Honte, a Go-Playing Program Using Neural Nets

Fredrik A. Dahl
Norwegian Defence Research Establishment
Division for Systems Analysis
P.O. Box 25
NO-2027 Kjeller
Norway
Fredrik-A.Dahl@ffi.no

Abstract

The go-playing program Honte is described. It uses neural nets together with more conventional AI-methods like alpha-beta search. A neural net is trained by supervised learning to imitate local shapes made in a database of expert games. A second net is trained to estimate the safety of groups by self play using TD(λ)-learning. A third net is trained to estimate territorial potential of unoccupied points, also based on self play and TD(λ)-learning. Although the program has not yet reached the level of the best commercial go-programs, results are encouraging.

1 INTRODUCTION

This article describes the go-playing program Honte. The name "Honte" (pronounced hon-teh) means "proper", "sound" or "correct" in Japanese. The idea behind Honte is to use neural nets together with other programming techniques, hopefully getting the best of all worlds. Not all sections of the article are relevant to machine learning as such, but for a problem as complex as go, integration of different techniques is vital. We therefore strive towards a balanced presentation of the different methods used.

1.1 RULES

We first give a short summary of the most important rules of go. For a complete discussion of go rules, see [1].

The *board* consists of a square graph, typically a 19 times 19 grid. The game starts with all *points* (nodes of the graph) open. Black and white take turns in placing stones at points that are open. Stones of identical color connect along edges, and a set of connected stones is called a *string*. Neighboring open points to a string are *liberties* for the string. If placing a stone removes the last liberty from any strings of opposing color, these are *captured*, and removed from the board. Captured stones are denoted *prisoners*. Placing a stone that does not make a capture, and has no liberties, is illegal. Placing a stone that results in repeating the position seen after the same player's previous play, is also illegal. This is called the *ko-rule*. A player may pass his turn. Stones that cannot be captured by alternating play are *alive*. Stones that can be captured by alternating play, even if the defender plays first, are *dead*. (The use of the terms "dead" and "prisoner" may sound odd, as people normally die *after* being prisoners, but with stones in go it is the other way around.)

Areas that are surrounded by strings that are either black and alive or white and dead, are *territory* points for black, and vice versa. When all strings are either dead or alive, and all open points are territory for one side or the other, the players pass, and the game ends. There are some relatively rare exceptions, however, of stalemate shapes that produce open points that neither player can turn into territory, called *seki*. The winner is the side that controls the bigger part of the board. As black begins the game, white receives a compensation called *komi*. The komi is often set at 5.5 territory points, which is considered fair and prevents draws.

Under the Chinese version of the rules, the area controlled is defined as the union of the player's live strings, the opponent's dead strings and his own open territory points. Under the Japanese rules, live stones do not count as territory, but instead prisoners and opposing dead stones count one point each, and the area under opposing dead strings count as territory. The rules are virtually equivalent, as both sides play almost the same number of stones throughout the game, so that awarding a point for surviving is roughly equivalent to awarding a point for killing. Figure 1 gives an example of a live black string in the lower right corner.

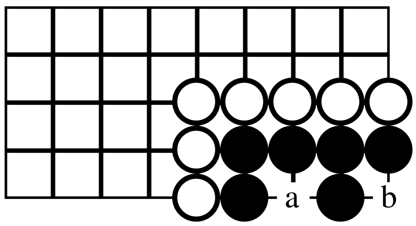

Figure 1: Black Is Alive

The black string has two liberties, marked **a** and **b**. To capture the string, white must play at both of these points. But this is impossible, as a white stone at **a** or **b** has no liberties, so playing there is illegal. The liberties **a** and **b** are commonly referred to as *eyes*. The fact that a string lives if it has two eyes is sometimes referred to as a rule, but it is in fact derived from the rules. If black is suicidal, and plays **a**, his stones die. White will then be allowed to play at **b**, as that stone produces liberties for itself by capturing the black string. This is not urgent, however, as the black stones are going nowhere. Normally white would never play **b**, but at the end of the game simply point out that he can, and remove the black stones as prisoners.

