Attempting a Machine Learning Solution for Scoring the Game of Go

Anthony Burzillo
Johns Hopkins University
2914 North Calvert St.
Baltimore, MD 21218, USA
aburzil1@jhu.edu

Abstract

This document explains how the author generated features and designed a learning algorithm to score the final board in the game of Go using the RPROP update method described in (Riedmiller and Braun, 1993). Furthermore, the reader will observe that the algorithm was inadequate to solve the scoring problem.

1 Introduction

Go is a board game typically played on a 19×19 board between two players, white and black. Each player, starting with black, successively places stones on the intersections of the board until either both of the players agree that the game is over or until one player resigns. The outcome of the game is determined by the number of empty intersections that each player has surrounded. These empty intersections are known as the territory surrounded by the player.

Since the board is a grid, each stone, except possibly on the edge of the board, has 4 intersections that it connects to. We call nonempty intersections adjacent to blocks liberties. When for any stone all liberties are covered by opponent stones then we say that the stone is captured so it is removed from the board and counted as a point for the opponent. If two or more stones are connected, they can be captured by covering all of the liberties of the collective group by opponent stones.

No player is allowed to commit suicide, i.e. placing a stone such that it's block would have no liberties, unless the move would cause some opponents stones to have no liberties, in which case the opponents stones would be captured. Due to this fact, each player can construct groups that cannot be captured by surrounding two unconnected territories within the group. Then the opponent cannot possibly cover the liberties in both territories without committing suicide. We call each of these nonconnected territories eyes and any group with two or more eyes is alive while a group with fewer than two eyes is dead.

The main problem of scoring is determining the life and death of groups of stones on the board. This is a nontrivial problem because the life and death of groups is not always obvious. For instance one position that can occur is when two groups of different colors each have only two liberties but the liberties are shared between the groups. Then if one player were to play in one of those liberties the other player would be able to immediately capture that player's stones. This situation is called a seki, and commonly occurs in games. The problem that this example shows is that the question of life and death is not local, instead it depends largely on the area around the group. If either player were able to capture one of the groups around his stones stuck in seki he will be able to capture his opponents stones.

To actually score a board all that is needed to do is to (1) determine life and death of all groups, (2) remove dead groups from the board, (3) sum up all territory that is completely surrounded by each player, (4) add the points from captured stones to each players score, and (5) add to white's score the komi, i.e. the points awarded to him for having to take the sec-

ond turn. This problem is hard to do with a deterministic algorithm constructed by a human, so therefore it naturally lends itself to a machine learning solution.

2 Background

Go has been studied in the computer science field since at least 1964 when *A partial analysis of Go* (Thorpe and Walden, 1964) proved that it is impossible for white to force black to lose. Since then it has remained popular in the AI community especially now as chess has been solved, whereas computers still cannot win against reasonably strong amateur players.

The study of of life and death for groups has been around since at least 1976 when *Life in the Game of Go* (Benson, 1976) derived rules to determine whether a group has unconditional life. Examples of algorithms that determine the life of a group can be found in *Static analysis of life and death in the game of Go* (Chen and Chen, 1999) and *Search versus Knowledge for Solving Life and Death Problems in Go* (Kishimoto and Müller, 2005). In fact, there is even a paper on sekis, *Recognizing Seki in Computer Go* (Niu et. al., 2006).

Machine learning has been used in many Go related applications. *Reinforcement Learning of Local Shape in the Game of Go* (Silver et. al., 2007) attempts to study the local shapes of blocks to determine the best move to play when encountering certain position. *Learning to score final positions in the game of Go* (van der Werf et. al., 2004) describes an algorithm to learn to score the final board in Go. Finally, *Learning to predict life and death from Go game records* (van der Werf et. al., 2005) describes an algorithm to learn to determine the life or death of blocks during the execution of the game.

3 Implementation Details

In the paper *Learning to score final positions in the game of Go* (van der Werf et. al., 2004), the authors describe how they were able to craft an accurate scorer for go using manually labeled life and death information for groups in final positions. This is tedious since the lack of an accurate scorer disallows easy labeling. Therefore, in this project the author tried to develop a scorer that did not rely on this specific to group labeling but instead depends only on the labeling of the final score. Then an energy function could be created which determined the accuracy of the estimator versus the actual final score. This implementation is an attempt to do just that.

First we define a block. A block is a group of locations on the board that are directly connected and share the same state, i.e. each location in the block is on an intersection adjacent to another location in the block and they share the same state. Specifically, a block's state can be `EMPTY`, `BLACK`, or `WHITE`.

