International Journal on Artificial Intelligence Tools
© World Scientific Publishing Company

EXPERIMENTS WITH LEARNING OPENING STRATEGY IN THE GAME OF GO

Timothy Huang, Graeme Connell, and Bryan McQuade
Middlebury College
Computer Science Department
Middlebury, VT 05753
{*huang,gconnell,bmcquade*}*@middlebury.edu*

Received July 13, 2003
Revised November 11, 2003

We present an experimental methodology and results for a machine learning approach to learning opening strategy in the game of Go, a game for which the best computer programs play only at the level of an advanced beginning human player. While the evaluation function in most computer Go programs consists of a carefully crafted combination of pattern matchers, expert rules, and selective search, we employ a neural network trained by self-play using temporal difference learning. Our focus is on the sequence of moves made at the beginning of the game. Experimental results indicate that our approach is effective for learning opening strategy, that including higher-level features of the game can improve the quality of the learned evaluation function, and that different input representations of higher-level information can substantially affect performance.

Keywords: Temporal difference learning, neural networks, strategy games, game of Go.

1. Introduction

The ancient game of Go is one of the most widely played strategy games in the world, especially in the Far East. More effort has been devoted to developing computer programs that play Go than to developing programs for any other game of skill except chess.[1] Unlike chess, however, where the best programs play as well as the top human players, the best Go programs play only at the level of advanced beginners. Standard game-playing techniques based on brute force minimax search are not sufficient for Go because the game tree is extremely large and because accurate evaluation functions are slow and difficult to construct. Hence, most current programs rely on a carefully crafted combination of pattern matchers, expert rules, and selective search. Unfortunately, the engineering effort involved suggests that making significant progress by simply fine-tuning the individual components will become increasingly difficult and that additional approaches should be explored.

We investigate a machine learning approach to constructing an evaluation function for Go based on training a neural network by self-play using temporal difference (TD) learning. We focus on learning opening strategy, i.e., the sequence of moves played at the beginning of the game. Also, we investigate a number of higher-level features of Go positions and identify those that most improve the quality of the evaluation function. Furthermore, we explore how alternate representations of higher-level feature information to the neural network affect performance.

We begin by presenting an overview of game play in Go and the challenges for computer Go. After describing our learning approach and experimental methodology, we present and analyze experimental results. We conclude by surveying earlier work and discussing future plans.

2. Background

To start, we present a brief description of how Go is played by humans and by computers. Kim and Soo-hyun provide an excellent, comprehensive explanation of the rules.[2] Our aim here is simply to give a sense of the object of the game and the factors that make it difficult for computers to play.

2.1. *Playing Go*

Go is a two-player game played with black and white stones on a board of 19×19 intersecting lines, though 13×13 and 9×9 boards are sometimes used. Starting with black, the players take turns either placing one stone onto one of the empty intersections or passing. The goal is to acquire territory (empty intersections) by surrounding it with stones. Once placed, a stone does not move; however, blocks of stones can be surrounded and captured by the opposing player. The game ends when both players pass. The player who has surrounded the most points of territory wins the game. In official play, there are no draws because the white player receives 5.5 additional points (called *komi*) to compensate for playing second.

Stones placed horizontally or vertically adjacent to each other form a *block*. The empty intersections next to a block are its *liberties*. A block is captured when all of its liberties are occupied by enemy stones. Consider the following examples from Figure 1: In the bottom left corner, a block of white stones has surrounded 12 points of territory. Towards the bottom right, a block consisting of a single black stone is surrounded on three of four sides. White can capture this stone by playing at its last liberty, the intersection marked A. Above those stones, white can capture the block of two black stones by playing at the liberty marked B. If it is black's turn, black can help those stones escape by playing at B first and creating a resultant block with three liberties.

In the upper left corner, black can capture the white stones by playing at C, thereby fully surrounding the three stones. Normally, it is illegal to play a stone where it will not have any liberties. However, playing a black stone at C is allowed because it captures at least one of the white stones directly adjacent to C. In this

case, it captures the entire white block of three stones. In the upper right corner, the white block is *alive*, i.e., safe from capture, because black cannot play a stone at D and a stone at E simultaneously.

