Move Prediction in Go with the Maximum Entropy Method

Nobuo Araki*, Kazuhiro Yoshida*, Yoshimasa Tsuruoka† and Jun'ichi Tsujii*†‡
* Graduate School of Information Science and Technology, The University of Tokyo
ark@is.s.u-tokyo.ac.jp, kyoshida@is.s.u-tokyo.ac.jp,tsujii@is.s.u-tokyo.ac.jp
† School of Computer Science, The University of Manchester
yoshimasa.tsuruoka@manchester.ac.uk
‡ NaCTeM (National Centre for Text Mining)

Abstract— **We address the problem of predicting moves in the board game of Go. We use the relative frequencies of local board patterns observed in game records to generate a ranked list of moves, and then apply the maximum entropy method (MEM) to the list to re-rank the moves. Move prediction is the task of selecting a small number of promising moves from all legal moves, and move prediction output can be used to improve the efficiency of the game tree search. The MEM enables us to make use of multiple overlapping features, while avoiding problems with data sparseness. Our system was trained on** 20000 **expert games and had** 33.9% **prediction accuracy in** 500 **expert games.**

Keywords: maximum entropy method, board games, Go, move prediction, re-ranking

I. Introduction

In Go[1], the size of the board is usually 19×19, and the player can place a stone on most of the empty spaces (i.e, those without stones). Therefore, as there are too many legal places to apply a simple Minimax search algorithm to a game tree search in Go, we need to select a small number of moves[2] from all legal moves (forward pruning) while searching the game tree. The prediction of moves is a way to perform such selection. Move prediction is ranking all legal places by the probabilities that experts (strong human players) will select the moves. Accurate move prediction routines can be used for forward pruning because experts select a small number of promising moves to think deeply (they seem to perform forward pruning unconsciously).

We used the maximum entropy method (MEM) to predict moves.

There have been several previous studies on predicting moves in Go. Bouzy and Chaslot [2] showed the first 40 moves could be accurately predicted using K-nearest-neighbor patterns. Van der Werf et al. [3] attained 25% accuracy[3] using a neural network with various features. Stern et al. [4] [5] achieved 34% accuracy with a simple system of pattern matching trained on a number of expert games.[4]

[1]A great deal of information about Go can be found at http://gobase.org/. [1]

[2]Move means placing a stone in Go.

[3]That is, the expert move was in the first rank in 25% of all the ranking lists prepared by the system.

[4]The accuracy of 34% seems very low, but it is the top score in various research. (The creator of Moyo Go Studio [6] claims that it attained 42% accuracy, but he gave no explanation about what data was used for training and evaluation.)

However, there is room for improvement in Stern et al.'s work because multiple characteristics that have their respective effects on move prediction are merged into a single feature. Their method does not treat multiple characteristics appropriately. We applied MEM in this research and used the features of previous moves because it can manage multiple features, and information on previous moves can be used as an important feature for predicting moves.[5] We used this method of re-ranking the candidate moves and we achieved an accuracy close to Stern et al.'s with a relatively small amount of training data.

We first describe Stern et al.'s research on predicting moves in Go in section II. This was the main basis of our research. We also explain Zobrist hashing [7]. We then explain MEM. Section III, explains our machine learning method which uses MEM. Section IV presents the experiments. We tuned a hyper parameter, changed the amount used for re-ranking, and trained our system with 20000 matches of data. We also had our system play with GnuGo [8]. Section V discuss the utility of our system and future work.

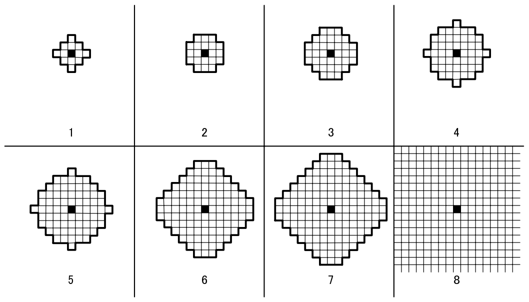
II. Background

We will first describe Stern et al.'s methods [4] [5], on which our work is based. They used patterns of stone positions as features for machine learning. As we also used them, we will explain mainly these patterns. We will next explain Zobrist hashing [7] that is used when comparing and storing patterns. We will then describe MEM, which is used to deal with multiple overlapping features.