3.1 Board Representation

The standard record for electronic Go records is the `sgf` file. This file is formatted in a specific way that allows various information such as location, final score, and handicap as well as the moves played in the game. Because of the diversity of possible game files we determined that the easiest way to read these into C++ was in a more constant format. Therefore we created a parser in Haskell using the Parsec parser combinator library to convert the file to a constant header of pertinent information and then the integer locations of moves.

It is important to note that since games are stored as a sequence of moves and since groups of stones can be killed and then removed from the board, allowing play in a previously occupied position, that the entire game must be played in order to create the final board. Therefore it was necessary to create a program that could run a Go game itself. To this pursuit a good representation of the board was necessary.

In order to create a good representation, we must be able to preserve the block membership of locations. To this end, we represented a board as a two dimensional array of pointers to blocks, and then pointed all locations to an initial block to which we set the state to `EMPTY`. The blocks themselves are just sets of locations, since they take up relatively small space and allow faster to iteration through the block than a similar way through the array.

We assume that game locations are correct, i.e. that the `sgf` files do not place stones on top of already placed stones. Therefore, to place a stone, we simply lookup the location's current block and remove the location from that block and create a new block for the stone. Then for each location di-

rectly adjacent to the stone, we lookup the blocks and check if they are friendly. If so we combine the stones current block and the adjacent block. Finally, for each nonempty block touched in this way, we update the liberties. If in this process a block has zero liberties, the block is set to be empty and the size of the block is added to the opponent's score.

After all of the moves have been played, the final step is to recombine the locations empty blocks into new empty blocks. This is needed because the algorithm never splits up an empty block when stones are placed within its borders such that the block's locations are physically split, so without this step there is possibly an empty block with only the locations 0×0 and 0×19, which violates the definition of a block.

3.2 Feature Generation

Feature generation turns out to be a costly process for this application. We need to generate features for each block of stones in the final position. Since we know that the life of any block depends not only on the state of the block but also the state of the board surrounding the stone, we must make sure to include features that describe both states. In this program we chose to use a slightly different feature set then that in (van der Werf et. al., 2004). Specifically we include 46 features for each block. First the local features

- The size of the block.
- The perimeter of the block, i.e. the number of intersections directly next to but not in the block.
- The number of opponent stones in the perimeter of the block.
- The liberties of the block.
- The liberties of the block that the opponent should not play, i.e. liberties which if played by the opponent would cause his stones to have fewer than 1 liberty.
- The liberties of the block that, if played by the player, would cause the block to have 1 liberty.
- The number of second-order liberties, i.e. empty intersections directly next to the liberties of the block that are not liberties.
- The number of third-order liberties, i.e. liberties of second order liberties.
- The number of opponent blocks that are adjacent to the block.
- The center of mass, i.e. the average distance of stones from the first and second closest edge of the board.
- The number of interior points of the smallest bounding box of the block.

Then features for color enclosed territories, i.e. territories surrounded by all one color, adjacent to the block

- The number of color enclosed territories.
- The size of the color enclosed territories as a sum.
- The perimeter of the color enclosed territories.
- The center of mass of the color enclosed territories.

Then features for eyespace adjacent to the block, which we define to be a color enclosed regions with a size less than 5

- The number of eyespace blocks.
- The size of the eyespace.
- The perimeter of the eyespace.

Next we include features for disputed territories adjacent to the block, i.e. territories that are adjacent to blocks of both colors.

- The number of disputed territories.
- The liberties touching the block in the disputed territories.
- The liberties of all friendly blocks in the territories.
- The liberties of all enemy blocks in the territories.

Then features for the optimistic chain of the block. An optimistic chain is a group of blocks that share liberties with each other. These chains represent what blocks a player could potentially connect if needed.

- Number of blocks in the optimistic chain.
- Size of the optimistic chain.
- The perimeter of the optimistic chain.
- The number of color enclosed regions adjacent to the chain.
- The size of the color enclosed regions.
- The perimeter of the color enclosed regions.
- The center of mass of the color enclosed regions.
- The number of eyspaces adjacent to the optimistic chain.
- The size of the eyespace.
- The perimeter of the eyespace.
- The number of disputed territories adjacent to the optimistic chain.
- The liberties touching the chain in the disputed territories.
- The liberties of all friendly blocks in the territories.
- The liberties of all enemy blocks in the territories.

