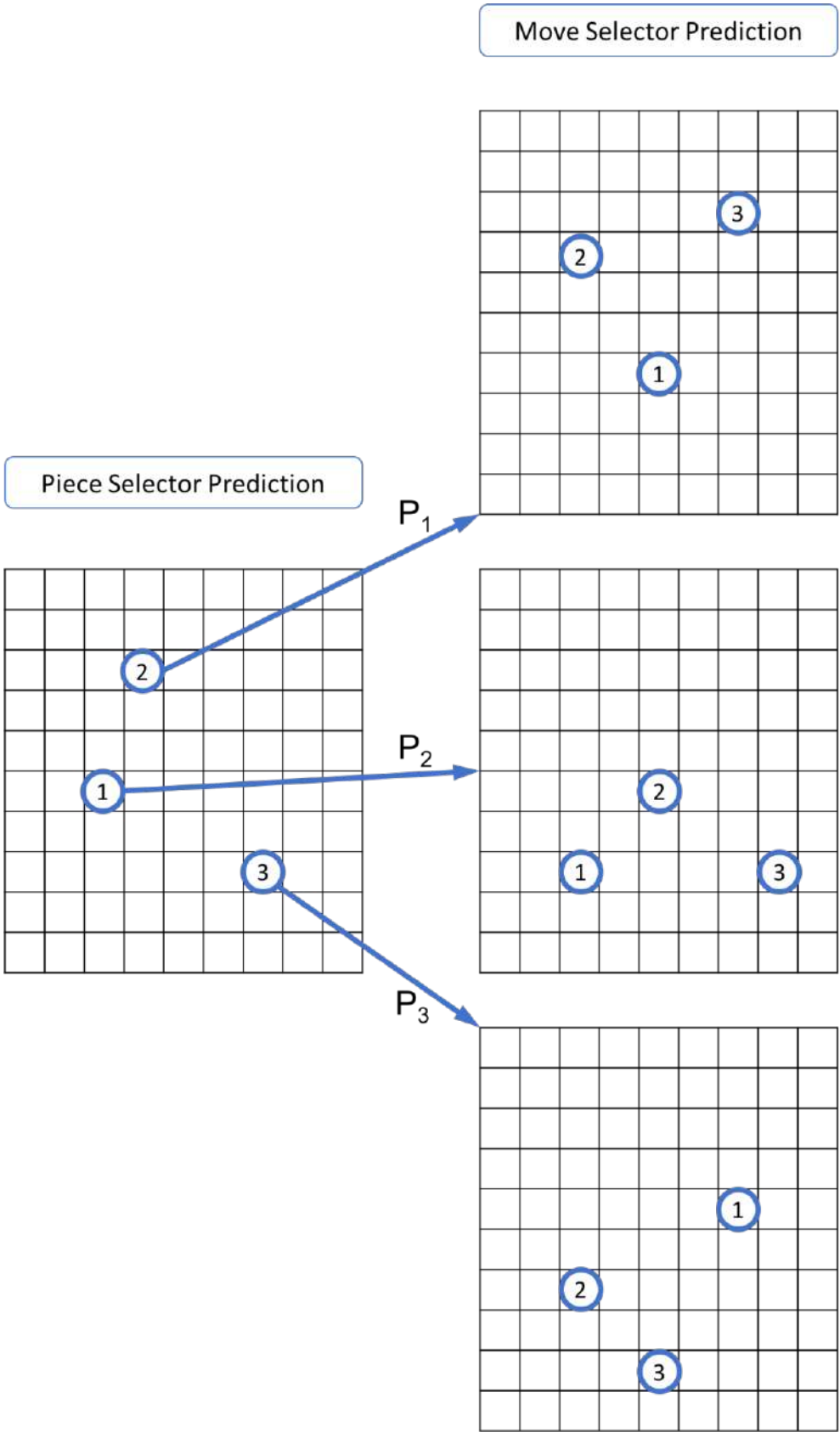


Figure 5.19. Selection Strategy 5

For Selection Strategy 5, it is an improved version of strategy 2. We take more moves into consideration. We define a threshold to exclude impossible moves like move a piece to a place that would be captured, and a max number of moves can be random selected so that it won't select a less possible move by accident. Then we use the value that the possibility of piece selector multiplied by the possibility of piece selector as the possibility the move may be selected. In this way, we make balance between new tries and known experience that fit the request of reinforcement training.

## 5.6. AI Model Version 018

Version 018 is a Policy Network model based on Version 001 trained by Supervised Learning while Version 001 is updated through Reinforcement Learning. Except that, the structure of Neural Network and the Selection Strategy used by the AI both remain the same.

The basic idea is that, based on the NN models trained by Supervised Learning, i.e. Piece Selector and Move Selectors, we let the AI engine compete with itself and update the model depending on whether it wins the game or not, using Reinforcement Learning. After several iterations, we let the newest version compete with some randomly selected intermediate version instead of the original one, and keep changing the version of the opponent in this way.

Here we met with one problem which is that, when doing Reinforcement Learning, we need to assign a reward to each move the model made.

Positive reward is quite straightforward: if the model won that game, then assign 1 as a positive reward to all the moves it made during that game. Of course, it is impossible for every move it made to be a perfect or good enough choice, even though it won that game. It is also very difficult, however, to detect which moves are good enough and which moves are terribly bad. After all, if we knew how to detect that, we could use that knowledge to build a chess engine directly instead of training Neural Network models to find out which moves are good choice. Therefore, it is reasonable for us to assign 1 to all the moves it made during the game it won.

As for negative reward, the situation is more complicated. Similarly, we can assume that since it lost that game, the moves it made are not the best choices, according to the same reason as above. When we tried to directly assign -1 to those moves, the results turned out not to be satisfying. The model behaved very strange, made illegal moves and even triggered errors during the training procedure. One possible reason is that although the

move may not be a good choice, at least it is better than illegal moves. So, directly assigning a negative value to the selected moves is not a reasonable way to assign negative rewards. And it also seems not to be reasonable to assign 0 or some small value to those moves. Then, thinking in the other way, we decided to assign a positive value to the moves that were valid but not selected. Although it appeared to be more reasonable, the results are still not satisfying and trigger errors in training sometimes.

In the end, as discussed above, we decided to give up negative rewards and use positive rewards only.

With such a reward assigning strategy and training procedure, we conducted 8 iterations (excluding the failed trials), in total around 40,000 games, over 2,000,000 moves, and get Version 018 model.

## 5.7. AI Model Version 018.1

Version 018.1 is the AI model combining the previous Policy Network model, i.e. Version 018, with a new Evaluation Network model.

When testing the performance of Version 018, though it is better than previous versions, the results are not as good as we expected. Instead, it seemed that the model made little progress after all the Reinforcement Learning. Similar with the original version, it appeared to arbitrarily make moves, have short insight, have no idea about attack and defense, respond terribly to being checked and so on. There even are some bugs that sometimes it will make illegal moves.

After analysis and discussion, we think the reason why the model did not improve much through Reinforcement Learning is that both its opponent and itself are too weak from the very beginning, and in order to do the training, we add some randomness to the move selection procedure, making them even weaker. So, it basically can learn nothing from the training as those choices are essentially not worth learning. But we can't let it compete with a strong engine as it is too weak and it will certainly lose all the time while negative rewards do not work well, which makes it impossible to improve as well. We kind of got stuck here and hardly made any improvement.

To further improve the model, we need to implement the Evaluation Network.

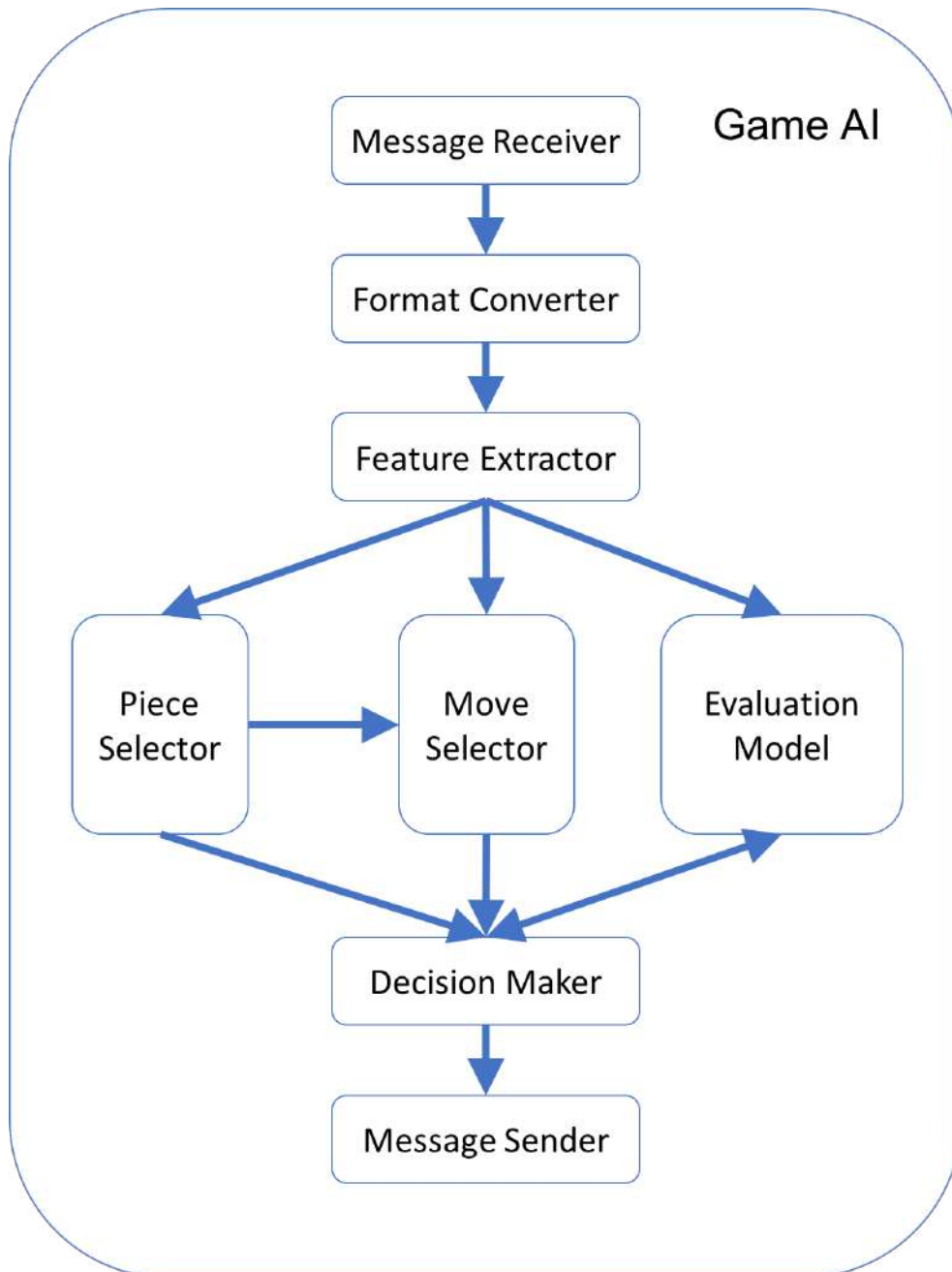


Figure 5.20. Structure of Game AI Version 018.1

As shown in Figure 5.20., except for previous Policy Network models, a new Evaluation Network model is added inside the Game AI. It takes some features of chessboard status as input and its output will help the Decision Maker to make move choices. The related details come as following.

### 5.7.1. Evaluation Model

To strengthen the model to some reasonable level, instead of persisting in Reinforcement Learning, we decided to train an evaluation model to help the model make better choice and then go back to Reinforcement Learning.

First problem to solve is how to evaluate a chessboard status and assign a score to indicate the relative advantage for either player.

To train a Neural Network model, we need to feed in the target value for each corresponding training case. Certainly, the evaluation function should be very sophisticated and requires lots of knowledge of Chinese chess so that the score can indicate the winning rate to some extent rather than being useless to refer to. Considering the fact that both my group mate and I know little about Chinese chess, we need some extra help. We found an open-source Chinese chess engine on the Internet and call one evaluation function inside it to get the scores for chessboard statuses.

In this way, we obtain around 1,000,000 training data to train the evaluation model by Supervised Learning.

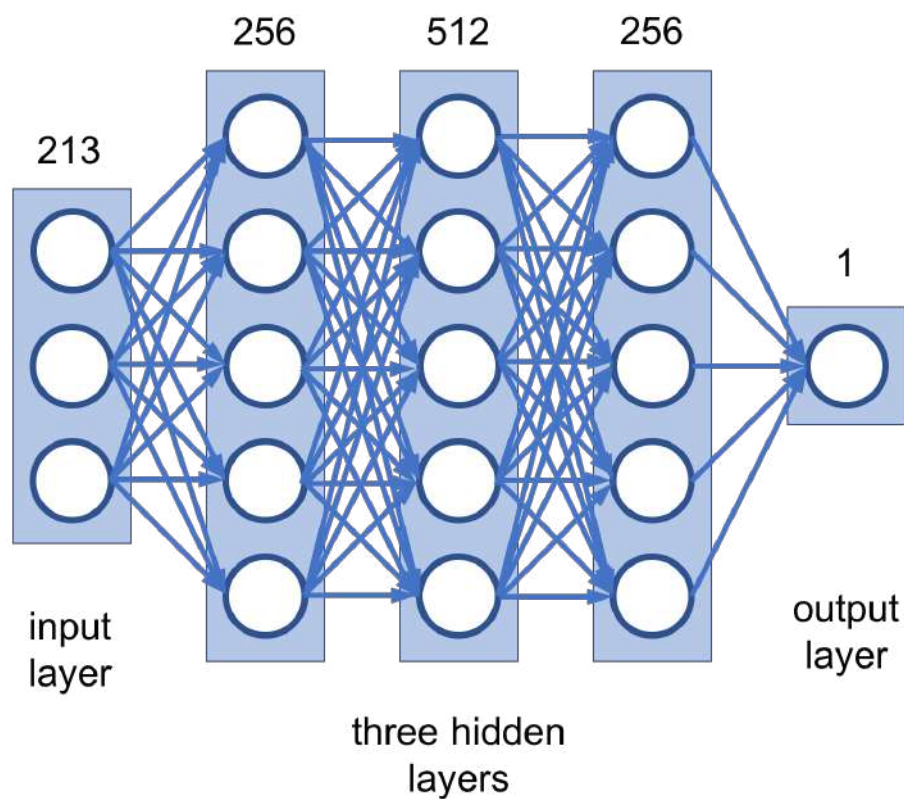
After Supervised Learning, we obtained our own Evaluation Model. The next problem is how to use the results of this Evaluation Model to help the existing AI make better move selection. Therefore, new selection strategies are needed. The structure of Evaluation Network and new selection strategies are explained in detail as following.

#### 5.7.1.1. Neural Network Structure

Our Evaluation Model is a fully-connected Neural Network, with five layers in total, one input layer, three hidden layers and one output layer. For input layer, there are 213 nodes. For three hidden layers, there are respectively 256, 512 and 256 nodes. For



output layer, there is one node.



*Figure 5.21. Structure of Evaluation Network*

This Neural Network model takes an array of length 213 as input which represents some features of the chessboard status, and output one score which indicates the relative advantage of current player. In general, a positive score means that current player has some advantage over the opponent. The larger the score is, the more advantage the player has. More advantage usually means a higher winning probability, given the two players have similar capabilities.

### 5.7.2. Selection Strategy

For Version 018.1, we first used such a selection strategy: First choose some relatively good move candidates by the predictions of Piece Selector and Move Selectors, i.e. moves with high possibilities suggested by Piece Selector and Move Selectors; Then pass all those candidates to Evaluation Model to evaluate the chessboard status after the player makes that move, and finally we can pick the one with lowest score, indicating the highest winning rate or largest advantage over the opponent by making that move. This is Selection Strategy 6.

For Selection Strategy 6, we use the evaluation model to examine the status on the chessboard after each move, and select the one with lowest score because it is the opponent's turn after this move, so a lower score means the opponent has less advantage after we make this move.

The main advantage of this strategy is that it uses Evaluation Model instead of random selection to make the decision. Using Evaluation Model will help the decision maker to look in the different view, and it can choose the move with greatest winning rate.

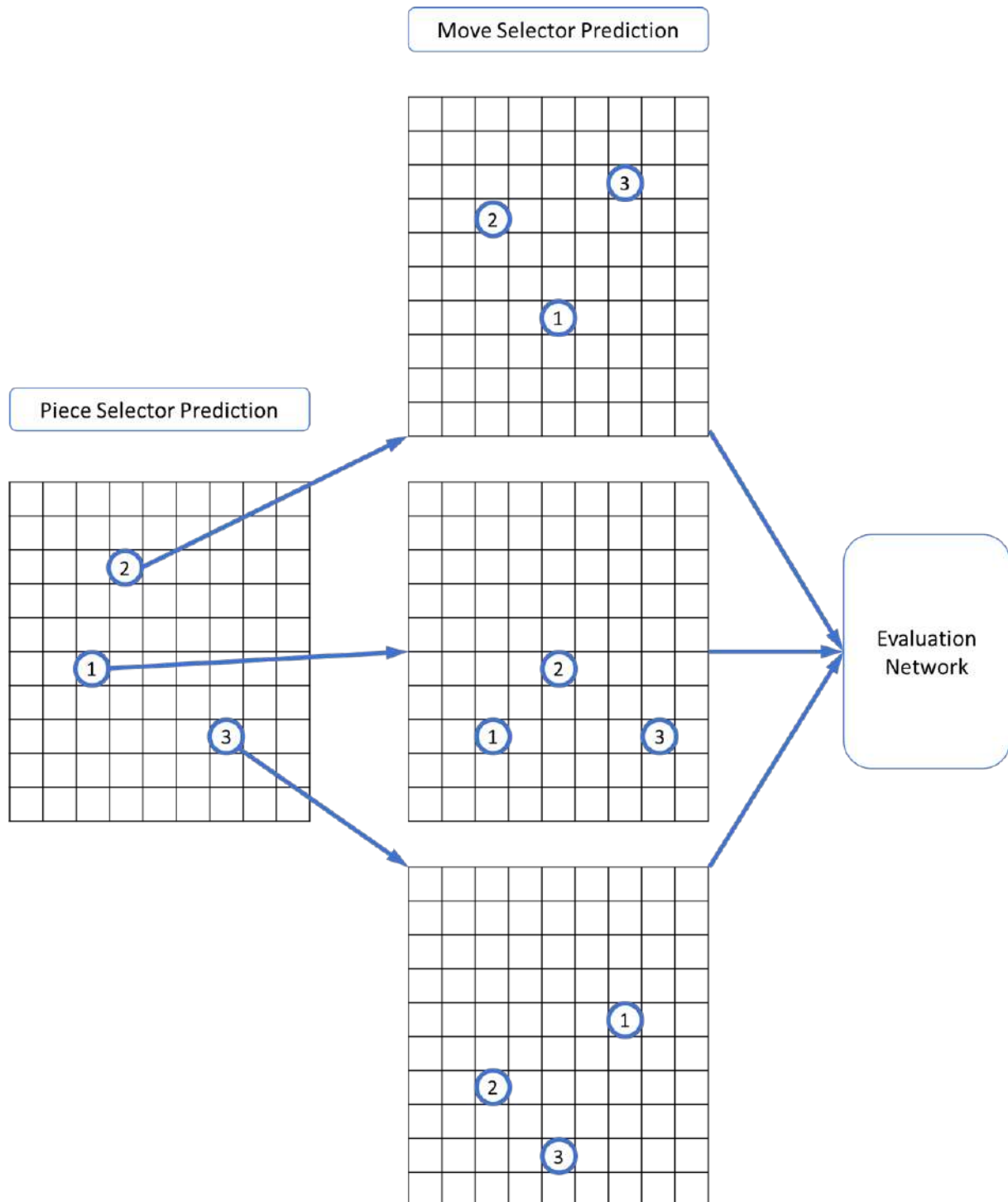


Figure 5.22. Selection Strategy 6

One important thing to notice here is that we cannot use Evaluation model to evaluate the chessboard status for the player after it makes a move and choose the one with highest score. For that state, the player has made a move, and a high score hardly indicates larger advantage as the player cannot make a move again and the opponent

may change the situation a lot and turn disadvantage into advantage. Such a kind of scenarios is quite common in Chinese chess, as if you can capture Piece B by Piece A, usually the opponent can capture Piece A by Piece B as well. So, which side to make the next move matters here.

