Reinforcement Learning of Local Shape in the Game of Go

David Silver, Richard Sutton, and Martin Müller
Department of Computing Science
University of Alberta
Edmonton, Canada T6G 2E8
{silver, sutton, mmueller}@cs.ualberta.ca

Abstract

We explore an application to the game of Go of a reinforcement learning approach based on a linear evaluation function and large numbers of binary features. This strategy has proved effective in game playing programs and other reinforcement learning applications. We apply this strategy to Go by creating over a million features based on templates for small fragments of the board, and then use temporal difference learning and self-play. This method identifies hundreds of low level shapes with recognisable significance to expert Go players, and provides quantitive estimates of their values. We analyse the relative contributions to performance of templates of different types and sizes. Our results show that small, translation-invariant templates are surprisingly effective. We assess the performance of our program by playing against the Average Liberty Player and a variety of computer opponents on the 9×9 Computer Go Server. Our linear evaluation function appears to outperform all other static evaluation functions that do not incorporate substantial domain knowledge.

1 Introduction

A number of notable successes in artificial intelligence can be attributed to a straightforward strategy: linear evaluation of many simple features, trained by temporal difference learning, and combined with a suitable search algorithm. Games provide interesting case studies for this approach. In games as varied as Chess, Checkers, Othello, Backgammon and Scrabble, computers have exceeded human levels of performance. Despite the diversity of these domains, many of the best programs share this simple strategy.

First, positions are evaluated by a linear combination of many features. In each game, the position is broken down into small, local components: material, pawn structure and king safety in Chess [3]; material and mobility terms in Checkers [7]; configurations of discs in Othello [2]; checker counts in Backgammon [13]; single, duplicate and triplicate letter rack leaves in Scrabble [9]; and one to four card combinations in Hearts [11]. In each case, with the notable exception of Backgammon, a linear evaluation function has proven most effective. They are fast to compute; easy to interpret, modify and debug; and they have good convergence properties.

Secondly, weights are trained by temporal difference learning and self-play. The world champion Checkers program Chinook was hand-tuned by expert players over 5 years. When weights were trained instead by self-play using a temporal difference learning algorithm, the program equalled the performance of the original version [7]. A similar approach attained master level play in Chess [1]. TD-Gammon achieved world class Backgammon performance after training by TD(0) and self-play [13]. A program trained by TD(λ) and self-play outperformed an expert, hand-tuned version at the card game Hearts [11]. Experience generated by self-play was also used to train the weights of the world champion Othello and Scrabble programs, using least squares regression and a domain specific solution respectively [2; 9].

Finally, a linear evaluation function is combined with a suitable search algorithm to produce a high-performance game playing program. Minimax search variants are particularly effective in Chess, Checkers, Othello and Backgammon [3; 7; 2; 13], whereas Monte-Carlo simulation has proven most successful in Scrabble [9] and Hearts [11].

In contrast to these games, the ancient oriental game of Go has proven to be particularly challenging. The strongest programs currently play at the level of human beginners, due to the difficulty in constructing a suitable evaluation function [6]. It has often been speculated that Go is uniquely difficult for computers because of its intuitive nature, and requires an altogether different approach to other games. Accordingly, many new approaches have been tried, with limited success.

In this paper, we return to the strategy that has been so successful in other domains, and apply it to Go. We develop a systematic approach for representing intuitive Go knowledge using local shape features. We evaluate positions using a linear combination of these features, and learn weights by temporal difference learning and self-play. Finally, we incorporate a simple alpha-beta search algorithm.

2 The Game of Go

The main rules of Go are simple. Black and white players take turns to place a single stone onto an intersection of the Go board. Stones cannot be moved once played, but may be captured. Sets of adjacent, connected stones of one colour

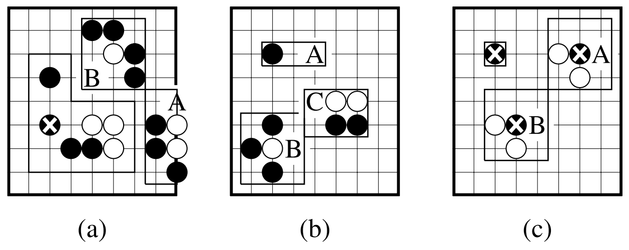

Figure 1: (a) If black plays at *A* he captures two white stones on the right. Playing at *B* is a common tesuji to capture the white stone at the top. The stones in the bottom-left form a common joseki from the marked stone. (b) Black can play according to one of three proverbs: *A* is the one-point jump; *B* is the ponnuki; and *C* is a hane at the head of two stones. (c) The safety of the marked black stone depends on context: it is safe in the top-left; should be captured by white *A* in the top-right; but should be safe from white *B* in the bottom-left.

are known as *blocks*. The empty intersections adjacent to a block are called its *liberties*. If a block is reduced to zero liberties by the opponent, it is captured and removed from the board (Figure 1a). At the end of the game, each player's score is equal to the number of stones they have captured plus the total number of empty intersections, known as *territory*, that they have surrounded.

