USING HARD AND SOFT ARTIFICIAL INTELLIGENCE ALGORITHMS TO SIMULATE HUMAN GO PLAYING TECHNIQUES

RICHARD CANT, JULIAN CHURCHILL, DAVID AL-DABASS

Department of Computing and Mathematics
The Nottingham Trent University
Nottingham NG1 4BU.
Email: richard.cant/david.al-dabass@ntu.ac.uk

Abstract: We describe the development of a Go playing computer program that combines the use of hard Artificial Intelligence (AI) techniques (alpha-beta search trees) with soft AI techniques (neural networks). The concept is based on a model of human play where selection of plausible moves is made using a gestalt process based on experience and the plausible moves are subjected to an objective analysis. The performance of the program is analysed by play against a standard computer Go program and it is shown that the use of hard AI enhances the performance of the soft AI system.

Keywords: Computer Go, Neural Networks, Alpha beta search algorithms.

INTRODUCTION

This paper investigates the application of neural network techniques to the creation of a program that can play the game of Go with some degree of success. Current Go playing programs have been markedly less successful than their chess playing counterparts. Whilst success in playing chess has come from a move away from attempting to copy human play, this approach has failed in the field of Go. In the present paper we explore an idea inspired by human modes of thought. The concept used is to combine instinctive, experience based move selection with rigorous analysis. In Artificial Intelligence (AI) terms this is represented as a combination of soft AI, such as neural networks, and hard AI methods, such as alpha-beta pruned mini-max game tree searching.

What is Go?

Go is an oriental game that is very popular in China, Japan and Korea in particular, but which has a very large following all around the world [Chikun 97]. It is a relatively simple game the complexity of which emerges as you become familiar with the ideas presented. A comparison with Chess is often made [Burmeister et al 95], as these are both board-based games of zero-chance. The rules are simpler in Go, however the board is larger and due to the unrestrictive nature the rules there are many more moves available for the Go player to consider.

The game is played on a board, which has a grid of 19x19 intersections. Two players, black and white, take turns to place a single stone on any unoccupied intersection, with the aim of surrounding as much territory as possible. A player can pass at any turn (giving one point to his opponent) instead of placing a stone. Capturing the opponent's stones is also used to increase a player's score but is usually a secondary concern unless significant territory is secured at the same time. A stone is captured when the last of its liberties is removed. A liberty is an empty intersection directly next to the stone. Liberties are shared amongst connected stones. Diagonals are ignored when looking at connectivity between points in Go. Suicide is not allowed unless it is to capture some opponent's stones in which case the suiciding stone remains uncaptured and is left on the board.

To prevent a stalemate occurring through an endless sequence of capturing and recapturing there is a special rule called Ko. The Ko rule states that a player cannot play a move that will produce the same board position as they left it with their last turn. There are variations on the Ko rule that are named after their various countries of origin.

The end of the game is usually reached by mutual agreement between the players, when they both pass consecutively. The remaining stones on the board are considered as to whether they would survive further or not. If they are decided to be effectively dead then those points count for the opposing player. The territory is then totalled up and the winner declared.

Ouline of the paper

The main sections of the paper will describe: neural network techniques with special reference to playing Go; definition of the concepts of hard and soft AI; why hard and soft AI should be combined; how hard and soft AI could be combined using Go as an example application; and an assessment of the usefulness of combining hard and soft AI with reference to Go and the general case.

CURRENT RESEARCH

Neural Networks

The inspiration behind the neural network idea is the simulation of neurons in a biological brain. Biological neurons receive stimulus signals from other neurons and when a certain activation level is reached the neuron fires signals to all the other connecting neurons. This provides the basis for modelling neurological activity on an electronic computer. In the brain, connections between neurons that are used frequently tend to become stronger. The change in the strength of the connections is dynamic and it is this particular feature that means networks of neurons simulated on a computer can be trained to recognise patterns of input and give appropriate patterns of output [Callan 99].

To implement a neural network on a computer a model of neurological activity as described above is used. This usually takes the form of two or more layers of connection weight values. These are changed during training using some specific mathematical rules until the network outputs the correct values for the input pattern. A function for calculating the output of a neuron given a set of input values also needs to be specified. This function is referred to as the activation function, since it determines whether a neuron ‘activates’, that is whether it generates an output signal or not and also determines the strength of the signal based on a set of parameters. This is frequently a simple summing function of the weights multiplied by the input signal along the associated weight.

Figure 1 shows a small but typical multilayer network of the form 3-2-3, 3 input neurons, 2 hidden neurons, and 3 output neurons. Note weights of connections only apply between neurons, so the intial values being fed into the input layer do not have to multiplied by a connection weight. The same applies for the output neurons final output of values. Also note that each neuron is connected to every other neuron in adjoining layers. This is not a universal practice but is representative of the networks developed in this project.

Typically a bias neuron will be present for each layer, except the final output layer, to provide a constant input signal to all the neurons in a layer. This allows simpler and more convienient calculations, since without it the program would have to store a firing threshold value for each neuron, rather than just a set of weights.

It should be noted that networks are not limited to the type suggested above, they can have as many or as few layers as required and can have many different types of architecture. The simplest neural network would be a single neuron, called a perceptron. Possible network models include backpropagation networks, radial basis function networks, both of which are examples of supervised learning methods and self-organising feature maps which involve unsupervised learning. Supervised learning means a training set of input and output data must be provided to allow the network to classify patterns of input. Unsupervised learning does not require training sets and the network is left to group patterns of data by clustering them according to things the items of data have in common.

A particularly interesting and relevant aspect of neural network techniques is that the
neural net developed will usually learn general trends in the relationships between input and output patterns, often these are things that people may not consciously notice themselves. This means the net will frequently be able to handle input that it has not been trained on and will hopefully give useful output. For new classes of input the net may have to be trained

further, but at least this shows that neural nets are flexible, can adapt to changing circumstances and learn from new experience even when installed in its application environment.

Hard And Soft AI

For the purposes of this project it is necessary to discuss the concepts of hard and soft AI and what is meant by these terms.

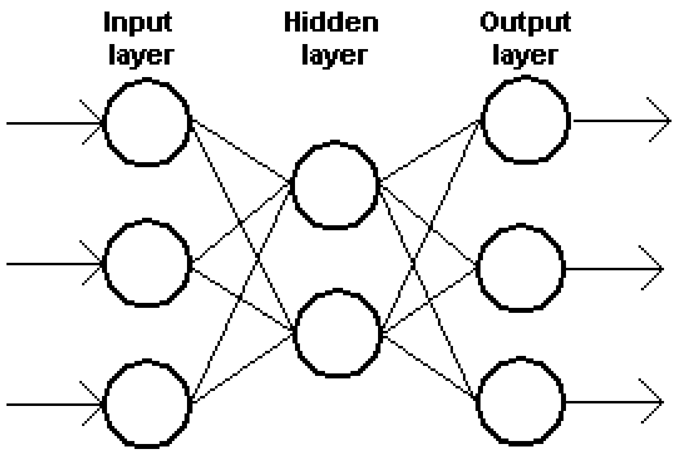


Figure 1: Small Multilayer Neural Network

Hard AI refers to the more traditional artificial intelligence techniques such as the various tree search methods, pattern matching and rule based expert systems. The term ‘hard’ is used to emphasise the predictable and exact nature of such methods, for example if the methods are used to completion you will always get a definite result one way or the other. For a given problem and a given method this result will also be consistent and repeatable.