1.2 STRENGTH OF PROGRAMS

When people develop programs that play games, they will always want to evaluate how *strong* their programs are. We tend to feel that there is a well defined concept of strength, by which our programs should be measured. We also naturally demand that if program A is stronger than program B, then program A should win against program B, at least more than half the time. Unfortunately, this is demanding too much from a strength measure, because game outcomes are *not in general transitive*: If program A beats program B, and program B beats program C, it may well be that program C beats program A. Our notion of a programs strength can simply not be defined consistently. Game theory [2] defines the solution concept for two-player zero-sum games, but does not provide evaluation criteria for sub-optimal play. For a general discussion of evaluation criteria of agents playing two-player zero-sum games, see Halck & Dahl [3].

Currently the program Handtalk [1] is considered the strongest, but as we have pointed out, measuring, and even defining strength is far from easy. Handtalk has received a 4-kyu diploma from the Japanese go federation, implying that it is comparable to a fairly strong amateur. This evaluation is based on performance against a human that was allowed no practice game, and had no prior knowledge of strengths and weaknesses of go programs in general. However, a professional player challenged and beat Handtalk in a 25-stone handicap [4], which means that the program started the game with 25 stones on the board placed in a predefined pattern. A handicap this large appears utterly ridiculous to a human player, and probably no human 4-kyu player would lose such a game. The professional had obviously experimented with Handtalk before the game, and knew exactly where the weaknesses were to be found. One could argue back and forth as to whether this means that Handtalk "is not really of 4-kyu strength", but because no consistent strength measure exists, that discussion leads nowhere.

However, even a 4-kyu level is not very impressive, and Mechner [4] gives a good analysis of why go programming is hard. Go belongs to the same game class as chess, being a deterministic game with full information. Therefore it is fair to ask why massive lookahead analysis using minimax alpha-beta search (see e.g. Russell & Norvig [5]), which works so well for chess, does not work well for go. An obvious point in comparing go with chess is that the game tree of go branches much faster, but as [4] points out, the problem with defining a fast algorithm for evaluating full board positions in a meaningful way is much more important. Even deep lookahead is useless without sensible evaluations of the leaf nodes of the search. A different explanation to why go is difficult comes from complexity theory. It has been proved that go belongs to the class of P-space complete problems [6]. P-space is the set of problems that can be solved by algorithms with memory requirement at most polynomial as a function of problem size. Being P-space *complete* means that go belongs to the class of the most memory-demanding problems in P-space. P-space complete problems also have the property that all known algorithms are exponential in terms of computation time.

Developers of the most successful commercial programs naturally keep some secrets about their algorithms. But from what has been published [1], the most important techniques appear to include pattern matching, alpha-beta search used for local problems, propagation of influence fields and heuristics for evaluating safety of stones and territories. Some experiments have been made with neural nets using TD-learning as defined by Sutton [7]. This has not been very successful when compared to performance of the most popular commercial go programs, [8] and [9].

2 GENERAL APPROACH IN HONTE

An objective in developing Honte has been to test neural nets for as many aspects of the game as seems reasonable, and use "more classical" artificial intelligence techniques for the others. We have tried to model from how humans think. One could argue that this is like constructing a mechanical bird, when the best "algorithm" for flying may be building a jet engine. This may well turn out to be correct, but for the time being, imitating the cognitive processes performed by humans seems the most fruitful approach.

For humans some sub-problems considered when deciding a play involve local alpha-beta search, such as life-and-death problems and calculation of connectivity. Others, like evaluating territorial potential in an area, or estimating the quality of a shape of stones, is more of a "judgment" process by humans. We believe that neural nets should primarily be used for problems that humans use judgment to solve.

