Learning to Predict Life and Death from Go Game Records

Erik C.D. van der Werf *, Mark H.M. Winands
H. Jaap van den Herik, Jos W.H.M. Uiterwijk

Department of Computer Science, Institute for Knowledge and Agent Technology, Universiteit Maastricht, P.O. Box 616, 6200 MD Maastricht, The Netherlands

Abstract

This article presents a new learning system for predicting life and death in the game of Go. It is called GONE. The system uses a multi-layer perceptron classifier which is trained on learning examples extracted from game records. Blocks of stones are represented by a large amount of features which enable a rather precise prediction of life and death. On average, GONE correctly predicts life and death for 88% of all the blocks that are relevant for scoring. Towards the end of a game the performance increases up to 99%. A straightforward extension for full-board evaluation is discussed. Experiments indicate that the predictor is an important component for building a strong full-board evaluation function.

Key words: Go, learning, game records, neural net, life and death

1 Introduction

Evaluating Go positions is one of the hardest tasks in Artificial Intelligence (AI). In the last decade the game of Go [1] has received significant attention from AI research [2,3]. Yet, despite all efforts, the best Go programs are still weak compared to human players. Partially this is due to the complexity of

* Corresponding author Tel.: (+31) (0)43-38 83491; fax: (+31) (0)43-38 84897
Email addresses: E.vanderWerf@cs.unimaas.nl (Erik C.D. van der Werf), m.winands@cs.unimaas.nl (Mark H.M. Winands), herik@cs.unimaas.nl (H. Jaap van den Herik), uiterwijk@cs.unimaas.nl (Jos W.H.M. Uiterwijk).

[1] For general information about the game including an introduction to the rules readers are advised to visit gobase.org [1].

19×19 Go. However, even on the 9×9 board, which has a complexity between Chess and Othello [2], the *current* Go programs perform nearly as bad. The main reason lies in the lack of adequate evaluation functions. Many (if not all) of the current top programs rely on (huge) static knowledge bases conceived by the programmers as they understand the Go skills and Go knowledge of a Go player. As a consequence the hand-coded knowledge in the top programs tends to become extremely complex and difficult to improve. In principle a learning system should be able to overcome this problem by reducing the dependence on hand-coded knowledge.

Over centuries humans have acquired extensive knowledge of Go. Since much of this knowledge is implicitly available in the games of human experts, it is tempting to apply machine-learning techniques to extract that knowledge from game records. One of the best sources of game records on the Internet is the NNGS archive [4]. Although the NNGS game records contain a wealth of information, the automated extraction of knowledge from these games is a non-trivial task at least for the following three reasons.

(1) *Missing Information.* Life-and-death status of blocks is not available. In scored games only a single numeric value representing the difference in points is available.
(2) *Unfinished Games.* Not all games are scored. Human games often end by one side resigning or abandoning the game without finishing it, which may leave the status of large parts of the board unclear.
(3) *Bad Moves.* During the game mistakes are made which may be hard to detect. Since mistakes break the chain(s) of optimal moves it can be misleading (and incorrect from a game-theoretical point of view) to relate positions before the mistake to the final outcome of the game.

Recently, we have set the first step towards making the knowledge in the game records accessible. We built a system that is able to score automatically 98.9% of the final positions correctly without any human intervention [5]. The reliable scores were obtained by a highly accurate classification of life and death for final positions ($\sim$99.7% of all blocks correct). As a result we now have a database containing 18,222 9×9 games with reliable and complete score information. From this database we intend to learn relevant Go skills which enable us to build a strong evaluation function.

In this article we present our latest work towards this purpose. It is a new learning system, called GONE (Go is Not Easy), for predicting life and death in the game of Go. Unlike in [5] where we only used final positions, this article focuses on predictions during the game. We believe that predicting life and death is a skill which is pivotal for strong play and an essential ingredient in any strong positional evaluation function.

The rest of this article is organised as follows: Section 2 presents the learning task in more detail. Section 3 introduces the representation and exhaustively lists all features. Section 4 provides information about the dataset. Sections 5, 6 and 7 report our experiments. Finally, Section 8 presents our conclusions.

2 The learning task

In order to learn to predict life and death we use a set of labelled game records, of which the labels contain the colour controlling each point at the end of the game. An intuitively straightforward implementation would classify each block [2] as the (majority of) occupied labelled points. Unfortunately this does not necessarily provide correct information for classification of life and death, for which at least two conflicting definitions exist.