Soft AI techniques on the other hand deal with methods that rely on statistical processes and may produce results with a probabilistic interpretation. This is in fact one of the benefits of these methods since often in real life problems there is not a single 100% correct answer to be found. These techniques usually make use of random or pseudo-random numbers at some stage and the outcome will vary depending on their values. Methods in this category include neural networks and other machine learning processes such as genetic algorithms and evolutionary techniques.

State Of The Art

The computer Go programming community is relatively small compared to computer Chess but interest in the topic is building now that computer chess has reached such a high level of success, the AI community are looking for a new holy grail. The central hub of this community can be found at the computer Go mailing list [Computer Go Mailing List 01], where many of the best Go programmers gather to thrash out new ideas, protocols and discuss recent tournaments. Also of importance is the computer Go ladder [Computer Go Ladder 01] within which programmers can enter their attempts in an ongoing tournament with most of the best programs available, including commercial programs. The low number of entrants, around 20, perhaps reflects the difficulty of completing a Go playing program that can at least play competently at each stage in a game of Go.

Many Faces Of Go

The information that follows is taken from a document written by the author of Many Faces, David Fotland, in 1993. As far as we are aware the general ideas described below are still the main concepts used in the current versions of the program [Fotland 93].

Many Faces Of Go uses many traditional or hard AI techniques the most important of which are alpha-beta search, pattern matching and rule based expert systems. The main principle is to supply the program with as much knowledge, about how to play Go and what makes a good move, as possible.

The program has been developed for around 17 years and has a very large amount of expert knowledge encoded in it, or in its pattern databases. The author is himself a high-ranking Go player and program will have greatly benefited from his experience.

The main mechanisms work together in a complementary way to determine the best move for the current position. The pattern database is used to retrieve moves that match the current position, tagged with other information such as whether the move is defensive of offensive and also contains a move tree to save calculation time.

There are lots of rule-based experts for many different types of moves that are triggered upon certain conditions.

The alpha-beta search algorithm is used in conjunction with a complex evaluation function to investigate moves where necessary.

The weakness of this approach is that it is totally dependent on the accuracy and completeness of its internal database. Should a flaw be discovered by its opponent then manual reprogramming would be required to correct it.

NeuroGo

One program that has been developed using neural network techniques is called NeuroGo [Enzenberger 96]. It achieved some amount of success against Many Faces Of Go set to a medium level. Many Faces is currently ranked one of the best in the world.

The method used by NeuroGo when using the neural network is to reduce the board position into strings and empty intersections. The relationships between these units are used to construct a neural net. It is unclear from the documentation how NeuroGo deals with the problem of non-static connectivity within the network.

The training target was determined by the Temporal Difference learning algorithm [Schraudolph et al. 94], which has been used with great success in a backgammon neural network program called TD-Gammon [Tesauro 94].

Several experts concerning relations between points and features on the board, relevant to determine the next move, are used in conjunction with the neural network. This is effectively feeding knowledge about important concepts in Go directly into the network, rather than attempting to make the neural net ‘discover’ these concepts itself. This saves time and may or may not be necessary to produce a successful neural network based Go program, although this has yet to be conclusively shown to be the case.

NeuroGo is really a crossover program, which does lean heavily towards soft AI techniques of machine learning, but also incorporates expert knowledge, which must ultimately come from human experience.

A NEW APPROACH

The foundation idea for this project was to use a neural network (NN) to suggest plausible moves and then pass these to a game tree analyser to deep search the suggested moves using a traditional game tree search method. The neural network would thus act as a replacement for the expert systems used in Many Faces Of Go. The use of a NN for selecting possible moves should be fast and also allows the opportunity to expand the NNs knowledge in an automated fashion. This is part of the strength of NNs, even when the NN is actually in use it is still possible to teach it where it went wrong and show it how to correct it’s behaviour by supplying it with the correct stimuli/reaction combination. Any flaw in the database contained in the NN could then be corrected automatically and the opponents opportunities to exploit it would be limited.

Combination Of Hard And Soft AI Techniques

By attempting to use neural networks with game tree search this project is bringing together hard and soft AI techniques and trying to meld them in a useful and productive fashion. The main purpose behind this is to take the best features of each component and combine them to produce something that performs better than either of the two components separately. In the case of game tree search the advantage of using it comes from its ability to look ahead into the probable results of playing a particular move. The disadvantage is that it can be very resource intensive and inefficient. For example without additional help a standard game tree search may spend as much time looking at what may be obviously bad moves to us, as it does looking at what may be obviously good moves.

It has no knowledge, no sense of what makes a good or bad move.

Hopefully this is something that the use of soft AI techniques, neural networks in this case, can remedy. The advantage of using neural networks is that they can provide some knowledge, learnt from experience, of actually playing the game. The disadvantage is exactly the benefit that game tree search provides, in that neural networks work with the posistion of the board as it is. They aren't directly able to handle looking ahead and considering the consequences of playing a particular move explicitly. For example, what appears superficially to be a good move may be refuted by a particular sequence of moves played by the opponent, producing unfortunate consequences.

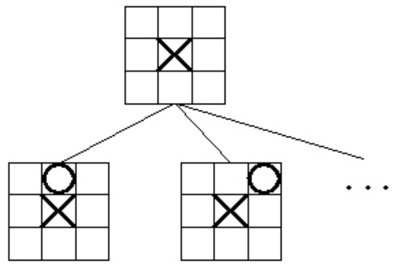


Figure 2: A Partial Game Tree for Tic-Tac-Toe

To summarise, the use of hard and soft AI techniques combined in a carefully thought out manner can negate the disadvantages of both and accent the advantages to produce an overall benefit to the system that it would not have been able to get through the use of only one or other of the classes of methods.

Application To Standard Game Tree Search

A game tree search works by taking a root node, usually the current board position and then generating all possible child board positions by applying plausible moves to the root node. These in turn have all plausible moves played upon them and so on, until a desired depth is reached or until some other cutoff criterion comes into play. Sometimes all possible moves are used, but it is common to cut this down by removing illegal and obviously bad moves from the tree.
The terminal nodes in a game tree are assigned a score based upon some static analysis of the board position at that node and the scores from all the terminal nodes are filtered up the tree toward the root node. Using a technique known as Minimax [Owsnicki-Klewe 99], the best move, the one that maximises the players score, regardless of the opponent's play, can be found, however there are still many problems with this approach. For instance the terminal node board scoring method must be fairly accurate else the minimax values will be irrelevant, and for many games the computational power required to generate a game tree large enough to produce a conclusive result is enormous. For example a game of chess could have around 40

moves and at each turn there maybe 10 legal moves available giving 10^{40} nodes. For Go it is much more complex due to the 19x19 board and very few restrictions on moves. With around 220 moves per game and an average of about 180 different moves to play each turn it has been estimated that the size of this particular search space is around 10^{170} [Allis et al. 91]. So an exhaustive search is completely out of the question, even a relatively shallow game tree search, 6 ply, could easily be beyond the resources of the computer.

Thankfully there are a few techniques to help reduce the search space size and hence the game tree. Some nodes in the tree can be pruned away, occasionally taking large branches too, but this is still not enough.