A. Patterns in Stern et al.'s work

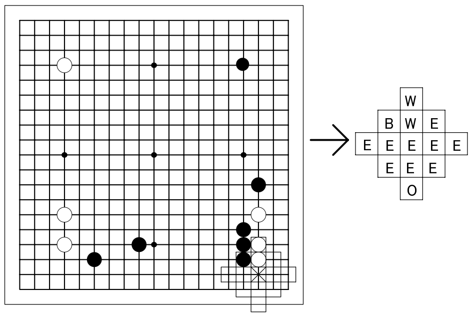
Pattern templates are first prepared in the pattern matching algorithm proposed by Stern et al. Some of the templates they used are shown in Fig. 1. (In Stern et al. [5], other pattern templates were added, but we didn't use them.) By using these, patterns are extracted from expert game records. These templates define the shape and range of patterns in stone position. There is an example of a pattern being extracted from a game record in Fig. 2. A pattern is represented by {'black', 'white', 'empty',or 'out of board'} for each position in the pattern. Patterns that are symmetrical, i.e. a set of patterns which can be exactly matched by rotation, mirroring,

[5]Strong players usually select moves that maintain consistency and previous moves are good clues to maintain consistency and predicting moves.



The black square in each template is its center of the template. This center is on the next-move candidate (details are described in Section II-C) in pattern extraction. The largest template, number 8, covers the whole board (only part of the template is shown because of space limitations).

Fig. 1. Pattern templates used by Stern et al. [4]



This shows pattern extracted from the record of a game with pattern template 1. 'B' means 'black', 'W' means 'white', 'E' means 'empty' (nothing placed on it yet), and 'O' means 'out of board'.

Fig. 2. Example of pattern being extracted

color reversal or their combination should be treated as the same pattern.

The patterns themselves are not used when comparing and storing them, but their hash values are used to save time and for storage. The hash values are calculated by 64 bit Zobrist hashing [7], (details of this are described in section II-B) where the symmetric patterns can be reduced to a single hash value by choosing the minimum of hash values, i.e., by calculating a hash value for every symmetric pattern and choosing the minimum.

Other features related to Go tactics and stone positions that have been introduced above are treated as one pattern in Stern et al.'s work [5], i.e., tactical features are converted to 64 bit numbers and XORed to the hash value of the stone positions.

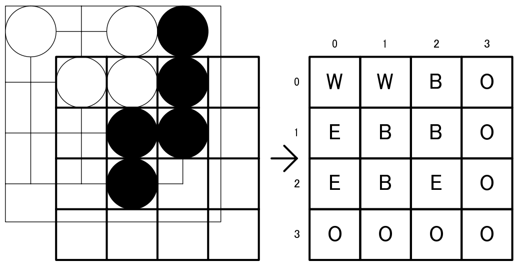


Fig. 3. Example for Zobrist hashing

B. Zobrist hashing

Zobrist hashing [7] is a technique for creating hash keys, usually from something like a Go position. It enables fast comparing patterns of stone position. First, a table of random values is created, with each position on the pattern having a value associated with it. Next, the hash key is initialized by 0. Then the values of the table, which corresponds to the pattern, are XORed together to create the final hash key. For example, the hash key calculation of Fig. 3 is like this:

1) Creating a table of random values

```
int table[4][4][4];
for (int i = 0; i < 4; i++){
  //row index
  for (int j = 0; j < 4; j++){
    //column index
    for (int k = 0; k < 4; k++){
      //state index
      table[i][j][k] = random();
}}}
```