3 Shape Knowledge in Go

The concept of shape is extremely important in Go. A good shape uses local stones efficiently to maximise tactical advantage. Professional players analyse positions using a large vocabulary of shapes, such as *joseki* (corner patterns) and *tesuji* (tactical patterns). These may occur at a variety of different scales, and may be specific to one location on the board or equally applicable across the whole board (Figure 1). For example, the joseki at the bottom left of Figure 1a is specific to the marked black stone on the 3-4 point, whereas the tesuji at the top could be used at any location. Many Go proverbs exist to describe shape knowledge, for example "ponnuki is worth 30 points", "the one-point jump is never bad" and "hane at the head of two stones" (Figure 1b).

Commercial Computer Go programs rely heavily on the use of pattern databases to represent shape knowledge [6]. Many years are devoted to hand-encoding professional expertise in the form of local pattern rules. Each pattern recommends a move to be played whenever a specific configuration of stones is encountered on the board. The configuration can also include additional features, such as requirements on the liberties or strength of a particular stone. Unfortunately, pattern databases suffer from the knowledge acquisition bottleneck: expert shape knowledge is hard to quantify and encode, and the interactions between different patterns may lead to unpredictable behaviour. If pattern databases were instead learned purely from experience, it could significantly boost the robustness and overall performance of the top programs.

Prior work on learning shape knowledge has focussed on predicting expert moves by supervised learning of local shape [10; 14]. Although success rates of around 40% have been achieved in predicting expert moves, this approach has not led to strong play in practice. This may be due to its focus on mimicking rather than evaluating and understanding the shapes encountered.

A second approach has been to train a multi-layer perceptron, using temporal difference learning by self-play [8; 4]. The networks implicitly contain some representation of local shape, and utilise weight sharing to exploit the natural symmetries of the Go board. This approach has led to stronger Go playing programs, such as Enzenberger's NeuroGo III [4], that are competitive with the top commercial programs on 9×9 boards. However, the capacity for shape knowledge is limited by the network architecture, and the knowledge learned cannot be directly interpreted or modified in the manner of pattern databases.

4 Local shape representation

We represent local shape by a template of features on the Go board. The shape *type* is defined by the template size, the features used in the template, and the weight sharing technique used.

A *template* is a configuration of features for a rectangular region of the board. A basic template specifies a colour (black, white or empty) for each intersection within the rectangle. The template is matched in a given position if the rectangle on the board contains exactly the same configuration as the template. A *local shape feature* simply returns a binary value indicating whether the template matches the current position.

We use weight sharing to exploit several symmetries of the Go board [8]. All rotationally and reflectionally symmetric shapes share the same weights. Colour symmetry is represented by inverting the colour of all stones when evaluating a white move. These invariances define the class of *location dependent* shapes. A second class of *location independent* shapes also incorporates translation invariance. Weights are shared between all local shape features that have the same template, regardless of its location on the board. Figure 2 shows some examples of weight sharing for both classes of shape.

For each type of shape, all possible templates are exhaustively enumerated to give a *shape set*. For template sizes up to 3×3, weights can be stored in memory for all shapes in the set. For template sizes of 4×3 and larger, storage of all weights in memory becomes impractical. Instead, we utilise *hashed weight sharing*. A unique Zobrist hash [15] is computed for each location dependent or location independent shape, and h bins are created according to the available memory. Weights are shared between those shapes falling into the same hash bin, resulting in pseudo-random weight sharing. Because the distribution of shapes is highly skewed, we hope that updates to each hash bin will be dominated by the most frequently occurring shape. This idea is similar to the hashing methods used by tile coding [5].