The Japanese Go rules state: "Stones are said to be 'alive' if they cannot be captured by the opponent, or if capturing them would enable a new stone to be played that the opponent could not capture. Stones which are not alive are said to be 'dead'."

The Chinese Go rules state: "At the end of the game, stones which both players agree could inevitably be captured are dead. Stones that cannot be captured are alive."

(a) (b)

Fig. 1. Alive or dead?

A consequence of both rules is shown in Figure 1a: the marked black stones can be considered alive by the Japanese rules, and dead by the Chinese rules. Since the white stones are dead under all rule sets, and the whole region is controlled by Black, the choice whether these black stones are alive or dead is irrelevant for scoring the position. However, whether the marked black stones should be considered alive or dead in training is unclear.

A more problematic position, known as '3 points without capturing', is shown in Figure 1b. If this position is scored under the Japanese rules all marked stones are considered alive (because after capturing some new stones would

[2] Following the nomenclature as used by Müller [3] we call connected stones of the same colour a *block*. The reader should however note that other authors sometimes use the terms *string*, *unit*, *worm* or even *chain* for the same thing.

eventually be played that cannot be captured). However, if the position would be played out the most likely result (which may be different if one side can win a ko-fight) is that the empty point in the corner, the marked white stone, and the black stone marked with a triangle become black, and the three black stones marked with a square become white. Furthermore, all marked stones are captured and can therefore be considered dead under the Chinese rules.

In this article we choose the Chinese rules for defining life and death. The learning task therefore becomes the task of predicting whether blocks of stones can or will be captured. When replaying the game backward from the labelled final position the following four types of blocks can be identified (in order of decreasing domination):

(1) Blocks that are captured during the game.
(2) Blocks that occupy points ultimately controlled by the opponent.
(3) Blocks that occupy points on the edge of regions ultimately controlled by their own colour.
(4) Blocks that occupy points in the interior of regions ultimately controlled by their own colour.

Blocks of type 1 and 2 should be classified as dead. Blocks of type 3 should be classified as alive. Type-4 blocks cannot be classified based on the labelling and are therefore not used in training. (As an example, the marked block in Figure 1a typically ends up as type 4, and the marked blocks in Figure 1b end up as type 2. However, if any of the marked blocks are actually captured during the game they will of course be of type 1.)

Obviously, perfect classification is not possible in non-final positions. Therefore the goal is to approximate the Bayesian *a posteriori* probability given a set of features or at least the Bayesian discriminant function, for deciding whether the block will be alive or dead at the end of the game. In pattern recognition there are several ways to do this. A popular choice is the use of a multi-layer perceptron (MLP) classifier. It has been shown [6] that minimising the mean-square error (MSE) on binary targets, for an MLP with sufficient functional capacity, adequately approximates the Bayesian *a posteriori* probability.

3 Representation of the block

Many representations for characterising blocks are possible and used in the computer-Go domain. The most primitive representations typically employ the raw board directly. Although such representations are complete, in the sense of containing all relevant information, they are known to be inefficient because of their high dimensionality and lack of topological structure. Our

representation employs a carefully selected set of features based on simple measurable geometric properties, some elementary Go knowledge, and some hand-crafted specialised features. Many of our features are typically used in Go programs to evaluate positions [7,8]. The features are calculated for single blocks (friendly and opponent), multiple blocks in chains, and colour-enclosed regions (CERs).

For each block we include the following features:

- *Size* measured in occupied points,
- *Perimeter* measured in number of adjacent points, including points over the edge,
- *Opponents*: the occupied adjacent points,
- (*First-order*) *liberties*: the free adjacent points,
- *Second-order liberties*: the liberties of (first-order) liberties (excluding the first-order liberties),
- *Third-order liberties*: the liberties of second-order liberties (excluding first- and second-order liberties),
- *Protected liberties*: the liberties which cannot be played by the opponent, because of suicide or being directly capturable,
- *Auto-atari liberties*: the liberties which reduce the liberties of the block from 2 to 1 if they are played on; it means that the blocks would become directly capturable (such liberties are protected for an adjacent opponent block),
- *Adjacent opponent blocks*,
- *Local majority*: the number of opponent stones minus the number of friendly stones within a Manhattan distance of 2 from the block,
- *Centre of mass* represented by the distance to the closest and second-closest edge,
- *Bounding box size*: the number of points in the smallest rectangular box that can contain the block.