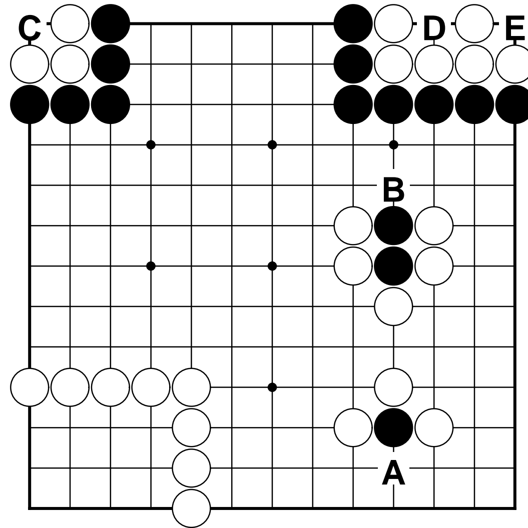

Fig. 1. White can capture one black stone by playing at A or two stones by playing at B. Black can capture three white stones by playing at C. Since black cannot play at D and E simultaneously, the white stones in the upper right corner are alive.

The stones in Figure 1 were artificially arranged for explanatory purposes. Figure 2 shows a more realistic board situation midway through a game on a 13 × 13 board. In this game, white has staked out territory in the upper right, upper left, and lower left sections of the board. Meanwhile, black has staked out territory in the lower right and upper middle sections of the board and has pushed into the middle left section, thereby reducing white's territorial gain. Furthermore, black has effectively captured the white stones in the middle right section of the board.[2]

Part of what makes Go so engrossing and challenging is the interplay between strategy and tactics. On one hand, players try to build stone patterns with "good shape," i.e., those with long-term strategic influence. On the other hand, they fight highly tactical "life-or-death" battles that concern whether a group of stones can be captured or not.

Figure 3 shows the board situation at the end of the same game from Figure 2. Captured stones are marked with X's. The final score, taking into account captured stones and komi, has white ahead by 4.5 points of territory.

2.2. *Computer Go*

From a computer scientist's perspective, what stands out about Go is that it

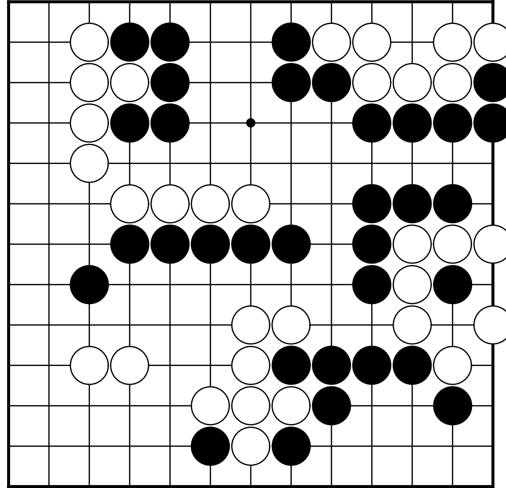

Fig. 2. White's turn to move midway through an example game on a 13 × 13 board.

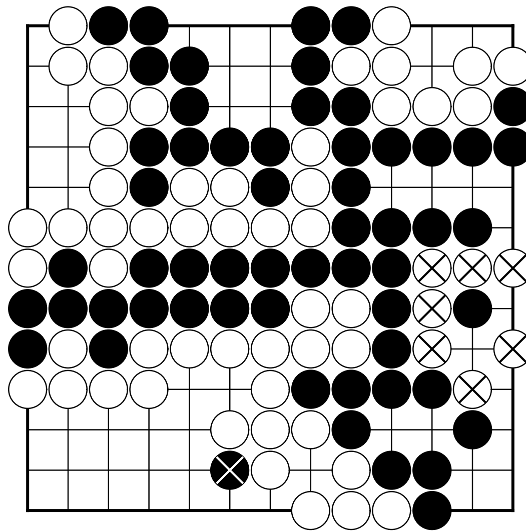

Fig. 3. Final board position. Captured stones are marked with X's. Taking into account captured stones and komi, white wins by 4.5 points.

remains a perplexing computational problem. The standard brute force minimax approach to strategy games is not sufficient for Go for two reasons. First, the game tree is extremely large; the average branching factor is about 250 in Go compared to about 35 in chess, and the average number of moves made in a game by each player is about 100 in Go compared to about 40 in chess. Second, and more significantly, accurate heuristic evaluation functions are difficult to construct and computationally expensive.