However, the evaluation model does not have ability to look forward, which means that it cannot deal with a series of moves. What's more, as it only evaluates the chessboard after its move, so it will have high possibility to capture opposite pieces and ignore if the pieces will be captured because evaluation model would consider the numbers of pieces but not the numbers of pieces will be capture.

## 5.8. AI Model Version 018.2

After combining the Evaluation Model together with our previous Policy Network, the AI model indeed improved its performance, but still not as good as we expected. Therefore, we may need a smarter selection strategy to make the decision.

In our project, we implemented one commonly used method, Minimax Searching, when using Evaluation Model to evaluate the chessboard statuses, to help our model make better and wiser move choices.

The basic idea is that when you choose a move, you should look a few steps ahead instead of only focusing on the immediate loss and win. For example, when you can check your opponent by moving your Rock forward, which is good, but meanwhile your opponent can capture this Rock by Bishop in his/her turn, then basically you should not make this move if you can look one step ahead. Similar cases happen very often in Chinese chess games, so such an ability is very important.

By Minimax Searching, our model can evaluate possible moves of the opponent in next several turns so that it can avoid making some stupid moves, have basic idea of attack and defense and so on.

With the assumption that the opponent is rational and always make best moves, our model will try to minimize the advantage of the opponent after the move, or equivalently maximize our own advantage after the opponent makes a move. In this way, the performance is sharply improved.

One problem we met here is that to perform better, the model needs to search deeper by Minimax Searching, which will largely slow down the process.

On one hand, if the model spends much time making a decision, even if the decision is better, it is also not tolerable, as there is some time requirement in real Chinese chess competitions. On the other hand, if we pass too few choices to Evaluation Model, it will

be much likely that the best move, in long term sense, is missed out in the very beginning.

Therefore, first we added a parameter called precision as the maximum number of candidates passed to Evaluation Model so that we can easily control the performance by changing the limit. Secondly, we assign different precisions to different layer of searching, larger in the first one and smaller in the rest.

As the number of nodes of the searching tree increases exponentially, assigning smaller precision in deeper layers can help reduce the amount of calculation a lot. By assigning a larger precision in the first layer, we are trying to ensure that no important moves missed in the very beginning. As the effects are actually decreasing as the searching goes deeper, we will not weaken the model a lot by pruning more possible moves in deeper layers.

Last, we assign different quota to pieces of different types.

Actually, the number of theoretical possible moves for each piece type is different. For example, King has at most 4 possible moves, Bishop has at most 2 possible moves, while Rock and Cannon can have at most 17 possible moves. So, we can assign larger quota to Rock and Cannon but less quota to Pawn and Bishop and so on.

By treating different pieces differently, we can slightly reduce the amount of calculation required and prune some meaningless moves which may be included when assigning the same quota to all pieces, hence improve the general performance of the model.

### 5.8.1. Selection Strategy

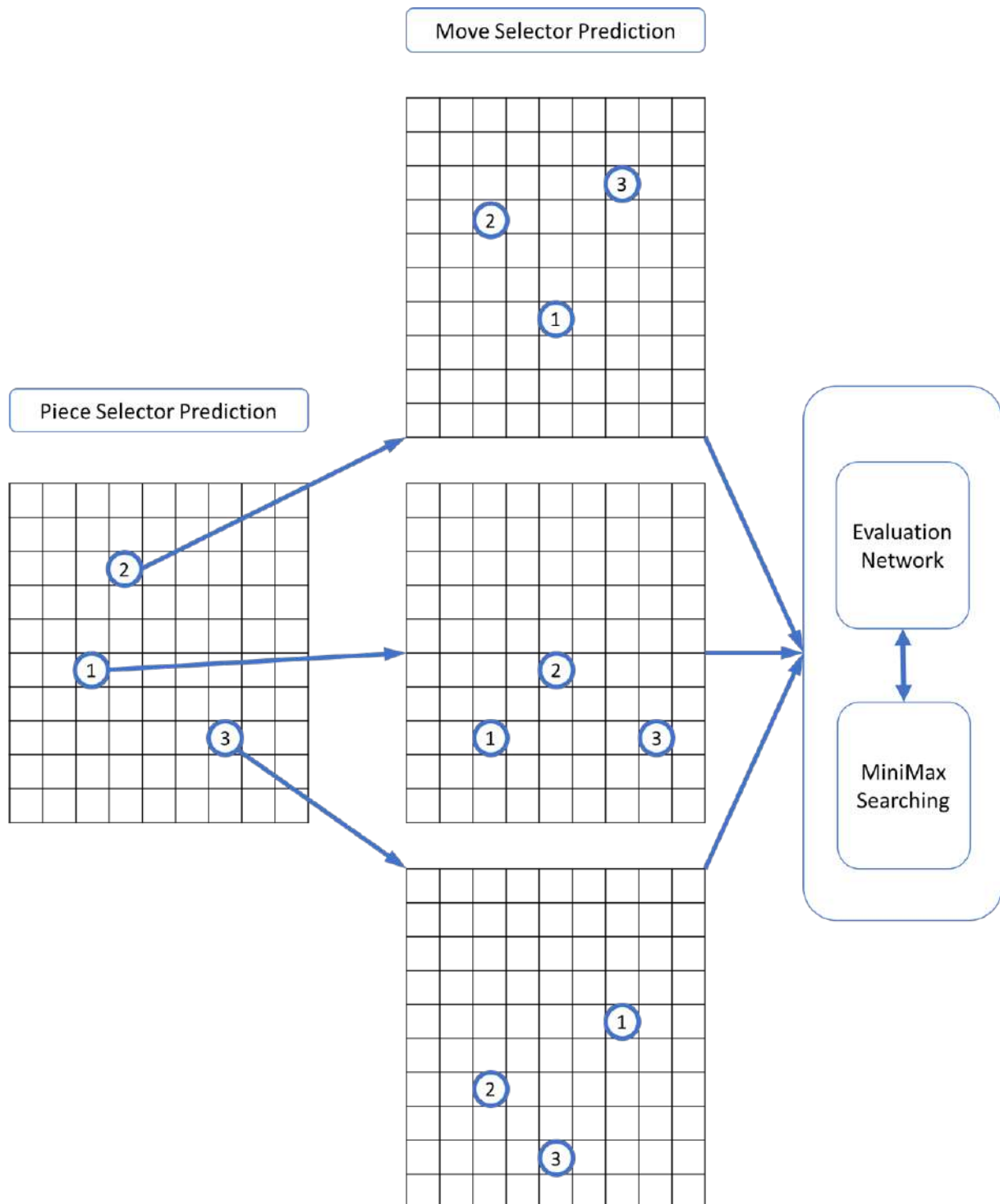


Figure 5.23. Selection Strategy 7

For Selection Strategy 7, we use minimax search to solve the problem. Minimax search, as we introduced before is a very powerful tool in this kind of chess game. Like strategy

5, we define a maximum branch number to reduce the amount of branches and increase. For each piece selected by the piece selector, we use a quota according to the types of the pieces as shown in Figure 5.24. By the ability of crossing rivers, we can separate the pieces into two groups, defensive pieces including advisors, bishops, and aggressive pieces including rock, cannons and knight.

Piece type	Quota
King	4
Advisor	2
Bishop	2
Rock	5
Cannon	5
Knight	4
Pawn	2

*Figure 5.24. Quota of Each Piece Type*

The king must have greatest freedom because it is the most important piece. If it is checked, we have to try every possible move to evade or we will lose the game. So, we set the number 4, which is the maximum number of moves a King can make.

For the advisors, normally they have only one legal move, but their moves can efficiently defend the king. When one advisor is at the central of the palace area, it may have more than one possibility. And the place it moves to is extremely important. So, we set the quota to 2.

For the bishops, they are similar as advisors. For the same reason, we set 2 as their quota.



For the pawns, before they cross the river, they can only move forward. Different from the chess, pawns in the Chinese chess can move horizontally but they cannot promote to another kinds of piece. The pawns in the Chinese chess are not so important. So we only give them quota of 2

For the rocks, they are the most powerful pieces in the Chinese chess as they can move straight along a row or column and they usually have much freedom to move. The maximum number of movements for a rock is 17, much more than defensive pieces like advisors and bishops. Also, the variation of the evaluation on the chessboard is high as the rock move to different places. So we need give it high amount of quota.

For the cannons, it is most difficult for the neural network to understand because they have two move patterns. One is for normal move like rocks moving straightly, the other one is move across only one piece in the middle and capture enemies' piece. Also, some traditional checkmate methods need the involvement of cannon. For example, the famous “马后炮” (cannon behind knights) needs cannon. So, we give cannon the same amount of quota like rocks.



Figure 5.25. Example of "马后炮"

For the knights, although they are also aggressive pieces, they have no so much possible movements like those pieces move straightly. The knights will also be blocked if there are pieces near it. So, we don't need to give them large amount of quota as they usually do not have so many places to move.

For the deeper layer of the minimax search, we reduce the quota of the aggressive pieces by half, because moves in these layers have less influence to the current chessboard. But for defensive pieces, the quota cannot be reduced. Or they may miss

only choice that evade from check.

For the strategy 8, it is a hybrid of strategy 2 and strategy 7. The reason to use it is the output from strategy 7 is unique and not suitable for training, so we try to add some randomness to it.

## 5.9. Other Improvements

### 5.9.1. Deploy Web Server

In the first term, our server could only run on local machine. So we were the only ones that could play game with it.

In the second term, we deploy our server on the Aliyun platform so that people can visit it outside the university campus. Every time the chess program receives a message from the server, it will record the current chess board and the move by the neural network model. And we can invite other people to play game with it and collect data for model training.

The reason we use Aliyun is that it has servers and data centers deployed at Hong Kong so that we can visit and deploy our server quickly. Also, using cloud services, we only need to consider our own code. In the old times, there are many trivial problem needed to be considered. For example, when the site is unvisitable, the owner needs to detect where the problem happened, and the crash may happen at program, server machine, or network devices. And if the problem is at hardware, he has nothing to do but wait for repairing. After using the cloud services, if the problem happens at the hardware as well, the cloud service provider can switch to the backup servers so that it can retain services.

In the project, we use Aliyun and the IP address is provided. The address of our website is <http://47.90.92.157::3001>

### 5.9.2. Multiple Login

Initially, our server is designed for one versus one game. But when we deploy it onto the Aliyun, we need to modify our server to ensure it can work well even if there are multiple users playing games with our engines in the same time. Our original designed is that the frontend on browser will send a message including current status on the board to the server, and the server should catch the message, calculate the next move and return a message to the browser.

However, the return messages are broadcasted to all the browsers connected with the server. But they cannot distinguish whom the return message is sent to. And if two users make their moves in the same time, the browsers will get confused and perform unpredictable.

To solve the problem, every time a user visit our website, we create a specific number for him and use it as the identify number. And every message will contain the identify number. In this way everyone can know the message recipient.

### 5.9.3. Reducing Responding Time

As our game engine uses minimax searching, the efficiency of our model is very important because the number of branches in the searching tree grows exponentially as the depths of search increase.

At first, even a simple calculation of piece selector will take 1 second time, and when we search further, the response of the engine is extremely low that we have to wait for over a minute for a move. That's too slow and unbearable.

In our old approach, every time we needed to use a model, we had to initialize a session to calculate the predicted probability. However, it needed time to load the model and to close the model. In fact, the process of calculation is very fast because it happens in the CPU, but the initializing and loading process is slow because the CPU has to wait for I/O with hard disk.

To solve the problem, we refactored our code to an object-oriented approach. Right after the time that we call the chess engine, it will create a class instance for every model, including an interactive session of TensorFlow. And we reserved interface for the chess engine to invoke. As a result, we don't need to open and close the sessions frequently and the time of each prediction is reduced to 1/10.

#### 5.9.4. Auto Training

The process of reinforcement training always needs lots of time. Normally, it needs days of training to get an iterated version. After then, we need to restart the training program and set new environment variables. Although the GPU machine in the department has very high performance, we cannot precisely predict when the program will finish. And it will waste lots of time if we cannot restart it in time.

To solve this problem, we decided to let it run automatically. There are some major problems in the running process.

The first problem is that TensorFlow has a stack structure for the interactive sessions. If we want to close a session and reload models with incorrect sequence, the error message will appear like below:

```
“Nesting violated for default stack of <class  
'tensorflow.python.client.session.InteractiveSession'> objects“
```

This bug means we have to modify and control the time and sequence of the interactive sessions. In our final version, we use a sequence to control the initialization the sessions and load model. And use inverse sequence to close the session

Another problem is that the games played by the training program are similar in the first. This will make the trained model overfitting and not flexible to different situation. The reason of this phenomenon is that our chess program behaves consistent if the opponent selects the same move. So, we define a factor  $p$ . With the probability of  $p$ , it will select the best move and with the probability of  $1-p$ , it will randomly select a move. In this way, the model can learn from different kinds of situations.

## 6. Training Process

### 6.1. Supervised Learning of Policy Network

#### 6.1.1. Training Dataset

We collected records of over 30000 Chinese chess games and about 2,000,000 moves in total, including games in professional competitions, classical ancient games, and online games between high-ELO players. As it was unable to determine an optimal or good enough move given certain chessboard status, the moves in collected were recognized as reasonable good moves and used as the supervised labels in training process. The objective of our supervised learning process was to train the model to predict the choice of professional experts given certain chessboard status.

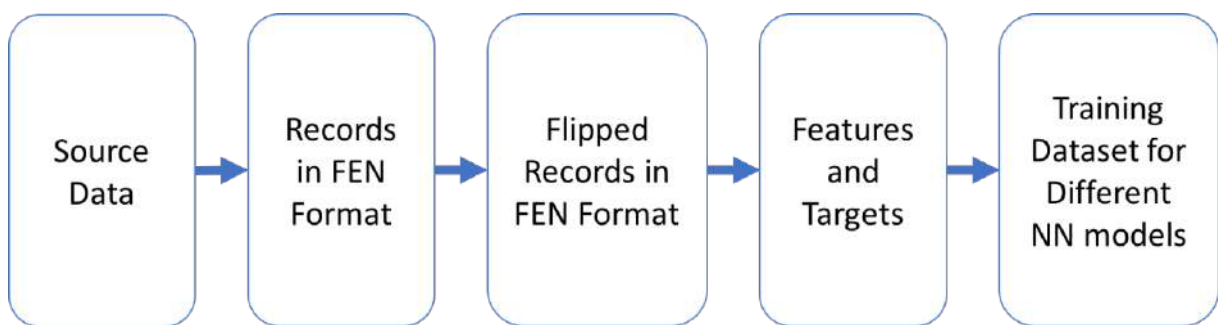


Figure 6.1. Generation of Training Data

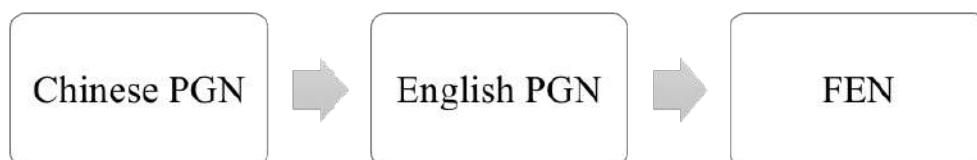
The source records downloaded from online libraries are in PGN format, which cannot be directly read by program. The PGN source data were firstly preprocessed and converted into FEN format. Then, the training dataset was generated by extracting the features of chessboard from FEN representation. The position from which a piece is moved is used as supervised label for Piece Selector and the position to which a piece is moved is used as supervised label for Move Selector. And for Move Selector, we need to classify the moves by the types of pieces moved. Though the total number of moves with chariot, cannon and horse is obviously greater than the moves with other types, the possible moves of these pieces are also greater than king and advisor, so training samples are enough for Move Selector of different types of pieces.

### 6.1.2. Preprocessing

After collecting source data of game records of Chinese chess, they cannot be directly used for training NN models, without being preprocessed.

For chess, we can find game records in PGN format which is easy to read and interpret for computers, and every move is represented by the type of the moved piece and the position where it is moved to. However, for Chinese chess, the game records are stored in Chinese version of PGN format, like “炮二平五”, and the place the piece will move to is only represented by its X coordinate while the coordinates of black and white are adverse, which is much harder for computer to directly process. Besides, PGN is not preferred for training usage, as only the whole PGN records sequence recording a game from start to end can make sense, each of which records only one move but not the status of the whole chessboard. However, our model is designed to be trained by each move, not each game. Considering this, FEN is a much better record format, as one FEN record contains quite complete information, of both the move and the chessboard status, for training usage.

To solve this problem, certain preprocessing is necessary to generate the training dataset. Firstly, we wrote a program to convert the records in Chinese into symbolic representations using only English letters and numbers, avoiding potential problems in coding. And then, we created an initial chessboard, followed the PGN records to make moves step by step and recorded every chessboard status in FEN format, which could be conveniently used for future NN model training.



*Figure 6.2. Format Conversion*

There are many special cases to be considered. For example, the phase “进五” will



have different meaning for different types of piece. For chariot and cannon, it represents move to five blocks forward. But for other pieces like knight or bishop, it will mean move to a block with X-coordinate 5. Another situation to be dealt with is when moving a piece with multiple this kind of piece in a row, for normal piece, the character “前” (front) or “后” (back) should be used. But for bishops and advisors, it won't do that because the available blocks for them are limited.