Adjacent to each block are CERs consisting of connected empty and occupied points, surrounded by stones of one colour or the edge. It is important to know whether an adjacent CER is fully accessible, because a fully accessible CER surrounded by safe blocks provides at least one sure liberty. To detect fully accessible regions we use so-called miai strategies as applied by Müller [9]. In contrast to Müller's original implementation we also add miai accessible interior empty points to the set of accessible liberties, and use protected liberties for the chaining too.

An example of a fully accessible CER is shown in Figure 2. Here the idea is that if White plays on a marked empty point, Black replies on the other empty point marked by the same letter. By following this miai strategy Black is guaranteed to be able to occupy or become adjacent to all points in the region. Often it is not possible to find a miai strategy for the full region, in

which case we call the CER partially accessible. In Figure 3 an example of a partially accessible CER is shown. In this case the 3 points marked x form the inaccessible interior for the given miai strategy.

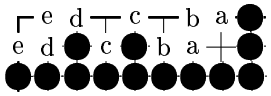


Fig. 2. Fully accessible CER.

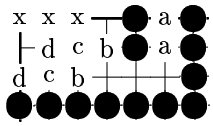


Fig. 3. Partially accessible CER.

For fully accessible CERs we include:

- *Number of regions*,
- *Size*,
- *Perimeter*,
- *Split points*: crucial points for preserving connectedness in the local 3×3 window around the point. (The region could still be connected by a big loop outside the local 3×3 window.) Examples are shown in Figure 4.

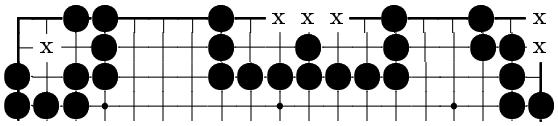


Fig. 4. Split points marked with x.

For partially accessible CERs we include:

- *Number of partially accessible regions*,
- *Accessible size*,
- *Accessible perimeter*,
- *Size* of the inaccessible interior,
- *Perimeter* of the inaccessible interior,
- *Split points* of the inaccessible interior.

The size, perimeter, and number of split points are summed for all regions. We do not address individual regions because the representation must have a fixed length, whereas the number of regions is not fixed. (Although one might think that summing over all regions harms performance, the alternative of treating a large number of regions separately actually tends to do more harm than good because of the increased dimensionality.)

Another way to analyse CERs is to look for possible eyespace. Points forming the eyespace should be empty or contain capturable opponent stones. Empty points directly adjacent to opponent stones are not part of the eyespace. Points on the edge with one or more diagonally adjacent alive opponent stones and points with two or more diagonally adjacent alive opponent stones are false eyes (opponent blocks which are directly adjacent to enough empty space for at most one eye are here assumed dead, and do not make eyes false). False

eyes are not part of the eyespace (we ignore the unlikely case where a big loop upgrades false eyes to true eyes). For directly adjacent eyespace of the block we include:

- *Size*,
- *Perimeter*.

In many positions, blocks of the same colour with shared liberties will be connected at the end of the game. An optimistic scenario therefore may assume that all blocks with shared liberties can form a chain. Examples of a black and a white optimistic chain are shown in Figure 5. For this so-called optimistic chain we include:

- *Number of blocks*,
- *Size*,
- *Perimeter*,
- *Split points*,
- *Adjacent CERs*,
- *Adjacent CERs with eyespace*,
- *Adjacent CERs, fully accessible* from at least one block,
- *Size of adjacent eyespace*,
- *Perimeter of adjacent eyespace*,
- *External opponent liberties*: liberties of adjacent opponent blocks which are not accessible from the optimistic chain.

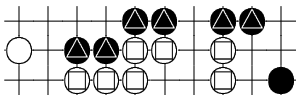


Fig. 5. Marked optimistic chains.

The reader may have noticed that our optimistic chain has some similarity with what human players call a group. Our current representation does not use groups. The reason is that the human notion of a group is not well defined. It typically relies on virtual connections between blocks which are more than one empty space apart (such connections can be non-transitive), and may even rely on an underlying notion of life and death. However, if heuristic group information is available in a program it could of course be used trivially with similar features as used for our optimistic chain.