In a standard game tree search much processor time will be spent on generating nodes for the next level of the tree. Often the algorithm will simply take all available legal moves and incorporate them into the search tree in an attempt at exhaustive search. A neural network can be used to quickly and efficiently suggest the most plausible moves given a board position and this can be used at every level of the game tree, assuming the neural network is trusted enough to suggest reliable and high enough quality moves. This means many pointless moves will be instantly ignored and left out of the search tree, saving resources. Ideally around 6 moves would be used from the suggestions of a neural net interfaced to a minimax game tree algorithm, allowing a depth of about 7 or 8 ply to be reached with out too much stress on the system resources. Possibly a dynamic element could be included to allow the inclusion of more moves depending on the type of position, available time and available resources. Compared to an exhaustive search the resource saving can be shown clearly by comparing the two equations for approximating the number of nodes required to construct a game tree to a given depth:

With neural network: Total nodes ~ n^d

Without neural network: Total nodes ~ 180^d

Where d is the desired search depth and n is the number of moves suggested by the neural network.

SOFTWARE/HARDWARE DEVELOPMENT

At the start of the project a target specification was established as follows.

System Specification

Provide facilities for the program to communicate with other Go playing programs, either via a network connection or on a local machine. The facilities should be capable of conducting a game after initial setting up, without later interference.
A generic neural network program should be developed to create and train neural nets.
The generic application should be expanded and applied to Go, in such a way as to be able to create and train a neural network to suggest plausible moves, using the current board position as input for the network.
To process training data for the network an automated procedure should be developed, which implies that some form of standard format Go data must be able to be read and processed by the program.
To apply the program to Go effectively, utilities must be provided to allow evaluation of a board position, standard game tree search and methods for gathering basic information about the board position such as the number of liberties a string of stones may have.
A user interface should be provided for both the neural network module and the Go playing part of the overall system. A graphical interface would be useful for displaying board positions and for showing what moves the system maybe considering.

System Overview

The system is split up into several modules to aid rapid development and future adaptability. Currently the system is designed to use these modules: Main, Neural Network, SGF and Test. These are described individually below. Note that the design includes these modules however due to time constraints not all features mentioned have been implemented at the time of writing. Please see the end of this section for a report of the actual implementation progress made.

Main Module

This module coordinates the others and provides a user interface to the user. This module also includes lots of Go specific code, such as GMP interfacing routines, game tree search algorithms and many utility functions.

The GMP routines allow the program to connect to other Go playing programs that use the Go Modem Protocol [Wilcox et al.] so the programs can play between themselves, or be used as interfaces to human players playing over a computer network.

An alternative to GMP is currently under discussion within the Go programming community called the Go Text Protocol, GTP [GTP 01]. There are no formal specifications available at the time of writing but it looks extremely likely that it will replace GMP as the standard program-to-program communications protocol. GMP had many problems due to it being designed for efficient use through modems or serial ports in general and suffered from being rather tricky to actually implement. GTP is based on a collection of text commands and does not concern itself with the lower layers of communication such as whether to use TCP or an alternative. It is currently in use by GNUGo 2.7.246 [GNUGo 01] where the main thrust of GTP development is being carried out.

The game tree search algorithm implemented is specifically a version of the classic Minimax search with alpha-beta pruning, called MTD(f) [Plaat 97]. This is currently one of the best performing and most reliable Minimax type search algorithms developed. It is intended

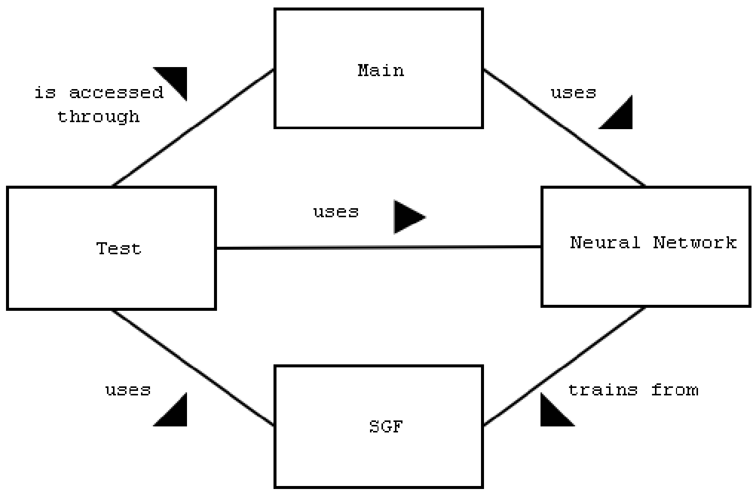


Figure 3: System Overview

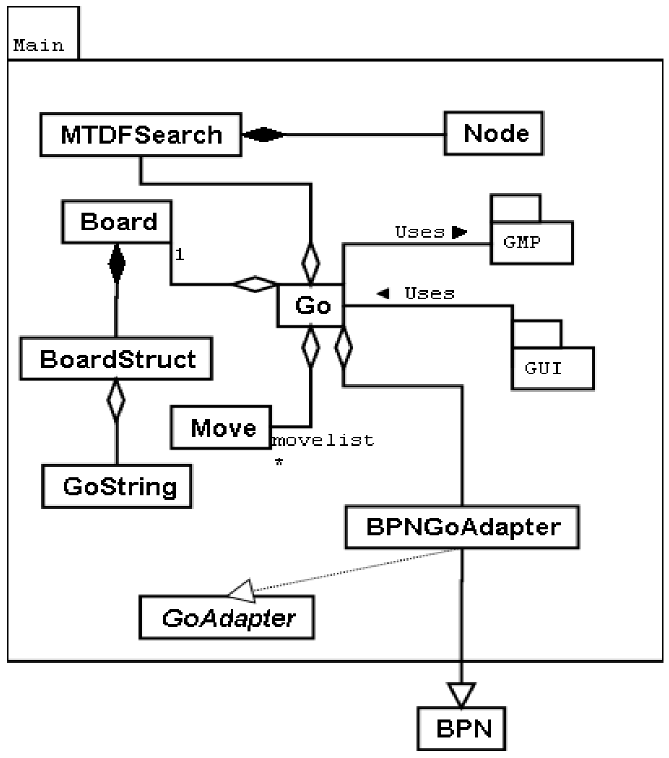


Figure 4: Main Module

that this search algorithm should use the neural network module to supply moves for expanding the tree. Also iterative deepening will be used, as this appears to be the most efficient method of expanding search trees, along with quiescent search. Quiescent search means that the branch being expanded will not be evaluated until a quiet position has been achieved, it will be expanded further until successive evaluations are within some tolerance limit. The purpose of this is to avoid the horizon effect, which occurs when a dramatic move (such as a large capture) can be made just one move beyond the current search depth, totally invalidating the results of the analysis.

a) Utility Functions

The utility functions include a Zobrist hashing function [Huima 99] to allow speedy and efficient storage and comparison of board positions, a function giving the program knowledge of Bensons unconditional life and death definitions [Muller 97], a static full board evaluation function and a board representation that monitors and updates various pieces of information about the board state.

The Zobrist hash function is included because the resource constraints can be so large that any amount of help to reduce usage is invaluable. There are many different ways to apply hashing but the Zobrist method seems to be the most useful and popular when the problem is concerned with a game that needs a board representation.

Bensons unconditional life and death definitions allow a program to find groups of stones that are currently or must be eventually either alive or dead. This helps the evaluation function to determine territory counts.

The evaluation function is essential to the proper use of the MTD(f) implementation so the nodes can be scored efficiently and accurately. One idea is that each stone on a Go board radiates influence, the strength of the influence reducing over distance. A simple algorithm calculates the influence generated by each stone and classifies each point on the board according to which side has the most influence over that point. Each point that each side owns counts as that sides territory. The difference between each side's amount of territory is basically the score of that board position.

