CWU-Chess: An Adaptive Chess Program that Improves After Each Game

Joseph Lemley*, Razvan Andonie† ‡, Ashur Odah†, Pushpinder Heer†, Jonathan Widger†,
Berk Erkul†, Lukas Magill† and Kyle Littlefield†
*School of Electrical and Electronic Engineering
National University of Ireland, Galway, Galway Ireland
†Department of Computer Science
Central Washington University, Ellensburg Washington, USA
‡Electronics and Computers Department
Transilvania University of Braşov, Romania

***Abstract*—Most approaches to computerized chess involve some variation of brute force, lookup tables, and Alpha Beta pruning to reduce the width of search trees. Given today's extensive computational power, this is a reasonable approach to designing a program that will win games. Since this method is very different from (and much less efficient than) the way that humans play chess and other similar strategy games, it is desirable for those interested in artificial intelligence to investigate other approaches that rely less on brute force.**

Our solution was to develop an adaptive program that uses a genetic algorithm in combination with a neural network that can learn after each game it plays. The entire genetic algorithm/neural network component is part of the evaluation function used to rank each board. Given a board configuration, ten attributes are evaluated, each of which is used as the input to the Neural Network. The weights used in the neural network are optimized using the genetic algorithm. Each time the evaluation function is called, the neural network outputs the value for a given board configuration, which will then be used in a recursive search algorithm that uses Alpha-Beta pruning. The weights are adjusted at the end of the game based on its outcome. Sets of weights and game scores are broadcast over a network to enable fully distributed learning.

I. Introduction

Since the Kramnik vs Deep Fritz match of 2006 [1], chess programs have proven able to defeat the best human opponents, reliably performing at a grandmaster level. Traditionally chess programs are written using dictionaries of opening moves, endgames, and min-max trees with millions of board configurations evaluated per second. Although this approach is very successful at creating winning chess programs, it does so at high computational cost. Human players evaluate several orders of magnitude fewer board configurations, and have only recently been outmatched by such methods.

In recent years Deep Neural Networks (DNN) have made strides in solving problems that previously were considered impossible [2] and such methods have been applied to tasks ranging from speech recognition to driver monitoring, super-resolution, and playing games like GO[3].

Until very recently, the use of Neural Networks (NN) for games like chess were not of significant interest to the research community because classic techniques could so trivially outperform Neural Networks. For this reason, games such as GO were typically the subject of NN, research because it is more resistant to exhaustive search techniques.

Recently, Alpha Zero [4] has generated significant attention for their Recurrent Neural Network (RNN) that learned to play chess without prior knowledge. The program learned to improve after each game by self play, and was able to surpass chess engines that used more classical approaches in play.

However, there have been several previous approaches that used Deep Learning and Neural Networks with some success before modern hardware and deep learning libraries enabled Neural Networks to match or surpass brute force methods.

In this paper, we describe a self training chess program developed before the most recent wave of Deep Learning progress, at Central Washington University in Ellensburg Washington, USA, as part of a series of undergraduate research projects that, although publicly presented, had yet to be published as an academic paper.

II. Related Work

The most obvious approach to writing a program that can play chess is to calculate every possible move tree, starting at the current board position until the closest checkmate is found similarly to how other games of perfect information, such as Tic Tac Toe [5] and recently Checkers [6] where the ideal move is known for every configuration [7]. If this approach was feasible, chess would be considered solved and the best chess engines would simply contain lookup tables that told them the best move for each board configuration.

Another approach is to use such a game tree, but only to a certain depth, and provide a board evaluation metric that can be used to score a board on some reasonable criteria (such as the number of pieces on both sides) unless a checkmate is present in the search tree. The problem with this approach is that choosing the correct evaluation criteria is not obvious,

This research is funded under the SFI Strategic Partnership Program by Science Foundation Ireland (SFI) and FotoNation Ltd. Project ID: 13/SPP/I2868 on Next Generation Imaging for Smartphone and Embedded Platforms. This work is also supported by an Irish Research Council Employment Based Programme Award. Project ID: EBPPG/2016/280.

and the earlier in a game, the more difficult it is to judge the best move. Most chess programs use this approach combined with a dictionary of openings and endgames, which allow the computer to play well even when the search depth is relatively low.

In contrast, methods such as [4], and the method suggested in this paper learn to play without any prior knowledge beyond the rules of the game (they may not make illegal moves).

NeuroChess [8] worked by learning board evaluation functions with an inductive Neural Network and temporal differencing solely from end games.

