Teaching Chess to Neural Networks: History, Timeline and Techniques

Priyanka de Silva

SUNY Oswego

1. INTRODUCTION

Everybody has heard of the game known as Chess. As soon as the word is heard, black-and-white pieces on a grid of black-and-white squares may spring to mind. A vast number of people are also aware of the basics behind the game, due to its widespread popularity across the globe. In the present era, Chess is not only played on the classic wooden board, but also on all forms of electronic devices. However, the game itself is highly complex, and a beginner may be able to casually enjoy it but will lose almost immediately to a player who has dedicated their life to learning the game. Since both players are human and are equipped with more or less the same hardware, winning or losing ultimately comes down to how the game has been learned and how the player processes ongoing information in the heat of battle. The same principles apply to a computer learning how to play chess, and teaching one to do so, has been a challenge that has stirred the minds of some of the greatest scientists ever known.

Although most of us may have an understanding of Chess, something not many people know or think about is the recent development of Neural Network technology. With its capacity learn to identify even the most obscure patterns, and its subsequent ability to categorize information, Artificial Neural Networks (ANNs) are being used for a wide variety of different applications. These networks were modelled after biological, human neurons and how they are organized, and since then have been modified to serve purposes they initially could not. One of the recent applications of Neural Networks have been to teach one how to play different games such as Chess, Go and Shogi. While there may be other approaches to teaching machines how to play these games, using neural networks is especially interesting, as there is an element of human learning associated with the using a simulation of human neurons to play the game. This paper will provide a brief history of computer chess programs, provide a timeline of some of the chess

programs that use Neural Networks, and attempt to identify similarities and differences between the techniques used.

2. HISTORY OF COMPUTER CHESS

The history of chess playing computers can be dated back to 1890, when Leonardo Torres y Quevedo succeeded in inventing a purely electro-mechanical machine capable of analyzing an end-game strategy, where the machine had to checkmate the player's king with its own king and rook (Ensmenger, 2012). However, it wasn't until the 1950s that scientists began to construct algorithms which could play a whole game of chess based on the concept of "search". The pioneers behind these ideas were Alan Turing, who crafted the first complete chess playing program on paper in 1953, and Claude Shannon, who distinguished between type A and type B chess playing programs in 1950 (Heath & Allum, 1997).

Essentially, Shannon identified the fact that chess itself is a finite game with a clear end goal which can ultimately be mapped on to a decision-tree, with the starting configuration leading to many branches according to the moves being made (Shannon, 1950). This meant that the outcome of every game of chess could be mathematically calculated. However, this would not be practical, as a typical game of chess would require the program to analyze about 10^{134} positions, which even modern-day supercomputers would be hard pressed to perform. This required a way for the decision tree to be made smaller, and Shannon proposed two potential ways in which this may be accomplished. Since the game of chess has the Markov Property, which is to say that the future states of the process depend only on its present state, past moves do not influence the future of the game. Therefore, Shannon's Type A solution would be to

reduce the number of moves for which the program has to think ahead, effectively reducing the size of the decision tree involved (Ensmenger, 2012). However, the program would still analyze all the possible moves, up to some number of moves ahead, earning this type of searching the name “brute-force searching”. Shannon himself was more interested in his Type B solutions, which would use complex heuristics to decide which branches of the decision tree to analyze, similar to how a human player would not consider moves that are already known to be bad, but instead focus only on the moves that will assist them in achieving the final goal of checkmate (Ensmenger, 2012).

Regardless, Shannon himself was only able to device a Type A algorithm, which was labelled ‘minimax algorithm’ (Ensmenger, 2012). The minimax algorithm was capable of minimizing the maximum loss in a worst case scenario and maximizing the minimum gains in an ideal scenario. Shannon anticipated this algorithm to be of little use, however, with the subsequent development of the alpha-beta pruning algorithm by John McCarthy in 1956, Type-A programs began performing better and better, overshadowing the idea of Type-B programs (Heath & Allum, 1997). Simply put, the alpha-beta pruning algorithm further reduces the number of nodes that would need to be analyzed by the program, by completely eliminating nodes that display at least a single possibility of being worse than a move which has already been analyzed. This was improved further by the investigation of move-ordering techniques, which orders the possible moves in a way that the cutoff from alpha-beta pruning would occur faster, further reducing the amount of information that the system has to process (Heath & Allum, 1997).