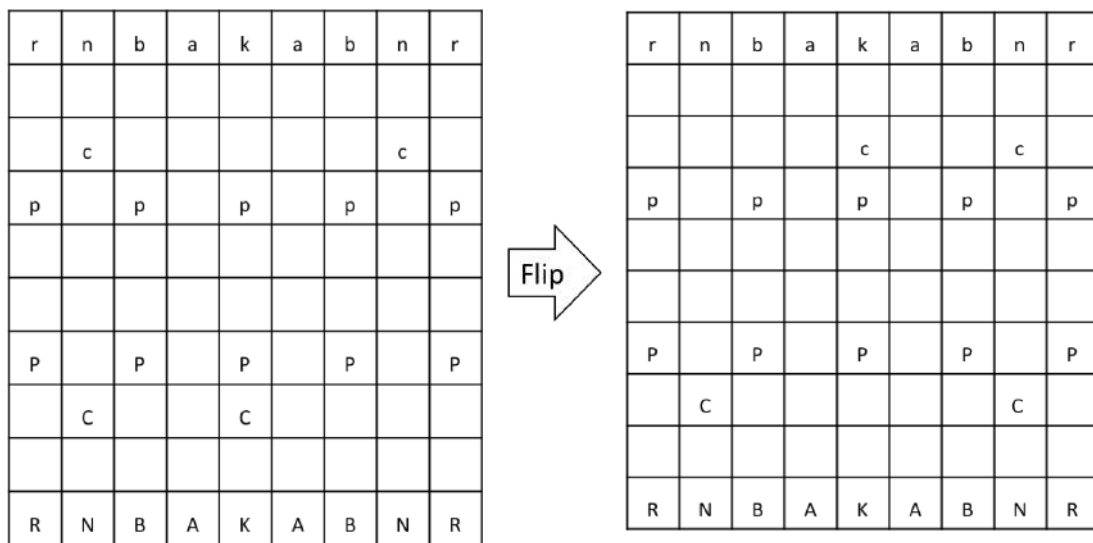


*Figure 6.3. A Special Case in PGN*

In Figure 6.3. above, two advisors are all in row 6, but if the next move is “仕六退五”, only the upper advisor can move backward. So, the upper advisor will be moved.

### 6.1.3. Chessboard Flipping

When preprocessing the source game records, a small trick was used, called flipping. Since there are two players in the game, to diminish the effect of different sides and accelerate the training speed, the chessboard would be flipped when generating the FEN information to ensure that the player to make next move is always the lower side.



*Figure 6.4. An Example of Chessboard Flipping*

For example, as shown in Figure 6.4., now it is the turn for the black side, i.e. the upper side to make next move, then the chessboard will be flipped so that our model can treat it as a turn of the red side.

#### 6.1.4. Training Strategy

Piece Selector and Move Selector were trained separately. Piece Selector was trained first, and after the accuracy of Piece Selector was over 40%, we started to train Move Selector. For Piece Selector, the training target was the position of the piece selected by the expert players under each chessboard status. For Move Selector, as mentioned above, seven different NN models were trained separately, one for each type of pieces. The training dataset only contained the moves where the expert players selected pieces of that type, and the training target was the destination of that move.

The training dataset was divided into batches of size 1000. The models were trained batch by batch, and every 50 batches they would be tested based on the next batch to be trained and the accuracy would be records. Also, a testing dataset was prepared containing about 1,000 games, near 100,000 moves and the trained models would be tested using this dataset at last. If the models also perform well in these unseen situations, we can safely conclude that the models are not overfitted.

The collected source data were game-based, i.e. the records were ordered game by game. If such an order is kept, however, the training results may not be good, as the chessboard statuses in first several turns or last several turns could be very similar in different games, making the whole dataset too regular and not random enough. To increase the randomness, the records were shuffled first, to break the order, before being used to train the NN models.

## 6.2. Reinforcement Learning of Policy Network

### 6.2.1. Training Dataset

We use the game records played between our iteration of model to do reinforcement training. At the beginning stage of our training, we use our model 001 from the last year to do self-training because we don't have other choice. After then, we have several versions of models and we can select among them as the opponent.

During the self-playing progress, we will record if our model wins or loses, using the result to decide we should give them positive reward or negative reward.

As the training process needs lots of data, we upload our program to the GPU machines own by CSE department. As the specific of the GPU machine is much better than our laptops, the time of training reduce obviously. On our laptops, playing 1000 games needs nearly 24 hours. But on the GPU machine, we can do it in 5 hours which is 1/5 of before.

### 6.2.2. Training Strategy

After we get the dataset, we should give every move made by the network a label if it is a good move.

If the AI wins a victory, we label all the moves in this game made by it good moves. Oppositely, if it loses a game, we label all the moves in this game made by it bad moves. In fact, we want to find a move that greatly change the winning rate. However, it is impossible because the working load is too large to distinguish them manually.

During our training process, we tried different kinds of selection strategy. A completely determined selection strategy would not work well because it would perform unique and not suitable for training. But an over randomized strategy would always play bad moves

and the efficiency would drop. We found strategy 8 could be a solution but the possibility  $p$  must be carefully set. A casually set  $p$  would also decrease the training speed.

Although the performance of the network grew well, when we tried to add negative feedback to the network, the training met some trouble. We tried to set 1 for good move and -1 for bad moves at first. But the model became strange like making illegal move or reporting errors directly. We explained that the strategy would depress normal move by mistake and then the model would get confused. And we try to use 0 or some small values to replace -1 but these methods didn't work as well.

In the end, we had no choice but give up the negative reward. But when we encourage the good move in the game won, the other move would be depressed automatically. So, even if we don't have negative reward, the performance of model can keep increasing.

## 6.3. Supervised Learning of Evaluation Network

### 6.3.1. Training Dataset

We collected records of over 30000 Chinese chess games and about 2,000,000 moves in total, including games in professional competitions, classical ancient games, and online games between high-ELO players. This part is the same as the supervised learning. However, in the training process of Evaluation Network, we only care about the distribution of pieces but not how they move. So we extract the fen from the previous data and remove the duplicate fen. And for each fen in our dataset, we use an open-source evaluation function to label a reference value,

And as our input features includes the side to move, this time we do not need to flip the chessboard.

### 6.3.2. Training Strategy

To train our evaluation model, at first, we use convolutional neural network the same as our previous design. However, the training result is not so good that the winning rate do not increase. The possible reason is that the evaluation of a chess board will oscillate with little changes, while a player move a piece, in the view of the convolutional network, most of the features are the same as before and only a few features will change that influence the final result. The same will also happen on the input channels. As a result, the evaluations of the current chessboard are rather continuous but in fact they are discrete after each move.

We decide to change it to fully connected neural network to solve this problem. Different from the input from convolutional network, where we give the convolutional network a group of matrix representing different channels, this time we use the coordinates of pieces, the amounts of each kind of pieces, and the mobility of rocks and

cannons as the input feature. When a player move a piece and capture an enemy's piece, the change on the chessboard can be easily observed by the changes on the input vector.

Using this strategy, the trained model performed much better than the model using CNN. But this is only a quiescence search according to the current chessboard and has no ability to look forward. We still need to use minimax to improve the performance of our game AI.

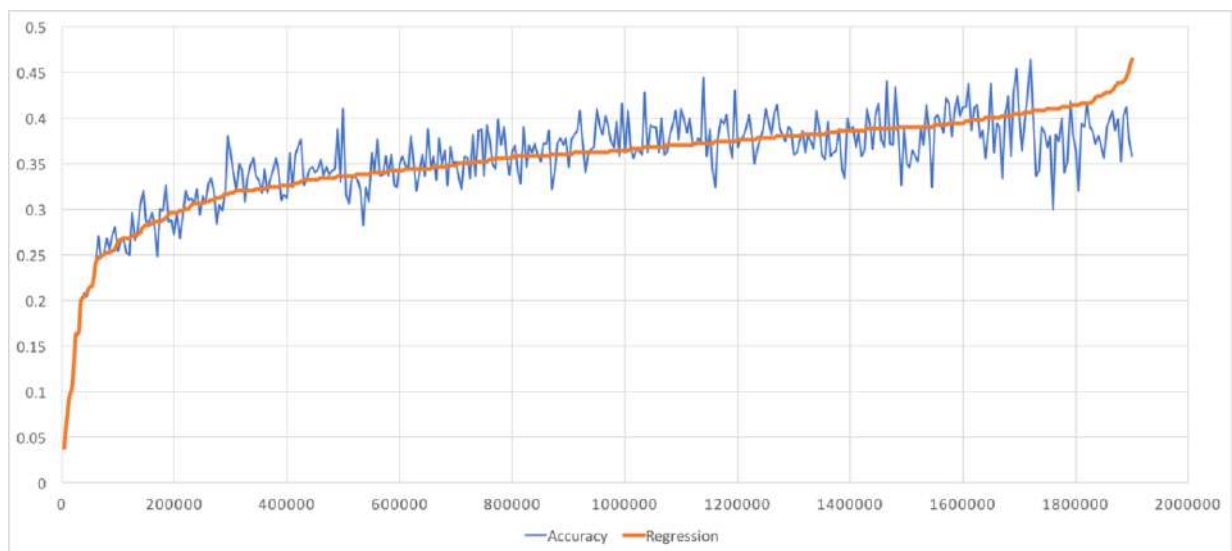
## 7. Results

### 7.1. Results of Supervised Learning

#### 7.1.1. Accuracy Testing

In Accuracy Testing, the AI model was simply tested based on a testing dataset in the same format with the training dataset, recording the moves made by professional expert players in realistic top-class competitions. This testing is to test the accuracy of our trained NN models predicting the choice of an expert player given a chessboard status. And this testing was done separately for Piece Selector and Move selector.

##### 7.1.1.1. Piece Selector



*Figure 7.1. Piece Selector Accuracy*

The accuracy of Piece Selector was recorded along the training process, as shown in Figure 7.1.. Evidently, the accuracy is generally increasing over the process, with reasonable oscillations. At last, for our testing dataset, Piece Selector has achieved an accuracy of 44.7%, which is quite high.



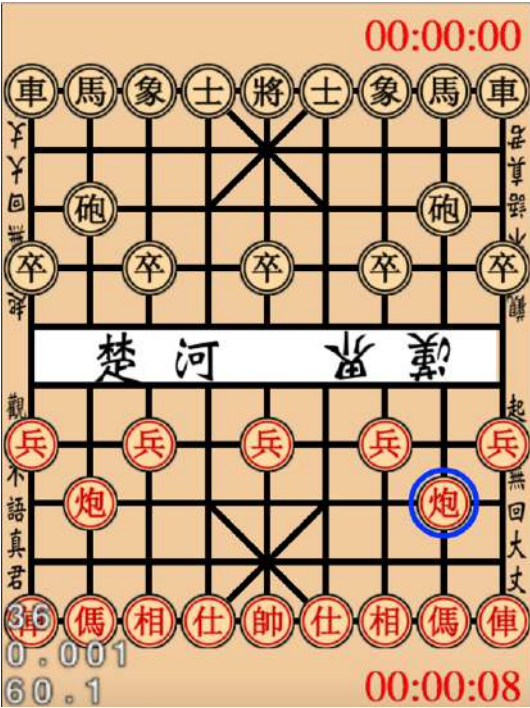


Figure 7.2. (a)

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%	0.0%	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%	22.1%	0.0%	0.0%	0.0%	2.7%	0.0%	0.0%
0.0%	2.8%	0.0%	0.0%	0.0%	0.0%	0.0%	57.8%	0.0%
0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
0.2%	2.8%	0.3%	0.0%	0.0%	0.0%	9.2%	1.7%	0.1%

Figure 7.2. (b)

For the initial chessboard status, the output from our Piece Selector when the AI plays the red side is as shown below. Figure 7.2. (a) shows the real chessboard, and Figure 7.2. (b) shown the corresponding output of Piece Selector with eliminating values less than 0.1% while Figure 7.2. (c) shows the source output from Piece Selector. The most suggested piece is the right red cannon in the blue cycle with 57.8% probability in the

red cycle, which is a popular opening way. The second highest suggested piece is the middle red pawn with 22.1% probability, which is also a good choice. Note that at positions with probability larger than 0.1% there always exists a red piece, indicating that our Piece Selector has learned to select pieces of its own side.

4.51E-07	5.74E-07	3.01E-06	9.88E-06	1.12E-06	1.04E-06	5.77E-07	9.86E-08	6.84E-07
1.41E-06	1.47E-06	7.70E-06	5.02E-06	1.63E-07	2.96E-06	8.18E-06	5.60E-06	5.68E-07
1.82E-06	2.27E-06	3.53E-06	7.90E-06	7.25E-06	1.12E-05	8.07E-06	2.06E-06	2.46E-06
9.92E-06	9.04E-06	6.32E-06	1.20E-06	1.78E-05	1.67E-05	2.64E-06	6.31E-06	3.12E-05
7.76E-06	2.26E-05	1.05E-05	2.83E-05	1.94E-06	4.13E-06	2.34E-05	1.29E-06	6.56E-06
7.64E-05	4.13E-05	1.50E-05	3.79E-06	6.28E-05	5.05E-06	5.76E-05	2.59E-06	8.95E-05
1.22E-04	2.06E-05	2.21E-01	3.15E-06	5.02E-05	1.01E-06	2.74E-02	4.28E-06	1.36E-04
1.39E-04	2.83E-02	4.53E-05	5.09E-06	2.47E-05	9.76E-06	5.09E-06	5.78E-01	3.85E-06
1.26E-04	2.06E-05	1.30E-06	4.88E-05	5.06E-06	6.93E-05	6.19E-06	7.04E-05	1.19E-04
2.44E-03	2.80E-02	2.86E-03	9.72E-05	2.06E-05	3.05E-04	9.18E-02	1.75E-02	9.00E-04

Figure 7.2. (c)

The testing dataset used here were collected independently from training dataset. The opening turns of different games, however, are quite similar because players tend to follow some fixed opening move sequences, which is considered to be optimal or at least good enough according to previous experiences, also called joseki in Go. In other word, there may hardly exist two same games, but they may very probably exist several same opening moves between games. Similarly, for middlegame moves and ending moves, there also exists such a phenomenon, more or less. This phenomenon would probably alter the testing accuracy because there may exist many duplicate testing examples, if comparing those records move by move but not game by game. If those duplicate moves were removed, the testing accuracy of Piece Selector was 40.2%, 4.5% less than before. However, we cannot certainly say which accuracy is correct.

On the one hand, it is consistent with reality because the frequency of every board may not be equal, not only in our collected records but also in realistic games. In fact, the

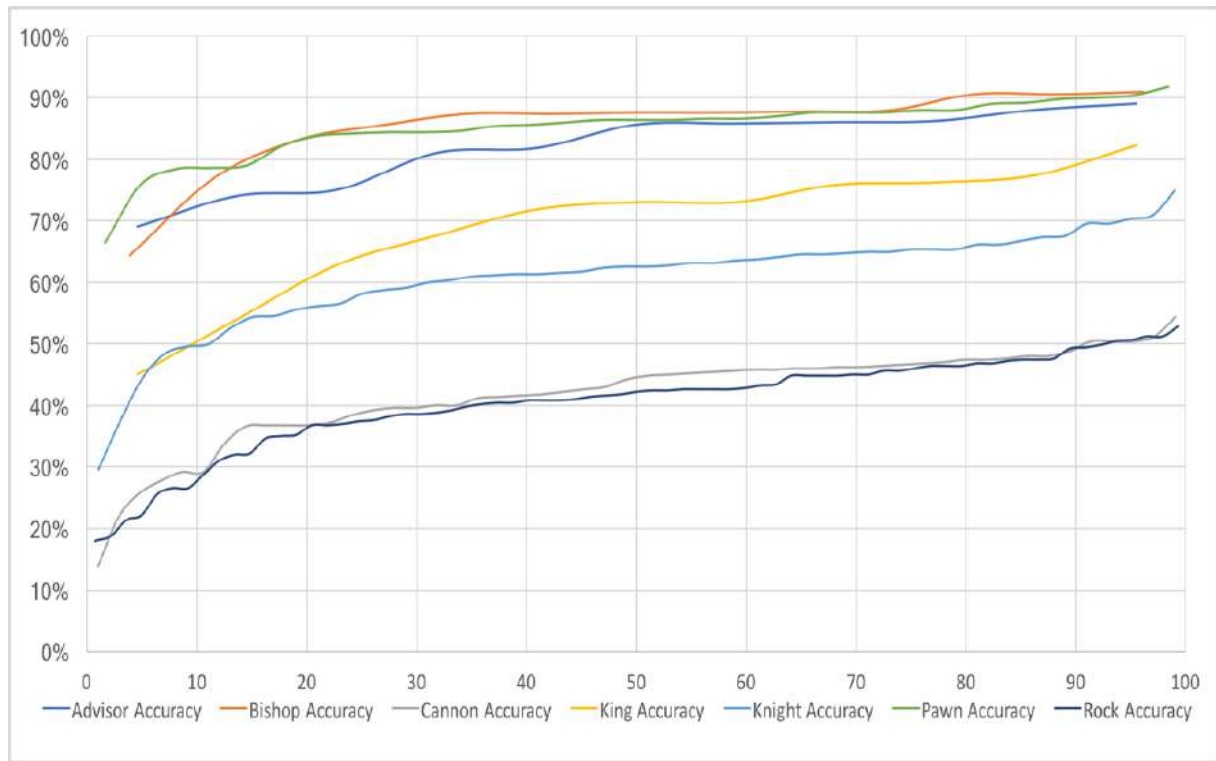
frequency of every situation in the dataset may reflect the realistic frequency of the situation. In this sense, the duplicate items do not need to be removed, as the accuracy can better measure the performance of Piece Selector in reality.

On the other hand, it is expected to have ability to deal with any situation, not only those very frequent situations but also the less frequent situations. In fact, the accuracy of predicting more frequent moves is higher than less seen moves because they are trained with more times, as they may also appear more frequently in our training dataset, indicating that the learning process would be better with larger dataset. Even worse, it can be treated as our model being overfitted into the training data.

Above all, we prefer to think that it's both OK whether to eliminate the duplicate records in testing dataset or not, but it's necessary to keep those duplicates in training dataset. More frequent moves in real games represent that more professional players think they are better moves, which is necessary in current phase.

Except for the issue discussed above, Piece Selector still needs to be further improved in other aspects. For example, in a case that a player is checked, the Piece Selector sometimes selects piece far away, which mean no matter how the selected piece moves, it can't save the king. The problem may attribute to lack of negative feedback. In the beginning phase of our training, we planned to set the moves of winner a positive weight and that of losers a negative way. But there are two reasons for us to abandon this idea. One is that many records are not complete, precisely not including the final result. The other reason is that it's hard to judge which move is the bad move. As most of our records were played by professional players, only one small mistake would lead to failure despite other moves were good. If we set them all negative, lots of good moves will be depressed.

### 7.1.1.2. Move Selector



*Figure 7.3. Move Selector Accuracy*

Similar with Piece Selector, the accuracy of Move Selector was recorded along the training process, as shown above. After training, Move Selector was also tested using the testing dataset and the results are as shown below. For Move Selector, the models of some types of pieces have achieved clearly better performance. For example, the Move Selectors of Advisor, Bishop and Pawn have achieved accuracies of near 90%, while the Move Selectors of Cannon and Rock have achieved accuracies of only around 50%. One possible reason may be that the possible moves of former pieces are relatively limited. Bishops have at most two legal moves in general, Advisors usually have only one possible move, and Pawns also only have few choices before they cross the river. So, Move Selectors of them are easier to train. But for Cannons and Rocks, the number of move choices is usually more than 10, and every move can be reasonable in some view, which means no absolute best move, and Move Selectors of them are more difficult to train.

Move Selector	Accuracy	Accuracy After Eliminating Duplicates
Advisor	89.8%	89.0%
Bishop	91.2%	89.8%
Cannon	54.1%	48.5%
King	79.8%	79.2%
Knight	70.1%	63.8%
Pawn	90.4%	88.5%
Rock	53.6%	48.1%

*Figure 7.4. Move Selector Accuracy for Different Piece Types*

The duplicates issue also exists for Move Selector. As shown in the table below, the accuracies of Move Selector models for different types all decreased, more or less, as expected. The discuss and conclusion is also similar, that we think it is fine, or even necessary, to keep those duplicates.

