A Machine-Learning Approach to Computer Go

Jeffrey Bagdis
Advisor: Prof. Andrew Appel

May 8, 2007

1 Introduction

Go is an ancient board game dating back over 3000 years. Although the rules of the game are simple, go strategy is extremely complex. Even the simple process of evaluating the strength of a position involves complex pattern interactions that can span the entire board. The difficulty of evaluating go strategy has stymied attempts to produce a computer go AI from attaining a level above that of a human amateur. This paper presents an approach for using a neural network to approximate a value function over the space of board states, and using temporal difference learning to refine that value function through experience. This approach alone does not seem to be an effective method of creating a go AI. Simple move selection based on a shallow alpha-beta search of the game tree using the derived value function is easily defeated by both human and computer opponents. However, the derived value function does seem to give a good evaluation of the strength of the board. Combined with other move selection techniques, it could prove effective. Additionally, the value function demonstrates continual improvement under training, indicating that additional training beyond the scope of this paper would likely provide additional gains in performance.

This paper assumes a basic knowledge of the rules of go, including terminology and fundamental strategy. For a brief treatment of the fundamentals of the game, consult appendix A before proceeding.

2 Challenges of Computer Go

Computers play chess very well. The general approach to computer chess involves an optimized minimax search of the game tree to a certain depth, coupled with a decent heuristic evaluation function to estimate the

value of a given board position. Deep Blue, the first computer to defeat World Champion Gary Kasparov, performed deep tactical searches into the game tree [7]. Since chess has a reasonably small branching factor (roughly about 35 on average [8]) a deep, tactical minimax search is computationally feasible. A simple heuristic function that compares the material value of each player's pieces on the board is also reasonably effective for computer chess, and slightly more advanced heuristic functions perform even better [8]. These techniques work extremely well for computer chess, and many other strategy games, but they do not work for computer go.

Go is an extremely difficult game for computers to play well. Four main factors combine to thwart normal computer game-playing AI techniques:

(i) The state space of the game of go is intractably huge.

(ii) The branching factor of go is intractably huge.

(iii) It is difficult to construct a heuristic function to evaluate board states.

(iv) Proper go strategy requires an extremely deep look-ahead.

A standard-sized go board contains 361 points. Each of these points may (in almost all cases) independently take on one of three values: being empty, containing a black stone, or containing a white stone. This puts the state space of go at 3^{361}, which is almost 10^{172}. By contrast, the total number of electrons in the universe is estimated to be only about 10^{79} [10]. The sheer number of possible board positions in the game of go makes it extremely difficult to generalize information about the value of a board position.

At any situation in a game of go, almost all empty points are valid, legal moves. In the beginning of the game, there are over 350 possible moves at each ply. Although the number of possible moves steadily decreases as the game progresses and more points on the board become occupied by stones, the average number of potential, legal moves at any given situation is still on the order of about 200. This is the average branching factor of the game tree. Such a large branching factor, compared to that of chess, for example, makes any form of tactical minimax search computationally infeasible to a depth of more than a few ply.

When analyzing board positions in the game of go, what makes one position better than another is often not well-defined or easily quantifiable [1]. Every stone has the same value, unlike in chess, and the mere

number of a player's stones on the board is a completely ineffective indication of the strength of that player's position. Rather, the patterns described by neighboring (and sometimes distant) stones on the board have complex implications. Often, a small, seemingly inconsequential difference in a sparse pattern can have decisive influence on the ultimate life or death of that group a number of moves down the road [1].

Not only is it computationally intractable to perform a tactical minimax search of the go game tree, it is also prohibitively difficult to design the type of accurate heuristic evaluation function that minimax search and any kind of minimax optimization technique (such as alpha-beta pruning) depends on. In addition, due to the large size of the board, and the fact that groups of connected stones may eventually stretch from edge to edge, a stone played in one area my have a direct strategic effect on another distant area that will not be manifested until many moves have elapsed. An excellent example of this phenomenon is the ladder-breaker, which is a move that is simple enough to be understood by most beginning amateurs, yet when played in one corner will have a strategic impact on the opposite corner that would not be observed for over 40 ply in the game tree. Thus, not only is it intractable to search the game tree and infeasible to accurately evaluate the states of the board, as the computer does when playing chess, but it is also necessary to search the tree to a depth much greater than is required for chess. All of these factors combine to foil any attempt at creating a strong computer go player [9].

2.1 Motivation for Computer Go

Chess and go are often seen as the two most challenging and most interesting strategy games in the world. The best chess-playing computers can compete evenly with the best human players in the world. Even a generic chess program on a home computer is generally better than most amateur players [9]. However, even the most advanced computer go algorithms can only compete with an amateur human player who has studied the game for a mere 6 months. Millions of humans, including young children, can play go at a substantially higher level than a computer.

Since the traditional AI paradigms used in computer chess cannot, for the reasons described above, be applied to computer go, alternative AI methods must be found and developed. Human techniques for playing go seem to involve complex pattern recognition as well as strategic value judgement. An effective algorithm

for computer go could represent a significant advance in the field of artificial intelligence, not just in the areas of computer game playing.

2.2 Similarities to Computer Backgammon

The game of backgammon provides an interesting analogy to the game of go. Backgammon is played with 15 pieces of each color (black and white) on a board with 26 locations. Multiple pieces of the same color may occupy the same location simultaneously. Although two pieces of opposite color may not occupy the same space simultaneously, the game still possesses a state space of roughly $30^{26} \approx 10^{38}$ positions. While not as large as the state space of go by any means, the state space of backgammon is still too intractably large for a computer to deal with each state individually [2].

Backgammon also includes an element of chance. On one's turn, one rolls a pair of dice, and moves one or two pieces along the board by the value shown on each die. On average, there will only be about 20 possible moves to consider. However, in searching the game tree, one needs to consider all possible dice rolls for one's opponent's moves and for one's own subsequent moves. There are $\binom{6}{2} = 15$ possible dice rolls, yielding a branching factor of close to 300 - similar to the branching factor of go. One can see that it is just as infeasible to perform a tactical minimax search in backgammon as it is in go [2].

Beginning in 1992, Gerry Tesauro applied the techniques of Temporal Difference (TD) learning to the game of backgammon, and produced a computer program which taught itself to play backgammon with great proficiency [2]. This paper is an investigation into the application of similar techniques to the game of go, in an effort to produce a computer program capable of teaching itself to play the game with reasonable skill.