Hence, most Go programs contain a mix of various AI components. Pattern matchers recognize common stone arrangements and recommend the best move in those situations. Expert rules identify important strategic regions, estimate territorial balance, and suggest candidate moves. Selective search resolves local questions about the life-or-death status of a block or a group of stones.

Researchers throughout the world continue to work on computer Go and gather regularly at computer Go tournaments to pit their updated programs against each other. Currently, the top programs in the world[3] include Handtalk/Goemate, by Zhixing Chen; Many Faces of Go, by David Fotland; Haruka, by Ryuichi Kawa; Go++, by Michael Reiss; and GNUGo, by Daniel Bump and other developers.

3. Learning Opening Strategy

In this section, we motivate our approach to learning opening strategy, provide details of our experimental methodology, and describe the neural network and higher-level feature information that make up our evaluation function.

3.1. *Temporal difference learning and Go*

Like many other strategy games, the moves played in the early part of a Go game dictate game play for the rest of the game and have the greatest influence on the outcome. At the beginning of a game, players place stones to stake loose territorial claims and to shape the flow of attack and defense in the middle game. Localized battles involving the life-or-death status of a group of stones tend to occur later in the game. Hence, opening strategy focuses more on developing desirable stone patterns than on finding tactical maneuvers to capture a group of stones.

We attempted to learn an evaluation function for opening strategy in Go using a neural network trained with temporal difference learning. We chose a neural network representation because it seemed well-suited for the emphasis on pattern recognition in opening strategy, and we chose temporal difference learning, a kind of reinforcement learning, because we wanted the system to learn by self-play and by play against others. In this subsection, we briefly outline how temporal difference learning can be applied to computer Go. For additional details, Sutton provides a thorough explanation of temporal difference learning,[4] and Sutton and Barto provide a comprehensive introduction to reinforcement learning.[5]

Unlike supervised learning, reinforcement learning relies only on periodic feedback or reinforcement and makes it possible to learn when classified training data

sets are unavailable. Temporal difference (TD) learning methods work by using successive predictions of an environment as it evolves over time. For Go, the inputs might be descriptions of board positions from move to move over the course of a game, and the outputs might be scalar values representing the probability of a win or the relative desirability of a position. The periodic feedback consists of the concrete score at the end of each game.

A traditional Backpropagation approach to neural network learning adjusts the network weights based on the error between the output generated from a particular input and the correct output.[6] Following the notation of Russell and Norvig,[7] the rule for updating the weight $W_{i,j}$ between node j and node i is

$$W_{j,i} \leftarrow W_{j,i} + \alpha \times a_j \times (T_i - o_i) \times g'(in_i), \tag{1}$$

where α denotes the learning rate, a_j denotes the output of node j, the difference between T_i (the "correct" output) and the network's actual output o_i denotes the error at node i, and $g'(in_i)$ denotes the gradient of the squashing function evaluated at the linear input to node i.

Although this approach cannot be applied directly to Go without a suitably large library of game positions and corresponding correct outputs, TD-learning can be used to adjust the network weights based on the difference between the current network output and future network output. We chose to use the TD(0) learning rule because of its simplicity, relatively low computational requirements, and effectiveness in past game-related applications.[8] This rule considers only the output one time step later. In other words, the weight-update rule using TD(0) is

$$W_{j,i} \leftarrow W_{j,i} + \alpha \times a_j \times (o_{i,t+1} - o_{i,t}) \times g'(in_i), \tag{2}$$

where $o_{i,t}$ denotes the output of node i at time (move number) t. It is acceptable for this output to be somewhat inaccurate as long as the learner periodically receives error-free feedback. Such feedback is always available at the end of each game, as an exact score can always be computed once both players pass and the game ends. In those cases, $o_{i,t+1}$ is replaced by the actual score difference. Aside from this change, the other steps of the Backpropagation algorithm can be used without modification.

3.2. *Experimental methodology*

In designing our experiments, we focused on three tasks. First, whereas others have previously investigated the effectiveness of neural networks and TD-learning for Go in general,[9] we wanted to determine their effectiveness for learning opening strategy in particular. Second, since the true value function over raw board positions is likely very non-linear, we wanted to determine whether and how much the inclusion of various higher-level features of the position would improve the learned evaluation function. Third, we wanted to explore how two different input representations of the higher-level features would affect performance. In particular, we

tried representing this information both as single input scalar values and as multiple input binary values. Subsection 3.3 provides further details.

