Final Report: Team 44

Travis Burrows Jeyanth John Britto Mihir Tulpule Pranshav Thakkar
Alen Polakof Gabriel Nakajima An

December 5, 2019

1 Introduction and Problem Definition

Our objective was to create an interactive tool for chess players that can evaluate a chessboard configuration and inform the user of relative chess piece values (and how they will change). This would help players develop their own intuition for chess strategy.

Our tool learns chess via neural networks, trained on a dataset of recorded chess games and self-play. Rather than output the optimal move for winning, our tool provides current/hypothetical piece values for the next move. A visualization has been provided for the user to configure their own chessboard and run analytics through the trained model. Value changes and a value advantage comparison would be reported graphically.

2 Motivation

Many chess user interfaces (CUIs) focus on move optimality or do not deliver useful information to the player in an accessible manner. We wanted to provide players with dynamic information on a move's advantages, rather than just the optimal move itself. We decided that a useful vehicle for this information would be informing the player of each piece's relative value based on a board configuration, and graphically demonstrating the value/advantage shift would make for an informative and interactive tool for players of all skill levels.

3 Literature Review and Survey

Silver et al. [14, 16] combine deep learning and reinforcement learning methods into an algorithm that can play chess with super-human level performance. They use similar techniques by generalizing the work of Silver et al. [15]. Their algorithms learn from self-play (only with domain knowledge of the rules) and don't use human games, which we plan to incorporate. They also present a technique to encode the board configuration for input to their neural networks, which we will utilize. For each piece, they have a "plane" vector that specifies where

the corresponding piece is located on the board. Limitations of their work are that they have a black-box model and that they cannot extract other information, like piece values - only the expectation of winning at each state as well as a probability vector of best moves.

To extract more information from their black-box models, different methods have been developed such as Verma et al. [20], which learns *policy networks* (refer to Appendix A.4) that are programmatically structured, making the process more interpretable. We pretend to structure our policies such that the move we make will increase the total value of our pieces. By using a programmatic policy, it will encode the value of a piece in a certain board configuration. Research in "policy networks" has also been done by Silver et al. [13], who built a deep-learning-based engine to play the game Go using "policy networks" and "value networks".

DeepChess [6] reached grandmaster level with a deep learning approach by learning how to compare two board configurations and selecting the better one, thus learning an *evaluation function* (refer to Appendix A.2). We modify this evaluation function so it will tell the value of each piece on the board. In addition, we will be using the work of Kaneko and Hoki [8] to analyze the performance of our evaluation function.

Beal and Smith [3, 4], Veness et al. [19], Block et al. [5], Thrun [18], Baxter et al. [2] have applied a reinforcement learning technique named temporal-difference (TD) learning to learn policies for chess, which we will adapt and use. Beal and Smith [3, 4] used TD to estimate the value of each chess piece [3] and shogi [4]. However, they don't estimate how such values change as the game progresses and the board configuration changes.

These TD approaches are based on *search trees* (refer to Appendix A.1) and have been developed further by Lai [9], which introduces a new method to pruning (refer to Appendix A.5) the search tree, and Marsland [11], which compares different search methods on trees that can be utilized by chess algorithms. These works, together with the work on tree search pruning by Hoki and Kaneko [7], will be useful, as we will use search tree and pruning techniques to alleviate computational demand.

ExIt [1] takes an imitation learning approach to solve the board game of Hex, in which an expert policy trains an apprentice policy. We will use a similar approach to how they obtained their expert policy via a tree search algorithm that considers a tree of possible moves.

Maddison et al. [10] utilize CNN to train a model to play Go. They encode the level of the player for each game in the training data, an approach we will explore. Sabatelli et al. [12] also utilize CNN but focuses on chess. They develop an innovative approach with 7 CNN, where one is responsible for deciding where a piece should escape when being attacked, and the others calculate the optimal options for that piece to develop. This predisposes the agent to escape and prioritizes pieces over board configuration - something we want to avoid.