Adjacent to the block under investigation there may be opponent blocks. For the weakest (measured by the number of liberties) directly adjacent opponent block we include:

- *Perimeter*,
- *Liberties*,
- *Shared liberties*,
- *Split points*,

– *Perimeter of adjacent eyespace.*

The same features are also included for the second-weakest directly adjacent opponent block, and the weakest opponent block directly adjacent to or sharing liberties with the optimistic chain of the block in question.

Regions of connected empty intersections adjacent to black and to white stones are disputed territory. (We ignore the fact that some blocks may already be considered dead for one side, which could resolve some disputes.) If the block is adjacent to disputed territory we include:

– *Direct liberties* of the block in disputed territory,
– *Liberties of all friendly blocks* in disputed territory,
– *Liberties of all enemy blocks* in disputed territory.

Finally, we include the following global features:

– *Player to move* relative to the block's colour,
– *Ko* indicates if an active ko is on the board,
– *Distance to ko* from the block,
– *Number of friendly stones* on the board,
– *Number of opponent stones* on the board.

4 The data set

In all experiments presented in this article we used 9×9 game records played between 1995 and 2002 on NNGS [4]. For the experiments reported in Sections 5 and 6 we used training and test examples obtained from 18,222 9×9 games that were played to the end and scored. In total, all positions from these games contain about 10 million blocks of which 8.5% are of type 1, 11.5% are of type 2, 65.5% are of type 3, and 14.5% are of type 4. Leaving out type-4 blocks gives as *a priori* probabilities that 76.5% of the blocks are alive and 23.5% of the blocks are dead.

Since NNGS game records only contain a single numeric value for the score, the fate of all points had to be labelled. For this we used GnuGo [10], some manual labelling, and our own system for scoring final positions [5]. Since automatic labelling is still imperfect, all games where the score based on the labelling was not identical to the numeric score in the game record, or where the final positions contained unsettled interior points, were inspected manually. Finally, an additional inspection of a few hundred randomly selected final positions revealed none that were labelled incorrectly.

In all experiments the test examples were extracted from games played in

1995, and the training and the validation examples from games played between 1996 and 2002. Since the games provide a huge amount of blocks with little or no variation (large regions remain unchanged per move) and because of constraints on time and working memory only a small fraction of blocks was randomly selected for training (<5% per game).

5 Choosing a classifier

An important choice is selecting a good classifier. In pattern recognition there is a variety of classifiers to choose from. A popular choice is the use of artificial neural networks, which have already been applied to other Go-related tasks at least with some degree of success [11–13]. Our previous work on scoring final positions showed that the Multi-Layer Perceptron (MLP) provides good performance with a reasonable training time [5]. The performance of the MLP mainly depends on the architecture, the number of training examples, and the training algorithm. In the experiments reported below we tested architectures with 1 and with 2 hidden layers containing various numbers of neurons per hidden layer. For training we compared: (1) gradient descent with momentum and adaptive learning (GDXNC) with (2) RPROP backpropagation (RPNC). For comparison we also present results for the Nearest Mean Classifier (NMC), the Linear Discriminant Classifier (LDC), and the Logistic Linear Classifier (LOGLC). More information on these classifiers can be found in [14–16].

Since the performance of the classifiers may be influenced by random initialisations each classifier was trained 10 times with respectively 1,000, 5,000, and 25,000 training examples. A validation set of equal size or at most 15,000 examples was used to stop training. In Table 1 the average performance of the various classifiers is shown on a test set of 22,632 blocks extracted from 920 games played in 1995 (to speed up this experiment we used only $\sim 5\%$ of all blocks available from the 1995 games). It is noted that the standard deviations over the 10 runs were around 0.1%. The results indicate that GDXNC performed slightly better than RPNC. Although RPNC trains 2 to 3 times faster than GDXNC, and converges at a lower error on the training data, the performance on the test data seems to be worse because of overfitting. For both GDXNC and RPNC it appeared that using one hidden layer with 25 neurons is sufficient at least for training with 25,000 examples. Adding a second hidden layer with 5 or 25 neurons did not improve performance.

Table 1
Performance of classifiers on a test set. The numbers in the names indicate the number of neurons per hidden layer.