There is a partial order between shape sets. We define the predicate $G_{i,j}$ to be 1 if shape set S_i is more general than shape set S_j, and 0 otherwise. Shape sets with smaller templates are strictly more general than larger ones, and location

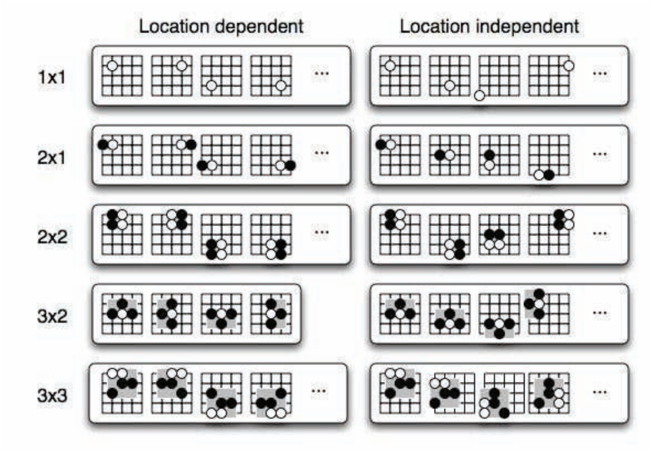

Figure 2: Examples of location dependent and location independent weight sharing, on a 5×5 board.

independent shape sets are strictly more general than location dependent. A more general shape set provides no additional information over a more specific shape set, but may provide a useful abstraction for rapid learning.

The frequency with which shapes occur varies by several orders of magnitude. For each of the m shape sets S_j, the total number of weights in the set is denoted by N_j, and the total number of local shape features in the set that are matched in any position is a constant value n_j. Figure 3 shows the breakdown of total number and frequency of shape sets on a 5×5 board.

5 Learning algorithm

In principle, a set of shape weights can be learned for any purpose, for example connecting, capturing, or surrounding territory. In our case, weights are learned that directly contribute to winning the game of 9×9 Go. At the end of each game, the agent is given a reward of $r = 1$ for a win and $r = 0$ for a loss.

The value function $V^{\pi}(s)$ is defined to be the expected reward from board position s when following policy π, or equivalently the probability of winning the game. We form an approximation $V(s)$ to the value function by taking a linear combination of all local shape features $\phi_{j,k}$ from all shape sets j, with their corresponding weights $\theta_{j,k}$. The sigmoid function σ squashes the output to the desired range $[0, 1]$.

$$V(s) = \sigma\left(\sum_{j=1}^{m}\sum_{k=1}^{n_j}\phi_{j,k}(s)\theta_{j,k}\right) \tag{1}$$

.

All weights θ are initialised to zero. The agent selects moves by a ϵ-greedy single-ply search over the value function, breaking ties randomly. After the agent plays a move, the weights are updated by the TD(0) algorithm [12]. The step-size is the same for all local shape features within a set, and is defined to give each set an equivalent proportion of credit, by normalising by the number of parameters updated on each time-step.

$$\delta = r + V(s_{t+1}) - V(s_t) \tag{2}$$

$$\Delta\theta_{j,k} = \frac{\alpha}{mn_j}\delta\phi_{j,k}V(s_t)(1 - V(s_t)) \tag{3}$$

Because the local shape features are binary, only a subset of features need be evaluated and updated. This leads to an an efficient $O(\sum_{j=1}^{m} n_j)$ implementation rather than the $O(\sum_{j=1}^{m} N_j)$ time that would be required to evaluate or update all weights.

6 Experiments with learning shape in 5×5 Go

We trained two Go-playing agents by coadaptive self-play, each adapting its own set of weights so as to defeat the other. This approach offers some advantages over self-play with a single agent: it utilises two different gradients to avoid local minima; and the two learned policies provide a robust performance comparison. To prevent games from continuing for excessive numbers of moves, agents were not allowed to play within their own single-point eyes [4].

To test the performance of an agent, we used a simple tactical algorithm for the opponent: ALP, the average liberty player. To evaluate a position, the average number of liberties of all opponent blocks is subtracted from the average over the player's blocks. Any ties are broken randomly. For every 500 games of self-play, 100 test games were played between each agent in the pair and ALP.

We performed several experiments, each consisting of 50 separate runs of 100,000 training games. At the end of training, a final 1,000 test games were played between each agent and ALP. All games were played on 5×5 boards. The learning rate in each experiment was measured by the average number of training games required for both agents to exceed a 50% win rate against ALP, during ongoing testing. Overall performance was estimated by the average percentage of wins of both agents during final testing. All experiments were run with $\alpha = 0.1$ and an exploration rate of $\epsilon = 0.1$ during training and $\epsilon = 0$ during testing. For shape sets using hashed weight sharing, the number of bins was set to $h = 100,000$.

