Applying Artificial Neural Networks to Playing Go

Simon Chase, Philippe Demontigny, Daniel Seita

December 13, 2013

Abstract

The game of Go is an extremely difficult problem for computers to solve, even when considering the 9x9 setting. Common AI game search techniques fail due to the enormous branch factor in these games: there are few restrictions on where a player can place a stone. In addition, human professionals may rely on concepts difficult to encode in a computer program. In our final project for CSCI 373, we created a 9x9 Go AI, which we name KamiGo, using the machine learning algorithm of artificial neural networks. We explain our methodology, the feature vectors constructed, and our AI's performance results.

1 Introduction to Go

1.1 Basic Rules

Go is a centuries-old, two-player, deterministic, zero-sum game with perfect information. Players control either black or white stones and alternate placing them on a grid. The official game board has 361 possible locations, arranged in a 19x19 square, but smaller versions are also used by players who are learning the game. In this report we will only focus on the 9x9 setting, which is the smallest board used regularly in professional play. The objective of Go is to capture *territory* by surrounding more spaces on the grid than the opponent. To surround an area is to completely enclose a section of the board with a *group* of stones. Stones form groups when they are placed directly adjacent to each other. Each empty location adjacent to a stone is called a *liberty*. As long as stone has one or more liberties, it is alive. Groups of stones of the same color share liberties, but if all of a group's liberties are occupied by opposing stones, that group is captured and removed from the board. See Figure 1 for an example of when black stones capture an opponent's white stone.

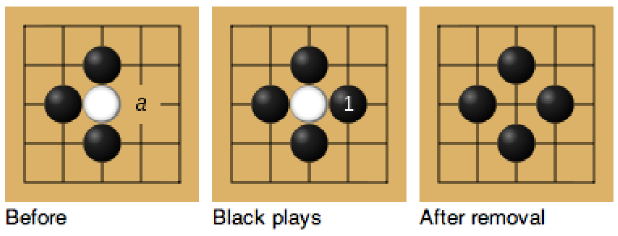

Figure 1: A move that results in the capture of a white stone.

Although the basic rules of Go are provided here, there are a few important concepts that are left out for the sake of brevity. The reader is encouraged to consult the multitude of references existing in print and online.

1.2 Go-Playing AI

Go is a difficult problem for computers to solve; indeed, it has recently been recognized as the "Holy Grail" for computer game playing due to its massive branching factor and difficult evaluation function [Low, 2008]. Even when considering the smaller 9x9 setting, it is only recently that programs have been able to reach a professional level. Evaluation functions to produce a heuristic for Go-playing AIs are challenging to implement since the best human players rely on concepts difficult to encode in programming (e.g., potential for an area to be live or dead), and because there are many local moves that can have unforeseen long-term effects.

One of the first successful attempts at creating a artificial intelligence for Go was one that stored and drew upon vast stores of data based on professional and conventional knowledge of the game itself. The most successful of these programs was called "The Many Faces of Go," which used a very large pattern database, an opening move dictionary, and a rule-based territory evaluation function, among other things [Fotland, 1993]. While this program was the strongest of its kind, it was not strong enough to beat pro-ranked players.

The next generation of computer Go programs made use of artificial neural networks, described in Section 2. The most widely recognized of these are NeuroGo [Enzenberger and Cazenave, 2003] and HonteGo [Dahl, 1999]. The success of these players can be attributed to the fact that many of them incorporated a knowledge-based agent that was able to override the decisions made by the neural net if a more obvious move was available [Enzenberger, 1996]. However, it cannot be said that neural networks were a strong improvement over the more widely used commercial programs, as NeuroGo and its cousin HonteGo were still unable to defeat even the lowest ranked pro players.

The world of computer Go dramatically changed when programs starting using Monte Carlo simulation to determine their moves. These programs were the first that were able to compete at the pro level, although the best versions could still only beat the lowest ranked professionals [Low, 2008]. Most of the currently maintained programs use this method, including Fuego (described in section 3.4) and the program MoGo [Gelly et al., 2006], which has successfully defeated mid-ranked pros on a 9x9 board (although not consistently).