2) Initialize a hash key

```
int hash_key = 0;
```

3) XORed together to create the final hash key

```
for (int i = 0; i < 4; i++){
  //row index
  for (int j = 0; j < 4; j++){
    //column index
    hash_key ^= table[i][j][at(i,j)];
    //at(i,j) means a color at (i,j)
}}

(in this case,
const int B = 0, W = 1, E = 2, O = 3;
hash_key
  ^= table[0][0][W]^table[0][1][W]
    ^table[0][2][B]^table[0][3][O]
    ^table[1][0][E]^table[1][1][B]
    ^table[1][2][B]^table[1][3][O]
    ^table[2][0][E]^table[2][1][B]
    ^table[2][2][E]^table[2][3][O]
```

```
        ^table[3][0][O]^table[3][1][O]
        ^table[3][2][O]^table[3][3][O];)
```

This method has several advantages. The most important one on the Stern's work and on our approach is that the hash key can be updated incrementally. If only one point on a pattern changes, just two XOR operations are necessary to calculate the new hash key:

```
hash_key
^= table[i][j][old]^table[i][j][new];
```

The importance of this in this work is discussed in section II-C.

C. Machine learning with patterns

Stern et al's system was trained with 181000 matches of game records of expert players. By using all the records, the system first constructed a pattern dictionary and then learned the scores of the patterns.

A dictionary of patterns was constructed by extracting patterns from the training data whose centers were on the actual experts' moves, for each pattern template. This dictionary contained patterns that had been selected by expert players (placing a stone on the center). Only patterns that had appeared more than once were included in the dictionary to limit their number and to ensure generalization to unseen positions.

The scores of the patterns were learned next. The learning routine is as follows:

On every expert move,

1) For every position that is a legal move, extract the largest pattern in the dictionary whose center is on that position and whose stone positions match the board configuration.
2) Train the system to classify the extracted pattern whose center is on the actual expert move as 'good' and to classify the other extracted patterns as 'bad'.

In this routine, the advantage of Zobrist hashing (a hash value can be incrementally updated, i.e., it can be calculated with the previous hash value and the difference) plays an important role in saving computational time when calculating hash values. The algorithm that uses this advantage is as follows:

1) Before the start of the game (no stones have been placed on the board), calculate a hash value for each position on the board, for each pattern template, and for each symmetric pattern.
2) On every move, update the hash values whose pattern have the position of the move in their own pattern templates.

Using this algorithm, hash values can be obtained much faster than directly calculating them for all patterns every time.

In Stern et al.'s experiment, a full ranking model with ADF and EP [9], and an independent Bernoulli model were used as the prediction model.

D. Predicting expert moves

Expert moves can be predicted with the scores of patterns obtained by the method in Section II-C as follows:

1) For every position that is a legal move, extract the largest pattern in the dictionary whose center is on that position and whose stone positions match the board configuration.
2) Rank the legal moves by the score of the extracted pattern whose center is on the position of each move.

E. Maximum entropy method

Binary feature functions such as in (1) are used for Maximum Entropy Method (MEM) to estimate joint probability distribution model $P(x, y)$ from the training data $\{(x_1, y_1), (x_2, y_2), \ldots, (x_N, y_N)\}$,

$$\mathcal{F} = \{f_i : (x, y) \longmapsto \{0, 1\}, i \in \{1, 2, \ldots, n\}\} \qquad (1)$$

Let $C(x, y)$ be the number of appearances of (x, y), then the relative frequency of (x, y) is:

$$\tilde{P}(x, y) = \frac{C(x, y)}{N}. \qquad (2)$$

The probability distribution that is defined by $\tilde{P}(x, y)$ is called the "empirical probability distribution". The expectation of feature f_i defined by $\tilde{P}(x, y)$ is:

$$E_{\tilde{P}}[f_i] = \sum_{x,y} \tilde{P}(x, y) f_i(x, y). \qquad (3)$$

Also, the expectation of feature f_i defined by model $P(x, y)$ is:

$$E_P[f_i] = \sum_{x,y} P(x, y) f_i(x, y). \qquad (4)$$

If model $P(x, y)$ properly represents the characteristics of the training data, $E_{\tilde{P}}[f_i]$ must be equal to $E_P[f_i]$. Therefore, the following equation must be true.

$$\sum_{x,y} P(x, y) f_i(x, y) = \sum_{x,y} \tilde{P}(x, y) f_i(x, y) \qquad (5)$$

This is called a "constraint equation".