### 7.1.2. Real Performance Testing

As mentioned before, except for Accuracy Testing, the AI model was also tested in real games, playing against human players. In this section, several real game-playing samples are analyzed in detail and the performance of our AI is judged by some evident criteria, such as the responsiveness to being checked, the responsiveness when one piece is to be attacked and so on.

## 7.1.2.1. Game-Playing Case 1

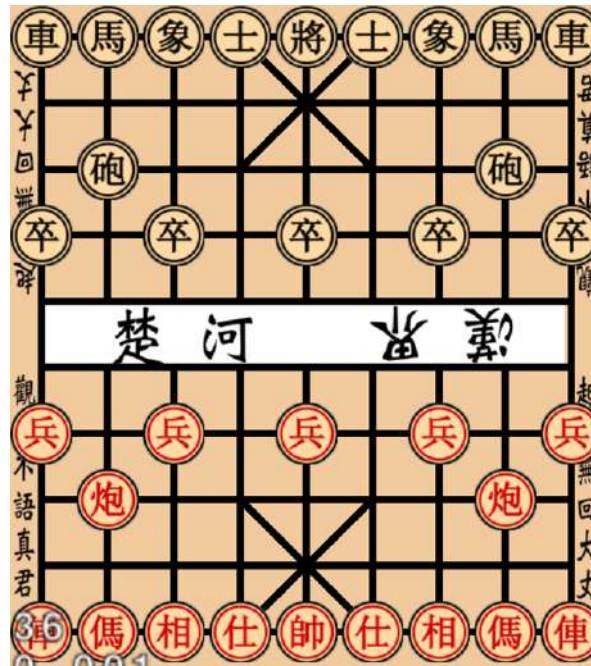


Figure 7.5 (a) Initial Status

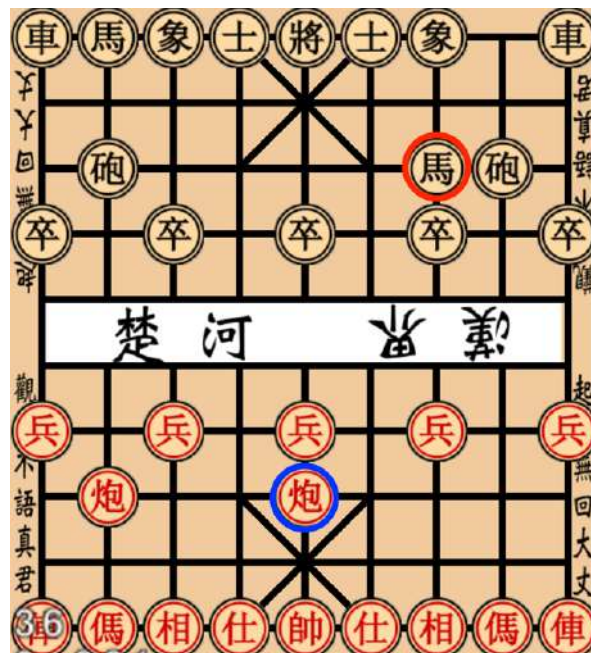


Figure 7.5 (b) Status after one move

This started from the initial game chessboard status, as shown in Figure 7.5. (a). AI played black side, and we played red side. In first turn, we moved the right cannon to the middle.

0.0%	15.2%	0.2%	0.0%	0.0%	0.0%	0.1%	75.1%	1.2%
0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
0.0%	0.5%	0.0%	0.0%	0.0%	0.0%	0.0%	6.6%	0.0%
0.0%	0.0%	0.1%	0.0%	0.0%	0.0%	0.8%	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%	0.0%	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 7.5 (c) Output of Piece Selector

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%	99.4%	0.0%	0.6%
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%	0.0%	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%	0.0%	0.0%

Figure 7.5 (d) Output of Move Selector.

This is one of the most popular opening moves, and after this step, the black pawn in the middle was under attack. The black side, i.e. our AI, chose to move the knight forward, as shown in Figure 7.5. (b), with 75.1% possibility given by Piece Selector and 99.4% possibility given by Move Selector, which is quite high, as shown in Figure 7.5. (c) &



(d). This is also one of the most popular opening moves, and after that, the middle black pawn was protected by this knight.

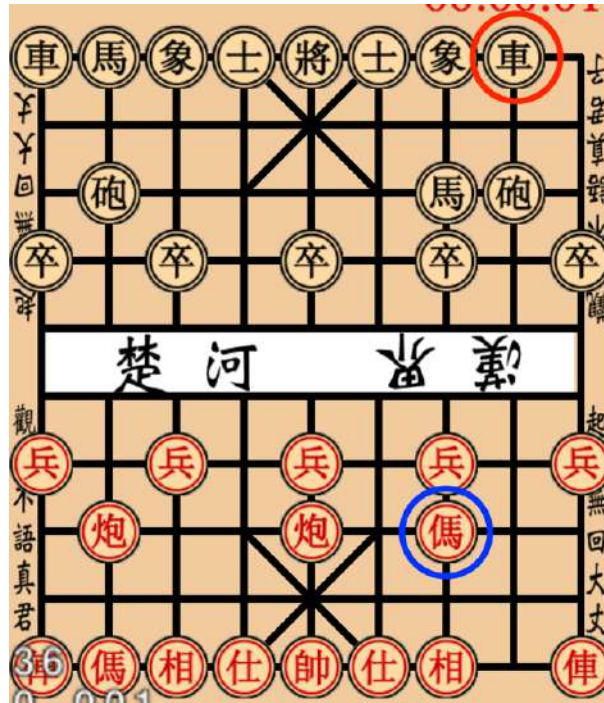


Figure 7.6. (a) Status after two moves

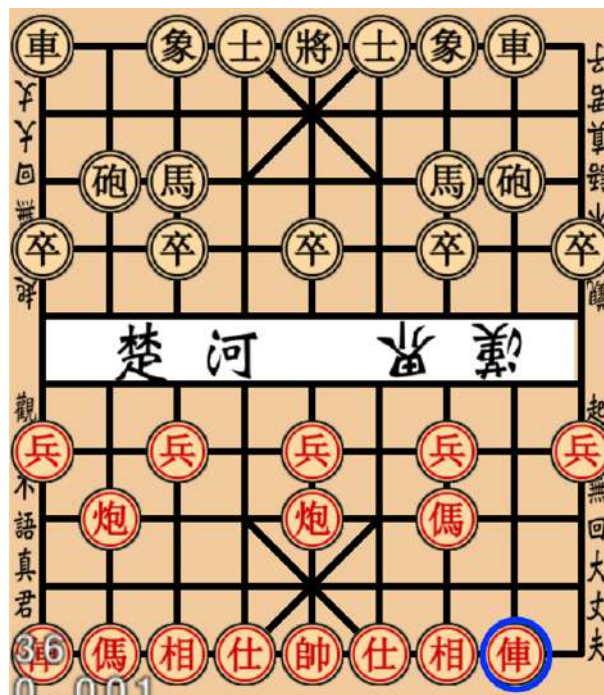


Figure 7.6. (b) Status after three moves

In second turn, we chose to move the right red knight forward and our AI chose to move



the right black rock left, as shown in Figure 7.6. (a), with 95.4% possibility given by Piece Selector and 99.8% possibility given by Move Selector, which is even higher, as shown in Figure 7.6. (c) & (d).

0.0%	1.4%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	95.4%
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%	0.3%	0.0%
0.0%	0.0%	0.2%	0.0%	0.0%	0.0%	2.4%	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%	0.0%	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 7.6. (c) Output of Piece Selector

0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	99.8%	0.0%
0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.2%
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%	0.0%	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%	0.0%	0.0%
0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%

Figure 7.6. (d) Output of Move Selector

Actually, it's a good move, as our next move was to move the right red rock out so that

the right black cannon would be under attack, as shown in Figure 7.6. (b). In this step, our AI predicted what the opponent would do and reacted effectively. As shown in these steps, our AI learned well in opening moves and reacted responsively. Generally speaking, the result is satisfying.

#### 7.1.2.2. Game-Playing Case 2

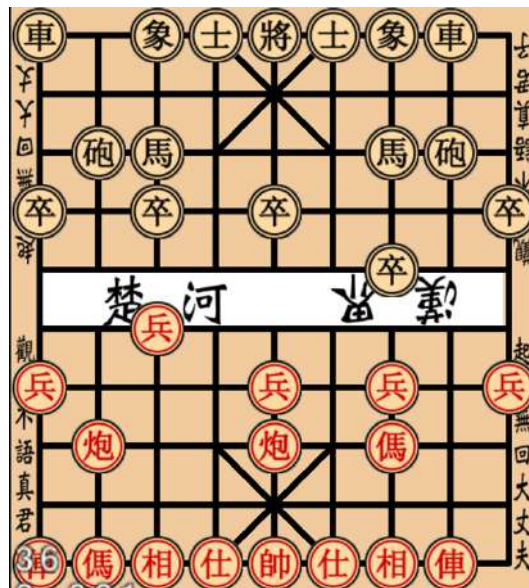


Figure 7.7. (a) Initial Status

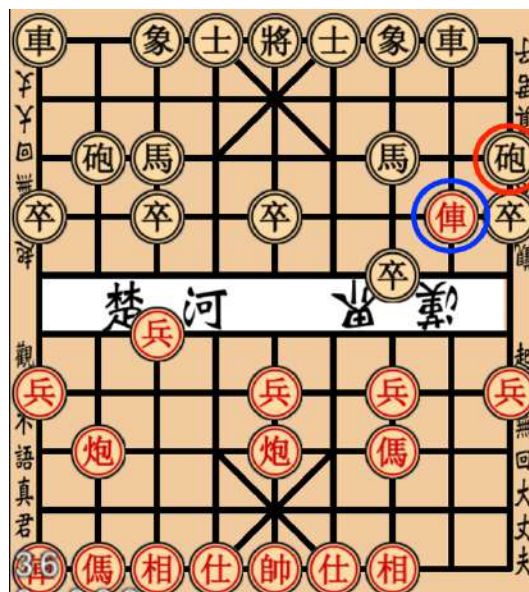


Figure 7.7. (b) Status after one move

1.5%	0.0%	4.3%	11.8%	0.0%	0.2%	1.1%	0.8%	0.0%
0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
0.0%	12.2%	0.2%	0.0%	0.0%	0.0%	11.2%	52.3%	0.2%
0.1%	0.0%	2.7%	0.0%	0.0%	0.0%	0.3%	0.0%	0.1%
0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.5%	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%	0.0%	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%

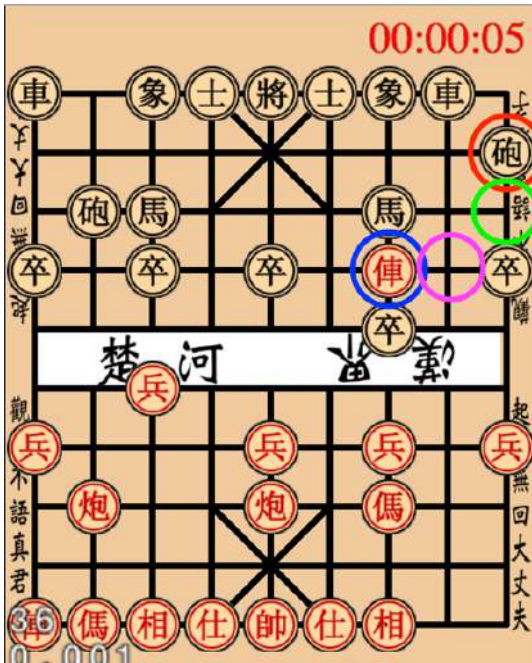
(c) Output of Piece Selector

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%	1.2%	0.0%
0.0%	0.0%	0.0%	0.0%	0.0%	0.1%	0.1%	0.1%	98.2%
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%
0.0%	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%	0.0%	0.0%	0.0%	0.0%	0.0%

(d) Output of Move Selector

This is an interesting turn where AI again plays the black side. The initial status is as shown in Figure 7.7. (a). We moved the right rock forward, and our AI chose to move the cannon to the right, which is a good move, as shown in Figure 7.7. (b). First, it left our rock to be attacked by the black rock. Secondly, it left the right pawn to be protected

by the black knight, as before this move, that knight can't protect that pawn because that will be an illegal move. Third, the middle black pawn is still protected by two knights. Actually, this move left us few choices to save our rock.



(a) Status after two moves

0.1%	0.0%	0.3%	1.3%	0.0%	0.0%	0.0%	1.7%	0.0%
0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
0.0%	0.3%	0.0%	0.0%	0.0%	0.0%	0.4%	0.0%	95.7%
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%	0.0%	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%	0.0%	0.0%

(b) Output of Piece Selector

0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.2%
0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	97.5%
0.0%	0.0%	0.0%	0.0%	0.0%	0.1%	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%	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%

(c) Output of Move Selector

Figure 7.8.

So, we chose to move rock one block left. And our AI chose to move the cannon

downward, as shown in Figure 7.8. (a), which turned out to be a good move but we did not realize that in the first place due to our limited skill in Chinese chess.

Two turns later, the AI chose to move the right black cannon from the position of green circle to the position of the red circle, as shown in Figure 7.9. (a), leaving the red rock under attack, with 84.6% possibility given by Piece Selector and 99.0% possibility given by Move Selector, as shown in Figure 7.9. (b) & (c).

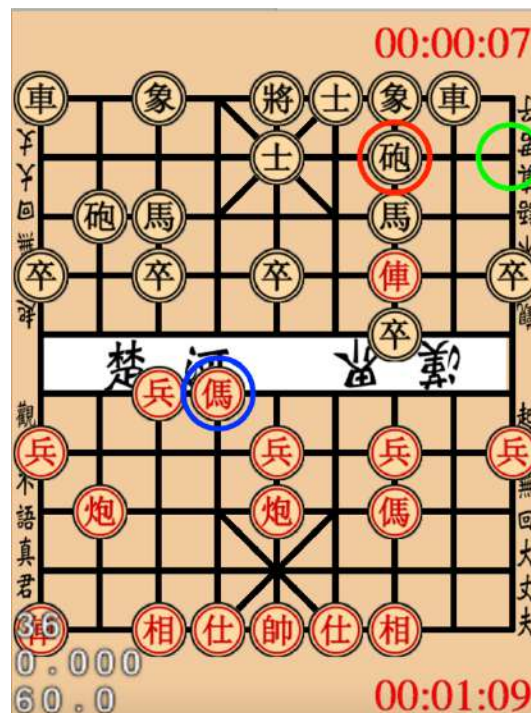


Figure 7.9. (a) Status after four moves



1.4%	0.0%	1.1%	0.3%	0.0%	0.0%	0.1%	9.4%	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%	84.6%	0.0%	0.0%	0.2%	99.0%	0.2%	0.1%	0.0%	0.0%	0.0%	0.0%
0.0%	1.5%	0.3%	0.1%	0.0%	0.0%	0.1%	0.1%	0.3%	0.0%	0.0%	0.0%	0.0%	0.0%	0.1%	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%	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%	0.0%	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%	0.0%	0.0%	0.0%	0.0%	0.2%	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%	0.0%	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%	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 7.9. (b) Output of Piece Selector

Figure 7.9. (c) Output of Move Selector

7.1.2.3. Game-Playing Case 3

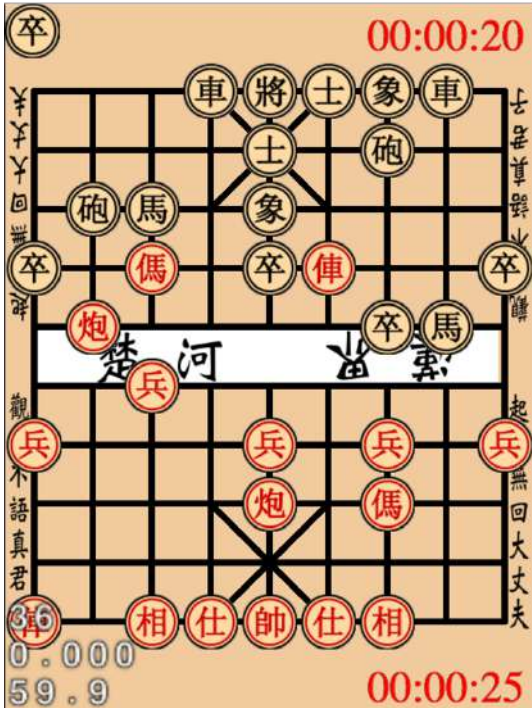


Figure 7.10. (a) Initial Status

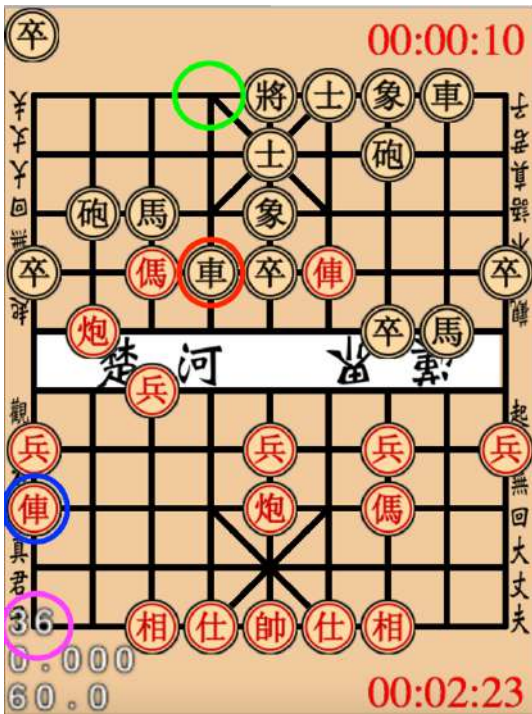


Figure 7.10. (b) Status after one move

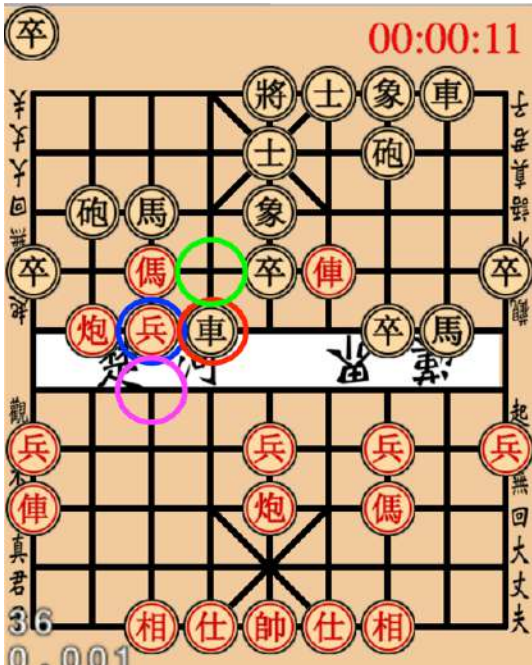
0.3%	0.0%	0.1%	34.9%	0.0%	0.5%	0.3%	6.6%	0.0%	4.7%	3.8%	2.2%	0.2%	0.0%	0.0%	0.0%	0.0%	0.0%
0.0%	0.0%	0.0%	0.0%	0.4%	0.0%	9.2%	0.0%	0.1%	0.0%	0.1%	0.1%	1.5%	0.0%	0.0%	0.0%	0.0%	0.0%
0.1%	5.2%	4.1%	0.1%	1.4%	0.1%	0.1%	0.0%	0.0%	0.0%	0.1%	0.1%	2.0%	0.0%	0.0%	0.0%	0.0%	0.0%
0.3%	0.0%	0.0%	0.0%	0.4%	0.0%	0.0%	0.2%	0.3%	0.0%	0.5%	0.0%	40.4%	0.0%	0.0%	0.0%	0.0%	0.0%
0.0%	0.0%	0.1%	0.1%	0.0%	0.1%	16.2%	17.6%	0.0%	0.0%	0.1%	0.1%	23.6%	0.0%	0.0%	0.0%	0.0%	0.0%
0.0%	0.0%	0.0%	0.0%	0.0%	0.1%	0.0%	0.1%	0.0%	0.0%	0.0%	0.1%	3.9%	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%	6.9%	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%	4.4%	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%	1.6%	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%	3.1%	0.0%	0.0%	0.0%	0.0%	0.0%

Figure 7.10. (c) Output of Piece Selector

Figure 7.10. (d) Output of Move Selector

And here is an example of bad performance of our AI. The initial status is as shown in Figure 7.10. (a). After we moved the red rock forward, the AI chose to move the black rock forward, from the green cycle to the red cycle, as shown in Figure 7.10. (b), so that it could attack the red knight in next turn and it also protected the black bishop in fifth

column from being attacked by that red knight because that move would be illegal by the moving rules of knight, which appeared to be good in the first place but turned out to be a bad move later.