An evolutionary approach was used to tune the evaluation function of a chess program in [9], but no neural neural network or distributed architecture was used. Similarly, a differential evolution approach was used in [10] for evaluation function tuning.

In [11] a three layer feed-forward neural network was trained to play chess endgames with backpropogation used to adjust weights.

In [12], a neural network was trained by an evolutionary strategy on a single computer. Although the network design, type of Neural Network used, and the Genetic Algorithm used differed, this method is the closest to ours in the literature.

III. CWU-CHESS

The goal of CWU-CHESS was to develop a chess program that could learn to improve its game after each play.

Our solution was to develop an adaptive program that used a genetic algorithm in combination with a Neural Network that can learn after each game it plays. The entire Genetic Algorithm/Neural Network component is part of an evaluation function used to rank each board configuration with a score.

This score is evaluated using 10 attributes, each of which is used as the input to the Neural Network. The weights used in the Neural Network are optimized using the Genetic Algorithm. Each time the evaluation function is called, the Neural Network outputs the value for a given board configuration, which will then be used in a minmax tree search (see figure 2) with Alpha-Beta pruning (see figure 1). The weights W_i are adjusted at the end of the game based on its outcome.

Arena is a chess interface that is supported by approximatly 250 chess tools. It allows chess programs to be compared, to compete against eachother, and supports the UCI and Winboard protocals.

A version of CWU-CHESS was written with the UCI interface and was playable on Arena [13]. The free and publicly available UCI and Winboard protocols are used for the communication between CWU-Chess and Arena GUI. CWU-Chess can play against any of the Arena supported engines, against itself, and against a human partner. However, for compatibility reasons, the learning components (The Neural Network and Genetic Algorithm) are not retained in the Arena version.

Since a genetic algorithm is used to find the optimal set of weights for the evaluation function of the search procedure, the

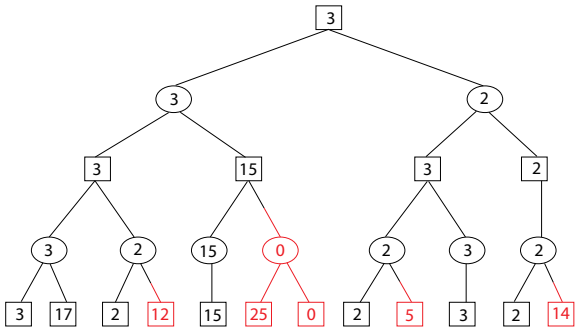

Fig. 1: Diagram of Alpha Beta search tree, showing example scores from an evaluation function. Leaf nodes are end game states or the state of the board at the search depth limit.

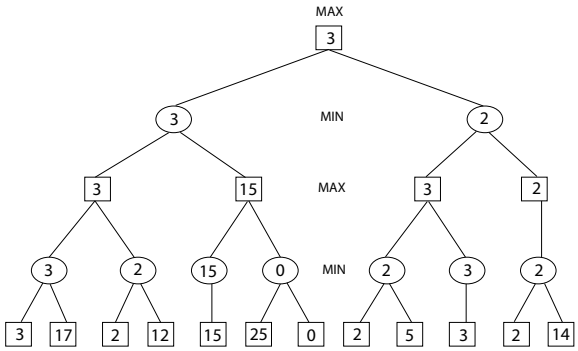

Fig. 2: Example Diagram of A minmax search tree, showing example scores from an evaluation function.

chess program is capable of learning in any of three possible modes.

1) The program plays multiple games with random weights. The weights are then ranked according to how well they score in games played against each other. To save time we made a 55 PC cluster using the sockets object included in .NET. The genetic algorithm ran on one computer and sent weight lists to the remaining 54 computers to play chess games with and return the result of the game (a win, a draw, or a loss.). The focus of this paper is on this self learning approach.
2) The chess game learns when the game is playing against a user using a different method. In this case the fitness value for a set of known good weights (determined using the first method) are recalculated each time the weight set chosen to play the user wins. This makes the program able to learn better chess strategies by playing with

users. In this way, the program can continue to learn indefinitely.
3) The third way is identical to the second except external programs are used in place of a user.