For computational reasons, we limited all our experiments to 9 × 9 boards. We created 16 different computer players, each of which used a neural network evaluation function. Of the 5 features described in Subsection 3.3, each player took as input the raw board position (feature A) and a selection from four additional features (features B, C, D, and E). We refer to a player by the features it takes as input, e.g., Player ACD takes features A, C, and D.

In a typical TD-learning experiment, the learning phase would involve playing a program against itself or another opponent for a series of complete games and using each game's outcome to adjust the neural network weights. Because we wanted to determine which player stands better at the end of the opening, we took a new approach. For each player, we played out an average length opening (10 moves per side on a 9 × 9 board), with each move choice determined by applying the evaluation function to every legal move on the board. After each side had played 10 moves, we handed off the position to the GNUGo program, which we treated as a Go "expert." GNUGo finished the game by playing stones for both sides, and the result was treated as the correct output for the final opening position. Assuming reasonable play by GNUGo, the side with a better position after the opening should end up winning the game. This approach focused our learning and testing on the opening moves, and it had the added benefit of reducing the interval between successive reinforcement signals.

We implemented two enhancements to the learning algorithm. To encourage exploration, we employed a simplified version of Gibbs sampling in which the computer players on occasion randomly choose a move from the classes of less appealing moves instead of the move with highest relative score. To accelerate the learning process, we exploited the eight-fold symmetry possible in any position by preprocessing the raw board information so that functionally equivalent positions yielded the same input values to the network, regardless of actual orientation.

We trained each computer player by initializing its corresponding neural network with random weights and then playing against itself for several thousand games, continually adjusting the network weights using the TD-learner. To measure learning progress, we tested the player against Player Random (a player that chooses its moves randomly) for 100 games after every 50 games of self-play. To measure the benefit of including the higher-level features, we matched the various learned players against each other in round-robin fashion. Finally, to observe the effect of input representation on performance, we matched each player using scalar-valued higher-level information with the corresponding player using binary-valued higher-level information.

3.3. *Neural network details*

Before moving onto experimental results, we briefly describe the neural network

structure and the features that were provided as input to the various players.

The evaluation function consisted of a fully connected two-layer feed-forward neural network with a single bias node and a sigmoid squashing function. The total number of inputs varied substantially (from 723 to 11553) depending on which higher-level features were included as input. The single hidden layer consisted of 40 nodes, each fully connected to the inputs and the output layer. The output layer consisted of a single node whose value represented the predicted difference in score between the two players. While the output ultimately ought to be the game-theoretic value of a particular position, the score difference seemed to provide information helpful to the learning process.

As human players learn Go, they quickly grasp some basic concepts and higher-level features of the game. Players learn that blocks with many stones and more liberties are generally stronger, and that game play near the edges and corners of the board varies substantially from game play near the center. While this sort of higher-level information about stones and blocks follows directly from the raw board position, we reasoned that the value function over these features might be smoother than that over just the raw board position, and we hoped that including them as input might improve the quality of the evaluation function or the speed of learning.

We identified five features, which we labelled A, B, C, D, and E. Feature A actually corresponds to the raw board position and is supplied to each of the 16 players. Features B, C, D, and E are higher-level features that many instructional texts cite as important for choosing a move. We first constructed networks in which these features were supplied as single input scalar values and later constructed networks in which they were supplied as multiple input binary values, i.e., several binary input nodes, only one of which is turned on at any time. In theory, the additional nodes used to represent features as multiple input binary values could provide the network with more freedom to learn the relative differences between, say, a block having 1, 2, or 3 liberties.

Feature A used two input nodes per board intersection to indicate the status of the intersection. One node indicated whether the intersection is occupied, and the other indicated whether an occupied intersection has a friendly or opposing stone. Also part of feature A was a single node (for the entire network) that contained the score difference not represented on the board itself. This included the difference in already captured stones and the komi points awarded to the white player.

Feature B indicated the distance from the intersection to the two closest edges of the board. This information may be significant because board edges and corners greatly influence game play. The binary-valued representation used 5 nodes to indicate 0, 1, 2, 3, or 4+ intersections from the closest vertical board edge, and 5 additional nodes for distance from the closest horizontal edge.