4 Proposed Method

4.1 Intuition - Improvement on State-of-the-Art CUIs

With the recent prominence of machine learning, computer chess engines have attained unprecedented ELO ratings. For instance, AlphaZero (the current best model) has been able to attain a rating proxy to 3750, surpassing the famous Stockfish engine by approximately 300 points. However, although they have high ratings, none of these models provide an informative visualization for human players who would like to analyze their game in a sophisticated manner. This is where our innovation appears. Our model is the first to extract each piece's value given a board configuration. This is done by training an auto-encoder and neural networks, utilizing millions of board configurations and values from the Stockfish engine as input. In addition, our visualization platform provides the ideal interface for any player trying to improve their strategy and gain better insights as to how moves affect play development. Our tool will provide users with a visual understanding as to when (for example) large sacrifices are strategically advantageous, which no tool is currently able to achieve. This project is the first communication channel between prior great models developed to play chess using machine learning and clearly displays the move-based reasoning underlying these models.

4.2 Description of Approach - Algorithm

We leveraged millions of recorded games to train our model, with the objective of learning piece importance given a specific board configuration. To train the piece value networks, we ran each board through the Stockfish chess engine to analyze the "centipawn" score for that specific configuration by playing against itself. (The centipawn score is the chess measurement unit for the relative advantage of a player, with one centipawn equivalent to 1/100 of a pawn). The board configuration was then input into an auto-encoder, and the centipawn scores were generated as the targets to a combination of piece-value networks (explained in the following subsections). Our final model was a deep neural network as shown in Figure 1.

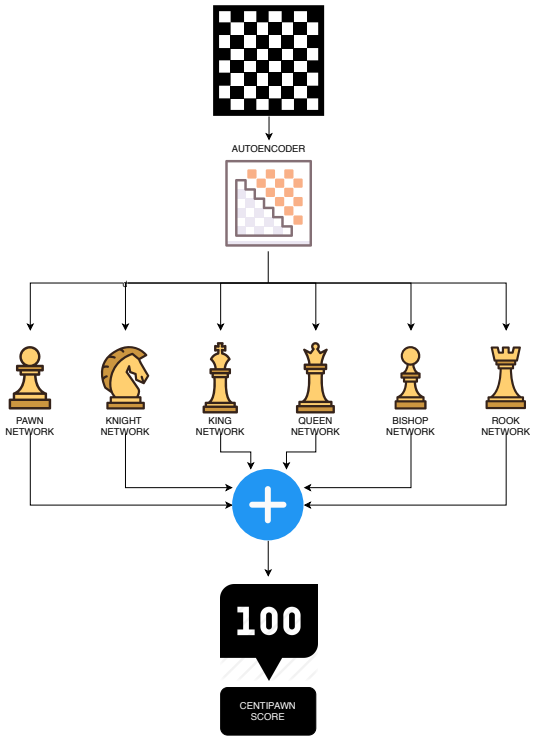

Figure 1: Model Overview.

4.2.1 Retrieving the Data

We trained our model utilizing a dataset of over 6 million board configurations from approximately 1.1 million games, downloaded from computerchess.org [17]. However, this dataset was only fully realized after multiple iterations of testing and expanding our dataset to increase the accuracy of our piece value predictions.

The final dataset was a 1.2 gigabytes text file, with each text block including encoded meta-data on the players, the game winner, and the piece positions/moves. To retrieve the board configurations needed for our auto-encoder, we utilized a Python chess library to parse through the recorded text file and retrieve configurations from each of the games. These configurations were stored in arrays that we could use to feed board data into our auto-encoder model.

4.2.2 Auto-Encoder Model

Once we retrieved the parsed board configurations, we used an auto-encoder model to learn a latent representation of our board and a square coordinate corresponding to the location of the piece to be valued. The input to this network was a "plane" one-hot representation, as described in [6] and [14]. For each of the 64 (8x8) squares in the board, we had a one-hot encoding of the type and color of the piece. Since there are 6 types of pieces in each chess game (pawn, knight, bishop, rook, queen, and king) and there are two colors (one for each player - white and black), each of our one-hot vectors was of size 12 (6x2). This resulted in a 768-dimensional (64x6x2) representation of the piece locations on the board. We appended 5 more dimensions to encode meta information about the state of the game. These were:

1. Whose turn it is (0 if white's turn, 1 if black's turn)
2. Possibility for king's side castling for white player
3. Possibility for king's side castling for black player
4. Possibility for queen's side castling for white player
5. Possibility for queen's side castling for black player

To encode each piece position, we appended a 64-dimensional one-hot vector to our input representation. Our auto-encoder model learned a latent space representation of our 837 (768 + 5 + 64) dimensional input vector. The auto-encoder was trained on a binary cross entropy loss to reconstruct the same input in the output layer. The encoder and decoder were fully connected, with the encoder having a layer of sizes 837-600-400-200-100, and the decoder with the reverse. Here, the latent space dimensionality was 100.