There are lots of possible ways of evaluating a position and I intend to include a small selection of evaluation functions for the user to choose from. The ones I have chosen to include are a liberty counter, a stone capturer and a neural network evaluator. The liberty counter simply counts each side's liberty points, subtracting the opponent's liberties from the current players to get a score for the position. The stone capture evaluation functions counts the number of stones on the board for each side and subtracts the opponents from the current player. The network is a little move complex but should be just as fast, if not faster compared to more complex functions than those discussed so far. The idea is to use Temporal Difference learning [Schraudolph et al. 94] to train a network to predict the chance of winning given a board position, using games played by the program against various opponents. The advantage of this is that the evaluation should get better and better the more games the program plays and the more experience it accumulates.

Board Representation

A few definitions first: a string is a connected line of stones (north, south east, west only, no diagonals), a liberty is an empty point next to the string and can be thought of as breathing space for that particular string. According to the rules if a string has no liberties all the stones in the string are captured and removed from the board.

The board is represented by a special 'Board' object that contains a two dimensional array, one value for each point on the board; 0 for empty, 1 for white and 2 for black. An array of 'GoString' objects is also stored that represent each string of stones currently on the board containing details such as which points belong to it, the colour of the string and the number of liberties the string has. This information is updated incrementally; everytime a stone is added or removed from the board. Other arrays are present to represent special markers that may be of use to show the neural networks responses at various points.

Graphical User Interface

A simple but functional GUI must be a part of this project since Go is a visual game and it would be very difficult to judge the performance of the neural network. The GUI would ideally allow a human to play a game of Go against the program thus enabling those

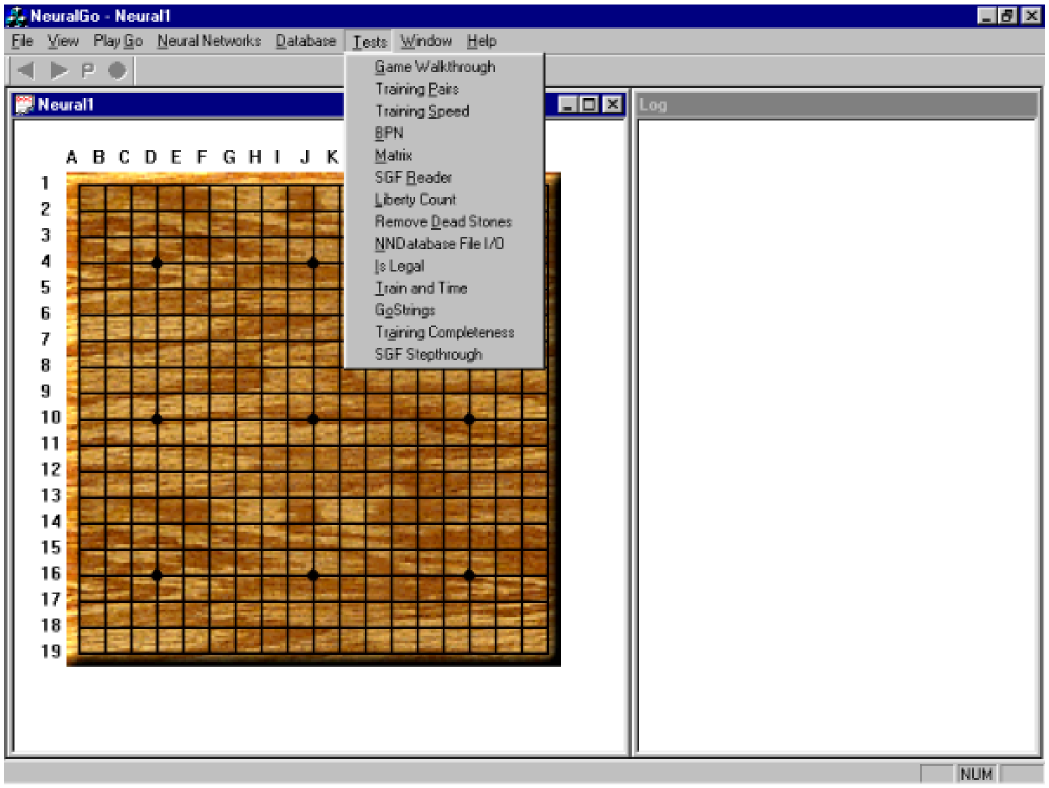


Figure 5: NeuralGo GUI

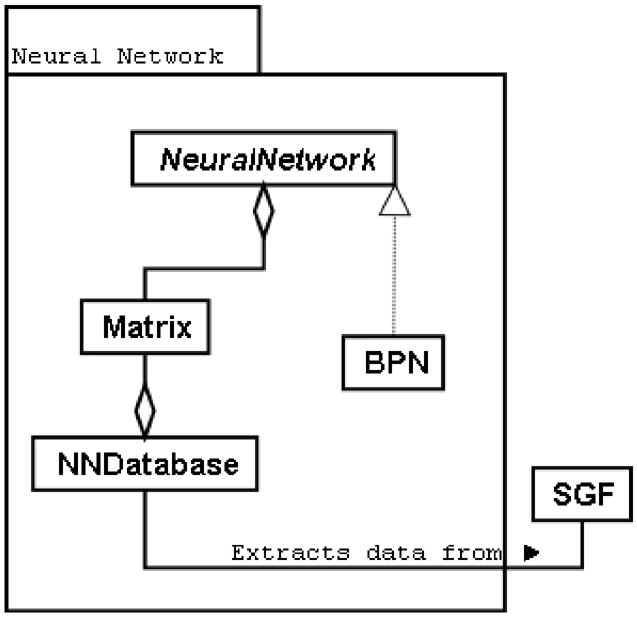


Figure 6: Neural Network Module

people with some knowledge of the game to test it's weaknesses and strengths. Also the user should be able to view the program playing against another program and to observe the neural networks responses during a game or whilst playing through an expert game so that the success of the program can be evaluated.

As can be seen from the screenshot above the interface provides menus to access all of the available functions of the program. A toolbar is present with for greyed out buttons. These currently allow the user to step forward or backwards through moves in an SGF game, to indicate a pass move during a game of Go and to abort the current action by using the round button at the end. A log window is provided to output text messages to the user. From this image the board can also be seen, with coordinate system along the edges. When necessary the user can select points on the board, for instance during a game to specify a move.

Neural Network Module

This contains all neural network specific code and only that. This allows it to be portable to other applications and does not make it dependant on Go. A simple back propagation network will be the only architecture available to start with but the design of the module should allow it to be easily expanded to cater for any number of different architecture types. A key component of this module is the Matrix object, which implements matrix operations and a model for representing matrices throughout the program in a consistent and hopefully efficient manner.

Other neural network types are catered for through the use of a parent class through which all networks must be derived. This defines the basic functions that all network implementations must provide.

Also in this module is a class to represent training pair databases. This allows input/output pairs of matrices to be stored and retrieved using a function to create the data provided by the user. Within the Main module a class called BPNGoAdapter is defined that derives from the BPN network object in this module and uses the training database class to extract training data pairs from SGF files containing expert Go games.

SGF Module

SGF stands for Smart Game Format [Hollosi 99]. This is the standard format for storing Go games as files on the Internet. It is a very flexible format and is widely supported. The SGF module includes object definitions to represent SGF files and methods to save, load and create them. This module is essential to the neural network, since all of its training data is derived from professional games stored as SGF files. The format allows multiple lines of play to be stored, so it is easy to store common game patterns such as opening move suggestions along with the actual play sequence all in a single file.