(a) Status after two moves

0.3%	0.0%	0.1%	0.0%	0.1%	0.1%	0.1%	1.2%	0.0%	0.1%	0.1%	0.4%	1.9%	0.0%	0.0%	0.0%	0.0%	0.0%
0.0%	0.0%	0.0%	0.0%	1.5%	0.1%	5.8%	0.0%	0.0%	0.0%	0.1%	0.3%	6.5%	0.0%	0.0%	0.0%	0.0%	0.0%
0.2%	2.4%	6.1%	0.1%	10.0%	0.0%	0.1%	0.1%	0.0%	0.2%	0.1%	0.4%	4.7%	0.0%	0.1%	0.0%	0.0%	0.0%
0.1%	0.1%	0.1%	27.9%	1.9%	0.0%	0.0%	0.1%	0.1%	0.2%	1.4%	23.8%	2.3%	0.1%	0.3%	0.0%	0.1%	0.0%
0.0%	0.0%	0.0%	0.2%	0.1%	0.2%	16.5%	23.8%	0.0%	0.0%	0.5%	1.2%	27.7%	0.0%	0.0%	0.0%	0.0%	0.0%
0.0%	0.1%	0.0%	0.0%	0.0%	0.1%	0.0%	0.0%	0.0%	0.0%	0.0%	0.2%	13.5%	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%	0.3%	7.8%	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%	0.2%	2.2%	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%	1.5%	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%	1.3%	0.0%	0.0%	0.0%	0.0%	0.0%

(b) Output of Piece Selector

(c) Output of Move Selector

Figure 7.11.

In next turn, however, we chose to move the red pawn forward, from the purple cycle to



the blue cycle. And unbelievably, our AI chose to move the rock forward again, leaving it under attack by the left red cannon, as shown in Figure 7.11. (a). As shown in Figure 7.11. (b) & (c), especially the Piece Selector prediction results, this choice was not a clearly good one. Piece selector gave only 27.9% possibility to choose this rock piece, while it also gave 23.8% and 16.5% possibility to move the knight and pawn respectively. Similarly, Move Selector gave only 27.7% possibility for the rock piece to move to the position of red cycle, while it also gave 23.8% and 13.5% possibility for other two choices respectively, which could be a little bit better.

Seen from the example above, we can find that always selecting the piece with highest possibility given by Piece Selector and then selecting a move for it doesn't work well in some situations, especially when the possibilities of several pieces, given by Piece Selector, are quite close, which also means that none of them is much better than others.

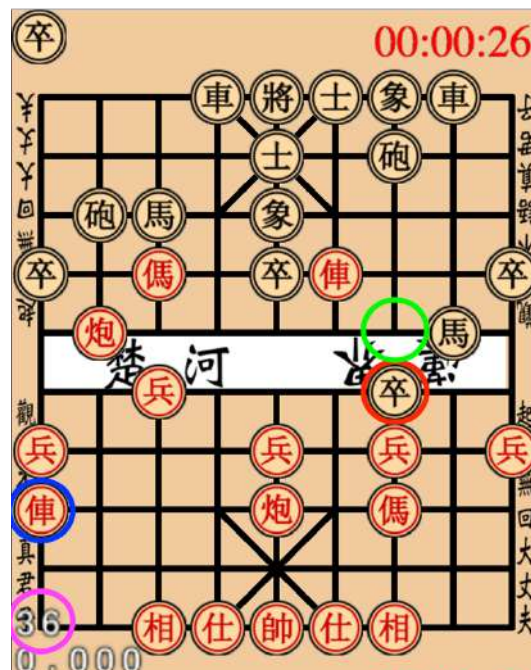


Figure 7.12. (a) New Status after two moves

0.3%	0.0%	0.1%	34.9%	0.0%	0.5%	0.3%	6.6%	0.0%
0.0%	0.0%	0.0%	0.0%	0.4%	0.0%	9.2%	0.0%	0.1%
0.1%	5.2%	4.1%	0.1%	1.4%	0.1%	0.1%	0.0%	0.0%
0.3%	0.0%	0.0%	0.0%	0.4%	0.0%	0.0%	0.2%	0.3%
0.0%	0.0%	0.1%	0.1%	0.0%	0.1%	16.2%	17.6%	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%	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%

(b) Output of Piece Selector

4.7%	3.8%	2.2%	0.2%	0.0%	0.0%	0.0%	0.0%	0.0%
0.0%	0.1%	0.1%	1.5%	0.0%	0.0%	0.0%	0.0%	0.0%
0.0%	0.1%	0.1%	2.0%	0.0%	0.0%	0.0%	0.0%	0.0%
0.0%	0.5%	0.0%	40.4%	0.0%	0.0%	0.0%	0.0%	0.0%
0.0%	0.1%	0.1%	23.6%	0.0%	0.0%	0.0%	0.0%	0.0%
0.0%	0.0%	0.1%	3.9%	0.0%	0.0%	0.0%	0.0%	0.0%
0.0%	0.0%	0.0%	6.9%	0.0%	0.0%	0.0%	0.0%	0.0%
0.0%	0.0%	0.0%	4.4%	0.0%	0.0%	0.0%	0.0%	0.0%
0.0%	0.0%	0.1%	1.6%	0.0%	0.0%	0.0%	0.0%	0.0%
0.0%	0.0%	0.0%	3.1%	0.0%	0.0%	0.0%	0.0%	0.0%

(c) Output of Move Selector for The Rock

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%	13.8%	0.0%	0.2%
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%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	85.3%	0.0%	0.7%
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%

(d) Output of Move Selector for the Knight

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%	0.0%	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%	99.9%	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%	0.0%	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%

(e) Output of Move Selector for the Pawn

Figure 7.12.

Then, we modified the selection strategy of our AI as: select the best three choices, i.e. pieces with the highest three possibilities, then generate the move possibilities for each of them by our Move Selector, multiply the piece possibilities and move possibilities respectively, and at last pick the move with highest possibilities. As a result, the AI would choose to move the black pawn this time, from the green cycle to the red cycle,

as shown in Figure 7.12. (a). The output of Piece Selector is as shown in Figure 7.12. (b). And the outputs from Move Selector for three different pieces, which are highlighted in Figure 7.12. (b), are as shown in Figure 7.12. (c) & (d) & (e) respectively. This is a good, or much better move. First of all, the left black rock wouldn't be under attack. Secondly, no matter whether this black pawn captured the red pawn in front of it or that red pawn captured it, the right red knight would be under attack by the right black cannon, or even better, by that black pawn as well.

#### 7.1.2.4. Game-Playing Case 4

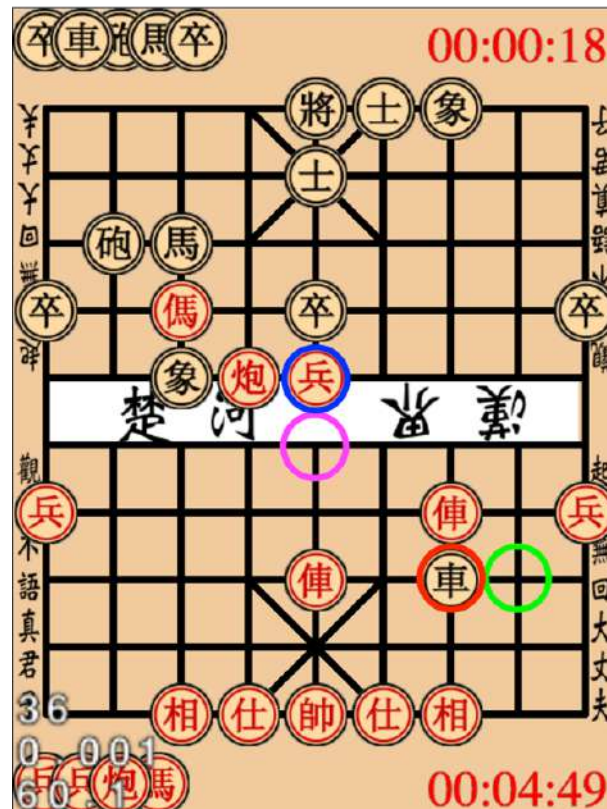


Figure 7.13. (a) Chessboard Status

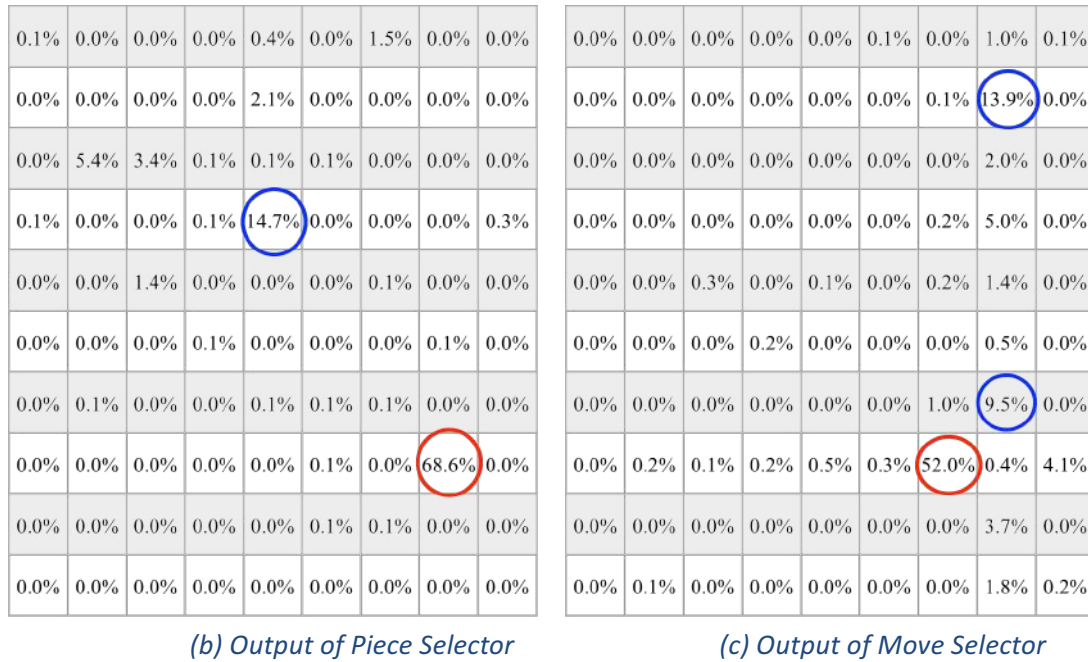
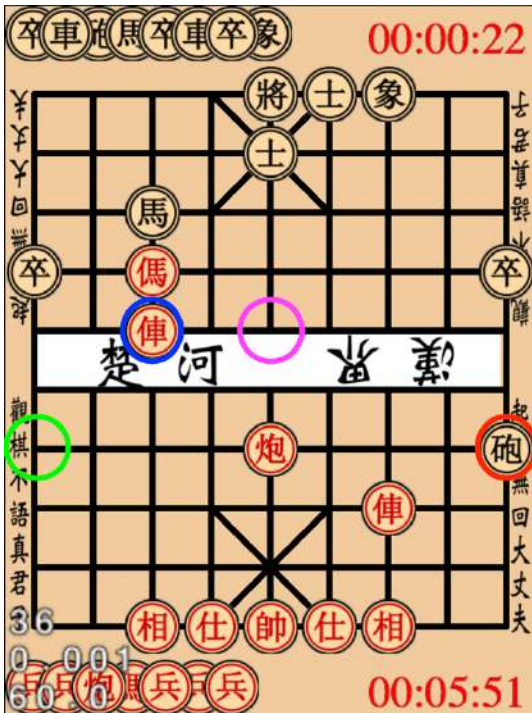


Figure 7.13.

In this turn, we chose to move the red pawn forward, from purple cycle to blue cycle. Then, the AI chose to move the black rock left, from green cycle to red cycle, as shown in Figure 7.13. (a), which was a definitely bad move. First of all, that black rock was under attack by the left red rock. We did not choose to capture it and wanted to see how the AI would react. The expected move is that the AI would choose to move that black rock to capture the red rock and could check the red side as well. Or at least, the AI would move the black rock away to avoid being attacked by red rock. However, the AI moved the black rock to the red cycle, leaving it under attack by both red rocks, not checking the red side and even did not save the middle black pawn which was under attack by the middle red pawn.

In a word, this is an example where the AI performed quite bad.

7.1.2.5. Game-Playing Case 5



(a) Chessboard Status

0.2%	0.1%	0.2%	0.0%	5.8%	0.0%	12.9%	0.0%	0.0%
0.0%	0.0%	0.0%	0.0%	2.4%	0.0%	0.0%	0.0%	0.0%
0.4%	0.0%	16.5%	0.1%	0.5%	0.0%	0.1%	0.1%	0.0%
1.7%	0.1%	0.0%	0.0%	0.1%	0.0%	0.0%	0.0%	0.2%
0.1%	0.1%	0.1%	0.1%	0.1%	0.0%	0.3%	0.0%	0.0%
0.2%	0.0%	0.0%	0.1%	0.0%	0.1%	0.0%	0.0%	0.0%
55.6%	0.1%	0.1%	0.1%	0.0%	0.0%	0.1%	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%

(b) Output of Piece Selector

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%	0.0%	0.1%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
1.6%	0.0%	0.2%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
7.4%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
0.2%	3.2%	10.0%	1.9%	0.2%	0.0%	0.0%	0.0%	70.9%
0.2%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
1.3%	0.0%	0.1%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
2.0%	0.0%	0.1%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%

(c) Output of Move Selector

Figure 7.14.

After several turns, pieces left on the chessboard became much less. And after we chose to move the red rock from purple cycle to blue cycle, the black side was under check by

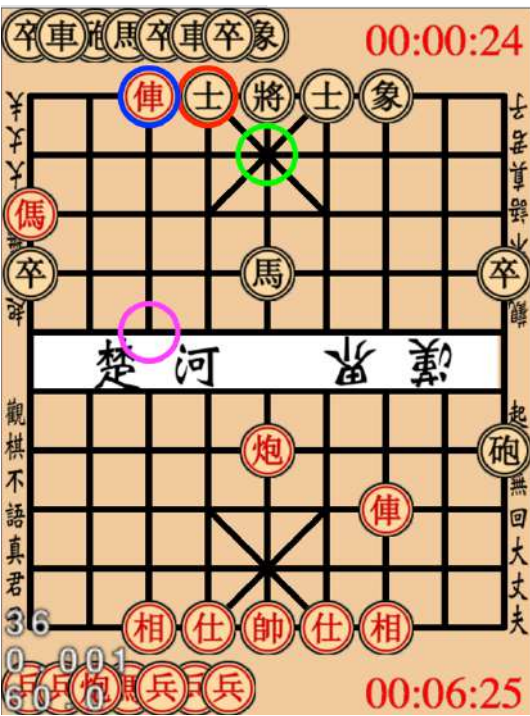
the red cannon. However, the AI chose to move the black cannon from green cycle to red cycle to capture a red pawn, as shown in Figure 7.14. (a), with 55.6% possibility given by Piece Selector and 70.9% possibility given by Move Selector, as shown in Figure 7.14. (b) & (c).

Obviously, this was a terrible move. After all, in next turn, we could use the red cannon to capture the black king and the AI would lose the game. This shows that our AI are not very responsive to the situation of being checked, which is a vital problem.

To further test its responsiveness to being checked, we did not capture the black king directly, but moved the red rock forward, from purple cycle to blue cycle, to check the black side again, as shown in Figure 7.15. (a). This time, the AI appeared to a little smarter and chose to move the advisor down to protect its king, with 82.2% possibility given by Piece Selector and 84.4% possibility given by Move Selector, as shown in Figure 7.15. (b) & (c), which was quite high, indicating that the AI was quite sure about this move.

Even though it was still being checked by the red cannon, it performed better in this situation. And this actually leads us to think why the AI responded effectively to being checked by rock but responded terribly to being checked by cannon. And more testing moves were made.





(a) Chessboard Status

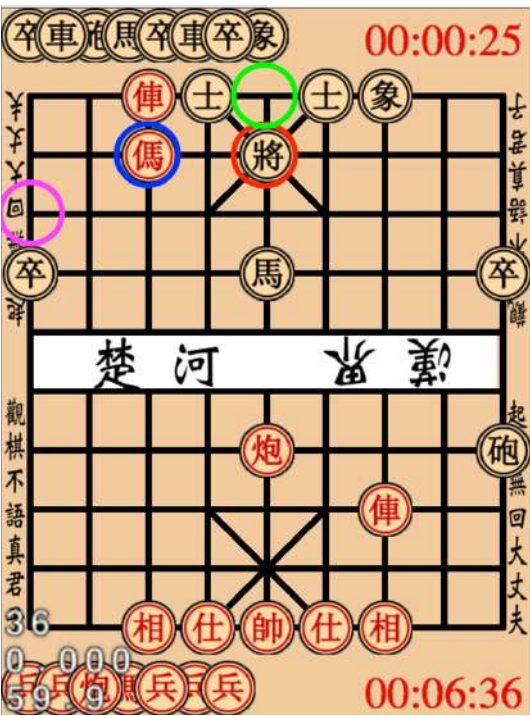
0.0%	0.0%	0.0%	0.0%	5.8%	0.0%	2.0%	0.0%	0.0%
0.0%	0.0%	0.0%	0.0%	82.2%	0.0%	0.0%	0.0%	0.0%
0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
0.0%	0.0%	0.0%	0.0%	4.3%	0.0%	0.0%	0.0%	0.0%
0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	5.3%
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%

(b) Output of Piece Selector

0.0%	0.0%	0.0%	84.4%	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%	11.5%	0.0%	4.1%	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%
0.0%	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%	0.0%	0.0%	0.0%	0.0%	0.0%

(c) Output of Move Selector

Figure 7.15.