4.2.3 Piece Valuator (Neural) Networks

The latent representation of our auto-encoder was fed into our Piece Valuator Network (PVN). This resulted in a fully connected network with 3 hidden layers, which would output a scalar value for a specific piece.

We instantiated 6 PVNs, one for each piece type. The output of our complete model was the sum of the values of the pieces on the board, valued according to the outputs of each PVN. This sum represented our approximation of the centipawn score for that particular board configuration. (The centipawn score is always calculated relative to the white player, so in the case that the black player is winning, the score will appear negative.) Subsequently, we regressed this score to

the real centipawn score calculated by the Stockfish engine for the loss function, which utilized Mean Square Error (MSE) as its guiding metric. $MSE = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2$, where Y_i are the targets (centipawn scores calculated by stockfish), and $\hat{Y}_i$ are the centipawn scores that were network outputted.

Here is a demonstrative example with the following board configuration below:

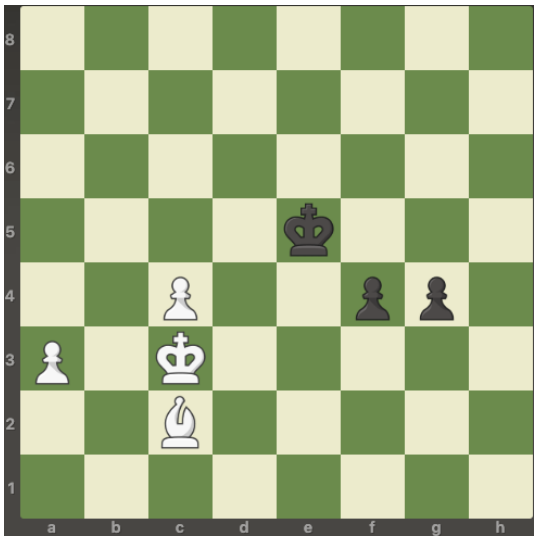

Figure 2: Board configuration example.

Let the subscript of PVN denote the piece type that the network is valuing. That is PVN_p is the PVN for pawn, PVN_k the PVN for king, etc.

Let AE be the autoencoder network, which takes in a board position and a piece position. Let's evaluate the value of the bishop at position "c2". Leting B denote the total board state, then the value of the bishop is $\text{PVN}_b(\text{AE}(B, c2))$. The total value of white relative to black is:

$$\Big[\text{PVN}_p(\text{AE}(B, a3)) + \text{PVN}_p(\text{AE}(B, c4)) + \text{PVN}_b(\text{AE}(B, c2)) + \text{PVN}_k(\text{AE}(B, c3))\Big] - \Big[\text{PVN}_p(\text{AE}(B, f4)) + \text{PVN}_p(\text{AE}(B, g4)) + \text{PVN}_k(\text{AE}(B, e5))\Big]$$

By generating an aggregate centipawn score, we could determine a holistic metric of player advantage based on board configuration. Although we had initially implemented a binary metric of which player was winning/losing based on the board setup, we felt that it did not adequately reflect the magnitude or duration of a player's advantage as well as the centipawn score.

4.3 User Interface: Interactive Chessboard Visualization

For developing the UI, we used a Flask server to run a Javascript-based webpage that contained an interactive chessboard. The chessboard displayed each piece's value and allowed users to move pieces on/off the board while viewing real-time changes in individual piece valuation.

The model relayed a data structure of each piece with its value and position upon startup, with Flask providing the translation from Python to Javascript, and the visualization queried the model upon any configuration change. The model then sent new values based on the piece moved and updated the visualization accordingly.