Test Module

This module simply contains some static test routines to allow consistent validation of algorithms implemented or changed within the Go program in general. Many test functions were included, all to be easily accessible through menu items. The tests included matrix calculation checks, verification of the GoString incremental update code, back propagation training validation and a game walkthrough test (See screenshot figure at the end of section 4.3). This last one was most useful since it allowed the user to see the programs response, and the neural network in particular, to moves made in professional games. This meant the user could compare the actual move and the suggested moves so some sort of quality analysis could be done.

Design Issues

The initial architecture I decided upon was a 3-layer network consisting of 25 neurons for the input layer, 10 for the hidden layer and 1 for the output layer. The 25 neurons of the input layer would map directly onto a 5 by 5 section of a Go board. The idea was to train a network so that all the legal moves possible in a particular board position could be fed into the network one at a time, incorporating a 5 by 5 area of board around the potential move and that the neural net would output a plausibility value for the given move. The move with the highest plausibility would then be chosen, or better still, several of the highest plausible moves would be taken and fed into an alpha-beta minimax search to investigate the moves impact on the future of the game.

After some experimentation with different sized networks I found that a network with 81 input neurons, representing 9x9 sections of a Go board, appeared to be the most useful. It

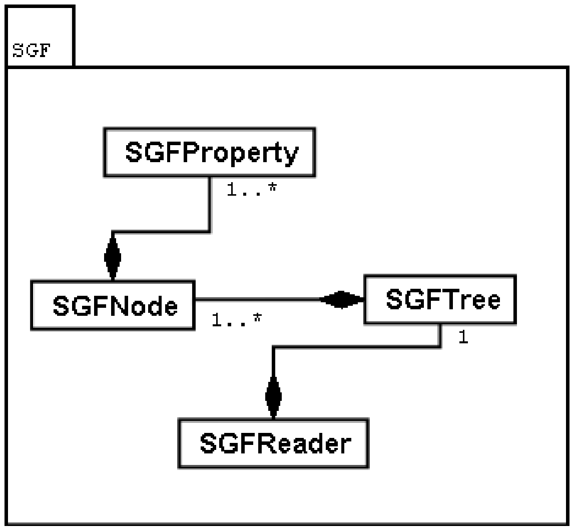


Figure 7: SGF Module

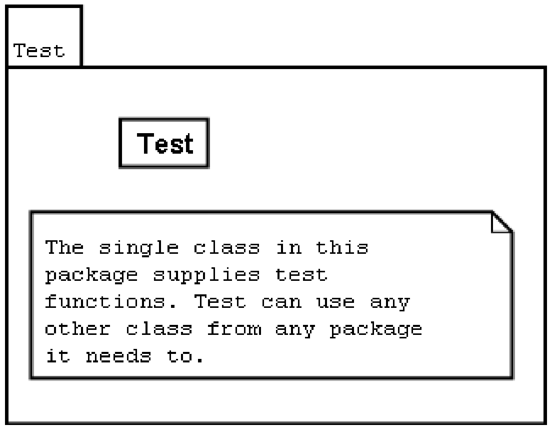

Test
Test
The single class in this package supplies test functions. Test can use any other class from any package it needs to.

Figure 8: Test Module

ran at a reasonable speed considering the benefit of using a larger board area. Larger networks such as 11x11 and 13x13 added considerable amounts of processing time, not just for the training but also during the game. On average a 2 step increase in size, e.g. 9x9 to 11x11, caused a time increase of a factor of 2. Ideally a 19x19 network would be used to allow the network to have all the information it can from the board, however due to the speed factor being so sensitive a 9x9 network was settled on as the optimum for the time available, processing power required and quality of results obtained.

The number of hidden neurons was chosen arbitrarily; all that matters is that there are not too many hidden neurons and not too few. If there are too many then the net may end up just memorising the patterns and not learning trends that connect the input/output pattern pairs. If there are too few hidden neurons there may not be enough room for the network to adapt to all of the training data, though this was of less concern because the network would not be required to converge completely.

The back propagation algorithm [Callan 99] for training was decided upon, mostly because of it's general applicability to a wide range of problems and also it would hopefully be simple to implement and use.

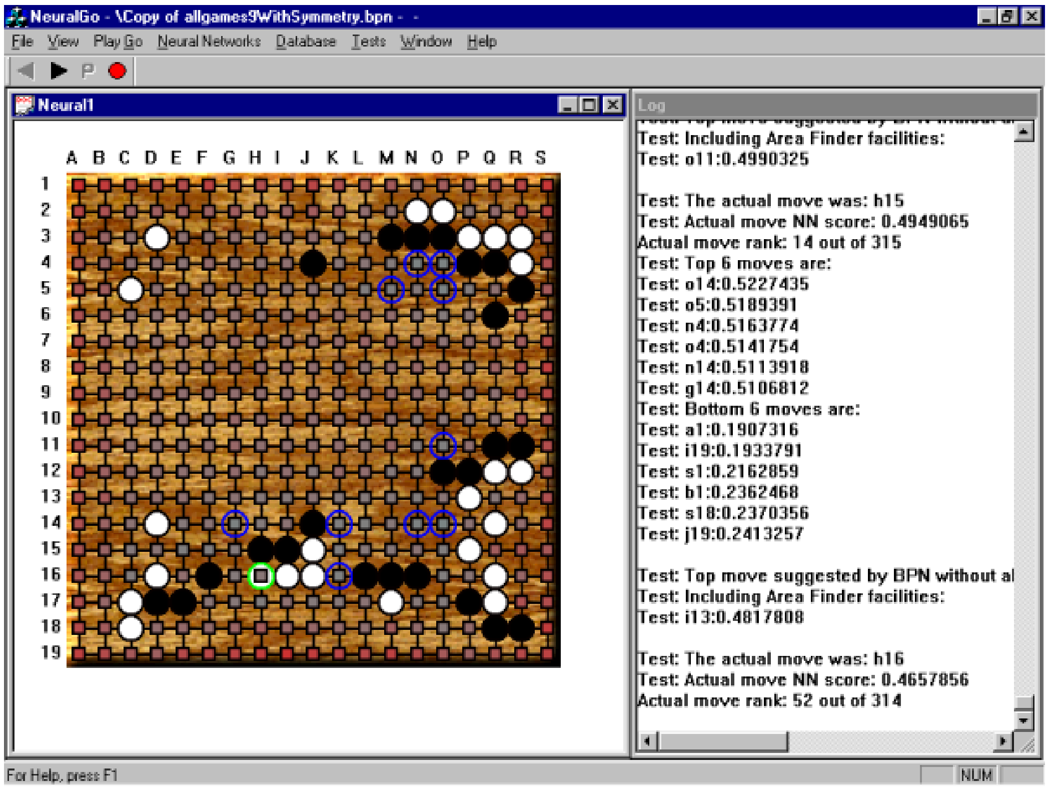
Training data was acquired as a collection of professional tournament games in SGF format from the Internet [Van Der Steen 01]. A training database generator then dissected these files into input/output pairs that could be fed directly into the neural network training algorithm. For each move the board position was analysed to produce several input/output training pairs. For every legal move in the board position an input matrix was generated that represented an n-by-n area centred on the move. It was determined whether the move was played up to 6 moves in the future and if so an output value was associated with it that started at 1.0 for the move occurring next turn, down to 0.2 for it occurring 6 moves in the future. If the move did not occur within 6 moves it was relegated to a set of unlikely moves and an output of 0 was connected to it. At the end 6 unlikely moves were randomly chosen and stored in the database with the 6 likely moves to balance the networks training.