2 Artificial Neural Networks

We used artificial neural networks to train a Go-playing AI. Artificial neural networks (ANNs) are a commonly-used machine learning algorithm motivated by biological systems. The smallest unit, the perceptron, tries to simulate the behavior of a single neuron: the perceptron takes in a series of inputs, each drawn from the output of a previous perceptron, and computes a single output value. By connecting several stages of perceptrons together, an ANN tries to mimic complex systems of interconnected neurons. Of course, real neural networks (e.g., the human brain) are made up of a far greater number of neurons, with each having a far greater number of connections than the ANN perceptron might. But it turns out that ANNs are excellent learners for problems in which training data may be noisy and complex, and when the classification may be highly nonlinear. ANNs are commonly applied to problems involving inputs from cameras and microphones, such as teaching a robot how to steer a car [Mitchell, 1997].

ANNs are built out of a series of densely connected layers of nodes. Their structure can be described as a series of two complete, bipartite graphs. Each node in the input layer forwards its output to each node in the hidden layer. Likewise, each node in the hidden layer forwards its output to each node in the output layer. We refer to each node as a *unit*, which takes some set of inputs and computes an output. For a node in the input layer, its output is simply its input. For a node in the output and hidden layers, its output is the result of a function computed over its inputs.

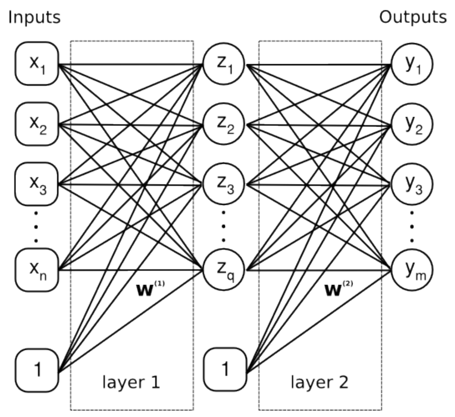

Figure 2: An artificial neural network, including a bias term.

Figure 2 is an example of an ANN with an input layer of n nodes, which corresponds to the n-dimensional vector of some element in the training data. There is a hidden layer consisting of q nodes, and an output layer of m nodes. Each hidden node z_i has its own weight vector $w_i^{(1)}$, and we compute its output (call it $out(z_i)$) by taking the weighed sum of the input values and then apply the sigmoid function σ.

$$out(z_i) = \sigma\left(\sum_{j=0}^{n} w_j x_j\right) = \frac{1}{1 + e^{-\sum_{j=0}^{n} w_j x_j}}.$$

The values for the output layer are computed similarly, only this time the function takes as input the output of the *hidden* layer's nodes.

In this particular application to Go-playing AIs, we use a single node to form the output layer. The input to the neural net is formed from a view of the board around an empty spot, with the output being the strength of placing a stone in that particular spot. The higher the score, the better. In our training, we set the number of hidden nodes to be ten, though an obvious continuation of this experiment would be to test our ANN with different amounts of these nodes.

To *train* an ANN to learn ideal weights, one can run the **Backpropagation Algorithm**. For a single round of training, the algorithm takes a set of inputs and a target value (what we wanted the ANN to output in an ideal circumstance), and modifies the tree (ever so slightly) so that it conforms more closely to the desired target output for that particular input. For a single round of training, the backpropagation algorithm will:

- Propagate the input forward through the network.
- For each node, compute its error term based on the target output value.
- Update each network weight.

The last two steps above are known as the "Backpropagation step" since the errors are propagated backwards through the ANN. We will refrain from listing the technical details; for that, we refer the reader to [Mitchell, 1997]. In our neural net, we use an array-based implementation to simulate the behavior of the Feed-Forward Tree described in [Mitchell, 1997].

3 Our Contribution: KamiGo

The contribution of our project is that we have constructed a Go-playing AI that computes its moves based on a trained ANN. Thus, given a current game state, our AI will iterate through all possible empty board spaces, compute the given feature vector that would result from playing the move, and then compute the score of it from the ANN. The move that has the highest result is thus chosen.[1] We can play games on the command line (we did not construct or use a GUI) by alternating between a human playing black and the computer playing white, or we can have the computer AI suggest any move for either player in any situation.

3.1 Obtaining Game Records and Setting Transcripts

In order to even think about using ANNs, we needed to obtain training data in the form of feature vectors plus an output, i.e., $\{(\mathbf{x}_1, y_1)), \dots, (\mathbf{x}_n, y_n)\}$. For this, we used an online repository[2] of 9x9 professional Go games. Fortunately, the games we found were all in the same *Smart Game Format*, a computer file format for storing records of board games. We copied over the records of 183 games into a file and then wrote code to produce *transcripts* for each game.