The interactive visualization was created using chessboard.js (which allowed us to create a customizable board with interactive pieces) and altered using Javascript to change square color dynamically based on the input from the model. Pieces were also encoded with the constraint of legal moves to prevent impossible combinations. Board clearing and reset buttons were added to increase utility for the player. There are two modes for the tool: standard game format (in which the player starts from a regular beginning setup and can play regularly) and free-form (in which the player can place pieces onto a blank board in any configuration).

5 Team Contributions

All teammates contributed equally to this project and performed their assigned tasks.

6 Experiments & Evaluation

6.1 Description of TestBed

To evaluate our model's effectiveness, we wanted to answer the following main questions:

1. Does our metric of measuring player advantage make sense?
2. Can we accurately predict piece values based on board configuration, even with freeform piece placement?
3. Does our board change dynamically with configuration/piece value changes?
4. Do all aspects of our UI (piece placement, color changes, board resets) work smoothly?
5. Is our representation of point value changes interpretable?
6. Is the integration between our model and UI successful?

6.2 Model Experiments and Evaluation

We trained our PVNs in two different ways. In the first experiment, we trained our model so that it would output the probability of the white player winning. In the second experiment, the output we regressed to was the "centipawn" score explained previously. In either case, the input was the same: a board configuration.

6.2.1 Predicting Probability of Winning

For our first experiment, our model (which predicted whether white was going to win or lose) achieved 75% accuracy. Figures 3 and 4 show the progression of accuracy and loss during supervised learning training for our initial dataset. This model employed a sigmoid function, which would output a probability of winning between 0-1 and then utilized as a loss function a binary cross-entropy loss. Although the model had a good testing accuracy, it did not perform well in the opening game (likely because binary cross-entropy loss utilizes hard labels (either 0 or 1)), making the model unable to capture ties or small advantages. Under this framework, winning by large or small margins would be the same.

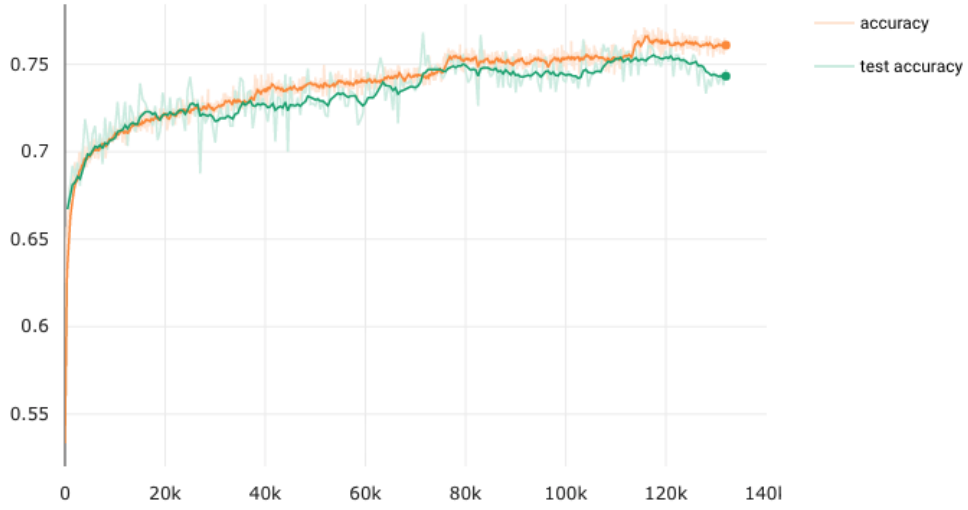

Figure 3: First model accuracy training/testing.(Y-axis: Accuracy. X-axis: Training iterations)

Therefore, we realized that a better metric would be to use the "centipawn" score. This new model was better able to understand such shortcomings and performed much better.

6.2.2 Predicting Centipawn Score

After training our model to predict the centipawn score given a board configuration, we calculated the average value of each piece based upon the scores outputted by each PVN. Since we instantiated our model by inputting game outcomes and calculated player advantage as the sum of our piece values, the network was able to self-learn how to correctly value each piece.