Finally we include the following features for each of the weakest and second-weakest enemy blocks adjacent to the block and the weakest enemy block adjacent to or sharing a liberty with the optimistic chain

- The perimeter of the enemy block.
- The liberties of the enemy block.
- The shared liberties of the block and enemy block (or the chain and the enemy block).

The general algorithm for generating these features for the block `block0` runs as follows:

1. For each location in the block as `location0`:
   (a) Determine the four blocks of the locations next to `location0`.
   (b) For each block as `block`:
      i. Run `calculateFeature(block0, block)`.

Although our algorithm could certainly be optimized further, we were able to rather efficiently compute the features. For instance, when computing the perimeter we were able to also compute the opponents, liberties, adjacent enemy blocks, center of mass, and the bounding box size. Furthermore, we cached the features for chains so that the chain specific features would only need to be computed once per chain.

3.3 Machine Learning

For the machine learning portion of the program we decided to use the RPROP gradient descent algorithm as recommended by (van der Werf et. al., 2004) for being the best update algorithm for determining the life and death of final positions. This turned out to be a poor choice because of the way we defined the scoring and energy functions. Due to the fact that we did not have enough time to manually label the blocks of a large number of games with labeled blocks, i.e. alive or dead, we attempted to define a scoring and energy function that depended only on the game itself and a known final score.

3.3.1 The Energy Function

We will define our energy function E. Suppose that we have a game with blocks $b_1, \ldots, b_n$, and empty territories $e_1, \ldots, e_t$ at the end of the game. and our network's hidden layer has m nodes. Our learning algorithm uses a hidden layer of size m, a matrix of input layer weights W of size $m \times n$, a vector of hidden layer weights $\mathbf{v} \in \mathbb{R}^m$, and two vector biases $\mathbf{a} \in \mathbb{R}^n$ and $\mathbf{b} \in \mathbb{R}^m$. Furthermore, for the second half of the algorithm, we use a matrix of weights V of size 4×5, a vector of hidden weights $\mathbf{g} \in \mathbb{R}^4$, and two bias vectors $\mathbf{c} \in \mathbb{R}^5$ and $\mathbf{d} \in \mathbb{R}^4$.

Suppose we have a game G for which we would like to determine $E(G)$. Let k be the komi, p_b and p_w the prisoners held by black and white, respectively, and the true final score S. Define $\text{tchs}_e(b)$ to be the indicator function taking the value 1 when the

territory e touches the block b. Define $\text{clr}(b)$ to be the indicator function taking the value 1 when the block b is white. Let $\text{size}(e)$ return the size of the territory e. Let $G(x)$ be the indicator function that returns 1 when x is greater than x. Note that when $x \neq 0$, $G(x) = (|x| + x)/2x$. The energy function runs algorithmically as follows:

1. For each block b_i calculate a feature vector $\mathbf{x}_i \in \mathbb{R}^n$.

2. Calculate $r_i = \mathbf{v}[W(\mathbf{x}_i + \mathbf{a}) + \mathbf{b}]$ for each i.

3. For each territory e_i compute

$$N_b^{(i)} = \sum_{j=1}^{n} (1 - \text{clr}(b_j))\text{tchs}_{e_i}(b_j)$$

$$N_w^{(i)} = \sum_{j=1}^{n} \text{clr}(b_j)\text{tchs}_{e_i}(b_j)$$

$$R_b^{(i)} = \sum_{j=1}^{n} (1 - \text{clr}(b_j))\text{tchs}_{e_i}(b_j) r_j$$

$$R_w^{(i)} = \sum_{j=1}^{n} \text{clr}(b_j)\text{tchs}_{e_i}(b_j) r_j$$

4. Construct the vector

$$\mathbf{r}_i = \{N_b^{(i)}, N_w^{(i)}, R_b^{(i)}, R_w^{(i)}, \text{size}(e_i)\}$$

5. Compute $s_i = \mathbf{g}[V(\mathbf{r}_i + \mathbf{c}) + \mathbf{d}]$.

6. Compute $S^* = k - c_b + c_w + \sum_{i=1}^{t} s_i$.

7. We will assume that $r_i \neq 0$. Compute

$$D_b = 2 \cdot \sum_{i=1}^{n} (1 - \text{clr}(b_i))\text{size}(b_i) G(r_i)$$

$$D_w = 2 \cdot \sum_{i=1}^{n} \text{clr}(b_i)\text{size}(b_i) G(r_i)$$

8. Output the energy $|S - (S^* + D_b - D_w)|$.

We define the scoring function to be $S^* + D_b - D_w$. Note that we can write out the energy function algebraically as

