Evolving Go Playing Strategy in Neural Networks

Paul Donnelly
Patrick Corr
Danny Crookes

Department of Computer Science,
The Queen's University of Belfast, BT7 1NN, UK.
Email: donnelly@elegabalus.cs.qub.ac.uk

Abstract

The game of Go is an ideal problem domain for exploring machine learning: it has simple rules yet requires more and more complex strategies to play well as the board size is increased. Despite much effort, existing Go programs, which have largely employed knowledge based and symbolic AI techniques, have failed to achieve a standard much above an average human amateur. This paper examines the possibility of using evolutionary techniques in the specification of neural networks, to play Go. This is a much more difficult problem than the 'toy-problems' with which evolved neural networks have been successful to date. Results of a simple experiment using a Genetic Algorithm to evolve a Go position evaluation network are reported, the factors affecting performance are discussed, and proposals are made for more sophisticated networks and evolutionary schemes that may potentially be better suited to complex problems that involve an interaction of pattern recognition and serial reasoning.

1 Introduction

Recently, there has been much interest in evolutionary techniques as learning algorithms for neural networks [1]. In addition to the obvious biological appeal of the idea, evolutionary techniques such as Genetic Algorithms have an advantage over more popular supervised learning techniques in that they can be used to train neural networks with unrestricted architectures and neuron types [2,3], and their 'blind search' characteristic means that networks can be evolved to perform unsupervised learning tasks where no immediate error information can be provided about the network output.

Performance depends on the existence of satisfactory networks within the search space of allowed networks, and on the effectiveness of the evolutionary algorithm in finding them. Justification for the belief that neural networks can be useful for Go is given in section 2. The effectiveness of evolutionary algorithms in searching the space of allowed networks is dependent on the characteristics of the search space, on the genetic encoding chosen, and on the genetic operators used. Most important is the quality of building blocks to be recombined by the crossover operator since this is what distinguishes evolutionary algorithms from other randomised search techniques [4].

The size of the search space can be reduced by placing restrictions on the networks that can be evolved such as specifying a fixed architecture and evolving only the connection weights. This may make the evolutionary algorithm's task easier initially, but may preclude convergence to a better solution with a different architecture. In fact if we are to copy nature, we should have the evolutionary algorithm specifying more rather than less. Consider that DNA, the genetic string used to specify all living things, must specify, in addition to the details of the functional units of an organism, its very own decoding mechanism. So we might have parameters for neuron activation functions and weight update rules (if phenotype learning is also allowed for).

Recently, several schemes have been proposed for indirect genetic specification of neural networks inspired by cell-splitting as occurs during the development of the brain [5,6]. These schemes are a step further in biological mimicry, but can have practical advantages including better scaling, better generalising, and avoidance of the competing conventions problem [1]. However to work, these schemes require modifications to the genetic algorithms such as sub-tree crossover. DNA inspired models such as Hofstadter's typogenetics [7] could possibly be the basis for an even further step in biological mimicry if applied to neural network evolution.

Our preliminary experiments reported in this paper, involve only small 3-layer feed-forward networks, optimised using a GA with a simple weight encoding scheme. Future work will investigate more sophisticated schemes.

2 Game Theory, Neural Networks, And Go

Most computer game playing algorithms use move-tree minimax search along with static position evaluation functions to pick the best move from a list of alternatives in any particular situation. If the evaluation function were perfect, ranking positions in order of highest probability of winning, there would be no need to search any deeper than a single ply - simply try each legal move, evaluate the resulting positions, and pick the most favourable. However for non-trivial games, the evaluation function can only approximate the value of a position. Using lookahead and seeking quiet or quiescent positions before evaluating can compensate for error in the evaluation function. Augmented with αβ-pruning, such searching techniques have been particularly successful in computer Chess [8].

There are several reasons why Go is not amenable to this approach. Firstly, for normal board sizes the number of legal moves at each position or branching factor is much higher than in Chess. Secondly, many situations in Go require very deep reading in order to assess correctly (since the pieces don't move around, human players can look ahead more reliably than in Chess). Thirdly, there is no simple evaluation function that could be applied to the leaf positions of a minimax search in Go such as piece count in Chess.