In MEM, the most uniform distribution of those that satisfy (5) is estimated. The uniformity is calculated using the entropy, $H(P)$:

$$H(P) = -\sum_{x,y} P(x, y) log P(x, y) \qquad (6)$$

The set of models that satisfies (5) when estimating $P(x, y)$ by using $f_i (1 \leq i \leq n)$ is defined as:

$$\mathcal{P} = \{P | E_P[f_i] = E_{\tilde{P}}[f_i], i \in \{1, 2, \ldots, n\}\} \qquad (7)$$

The estimated model maximizes entropy.

$$P* = argmax_{P \in \mathcal{P}} H(P) \qquad (8)$$

Model P, which satisfies (8), can be represented as:

$$P_\Lambda(x,y) = \frac{1}{Z_\Lambda}\exp\left(\sum_i \lambda_i f_i(x,y)\right) \quad \textit{and} \tag{9}$$

$$Z_\Lambda = \sum_{x,y}\exp\left(\sum_i \lambda_i f_i(x,y)\right) \tag{10}$$

$\Lambda = \{\lambda_1, \ldots, \lambda_n\}$ is a set of parameters of model $P(x,y)$, and λ_i is a weight of f_i. Z_Λ is the normalizing factor for $\sum_{x,y} P_\Lambda(x,y) = 1$.

In MEM with inequality constraints [10], (5) is alleviated as:

$$A_i \geq E_{\tilde{P}}[f_i] - E_P[f_i] \geq -B_i \quad (A_i, B_i > 0) \tag{11}$$

We used the "SS Maxent" library [11] in this research, which is a simple C++ class library for maximum entropy classification. In this library, A_i, B_i in (11) is:

$$A_i = B_i = W \times \frac{1}{N} \quad (W \text{ is a width factor}). \tag{12}$$

III. Method

When dealing with multiple characteristics that are not independent, merging them into one feature is inappropriate because it causes problems with data sparseness, i.e., it treats two features that are almost the same but only one characteristic is different as two completely separate features.

For example, in our experiment, we wanted to use the characteristics of previous moves for machine learning. However, if we combine the characteristics of previous moves and the current pattern of stone positions as one feature, two situations that have the same current pattern but have different previous moves will be treated as completely separate situations. This is not efficient because some moves are almost independent of the characteristics of previous moves but only dependent on the current pattern of stone positions.

Therefore, we used MEM, which can manage multiple overlapping features, while avoiding the problems of data sparseness. However applying MEM to machine learning to predict moves in the same way as Stern et al. [4] [5] has a problem with lack of memory. Applying MEM to this means that we have to store features on each legal place on each actual expert move on each game in the training data in the machine memory. If the features of one legal place occupy 180 bytes[6], the average number of moves in one game is 250, and the amount of training data is 20000[7], we need $180 \times \frac{361+(361-250)}{2} \times 250 \times 20000 = 212.4GB$ of machine memory merely to store the features. We therefore used MEM for re-ranking, i.e., we used relative frequencies[8] to generate a ranked list of moves, and then applied MEM to the list to re-rank the moves.

We shuffled the order of the training data for learning[9], and divided them into two equally sized sets. We used one (data A) for preparing a pattern dictionary and for learning relative frequencies, and the other (data B) for MEM.

The training phase is as follows:

1) Prepare a pattern (of stone positions, without tactical characteristics) dictionary in the same way as in [4] from data A.
2) Calculate the relative frequencies of the patterns using data A.
3) Rank all legal moves by using data B utilizing the relative frequencies and train the system with MEM using the top n samples in the ranking.

The move-predicting phase is as follows:

1) Rank all legal moves using the relative frequencies.
2) Re-rank the top n moves in the ranking with the MEM system.

The details on the training phase and move-predicting phase are described in the following sections.