$$\begin{aligned}
E(G) &= |E^*(G)| = |S - (S^* + D_b - D_w)| \\
&= \Bigg| S - \Bigg(k - c_b + c_w + \sum_{i=1}^{t} s_i \\
&\quad + 2 \cdot \sum_{i=1}^{n} (1 - \text{clr}(b_i))\text{size}(b_i) G(r_i) \\
&\quad - 2 \cdot \sum_{i=1}^{n} \text{clr}(b_i)\text{size}(b_i) G(r_i) \Bigg) \Bigg| \\
&= |S - k + c_b - c_w \\
&\quad - \sum_{i=1}^{t} \mathbf{g}[V(\mathbf{r}_i + \mathbf{c}) + \mathbf{d}] \\
&\quad + 2 \cdot \sum_{i=1}^{n} \text{size}(b_i)(\text{clr}(b_i) G(r_i) \\
&\quad - (1 - \text{clr}(b_i)) G(r_i))|
\end{aligned}$$

Then we find that

$$\frac{\partial E^*(G)}{\partial \mathbf{g}_i} = -\sum_{k=1}^{t} [V(\mathbf{r}_k + \mathbf{c}) + \mathbf{d}] \frac{\partial}{\partial \mathbf{g}_i} \mathbf{g}$$

$$\frac{\partial E^*(G)}{\partial V_{ij}} = -\sum_{k=1}^{t} \mathbf{g} \left(\frac{\partial}{\partial V_{ij}} V \right) (\mathbf{r}_k + \mathbf{c})$$

$$\frac{\partial E^*(G)}{\partial \mathbf{c}_i} = -\sum_{k=1}^{t} \mathbf{g} V \frac{\partial}{\partial \mathbf{c}_i} \mathbf{c}$$

$$\frac{\partial E^*(G)}{\partial \mathbf{d}_i} = -\sum_{k=1}^{t} \mathbf{g} \frac{\partial}{\partial \mathbf{d}_i} \mathbf{d}$$

Furthermore, since $dG(x)/dx = 0$ then when $r_i \neq 0$

$$\begin{aligned}
\frac{\partial E^*(G)}{\partial r_i} &= \sum_{k=1}^{t} \mathbf{g} V \frac{\partial}{\partial r_i} \mathbf{r}_k \\
&= \sum_{k=1}^{t} \mathbf{g} V \left\{ 0, 0, \frac{\partial}{\partial r_i} R_b^{(k)}, \frac{\partial}{\partial r_i} R_w^{(k)}, 0 \right\}
\end{aligned}$$

Therefore,

$$\frac{\partial E^*(G)}{\partial \mathbf{v}_i} = \sum_{k=1}^{t} \mathbf{g} V \{0, 0,$$
$$\sum_{j=1}^{n} (1 - \text{clr}(b_j)) \text{tchs}_{e_k}(b_j)[W(\mathbf{x}_j + \mathbf{a}) + \mathbf{b}] \frac{\partial}{\partial \mathbf{v}_i} \mathbf{v},$$
$$\sum_{j=1}^{n} \text{clr}(b_j) \text{tchs}_{e_k}(b_j)[W(\mathbf{x}_j + \mathbf{a}) + \mathbf{b}] \frac{\partial}{\partial \mathbf{v}_i} \mathbf{v}, 0\}$$

$$\frac{\partial E^*(G)}{\partial W_{ij}} = \sum_{k=1}^{t} \mathbf{g} V \{0, 0,$$
$$\sum_{\ell=1}^{n} (1 - \text{clr}(b_\ell)) \text{tchs}_{e_k}(b_\ell) \mathbf{v} \left(\frac{\partial}{\partial W_{ij}} W \right) (\mathbf{x}_\ell + \mathbf{a}),$$
$$\sum_{\ell=1}^{n} \text{clr}(b_\ell) \text{tchs}_{e_k}(b_\ell) \mathbf{v} \left(\frac{\partial}{\partial W_{ij}} W \right) (\mathbf{x}_\ell + \mathbf{a}), 0\}$$

$$\frac{\partial E^*(G)}{\partial \mathbf{a}_i} = \sum_{k=1}^{t} \mathbf{g} V \{0, 0,$$
$$\sum_{j=1}^{n} (1 - \text{clr}(b_j)) \text{tchs}_{e_k}(b_j) \mathbf{v} W \frac{\partial}{\partial \mathbf{a}_i} \mathbf{a},$$
$$\sum_{j=1}^{n} \text{clr}(b_j) \text{tchs}_{e_k}(b_j) \mathbf{v} W \frac{\partial}{\partial \mathbf{a}_i} \mathbf{a}, 0\}$$