In the first set of experiments, agents were trained using just one set of shapes, with one experiment each for the 1×1

Template size	Location independent		Location dependent	
	n_i	N_i	n_i	N_i
1×1	25	3	25	27
2×1	40	9	40	54
2×2	16	81	16	324
3×2	24	729	24	2916
3×3	9	19683	9	78732
4×3	12	h	12	h
4×4	4	h	4	h
5×4	4	h	4	h
5×5	1	h	1	h

Figure 3: Number of shapes in each set for 5×5 Go.

location independent shape set, up to the 5×5 location dependent shape set. The remaining experiments measured the effect of combining shape sets of various degrees of generality. For each shape set S_i, an experiment was run using all shape sets as or more general, $\{S_j : G_{j,i}\}$.

Amongst individual shape sets, the 2×2 shapes perform best (Figure 4), achieving a 25% win rate within just 1000 games, and surpassing an 80% win rate after further training. Smaller shapes lack sufficient representational power, and larger shapes are too numerous and specific to be learned effectively within the training time. Location independent sets appear to outperform location dependent sets, and learn considerably faster when using small shapes. If training runs were longer, it seems likely that larger, location dependent shapes would become more effective.

When several shape sets are combined together, the performance surpasses a 90% win rate. Location independent sets again learn faster and more effectively. However, a mixture of different shapes appears to be a robust and successful approach. Learning time slows down approximately linearly with the number of shape sets, but may provide a small increase in performance for medium sized shapes.

7 Board growing

The training time for an agent performing one-ply search is $O(k^4)$ for $k \times k$ Go, because both the number of moves and the length of the game increase quadratically with the board size. Training on small boards can lead to significantly faster learning, if the knowledge learned can be transferred appropriately to larger boards.

Of course, knowledge can only be transferred when equivalent features exist on different board sizes, and when the effect of those features remains similar. Local shape features satisfy both of these requirements. A local shape feature on a small board can be aligned to an equivalent location on a larger board, relative to the corner position. The weights for each local shape feature are initialised to the values learned on the smaller board. Some new local shape features are introduced in the centre of the larger board; these do not align with any shape in the smaller board and are initialised with zero weights.

Using this procedure, we started learning on a 5×5 board, and incremented the board size whenever a win rate of 90% against ALP was achieved by both agents.

8 Online cascade

The local shape features in our representation are linearly dependent; the inclusion of more general shapes in the partial order introduces much redundancy. There is no unique solution giving the definitive value of each shape; instead there is a large subspace of optimal weight vectors.

Defining a canonical optimal solution is desirable for two reasons. Firstly, we would like the goodness of each shape to be interpretable by humans. Secondly, a canonical weight vector provides each weight with an independent meaning which is preserved between different board sizes, for example when using the board growing procedure.

To approximate a canonical solution, we introduce the *online cascade* algorithm. This calculates multiple approximations to the value function, each one based on a different subset of all features. For each shape set S_i, a TD-error δ_i is calculated, based on the evaluation of all shape sets as or more general than S_i, and ignoring any less general shape sets. The corresponding value function approximation is then updated according to this error,

$$V_i(s) = \sigma\left(\sum_{j=1}^{m}\sum_{k=1}^{n_j} G_{j,i}\phi_{j,k}(s)\theta_{j,k}\right) \quad (4)$$

$$\delta_i = r + V_i(s_{t+1}) - V_i(s_t) \quad (5)$$

$$\Delta\theta_{j,k} = \frac{\alpha}{mn_j}\delta_j\phi_{j,k}V_j(s_t)(1 - V_j(s_t)) \quad (6)$$

By calculating separate TD-errors, the weights of the most general features will approximate the best representation possible from just those features. Simultaneously, the specific features will learn any weights required to locally correct the abstract representation. This prevents the specific features from accumulating any knowledge that can be represented at a more general level, and leads to a canonical and easily interpreted weight vector.

9 Generalised shape features

The local shape features used so far specify whether each intersection is empty, black or white. However, the templates can be extended to specify the value of additional features at each intersection. This provides a simple mechanism for incorporating global knowledge, and to increase the expressive power of the representation.