We now give a general overview of algorithms and modules used, which will be explained separately later in the article. Honte uses a full-board, or *global*, evaluation function for comparing plays. It performs the following steps:

- Calculating which strings can be captured, using alpha-beta search
- Calculating which strings can connect, using alpha-beta search
- Finding the set of groups according to connections, using graph calculations
- Calculation of life and death status of encircled groups, using alpha-beta search
- Initial calculation of influence, using graph propagation
- Evaluation of group safety, using neural nets
- Modified calculation of influence, using graph propagation
- Evaluation of territories, using neural nets
- Summing territory estimates and group safety values

The evaluation procedure is time consuming, and a full evaluation can not be performed on too many plays. Candidate plays are identified by a neural net trained to identify good shapes. A module containing a set of standard sequences, called a *joseki library*, also produces candidate plays. Besides screening out candidate plays, both the shape evaluating net and the joseki module contribute directly to the output of the global evaluation function.

The neural nets used in Honte belong to the relatively simple and well understood class of fully connected feed-forward nets with sigmoid activation functions and one layer of hidden nodes. Training uses so-called "back propagation of errors", which implements the chain rule for calculating gradients of the output with respect to weights in the net. For a general introduction to neural nets, we refer to [10].

3 JOSEKI LIBRARY

A Joseki library defines local patterns of play, typically close to the corner, that are established as good play. In this respect it is similar to opening theory in chess, except that chess openings address the whole board. Ideally, a joseki library should include information about whole-board conditions, but this is not currently handled in Honte. On the other hand, the library's recommendations are not absolute, and the general play choosing algorithm may override them.

The library is implemented as a hashed table of positions and plays, so that transpositions can be handled. The library also includes information about urgency of different plays. Also it contains information about plays that Honte should not itself make, so that correct responses to unsound trap-plays can be encoded. Joseki data is given in the form of a game tree, and the program identifies transpositions and symmetries automatically.

4 SHAPE EVALUATING NEURAL NET

Based on a set of expert games, a neural net has been trained to estimate the likelihood that a given play would be made by an expert in a local context. The input represents a local situation close to an open point, and the network's output is interpreted as the quality of playing at that point. The shape-quality of a play is a very important concept to human players, and often experts make plays almost entirely for the purpose of "making good shape". An expert can often immediately criticize a play for "making bad shape", even though it may take much more effort to identify how the opponent can utilize it. Our hope is that a net trained from professional play will capture some of this almost mysterious concept of shape.

The input to the net is relative to an open point on the board. A *receptive field* is defined as an approximately circular area with diameter seven, centered at the point of reference. The receptive field is reflected and rotated so that the edge of the board closest to the field is below it, and the second closest edge is to the left. This is a way of choosing a canonical member of the eight-element symmetry group of the board. To eliminate the symmetry of black and white pieces, the neural net input encodes the stones of the side on turn as black. Figure 2 shows an example of a receptive field. The net should give a high shape value in this case, as playing at "?" is a very nice move, making it impossible for white to keep black's stones separated.

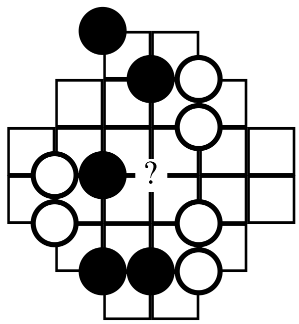

Figure 2: A Receptive Field

Each point in the receptive field is associated with two binary input fields for the net (B,W). The B input is 1 if the point has a black stone, and 0 otherwise. The W input signals a white stone in the same way. For most shapes, the distance to the edges are important. The distance to the closest edge is encoded using 9 binary input fields. If the distance to the edge is i, the first $(9 - i)$ input values are 1, and the last i are 0. The distance to the second closest edge is encoded in a similar way.

In many situations the shape value of a play is closely linked to its effect on liberties. Therefore features related to this are encoded. These features are encoded as real variables scaled so that they tend to give values between 0 and 1. Examples of these features are number of captured opponent stones and number of liberties for the stone played. The net uses a total of 190 input fields and 50 hidden nodes.

The network is trained from a set of 400 games played by human experts. Training patterns are generated from the sample games in this way:

- A random position from a random game is drawn.
- The actual play made from that position gives a training pattern with input produced from the point played, and desired output as 1.
- A random point, where playing is legal, is then drawn, and used as input pattern associated with desired output of 0.