3 Approach

TD-Gammon uses a combination of several techniques from the field of artificial intelligence to learn backgammon. This section will examine those techniques - how they work, how they were applied to backgammon, and how they will be applied to go.

3.1 Neural Networks

The primary goal of TD-Gammon, and the primary goal of this paper, is to produce a function that maps a vector describing the state of the board to a scalar describing the value of that board state. This scalar takes on real values between 0 and 1, and roughly corresponds to the probability that black will win the game from this board state. This function is certainly non-linear, and almost certainly extremely complicated. One would like to describe this value function by a giant lookup table - mapping every possible input vector to a unique output value. This model could obviously be made to represent all possible value functions, regardless of complexity, with no limits on precision. However, since the state space of both go and backgammon is so intractably huge, modeling the value function by a simple lookup table would require too much storage space and too much time to be feasible. Some sort of function approximation technique is required.

A neural network is a machine learning technique that can be used to approximate a complicated, non-linear function. There are several types of neural networks in the field of AI. Feedforward networks of perceptrons are by far the most common, and are the type of neural networks used in Tesauro's TD-Gammon [2]. This is also the type of neural network used in this paper.

3.1.1 The Perceptron[1]

The perceptron is the basic unit of a neural network. Roughly speaking, it is a simplified computational model of the biological neurons in the human brain. A perceptron has a certain number of inputs, each with a corresponding weight, and a single output. To calculate its output value, the perceptron takes the weighted sum of its inputs and passes this value through a non-linear transfer function. The transfer function must be non-linear, or else a multilayer network of perceptrons would simply collapse into a single linear function.

Any non-linear function can be used, but in practice the sigmoid function:

$$y(x) = \frac{1}{1 + e^{\alpha x}} \tag{3.1}$$

is often used.[2] This function of $\mathbb{R} \to \mathbb{R}$ produces a continuous output between 0 and 1. Additionally, the

[1] This section essentially follows [3].
[2] In this paper, α is chosen to be -5.

derivative of the sigmoid function:

$$\frac{dy}{dx} = y(1 - y) \tag{3.2}$$

is easily calculable, a feature which will be useful for training larger networks of perceptrons.

Generally, a perceptron has one additional input, called the bias. Conceptually, although it is often referred to differently from the other weights, the bias is really nothing more than an extra weighted input from a source that is always 1. The bias allows the perceptron to output an arbitrary value even when all its actual inputs are zero.

Mathematically, the output of a single perceptron which takes $\vec{x}$ as input, has $\vec{w}$ weights, b bias, and $y(t)$ transfer function will produce

$$out = y(\vec{x} \cdot \vec{w} + b) \tag{3.3}$$

as output.

A single perceptron can be trained to produce a certain output for a corresponding input by adjusting its input weights appropriately. By adjusting the weights in small increments at each step (nudging the perceptron toward the desired output), and by repeatedly presenting the perceptron with many different example input/output pairs from across the input domain, a perceptron can be trained to approximate a function over its entire domain.

3.1.2 Multi-layer Perceptron Networks[3]

A single perceptron is severely limited, however, in the complexity of function approximation that it can provide. In order to achieve a more precise approximation, perceptrons can be connected together to form multi-layer networks, with the outputs of some perceptrons connected to the inputs of others.

[3]This section essentially follows [3].

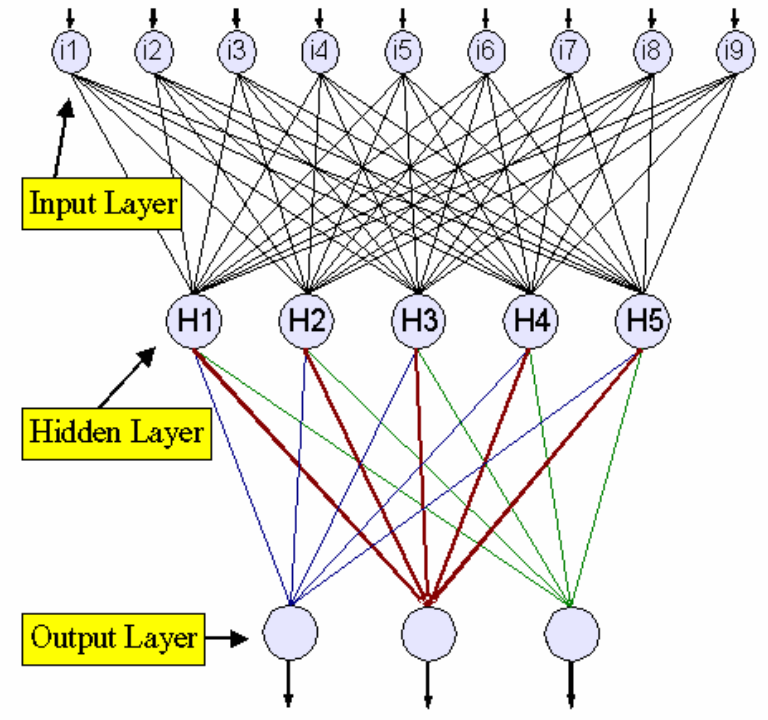


Figure 3.1: A Multi-Layer Perceptron Network

In a feedforward neural network, perceptrons are divided into layers, with the outputs of every perceptron in each layer connecting as an input to every perceptron in the subsequent layer. The outputs of the perceptrons in the final layer comprise the output vector of the network. The initial layer of the network is composed of a number of dummy nodes, which simply take on the values of the components of the input vector. Since every perceptron in the network takes input from the perceptrons in the layer before it, the output of the network is calculated sequentially by first calculating the outputs of the perceptron in the first layer, and feeding these values forward as inputs to the second layer, etc. The layers between the input and output layers are referred to as *hidden* layers, as their outputs have no direct effect on the observed output of the network.

Feedforward networks can be trained in a similar manner to a single perceptron. Instead of simply adjusting the input weights, though, the error on the output of the final unit is propagated backward through the network. All of the weights are updated in the direction of the error gradient, to a degree corresponding to their respective contributions to the error.[4]

[4]Backpropogation algorithm from [4].