An idea that occurred later in the project was of varying the granularity of consideration of the networks. Just as a human player might, the idea was to look at the board and narrow down an area in stages and eventually identify the best move to play. Until now the program had been designed to look at as many points on the board, and their neighbours in relation, as time and resources allowed. This new method would allow a succession of networks, trained to varying levels of detail to be used. For instance, the top level network could select a single quarter of the board to look at having been fed the entire 19x19 board, and then the next network could look within that quarter, maybe at 9x9 sections around each point present and narrow down the range further. Finally a small 5x5 network could be used to select the best move within the most promising 9x9 section. The method needs to be looked at further to see if the performance gained is worthwhile and if the combination of networks with varying types of expertise is viable.

The figure above shows the game walkthrough test in progress, which allows the user to play through a game of Go that has been stored in SGF format. The program itself makes suggestions and comparisons are made between the program and the actual moves made. This test is most useful is professional quality SGF games are used. The green circle shows the most recent move, the blue circles highlight the programs top ten suggested moves and each point on the board has a small coloured square. This indicates the plausibility score given to a move at that point in the current situation by the neural network. The redder the square the lower the score and the high end goes to blue. Lots of log information is also outputted in the window on the right, showing the actual scores associated with the best and the worst suggested moves. Also the rank of the actual move is outputted, in this case being 52nd out of 314 legal moves.

IMPLEMENTATION

Smart Game Format files can be read and interpreted allowing training databases to be compiled from SGF files. However SGF files cannot be saved and only the main line of play can be read, not any saved alternative lines of play. Many testing functions have been written to maintain consistency throughout development of the program and to allow progress to be measured. The code to create and maintain GoString information has been written and stores which stones belong to each string, what colour the string is and how many liberties the string has. This greatly increases the speed when detecting and removing captured groups and could provide useful


NeuralGo - \Copy of allgames9WithSymmetry.bpn - -
File View Play Go Neural Networks Database Tests Window Help
Neural1
A B C D E F G H I J K L M N O P Q R S
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
Log
For Help, press F1
NUM

```
Test: Including Area Finder facilities:
Test: o11:0.4990325

Test: The actual move was: h15
Test: Actual move NN score: 0.4949065
Actual move rank: 14 out of 315
Test: Top 6 moves are:
Test: o14:0.5227435
Test: o5:0.5189391
Test: n4:0.5163774
Test: o4:0.5141754
Test: n14:0.5113918
Test: g14:0.5106812
Test: Bottom 6 moves are:
Test: a1:0.1907316
Test: i19:0.1933791
Test: s1:0.2162859
Test: b1:0.2362468
Test: s18:0.2370356
Test: j19:0.2413257

Test: Top move suggested by BPN without al
Test: Including Area Finder facilities:
Test: i13:0.4817808

Test: The actual move was: h16
Test: Actual move NN score: 0.4657856
Actual move rank: 52 out of 314
```

Figure 9: Game walkthrough in progress

Configuration	Average Time Taken Per Move (seconds)
1. 9x9 Neural Network	0.784
2. 9x9 Neural Network + Area Finder	0.136
3. Alpha-Beta (Liberty Count)	40.712
4. 9x9 Neural Network + Alpha-Beta (Liberty Count)	9.02
5. 9x9 Neural Network + Alpha-Beta (Liberty Count) +Area Finder	1.56

Table -1

Neural Network	Average Time Per Move	Percentage of time that actual move is in top n percent				
		10%	20%	30%	40%	50%
5x5	0.576	19.2%	36%	50.4%	59.2%	64.8%
7x7	0.736	16.8%	29.6%	41.6%	60%	65.6%
9x9	0.936	12%	30.4%	39.2%	52.8%	64%
Copy of 9x9	0.912	18.4%	34.4%	47.2%	56%	68.8%
11x11	1.256	9.6%	19.2%	30.4%	35.2%	44%
13x13	1.664	18.4%	28%	36%	51.2%	61.6%
Random 9x9	0.824	4.8%	20.8%	28.8%	35.2%	44.8%

Table-2

information for additional tactically oriented modules. A graphic board display is available to easily view the program playing through an SGF and suggesting moves to compare to the actual move. The GUI is also intended to be used to watch the program play against another program using the GMP interface however the GMP protocol code has not ported well from the Java version, more on this later. The user may also play through the GUI against the neural network.

The neural network algorithms include back-propagation learning and a matrix implementation, allowing neural networks to be loaded, saved, trained and used to suggest moves for Go. A representation of a training database has been programmed so training databases can be created to a specified format using SGF files as input

The structure of the training data caused some amount of trouble due to the enormous amount that was created. For instance, taking a single SGF file of a professional game the average number of moves was around 230. This meant that for every move 6 input/output pairs of matrices were generated for plausible moves in the near future and 6 input/output pairs were generated for implausible moves. Using 5x5 networks as an example, the input matrix was 25 by 1, all calculations using floating-point numbers 32 bits long, giving 12*25*32=9600 bits of data for the input matrices of one move. Add to that the output matrices which were only 1 by 1, so that is 9600+(12*32)=9984 bits per move, or around 287Kb per game. The largest sized database I was able to generate, due to memory constraints was 10Mb, which contained the distilled knowledge of only 36 professional games to be used with a 5x5 network. Any more than this and the computer would complain about memory usage and grind to a halt, and this was with 64Mb of RAM. Even though 36 games is a fair amount, producing over 50,000 training pairs, it does limit the neural networks experience that it can draw from to only the moves in those 36 games. Whether this is enough material and a wide enough range of move types for the neural net to induce tactical/strategic guidelines from will remain to be seen after testing. A much larger spread of games would be desirable. Extracting and storing training data from a much larger number of games is one of the problems that must addressed before this particular program can reach it's full potential.

Due to the training set actually being fairly small in experience content, the networks that were trained tended to reach their capacity towards the end of the project time. This meant they were in danger of over training and becoming too specialised on just the set of games used to produce the training set. One way around this would be to use a different training set after this point has been reached, but this might cause the net to lose some of what it has learnt from the other set. Rotation of training sets appears to be a plausible alternative possibly as often as each epoch, to make certain no learning is wasted at all.

The previous version of the project used Java as the implementation language, which was fast and easy to produce a working prototype of the design at that stage. Considering the time available for that stage, 4 weeks, using Java proved to be very effective.

For this version however, it was decided to move to Visual C++ so that the program would run as fast as possible.
At the conclusion of the project the 9x9 neural network had reached a point in training where it appeared to have extracted all the useful knowledge from the training set that it could and was starting to over train. This was evident from a slight degradation of the quality of move suggestions it was making and was probably because it was extracting some sort of mistaken connections between moves resulting from using a small and restricted data set.

Attempts were made at training 11x11 and 13x13 networks but it was soon realised that these would not have enough time to train to produce any meaningful results before the end of the project.

An Area Finder network was trained for a reasonable amount of time and started to show some signs of sensible suggestions, certainly enough to warrant further investigation to ascertain how worthwhile it would be to use it in conjunction with a standard network of the sort just discussed.

Two restricted move range networks were trained, both showing promising results. The first covered the first five ply moves and second covers from five ply to ten ply. An advantage of using restricted ranges is that the training is much quicker and the networks gain a higher degree of skill in their area than a more generalised network ever would. Of course this has to be balanced by considering how specialised should these networks become and at what point do they become over specialised and so the disadvantages of specialisation outweigh the advantages.

The MTD (f) variation of the alpha-beta minimax search algorithm was implemented along with various enhancements. The algorithm works within an iterative deepening framework to be as efficient as possible. A transposition table is included to reuse nodes that have already been created and the information calculated about the node. The program also uses the best move from previous iterations as the first node to expand as this has been shown to give a considerable performance boost [Schaeffer et al.].