The strategy of choosing the point for negative feedback at random may sound sub-optimal. Firstly, it might generate too many completely pointless plays that offer the net little relevant information. Secondly, the randomly chosen points will sometimes produce negative feedback for plays that do make good shape. One might consider more sophisticated strategies for constructing negative feedback to the net, such as over-representing plays that “almost” make good shape. This may work better than the random strategy chosen, but has the serious downside that it needs information not present in the logs of expert games.

To evaluate the abilities of the net, one would ideally measure the average strength of the plays preferred by the net. But this too would require information not present in game logs, and for a human to manually evaluate the score of a large number of computer plays would require an enormous effort. Instead we do the opposite, using the net to rank the plays made by the experts in the sample games: For each position, all legal plays are evaluated by the net, and the fraction of the plays that are evaluated as weaker than the expert’s play, is defined as the score. The average score over all positions in the sample games gives the total score of the net. If the net were to agree completely with all plays made by the experts, the score would be 1.0, and if there were no correlation between the neural net’s output and the plays made, the score would be 0.5.

After training the net scores as high as 0.94 on the training set. On a set of games not used in training it received a score of 0.92. Figure 3 gives the score as a function of training.

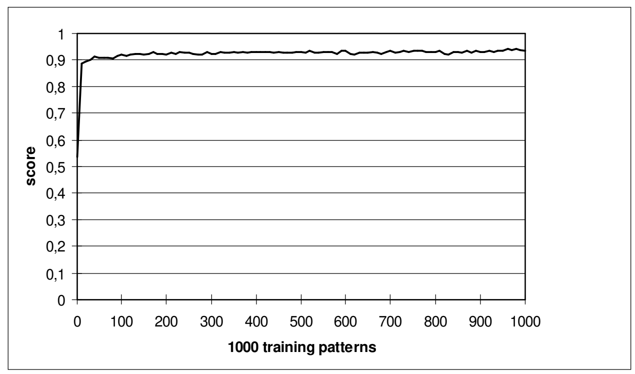

Figure 3: Learning Curve for the Shape-Evaluating Net

One should probably not expect the shape evaluator to score much higher than this, because not all plays made by professionals make good shape. Some "ugly" moves have to be made for tactical reasons. Figure 4 gives an example of this near the lower right hand corner.

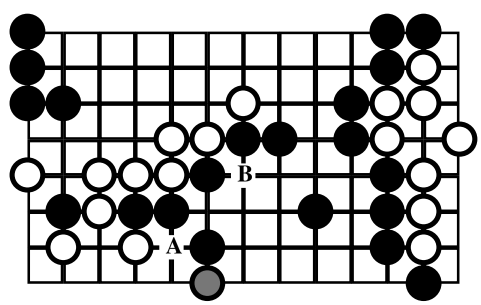

Figure 4: An Expert Move with “Bad Shape”

The expert made the grey play as black, and the neural net evaluates this as extremely bad shape (0.001 on the scale from 0 to 1). The play really does look odd, at least to author, being submissive for no obvious reason, and allowing a white forcing play at **A**, reducing his territory further. The play is good, though, as it indirectly protects against a future cut at **B**. Still, the play can be said to make bad shape, and we would not expect (nor want) our neural net to like it.

The shape evaluating net has proved surprisingly reliable and very useful. In most positions Honte discards about 90 % of the legal plays immediately due to low shape value.

Stoutamire [11] has used a similar learning paradigm of extracting good shapes from a database of expert games, using hashed sets of patterns rather than neural nets.

5 ALPHA-BETA SEARCH

As mentioned previously, alpha-beta search based on evaluating full-board positions is impractical. However, alpha-beta search is important for solving local well defined problems. Typical local problems that can be handled by alpha-beta search are:

Capturing of checkers: Can a given string of stones be captured within the next few plays?

Connections: Can two given strings connect?

Life and death: Can a given group of strings create two eyes in a local area?