In Table 1, we juxtaposed the *average* values outputted by our neural networks with piece values based on conventional chess logic (excluding the king as a special case). While our model numbers matched closely, the results demonstrated that our neural network was capable of learning accurate piece valuation (comparable to decades' worth of recorded games) based on reading board configurations and even surpassed our expectations. For example, it has been hotly debated in the chess community that the "bishop" is slightly stronger than the "knight". Impressively, our model was able to capture this human observation, as it valued the bishop slightly higher.

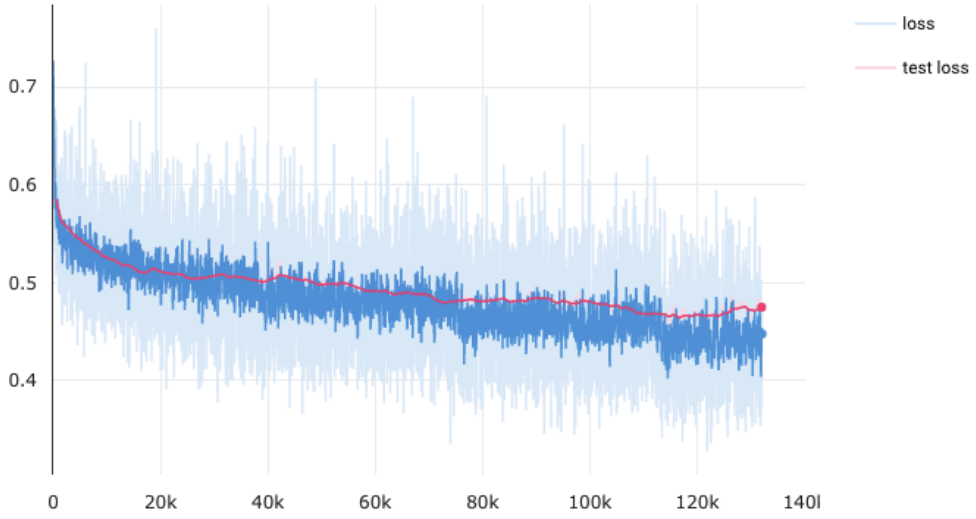

Figure 4: First Model:Loss of training and test sets. (Y-axis: Loss. X-axis: Training iterations)

Piece	**Traditional**	**Our Model**
Pawn	1.00	1.00
Knight	3.00	2.94
Bishop	3.00	3.13
Rook	5.00	4.96
Queen	9.00	9.87

Table 1: Comparison of pieces value between our model and chess books. Network values are normalized so the pawn is unit-valued.

This approach worked better than simply predicting the probability that white was going to win, as centipawn scores contain much more information than a win/loss indicator.

6.2.3 Playing against Chess Engines

To evaluate the model quality, we had our model play against the Stockfish chess engine. Although our model outputted a centipawn score, we could still use our model to select moves and play games. For every turn, we considered all possible moves. For each move, we ran our model on the board configuration under the assumption that move was made, and we picked the move that yielded the highest centipawn advantage.

We instantiated Stockfish with varying strength levels with ratings varying from 1200 to 3400. We did not obtain good results, however, and we hypothesized that this was because our dataset was composed of professional games. As a result, our model did not initially learn to how evaluate board positions that were sub-optimal.