A. Genetic Algorithm

The chess program uses a genetic algorithm in place of SGD (Stochastic Gradient Descent) for training. In this approach, each set of 10 weights is regarded as an individual in a population. A set of weights can also be regarded as a specific evaluation strategy in the context of chess. The following steps are performed to calculate the weights of the individuals in the next generation:

1) 100 sets of random weights are generated and used to play 10 matches against randomly selected opponents.
2) A fitness value is calculated for each set of weights based on the total number of wins and losses.
3) Choose individuals that will be represented in the next generation based on their fitness value while giving preference to individuals with higher fitness. A straightforward roulette-wheel selection operation is used for this task. Every set of weights has a chance to be selected
4) Perform crossover on pairs of the previous individuals from random crossover points.
5) Searching from the first bit of the first individual to the last bit of the last individual, one or more bits are flipped (mutated) based on a probability function. In our experiments a mutation rate of $\frac{1}{769}$ was used, meaning that for every weight there was 0.13% chance of a bit within that number changing.
6) Repeat steps 1-5 until a suitable level of play is reached or until a set number of generations has passed, in this case 10 generations. 10 Generations was chosen as the stopping point because by the 10th generation it had been shown that play style dramatically improved.

B. Evaluation Function and Neural Network

The chess program uses a Neural Network to make decisions on what move to make. A tree of a fixed depth containing possible legal future board positions is created by use of Min-Max with Alpha-Beta pruning. Each leaf node is evaluated using the Neural Network as an evaluation function.

A Neural Network is used to give a score to each board configuration using the following 10 well-studied [12], features as input:

1) Bishop Pair: A score based on the number of bishops on either side.
2) Double Pawns: The number of White's double pawns with respect to the number of Black's.
3) Numerical Advantage: A value assigned to the advantage of having more pieces than the opponent.
4) Isolated Pawns: The number of unprotected pawns.
5) Pawns Advance: The difference of the sum of pawn row numbers of white with respect to black.
6) Passed Pawns: The number of pawns for which there are no opposing pawns in front or adjacent which could prevent it from advancing.
7) Mobility: The number of squares not dominated by the opponent that can be occupied.
8) Defensive Coordination: The number of pieces protecting the current players pieces.
9) Center Control: The number of pieces controlling the center of the board.
10) King Safety: The number of pieces that can put the opponent's king in check, as illustrated in figure 3.

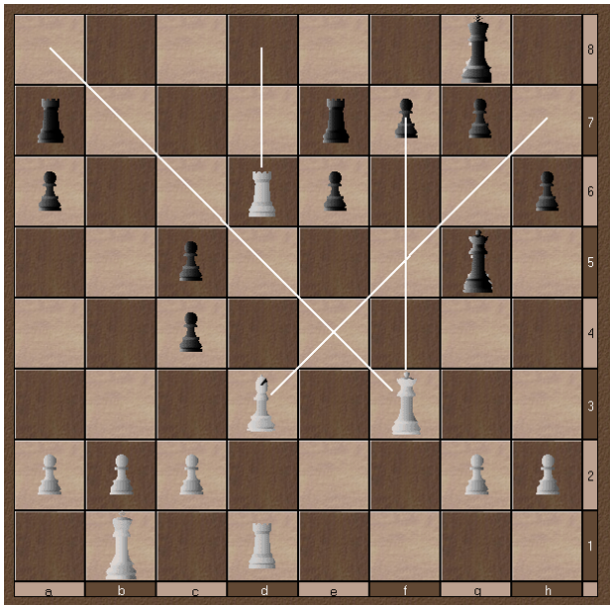

Fig. 3: The King Safety metric illustrated on the CWU-CHESS user interface. Notice that there are 3 pieces that can put the king in check, and they can do so in 4 different ways. The value of the King Safety metric for the white player in this case will be 3. This number would be one of the 10 inputs to the Neural Network and its importance relative to the other scores (as determined by the Neural Network) will be used in determining the score given to the board in the alpha-beta search tree.

The Neural Network utilized the rectifier nonlinearity(RELU) defined as max(0,x). [14][15] as its output layer.

In playing chess, the depth(or number of moves ahead) that one searches while playing is correlated with skill[16]. Traditional chess engines typically look at several orders of magnitude more board configurations, and greater search depth, to be equal to humans who search a much shallower tree [17].

To allow speedy training and enable more human like play and learning, the search depth for the game tree was limited to two, meaning that the computer player could only anticipate up to two moves ahead.

C. Cluster

The program was implemented entirely in C# using the .Net framework. Custom Neural Network, Genetic Algorithm, and high level networking code over TCP/IP sockets were written.