In Honte a generic alpha-beta search engine is implemented, which gives one of three results: "Yes", "No" and "Unknown". Limiting the outcome in this way increases efficiency of the alpha-beta pruning. A search problem is defined by a procedure determining if the objective is reached, open or unreachable, procedures for generating candidate plays, and search depth. Currently the objective is either to capture a string, connect two strings or make life, or preventing the opponent from one of these. The search uses quiescence and iterated deepening [5], and the search depth parameter gives the maximum depth allowed.

If the search is deep, the search space can be reduced significantly by identifying positions that have already been analyzed, converting the game tree to a game DAG (Directed Acyclic Graph). This is done in Honte by storing positions visited by the search, and their alpha and beta values, in a hashed data structure. Storing positions visited also has a different purpose than reducing the search space. After the search is completed, a proof DAG is constructed. This is a subset of the DAG visited in the search, that proves how the player achieves his best result in the search. The converse is also produced, i.e. a DAG proving that the opponent can keep the first player from getting better results than the calculated value. As an example, assume the problem is for black to capture a white string, and the search gives "Unknown". Black's proof DAG will show that white cannot escape to safety within the search depth, and white's proof DAG shows that black cannot capture the string within the search depth.

To make a go program that uses alpha-beta search run efficiently, it is important to be able to perform calculations incrementally. Suppose a given pair of strings are barely able to connect, and the search needed to determine this is significant. If no information from the search is stored, it would have to be recalculated over and over, not only for each play of the game, but even when analyzing different candidate plays. A possible way to handle this is to store only the set of points visited in the search, and recalculate when this area is touched. In Honte we use a more ambitious scheme, that stores the proof DAGs together with the objects that the calculations are made for. Verifying if a given proof DAG is still valid is orders of magnitude faster than performing the actual search. Also proof DAGs associated with hypothetical objects are stored. If a candidate play that is analyzed produces a new string, and possible connections between this and other strings are calculated, these connection objects with proof DAGs are stored for later use.

The shape-evaluating net is used within the alpha-beta search for identifying candidate plays. This is done not primarily to speed up the search, as the time spent on evaluating the neural net for each potential candidate play largely compensates for the reduction of the search space. The main purpose is to modify the outcome of the search. An example of this is given in figure 5, showing the lower side of a board.

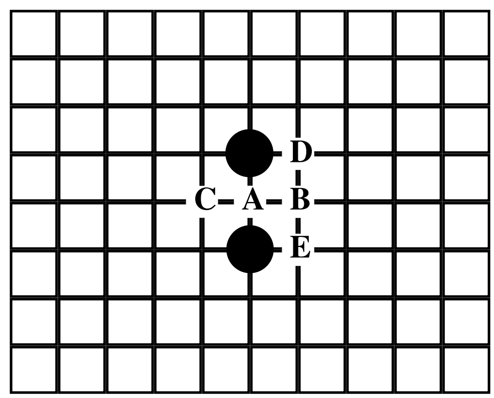

Figure 5: A Search Problem

The search problem is to decide if the two black stones can be cut apart. White must start with **A**, of course, otherwise black plays there and connects. Then black can play **B**, threatening to connect by capturing, and white must defend by playing **C**. Black can then defend the cut at **D** or **E**, and white must play the other, after which the search must decide if that white cutting stone can be captured. In the present position, it seems that white may be able to escape, and the search needed to establish this is significant. But although white may technically be able to cut black apart, it can hardly be correct. White would end up with weak groups to defend, or if white is strong in the surrounding areas, black might even sacrifice the lower stone in an advantageous way. The alpha-beta search analyzes as if connecting those two stones were the only goal in the game, disregarding side effects of pursuing that goal. From a human point of view the entire search is ludicrous, as a white play at **A** is completely unreasonable. This thought process is imitated in Honte, as **A** is disregarded due to its low shape value, so that the stones are deemed connected without deep search.