A. Learning relative frequencies

The algorithm for calculating the relative frequencies of stone positions is as follows:

1) Construct the dictionary of the patterns of stone positions in the same way as in [4]. (We used eight pattern templates in Fig. 1).
2) Prepare two counters, 'used' and 'unused' for each pattern in the dictionary.
3) On every expert move,
    a) For every legal-move position, extract the largest pattern in the dictionary whose center is on that position and whose stone positions match the board configuration.
    b) For each pattern extracted in (3a), if the center of the pattern is the actual expert move, increment the 'used' counter of the pattern, otherwise, increment 'unused'.
4) Calculate the relative frequency of each pattern using Laplace's law[10] as:

$$\frac{(\text{'used' of the pattern}) + 1}{(\text{'used' of the pattern}) + (\text{'unused' of the pattern}) + 2} \tag{13}$$

B. Learning with MEM

After the training described in Section III-A, our system was trained with MEM as follows:

On each board configuration,

1) Rank all legal moves using the respective relative frequencies of the largest patterns in the dictionary whose centers are on the positions and whose stone positions match the board configuration.
2) Use the top n rankings as training samples for MEM. The features used for training are (See Fig. 4.):

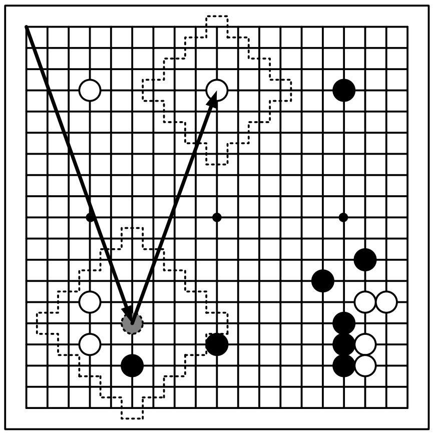
[6]This is the amount in our experiment discussed in this paper.

[7]This is also the amount in our experiment discussed in this paper.

[8]Calculating relative frequencies is fast and consumes less memory.

[9]The bias in the two parts of the training data may have an adverse influence on results because Go tactics change over time.

[10]Laplace's law is a discounting method and relative frequencies are calculated with it as: $P(X_i) = \frac{C_i+1}{N+V}$ $(i \in \{1, \ldots, V\})$ In this experiment, $V = 2$.



As the features of place α (gray stone), we used the pattern around it, its coordinates, the patterns around the previous moves (these patterns matched the board configuration when the previous moves were played), and the relative coordinates of the previous moves to α. Only one previous move is shown in this figure as the features of α, but we actually used four previous moves.

Fig. 4. Features used for MEM

- The largest pattern in the dictionary whose center is on the current move and whose stone positions match the current board configuration,
- The coordinates of the current move,
- The largest patterns in the dictionary whose centers are on the four previous moves and whose stone positions match the board configuration at the time the moves were played, and
- The relative coordinates of the four previous moves to the current move.

The actual expert move is labeled as 'good', and the others are labeled as 'bad'.[11]

Finally, the MEM system is trained to divide 'good' and 'bad'.

C. Predicting moves

All legal moves are first ranked using the respective relative frequencies of the largest patterns in the dictionary whose centers are on the positions and whose stone positions match the board configuration to predict moves. Then, the top n rankings are re-ranked by MEM. The MEM system estimates probability $P(\text{'}good'|features)$ and moves are re-ranked by using it.[12]

[11]If a 'good' sample is not in the top n, add a 'good' sample to the top n training samples.

TABLE I
TUNING OF WIDTH FACTOR

Width factor	Rank 1 accuracy
0.0	27.03%
0.1	27.46%
0.2	27.67%
0.3	27.95%
0.4	28.02%
0.5	28.15%
0.6	28.21%
0.7	28.19%
0.8	28.20%
0.9	28.24%
1.0	28.20%

IV. Experiment

We conducted our experiment using the records in the GoGoD database [12].

A. Tuning hyper-parameter

We adjusted the width factor in MEM [10] by using 2000 matches as the training data and 500 matches as the development data for evaluation. We divided the training data into two equal sets. We used one to prepare the pattern dictionary and learn the relative frequencies. We used the other to learn with MEM. We used the top 20 for re-ranking.

The results are in Table I. A width factor of 0.9 appears acceptable..