Classifier	Test error (%)		
Training examples	1,000	5,000	25,000
NMC	21.5	21.0	21.0
LDC	14.2	13.7	13.6
LOGLC	14.8	13.3	13.1
GDXNC-5	13.8	12.9	12.2
GDXNC-15	13.9	12.9	12.2
GDXNC-25	13.7	12.8	12.0
GDXNC-25-5	13.8	12.9	12.1
GDXNC-25-25	13.9	12.8	12.0
GDXNC-50	13.8	12.8	12.0
RPNC-5	14.7	13.4	12.4
RPNC-15	14.4	13.2	12.4
RPNC-25	14.7	13.3	12.4
RPNC-25-5	14.3	13.4	12.6
RPNC-25-25	14.3	13.5	12.8
RPNC-50	15.0	13.3	12.5

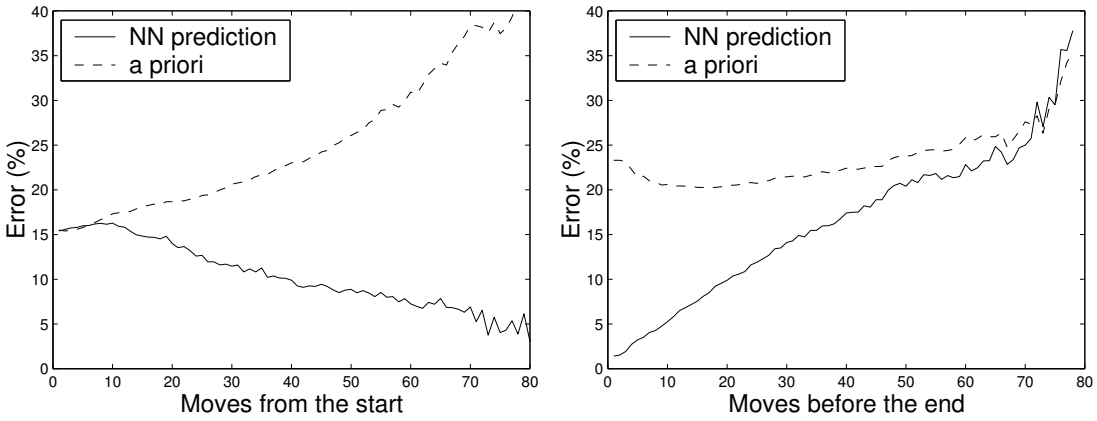
6 Performance over the game

In the previous section we calculated the average classification performance over whole games. Although this is an interesting measure, it does not tell us how the performance changes as the game develops. We believe that for standard opening moves the best choice is pure guessing based on the highest *a priori* probability (always alive). Final positions, however, can (at least in principle) be classified perfectly. Given these extremes it is interesting to see how the performance changes over the game, either looking forward from the start position or backward from the final position.

To test the performance over the game we trained a new GDXNC classifier using one hidden layer with 25 neurons on 175,000 training examples. A validation set of 25,000 examples was used to stop training. Training was performed 3 times with different random initialisations after which the best classifier was selected based on the performance on the validation set. This classifier achieved a prediction error of 11.7% on the complete test set (containing 443,819 blocks

from 920 games).

In Figure 6a it is shown that pure guessing performs equally well for roughly the first 10 moves. As the length of games increases the *a priori* probability of blocks on the board ultimately being captured also increases (which makes sense because the best points are occupied first and there is only limited space on the board). It should be noted that although the plots extend only up to 80 moves this does not mean that there were no longer games. However, the number of games with a length of over 80 moves is too low for meaningful results (too much noise).



(a) (b)

Fig. 6. Performance over the game.

Good evaluation functions typically aim at predicting the final result at the end of the game as soon as possible. It is therefore encouraging to see in Figure 6b that towards the end of the game the error goes down rapidly, predicting about 95% correctly when 10 moves before the end. For final positions our system classifies over 99% of all blocks correctly. Our previous work [5] showed that this performance is at least comparable to that of the average rated NNGS player (for scored 9×9 games such a player has a rating of 7 kyu). Furthermore, scaling up to 19×19 seems to be possible without significant loss in performance. Whether this performance is similar for non-final positions is difficult to say.

7 Towards a full-board evaluation function

In Go, full-board evaluation functions typically aim at predicting the number of intersections controlled by each player at the end of the game. By predicting life and death for all occupied intersections GONE provides the basis for such

a full-board evaluation function. A straightforward extension [3] of GONE to classify all intersections is implemented by assigning each intersection to the colour of the nearest living block. An example is presented in Figure 7. Here the left board shows the predictions of GONE, the middle board shows all blocks which are assumed to be alive, and the right board shows the territory which is calculated by assigning each intersection to the colour of the nearest living block. (Notice that even though our system failed to detect one white dead block the estimated territory is still sufficient to predict the correct winner.)