One natural extension is to use *liberty templates*, which incorporate an *external liberty count* at each stone, in addition to its colour. An *external liberty* is a liberty of a block that lies outside of the template. The corresponding count measures whether there is zero, one, or more than one external liberty for a particular stone. This provides a primitive measure of a stone's strength beyond the local shape. A *local liberty feature* returns a binary value indicating whether its liberty template matches the current position. There are a large number of local liberty features, and so we use hashed weight sharing for these shapes.

10 Results

To conclude our study of shape knowledge, we evaluated the performance of our shape-based agents at 9×9 Go against a variety of established Computer Go programs. We trained a final pair of agents by coadaptive self-play, using the online cascade technique and the board growing method described above. The agents used all shapes from 1×1 up to 3×3, including both location independent and location dependent shapes based on both local shape features and local liberty features, for a total of around 1.5 million weights.

To evaluate each agent's performance, we connected it to the Computer Go Server[1] and played around 50 games of

[1]http://cgos.boardspace.net/9x9.html

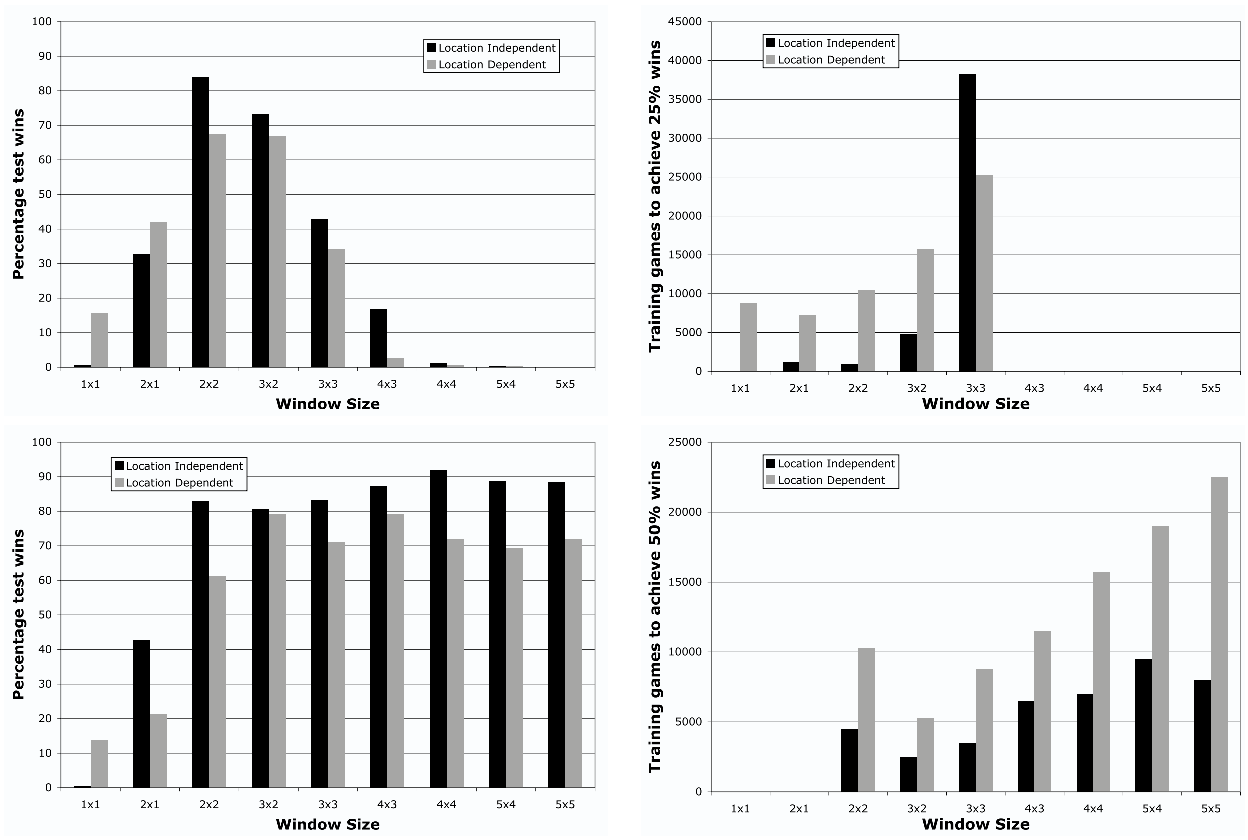

Figure 4: *(Top left)* Percentage test wins against ALP after training with an individual shape set for 100,000 games. *(Top right)* Number of training games with an individual shape set required to achieve 50% test wins against ALP. *(Bottom left)* Percentage test wins using all shape sets as or more general. *(Bottom right)* Training games required for 50% wins, using all shape sets as or more general.