The total error of the network is:

$$E = \frac{1}{2} \sum_{i \in output} (target_i - act_i)^2 \tag{3.4}$$

where *output* represents the set of nodes (perceptrons) in the output layer, $target_i$ is the desired output value of node i, and act_i is the actual output value of node i. For each node, a δ value is calculated, which represents that node's individual contribution to the overall error of the network. For output nodes:

$$\delta_i = -\frac{dE}{d\ inputs_i} = (target_i - act_i) \cdot act_i \cdot (1 - act_i) \tag{3.5}$$

Hidden nodes do not have target values. Therefore, for hidden nodes, δ is calculated as a function of the successor nodes ($succ_i$ is the set of nodes which take input from the output of node i).

$$\delta_i = -\frac{dE}{d\ inputs_i} = \left(\sum_{j \in succ_i} \delta_j \cdot w_{i \to j} \right) \cdot act_i \cdot (1 - act_i) \tag{3.6}$$

$w_{i \to j}$ is the weight on the connection from node i to node j. The δ values must be calculated in reverse order to the network flow, as δ of a hidden node depends on δ of its successors. Once all δ values have been calculated, the weights of each connection can be updated. The weight on the connection between node i and node j is updated in this manner:

$$w'_{i \to j} = w_{i \to j} + L \cdot \delta_j \cdot act_i \tag{3.7}$$

and the bias on node i is updated in this manner:

$$b'_i = b_i + L \cdot \delta_i \tag{3.8}$$

In both of these equations, L is a small constant called the *learning rate.* A lower learning rate constant will cause the network to learn more slowly, while a higher constant will cause the network to more quickly forget previous data points.

3.2 TD Learning[5]

Neural networks provide a tractable method for approximating a value function over the state space. Now it is necessary to determine what that value function should be. Temporal Difference (TD) learning is a machine learning technique whereby a computer learns a value function over a state space by walking through the state space many times, and remembering its experiences. Whenever the computer transitions to a new state, it receives a reward r (which may be positive or negative). The optimal value function is one that allows the computer to maximize its total reward, when choosing moves according to the corresponding policy. In most cases, the policy will be to choose the move that leads to the subsequent state with the best value. In this paper, the chosen policy is for black (the maximizing player) to choose the move that leads to the highest subsequent value, and for white (the minimizing player) to choose the move that leads to the lowest subsequent value.

The process of TD learning begins with a randomly initialized value function, $V(s)$. This is the computer's initial estimate of the optimal value function $V^{\pi}(s)$, for some externally defined policy π. The computer learns by repetitively playing games against itself, and revising its estimate of V^{π}. For each game, the computer begins in the initial state s_0, and chooses the action prescribed by π to attain the subsequent state, s_1. The estimated value function is updated based on the computer's observation of state s_1 and another move is selected. This process repeats until a terminal state is reached, and the game ends. At this point, the computer begins a new game, and continues to learn.

When the computer arrives at a state s' upon transitioning from state s, it updates its estimate of the optimal value function. The computer can observe two pieces of information about state s' that refine its estimate: the currently estimated value of $V(s')$, and the reward r obtained by transitioning to s'. The updated value of $V(s)$ is calculated from this data in the following manner:

$$V(s) \longleftarrow V(s) + \alpha \left[r(s') + \gamma V(s') - V(s) \right] \tag{3.9}$$

In this equation, α is the learning rate, similar to the learning rate of the neural network, which controls how quickly the value function estimate is updated. α is generally some small positive number $0 < \alpha < 1$. γ is

[5]This section essentially follows [2].

the discounting factor, which will always be one, unless the TD learning function is updating from additional subsequent states farther in the future (which it is not doing in this paper).

3.3 Tactical Searching[6]

When learning from experience by playing against itself (as described above), the computer follows the simple policy of choosing the move that leads to the immediately subsequent board position of greatest value, according to the approximated value function. This technique will more often than not produce an acceptable (if not ideal) decision, which will, in turn allow the computer to gain experience and likely make a slightly better decision next time it finds itself in a similar position. Additionally, the best move under this policy is fairly inexpensive to calculate, allowing to computer to gain new experiences faster than if more thought was put into move selection.

However, for actual competitive play, either against humans or other computers in tournaments or the like, it is desirable for the computer to put addition effort into its consideration of how to move. If the value function were entirely optimal, then the simple action selection policy described above would be sufficient. However, the value function approximated by the computer in this paper is far from optimal for several reasons. For one, it is an approximation produced by a neural network, and thus likely does not represent every facet of the optimal value function to an acceptable precision. Additionally, even if the neural network's approximation were perfect, the value function it models is still derived solely from the computers own experiences traversing the state space. (The computer obviously cannot have visited every one of the $\approx 10^{179}$ states in the game of go even once, let alone the many times that would be required to synthesize an accurate value assessment.) Because the value function is not optimal, there are situations where additional consideration can improve the quality of the best move selected.

3.3.1 Minimax Search

A tactical minimax search through the game tree to a certain depth is the standard method of computer move selection that uses a heuristic evaluation function of board positions (the approximated value function, in this case). In fact, minimax search to a depth of 1 is exactly the same as the simple move selection policy

[6]This section essentially follows [5].

used during learning. Minimax search to a greater depth can additionally consider the ramifications of the opponent's potential responses to a given move, etc. Depth of tactical search generally appears to be directly correlated to playing strength in all games for which tactical searching is appropriate (like chess).

In a minimax search of depth 2, for example, the computer chooses a move that maximizes the value of the state that if will end up in not only after it makes that move, but after its opponent makes his most effective response. Such a search is conducted by examining the values of the board states resulting from each of the opponent's possible moves in response to each of the computer's possible moves. The move that would produce the maximum value, if the opponent plays to attain the minimum value, is the move that will be selected. A minimax search to a greater depth continues along the same line of reasoning, simply expanding the game tree more deeply.

If at every point the player to move has k moves available (the *branching factor* of the game tree), and the computer desires to search the tree to a depth of n ply, it is required to consider $O(k^n)$ positions. Minimax searching is an exponential-time computation, and therefore intractable to substantial depth. If the branching factor k is very large (as it is in go), brute-force minimax search become intractable much more quickly.