Over time, technology improved according to Moore’s law which states that the number of transistors per square inch of integrated circuits has doubled each year, and will continue to, and the processing power of computers grew exponentially. After a few decades of struggle on

the part of computer scientists, in 1997, IBM's super computer "Deep Blue" was able to beat the then reigning chess champion Garry Kasparov in six games using. It should be noted that this was accomplished purely through brute-force algorithms by a machine that could analyze 200 million chess positions per second. Since then, Type-A chess programs have been at the pinnacle of their existence, leading researchers to move on to more complex games such as the ancient Chinese game of Go, or the Japanese game of Shogi. However, it also led a different group of researchers to change the way chess-programs are thought of, and to give a more "human" element to them.

3. BRIEF OVERVIEW OF SOME TECHNIQUES

3.1 Minimax Search Algorithm

The minimax search algorithm has been of utmost importance to chess programs since Shannon (1950) showed how it could be used in one. As explained above, this algorithm examines the tree structure of the game, minimizing losses, and maximizing games. The basic concept is that if the opponent is of a similar skill level, it would be best to take low-risks, which would still output a reward. However, for a game as complex as chess, searching the entire tree would be impossible, therefore the depth of the search generally has a limit. After searching for a fixed depth, the algorithm would assign values to each of the branches examined, and the best of them would be picked, and so on (Lai, 2015). This assignment of values is done via an evaluation function, and a lot of research goes into determining the best evaluation function for a chess program.

The programs discussed below attempt to use neural networks to allows the program to learn the best evaluation function using a variety of techniques. However, they all, except for AlphaZero, use the same basic principle of minimax search as a basis for doing so.

3.2 Alpha-Beta Pruning

Processing power is the biggest limitation of conducting a full-scale minimax evaluation to determine the outcome of any game. Another concept that reduces the amount of processing required is Alpha-Beta Pruning. This technique works by comparing a lower bound, alpha, and an upper bound, beta, against the minimax values of the nodes of the tree (Lai, 2015). This way, if the current score is below alpha, the system ignores that node of the tree, and conversely when checking the opponents moves, if the score is higher than the beta, the system ignores those nodes. Most of the networks discussed below, use alpha-beta search to minimize the processing requirements, with the exceptions of NeuroChess and AlphaZero.

The differences between the techniques used causes KnighCap to update the alpha-beta search leaf by leaf and Meep updates all nodes of the alpha-beta search (Silver et al., 2017). Giraffe uses the same technique as KnightCap (Lai, 2015). DeepChess, however, uses a comparison-based variant of the alpha-beta search which does not score positions to perform the search, but instead stores position values for alpha and beta, and compares the current position values against new position values. If the new position is better than alpha, the new position will replace alpha, and if the new position is better than beta, then node is pruned (David, Netanyahu & Wolf, 2016).

3.3 Temporal Difference Learning

Temporal Difference Learning was the first technique used to train neural networks to play chess. Different variants of this algorithm were used by NeuroChess, KnightCap, Meep and Giraffe. In a nutshell, temporal difference learning attempts to learn the best way to reach a set goal, when the goal is delayed by an unknown number of time steps (Thrun, 1995). Using this approach, instead of attempting to predict the outcome based on the current position, the network attempts to evaluate its own future move, leading to temporal consistency (Lai, 2015). Details of the specific variants used will be provided with each different program.

4. TIMELINE OF CHESS-PLAYING NEURAL NETWORKS

This section will discuss six different programs implemented to play chess using neural network technology. The programs are ordered chronologically. For each program, an overview will be provided, consisting of information about the program, techniques the program uses, how neural networks are involved and how well the program performed. The next section will attempt to investigate the common learning algorithms and training techniques used by the programs and how the they differ from program to program.