Human skill in games is more dependent on positional evaluation than on lookahead [9]. This is probably due to the massively parallel but slow nature of brain computation. The evaluation functions used in computer game-playing algorithms often assess positions by linear combination of a set of features, for example in Chess there might be piece count plus bonuses for piece mobility, passed pawns etc. minus penalties for enemy pieces around the king, isolated pawns etc. For many games, such heuristics work quite well, however a criticism is that they are chosen somewhat ad-hoc. The more subtle we try to make them, the more difficult it becomes to set the parameters - values of bonuses and penalties. Another criticism is that expert human players do not always accumulate bonuses and penalties of positional features, but know when one or two features will override all others and decide the result of a game.

Neural networks have several potential advantages over rule based approaches to evaluating game positions. Firstly, network learning algorithms are designed to automatically map their input space to a feature space, and should be capable of discovering and weighting salient features of game positions. Secondly, these features can be combined non-linearly if appropriate. Neural networks are likely to be especially suitable for Go since Go position evaluation involves explicit pattern matching [10].

To avoid the problem of large branching factor and improve the efficiency of the αβ-pruning algorithm, most existing Go programs use a set of pattern templates to suggest promising moves for further analysis. For example, David Fotland's program, ' Many Faces of Go' associates a small move-tree with each pattern and searches this tree when a match occurs [11]. Neural networks could perform a similar task by ranking each legal move, with the benefit that the move suggestion is learned rather than specified.

3 The Rules Of Go

To aid the discussion, a brief informal description of the rules of Go is included. An excellent introduction to the game can be found in [12].

Go, like Chess, is a two player, deterministic, perfect information, zero sum game. It is purely a game of skill with no element of chance. Players alternate placing black and white pieces called stones on the board with the object of surrounding more board area (*territory*) than the opponent. The first player takes the black stones and the second player takes white. Stones are placed at the intersections of a grid (usually of size 19 x 19). Once a stone has been placed it is not subsequently moved (except that it may be captured). Stones of the same colour which are horizontally or vertically adjacent are said to be connected, and form a *string*. A string may be captured if it is completely surrounded, when all adjacent empty intersections, called *liberties*, are occupied by stones of the opposite colour. When a string is captured it is removed from the board.

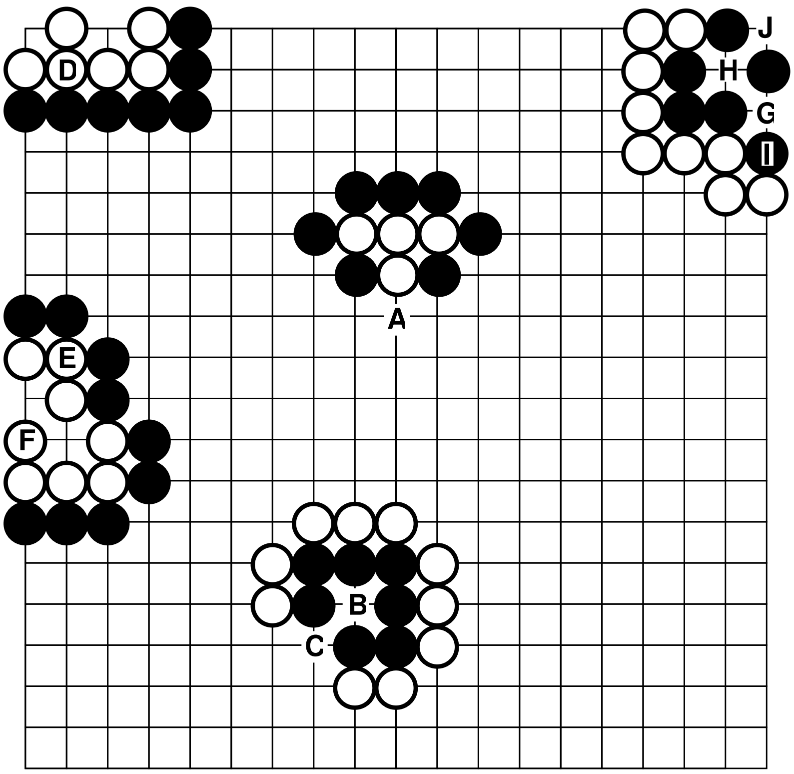

figure 1 - A Go board with examples illustrating the rules (see below)

In figure 1, the T-shaped white string has only one liberty left at 'A'. If Black plays at 'A' the white string is surrounded and captured.

To capture the black string, around 'B', White must play at both 'B' and 'C'. However 'B' is itself fully surrounded by black stones, called an *eye* of the black group. White cannot play here for such a move would be suicide and would be immediately removed from the board. White can only play at 'B' after playing at 'C'. In this case, playing at 'B' removes the final liberty from the black stones and kills the black string.