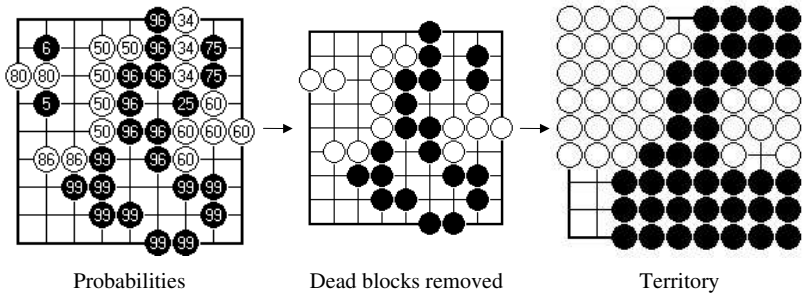


Fig. 7. An example of a full-board evaluation.

For final positions (where both sides have completely sealed off their territory) our straightforward extension suffices to predict the correct winner with over 99% certainty [5]. However, since we are interested in predictions during the game, it is probably more informative to test the performance on non-final positions. Since players make mistakes during the game, the most reliable non-final test positions which can be extracted from game records occur when one player resigns. We tested our straightforward extension of GONE on 2,786 resigned 9×9 games played between 1995 and 2002 by rated players on NNGS [4]. On average it predicts the correct winner for 87% of all positions. For comparison, if GONE is not used to remove dead blocks, and all empty points are assigned to the colour of the nearest stone, the performance drops to 69% correct.

The strength of players is a factor influencing the difficulty of positions and the reliability of the results. Therefore, we calculated statistics for all rank categories between 20 kyu and 2 dan. Figure 8 shows the relation between the rank of the player that resigned and the average error at predicting the winner, the average estimated difference in points, as well as the number of game records available. It is shown that predicting the winner tends to become more difficult with increasing strength of the players. This makes sense because strong players usually resign earlier and tend to create more difficult positions. It is also no surprise that the estimated difference in points (when one player resigns) tends to decrease with playing strength.

[3] More knowledgeable approaches to extend GONE are possible. These are, however, beyond the scope of this article.

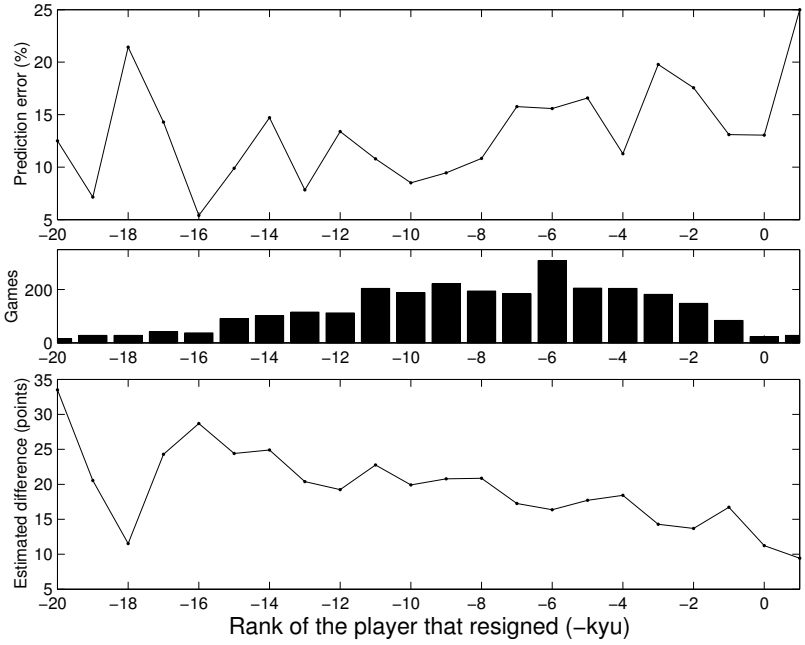


Fig. 8. Predicting the outcome of resigned games.

8 Conclusions and future research

Our system GONE learned to predict life and death from labelled examples quite accurately. On unseen game records and averaged over the whole game, it classified around 88% of all blocks correctly. Ten moves before the end of the game it classified around 95% correctly, and for final positions it classified over 99% correctly.