3.3.2 Alpha-Beta Pruning

There is a way to prune the game tree as it being search, and cut off entire sections of the tree from consideration, without at all reducing the optimality of accuracy of an exhaustive minimax search. This pruning technique is called *alpha-beta pruning*. In a nutshell, the alpha-pruning algorithm keeps track of the best position that the player to move can force (based on the parts of the game tree that have already been expanded). These statistics are tracked for both players, not just the player currently deciding. As new positions are examined, they are compared to these known lower bounds. If it is determined that the player to move can force an outcome that is worse for his opponent than his opponent could have forced elsewhere (in the previously examined tree), it can be concluded that his opponent would never choose to move to the current position, and this position and all subordinate positions can be instantly discarded. Alpha-beta pruning obviously produces cutoffs like this more frequently (and is thus more effective) if good

moves are considered first at each level. On average, though, alpha-beta pruning will produce enough cutoffs as to effectively halve the branching factor - allowing the computer to search twice as deeply as it could have using exhaustive minimax search. Since alpha-beta pruning only produces a cutoff when it has been already demonstrated that a better move exists elsewhere, it runs no risk of cutting off the actual optimal move, and thus is purely beneficial to the techniques of minimax searching.

3.4 Synthesis of Techniques

The primary focus of this paper is the synthesis of the above-described techniques of computer game playing into an algorithm capable of playing go with reasonable skill. Temporal Difference learning is used as described to learn an estimated value function through experience. This function is stored as an approximation in a neural net. When learning, the move selection policy is simply to choose the move that produces the best subsequent position with probability $1 - \varepsilon$, or to choose a random legal move with probability ε, for some $0 < \varepsilon \ll 1$.[7] When competing, the move selection policy will use the derived value function to perform a tactical minimax search (augmented by alpha-beta pruning) as deep as is temporally feasible.

4 Implementation

4.1 Board size

Go is generally played on a 19x19 board, although the rules generalize very easily to any nxn board. Most experimentation in this paper uses a 9x9 board, primarily because the smaller board size allows the computer to learn more quickly. Techniques used in this paper should also generalize seamlessly to an nxn board, with at most a quadratic slowdown (except for alpha-beta pruning, which is necessarily exponential).

4.2 Input Vector design

The board state is stored internally in a 2-dimensional array format that makes it easy to generate lists of legal moves, and easy to incrementally apply and un-apply moves to travel up and down the game tree. This

[7] This small probability of choosing a random move guarantees that all moves will eventually be explored, while simultaneously guaranteeing that move selection is predominately intelligent. An appropriate value of ε should be chosen to balance the merits of reinforcement and exploration.

format, although efficient for maintaining the state of the board as the game progresses, is not appropriate for use as input for the neural network.

The input vector to the neural network should contain in individual node for every independent, important feature in the state of the game, even if this results in redundant information [2]. The neural networks used in this paper expect all nodes (including input nodes) to assume continuous real values between 0 and 1.

For this paper, there are 2 nodes in the input vector for every point on the board, in right-to-left, top-to-bottom order from the top left. The first node for a given point is 1 if the point contains a black stone, and 0 otherwise. The second node for a given point is 1 if the point contains a white stone, and 0 otherwise. Clearly it is not possible that a single point in a legal board position could be represented as 11, since no more than one stone can occupy the same point at once. However, the separation of the concepts of black stone and white stone at each point in the input vector allows the neural network to process the effects these stones may have on the board independently [2].

The input vector used in this paper has 2 additional nodes at the end, which contain additional information about the state of the board not germane to any individual point. The first of these extra nodes encodes the color of the player to move: 0 for black, 1 for white. This allows the neural network to evaluate the implications of sente[8] and gote[9] on the value of a board position. The second of these extra nodes encodes the progress of the game. In this paper, the progress of the game is represented as $1 - e^{-n}$, where n is the total number of moves that have been made. For $n \geq 0$, which will always be the case, this function takes on real values between 0 and 1. This node of the input vector allows the value function to assign two similar positions different values based on whether they occur early or late in the game, and thus allows the computer to potentially develop separate opening, midgame, and endgame strategies.

4.3 Neural Network Topology

The design of the neural network's topology has a dramatic impact on its ability to accurately approximate an arbitrary function. All feedforward neural networks have roughly the same topology: an input layer that

[8] *Sente* is the property of having the initiative in the choice of moves. A player has sente when his opponent has not just played a move that requires an immediate response.

[9] *Gote* is the opposite of sente. It is the property of not having the initiative in the choice of moves – as when the one's opponent has just played a move that requires an immediate response, such as a threat to capture a large group.

is completely connected to the first hidden layer, which is completely connected to another hidden layer (etc.), which is completely connected to the output layer. The size of the input layer is determined entirely by the size of the input vector used to describe the board state. The size of the output layer is determined entirely by the manner of output desired (in this case, a single scalar value between 0 and 1). Therefore, in this discussion of neural network topology, the size of the input and output layers is fixed at some arbitrary size, and the only variable parameters are width and number of hidden layers.

In this paper, several small experiments were undertaken to determine reasonable constraints on the necessary size of the neural network, and on the complexity of functions that could be approximated by a network of a given size. This investigation was not particularly extensive, and was not meant to be. It was simply intended as a brief exploration into the general behavior of neural networks of varying topologies so that the topology chosen for this paper would not be completely arbitrary. The results of this brief investigation are summarized below.

In general, a larger network seems to be capable of more accurately approximating a complicated function than a smaller network. However, between two networks that are both capable of approximating a given function to the required precision, the larger network will require more time to train. Even if both networks attain the desired precision in the same number of back-propagation training events, the larger network will have had to perform more edge weight updates in each event, and will thus require more total CPU cycles to train.

Figures 4.1 through 4.4 describe the output of a small neural network with a 2-dimensional input vector and a 1-dimensional output vector. The color at each point represents the output value for that point in the input space – red = 1, blue = 0. The pattern that these networks are attempting to approximate is a 6x6 checkerboard of 0's and 1's. Only those 36 points are specified in the pattern. The intervening values are allowed to float, but should ideally assume a smooth contour between the 0 and 1 points. Each row represents a neural network with a different size hidden layer. Each plot represents a different stage of training for each network. From left, these are: 1,000; 10,000; 100,000; 200,000; 500,000; and 1,000,000 iterations.