B. Changing amount used for re-ranking

We changed the amount, used for re-ranking the list generated with relative frequencies, from 20 to 80. We set the width factor to 0.9. We used 2000 matches as the training data and 500 matches as the development data for evaluation. We divided the training data into two equal sets. We used one to prepare the pattern dictionary and learn relative frequencies. We used the other to learn with MEM.

We also conducted an experiment with no re-ranking, i.e., all the training data were only used to prepare the pattern dictionary and learn the relative frequencies.

The results are in Table II.[13] The "0" column means that there was no re-ranking.

At ranks 1, 5, and 10 the results with re-ranking yielded better outcomes than those without re-ranking. However, for the others, the results with re-ranking did not yield better outcomes than those without re-ranking.

Increasing the amount used for re-ranking had a bad influence on rank 1, but a good influence on ranks 10, 20, 40, and 60. However, the results with re-ranking only yielded better results at rank 10 than those without re-ranking.

[12]The '$features$' are the same as those that are used for MEM training in Section III-B.

[13]"The cumulative density of rank x is $y\%$" means that $y\%$ of all expert moves are in the top x in the ranking by our system. The higher $y\%$ is, the better the system is.

TABLE II

CHANGING AMOUNT USED FOR RE-RANKING

Rank	Cumulative density				
	0	20	40	60	80
1	21.05%	28.24%	27.55%	26.93%	26.92%
5	46.19%	52.70%	52.91%	52.54%	52.85%
10	59.25%	61.68%	63.05%	62.52%	62.45%
20	72.43%	69.11%	72.58%	72.37%	71.91%
40	83.75%	79.85%	79.85%	81.38%	81.38%
60	88.62%	84.63%	84.63%	84.63%	85.85%
80	91.32%	87.53%	87.53%	87.53%	87.53%

TABLE III

USING 5000, 10000, 15000,AND 20000 TRAINING DATA

Rank	Cumulative density			
	5000	10000	15000	20000
1	31.04%	32.72%	33.73%	33.94%
5	56.54%	58.61%	59.81%	60.47%
10	65.56%	67.85%	68.94%	69.50%

Applying re-ranking with MEM was good for selecting about 10 moves from the list generated with relative frequencies. The amount that should be used for re-ranking depends on the amount that will be used after the move is predicted (i.e., if only the top 1 is necessary, re-ranking the top 20 is good, but if the top 10 is necessary, re-ranking the top 40 is good.).

C. Using 5000, 10000, 15000,and 20000 training data

Setting the width factor to 0.9, we carried out experiments using 5000, 10000, 15000,and 20000 matches as the training data and 500 matches as the test data. We divided the training data into two equal sets. We used one to prepare the pattern dictionary and learn the relative frequencies. We used the other to learn with MEM. We used the top 20 for re-ranking.

The results are listed in Tables III and IV.

Compared with Stern et al. [4] who used 20000 matches as the training data and 500 matches as the test data, our system yielded better results in cumulative density at ranks 1, 5 and 10. However, our system was outperformed at rank 20. We could not obtain better results than Stern et al. [5] who used 181000 matches for training, but we attained almost their accuracy at rank 1.

The experiment using 20000 matches for training took about 8.75 days[14] and used about 16 GB of memory. Moreover, the expected increase in accuracy created by increasing the amount of training data is very small (see Table III). Therefore, we can say that using more than 20000 matches for training data is not practical to attain greater accuracy. We believe we need to use better features instead.

*D. Match with GnuGo*3.6 *[8]*

We had our system (trained with 20000 matches of data described in section IV-C) play against GnuGo3.6 [8]. Our system always selects moves ranked 1. The results are presented in Figs. 5, 6, and 7. Our system was beaten by GnuGo3.6, but many of the moves were not too bad.

[14]We used Intel(R) Xeon(R) 3.0 GHz machine.