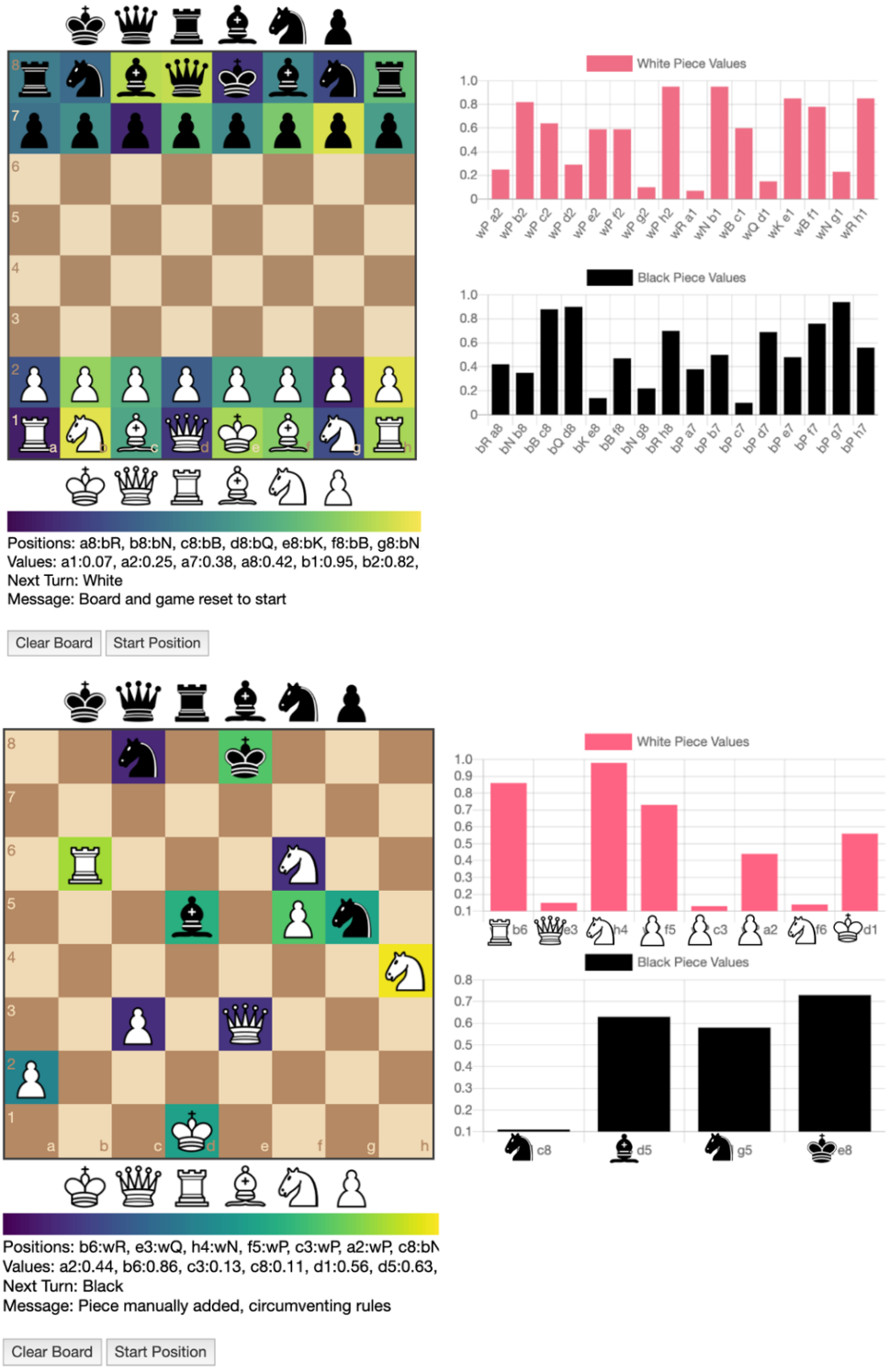

Figure 5: Interactive visualization at start (top) and in play (bottom)

6.3 Interactive Visualization

After implementing a chessboard with interactive pieces and dynamically-colored board squares dependant on the piece's value, we had to shift the color gradient and square coloration numerous

times to make the tool as visually appealing and interpretable as possible. While we initially created just the base chessboard for our visualization, this was insufficient for conveying useful piece value information. As a result, we implemented two bar charts (one for each player) outside the board that would change in tandem with board configuration changes and piece moves, displaying point value shifts in a holistic and interpretable manner.

Two examples of the visualization can be seen in Figure 5, where the left image is an example starting position, with piece values visualized underneath each piece and bars for each piece for each player. In gameplay (on the right), there are fewer bars because there are fewer pieces on the board, and each piece value changes with its relative position to both of the players' pieces.

7 Discussion and Conclusions

We were successful in creating a deep neural network that would accurately predict piece value shifts due to board configuration, developing an interactive visualization that would dynamically display piece valuation in an interpretable manner, and integrating the two into a functional and accessible tool.