4.1. NeuroChess (1995)

NeuroChess is possibly the first program which uses Temporal Difference (TD) learning to calculate an evaluation function for playing chess (Thrun, 1995). The program uses TD(0), a variant of TD learning, which uses the final outcome of a large number of chess boards to categorize them as being a win, a loss or a draw. The network then uses this information to form an evaluation function V, which functions as a target for playing future games. The function also uses a constant labelled a *discount factor*, which decays V over time, to favor early game wins,

over late game wins. For the purpose of the program, TD represents entire chess games into a sequence of positions, which is then used as a training pattern to allow the network to learn what a win and a loss is, and the best next move to make to arrive at the goal of winning (Thrun, 1995).

Due to the large amount of time it would take to train the network using conventional means, the network employs Explanation-Based Neural Network (EBNN) learning, which uses available domain specific data to generalize training examples (Thrun, 1995). For this purpose, the program uses another neural network called the *chess model M*, which maps random chess boards to corresponding expected chess boards, 2 moves ahead. This network is trained before V and is done so using a large number of grandmaster games. EBNN uses the information learned by M to bias the evaluation function V, so that V is adjusted towards the next best move. This way, the network learns the importance of 'looking ahead' in a chess game, rather than evaluating the current static position.

The network architecture consists of 175 input, 165 hidden and 175 output units for network M, which after training using 120000 expert games learned an evaluation network V with 175 input, 0-80 hidden and 1 output units (Thrun, 1995). The program was able to defeat GNU-Chess, 13% of the games played. These disappointing results were attributed to a limited training time, and a loss of information with each step of learning which results in inadequate openings. The program was also noted to take two orders of magnitude longer than GNU-Chess did with its linear evaluation function.

4.2. KnightCap (2000)

Similar to NeuroChess, KnightCap also uses a variant of TD learning, which they dub *TDLeaf* (Baxter, Trigdell & Weaver, 2000). This approach uses a tree built through a comprehensive minimax search, and applies the TD function to its leaf nodes, instead of simply applying it to board positions.

KnightCap uses a variant of the TD algorithm labelled TD-Leaf. This algorithm essentially applies the same principle as NeuroChess but instead of applying it to position vectors, it applies it to the principle variation of the alpha-beta search tree (Baxter, Trigdell & Weaver, 2000). The principle variation is the path that the tree must take in order to arrive at the desired goal of the depth-space. By applying TD-Leaf to the leaves of the tree, the information gained will tell the network the next best move for following that ideal path.

For this program, the initial parameters are all set to zero, except for the material values of the pieces (i.e. pawn is 1, rook is 6) (Baxter, Trigdell & Weaver, 2000). The network was then trained by having it play online against human players, where the skill level of the program was of an average human player. By applying the TDLeaf learning algorithm, the skill level of the program increased to that of a human master in just 300 games.

The evaluation function for KnightCap uses 5872 features, with equal amounts for opening, middle, end and checkmate stages (Baxter, Trigdell & Weaver, 2000). Unlike NeuroChess, KnightCap also uses an opening-book learning algorithm, which learns the best ways to open a game through experience. However, the problems with search still remain, as the program does not work well against other programs which can search deeper and faster.

The authors attribute KnightCap's high rate of learning to four factors (Baxter, Trigdell & Weaver, 2000). The initial weights being set to zero caused large changes in ordering positions

even through small changes to the weights. Its ability to learn parameters put it on par with many far superior parameter settings which were predefined. The player base it was trained against usually play with players of similar skill, allowing the program to improve along with its opponents. and finally, learning online, as opposed to self-play, exposed the program to a variety of unconventional moves.