The white string marked D has two eyes and cannot be captured. Black cannot play in either eye since in neither case would it remove the last liberty of the white string. The white string is said to be *alive*.

On the left side, the two white strings 'E' and 'F' are not horizontally or vertically connected, but nevertheless lie in close proximity and form a single *group* of white stones. This group is alive since it has two eyes - above and to the right of 'F'. The rule is that two (or more) eyes gives unconditional life.

The black group on the top right seems to be alive since there are 3 spaces in the configuration. However the point 'G' is a *false eye*. If it is black's turn, black can give life to the corner by placing a stone at 'G' and the black group would then have real eyes at 'H' and 'J'. If it is White's turn, white can play at 'G' capturing the single black stone marked 'I' planning to capture three black stones at 'H' next. However it is now Black's turn and to prevent white from capturing at

'H', Black will wish to recapture the white 'G' stone by playing back at 'I'. This reveals a difficulty with the rules as explained so far. If Black recaptures at 'I' then the position of two moves previous will be repeated and the same sequence of two moves could continue for ever, neither player wishing to concede. For this reason the *ko* rule must be introduced which in its simplest form states that the same global board position may not be repeated more than once in a game.

At any stage during the game, a player can elect to pass, (when the player considers that all groups are safely dead or alive and there are no territory points left to contest). After two consecutive passes, one by each player, the game is over and scoring begins.

In fact at the end of the game, there are 2 commonly used scoring methods - 'Chinese counting' and 'Japanese counting'. The result is almost always identical but Chinese counting is simplest to describe: remove any dead groups then fill in each player's territory with stones of the same colour. The player with the most stones on the board wins. (To make the game even, White often has 5.5 points added in compensation for playing second. The .5 is to prevent draws).

It should be noted that there are some rare awkward situations mainly involving multiple kos which the commonly used rules don't handle satisfactorily. After consideration of these defects, Ing Chang-ki, sponsor of the 'World Computer Go Congress', has formulated a new version of the rules which are used at all Ing-sponsored human and computer tournaments [13].

4 Experiments

Some preliminary experiments have been carried out using a Genetic Algorithm to evolve fixed architecture 3-layer feed forward neural networks to perform 9x9 Go-position evaluation. This is the size often used by human beginners learning the tactical elements of the game, and helps keep the complexity down and speed up the experiments.

The input layer is a grid of 9x9 groups of 3 mutually exclusive neurons. Each group of 3 represents the state of a board location - {black_stone, white_stone, liberty}, with the activation patterns (1,0,0) (0,1,0) and (0,0,1). If it is White's turn to play then the representation of black and white stones are switched to take account of the colour reversal symmetry of the Go board.

The input layer is fully connected via an array of weights to a hidden layer which is a square grid of 9 real-valued sigmoidal neurons. The neurons in the hidden layer are fully connected via an array of weights to a single summation output unit. The value of the output unit is used as a measure of favourability of the Go-position presented at the input units.

To choose the best move in a particular position, a one ply search of the move tree is performed with the network applied to all the positions resulting after a single move (including the pass move). The move chosen is the one with the highest evaluation.

The array of connection strengths between the input and hidden layer and between the hidden and output layer specify the network. These are encoded as a genetic string of real numbers. The weights relating to each hidden unit are stored together, and the encoding takes account of the 8-fold reflective and rotational symmetry of the Go board. These optimisations reduce the string length and try to help genetic crossover.