TABLE IV

COMPARISON WITH STERN ET AL. [4] AND [5]

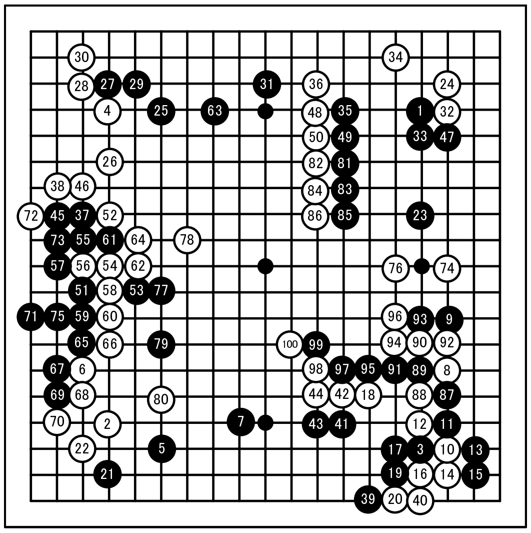
Rank	Cumulative density in our experiment (20000 data)	Cumulative density in [4] (20000 data)	Cumulative density in [5] (181000 data)
1	33.94%	26%	34%
5	60.47%	55%	66%
10	69.50%	68%	76%
20	77.16%	81%	86%

$1 - 14$ were not bad. 15 was a bad move but our system could not correspond to it correctly (16 and 17 were not bad, but 18 and 20 were bad). $21 - 38$ were not bad. 39 and 40 were bad. $41 - 54$ were not bad. 55 was strange. $56 - 76$ were not bad 77 and 79 were bad and 78 and 80 correctly corresponded to them. $81 - 92$ were not bad. 93 was bad. $94 - 115$ were not bad. 116 was bad but GnuGo could not correspond to it correctly. $117 - 129$ were not bad. 130 was bad (our system could not understand the capture of stones). $131 - 163$ were not bad. 164 was bad. $165 - 185$ were not bad. 186,188, and 190 were nonsensical (187,189, and 191 correctly corresponded to them). $192 - 199$ were not bad. 200 was bad. $201 - 205$ were not bad. 206 was bad (our system could not understand the life and death of stones). $207 - 221$ were not bad. 222 was nonsensical. 223 was not bad. 224 was bad (our system could not understand the connection of stones). 225,227, and 229 correctly cut and killed white stones. 230,232,234,236, and 238 were bad (231,233,235,237, and 239 correctly corresponded to them). 242 and 244 were nonsensical and 243 and 245 correctly corresponded to them. $246 - 249$ were not bad. 250 was nonsensical. $251 - 255$ were not bad. 256 was nonsensical. $257 - 261$ were not bad. 262 and 264 were bad. 263 and 265 were not bad. 266,268,and 270 were nonsensical. 267,269,and 271 were not bad. $272 - 292$ were nonsensical. 293 was passed.

V. DISCUSSION AND CONCLUSION

We demonstrated that using MEM can attain a high degree of accuracy with a relatively small amount of training data.

However, there might be a problem in using our system in a Go program. We used experts' previous moves in section IV, as a feature for machine learning, but in a Go program we have to use its previous moves and its opponent's previous moves, which may be bad moves because computer Go is still so weak. Using bad moves as features to predict moves could have a bad effect on prediction because such bad moves are rarely observed in the training data. We may have to consider features other than previous moves or we may have to consider using non-expert (poor player) matches as training data. We also have to balance prediction accuracy, time consumption, and memory consumption.



Black was GnuGo's turn and white was our system's turn.