Feature C indicated the number of liberties for the block to which the stone at that intersection belonged. In general, a block with many liberties is stronger and less likely to be captured. The binary-valued representation used 5 nodes to indicate 1, 2, 3, 4, or 5+ liberties for the block.

Feature D indicated the total number of stones in the block to which the stone belonged. This information may be relevant because larger groups tend to be difficult to kill. The binary-valued representation used 5 nodes to indicate 1, 2, 3, 4, or 5+ stones in the block.

Feature E indicated the number of friendly and enemy stones close to the block to which a stone belonged. Nearby friendly stones can help a block to attack and defend. Likewise, nearby enemy stones can hurt its ability to do so. The binary representation used 5 nodes to indicate 0, 1, 2, 3, or 4+ nearby friendly stones, i.e., those within a Manhattan distance of 3 from the block containing the current intersection, and 5 additional nodes for nearby enemy stones.

4. Results and Analysis

Our experiments yielded data that shed light on all three tasks. First, the data showed that TD-learning is effective for training a neural network-based evaluation function for Go opening strategy. Second, the data showed that including higher-level features among the input to the network can improve the quality of the learned evaluation function. Third, the data showed that different representations of the same higher-level feature information can significantly affect the performance of the learned evaluation function.

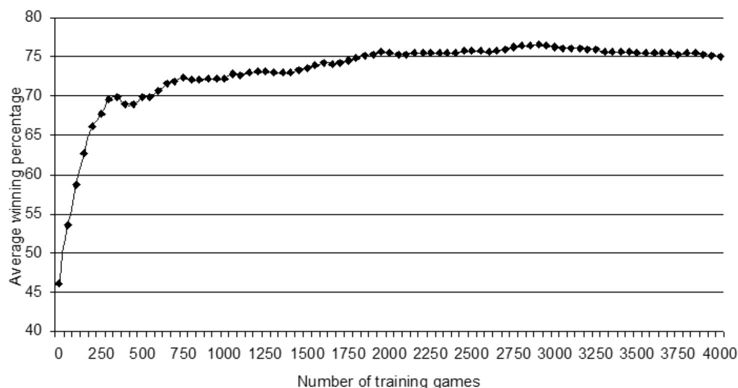

Fig. 4. Learning curve for Player ABCDE.

For the first task, we trained Player ABCDE, the player whose input included the raw board position (Feature A) and all other higher-level features. After every 50 training games, we played 100 games between Player ABCDE and Player Random, a player that randomly chooses a move among all legal moves with uniform probability. As in the learning phase, each player made 10 moves, and then GNUGo played out the game to determine the winner. The learning curve in Figure 4 shows the progress of Player ABCDE. As expected, it initially split about half its games with Player Random. By 3000 training games, it was winning just over 75% of

the games. We observed similar learning curves with all of the other players; each player's performance increased rapidly during the first several hundred games and then leveled off by 3000 training games.

For the second task, evaluating the benefit of including higher-level features in the input to the neural network, we trained all 16 players using scalar-valued higher-level features and then played 100 games between each player in round-robin fashion. Table 1 shows the percentage of games won by each player against all other players. In general, players that used more features won more games. As expected, Player ABCDE, which used all four additional features, performed relatively well against the other players. Somewhat surprisingly, Player ADE won marginally more games than Player ABCDE. In a few cases, adding an individual feature actually caused worse performance. For example, Players AC and AE both won fewer games overall than Player A.

Table 1. Winning percentages of each player using scalar-valued higher-level features in a round-robin tournament against all other players.

Player	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A
				B	B	B		B			B	B	B		B	
		C		C			C		C		C	C		C	C	
					D				D	D	D		D	D	D	D
	E						E	E				E	E	E	E	E
Win %	42	43	46	46	46	48	48	49	50	52	54	54	54	55	55	56

To evaluate the incremental benefit of including each individual high-level feature, we considered all of the players that did not use that feature and computed the percentage improvement in number of games won when that feature was added. Table 2 shows the average number of additional games won when each of the four features was added. Overall, each feature showed a positive improvement when included in the input. The most beneficial feature was D, the size of a stone's block. The next most beneficial feature was E, the number of nearby friendly and enemy stones. We believe that Feature B, the distance from the edges, is more relevant for larger board sizes, where there is much more of a center region. Also, we suspect that Feature C, the number of liberties, becomes a greater factor in the middle game, when life-or-death issues occur more frequently.