Moreover, we presented a straightforward extension of GONE towards a full-board evaluation function which gave quite promising results. To obtain more insight into the importance of this work, GONE should be incorporated into a more advanced full-board evaluation function. Testing this full-board evaluation function on non-final positions, resigned games, and in a full go-playing engine will be an important next step.

Although training with more examples still has some impact on performance, we believe that most can be gained by improving the representation of blocks. Some features, such as those for loosely defined groups, have not yet been characterised and implemented, whereas some other features may be correlated or could even be redundant. Most likely, automatic feature extraction and feature selection methods have to be employed to improve the representation.

Acknowledgements

This work was funded, in part, by the Netherlands Organisation for Scientific Research (NWO), dossier number 612.052.003. We gratefully acknowledge financial support by the Universiteitsfonds Limburg / SWOL.

References

[1] J. van der Steen, Gobase.org - Go games, Go information and Go study tools (2003).
URL `http://gobase.org/`

[2] B. Bouzy, T. Cazenave, Computer Go: An AI oriented survey, Artificial Intelligence 132 (1) (2001) 39–102.

[3] M. Müller, Computer Go, Artificial Intelligence 134 (1-2) (2002) 145–179.

[4] NNGS, The no name Go server game archive (2002).
URL `http://nngs.cosmic.org/gamesearch.html`

[5] E. C. D. van der Werf, H. J. van den Herik, J. W. H. M. Uiterwijk, Learning to score final positions in the game of Go, in: H. J. van den Herik, H. Iida, E. Heinz (Eds.), Advances in Computer Games: Many Games, Many Challenges, Kluwer Academic Publishers, Boston, 2003, pp. 143–158.

[6] J. B. Hampshire II, B. A. Pearlmutter, Equivalence proofs for multilayer perceptron classifiers and the Bayesian discriminant function, in: D. Touretzky, J. Elman, T. Sejnowski, G. Hinton (Eds.), Proceedings of the 1990 Connectionist Models Summer School, Morgan Kaufmann, San Mateo, CA, 1990, pp. 159–172.

[7] K. Chen, Z. Chen, Static analysis of life and death in the game of Go, Information Sciences 121 (1999) 113–134.

[8] D. Fotland, Static eye analysis in 'The Many Faces of Go', ICGA Journal 25 (4) (2002) 201–210.

[9] M. Müller, Playing it safe: Recognizing secure territories in computer Go by using static rules and search, in: H. Matsubara (Ed.), Proceedings of the Game Programming Workshop in Japan '97, Computer Shogi Association, Tokyo, Japan, 1997, pp. 80–86.

[10] GnuGo (2003).
URL `http://www.gnu.org/software/gnugo/`

[11] F. Dahl, Honte, a go-playing program using neural nets, in: J. Fürnkranz, M. Kubat (Eds.), Machines that Learn to Play Games, Nova Science Publishers, Huntington, NY, 2001, Ch. 10, pp. 205–223.

[12] M. Enzenberger, Evaluation in Go by a neural network using soft segmentation, in: H. J. van den Herik, H. Iida, E. Heinz (Eds.), Advances in Computer Games: Many Games, Many Challenges, Kluwer Academic Publishers, Boston, 2003, pp. 97–108.

[13] N. Schraudolph, P. Dayan, T. Sejnowski, Temporal difference learning of position evaluation in the game of Go, in: J. D. Cowan, G. Tesauro, J. Alspector (Eds.), Advances in Neural Information Processing 6, Morgan Kaufmann, San Francisco, 1994, pp. 817–824.
URL `ftp://ftp.idsia.ch/pub/nic/nips93.ps.gz`

[14] M. Riedmiller, H. Braun, A direct adaptive method for faster backpropagation: the RPROP algorithm, in: H. Rusini (Ed.), Proceedings of the IEEE Int. Conf. on Neural Networks (ICNN), 1993, pp. 586–591.

[15] R. P. W. Duin, PRTools, a Matlab Toolbox for Pattern Recognition (2000).
URL `http://www.ph.tn.tudelft.nl/prtools/`

[16] A. K. Jain, R. P. W. Duin, J. Mao, Statistical pattern recognition: A review, IEEE Transactions on Pattern Analysis and Machine Intelligence 22 (1) (2000) 4–37.