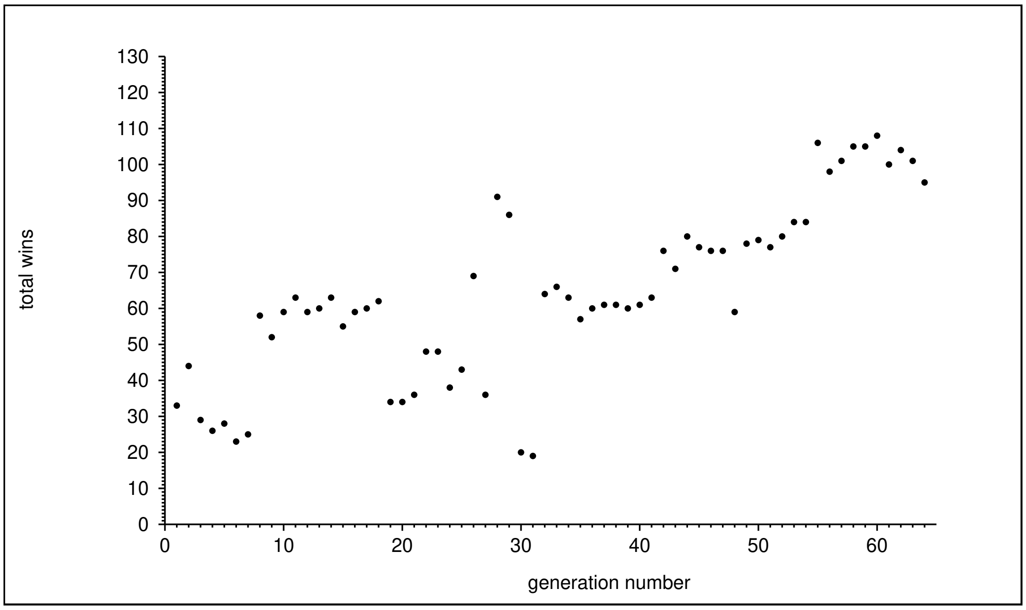

figure 2 - number of wins in all-play-all tournament among generation winners

A Genetic Algorithm with crossover and mutation is used to train the network. A population of 32 networks is played off in an all-play-all tournament, where every network plays every other network twice, once as black and once as white. The network loosing most games in each generation is overwritten with the network winning most games. Initially it had been intended to perform a two-pronged knockout tournament giving an overall winner and an overall looser, but all-play-all was eventually used to give more reliable information about the relative fitness of networks.

At each generation, the network winning most games is stored. At the end of 64 generations, the generation winners were played against one another in a final all-play-all to determine whether the genetic algorithm had been successful in improving the networks. The graph of figure 2 shows the number of wins plotted against generation number.

It can be seen that there is a general trend of improvement, but that this is superimposed with a lot of local variability. In terms of objective performance, the resulting networks were very poor, and were unable to beat even the weakest available Go programs.

The experiment takes 2 days on a Sun 4 machine. One of the main factors affecting performance seems to be the difficulty of obtaining reliable fitness information when using win/loss results against other networks. With a small population size, the networks are liable to adapt to one another's particular weaknesses rather than learn general concepts of the game. While this may lead to eventual improvement, it can be very slow and unreliable.

5 Discussion

The neural network, the network evolving scheme, and the Go playing framework presented here are all preliminary and rather simple. There are many areas where improvement is necessary before the idea of evolving neural networks to play Go can be tested properly.

A major criticism is that these networks have no way to take advantage of the significant degree of spatial localisation and related translation invariance found in positional features of Go. In [14], Schraudolph et al. have designed a network architecture that takes account of translation invariance as well as the reflective, rotational, and colour-reversal symmetries.

A single hidden layer is unlikely to be the best architecture for position evaluation in Go. In order to make a reasonable evaluation, the neural network would have to be able to discriminate between live and dead groups, and for the live ones, estimate how much territory they surround, or are likely to surround by the end of the game. To recognise safely living groups requires (at least) recognition of connectedness and eyes. This suggests that features relevant to positional evaluation cannot be extracted directly from the pattern of stones on the board, and that several hierarchical layers of hidden units detecting progressively more abstract features are necessary.

In fact, it has been shown [15] that connectedness - determining whether two strings can be connected despite the opponent playing to keep them separate - is p-space hard. To properly recognise such features requires an iterative procedure. For this reason, recurrent networks may be more powerful than feed-forward networks for Go.

An interesting approach to Go can be found in [16] where Allan Scarff has used a cellular automaton which is similar to a recurrent single layer neural network. This iterates until an equilibrium is reached when the answer (in this case suggested move), can be read off. We propose to investigate neural networks operating in a similar manner, evolved using evolutionary algorithms. Neurons are incrementally updated when their inputs change, and the network iterates until it reaches an equilibrium. The network will arrive at a consistent assessment and hopefully accurate evaluation of the input position. It may be necessary to guarantee that equilibrium is reached, by some damping mechanism.

However it would seem from the experiments carried out so far that learning by playing other networks may be too slow. It would be useful to provide some mechanism for additional phenotype learning which is quicker and has the additional advantage of giving the Baldwin effect on evolution [5]. Perhaps, networks could first be trained on a set of problems or professional games, before playing against one another to decide which networks are selected. Alternatively, Temporal Differencing where lookahead is used to provide the phenotype learning goals [14], could operate while the network plays.