6 INFLUENCE

An important concept in go is *influence*. Strong stones make it hard for the opponent to establish safe groups in their neighborhood, and make it easier for the player with the strength to create territory in the area. Most go playing programs use some kind of influence function that evaluates strong stones' impact in surrounding areas. In Honte, influence is evaluated for both sides.

Influence calculations in Honte are inspired by stochastic process theory, and can be interpreted in the following way for a point p: Start a random walk at p, with each edge being equally probable. Terminate the walk when a stone is hit, or after 20 steps. Define black's influence as the probability of terminating at a black stone, and conversely for white. This procedure is modified to account for the safety value of the stones reached. Evaluating influence is done by solving difference equations for the influence values, which is rather time consuming. Currently this is not a big problem, as the bottlenecks of the program lie elsewhere.

7 NEURAL NETS EVALUATING SAFETY AND TERRITORY

One of the major problems with go programming is to evaluate the safety of stones. When a human player evaluates a go position, he first decides which strings belong to the same *group*, and then evaluates the safety of each group. If a group is firmly sealed in, and has a small space for making two eyes, alpha-beta search can settle the safety problem. But most of the time groups will only be partly sealed in, and the area for making two eyes will not be strictly defined. In these cases humans rely heavily on judgment based on experience, so this might be a good area for testing neural nets.

The human group concept is not easy to imitate in a program, and in Honte a group is simply defined as a set of strings that can connect, which tends to give smaller groups than humans normally consider. Our safety-evaluating neural net takes properties of the group as input, and produces as output an estimate of the probability that the group will eventually survive. The learning-paradigm is TD-learning from self play [7], which means that the program plays games against itself, and uses this experience to improve safety estimates.

Defining a good set of input features is much harder for this safety-evaluating net than for the shape-evaluating net. Groups can have any shape and size, so encoding the location of all its stones is not practical. Here follow some of the features that have been chosen. (For all features, the input value is a real number achieved by scaling by some constant to give an approximate value range of 0 to 1.)

Size: Experience shows that the sheer size (number of stones) of a group correlates well with its safety.

Number of liberties: The total number of liberties for all the strings of the group must be relevant, as this gives information about how close it is to being captured.

Number of established eyes: If the group has two eyes, neural net evaluation is unnecessary, but if the number is less than two, it is very important. Go theory defines half eyes in a consistent way, so values from 0 to 1.5 are relevant.

Average opponent influence over liberties: If the opponent has strong influence over the group's liberties, he has strong stones close to the group, which obviously affects the chances for survival.

The net has a total of 13 input fields and 20 hidden nodes. Output of the net is a real value between 0 and 1, which is interpreted as probability of survival, under the assumption that Honte plays both sides of the games.

The training procedure is a version of the famous algorithm of temporal differences TD(λ), used successfully by Tesauro on backgammon [12]. How to implement the algorithm is not obvious, however, as groups are volatile entities, which may connect and be cut and reconnect several times throughout the game.

The only stable objects through a game are points. At the end of the game stones are evaluated as being dead or alive. Stones that are alive, are given at safety value 1, while those that are dead are given safety value 0. Then the game is re-winded one play at the time, and the following is performed for each of the positions generated:

1. Each group's feedback-safety is calculated as an average of its stones' current safety value. These are used as training signals for updating the neural net's evaluations.
2. Each stone's new safety value S is calculated from the current safety value s and the neural net's estimate of its group's safety g, according to the formula

$$S \leftarrow (1-\lambda)g + \lambda s$$

As the calculations go backwards in the game, S is the safety value of the point prior to s. Just like for the standard TD-algorithm, the feedback signal is a weighted average of the net's evaluations of states later in the game.

If a play captured enemy stones, these reappear when the play is taken back. These stones are then introduced with a safety value of 0. This works quite well in practice, but there are some problems when a stone is sacrificed as part of a successful attack. Although these stones are captured, they are still successful, as the points they sit on end up belonging to their side.