(a) Chessboard Status

0.0%	0.0%	0.0%	0.4%	91.1%	0.3%	1.9%	0.0%	0.0%
0.0%	0.0%	0.0%	0.0%	0.2%	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%	1.3%	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%	4.3%
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%

(b) Output of Piece Selector

0.0%	0.0%	0.0%	0.1%	0.1%	0.8%	0.0%	0.0%	0.0%
0.0%	0.0%	0.0%	0.1%	98.5%	0.1%	0.0%	0.0%	0.0%
0.0%	0.0%	0.0%	0.3%	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%
0.0%	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%	0.0%	0.0%	0.0%	0.0%	0.0%

(c) Output of Move Selector

Figure 7.16.

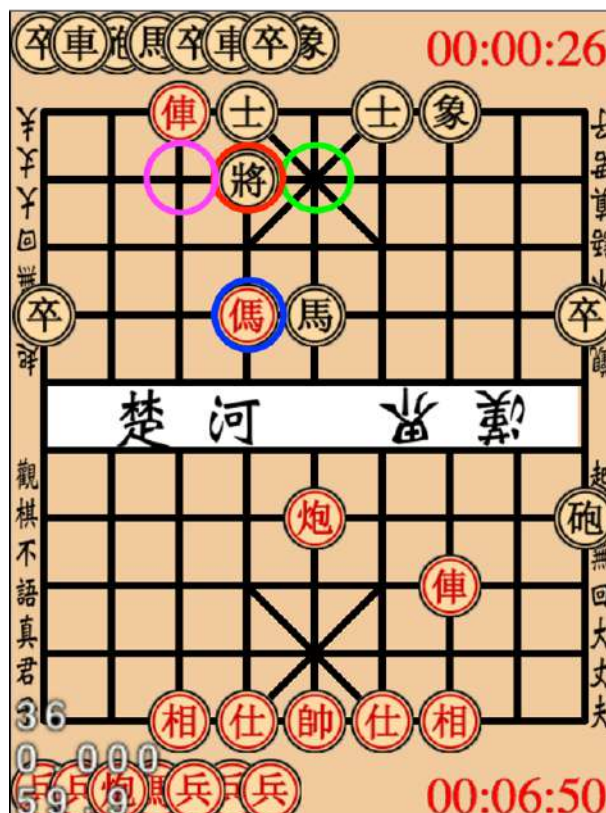
In this turn, we chose to use the red knight to check the black side, moving it from purple cycle to blue cycle. Surprisingly, the AI chose to move the black king forward to avoid being attacked by the red knight, as shown in Figure 7.16. (a), with 91.1%



possibility given by Piece Selector and 98.5% possibility given by Move Selector, as shown in Figure 7.16. (b) & (c), which indicated that the AI was almost 100% sure about this move. So, in this move, the AI also performed quite good.

However, the AI still did not respond to being checked by the red cannon. After all, it could choose to move the black knight downward, to protect the king from being attacked by both the red knight and the red cannon.

Again, we continued to use the red knight to check the black side, and the AI also responded well and moved the black king left, with 75.7% possibility given by Piece Selector and 45.4% possibility given by Move Selector, as shown in Figure 7.17.. Eventually, it escaped from being checked by the red cannon. But obviously, it was not due to that the AI realized it was checked by the red cannon. It was just a coincidence.



(a) Chessboard Status

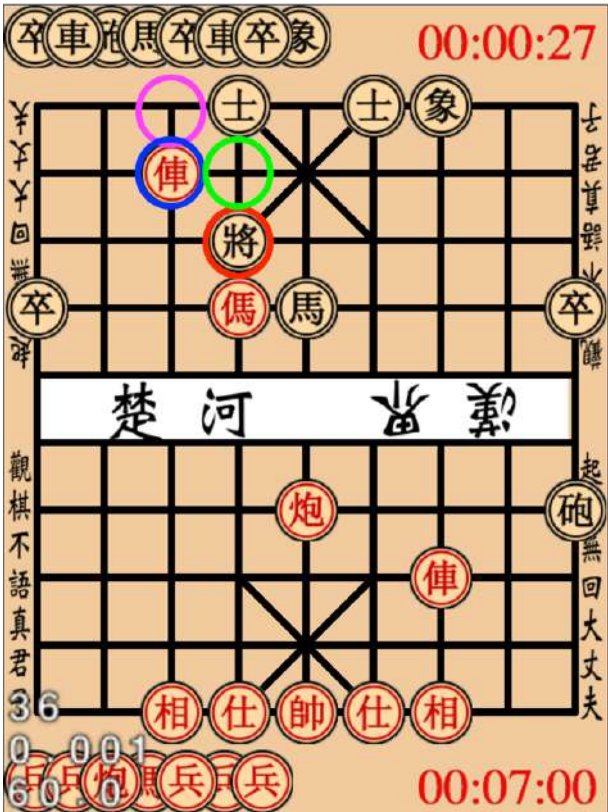
0.0%	0.0%	0.0%	0.1%	0.1%	0.1%	4.7%	0.0%	0.0%
0.0%	0.0%	0.0%	0.0%	75.7%	0.0%	0.0%	0.0%	0.0%
0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
0.3%	0.0%	0.0%	0.0%	5.7%	0.0%	0.0%	0.0%	0.2%
0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	12.8%
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%

(b) Output of Piece Selector

0.0%	0.0%	0.0%	0.0%	13.6%	0.0%	0.0%	0.0%	0.0%
0.0%	0.0%	0.0%	45.4%	0.0%	31.1%	0.0%	0.0%	0.0%
0.0%	0.0%	0.0%	0.0%	9.8%	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%	0.0%
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%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%

(c) Output of Move Selector

Figure 7.17.



(a) Chessboard Status

0.0%	0.1%	0.0%	0.8%	0.0%	0.3%	0.7%	0.0%	0.0%
0.0%	0.0%	0.0%	86.1%	0.4%	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%
0.2%	0.0%	0.0%	0.0%	3.2%	0.0%	0.0%	0.0%	0.2%
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%	7.6%
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%

(b) Output of Piece Selector

0.0%	0.0%	0.0%	1.1%	0.3%	0.2%	0.0%	0.0%	0.0%
0.0%	0.0%	0.0%	0.3%	23.8%	0.4%	0.0%	0.0%	0.0%
0.0%	0.0%	0.0%	73.3%	0.2%	0.5%	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%
0.0%	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%	0.0%	0.0%	0.0%	0.0%	0.0%

(c) Output of Move Selector

Figure 7.18.

After that, we moved the red rock backward, from the purple cycle to the blue cycle, and checked the black side again. This time, the AI chose to move the black king forward again, as shown in Figure 7.18. (a), to escape from being attacked by the red rock and avoid from being attacked by the red knight at the same time, with 86.1% possibility given by Piece Selector and 73.3% possibility given by Move Selector, as shown in Figure 7.18. (b) & (c). Up to this point, the AI had responded well to being checked by the red rock and the red knight, twice for each. So, we came up with a hypothesis that the AI could respond well if it is checked in a shorter distance, but cannot perform reasonably if it is checked in a longer distance.

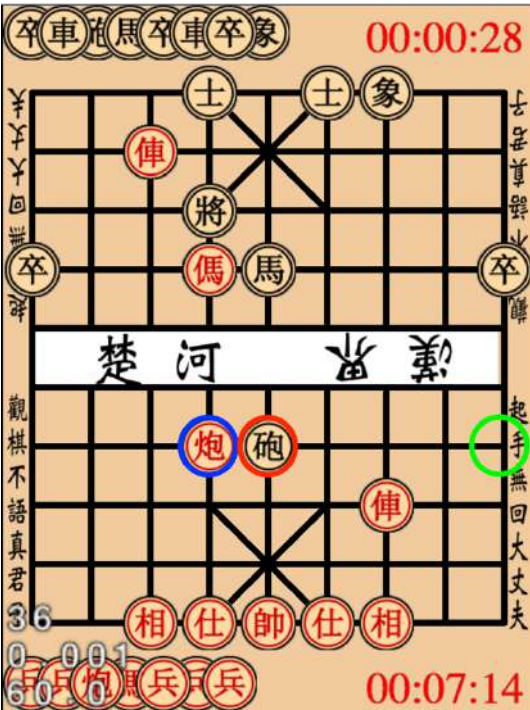


Figure 7.19. (a) Chessboard Status

0.0%	0.0%	0.2%	10.1%	0.0%	2.0%	8.6%	0.0%	0.0%
0.0%	0.0%	0.0%	0.3%	0.2%	0.1%	0.0%	0.0%	0.0%
0.0%	0.0%	0.0%	15.7%	0.1%	0.2%	0.0%	0.0%	0.0%
0.7%	0.0%	0.0%	0.0%	26.0%	0.0%	0.0%	0.0%	1.7%
0.0%	0.0%	0.1%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
0.0%	0.0%	0.1%	0.0%	0.0%	0.0%	0.1%	0.0%	0.0%
0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.1%	0.0%	32.7%
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%

(b) Output of Piece Selector

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%	0.1%
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%
0.1%	0.0%	0.0%	0.1%	0.0%	0.0%	0.0%	0.0%	23.5%
0.0%	0.0%	0.0%	0.0%	0.1%	0.1%	0.0%	0.0%	6.8%
0.0%	0.0%	0.0%	0.1%	28.4%	14.9%	7.4%	9.9%	0.4%
0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.5%
0.0%	0.0%	0.0%	0.1%	0.0%	0.0%	0.1%	0.0%	1.8%
0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	4.6%

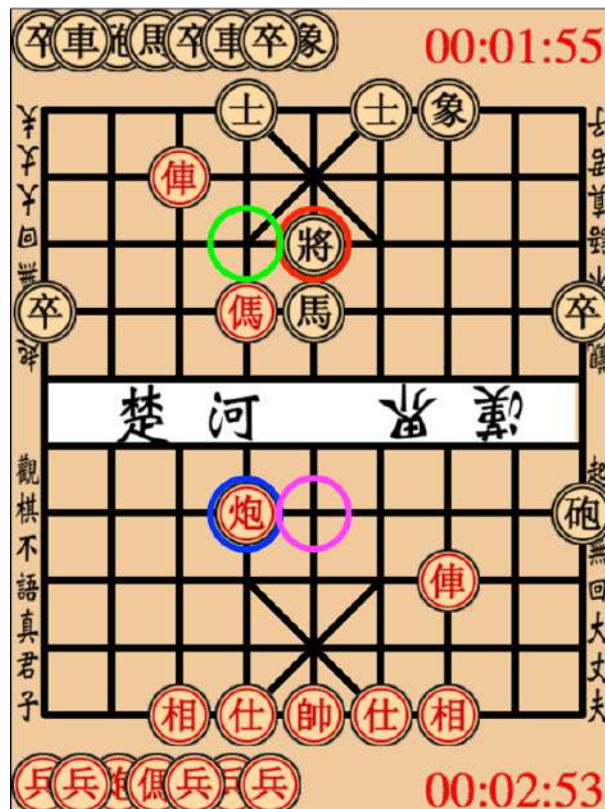
(c) Output of Move Selector for the Cannon

Figure 7.19.

To prove our own hypothesis, we chose to move the red cannon to check the black side again, from purple cycle to the blue cycle. As expected, the AI did not perform well and did realize that it was being checked. It chose to move the black cannon from the green cycle to the red cycle, as shown in Figure 7.19. (a). But we noted that the possibility of

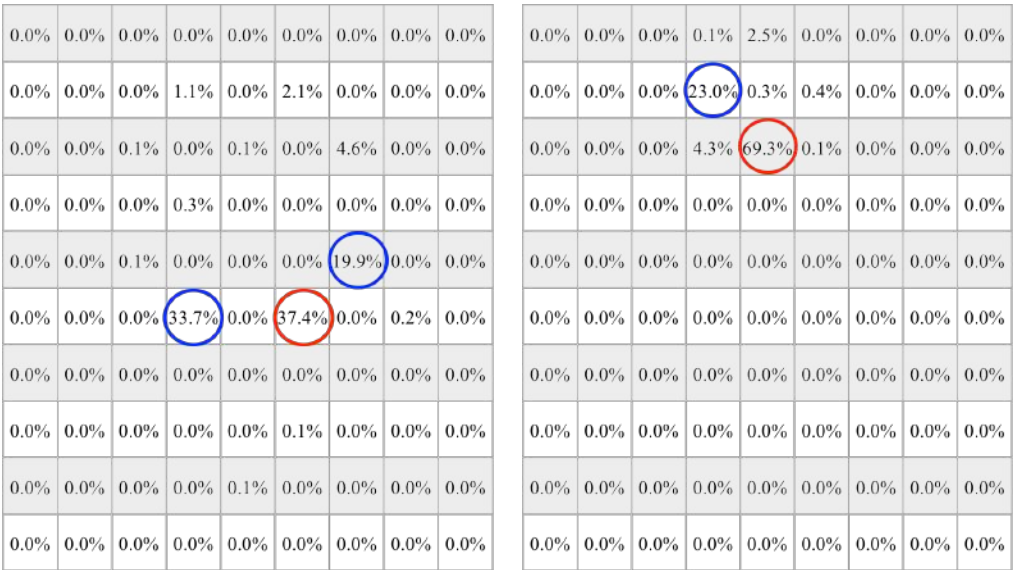
this move was not clearly better than other choices, as Piece Selector only gave it 32.7% possibility but also gave other pieces 26.0% and 15.7% possibilities respectively, as shown in Figure 7.19. (b).

Therefore, we decided to apply the other selection strategy again, which would consider the possibilities given by Piece Selector and Move Selector together. By this selection strategy, the AI chose to move the black king right, from the green cycle to the red cycle, as shown in Figure 7.20. (a), so that it successfully escaped from being checked by the red cannon and avoid from being attacked by the red rock and red knight at the same time. Using this selection strategy, the AI performed much better in this situation and even responded well to being checked by the cannon in longer distance.



(a) Chessboard Status

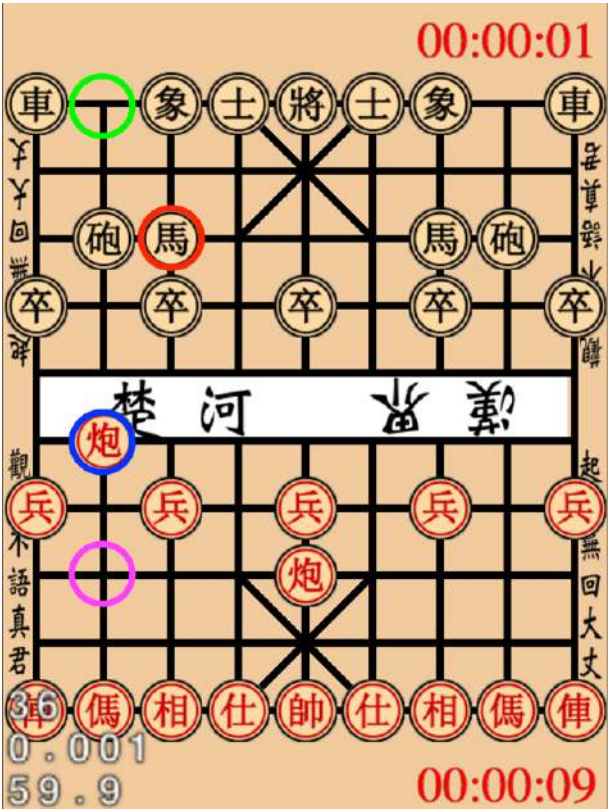




(b) Output of Move Selector for the Knight (c) Output of Move Selector for the King

Figure 7.20.

7.1.2.6. Game-Playing Case 6



(a) Chessboard Status

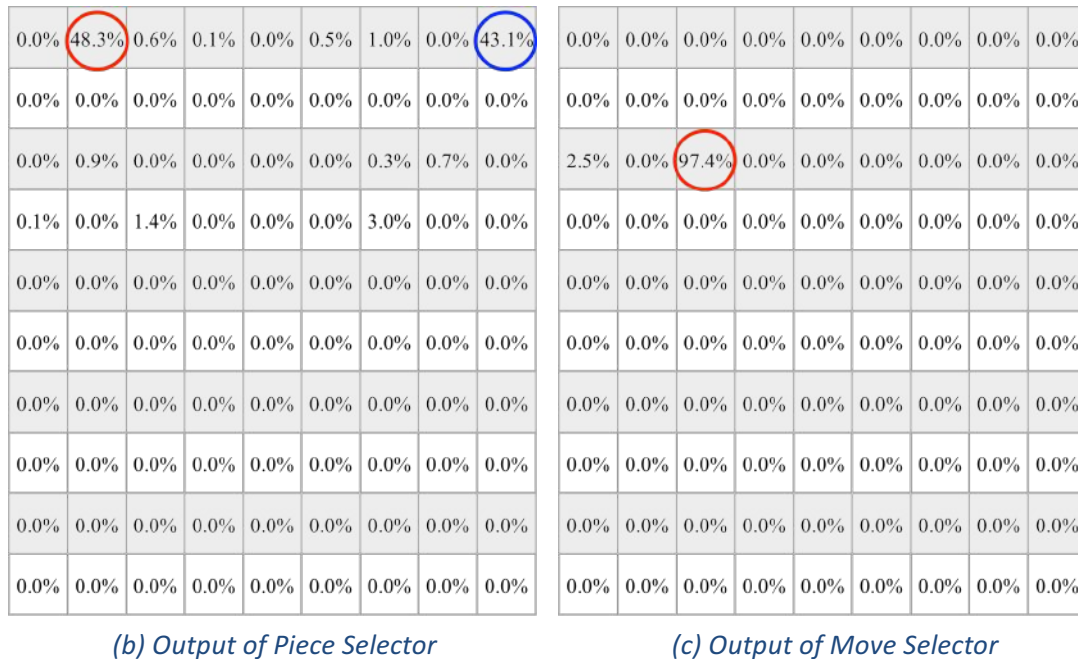


Figure 7.21.

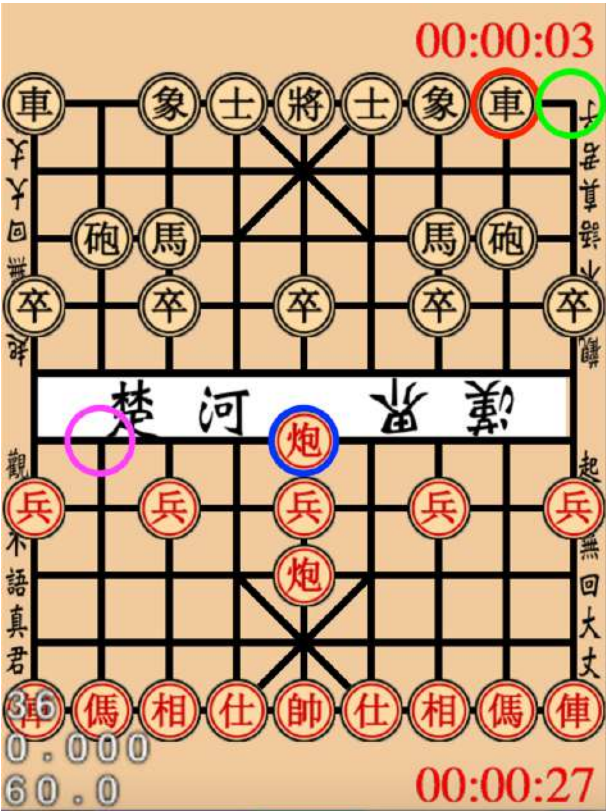
This is another example where AI performed bad. We noted that in the game records we collected for model training and testing, the most common opening moves were roughly always to move one cannon to the middle, move one knight forward and then move one rock out. So, we used another very common opening way which was seldom used in professional competitions since it was not that effective actually.