9×9 Go with a 10 minute time control. At the time of writing, this server includes over fifty widely varying Computer Go programs. Each program is assigned an Elo rating according to its performance on the server, currently ranging from around -170 for the random player up to +1860 for the strongest current programs. Programs based on a static evaluation function with little prior knowledge include ALP (+850) and the Influence Player (+700). All of the stronger programs on the server incorporate sophisticated search algorithms or complex, hand-encoded Go knowledge.

The agent trained with local shape features attains a rating of +1070, significantly outperforming the other simple, static evaluators on the server. When local liberty features are used as well, the agent's rating increases to +1140 (Figure 5). Finally, when the basic evaluation function is combined with a full-width, iterative-deepening alpha-beta search, the agent's performance increases to +1210.

CGOS name	Program description	Games	Elo rating
Linear-B	Shape features	71	+1070
Linear-L	Shape and liberty features	28	+1140
Linear-S	Shape features and search	43	+1210

Figure 5: CGOS ratings attained by trained agents.

11 Discussion

The shape knowledge learned by the agent (Figure 6) represents a broad library of common-sense Go intuitions. The 1×1 shapes encode the basic value of a stone, and the value of each intersection. The 2×1 shapes show that playing stones in the corner is bad, but that surrounding the corner is good; and that connected stones are powerful. The 3×2 shapes show the value of cutting the opponent stones into separate groups, and the 3×3 shapes demonstrate three different ways to form two eyes in the corner. Each specialisation of shape adds more detail; for example playing one stone in the corner is bad, but playing two connected stones in the corner is twice as bad.

However, the whole is greater than the sum of its parts. Weights are learned for over a million shapes, and the agent's play exhibits global behaviours beyond the scope of any single shape, such as territory building and control of the corners. Its principle weakness is its local view of the board; the agent will frequently play moves that look beneficial locally but miss the overall direction of the game, for example adding stones to a group that has no hope of survival. Our