Enhanced transposition cutoffs [Schaeffer et al.] are also implemented which means that all nodes arising from a position are first quickly checked to see if they cause a cutoff in the tree before deep searching that branch.

The expand function which creates child nodes given a position uses a neural network to suggest a specified number of the most plausible moves as child nodes. Also two alternative evaluation functions to score the nodes are available, one that counts each sides liberties and attempts to maximise one sides liberties whilst minimising the opponents. The second one simply counts the number of stones on each side and so encourages capturing and aggressive play. A third alternative that was not implemented but was mentioned in the design section involves using a TD (λ) trained neural network as an evaluation function and the code structure written can easily accommodate it if required in the future. In fact this approach appears to be the most promising method of implementing an effective evaluation function, judging from research done by other parties [Tesauro 94].

RESULTS & DISCUSSIONS

Looking at how long it takes various configurations to reach a move decision shows us some important points and in combination with some quality of result analysis can tell us to what degree the combination of soft and hard AI has been successful and if it is worth pursuing in the future.

First of all the difference between configuration 1, which used just a 9x9 network and configuration 2 which used a 9x9 network with an Area Finder network is easily explainable. Using the coarser grained Area Finder network divides the board into 9 sectors and given the full 19x19 board selects the most appropriate sector out of the 9. Then the 9x9 network looks at all legal moves within that sector only. For the first configuration all legal moves in the entire 19x19 board must be considered so we see a logical time difference of around a factor of 9. A similar affect is seen when comparing configurations 4 and 5. The use of the Area Finder network gives a substantial speed boost without adding any unreasonable overheads. If the quality of suggestions presented by the Area Finder network can be measured and built upon then this could be an effective and efficient method of incorporating neural network technology within a Go playing program.

The use of alpha-beta search added considerable computational overheads, however that is expected considering the nature of the algorithm. The liberty count evaluation function was used in all cases.

To assess and compare the quality of the neural networks and different configurations involving either or both soft and hard AI two approaches were taken.

First, the configurations used for timing an average move were used to play proper games of Go against GNUGo 26b [GNUGo 01]. It is important to play actual games since this was the original intention of creating such a program and is really the best way of judging its success in its intended environment. The program itself still unfortunately has a few problems and bugs that were given special provision. The program did not have any method to decide when to pass, so a game would continue until GNUGo passed or until a crash occurred. Where a crash occurred it is marked in the results table as N.C. (not completed). When a game has reached an end, either on purpose or by fault, the board was scored by Jago [Grothmann 01], which functioned as arbiter between the programs. A final point of note is that the program had no knowledge of the Ko rule and as such could have broken it and forfeited the game, however such a situation did not occur in any of the test games played.

The second method was used to determine the extent and level of training achieved by the neural networks. Several measures were used and information gathered about 6 different networks. The statistics were collected as each

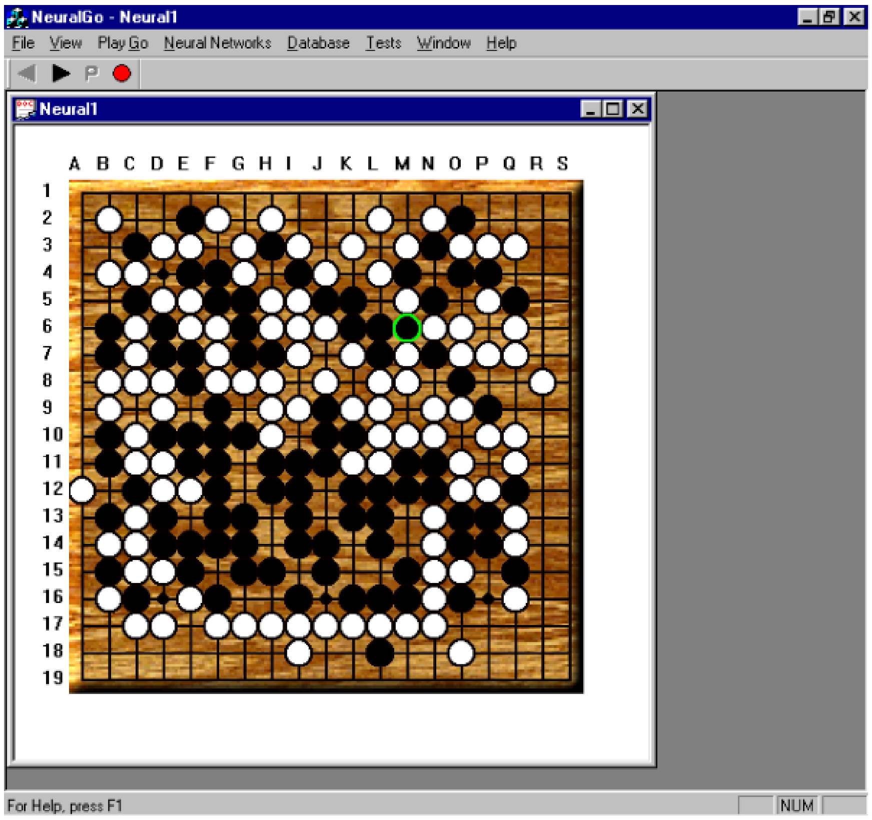

NeuralGo - Neural1
File View Play Go Neural Networks Database Tests Window Help
Neural1
A B C D E F G H I J K L M N O P Q R S
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
For Help, press F1
NUM

Figure 10: Game in progress

Configuration	Game Score (we play black), (J= Japanese, C= Chinese)
1. 9x9 Neural Network	J: B-8, W-49 C: B-106, W-146
2. 9x9 Neural Network + Area Finder	J: B-3, W-21 C: B-54, W-71
3. Alpha-Beta (Liberty Count)	J: B-9, W-12 C: B-35, W-38 * Not Compete
4. 9x9 Neural Network + Alpha-Beta (Liberty Count)	J: B-9, W-25 C: B-107, W-124
5. 9x9 Neural Network + Alpha-Beta (Liberty Count) +Area Finder	J: B-7, W-18 C: B-55, W-66

Table –3

network played through the same professional game; the average time to select a move was recorded, as was the average rank of the actual move within all the moves considered by the neural networks. To give a more detailed look at the quality of the moves being selected the percentage of time that the actual move was ranked in certain percentiles was also noted, for example for the 5x5 network the actual move was tanked in the top 10% of moves 19.2% of the time and was tanked in the 20% of moves 36% of the time. For comparison sakes a newly created, hence untrained, network was tested also, so we should expect, if training has worked at all, that the new network should have the lowest scores.

Figure 10 above shows a game in progress against GNUGo. GNUGo is playing white and the program is playing black. The move marked with a green circle is the most recent move.
Looking at the results the first thing of note is that the larger the network the longer it takes to suggest a move. This is quite expected and reminds us that even though neural networks are fast compared to other AI techniques they can still easily build up a substantial overhead that must be kept in mind and minimised wherever possible.

The average move rank column appears to paint a gloomy picture of progress but it must be made clear that the average number of moves being considered each turn was around 310. Also the lack of detail and insight that can be extracted from such a figure is partly the reason why the more indepth and informative percentile statistics were gathered. The best performing network appears to be the 5x5 with the actual move being ranked in the top 30% of moves just over half of the time.