A simple neural net is also trained for evaluating the probability that an open point ends up belonging to black or white. This net uses as input only the influence values of the two sides together with distance from the edge. Influence values are calculated by propagation of safety values, so the inputs to the territory evaluating net depend on the output of the group safety evaluating net. The training procedure is similar to that used for the group safety net. The territory evaluating net has a total of four input fields, 10 hidden nodes and one output node.

It is not easy to measure the merits of these safety- and territory-estimating nets in an objective way, and it has not been attempted so far. However, some effort has been invested in programming heuristic functions that estimate safety and territory, and the nets appear to outperform these.

8 EVALUATION OF HONTE

As we have pointed out earlier, evaluating, and even defining, the strength of a go program is not easy, and we do not make any strong claims about Honte's abilities. However, for what it is worth, Honte does appear to win more than half the time against the shareware program Ego 1.0 [1], and it plays approximately evenly against Handtalk 97-06e with three handicap stones (on full 19-times-19 board). Honte has defeated this version of Handtalk without handicap stones once. These comparisons are not completely fair, as Honte uses more time than the other two programs.

We could make an endless list of weaknesses in the program's play, but we will just mention a few. It does not evaluate whether a play requires a response from the opponent, which is very important for evaluating the dynamics of the game. It has little clue about the endgame, and often plays for shape when it should only maximize territory. Also the program does not really know when the game is over, and often keeps playing after all areas are settled. (This is bad only by the Japanese rules.)

There is a more subtle and interesting weakness in the program, however, that we will explain more thoroughly. When it plays against Handtalk, it often spreads out too many weak stones in the opening, and spends the rest of the games trying to save these. To begin with we believed that this meant that the safety evaluating net was too optimistic. This would not be so strange, as the program is trained from experience, and if it is weak at attacking groups, it might underestimate the risk of being killed itself. But when we checked the evaluations given by the program, they seemed reasonable enough. The groups that we see as weak were typically given a safety of 0.6. The influence evaluation, followed by the territory evaluation, then typically gives as result that the area around the group is neutral; both sides have equal chance of making territory. This also makes perfect sense. The problem with the calculations is that they disregard *how* the groups live or die. If a group has 60 % chance of living, under the assumption that the opponent attacks it, and the player tries to save it, it has huge side effects. While attacking the group, the opponent builds up strong positions on the outside, which are very valuable. This second-order effect of the weak group does not enter the calculations in Honte. The fact is that even when the weak group is saved, the result is bad, as it will have spent all its plays saving it, while the opponent has made territory while attacking. The 40 % of the time when the group dies are far worse, even disastrous.

The problems stem from a basic misconception, often seen with human amateurs as well. The concept of *safety* (meaning probability of survival) is confused with the concept of *strength*. A strong group is a group that is safe, and cannot be harassed. In figure 6, showing the lower left of a board, white has just invaded black's corner at the 3-3-point.

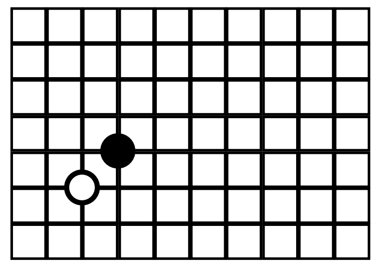

Figure 6: White Is Safe, but Not Strong

This invasion is safe, and Honte tends to like it, as it transforms potential black territory into white territory. However, black can put pressure on the white stone, typically producing a position like figure 7. Although white has survived, black's outside wall is normally worth more.

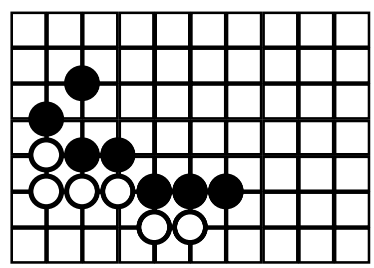

Figure 7: White Survives, but Is Hemmed In

Honte, like human amateurs, also has trouble with a dual phenomenon. When a group ends up being captured, Honte learns that the stones were unsafe and had no value. But just like a safe stone can be a liability, a dead stone can be an asset. This is called *aji* in Japanese. If a group of stones have aji, they can, and probably will be, captured, but the side owning them can threaten to bring them back to life. When the opponent defends against these threats, he wastes time killing stones that were technically dead in the first place, while the opponent makes valuable plays. Determining which stones to defend, and which to use as aji, is one of the deepest mysteries to human amateur players. Figure 8 shows a position similar to the previous one, except that white has a stone on the outside.