While it would have been nice to expand our dataset even further and improve the accuracy of our piece valuation, our tool is a good starting point for further experimentation in the field of chess strategy. It is our hope that in the future, chess players of all skill levels will be able to use tools like ours to improve their understanding of complex games like chess.

References

[1] Thomas Anthony, Zheng Tian, and David Barber. Thinking fast and slow with deep learning and tree search. *CoRR*, abs/1705.08439, 2017. URL http://arxiv.org/abs/1705.08439.

[2] Jonathan Baxter, Andrew Tridgell, and Lex Weaver. Learning to play chess using temporal differences. *Machine Learning*, 40(3):243–263, Sep 2000. ISSN 1573-0565. doi: 10.1023/A:1007634325138. URL https://doi.org/10.1023/A:1007634325138.

[3] D. F. Beal and M. C. Smith. Temporal difference learning for heuristic search and game playing. *Inf. Sci.*, 122(1):3–21, January 2000. ISSN 0020-0255. doi: 10.1016/S0020-0255(99)00093-6. URL https://doi.org/10.1016/S0020-0255(99)00093-6.

[4] Donald F. Beal and Martin C. Smith. Temporal difference learning applied to game playing and the results of application to shogi. *Theoretical Computer Science*, 252(1-2):105–119, February 2001. doi: 10.1016/s0304-3975(00)00078-5. URL https://doi.org/10.1016/s0304-3975(00)00078-5.

[5] Marco Block, Maro Bader, Ernesto Tapia, Marte Ramírez, Ketill Gunnarsson, Erik Cuevas, Daniel Zaldivar, and Raúl Rojas. Using reinforcement learning in chess engines. *Research in Computing Science*, pages 31–40, 2008.

[6] Eli David, Nathan S. Netanyahu, and Lior Wolf. Deepchess: End-to-end deep neural network for automatic learning in chess. *CoRR*, abs/1711.09667, 2017. URL http://arxiv.org/abs/1711.09667.

[7] Kunihito Hoki and Tomoyuki Kaneko. Large-scale optimization for evaluation functions with minimax search. *J. Artif. Int. Res.*, 49(1):527–568, January 2014. ISSN 1076-9757. URL http://dl.acm.org/citation.cfm?id=2655713.2655728.

[8] Tomoyuki Kaneko and Kunihito Hoki. Analysis of evaluation-function learning by comparison of sibling nodes. pages 158–169, 01 2012. doi: 10.1007/978-3-642-31866-5_14.

[9] Matthew Lai. Giraffe: Using deep reinforcement learning to play chess. *CoRR*, abs/1509.01549, 2015. URL http://arxiv.org/abs/1509.01549.

[10] Chris J. Maddison, Aja Huang, Ilya Sutskever, and David Silver. Move evaluation in go using deep convolutional neural networks. 2015. URL https://arxiv.org/pdf/1412.6564.pdf.

[11] T. Anthony Marsland. Computer chess methods. 10 2001.

[12] Matthia Sabatelli, Francesco Bidoia, Valeriu Codreanu, and Marco Wiering. Learning to evaluate chess positions with deep neural networks and limited lookahead. 01 2018. doi: 10.5220/0006535502760283.

[13] David Silver, Aja Huang, Christopher J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–503, 2016. URL http://www.nature.com/nature/journal/v529/n7587/full/nature16961.html.

[14] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy P. Lillicrap, Karen Simonyan, and Demis Hassabis. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *CoRR*, abs/1712.01815, 2017. URL http://arxiv.org/abs/1712.01815.

[15] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of go without human knowledge. *Nature*, 550:354–, October 2017. URL http://dx.doi.org/10.1038/nature24270.

[16] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144, 2018. ISSN 0036-8075. doi: 10.1126/science.aar6404. URL https://science.sciencemag.org/content/362/6419/1140.

[17] CCRL team. CCRL 40/40 - games, 2013. URL http://ccrl.chessdom.com/ccrl/4040/games.html.

[18] Sebastian Thrun. Learning to play the game of chess. In *Proceedings of the 7th International Conference on Neural Information Processing Systems*, NIPS'94, pages 1069–1076, Cambridge, MA, USA, 1994. MIT Press. URL http://dl.acm.org/citation.cfm?id=2998687.2998820.