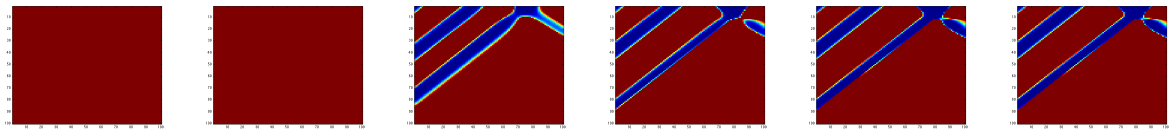


Figure 4.1: 8 Nodes in Hidden Layer

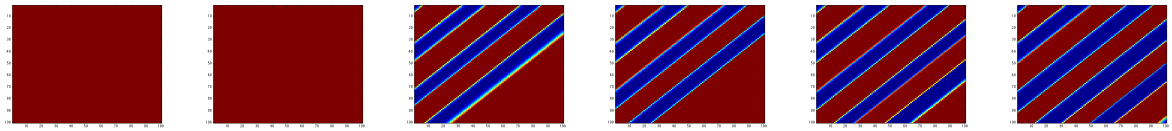


Figure 4.2: 16 Nodes in Hidden Layer

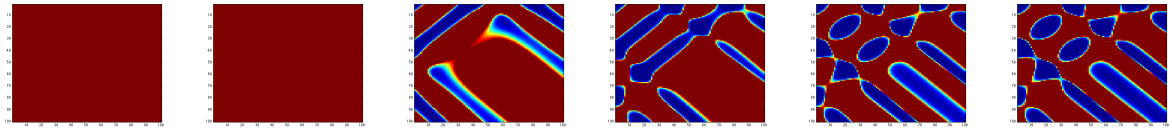


Figure 4.3: 32 Nodes in Hidden Layer

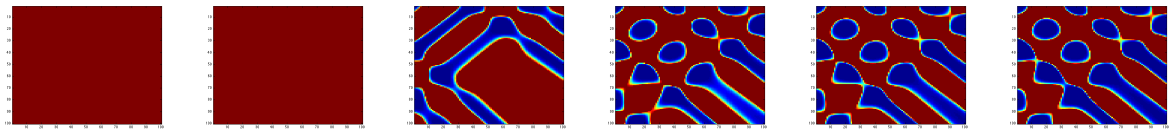


Figure 4.4: 64 Nodes in Hidden Layer

The 8-node neural network in the first row is seemingly incapable of rendering the desired pattern. The 16-node network in the second row is capable of correctly evaluating the 36 target points, but cannot construct a well-formed checkerboard in the intervening space. The 32- and 64- node networks are both capable of correctly modeling the 36 target points, as well as the intervening checkerboard pattern.

Clearly the goal is to find the optimal neural network size that balances training speed with approximation complexity. This is not something that is easy to determine, even empirically, without exhaustive calculation, and it is possible that a better network topology would have improved the final results of this paper. A less daunting task, however, is determining the total number of hidden layers that should be included. The question here is, for a given total number of hidden nodes, is it more effective to arrange them into 1 layer

of width n, or 2 layers of width $\frac{n}{2}$, or many layers of width $\frac{n}{k}$? Because of the way the back-propagation training algorithm works, networks with more than one hidden layer seem to take considerably longer to train that networks with a single, large, hidden layer.

For this paper, a neural network with a single hidden layer was ultimately chosen. In an attempt to balance training speed for approximation accuracy, the width of the hidden layer was chosen to be roughly equivalent to the width of the input layer. A 9x9 board contains $9^2 = 81$ points, which results in an input vector of size $2 \cdot 81 + 2 = 164$ nodes. For a 9x9, board, a hidden layer width of 200 was arbitrarily chosen.

4.4 Priming

The transfer function used in the perceptrons making up the neural network used in this paper is the sigmoid function $y(x) = \frac{-1}{1+e^{-5x}}$ described in equation 3.1. This function (left) and its derivative (right) are depicted in figure 4.5.

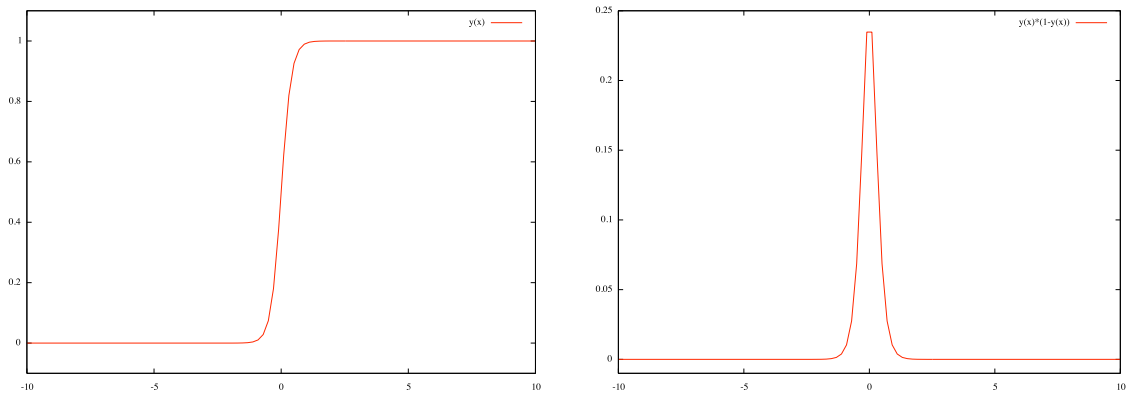


Figure 4.5: The sigmoid function and its derivative

As described in Equation 3.5 and Equation 3.6, the δ value of a node, is dependent upon the derivative of the sigmoid function on that node. Since the derivative of the sigmoid function falls off to near-zero very quickly outside of the range where the output of the sigmoid is near $\frac{1}{2}$, nodes in the network outputting values near $\frac{1}{2}$ will change more quickly under back-propagation.

If the current output of the neural network is far from 0.5 (say it is close to 1), then the network will be slower to react to an error of a given size than it will be if the network output is closer to 0.5. Additionally,

if the current output of the network is near 1 and the desired output is near 0, for example, the network will take *much* longer to reach the desired output than it would if its current output was near 0.5, not only because of its slower response at the edge of the range, but also because it has farther to travel.

It is clearly desirable for the neural network in general to react more quickly to training, without increasing the learning rate constant (which would cause it to forget old patterns more quickly as well). Additionally, it is reasonable to assume that there are roughly an equal number of board positions favorable to white as there are favorable to black,[10] and thus there is an equal probability that an unknown position will favor either player. Therefore, it seems that the performance of the neural net would be enhanced if it was initialized to the value 0.5 for all inputs.