Table 2. Average number of additional games won when the individual scalar-valued feature was added to the input.

Feature	Average Improvement
B	4%
C	3%
D	13%
E	8%

We ran the same round-robin tournament with the 16 players using binary-

valued higher-level inputs, and Table 3 shows the percentage of games won by each player against all other players. As was the case with scalar-valued higher-level inputs, the players using more higher-level features generally performed better than those using fewer. However, the difference in performance between the best and worst players was much greater. Player A, which took only the raw board position as input, performed very poorly against the other players.

Table 3. Winning percentages of each player using binary-valued higher-level features in a round-robin tournament with all other players.

Player	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A
		B	B	B	B			B				B	B	B		
			C		C	C	C		C			C		C	C	
				D	D	D					D		D	D	D	D
							E	E		E		E	E	E	E	E
Win %	21	22	28	47	48	52	52	54	54	55	57	57	58	64	65	65

Likewise, Table 4 shows the average number of additional games won when each of the four higher-level features was included. Again, including feature D and feature E provided the greatest performance improvement. Here, however, including feature B actually caused a minor decrease in performance. It appears that the additional degrees of freedom from representing the distance to the board edges as binary values actually made it more difficult for the network to learn their relevance and significance.

Table 4. Average number of additional games won when the individual binary-valued feature was added to the input.

Feature	Average Improvement
B	-5%
C	5%
D	14%
E	18%

For the third task, we compared how each player using scalar-valued inputs fared against the corresponding player (i.e., the player with the same higher-level features) using binary-valued inputs. Table 5 shows the results of that testing. While players using scalar-valued inputs won almost the same total number of games as the players using binary-valued inputs, results varied substantially by specific feature. Using binary-valued representations for features C and D improved performance, as players using features C and D won 54% and 58% of their games, respectively, against their counterparts. By contrast, using binary-valued representations for features B and E decreased performance, as players using features B and E won only 48% and 43% of their games, respectively, against their counterparts. Not surprisingly, player ACD using binary-valued inputs had one of the highest percentages, winning 69% of its games against its counterpart. Overall, binary-valued higher-level inputs

are not always beneficial; they were helpful when representing a block's liberties and size but unhelpful when representing a stone's position on the board or a block's relation to other stones.

Table 5. Percent of games won by the player using binary-valued inputs against the corresponding player using scalar-valued inputs.

Player	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A
		B		B		B		B		B		B		B		B
			C	C			C	C			C	C			C	C
					D	D	D	D					D	D	D	D
									E	E	E	E	E	E	E	E
Win %	50	29	63	48	70	63	69	66	35	26	33	52	49	43	49	54

5. Previous Work

Machine learning has long been an appealing approach to constructing evaluation functions for strategy games. It promises not only to reduce the level of human effort required but also to identify and represent knowledge that may not be easy to encode manually. The challenges include choosing an appropriate representation language, devising an efficient learning algorithm, and obtaining beneficial training experience.

When classified training data sets are available, traditional supervised learning approaches have been used to learn various aspects of game play in Go. Using a database of expert games and treating each played move as correct and every other legal move as incorrect, Dahl trained a neural network to identify stone patterns with "good shape" vs. those with "bad shape."[10] In related work, Enderton used neural networks for move ordering and forward pruning of selective search in his Golem Go program,[11] and Stoutamire used hashed sets of patterns rather than neural networks to identify good and bad shape.[12] Using neural networks in conjunction with automatic feature extraction methods and informed pre-processing, van der Werf and colleagues learned to predict good local moves from game records.[13]

Among reinforcement learning approaches, Tesauro first showed how to apply TD-learning of neural networks to strategy games with his TD-Gammon program, a world class backgammon player.[8] Schraudolph and colleagues applied the same approach to Go with more modest success.[9] Their work is most closely related to ours, though it does not focus on opening strategy or alternate input representations, and it uses a more complicated neural network for its evaluation function. Enzenberger worked to integrate this approach with manually encoded expert Go knowledge in his NeuroGo program.[14]