[19] Joel Veness, David Silver, Alan Blair, and William Uther. Bootstrapping from game tree search. In Y. Bengio, D. Schuurmans, J. D. Lafferty, C. K. I. Williams, and A. Culotta, editors, *Advances in Neural Information Processing Systems 22*, pages 1937–1945. Curran Associates, Inc., 2009. URL http://papers.nips.cc/paper/3722-bootstrapping-from-game-tree-search.pdf.

[20] Abhinav Verma, Vijayaraghavan Murali, Rishabh Singh, Pushmeet Kohli, and Swarat Chaudhuri. Programmatically interpretable reinforcement learning. *CoRR*, abs/1804.02477, 2018. URL http://arxiv.org/abs/1804.02477.

A Technical methods for chess engines

A.1 Search tree

When considering possible moves from a specific board position, we analyze what is named a *search tree*. This is a graph tree in which each node represent a board position, and an edge connecting two nodes represents a move performed that took the board position corresponding to one node to another. When searching for best strategies and moves, chess engines construct this search tree. Figure 6 displays a search tree for the simple game of tic tac toe.

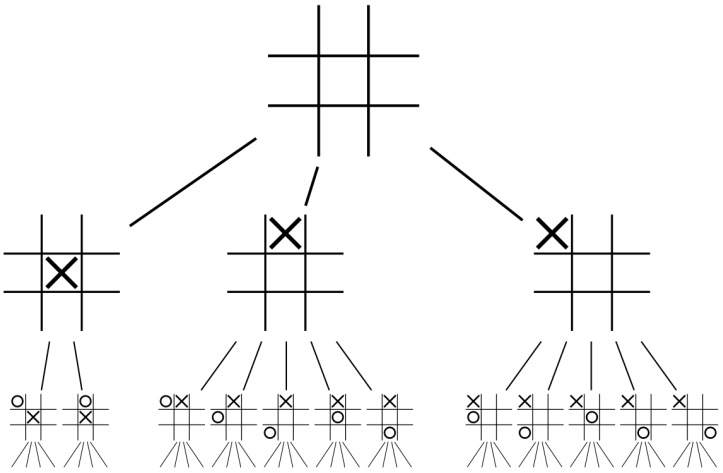

Figure 6: Search tree of tic tac toe game

A.2 Evaluation function

Chess engines usually use what is referred to as an *evaluation function*. This is a function that maps a specific board position to a real number, which represents how "good" of a position it is for a specific player. For ordinary chess engines, this evaluation function is handcrafted based on human domain knowledge. Recent deep learning methods aim to learn this evaluation function.

A.3 Minimax Function

Chess agents try to minimize the possible loss for a worst case (maximum loss) scenario. This is perfectly suitable for this game since chess can be formulated as a two-player zero-sum game. When viewed as a function in R^n, this refers to finding the saddle point in a function, and this idea is important for pruning trees since when looking for different scenarios, the ones that are worst to a scenario we have already seen, need not be considered.

A.4 Policy

A policy is another term for a player in the context of game playing. Formally, a policy is a function that maps a board position to a specific move, or a distribution of moves.

A policy network is simply a policy whose function is represented by a neural network.

A.5 Pruning

Pruning is a technique in search algorithms that reduces the size of decision trees by removing sections of the tree that provide little or no power at all to instances that need to be classified. In chess, more specifically, pruning reduces the complexity of the final classifier, increases the speed to make a decision and because of how the game is structured, it provides a framework for effective removal of possible branches without loss in accuracy.

B Overview of Chess Strategies

Each chess piece has a *material value*, which is independent of its position on the board. Each piece has its own range of *activities* dependent on its type and position on the board. The number of squares a particular piece "controls" is the possible squares at which an opponent's pieces can be captured by that piece. This is akin to the *potential* of the piece, and it is a generally good strategy to maximize the potential of one's pieces when formulating a chess strategy. Although piece values/potential can change relative to the positioning of other pieces on the board, a piece's value can be roughly estimated solely based on a particular board configuration at a set point in time. By maximizing each piece's utility across board configurations, a player can devise optimal strategies for winning chess matches.