In this paper, neural nets are generated in the following manner. First, edge weights are chosen uniformly at random between $\frac{-1}{2}$ and $\frac{1}{2}$. Then, the network is trained for several thousand iterations with random input vectors, all with a target output of 0.5. This procedure produces a semi-randomized neural network that generally produces outputs near 0.5. It is unclear how effective this "priming" technique is overall, but it seems to expedite the training of the network at least to some small degree, and also seems to produce an even distribution of output values centered around 0.5, as would be desired.

5 Results

In this section, the approach to computer go described in this paper is tested and evaluated against several benchmarks to determine its overall effectiveness.

5.1 Effectiveness of Training

More fundamental than testing this approach's skill at playing go is testing whether or not the training has produced a noticeable improvement. If this were not the case, then the performance characteristics of this algorithm must derive from eccentricities of the implementation, and not from the underlying technique. This section compares progressively-more-trained versions of the neural net against each other, in demonstration that increased training actually does improve go-playing performance.

[10] Simply reversing the colors of all stones, and reversing the player to move, creates a bijection between all positions favorable to black and the obviously equal number of positions equally favorable to white.

Table 5.1 shows the performance of 4 different versions of the same initial neural net at different stages of training. Each version is identified by the total number of plies on which it has been trained.[11] Network 0 is the original untrained neural network. Network 5,394,209 is the most highly trained neural network produced in this experiment.

Black	White	Score
5,394,209	0	90 - 10
5,394,209	1,331,144	42 - 58
5,394,209	2,663,064	25 - 75
0	5,394,209	29 - 71
1,331,144	5,394,209	37 - 63
2,663,064	5,394,209	43 - 57

Table 5.1: Performance Against Self Over 100-Game Tournaments

As can be seen in the table, increasingly-more-trained networks perform increasingly well against Network 5,394,209. The apparent advantage to seen by white is probably due to the komi[12] being set higher than black's skill warrants.

5.2 Skill of Play

GNU Go is an open-source computer go engine. Version 3.6 of GNU Go was released in 2004, and has consistently achieved a ranking of about 9 kyu[13] in online play with humans on internet go servers. In this paper, GNU Go 3.6 is the benchmark used to represent the performance of existing computer go algorithms [6].

GNU Go defeats this paper's approach in every game it plays, even on GNU Go's lowest difficultly setting.

This is not necessarily indicative of a fundamental flaw in this paper's approach, simply a few inadequacies in the implementation. Further training may improve the value function to the point that it becomes more competitive. It is also possible that the chosen neural network topology cannot approximate the complexities of the value function precisely enough to play well, and that a larger topology is also required. Additionally, even with alpha-beta pruning, this approach was only able to search the game tree to a depth of 3 plies in

[11] On a 9x9 board, a single game seems to consist of roughly $\approx$ 100 plies.

[12] Komi in this paper is set at 6.5.

[13] The *kyu* is the unit of beginning amateur go ranking. A human beginner usually starts out around 30 kyu, and most people can achieve a single-digit rank with about a year of play and study. 8 or 9 kyu is about the best ranking that a computer algorithm has yet reached. Above 1 kyu is the amateur *dan* ranks, a level which computers have never attained.

a reasonable amount of time. This is simply not deep enough to be an effective tactical search, even with a near-optimal value function.

5.3 Value Function

This algorithm may perform poorly in actual competition against GNU Go, but the concept of applying TD-learning and neural networks to learn an approximate value function for the game of go seems to have been reasonably effective. Figure 5.1 shows the evolution of the computer's value estimate throughout several games. The first set of graphs corresponds to a game played between Network 5,394,209 as black and Network 5,394,209 as white in which black eventually won. The second set of graphs corresponds to a game played between the same network, in which, this time, white eventually won. The last graph corresponds to a game played between Network 5,394,209 as black and a reasonably knowledgeable human[14] as white, in which white handily won.

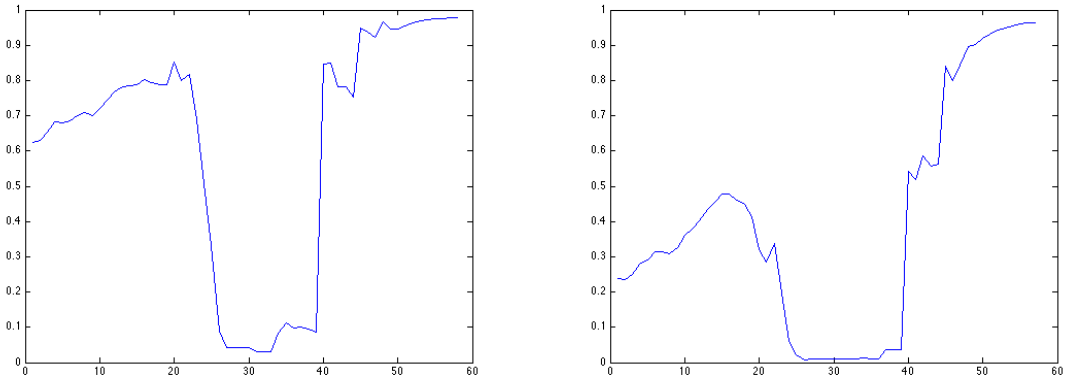
These graphs seem to indicate that the computer has learned a fairly accurate evaluation function for estimating the value of board states. In both games played between computer players, neither player took a commanding lead until the end, and thus the value function fluctuated for most of the game. In the game played between the computer and a human, the computer realizes fairly early, in comparison, that it is losing badly.

6 Conclusion

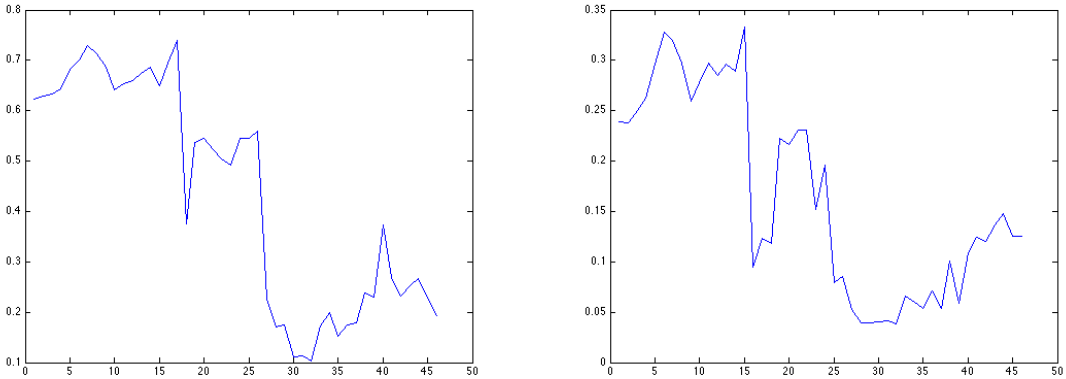
The approach described in the paper seems effective at producing an approximation of a value function that can be used to play go. However, it is also evident that this value function is not by itself sufficient for strong play. This value function proves inadequately precise to allow a depth-3 tactical search to select strong moves. Either a deeper search or a smarter value function is necessary for this approach to have merit.