Experiments were repeated multiple times for 1000 total games. A client-server architecture was used with the chess game and the Neural Network running on the client computers while the Genetic Algorithm ran on the server. The random weights were generated by the server and sent to waiting clients. After each game, the client computers would report back either win, draw, or loss for the sets of weights they were given. To ensure timely training, any games lasting over 7 minutes were terminated by the server and marked as a draw.

IV. Conclusion and Results

In this paper a method for creating a chess program that learns to improve with each game was described.

It was observed that the fitness function of the Genetic Algorithm improved with each generation, and by this we deem the network to have improved as this score increased. By the 10th generation the population was dominated by a single set of weights signifying that the system had encountered a local or global optima. On average, evolved strategies won four times as often as random strategies.

With Genetic Algorithms the mutation rate prevents getting stuck in a local minima but it does so by destroying data. The mutation rate used in this study was experimentally identified by observing the convergence and by visually inspecting population samples.

At the beginning, the chess program played randomly, making nonsensical moves but as training progressed the moves made resembled those that would be chosen by a beginning human player.

The program showed significant improvement in playing style within the first four generations. Within the first 2 generations most games were playing similar strategies with just one or two variables being modified per generation. It has been demonstrated that concepts from Neural Networks and Genetic Algorithms can be used to learn strategies in a complex game such as chess even without using the previous experience of human players.

References

[1] M. Newborn, "2006: Deep fritz clobbers kramnik, 4–2," in *Beyond Deep Blue*. Springer, 2011, pp. 141–148.

[2] J. Lemley, S. Bazrafkan, and P. Corcoran, "Deep learning for consumer devices and services: Pushing the limits for machine learning, artificial intelligence, and computer vision." *IEEE Consumer Electronics Magazine*, vol. 6, no. 2, pp. 48–56, 2017.

[3] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton *et al.*, "Mastering the game of go without human knowledge," *Nature*, vol. 550, no. 7676, p. 354, 2017.

[4] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel *et al.*, "Mastering chess and shogi by self-play with a general reinforcement learning algorithm," *arXiv preprint arXiv:1712.01815*, 2017.

[5] H. J. Van Den Herik, J. W. Uiterwijk, and J. Van Rijswijck, "Games solved: Now and in the future," *Artificial Intelligence*, vol. 134, no. 1-2, pp. 277–311, 2002.

[6] J. Schaeffer, N. Burch, Y. Björnsson, A. Kishimoto, M. Müller, R. Lake, P. Lu, and S. Sutphen, "Checkers is solved," *science*, vol. 317, no. 5844, pp. 1518–1522, 2007.

[7] T. J. Schaefer, "On the complexity of some two-person perfect-information games," *Journal of Computer and System Sciences*, vol. 16, no. 2, pp. 185–225, 1978.

[8] S. Thrun, "Learning to play the game of chess," in *Advances in neural information processing systems*, 1995, pp. 1069–1076.

[9] G. Kendall and G. Whitwell, "An evolutionary approach for the tuning of a chess evaluation function using population dynamics," in *Evolutionary Computation, 2001. Proceedings of the 2001 Congress on*, vol. 2. IEEE, 2001, pp. 995–1002.

[10] B. Boskovic, S. Greiner, J. Brest, and V. Zumer, "A differential evolution for the tuning of a chess evaluation function," in *Evolutionary Computation, 2006. CEC 2006. IEEE Congress on*. IEEE, 2006, pp. 1851–1856.

[11] J. Si and R. Tang, "Trained neural networks play chess endgames," in *Neural Networks, 1999. IJCNN'99. International Joint Conference on*, vol. 6. IEEE, 1999, pp. 3730–3733.

[12] A. Carrascal, D. Manrique, and J. Rios, "Neural networks evolutionary learning in chess game." in *MLMTA*, 2003, pp. 148–153.

[13] "Arena chess gui website," 2018, available: https://www.playwitharena.com.

[14] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in *Proceedings of the 27th international conference on machine learning (ICML-10)*, 2010, pp. 807–814.

[15] R. H. Hahnloser, R. Sarpeshkar, M. A. Mahowald, R. J. Douglas, and H. S. Seung, "Digital selection and analogue amplification coexist in a cortex-inspired silicon circuit," *Nature*, vol. 405, no. 6789, p. 947, 2000.

[16] G. Campitelli and F. Gobet, "Adaptive expert decision making: Skilled chess players search more and deeper," *ICGA Journal*, p. 210, 2004.

[17] S. J. Russell and P. Norvig, *Artificial intelligence: a modern approach*. Malaysia; Pearson Education Limited,, 2016.