4.3. Meep (2009)

Meep also uses a variant of the TD-Leaf algorithm to train the evaluation function of the network, which the authors dub “TreeStrap” (David, Netanyahu & Wolf, 2016). They first test the network by using an algorithm labelled RootStrap to evaluate a minimax search. This algorithm follows KnightCap’s and NeuroChess’s method of updating the values of the minimax search one subsequent time step at a time. However, they claim that networks trained in such a manner find it difficult to deviate from the training patterns and are vulnerable to blunders or unexpected moves made by the opponents. They also claim that such networks also lack the ability to learn from self-play effectively, as was demonstrated in the KnightCap program (Baxter, Trigdell & Weaver, 2000). Another problem they claim is that the values of the pieces had to be initialized by humans for the system to achieve any level of play.

To counter these problems, Meep uses the TreeStrap algorithm, which uses the network to update all the nodes of the alpha-beta search, instead of just the root node (David, Netanyahu & Wolf, 2016). The network uses a weighted linear combination of 1812 features to create an evaluation function, with random initialized weights assigned to each feature value. Instead of using the values of a subsequent search, the network uses the outcomes of a deep search to learn the evaluation function. It is important to note that no prior knowledge was hand-coded into the network except for a small opening book that was used to maintain diversity during training.

In Meep, TreeStrap is applied to both minimax search as well as alpha-beta pruning (David, Netanyahu & Wolf, 2016). The minimax version of this variant, instead of updating just the principle variation of the tree, updates all the nodes in the tree as it searches for the minimax value for the next move. This is made better using alpha-beta search instead, as the cutoff values can be exploited by this method. If the value from the heuristic evaluation is higher than alpha, then it is reduced towards alpha, and if the value is smaller than beta, then it is increased towards beta. For this technique to be implemented, a procedure called *DeltaFromTransTbl* is also used, which recursively steps through a transposition table to collect any relevant information about the search depths and boundaries.

The program was trained exclusively using self-play and was able to achieve a human-master level of play after training for 1000 games (David, Netanyahu & Wolf, 2016).

4.4. Giraffe (2015)

Giraffe is a more recent program which uses TD-leaf and a probability-based search instead of a typical depth-based search. The program has achieved the level of a FIDE international master or the top 2.2% of all chess players (Lai, 2015).

The architecture of the network consists of two hidden layers and an output layer (Lai, 2015). The hidden layers use Rectified Linear Activation instead of the typical hyperbolic tangent and logistic activation functions. The output layer however, uses a hyperbolic tangent activation function. The input layer consists of three separate feature representations, which are piece-centric, square-centric and position-centric, to represent low-level features as well as high-level features.

The TD-Leaf variant used by the developers, randomly selects 256 positions from the training set, and uses self-play to generate 12 moves (Lai, 2015). These moves are then used to calculate the error, by adding up all the changes weighted by how far away it is from the start. Once all the errors have been obtained, the network is trained by using backpropagation to calculate the gradient of the loss functions and using the gradients to train the network using stochastic gradient descent.

This engine also uses a probability-based search instead of a depth-based search (Lai, 2015). In a general minimax search, the decision tree is examined up to a certain depth, and the best branches are chosen. In this engine, a separate neural network is used to identify all nodes that have a greater than 0.000001 chance of being higher than a probability threshold. The researchers claim that this is comparable to or better than the typical depth-limited search.

4.5. DeepChess (2016)

The most significant difference of DeepChess compared to other programs is its use of both supervised and unsupervised learning. The unsupervised portion of the program is labelled pos2vec, and it acts as a feature extractor which transforms given chess positions into a vector of values that contain high-level information (David, Netanyahu & Wolf, 2016). For this purpose, they establish a Deep Belief Network (DBN), which consists of five layers of sizes 773, 600, 400, 200, 100 respectively. Then they train the DBN layer by layer to extract relevant features from a random subset of 2,000,000 chess positions. The DBN also uses a rectified linear unit activation function similar to Giraffe.