A deep full-board tactical search of the game tree is just as impractical with this value function as it is with any other heuristic. Improving the value function, then, is the primary manner in which the techniques described in this paper could be extended or improved.

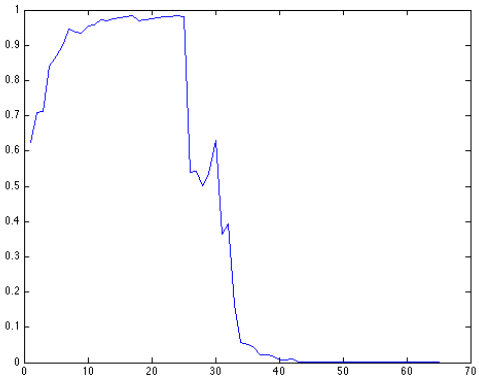
[14]The author.



(a) 5,394,209 (black, left) vs 5,394,209 (white, right) – Black Victory



(b) 5,394,209 (black, left) vs 5,394,209 (white, right) – White Victory



(c) 5,394,209 (black) vs. Human (white) – White Victory

Figure 5.1: Value of Board Positions Throughout Several Games

There are several ways to possibly improve the performance of the value function generated in this paper. Their relative efficacies depend on the limiting factor of this approach. The observation that Network 5,394,209 performs better than the less-trained networks would seem to indicate that continued training will produce continued improvement. In this case, simply allowing the learning algorithm more time to play itself should improve the value function. Network 5,394,209 was trained over 40,000 games. TD-Gammon, on the other hand, was generally trained over 100,000 to 1,000,000 games [2]. It would be reasonable to expect that a value function for the game of go would require at least as much, if not more training than one for backgammon.

If more training at some point fails to produce continued improvements, it is likely that the neural network being used is too small to precisely render the necessary complexities of the value function being learned. In this case, improvement can again be obtained by increasing the size of the neural network and retraining.

Given the improvement demonstrated during training over only 40,000 games in this paper, additional improvement beyond this point is expected upon additional training.

References

[1] A. Hollosi, M. Pahle, et. al. "Sensei's Library," [Online Document], 2007 Jan 6, [cited 2007 May 8], Available HTTP: http://senseis.xmp.net/

[2] R. Sutton and A. Barto. Reinforcement Learning: An Introduction, Cambridge: MIT Press, 1998.

[3] A. Blais and D. Mertz, "An introduction to neural networks," [Online Document], 2001 Jul 1, [cited 2007 May 8], Available HTTP: http://www-128.ibm.com/developerworks/library/l-neural/

[4] Z. DeVito "Handwriting Recognition," [Unpublished Document], Princeton University, 2006.

[5] S. Russell and R. Norvig. Artificial Intelligence: A Modern Approach – Second Edition, Upper Saddle River, NJ: Pearson Education, Inc., 2003.

[6] Free Software Foundation. "GNU Go," [Online Document] 2006 Nov 23, [cited 2007 May 8], Available HTTP: http://www.gnu.org/software/gnugo/gnugo.html

[7] IBM. "IBM Research — Deep Blue — Overview," [Online Document], [cited 2007 May 8], Available HTTP: http://www.research.ibm.com/deepblue/

[8] G. Fox, R. Williams, and P. Messina. Parallel Computing Works, [Online Document], 1994 [cited 2007 May 8], Available HTTP: http://www.netlib.org/utk/lsi/pcwLSI/text/node342.html

[9] Intelligent Go Foundation, "Overview of Computer Go," [Online Document], 2000, [cited 2007 May 8], Available HTTP: http://www.intelligentgo.org/en/computer-go/overview.html

[10] "1, 2, 3, 4 - four digits that DWARFED the universe," [Online Document], 1994, [cited 2007 May 8], Available HTTP: http://www.mathsnet.net/articles/article_1234.html

A The Game of Go

The Game of Go originated in China over 3000 years ago. Throughout its entire history, go has been one of the most popular and most played board games in the world. The rules of go are few and simple, yet the strategy required to play well is extremely complex [1].

A.1 Rules of the Game

Go is a board game for two players. The board is square, and is composed of criss-crossing lines, the intersections of which form *points* on which to place stones. Simply put, the object of the game is to surround territory (empty points) with stones of one's own color. In general, a player scores a point for each square of empty territory that is entirely surrounded by his stones. An example of a 5x5 board is depicted in Figure A.1. Most games of go are played on a standard-sized 19x19 board, although new players often begin learning on a smaller 9x9 or 13x13 board. The standard size of 19x19 is not a wholly arbitrary number. This board size provides a stable balance between the amount of territory along the sides of the board (which would be higher on smaller boards) and the amount of territory in the center (which would be higher on larger boards). The strategic significance of this fact will be discussed later.

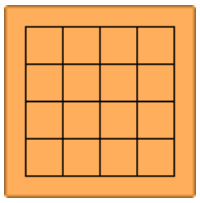


Figure A.1: An Empty 5x5 Go Board [1]

A.1.1 The Play

The players, black and white, alternately place stones of their color on the points of the board, one at a time. Traditionally, black is always the first to move. To offset the advantage of the first move, white is generally granted some small amount of compensation points (called *komi*), which is typically a non-integral value in order to prevent ties. Except for a few small restrictions discussed below, a stone may be played on any open point at any time. A player may also choose to pass, and not play a stone, at any turn. A move

of a single player is referred to as a *ply.* An example sequence of 6 ply is depicted in Figure A.2.