We define a transcript of a single Go game to be a list of strings, which consists of the board state, followed by the given move, followed by the *updated* board state, then the next move, and so on. The strings representing the board state consists of 81 letters, either `E`, `B`, or `W`, representing board states that are empty, occupied by a black piece, or occupied by a white piece. The following text shows the first six lines of a possible game transcript. The strings representing boards first list the nine spots in the first row, then the nine spots in the second row, and so on. Moves are of the form `PXY`, where `P` indicates either black or white, and `X` and `Y` represent the coordinates. These moves needed to be interpreted from the system used to store game records in .sgf files. Their coordinate systems uses the letters *a* through *i* to represent rows and columns.

```
E E E E E E E E E E E E ...
B00
B E E E E E E E E E E E ...
W11
B E E E E E E E E E W E ...
B73
...
```

While we had the full record of *moves* from the Go game files, they did not include a way to compute the *board configuration*, and that is difficult to code due to the process of capturing pieces. Thus, to assist us in this last step, we used Fuego, discussed in Section 3.4. This allowed us to obtain transcripts for all 183 games, which meant that one major step of the preprocessing was already done before training started.

3.2 Setup and More Pre-Processing

We also define our coordinate system to be as follows: 0 to 8 for the x-axis, and 0 to 8 for the y-axis, with the $(0, 0)$ point on the upper left hand corner. Moves (x, y) are thus defined as going *across* by x and going *down* by y. Figure 3 represents a possible board state by our Go program.

[1]Technically, we should only consider *legal* moves. Unfortunately, we were unable to completely finish this aspect of our project. Our plan was to use Fuego (see Section 3.4 for details) for this purpose.

[2]`http://gobase.org/9x9/`

```
  0 1 2 3 4 5 6 7 8
0 E E E E E E E E E
1 E E E E E E E E E
2 E E E E E E E E E
3 E E E E E E E E E
4 E E W E B E E E E
5 E E E E E E E E E
6 E E E E E E E E E
7 E E E E E E E E E
8 E E E E E E E E E
```

Figure 3: A possible board configuration using our coordinate and labeling system. Here, both black and white have placed one stone on the board. This is the kind of output one gets from our program after any move, or if one types `showboard`.

Here, `E` represents an empty board spot, and `B` and `W` represent board positions occupied by black and white pieces, respectively. In this particular state, black has played in the center spot $(4, 4)$, and white has countered with $(2, 4)$.

With our own coordinate system made clear and the transcripts set up, the next step was preparing our ANN code. For this, we modified Simon Chase's old backpropagation code he wrote in CSCI 374 to make it suitable for our problem and to add in the extra bias term we desired.

We also needed to agree on the features to use, and the parameters of the ANN. For this, we selected the following set of attributes:

- 10 hidden nodes (not including the bias)
- 42 input nodes based on features (not including the bias)
- One output node, indicating the strength of a move on a 0-to-1 scale
- Stopping criteria of 100,000 iterations for each game
- A constant learning rate of 0.1

3.3 The Feature Vector

The interesting part about our ANN is the way we designed our feature vector, which consisted of 43 binary-valued components (the bias term is always one). Most of these — 36 to be precise — come from "influence," which is based on features used by the Honte Go AI [Dahl, 1999]. For a given move choice, we chose to look at all spots that were of Manhattan distance 2 from the move choice (not including the move choice itself). This gave us 12 spots, which we numbered one through twelve. This representation is similar to the one utilized by HonteGo, except they used 36 spots instead of our 12. An example of this input scheme can be found in Figure 4.

To actually construct features from these, we look at each of our 12 spots and associate to it a three-tuple of values. If the spot has a black piece, we assign it as `[1,0,0]`. If it's white, we assign it `[0,1,0]`. Finally, if it is empty or if it is not a legal board state (i.e., it is off the edge of the map) it is `[0,0,0]`. We concatenate all these together to get 36 features.

We also wanted to take into account the location of each move relative to the edges of the board, and the number of moves that have been played. These variables help to establish where and when a move is to be played, since our 12-space system is limited in this regard.

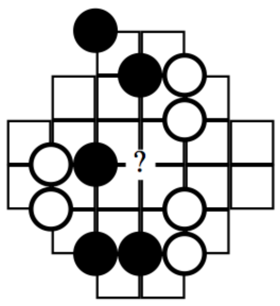

Figure 4: One possible example of a way to determine features for a given board state and move.