When such possibilities are allowed, the search space is greatly increased, and the evolutionary algorithm's task becomes more difficult. To counteract this, we would like the evolutionary algorithm to make better use of the information it can glean from sampling the search space. One of the ideas is to try to include knowledge about the problem by biasing the genetic encoding, for example towards local neighbourhood connections between neurons in adjacent layers.

6 Conclusion

Go-playing algorithms have proved to be very difficult to construct. Most of the strongest current Go programs are knowledge based systems, fully specified by their programmers. As more knowledge is added to these systems, it becomes increasingly difficult to manage the complexity, and it is doubtful that this approach will ever lead to programs comparable with the best human players. Programs based on automatic machine learning are a promising alternative, although the difficulties involved are formidable.

This paper has argued that neural networks should be suitable for positional evaluation and move suggestion in Go-playing algorithms. Evolutionary algorithms are useful for specifying neural networks because they impose less restrictions on the network architecture and operating mode. However when the restrictions are reduced, the search space becomes larger, and the evolutionary algorithm must search it more efficiently to make it worth while. Critical aspects are the network encoding, the use of suitable crossover operators, and methods of accurate fitness evaluation.

Future work will investigate more complex neural network specification including varying degrees of freedom to specify the architecture and neuron type genetically. It is expected that more complex networks will require more complex indirect geneotype to phenotype mappings for their successful genetic specification. The genotype to phenotype mapping rules chosen also affect the suitable choices for genetic operators and evolutionary scheme in general.

To compare differing neural network specifications and evolutionary schemes, the winner in each generation is stored along with the execution time taken to produce it. The best networks produced by each scheme after various amounts of execution time can be played against one another. Provided each scheme is reasonably optimised, execution time vs strength is an informative measure of algorithm performance.

References

[1] Schaffer D.J.,Whitley D.,Eshelman L. J.; 'Combinations of Genetic Algorithms and Neural Networks: A Survey of the State of the Art', proceedings, COGANN 1992.

[2] Bornholdt S., Graudenz D.;'General Asymmetric Neural Networks and Structure Design by Genetic Algorithms', Neural Networks Vol 5, 327-334, 1993

[3] Janson D.J., Frenzel J.F.; 'Training Product Unit Neural Networks with Genetic Algorithms', IEEE Expert. 26-33, Oct 1993

[4] Goldberg, D. ; 'Genetic Algorithms in Search, Optimization and Machine Learning' Addison Wesley, 1989

[5] Gruau F., Whitley D.; 'The Cellular Development of Neural Networks: 'The Interaction of Learning and Evolution', LIP Ecole Nationale Superieure de Lyon, Jan 1993

[6] Boern E.J.W, Kuiper H., Happel B.L.M, Sprinkhuizen-Kuyper I.G.; 'Designing Modular Artificial Neural Networks', Leiden University, Netherlands, tech rpt. 1993

[7] Hofstadter D.R.; 'Gödel, Escher, Bach: An Eternal Golden Braid', Basic Books, 1979

[8] Hsu F., Anantharaman T.,Campbell M.,Nowatzyk A., 'A Grandmaster Chess Machine', Scientific American, 263 no. 4 pp 44-50, 1990.

[9] De Groot A.; 'Thought and Choice in Chess', Mouton & Co. Paris, 1965

[10] Reitman W, Wilcox B; 'Pattern Recognition and Pattern-Directed Inference in a Program for Playing Go'. Pattern-Directed Inference Systems (D. Waterman and F. Hayes-Roth, Eds). Academic Press, 1978

[11] Fotland D., 'Knowledge Representation in the Many Faces of Go', (available on Internet via anonymous ftp to bsdserver.ucsf.edu, file Go/comp/mfg.Z), 1993.

[12] Iwamoto K.; 'Go for Beginners', Ishi Press, Tokyo, 1972

[13] Ing Chang-ki; ' Ing's SST laws of Wei ch'i ', Ing Chang-ki Wei-ch'i Educational Foundation, Taipei, 1991

[14] Schraudolph N., 'Temporal Difference Learning of Position Evaluation in the Game of Go'. Advances in Neural Information Processing 6, 1994.

[15] Lichtenstein D., Sipser M., 'Go is polynomial space hard. Journal of the ACM', Vol.27, No.2, pp.393-401, 1980.

[16] Durham T., 'A program with a touch of Zen', Computing Horizons, Addison Wesley, pp 183-188, 1985