Comparing all of these figures to the random 9x9 network shows that they all improved after training and reveals an interesting and very important point about over training. The ‘copy of 9x9’ network was an earlier version of the current 9x9 and has much better figures. This does suggest rather strongly that the current 9x9 has been over trained and the quality of its output has been degraded as a result. This also implies that some sort of peak of training can be reached and by considering the figures they also suggest that the peak does not mean getting the very best move at rank number one. Rather it suggests, perhaps viewing it optimistically, that the network realises there may not be one perfect move but maybe lots and using a neural network allows the moves to be ranked effectively as opposed to selecting a single best move. This may be the strength of using neural networks in this instance.

A rather large hindrance that should be kept in mind is the time it takes to train a network. The 9x9 took around 2 months to reach a point where it started to over train. Larger networks take proportionately more time. This meant there was not a lot of spare time for experiments and trying alternatives, so I think this will be a stumbling block for quite a while in the future.

If we now move onto the results of the test games against GNUGo we can observe several things from the scores. Both Chinese and Japanese scores are presented, with our program playing black for each game.

The first thing to note is that where the alpha-beta algorithm was included the score gap has been considerably narrowed. This would imply an overall improvement in defence and offence by the program thanks to the look ahead facilities provided by the alpha-beta routine. It also shows that although only a simple and occasionally probably inhibitory evaluation function was used, the liberty counter that a general improvement in play was fairly easy to achieve. Currently the alpha-beta is limited to a 6 move look ahead but it would probably be beneficial to make this a dynamic factor based on the line of play and resource availability.

Unfortunately the plain alpha-beta configuration would not complete an entire game so the scoring is pretty inaccurate and may not reflect the final state of the game had it reached its conclusion. This means we can see that the soft AI, neural networks, benefits from using hard AI, alpha-beta, techniques but we cannot be certain vice versa. It is possible that the neural networks may actually hinder the optimum play of the alpha-beta routine although this seems unlikely.

What more we can tell is that the addition of the Area Finder not only gives a speed advantage as discussed earlier but also increases the quality of suggestions. From observation of the games I would suggest that this is partly because a wider area of the board was played across when using the Area Finder than without, so the opponent found it harder to establish solid territories. When the Area Finder was not used the play tended to a single area and usually stayed there throughout the game, allowing the opponent to establish unchallenged territories.

All these results show that soft AI has something to offer the problem area of Go, the limitations and extent of which require further investigation, however we have made some connections, various ideas have been tried and tested and a system that can support further research has been developed and implemented. More than anything else I think, questions have been raised and pointers for promising future investigations have been found.

There could be any number of ways to combine soft and hard AI. The trick is to do it in such a way as to maximise the strengths of each and minimise the weaknesses. If in doing so the combination is greater than the parts then the job is a success. From the results it seems clear that hard AI benefits soft AI and it is a pity the reverse cannot be concretely induced from the results collected but it would be reasonable to assume so.

CONCLUSIONS AND FUTURE WORK

There are many things in this project that could be improved given time, the game of Go opens up a wide area of problems and naturally tends towards combining many different AI techniques. Much of the research carried out over the past few months only touches upon each idea, any one of which could provide a lengthy and fruitful line of research. Amongst these are combining various grain networks, how to combine them and which to use, limited range networks which tend to specialisation and of course other methods of combining soft and hard AI. There are other soft AI techniques apart from neural networks that may be worth looking at such as genetic algorithms and evolutionary programming. Ther are also other hard AI techniques such as rule based systems that could also be included to provide more direction to the soft AI, since the choice of a plausible move may be governed by the concept of what one is trying to achieve.

Interesting extensions to the ideas could include a closer look at the actual choice of neural network architecture and construction, and a more thorough review of the choices available. A deeper understanding of the way humans perceive and play Go could be developed from further work, perhaps leading to a better understanding of human pattern recognition. Go is an excellent environment that provides many opportunities for developing knowledge, not just of artificial intelligence and how to use it, but also of human intelligence and how the two compare, and it may also provide some impetus for the acceleration of development of soft AI techniques such as neural networks and other machine learning processes.

REFERENCES

1. Allis, L.V., Van Der Herik, H.J., Herschberg, I.S., "Which Games Will Survive?", Heuristic Programming in Artificial Intelligence 2 - The Second Computer Olympiad, pages 232 – 243, Ellis Horwood, 1991.

2. BGA 99, British Go Association (BGA), 1999, Available on the Internet at http://www.britgo.org/

3. Burmeister, J, "An Introduction to the Computer Go Field", and,
Burmeister, J., Wiles, J., 1995, "Associated Internet Resources", Available on the Internet at http://www2.psy.uq.edu.au/~jay/go/CS-TR339.html

4. Callan, R., "The Essence Of Neural Networks", Prentice Hall, 1999.

5. Chikun, C., "Go: A Complete Introduction to the Game", Kiseido Publishing Company, 1997.

6. Ladder 2001, "Computer Go Computer Go", See http://www.cgl.ucsf.edu/go/ladder.html

7. Computer Go Mailing List, 2001, See http://www.cs.uoregon.edu/~richard/computer-go/index.html.

8. Enzenberger, M, "The Integration of a Priori of Knowledge into a Go Playing Neural Network", 1996, Available on the Internet at http://www.uni-muenchen.de.

9. Fotland, D, "Knowledge Representation In The Many Faces Of Go", 1993, Available on the Internet at [GNUGo 01] GNU Go latest version can be found at http://freedom.sarang.net/software/gnugo/beta.html

10. Grothmann, R, "Jago", 2001, available on the internet at http://mathsrv.ku-eichstaett.de/MGF/homes/grothmann

11. “GTP2001”, Go Text Protocol information can be found 2/3 down the page at http://freedom.sarang.net/software/gnugo/beta.html.

12. Hollosi, A, “SGF User Guide”, 1999, available on the Internet at http://www.red-bean.com/sgf/user_guide/index.html.

13. Huima, A, “A Group-Theoretic Hash Function”, 1999, available on the Internet at http://people.ssh.fi/huima/compgo/zobrist/index.html.

14. Muller, M, “{Playing It Safe: Recognizing Secure Territories in Computer Go by Using Static Rules and Search”, 1997, available on the Internet at,
http://www.cs.ualberta.ca/~mmueller/publications.html.

15. Plaat, A, “MTD(f), A Minimax Algorithm Faster than NegaScout”, 1997, available on the Internet at,
http://www.cs.vu.nl/~aske/mtdf.html.

16. Owsnicki-Klewe, B, “Search Algorithms”, 1999, available on the Internet at http://www.informatik.fh-hamburg.de/~owsnicki/search.html

17. Tesauro, G, “TD-Gammon, a self-teaching backgammon program, achieves master level play”, Neural Computation, Vol. 6, No.2, 1994.

18. Schaeffer, J and A. Plaat, “New Advances In Alpha-Beta Searching”.

19. Schraudolph, N, Dayan, P, Sejnowski, T, “Temporal Difference Learning of Position Evaluation in the Game of Go”, Neural Information Processing Systems 6, Morgan Kaufmann, 1994, available on the Internet at ftp://bsdserver.ucsf.edu/Go/comp/td-go.ps.Z

20. Van Der Steen, J, “Go Game Gallery”, 2001, available on the Internet at http://www.cwi.nl/~jansteen/go/index.html

21. Wilcox, B, “The Standard Go Modem Protocol – Revision 1.0”, available on the Internet at http://www.britgo.org/

Other publications that may be of interest but are not cited in this report:

22. Dahl, F A, “Honte, A Go-Playing Program Using Neural Nets”, 1999.

23. Muller, M, “Review: Computer Go 1984-2000” 2000.

24. Stoutamire, D, “Machine Learning, Game Play and Go”.