6. Conclusions and Future Work

We described an approach for learning opening strategy in the game of Go by training a neural network evaluation function using TD-learning. To focus on opening strategy, we played only the opening moves and then employed an "expert"

Go program to play out and score positions at the end of the opening. We also presented an experimental methodology in which we trained 16 computer players, each taking as input a different combination of four higher-level features, and we compared the resulting players with each other. While there remains ample room for improvement, the experimental results indicate that our approach is effective for learning opening strategy and that all four higher-level features help improve the quality of the learned evaluation function. Moreover, we compared the effect of representing higher-level feature information as single input scalar values vs. multiple input binary values. Our results suggest that while the choice of input representation can significantly affect performance, neither type of representation is always preferable to the other.

While this research itself is specific to the game of Go, it has ramifications for other work in machine learning. First, it provides an example of how TD-learning of neural networks can be applied even in situations where the interval between reinforcement signals is normally quite long. Second, it provides a methodology for comparing the relative benefit of various features to the quality of a learned evaluation function.

Looking forward, there are several possible ways to build upon this work. One way involves trying other network structures, hidden layer sizes, input features and combinations, and output representations. For example, the current results suggest that a player combining scalar-valued inputs for features B and E with binary-valued inputs for features C and D could be especially strong. A second way involves testing on larger board sizes. While our approach is directly applicable to 13×13 and 19×19 boards, it will require much more computation time, most of which is used by GNUGo to play out games. A third way involves learning other aspects of game play besides opening strategy. One possibility would be to learn *joseki*, the common patterns of play that are often hard-coded into the pattern matchers of many programs. A fourth way involves exploring methods for combining *a priori* Go knowledge with our learning approach. Earlier efforts[14] suggest that integrating learning and other AI techniques in a Go-playing system has substantial potential for increasing performance.

Acknowledgements

This work was supported by the National Science Foundation under Grant No. 9876181, and by Middlebury College. We thank Mark Harrington, Yunpeng Li, Kimberly Lommler, David Mendelson, Ben Nobel, Ryan Richards, and Matthew Wilder for their contributions to this work.

References

[1] M. Müller, *Computer Go*, Artificial Intelligence Journal, **134(1-2)** (2000) 145–179.
[2] J. Kim and J. Soo-hyun, *Learn to Play Go, Volume 1, 2nd Edition*, Good Move Press (1997).

[3] M. Reiss, *Mick's Computer Go Page: News and information about computer programs that play Go*, Internet, http://www.reiss.demon.co.uk/webgo/compgo.htm (2003).

[4] R. Sutton, *Learning to Predict by the Methods of Temporal Differences*, Machine Learning, **3** (1988) 9–44.

[5] R. Sutton and A. Barto, *Reinforcement Learning: An Introduction*, MIT Press, Cambridge, MA (1998).

[6] J. Hertz, A. Krogh, and R. Palmer, *Introduction to the Theory of Neural Computation*, Addison-Wesley (1991).

[7] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach, 2nd Edition*, Prentice-Hall, New Jersey (2003).

[8] G. Tesauro, *Temporal Difference Learning and TD-Gammon*, Communications of the ACM, **38(3)** (1995) 58–68.

[9] N. Schraudolf, P. Dayan, and T. Sejnowski. *Temporal Difference Learning of Position Evaluation in the Game of Go*, Advances in Neural Information Processing Systems, **6** (1994).

[10] F. Dahl, *Honte, a Go-Playing Program Using Neural Nets*, International Conference on Machine Learning Workshop on Machine Learning in Game Playing, http://www.ai.univie.ac.at/icml-99-ws-games/ (1999).

[11] H. Enderton, *The Golem Go Program*, Carnegie-Mellon University Technical Report #CMU-CS-92-101 (1991).

[12] D. Stoutamire, *Machine Learning, Game Play, and Go*, Case Western Reserve University Technical Report #TR91-128 (1991).

[13] E. van der Werf, J. Uiterwijk, E. Postmas, and J. van den Herik, *Local Move Prediction in Go*, Proceedings of the 3rd International Conference on Computers and Games, Edmonton, Canada (2002).

[14] M. Enzenberger, *The Integration of A Priori Knowledge into a Go Playing Neural Network*, Internet, http://www.markus-enzenberger.de/neurogo.html (1996).