$$\frac{\partial E^*(G)}{\partial \mathbf{b}_i} = \sum_{k=1}^{t} \mathbf{g} V \{0, 0,$$
$$\sum_{j=1}^{n} (1 - \text{clr}(b_j)) \text{tchs}_{e_k}(b_j) \mathbf{v} \frac{\partial}{\partial \mathbf{b}_i} \mathbf{b},$$
$$\sum_{j=1}^{n} \text{clr}(b_j) \text{tchs}_{e_k}(b_j) \mathbf{v} \frac{\partial}{\partial \mathbf{b}_i} \mathbf{b}, 0\}$$

Therefore, we can easily compute the same for $E(G)$ knowing that $d|x|/dx = x/|x|$, i.e. flop signs if $E^*(G) < 0$. When $E^*(G) = 0$ we set the derivative equal to 0.

The idea behind this algorithm is to first determine an estimate, the r_i, as to whether blocks are alive or dead. Add to the score two times the value of the dead groups (empty territory gained by opponent and capture points). Then using the life and death information, the empty territories are iterated over and using the L&D information as well as the size of the territory a change in score is determined from the territory. For instance, a territory with no living adjacent black stones should be awarded to white.

We also tried another energy function $E(G) = |S - (S^* + D)|$ where

$$D = 2 \cdot \sum_{i=1}^{n} \text{size}(b_i) \text{clr}^*(b_i) r_i$$

where $\text{clr}^*(b)$ returns -1 if b is white and 1 otherwise, but obtained similar results.

4 Details of Work

For this project we have programed 5,325 lines of C++, using only the standard libraries. By running `make` one obtains the main program `gobot`. This program takes as input reformatted games in a specific format. This format can be created from `*.sgf` files by running the Haskell program `reformat`[1] on individual files or by running `reformatGames.sh` on a directory. In case `ghc` is not available we have included reformatted programs in `reformattedGames.tgz`[2].

Using `gobot -generate inputDirectory outputDirectory` feature files can be generated for all games in `inputDirectory` in `outputDirectory`. Or alternatively one can simply run the script `reformatAndGenerateFeatureFiles.sh gameDirectory reformattedDirectory featureDirectory` to run both the reformatting and generation procedures in succession.

A model can be trained over a data set using the command `gobot -train modelFile iterations gameDirectory [featureDirectory]` where `modelFile` is the file to save the model to, `iterations` is the number of iterations to perform, `gameDirectory` is the location of the games to train on, and optionally `featureDirectory` is the location of the feature files corresponding to the game (note that if

[1] if `ghc` is installed this is created by running `make reformat`

[2] Created from a small subset of the collection of games at `https://badukmovies.com/pro_games`

a game is named `game.sgfo` then the feature file must be named `game.sgfof`).

A model can be tested over a data set using `gobot -test modelFile gameDirectory [featureDirectory]`, whereas model can predict the score of a single game using `gobot -predict modelFile gameFile [featureFile]`.

Overall, in terms of time usage, the most taxing parts of creating this program were, in order, feature generation and then RPROP training. This can be seen just from their files, `src/board.features.cpp` is 1,125 lines long and `src/rprop.training.cpp` is 1,440 lines long. However, the debugging process was much harder for the training as opposed to feature generation since we had to make sure that the problems in our algorithm were not code related but rather model related, i.e. relating to our energy and scoring functions. All in all, we spent well over 100 hours on this project.

5 Results

By iterating over the files in a directory rather than reading them all in at once, `gobot` is able to train and test massive datasets in while using a small amount of memory (under 800kb for training, testing, and predictions). Unfortunately because of the complex nature of the game, specifically the fact that the moves in a game can overlap in the case where a dead group has previously been removed from the board, it takes a long time to train and test massive sets of games, i.e. those with more than 10,000 games.

Training our data over the 500 professional games in `reformattedGames.tgz` takes approximately 5 minutes on 1.6GHz processor. Testing over the same 500 games takes about 5 minutes as well, unfortunately the test returns a result of 0.0% accuracy over the data.

Similarly we trained our data on other distinct sets of 5,000 and 15,000 games and also obtained a 0.0% accuracy over the training data. This means that our learning algorithm has failed to learn to score the game. Upon closer inspection of the actual predictions that the model was making, we see score estimations like -58,583,120.0 or 3,903,552.7, scores which are not physically possible to obtain on a Go board (3,903,552.7 means that white won by 3,903,552.7 points).