The second phase of the program is what is dubbed DeepChess, and it uses two fully trained pos2vec networks side by side (David, Netanyahu & Wolf, 2016). These two networks

then connect to three fully connected layers, which lead to a two-value softmax output layer. The network is trained to determine which of the two outputs is most likely to result in a win. The first four layers act as a feature extractor and the last four layers compare the features of the positions to determine which is better. The training is conducted for 1000 epochs, where for each epoch, 1,000,000 input pairs are generated. One input is randomly selected from a set of winning positions, and the other from a set of losing positions. The pair is ordered randomly as win-loss, or loss-win. The network learns to evaluate which is a winning position and which is a losing position, and the testing phase gained exceptional results of 98% success.

The network uses a novel variant of the alpha-beta search to prune the search tree for this program (David, Netanyahu & Wolf, 2016). Typically, the positions are given scores which are then evaluated to see if a branch should be considered or pruned. This program assigns positions to alpha and beta instead of position scores and compares the current position to the previous alpha and beta using the DeepChess network. If the position is better than alpha, it replaces alpha, and if the position is better than beta, then that node is pruned.

4.6. AlphaZero (2017)

AlphaZero is an algorithm derived from AlphaGo Zero, which recently became the first AI system to reach superhuman level of play at the game of Go, by learning the game by itself, with no information provided by its developers (Silver et al., 2017). In AlphaZero, the developers have generalized the algorithm to be applicable to other domains such as Chess and Shogi. In both games, the algorithm was able to easily defeat a human world-champion within 24 hours, with only encoded knowledge of the games' rules.

AlphaZero uses a neural network which takes a board position as an input and outputs a vector of move probabilities that stems from the given position, and a scalar value which estimates the expected outcome (Silver et al., 2017). These values are learned through self-play.

Instead of a traditional alpha-beta search, AlphaZero uses a Monte-Carlo tree search (MCTS) algorithm (Silver et al., 2017). For each search, the network runs a series of game simulations where it traverses a search tree, which are then used to output a vector representing a probability distribution of the best possible moves.

The parameter of the neural network is trained through self-play reinforcement, using a MCTS algorithm for both players (Silver et al., 2017). At the end of each game the terminal position is scored as a win, loss or draw. The neural network updates itself so as to reduce the error between the outcome of the game, and the expected outcome.

6. CONCLUSION

In conclusion, it is evident that neural network technology has been improving at an exponential rate, as indicated by the gaps between each program. The minimax and alpha-beta search algorithms have played a crucial role in this development. Temporal Difference Learning brought the networks to a very high standard. However, these techniques can only get the technology so far because as AlphaZero has indicated, thinking outside the box and using a MCTS pushed such networks from a human level to a superhuman level, without the use of any hand-coded evaluation functions or prior knowledge that earlier programs possessed. If neural networks can be made to learn how to make complex decisions such as in a game of chess, their applicability in other forms of decision making processes could be further improved.

References

Baxter, J., Tridgell, A., & Weaver, L. (2000). Learning to play chess using temporal differences. *Machine Learning*, *40*(3), 243-263.

David, O. E., Netanyahu, N. S., & Wolf, L. (2016, September). Deepchess: End-to-end deep neural network for automatic learning in chess. In *International Conference on Artificial Neural Networks* (pp. 88-96). Springer, Cham.

Ensmenger, N. (2012). Is chess the drosophila of artificial intelligence? A social history of an algorithm. *Social studies of science*, *42*(1), 5-30.

Heath, D., Allum, D., & Square, P. (1997, January). The Historical Development of Computer Chess and its Impact on Artificial Intelligence. In *Deep Blue Versus Kasparov: The Significance for Artificial Intelligence* (p. 63).

Lai, M. (2015). Giraffe: Using deep reinforcement learning to play chess. *arXiv preprint arXiv:1509.01549*.

Shannon, C. E. (1950). XXII. Programming a computer for playing chess. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, *41*(314), 256-275.

Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., ... & Lillicrap, T. (2017). Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm. *arXiv preprint arXiv:1712.01815*.

Thrun, S. (1995). Learning to play the game of chess. In *Advances in neural information processing systems* (pp. 1069-1076).

Veness, J., Silver, D., Blair, A., & Uther, W. (2009). Bootstrapping from game tree search. In *Advances in neural information processing systems* (pp. 1937-1945).