- We had three features based on whether a given move was at the center, edge, or corner regions. The 37^{th}, 38^{th}, and 39^{th} input values were set to `[1,0,0]` if the move was in the center region, `[0,1,0]` if it was in a non-corner edge region, and `[0,0,1]` if it was in an edge. These are the three major territories in any sized Go board, and it is commonly said that "in Go, the corners are gold, the edges are silver, and the center is bronze."
- The final three features were based on whether a given move occurred in the first 10 turns of the game, during turns 11 through 40, or beyond turn 40. The 40^{th}, 41^{st}, and 42^{nd} input values were set to `[1,0,0]` if the move occurred in the first 10 turns, `[0,1,0]` if it occurred between turns 11 and 40 (inclusive), and `[0,0,1]` if it occurred on turn 41 or beyond. These time periods separate out the early, middle, and late game, respectively.

Concatenating the 36 influence features, the three region features, the three move counter regions, and the bias created a length-43 input vector for each given board state and move. We trained our ANN by alternating the input view with respect to the actual professional's move in that situation, followed by a random move.

For a single professional move, we gave the ANN the input view centered on the empty spot where the professional player went during that turn, with the target output value set as *one*. For a single random move, we gave the ANN the board view centered on an empty spot chosen at random from all available *non-professional* moves, with the target output value set as *zero*. The goal of this training was to encourage the ANN to associate better moves — the ones made by a professional player — with higher output values, and lesser moves with lower output values. By limiting the amount of non-professional moves to be equal to the number of professional moves, we also prevented any skewed data distribution.

3.4 Fuego

Fuego[3] is an online collection of C++ libraries for developing software to play Go. Our original plan for this project was to use Fuego for two purposes. The first was to update the board for us as we were training. This would greatly assist us in forming the game transcripts, described in Section 3.1.

The second planned use of Fuego was to enforce the rules of Go as we were playing, as we had no desire to write our own rules from scratch. We wrote scripts so that our code could read in output from Fuego and vice versa, but we ran into some issues with the whole piping process, so we decided to scrap the second use of Fuego.

[3]`http://fuego.sourceforge.net`

```
  0 1 2 3 4 5 6 7 8
0 W W E E E E E E E
1 B B E E E E E E E
2 W E E E E E E E E
3 B W E E E E E E E
4 B B W E B E E E E
5 W E E E E E E E E
6 B E E E E E E E E
7 W E E E E E E E E
8 B E E E E E E E E
```

Figure 5: A possible board state after 15 moves of KamiGo, playing for both black and white.

4 Results and Conclusions

4.1 The Artificial Neural Network

We trained our ANN on 100,000 iterations, but we also kept "intermediate" ANNs on file that were based on 10,000 iterations, 20,000 iterations, and so on, so we could observe values of weights as the algorithm was running. By looking at the weights, we saw that the vast majority of them for both the hidden nodes and the output node were between -1 and 1, but strangely, two of the output node's 11 weights — 10 from the 10 hidden units and the last one from the bias — were almost 100. We observed that those weights were consistently growing with more training runs, but the growth rate was decreasing, possibly indicating convergence.

Like those two large weights, most of the other weights appeared to be converging to some values. Still, even with information about all the weights, it is difficult to interpret the output of an ANN. By playing games with KamiGo, which we discuss in Section 4.2, we noticed that there did not seem to be much of a difference in the performance of KamiGo when we trained with 5,000 iterations versus 100,000 iterations.

4.2 Results for KamiGo

Due to the relative simplicity of KamiGo compared to most other state-of-the-art Go software, we were not surprised to see that our AI performed poorly. KamiGo performed strong initial moves by placing stones near the center of the board. After the opening move, KamiGo would inexplicably favor moves that hugged the left edge of the board. Upon reaching the left edge of the board, KamiGo would go up to the corner at $(0, 0)$, and then proceed to move down whenever possible, and then do the same for the second column, then the third, and so on. Figure 5 indicates the board state after one of our trials, where we had KamiGo generate all moves for both black and white. This game was representative of most other games we ran, and seemed to occur no matter which training sample we used (when it was at least 10,000 iterations). The following is the list of moves for the game shown in Figure 5:

```
Moves = {B44, W24, B14, W13, B03, W02, B01, W00, B04, W05, B06, W07, B08, W10, B11}
```

We are unable to confidently explain why our KamiGo AI acted the way it did. One might assume that since the moves seem to be going down column-by-column, we might simply be iterating through the board's elements, but that also doesn't explain the first few moves. We do have some theories, though:

- KamiGo has no conception of *Moyo*, or spread across the board. It cannot determine a region on the board that is *likely to become* a player's territory.
- Since most of the board is empty, KamiGo favors moves close to other stones, because professionals almost always play a stone adjacent to, or in the immediate vicinity of a previous move.
- Inputs are not diverse enough. Ideally, we would expand the set of inputs to include information such as whether a piece is enclosed within some player's territory, whether there are eyes nearby, whether it can capture another piece, and so on.
- Go is not a game where good moves are entirely dictated by pattern, and thus some external calculation is required.