Clearly these scoring problems arise from our choice of the energy function.We believe that we are seeing numbers like this for two reasons. First of all, the use of the absolute value in the energy function causes a huge loss of information about how the estimated score compares with the true score; the algorithm as is cannot possibly know if black or white was given too many points, only that one of them was, with either one being equally likely to be the occurring event. Secondly, since as more and more games are in the training set the estimated score moves further and further away from 0, we are led to the conclusion that our energy function is in fact a concave function since performing gradient descent in the RPROP algorithm moves the estimate further from the true value (by moving very far away from zero). Therefore, we conclude that the RPROP algorithm was not the correct update algorithm to choose for this energy function.

6 Future Directions

There are many ways to speed up the algorithm. For instance one could write the final board, i.e. current score (komi plus captured stones), to a file and then read it during training, thereby skipping the necessity of having to internally play the game each time you want to test, train, or predict a game.

Clearly, though, there is a huge problem with the scoring and energy functions as is. Since we already have the ability to generate a large subset of the features used by the algorithm in (van der Werf et. al., 2004), it would be simple to adjust the code to run the paper's model. Still, to use this model, we need to generate labels for a good number of games. Fortunately there is a easy bootstrapping method that works as follows:

1. Manually label a small dataset $\mathcal{G}$ of around 20 games.
2. Choose a much larger dataset $\mathcal{G}^*$ that needs to be labeled.
3. Train the model on $\mathcal{G}$.
4. For each game $G \in \mathcal{G}^*$, estimate the score for the game. If the estimated score equals the true

final score save the labels of the blocks in G, remove G from $\mathcal{G}^*$ and add G to $\mathcal{G}$.

5. If $\mathcal{G}^* \neq \varnothing$ go to (3).

At termination $\mathcal{G} = \mathcal{G} \cup \mathcal{G}^*$ and each game $G \in \mathcal{G}$ has labels. With this information we would not only be able to train the scoring model in (van der Werf et. al., 2004), but, as originally planed, also the life and death model in (van der Werf et. al., 2005).

7 Comparison to Proposal

In the project proposal I said that I would:

1. Provide details of the internal representation of the board
2. Describe algorithms implemented to calculate nontrivial features.
3. Provide implementation notes on RPNC and GDXNC.
4. Detail the performance of both algorithms.
5. Provide a detailed description of the final scoring algorithm.
6. Propose further ways of improving the algorithm using machine learning.
7. Provide C++ code.

We have completed points (1), (2), (5), and (6). But due to time constraints and the poor choice of a learning algorithm we were only able to complete the RPNC portions of (3), (4), and (7) due to the massive amount of time that was needed to implement both the feature generation portions of the program and the RPROP training.

References

David B. Benson. *Life in the Game of Go*. Information Sciences, 10, pages 17-29. 1976.

K. Chen and Z. Chen. *Static analysis of life and death in the game of Go*. Information Sciences, 121, pages 113-134. 1999.

A. Kishimoto and M. Müller. *Search versus Knowledge for Solving Life and Death Problems in Go*. Twentieth National Conference on Artificial Intelligence (AAAI-05), pages 1374-1379. 2005.

X. Niu, A. Kishimoto, and M. Müller. *Recognizing Seki in Computer Go*. Lecture Notes in Computer Science, pages 225-238. Springer, 2006.

M. Riedmiller and H. Braun. *A Direct Adaptive Method for Faster Backpropagation Learning: The RPROP Algorithm*. IEEE Int. Conf. on Neural Networks, pages 586-591. San Francisco, 1993.

R. Silver, R. Sutton, and M. Müller. *Reinforcement Learning of Local Shape in the Game of Go*. Twentieth International Joint Conference on Artificial Intelligence (IJCAI), pages 1053-1058. Hyderabad, India. 2007.

E. Thorpe and W. Walden. *A partial analysis of Go*. Computer Journal, 7, No. 3, pages 203-207. 1964.

E.C.D. van der Werf, J. van den Herik, and J.W.H.M. Uiterwijk. *Learning to score final positions in the game of Go*. Advances in Computer Games 10, pages 143-158. Kluwer, 2004.

E.C.D. van der Werf, M.H.M. Winands, and H.J. van den Herik, J.W.H.M. Uiterwijk. *Learning to predict life and death from Go game records*. Information Sciences, 175, pages 258-272. 2005.