Fig. 5. Match with GnuGo3.6 [8] I

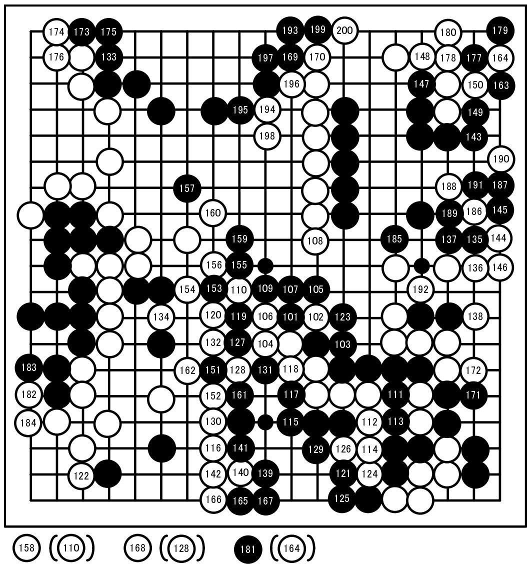


Fig. 6. Match with GnuGo3.6 II

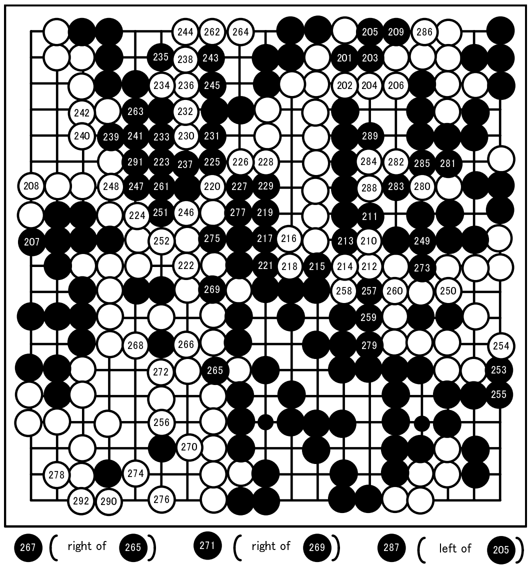


Fig. 7. Match with GnuGo3.6 III

REFERENCES

[1] "GoBase.org," accessed 25-October-2006. [Online]. Available: http://gobase.org/

[2] B. Bouzy and G. Chaslot, "Bayesian generation and integration of K-nearest-neighbor patterns for 19x19 go," *IEEE 2005 symposium on computational Intelligence in Games, Colchester, UK, G. Kendall & Simon Lucas (eds)*, pp. 176–181, 2005.

[3] E. van der Werf, J. Uiterwijk, E. Postma, and J. van den Herik, "Local move prediction in Go." *3rd International Conference on Computers and Games, Edmonton*, pp. 393–412, 2002.

[4] D. Stern, R. Herbrich, and T. Graepel, "Bayesian Pattern Ranking for Move Prediction in the Game of Go." 2005, Draft.

[5] D. Stern, R. Herbrich, and T. Graepel, "Bayesian Pattern Ranking for Move Prediction in the Game of Go." in *Proceedings of the 23rd International Conference on Machine Learning*, 2006, pp. 873–880.

[6] F. de Groot, "Moyo Go Studio," accessed 25-October-2006. [Online]. Available: http://www.moyogo.com/

[7] A. Zobrist, "A new hashing method with applications for game playing." *ICCA Journal*, no. 13(2), pp. 69–73, 1990.

[8] "Gnugo3.6," 2004, Free Software Foundation. [Online]. Available: http://www.gnu.org/software/gnugo/gnugo.html

[9] T. P. Minka, "A family of algorithms for approximate Bayesian inference." Ph.D. dissertation, Massachusetts Institute of Technology, 2001.

[10] J. Kazama and J. Tsujii, "Evaluation and Extension of Maximum Entropy Models with Inequality Constraints," in *Proceedings of the 2003 Conference on Empirical Methods in Natural Language Processing (EMNLP 2003)*, 2003, pp. 137–144.

[11] "A simple C++ library for maximum entropy classification," accessed 25-October-2006. [Online]. Available: http://www-tsujii.is.s.u-tokyo.ac.jp/%7Etsuruoka/maxent/

[12] T. Mark and J. Fairbairn, "GoGoD," accessed 25-October-2006. [Online]. Available: http://www.gogod.demon.co.uk/

[13] E. van der Werf, "AI techniques for the game of Go." Ph.D. dissertation, Universiteit Maastricht, 2004.

[14] A. L. Berger, S. A. D. Pietra, and V. J. D. Pietra, "A Maximum Entropy Approach to Natural Language Processing," *Computational Linguistics*, vol. 22 Issue1, pp. 39–71, 1996.