The last two are especially important design considerations for a Go AI. Having a length-43 input vector was unlikely to be successful, especially when other Go AIs using neural nets had hundreds of features. Furthermore, another downside with our features is its sparsity: most features have value 0, and the rest that don't must have value 1. This might make it harder to extract meaningful numbers. After all, we can only have at most one-third of the 42 non-bias inputs be one; the rest must be zero by design.

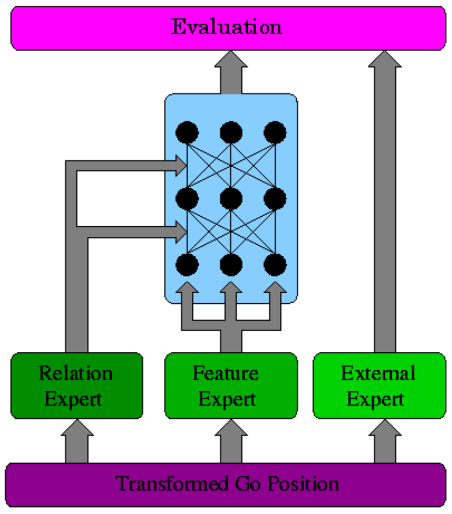

Figure 6: A better paradigm to construct a Go AI.

In addition, it is not enough just to observe patterns, as moves that experts make can rarely be captured by patterns alone. Figure 6 demonstrates a possible framework to take other input into consideration.

4.3 Final Thoughts

This final project taught us a lot about the intricacies of creating a competent Go AI. It showed us that there is great depth in a human Go player's ability to understand the game that cannot be fully captured by a simple set of features and a machine learning algorithm.

We also learned about the importance of good documentation and organization of methods. With the amount of code that we produced over the past few weeks, it was easy to get lost or forget where we implemented a certain algorithm. Also, when we were searching for a Go API to build upon, we noticed that many of the publicly available databases were not well maintained or documented, so deciphering their cryptic instructions often took longer than necessary.

- Philippe: This was the first time I have ever embarked on a project this big starting from scratch, so it was an excellent experience in using high-level programming functions such as writing and reading files, piping input and output, and handling large data sets.
- Simon: I don't have anything to say.
- Daniel: This project taught me quite a few things. To keep it brief: (1) I should have taken operating systems last semester, since that might have helped me understand the process of connecting to Fuego, (2) I should have have a clear plan and documentation for code before writing much of it, and (3) I should have prioritized the process of getting Fuego's input/output working with our programs.

Overall, though, we worked well as a team. We did get a working AI, and did so without any conflicts, quarrelling, and hard feelings. We look forward to additional opportunities to work together in the future.

References

[Dahl, 1999] Dahl, F. A. (1999). Honte, a go-playing program using neural nets. In *In Workshop on Machine learning in Game Playing*, pages 205–223. Nova Science Publishers.

[Enzenberger, 1996] Enzenberger, M. (1996). The integration of a priori knowledge into a go playing neural network. Technical report.

[Enzenberger and Cazenave, 2003] Enzenberger, M. and Cazenave (2003). Evaluation in go by a neural network using soft segmentation. In *In 10th Advances in Computer Games conference*, pages 97–108. Kluwer Academic Publishers.

[Fotland, 1993] Fotland, D. (1993). Knowledge representation in the many faces of go.

[Gelly et al., 2006] Gelly, S., Wang, Y., Munos, R., and Teytaud, O. (2006). Modification of UCT with Patterns in Monte-Carlo Go. Rapport de recherche RR-6062, INRIA.

[Low, 2008] Low, Y. (2008). Investigating the use of Machine Learning Methods in Go. Master's thesis, Carnegie Mellon University.

[Mitchell, 1997] Mitchell, T. M. (1997). *Machine learning*. McGraw-Hill, New York, NY.