Here, we moved one red cannon to the middle and then moved another cannon forward, from the purple cycle to the blue cycle. And the AI seemed still to follow the fixed opening way, move the black knight first, with 48.3% possibility given by Piece Selector and 97.4% possibility given by Move Selector, as shown in Figure 7.21.. It is still fine up to this point.

In next turn, we moved the another red cannon to the middle as well, from purple cycle to the blue cycle, also known as “双炮将”. However, the AI chose to move the black rock out, from the green cycle to the red cycle, with 67.5% possibility given by Piece Selector and 99.6% possibility given by Move Selector, as shown in Figure 7.22..

Indeed, the AI still stuck to the most common fixed opening moves, but did not respond

well to being checked by the cannon again. This time, even after we used the second selection strategy, the AI still made the same choice.



(a) Chessboard Status

11.7%	0.0%	1.0%	0.8%	0.1%	0.5%	1.4%	0.0%	67.5%
0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
0.0%	1.1%	0.2%	0.0%	0.0%	0.0%	1.4%	1.1%	0.0%
0.1%	0.0%	7.2%	0.0%	0.0%	0.0%	5.2%	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%	0.0%	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%

(b) Output of Piece Selector

0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	99.6%	0.0%
0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.2%
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%	0.0%	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%	0.0%	0.0%

(c) Output of Move Selector

Figure 7.22.

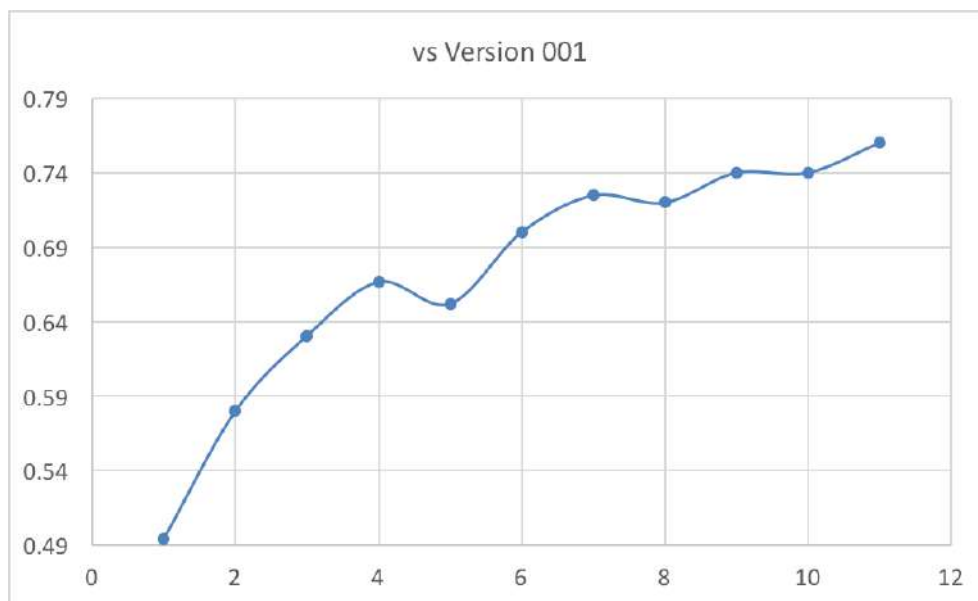


One important reason is that this situation has never appeared in our training dataset as it has seldom happened in realistic professional Chinese chess competitions. So, in this aspect, the result is acceptable but still not satisfying.

Another issue is that the AI could not respond well to being checked by cannon, or more generally, being checked by pieces in long distance. This may be due to that the training dataset is not larger enough, or more likely, due to that the CNN in Piece Selector and Move Selector is not deep enough.

## 7.2. Results of Reinforcement Learning

To test or demonstrate the results of Reinforcement Learning, letting two models from different training stages compete with each other for many games and calculate the winning rate is a reasonable and convincing way. If the winning rate of one version is far larger than 0.5, then we can safely conclude that that version is stronger than the other one.



*Figure 7.23 Winning Rate against Version 001*

During Reinforcement Learning, after several rounds training, we recorded that model version and let it compete with the original version so that we can judge whether the training worked by the winning rate. As shown in Figure 7.23., it showed the winning rate of different versions versus Version 001. Generally, the winning rate is increasing, indicating that Reinforcement Learning worked and the AI model has improved.

Except for competing with Version 001, we also let the newest version compete with the last version and recorded the winning rate. As shown in Figure 7.24., all the winning rate is larger than 50%, which means at least the model is making progress and better than previous one, though the winning rate decreased at some point which means the progress is not that obvious.

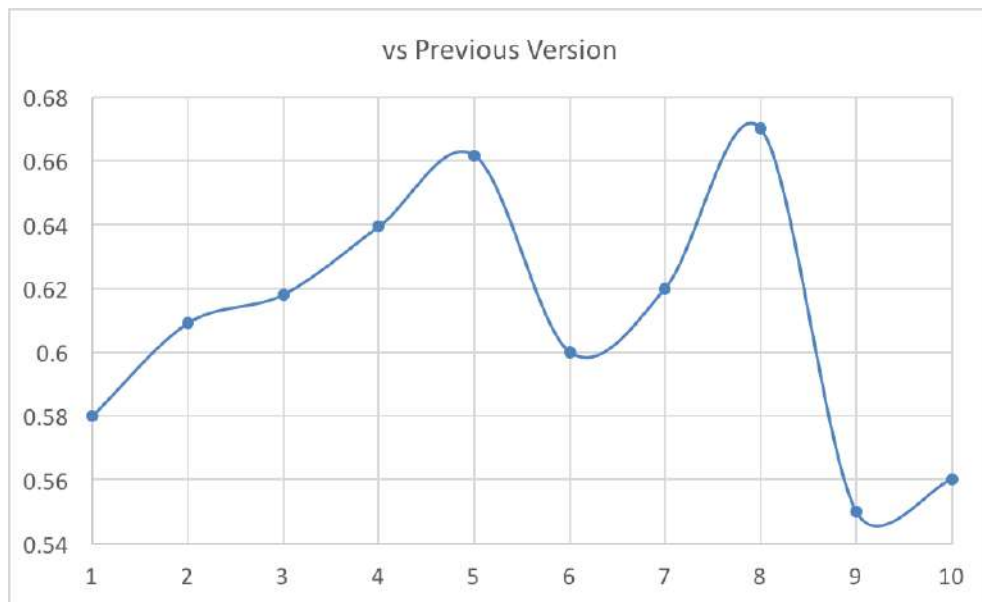


Figure 7.24 Winning Rate against Previous Version

At last, after we obtained Version 018, the Policy Network model used in our final AI model, we let Version 018 to compete with all the previous versions and recorded the winning rate. As shown in Figure 7.25., the winning rate is all smaller than 0.5, which shows that Version 018 is stronger than them all.

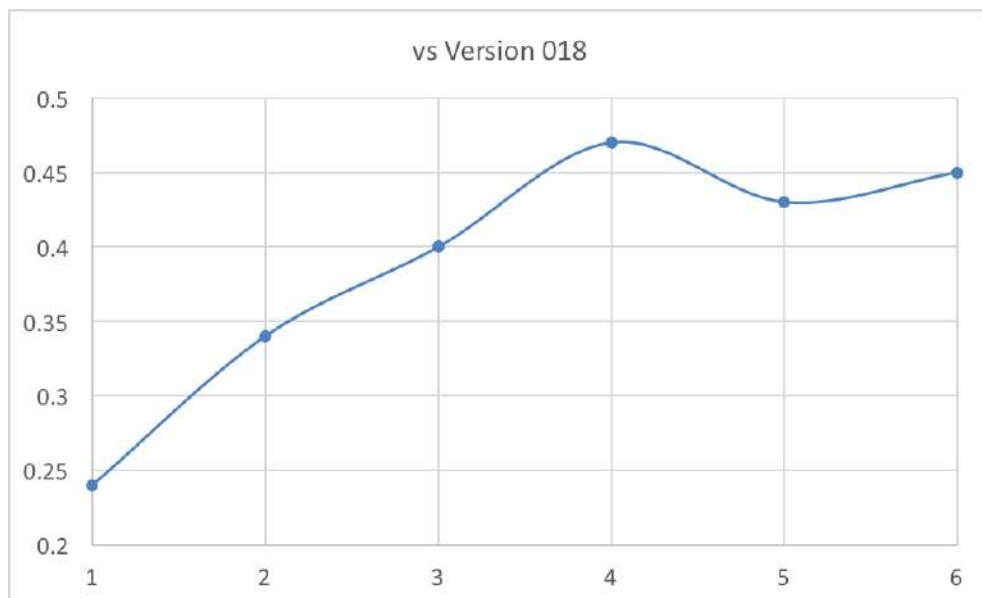


Figure 7.25 Winning Rate against Version 018

### 7.3. Results of Final Model

After Supervised Learning, there are some types of situations where our model behaved constantly bad, such as no reaction towards attack from Cannon, inappropriate response when being checked and so on. After Reinforcement Learning and combining with Evaluation Model and Minimax Searching, our model clearly performs better in those situations. Here are several specific real testing cases which demonstrate the improvement.

#### 7.3.1. Data Collected via Web Server

As we put our AI model on the Aliyun server, any people can play with the AI through Internet. And we also save all the game records, mainly the moves they made and the game results, so that we can calculate the winning rate of our AI against human players.

In our project, we post the URL in social network and invited our friends to play Chinese chess with our AI and here is the data we collected. As shown in Figure 7.26., the winning rate is 76%, wining 19 out of 25 games. On average, it takes 26.3 moves for our AI to win. The winning rate is quite high, though the tester are not professional Chinese chess players, this can still show that our Game Ai has reached a reasonable level and can compete with ordinary people.

	Number of Games	Average Number of Moves
Win	19	26.3
Lose	6	37.5

*Figure 7.26 Winning Rate against Human Players*

## 7.3.2. Testing Case 1

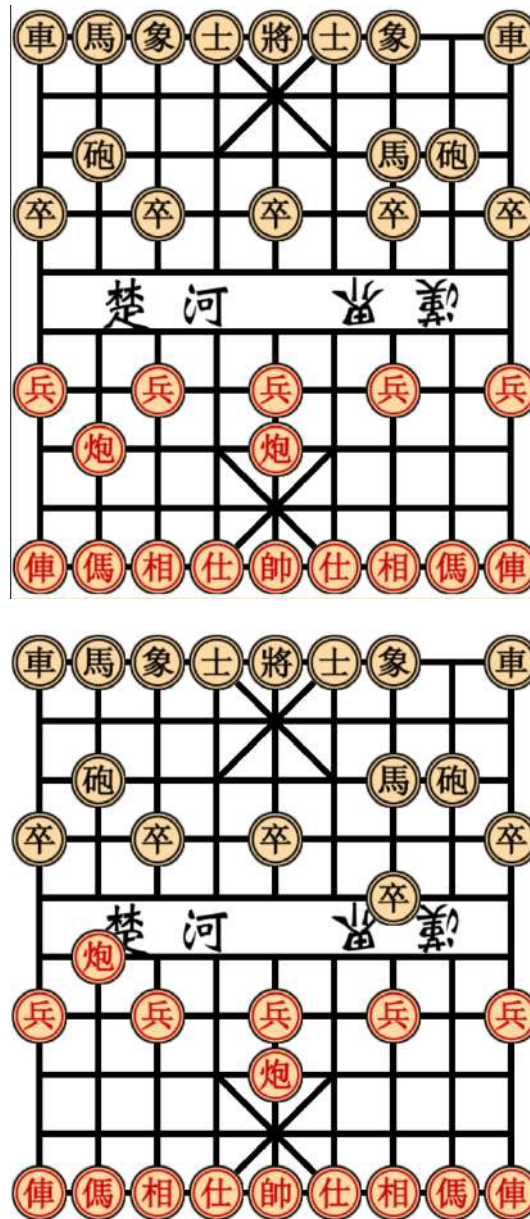


Figure 7.27

As shown in Figure 7.27, after we moved the red Cannon to the middle, our AI chose to move the black Knight forward so that it can protect the middle black Pawn, which is a common move in real games.

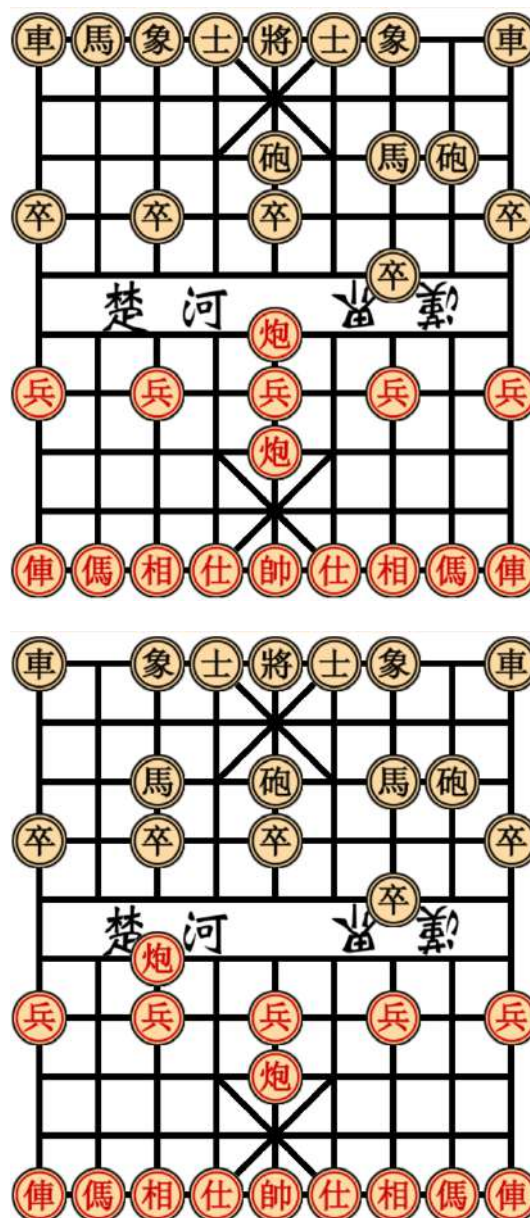


Figure 7.28

After we used the other red Cannon to check the black King, our AI can react appropriately and move the black Cannon to the middle, which is a good enough move, as this black Cannon is under protection from the black Bishops and the other black Cannon and can capture the red Cannon in next move, then we either choose to exchange the red Cannon with the black Cannon or choose to move the red Cannon away.

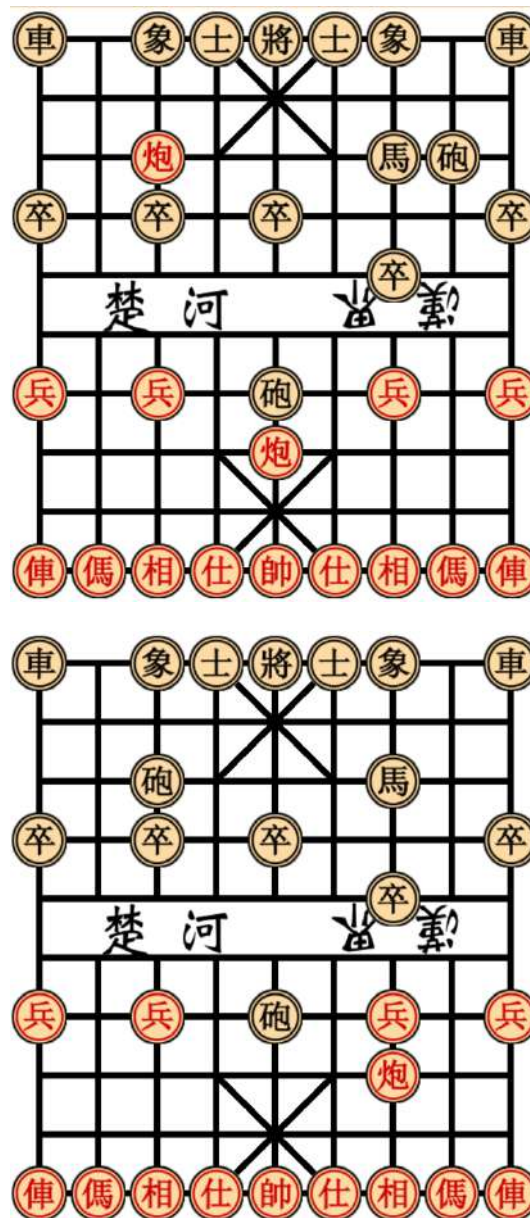


Figure 7.29

In next move, after moving the red Cannon away and capturing the black Knight, our AI chose to move the black Cannon forward to capture the middle red Pawn and check the red King. Though it appeared to leave the other black Canon under attack by the red Cannon, actually it is a smart move, as the red side has to move other pieces to protect the red King from being checked, leaving the red Cannon under attack by the black Cannon instead.



## 7.3.3. Testing Case 2

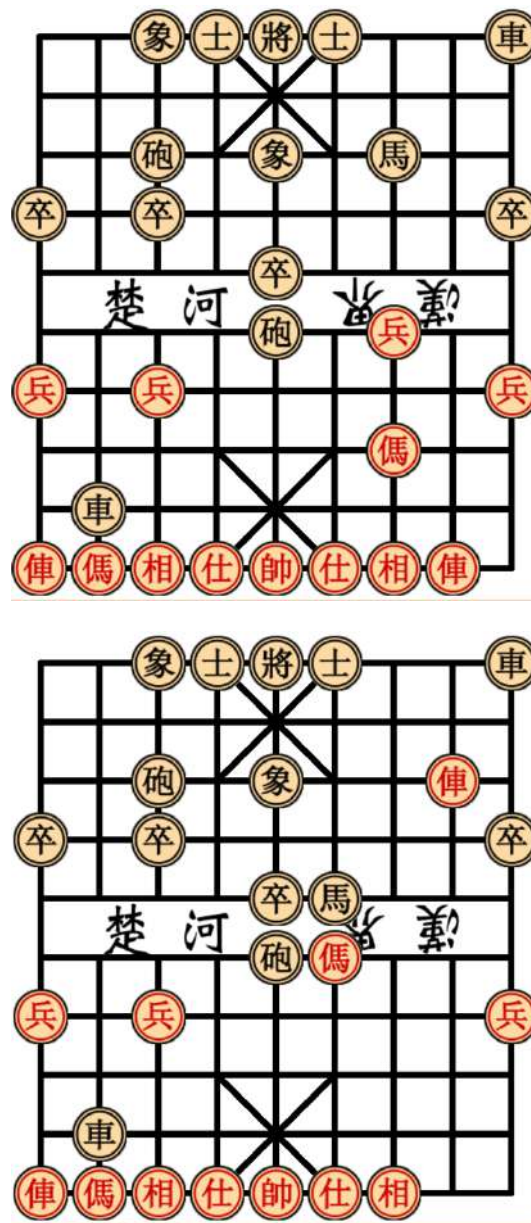


Figure 7.30

Here is an example where our AI model successfully checkmated us. After we move the red Knight forward trying to attack the black Cannon, our AI responded correctly by moving the black Cannon backward to avoid being captured. Then, we chose to move the right red Rock out so that it had more mobility. Meanwhile, our AI moved the black Rock forward right in front of the red Knight, which is really a good move, as in this way it kind of blocked the red Knight and the left red Rock at the same time and the



black Rock has high mobility, exerting great pressure on the red side.