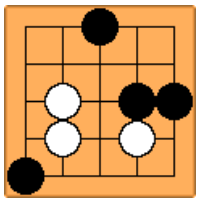


Figure A.2: Play of Stones [1]

Once played, a stone is never moved or removed from the board, unless it is captured. A stone may be captured (and removed from the board) if it is entirely surrounded by stones of the opposing color. A stone thus surrounded is said to be *dead.* Only the four cardinal directions are of consequence in determining the life of stones. Figure A.3 may help explain this rule. In the figure, the white stone is connected to a single open point (indicated by the small red square) by the grid lines on the board. This point is referred to as a *liberty* of the white stone. In contrast, the four points indicated by the small red circles are not liberties of the white stone, as they are not adjacent to it in one of the four cardinal directions. The white stone in this figure has only one liberty, and is thus said to be in *atari.* A black play at the point marked by the small red square would remove the last liberty from the white stone, capturing it and removing it from the board. Stones that have been captured are referred to as *prisoners.* Prisoners must be immediately removed from the board as soon as they are captured.

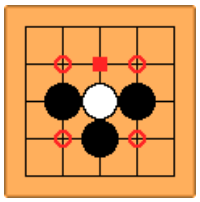


Figure A.3: Liberties [1]

Stones of the same color that are adjacent in one of the four cardinal directions are called *connected.* Connected stones share liberties, and thus share the same fate. A set of connected stones is referred to as a *group.* Figure A.4 presents an illustration on the life of groups. The left board is the initial position. The three white stones indicated by red circles are connected, and (before black's play at ❶) share a single liberty

at ❶. Black's play at ❶, however, removes that liberty, and thus captures the entire white group.

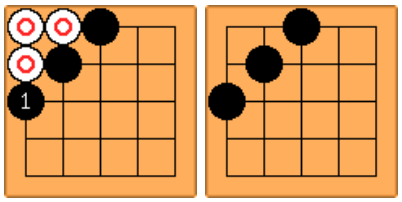


Figure A.4: Capturing a Group of Stones [1]

Typically, one may play a stone on any open point of the board on one's turn. However, there are two rules which slightly restrict the points on which one may legally play. The first of these rules is the suicide rule. A *suicide* is a move that places a stone on the board in such a way that it has no liberties, and is thus instantly captured. Such moves are illegal under the rules of go. Figure A.5 illustrates this scenario. In the board on the left, black may not play at ❶, because his stone would have no liberties, and the entire group would be instantly captured. It is important to note that, in the board on the right, white *is* allowed to play at ①, even though it would appear that his stone would have no liberties, because his play would capture the two black stones to the right of ① and thus create a liberty.

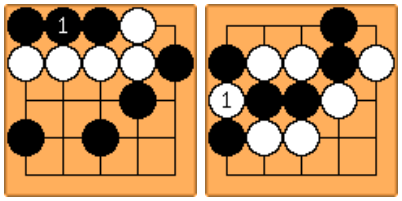


Figure A.5: Suicide [1]

The final rule restricting the placement of stones is called the *ko* rule. This rule is intended to prevent a situation, called a ko, from developing in which both players alternately recapture the same stone ad infinitum. Such a situation is illustrated in Figure A.6. Starting at the position indicated by the upper board, if white plays at ① (capturing the black stone) to attain the position indicated by the leftmost board, black could in theory immediately recapture at ❷ to attain the position indicated by the rightmost board. Because this is the same as the initial position, the above sequence could be repeated forever - and would be if the point in question were of vital tactical significance (e.g. determining the life or death of a large

group). To prevent such an infinite game from developing, the ko rule prevents a player from making a move that repeats the immediately previous board position. Thus, if black wishes to play at ❷, he must first play elsewhere, and wait for white to respond, thus altering the overall board position.

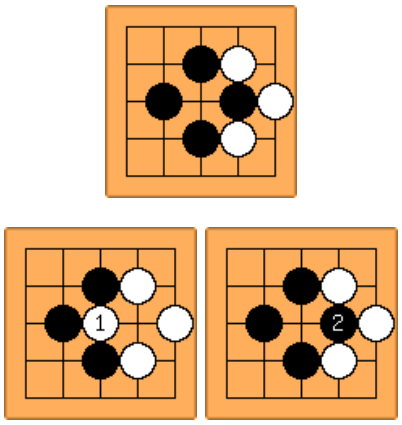


Figure A.6: Ko [1]

A.1.2 Scoring and the Endgame

A game of go is over when both players pass consecutively. This may occur at any point, although as a player will generally only pass when he sees no possible moves that improves his position, it is likely that when both players simultaneously choose to pass the game has been resolved. Once the game has ended, it is necessary to count the score. Before scoring, dead stones (stones that remain on the board because they have at least one liberty, but have such insufficient space that they cannot possibly survive an attack) are agreed upon and removed as prisoners. It is not necessary to play out the capture of these stones unless there is a disagreement.

There are two main techniques used to count score in go: Japanese rules and Chinese rules. Under Japanese rules, a player receives 1 point for each square of territory surrounded entirely by his stones, and 1 point for each prisoner he has captured. Under Chinese rules, a player receives 1 point for each square of territory surrounded entirely by his stones, and 1 point for each of his stones on the board. In both cases the player with the most points wins.

The two are fairly similar, and almost always produce the same result (at least in terms of the difference of scores). The most substantial way in which the two scoring systems differ is that Japanese rules *punish*

a player for playing inside his own territory, while Chinese rules do not. For the purposes of this project, I have decided to use Chinese rules, because I do not wish the learning algorithm to be unnecessarily penalized for playing too defensively.

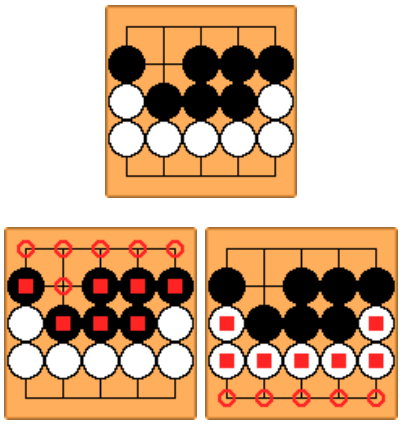


Figure A.7: Scoring [1]

Figure **??** illustrates the method for counting score under Chinese rules. The game has ended at the position shown on the top board. The left board indicates the territories and stones which contribute points to black. Black has 6 points of territory (indicated by red circles) and 7 stones (indicated by red squares) for a total of 13 points. White (as indicated in the board on the right) also as 7 stones, but only 5 points of territory, and thus scores only 12 points.

This paper represents my own work in accordance with University regulations.