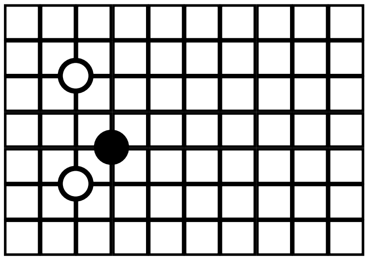

Figure 8: White's Invasion Is Strengthened by an Outside Stone

Black can, and normally should, break the connection between these white stones, but nevertheless the outside stone helps white's invasion. A common joseki gives the position in figure 9 as result.

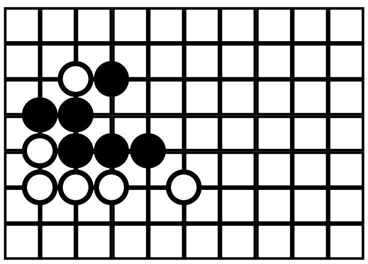

Figure 9: White Sacrifices the Outside Stone for a Better Corner Position

White has used the outside stone as aji, trading it for a stronger position in the corner, getting more territory and more influence over the lower side. And there is still some aji left in the sacrificed stone.

We believe that a strong go program cannot be made without using the concepts of strength and aji. It may be possible to train neural nets to estimate these properties from self play, but it is not clear to the author how this should be done.

9 CONCLUSIONS

Although Honte has not yet reached the level of the best commercial go programs, results are promising. The strategy of designing neural nets to imitate human concepts has been successful, and the nets developed appear to capture the meaning of the concepts, at least to some degree. Our work also highlights a problem associated with modeling based on human concepts, in that any misconception held by the programmer may be inherited and exaggerated by the program. In our case, the programmer's belief that the strength of a group could be measured by its probability of surviving only, lead to a systematic weakness in the program.

REFERENCES

[1] Burmeister J, Wiles J (1995): "An Introduction to the Computer Go Field and Associated Resources", Technical Report 339, Department of Computer Science and Psychology, The University of Queensland, QLD 4072, Australia. WWW: http://www.psy.uq.edu.au/~jay/.

[2] Luce R D, Raiffa H (1957): "Games and Decisions", Wiley, New York.

[3] Halck O M, Dahl F A (1999): "On Classification of Games and Evaluation of Players – with Some Sweeping Generalizations About the Literature", the present volume.

[4] Mechner D (1998): "All systems go", The Scientist, Volume 38, Number 1, New York Academy of Science.

[5] Russell S, Norvig P (1995): "Artificial Intelligence, A Modern Approach", Prentice Hall.

[6] Lichtenstein D, Sipser M (1980): "Go is Polynomial-Space Hard", Journal ACM 27,2, 393-401.

[7] Sutton R S (1988): "Learning to predict by the methods of temporal differences", Machine Learning 3, 9-44.

[8] Enzenberger M (1996): "The Integration of A Priori Knowledge into a Go Playing Neural Network" WWW: http://www.cip.physik.uni-muenchen.de/~enz/biblio.html.

[9] Schraudolph N N, Dyan P, Sejnowski T J (1994): "Temporal Difference Learning of Position Evaluation in the Game of Go", Cowan J D, Tesauro G , Alspector J (eds), Advances in Neural Information Processing 6, Morgan Kaufmann, San Francisco.

[10] Hassoun M H (1995): "Fundamentals of Artificial Neural Networks", Massachusetts Institute of Technology.

[11] Stoutamire D (1991): "Machine learning, game play and Go", Technical Report TR 91-128, Case Western Reserve University, Cleveland Ohio, 1991.

[12] Tesauro G J (1992): "Practical issues in temporal difference learning", Machine Learning 8, 257-277.