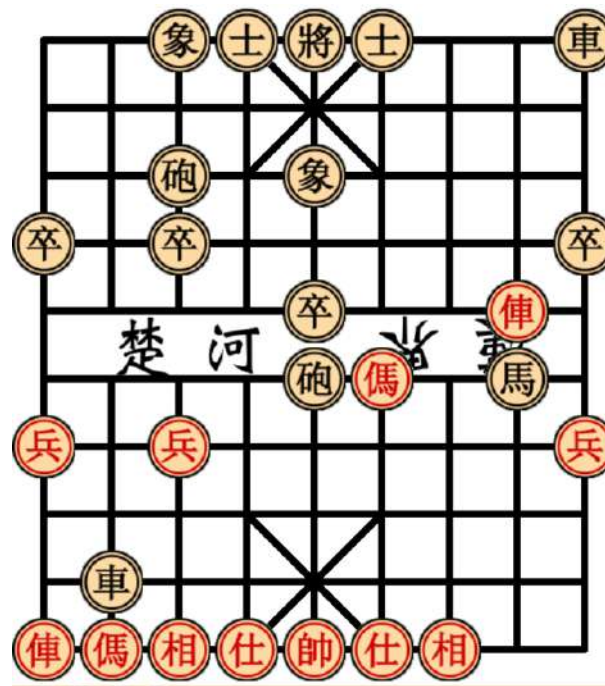


Figure 7.31

After several moves, when we moved the red Rock forward to capture the black Cannon, our AI chose to move the black Bishop backward so that the red Rock cannot directly capture the black Cannon and it is under protection from the other black Bishop so that it is not worth for the red Rock to capture the black Bishop, meanwhile leaving the red Rock under attack by the black Cannon, which is a smart move. This also indicates that our AI has learned well about the attack pattern of Cannon pieces and can properly use the pattern.

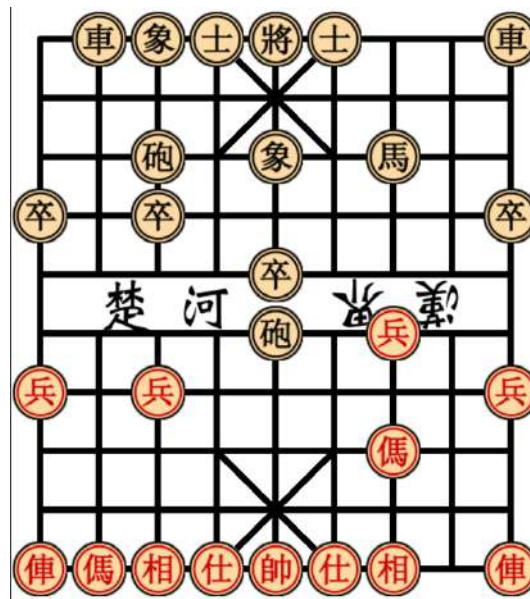


Figure 7.32

Afterwards, we moved the red Rock backward to attack the black Knight, but our AI managed to escape from being captured by moving the black Knight backward so that it is protected by the black Cannon.

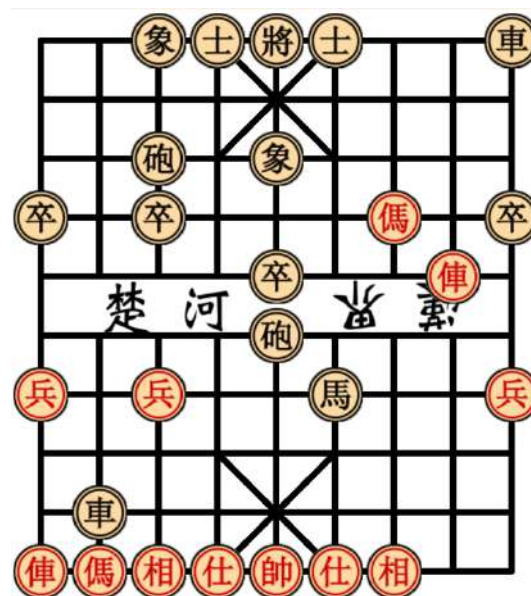


Figure 7.33

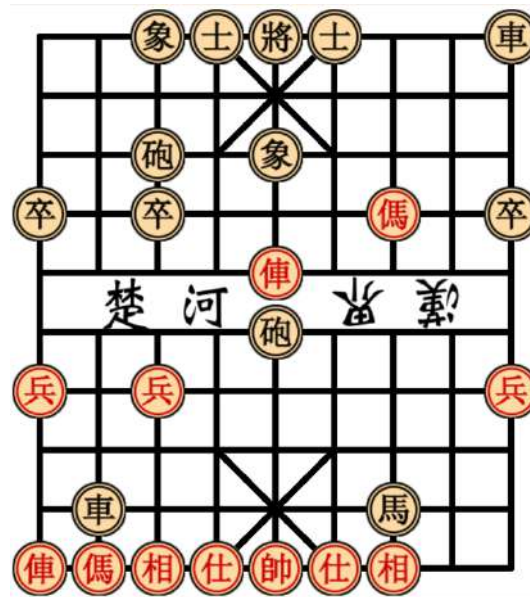


Figure 7.34

Magic happened here. We chose to move the red Knight away so that the black Cannon cannot protect the black Knight. Then our AI moved the black Knight away and we did not really realize the reason behind. After one more turn, our AI moved the black Knight forward again and successfully checkmated the red side. The red King cannot move forward otherwise it will be captured by the black Rock and it cannot move right or left as it is blocked by the red Advisors and it is being checked by the black Knight. No choice left for red side, we had to resign.

## 8. Contribution

		Zhang Haoze	Meng Zhixiang
Frontend		✓	
Server		✓	
Policy Network	Design	✓	✓
	Build		✓
	Train (SL)	✓	✓
	Test (SL)	✓	✓
	Train (RL)	✓	
	Test (RL)	✓	✓
Evaluation Network	Design		✓
	Build		✓
	Train	✓	
	Test	✓	✓
Data collection		✓	
Data process		✓	✓
Chinese chess game API			✓
Selection strategy		✓	✓

For our project, the workflow can be roughly divided into four phases: Model Design, Model Building, Model Training and Model Testing. In our project, my groupmate and I discussed and analyzed together to make the model designs. I mainly took the task to build the models. And we together tested the models, analyzed the results, modified the model design accordingly and started the process above again.

For the training process, we need training dataset and training strategy. Usually, my groupmate collected the source training data and we processed the data together so that the data can be used in our training. I also implemented the functions to extract corresponding features out of chessboard statuses for usage in model training.

For our whole game engine, there are mainly three components: user interface, game AI and server. My groupmate is mainly responsible for frontend and server, and I am also involved in it. I mainly implemented the Chinese chess game API for usage in the model. I implemented some necessary functions, such as move validation, move generation, counting possible moves, check whether one player is being checked or checkmated, and so on, so that the game can function smoothly and correctly.

For the game AI, it can be divided into three major modules: Policy Network, Evaluation Network, and Selection Strategy. My groupmate and I both are involved in the development and implementation. Through discussion, we came up with a draft design first and then modified it based on our later reflections, errors we met during implementation and the testing results.

## 9. Discussion

### 9.1. The Difficulties in Our Project

In the beginning, there were two more feature channels to be extracted from the chessboard status and fed into the Policy Network models as input, which represented some high-level information like attack-defend map and liberties of each piece. The training results, however, were not very satisfying. One possible reason would be that the values in these two feature channels could be much different from the values of other feature channels which mainly contained -1's, 0's and 1's. So, in our final model design, these channels were not included.

Although our trained Policy Network models have achieved quite good accuracy, there is one issue to be further discussed. Given one certain chessboard status, there will exist different move choices even in our training dataset, as different people would apply different strategies which may all be quite good. It would affect our training results, and more importantly inspired us to encourage exploration of different choices and add randomness when deciding the move per the output of our models.

In Term 1, after Supervised Learning, the Policy Network could hardly make effective responses when it was in check or it could capture the opposite King. It may be due to that in our training dataset, there is no training example where a King is captured, as our training dataset is extracted from realistic Chinese chess matches where the games would always end before that move is made. Besides, there are some cases where one player resigned in the middle of the game. In term 2, we fixed this problem. With the help of Evaluation Network, such statuses can be effectively detected. Additionally, we added one more checking, to check whether one side is being checked, to ensure that such cases will not happen.

Another problem is the reaction to rarely seen chessboard statuses. The Piece Selector after Supervised Learning performs well in a situation with large number of appearances

in the training dataset like initial chessboard position. And it can deal with normal unseen situation if player think normally that the Neural Network can recognize those features learned from datasets. However, in some special cases that a player did a new move which never happened before, it would be a challenge to the game AI. In fact, in the 4<sup>th</sup> game of AlphaGo VS Lee Sedol, Lee's 78<sup>th</sup> move is out of AlphaGo's mind. Neither its Neural Network nor search tree had considered this move, which led to its failure. In our plan, the Policy Network trained by supervised learning cannot deal with the problem. Figure 9.1. below shows the decisive move 78 by Lee Sedol.

This problem is sort of solved by Reinforcement Learning. In Reinforcement Learning, as we added more randomness in move selection strategies and encouraged exploration, theoretically the AI model tried some rare moves and met with many previously unseen statuses. This is also one advantage of Reinforcement Learning.



*Figure 9.1. The Decisive Move 78 by Lee Sedol*

In our project, the parameters of Neural Network must be carefully treated because a tiny change in these parameters can lead to different result. For instance, the size of filter is a key parameter. Applying 3\*3 filters, the Neural Network had relative poor performance to detect long-distance threats from cannons and rocks. When we changed the size of fields to 5\*5, the result is improved. Based on this phenomenon, we suppose that larger filters can read the global situation better because it can detect features with large size, which means it can do better in detecting long-distance moves of cannons and rocks.

The procedure through which our Game AI makes a move decision is imitating human players. When a human player recognizes a chessboard, his intuition will give him a first impression move from his experience before and eliminate some absolutely bad moves. If he is a good player, the first impression will have high accuracy. But first impressions are not consistently the best solution in the specialized case. He will do some calculation and prediction to examine if the move is good in long term or there will be some hidden trap in front.

As for our game AI, the Policy Network performs like the intuition of human, it will give some alternatives. If we only consider those alternatives themselves, it will surely perform better than completely stochastically selection. And alternatives from reinforcement trained model will be better than those do not, just like the ability of human player will increase after training with others and learn from the games played.

But that is not enough. A good player must have the ability to look forward. It is said that the top-class players can look forward to over 10 moves after the current situation. And in their minds, they will build a search tree for the entire possible alternatives to find a move that may lead to victory. During this period, they also need the intuition to reduce the complexity or they will be too much branches to think. That's how our selection strategy works. In this part, the Policy Network will help in giving the basic choices and reducing branches.

Last term, we use the accuracy as our evaluation on neural network. But in this term, we decide to use winning rates to test if the model performs well.

In term 1, our main objective was to teach the model how to play chess and to learn to obey the rules. So, we wanted the model to imitate the way professional player moves. And the accuracy of the model means how well they can move like humans. However, the high accuracy of predicting next move could not ensure it to perform well in real performance. Even with pretty high accuracy, our AI could not win against human.



Although it made high quality moves in some time, it did not behave consistently during whole game especially when facing a chessboard, it had not met before.

As our goal is to build a chess AI, we need to prove that it is at good level. So we use winning rate to make comparison between two models. Obviously, if we use a version of AI playing with itself, the winning rate of each side should be around 50%. If iteration x has over 55% winning rate against another iteration y, we can assert x outplay y. But we cannot ensure if x outperforms y and y outperforms z then x will outperform z. All the versions need to be test separately.

## 9.2. The Reason to Use Neural Network

When we trained our Evaluation Network, we mentioned that we used one evaluation function from an open-source API to assign target values for our training dataset.

As the training results cannot be perfect, the outputs from the trained Evaluation Network cannot be as accurate as the evaluation function we used. Consequently, our Game AI could be stronger if it obtained outputs from the evaluation function directly instead of using the Evaluation Network.

We still chose to build and train an Evaluation Network on our own, however, because the neural network can learn to evolve.

On one hand, the evaluation function we found may not be perfect and we may find some better way to evaluate a chessboard status. Using the Evaluation Network, instead of coding a function and doing the calculation in a fixed way, makes it possible that the Game AI can adopt new methods to do evaluation and learn the new results.

On the other hand, by using neural network, the Game AI can be easily updated after competing with other game engines, human players or even itself. AlphaGo made a huge success by competing with itself and learning from the experiences of master human players.

Learning is a precious and powerful skill that cannot be more empathized. Learning can lead to great potential that cannot be ignored. By using neural network, the Game AI can learn to make progress and evolve to be stronger.

## 10. Conclusion

As discussed in the section above, we can safely conclude that the performance of our current game AI on real game has greatly improved. In the first term, our AI could not react to some specific situations like long-distance check from cannon and made irrelevant move. In this term, it has learned how to evade from check after Reinforcement Learning and adding evaluation model. For complex situation, it will generate a list of alternative moves and choose among them. From the game playing cases, we can see it can know when to capture opposite pieces, when to check and how to checkmate. Against website visitors, it reaches a winning rate of 76%, meaning that the game AI can win against normal level of amateur players.

On the opposite hand, the game AI still has some aspects to improve. Our Policy Network needs more training to improve its performance. And we should provide more features to the evaluation model or find some better structure because the accuracy of the prediction from the evaluation model is not so satisfying.

In conclusion, our game AI has reached a fair level but further training is still needed to improve its performance.

## 11. Future Work

Our future work should do further Reinforcement Learning. From our results, the performance of our neural network keeps increasing. We can assume that the level of the game AI can grow higher with more times of Reinforcement Learning iteration.

Also, we can find some other way to improve the AI. We shall try various value of parameters and different structures of the neural network, and see if we can improve the performance of the game engine. For example, we only try to use convolutional neural network to piece selector and move selectors. We can try some other structure like full connected network or Long-Short Term Memory.

Another possible method is that not to use the combination of piece selector and move selectors and use only one neural network to give the predicted moves. In our current model, we have eight neural networks in the same time. These are not very intuitive and making our training process hard. Maybe a single model will work better and easier to train.

In this term, we tried different kinds of approach to give negative reward to the neural network. The missing of negative result causes the prediction of network centralized and the training efficiency can't increase. We shall find a method to apply negative feedback and train a stronger model.

## 12. Reference

- [1] Edwards, S. J. (1994). Portable Game Notation Specification and Implementation Guide.
- [2] Mohri, M., Rostamizadeh, A., & Talwalkar, A. (2012). Foundations of machine learning. MIT press.
- [3] Geman, S., Bienenstock, E., & Doursat, R. (1992). Neural networks and the bias/variance dilemma. *Neural computation*, 4(1), 1-58.
- [4] James, G. M. (2003). Variance and bias for general loss functions. *Machine Learning*, 51(2), 115-135.
- [5] Nielsen, M. (2016). *Neural Networks and Deep Learning*
- [6] Hahnloser, R. H., Sarpeshkar, R., Mahowald, M. A., Douglas, R. J., & Seung, H. S. (2000). Digital selection and analogue amplification coexist in a cortex-inspired silicon circuit. *Nature*, 405(6789), 947-951.
- [7] Anzai, Y. (2012). *Pattern Recognition & Machine Learning*. Elsevier.
- [8] [Online]. Available: <https://www.tensorflow.org>
- [9] [Online]. Available: <https://github.com/leenmie/chinese-chess>
- [10] Oshri, B., & Khandwala, N. Predicting Moves in Chess Using Convolutional Neural Networks.
- [11] Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., ... & Dieleman, S. (2016). Mastering the Game of Go with Deep Neural Networks and Tree Search. *Nature*, 529(7587), 484-489.
- [12] LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep Learning. *Nature*, 521(7553), 436-444.
- [13] Lai, M. (2015). Giraffe: Using Deep Reinforcement Learning to Play Chess. arXiv preprint arXiv:1509.01549.
- [14] O'Shea, K., & Nash, R. (2015). An Introduction to Convolutional Neural Networks. arXiv preprint arXiv:1511.08458.
- [15] Schmidhuber, J. (2015). Deep Learning in Neural Networks: An Overview. *Neural Networks*, 61, 85-117.
- [16] [Online]. Available: [https://en.wikipedia.org/wiki/Convolutional\\_neural\\_network](https://en.wikipedia.org/wiki/Convolutional_neural_network)
- [17] [Online]. Available: [https://en.wikipedia.org/wiki/Supervised\\_learning](https://en.wikipedia.org/wiki/Supervised_learning)
- [18] Tromp, J. (2016). Number of legal Go positions. preprint <http://homepages.cwi.nl/tromp/go/legal.html>.
- [19] David, O. E., Netanyahu, N. S., & Wolf, L. (2016, September). DeepChess: End-to-End Deep Neural Network for Automatic Learning in Chess. In *International Conference on Artificial Neural Networks* (pp. 88-96). Springer International Publishing.
- [20] Plaat, A., Schaeffer, J., Pijls, W., & De Bruin, A. (2017). A Minimax Algorithm Better Than Alpha-beta?: No and Yes. arXiv preprint arXiv:1702.03401.
- [21] Vandermosten, T. Monte Carlo Tree Search applied to Go: reaching omega.

- [22] Korf, R. E., & Chickering, D. M. (1996). Best-first minimax search. *Artificial intelligence*, 84(1), 299-337.
- [23] Beal, D. F. (2014). Benefits of minimax search. *Advances in computer chess*, 3, 17-24.
- [24] Block, M., Bader, M., Tapia, E., Ramírez, M., Gunnarsson, K., Cuevas, E., ... & Rojas, R. (2008). Using reinforcement learning in chess engines. *Research in Computing Science*, 35, 31-40.
- [25] Chen, S. J. Y. J. C., Yang, T. N., & Hsu, S. C. (2004). Computer chinese chess. *ICGA Journal*, 27(1), 3-18.
- [26] Ong, C. S., Quek, H. Y., Tan, K. C., & Tay, A. (2007, April). Discovering chinese chess strategies through coevolutionary approaches. In *Computational Intelligence and Games, 2007. CIG 2007. IEEE Symposium on* (pp. 360-367). IEEE.
- [27] Trinh, T., Bashi, A., & Deshpande, N. (1998). Temporal difference learning in Chinese Chess. *Tasks and Methods in Applied Artificial Intelligence*, 612-618.
- [28] Mannen, H., Wiering, M., & Intelligence, C. A. (2004, January). Learning to play chess using TD ( $\lambda$ )-learning with database games. In *Benelearn'04: Proceedings of the Thirteenth Belgian-Dutch Conference on Machine Learning* (pp. 72-79).
- [29] Sutton, R. S., & Barto, A. G. (1998). *Introduction to reinforcement learning* (Vol. 135). Cambridge: MIT Press.
- [30] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. (2013). Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*.