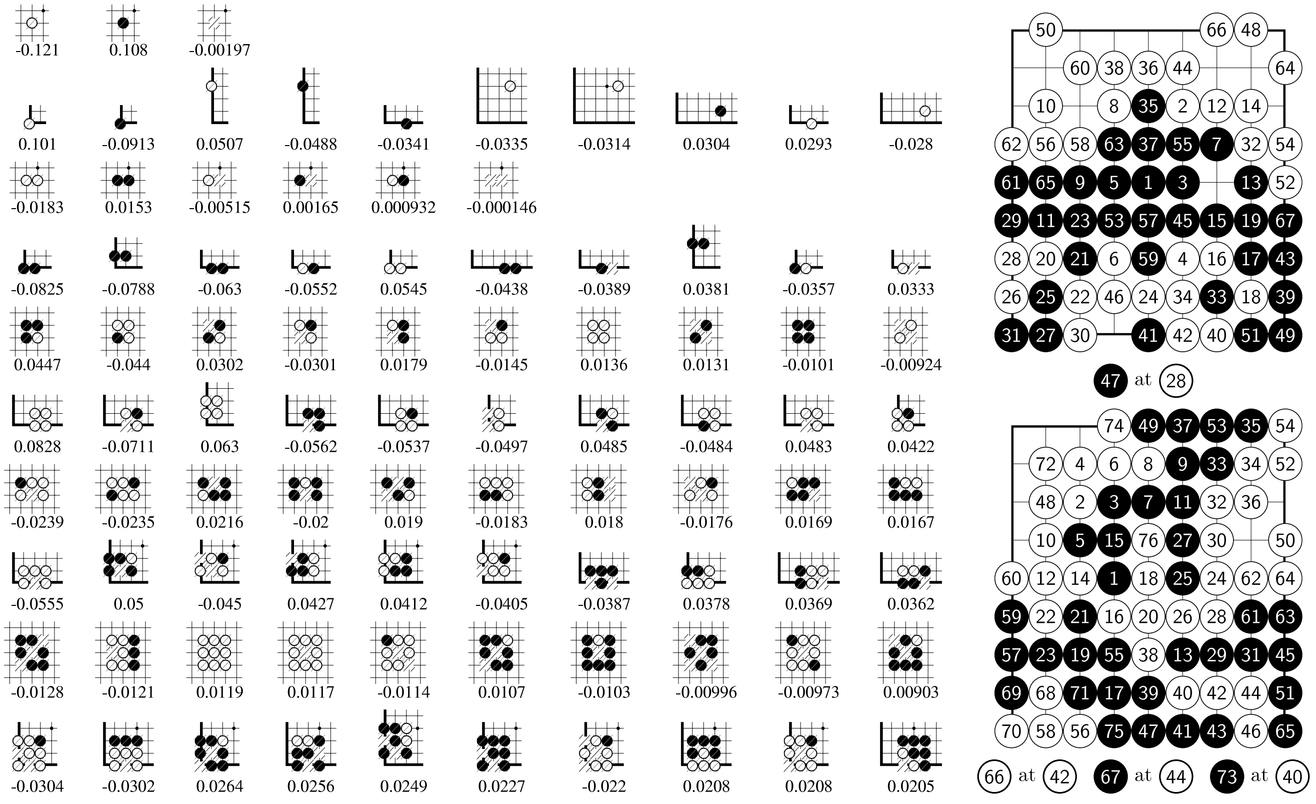

Figure 6: *(Left)* The 10 shapes in each set from 1×1 to 3×3, location independent and location dependent, with the greatest absolute weight after training on a 9×9 board. *(Top-right)* A game between Linear-B (white) and DingBat-3.2 (rated +1580). Linear-B plays a nice opening and develops a big lead. Moves 48 and 50 make good eye shape locally, but for the wrong group. DingBat takes away the eyes from the group at the bottom with move 51 and goes on to win. *(Bottom-right)* A game between Linear-L (white) and the search based Liberty-1.0 (rated +1110). Linear-L plays good attacking shape from moves 24-37. It then extends from the wrong group, but returns later to make two safe eyes with 50 and 62 and ensure the win.

current work is focussed on using generalised shapes to overcome this problem, based on more complex features such as eyes, capture and connectivity.

References

[1] J. Baxter, A. Tridgell, and L. Weaver. Experiments in parameter learning using temporal differences. *International Computer Chess Association Journal*, 21(2):84–99, 1998.

[2] M. Buro. From simple features to sophisticated evaluation functions. In *First International Conference on Computers and Games*, pages 126–145, 1999.

[3] M. Campbell, A. Hoane, and F. Hsu. Deep Blue. *Artificial Intelligence*, 134:57–83, 2002.

[4] M. Enzenberger. Evaluation in Go by a neural network using soft segmentation. In *10th Advances in Computer Games Conference*, pages 97–108, 2003.

[5] W.T. Miller, E. An, F. Glanz, and M. Carter. The design of cmac neural networks for control. *Adaptive and Learning Systems*, 1:140–145, 1990.

[6] M. Müller. Computer Go. *Artificial Intelligence*, 134:145–179, 2002.

[7] J. Schaeffer, J. Culberson, N. Treloar, B. Knight, P. Lu, and D. Szafron. A world championship caliber checkers program. *Artificial Intelligence*, 53:273–289, 1992.

[8] N. Schraudolph, P. Dayan, and T. Sejnowski. Temporal difference learning of position evaluation in the game of Go. In *Advances in Neural Information Processing 6*, 1994.

[9] B. Sheppard. World-championship-caliber Scrabble. *Artificial Intelligence*, 134(1-2):241–275, 2002.

[10] D. Stoutamire. *Machine Learning, Game Play, and Go*. PhD thesis, Case Western Reserve University, 1991.

[11] N. Sturtevant and A. White. Feature construction for reinforcement learning in hearts. In *5th International Conference on Computers and Games*, 2006.

[12] R. Sutton. Learning to predict by the method of temporal differences. *Machine Learning*, 3(9), 1988.

[13] G. Tesauro. TD-gammon, a self-teaching backgammon program, achieves master-level play. *Neural Computation*, 6:215–219, 1994.

[14] E. van der Werf, J. Uiterwijk, E. Postma, and J. van den Herik. Local move prediction in Go. In *3rd International Conference on Computers and Games*, 2002.

[15] A. Zobrist. A new hashing method with application for game playing. Technical Report 88, Univ. of Wisconsin, 1970.