Chess Programming and Engine Evaluation

Anonymous

The University of Texas at Austin

December 2024

1 Introduction

Machine learning and data is becoming an increasingly important aspect of our lives. Industries capable of garnering, manipulating and employing large amounts of data see a direct correlation towards their efficacy as well as efficiency. Additionally, games such as Chess and Go utilize machine learning to employ powerful engines well beyond human capability. Chess, in particular, can be seen as having gone through a revolutionary period with this newfound computing. For one, having access to games via the internet has tremendously expedited studious efforts. Likewise, the utilization of engines has modernized all aspects of chess. In fact, there are openings in chess called "modern chess openings" which get their names from the fact that computational power has produced new openings, while rebuking old ones.

Arguably the most studied and feared chess opening in history, the King's Gambit, has since been concretely exposed as a weak opening. Interestingly enough, Bobby Fischer famously "cracked" the King's Gambit in his 1961 article "A Bust to the King's Gambit" showcasing a refutation of the opening (Chess.com, 2009, "A bust to the King's Gambit… Bobby Fischer"). 63 years later and this defense is considered "optimal" play against the King's Gambit.

As an aside, throughout the text I will be using the term "optimal" to represent an engine's attempt at picking the best move. Chess is not a solved game, and hence, even the most advanced engines are unable to truly make optimal moves. While the term "optimal" isn't quite accurate, I think it is the most appropriate term in this case as it presents the idea in the most succinct manor.

2 Motivation

The ultimate reasoning behind developing a chess engine stems from the inadequacies prevalent in top engines today, albeit many would not consider our personal gripes as inadequacies. For one, top chess engines exhibit a level a play well beyond that of any human. In fact, even grandmasters consider certain moves played by Stockfish, the strongest engine, as "not human" implying that no human would ever even consider the optimal move played (Chess.com, n.d., "Chess Engine"). As a specific example, engines will often slide rooks over one square in preparation for a stronger position many moves down the line. The level of depth engines are capable of calculating to allows them to make these predictive moves well beyond that of a human. While this doesn't immediately seem noteworthy, my issue lies in the fact that using engines to review games becomes worthless when engines are making unhuman-like decisions. The objective, then, is to develop an engine that is more inclined towards teaching, rather than achieving optimal play. While this is the main motivation overall, the focus initially has been on developing a strong engine (although not too strong such that it plays beyond current human capability). All that being said, there is no engine to be had if there is no working chess program. Hence, the following is a run-through on the various chess program prototypes with a focus on future AI learning.

3 Methods

3.1 Chess Programming

3.1.1 Iteration 1

I would more so consider this a precursor as the original idea never fully came to fruition. Reason being is the numerous flaws in the idea. That said, many of the original ideas are still prevalent in the most up to date engine, and the flaws paved the way for improvements. The major flaws with iteration 1 can be broken down into two categories: Infeasibility and Inefficiency.

Infeasibility

Simply put, the original engine idea was inadequate for learning. The original engine was highly efficient at producing a single move at the cost of infeasibility in its entirety. Specifically, there

was only a single move generated. In order to train an engine, at each new board state all possible moves need to be presented in order to choose the optimal action. To put it bluntly, the engine needs choices that it can optimize. While this oversight completely invalidates the first iteration, the general ideas behind move generation had already been thought out as we'll see in Iteration 2.

Inefficiency

While the original engine was highly efficient at producing a single move, and as stated previously it was entirely overshadowed by the fact that it was infeasible, it was also highly inefficient at producing a single type of move. Specifically, moves involving check. A move in chess is a checking move if the king is under attack. Perhaps then, the most intuitive way to determine if white's king, for instance, is in check is by examining all of black's moves. White's king will not be under attack so long as black has no moves that can directly attack the king.

While the above definition of check is entirely accurate as it pertains to chess, it also brings with it an inefficient approach in finding checks. Checking moves in chess **must** be resolved immediately. Failing to do so, or rather the inability to do so, results in checkmate, ending the game. For this reason, in order for white to make a move, white must first consult with all of black's following moves. If black is able to capture white's king following white's move, then white is making an illegal move. The inefficiency arises from the fact that at each step of the way, all of white's move must be generated as well as all of black's move in response to each one of white's moves. This expansion of moves in order to determine check is precisely where the inefficiency lies.

Takeaways

The feasibility in a chess engine relies upon its ability to generate and store all legal moves, and the efficiency lies in how these moves are generated. At this particular point, moves pertaining to check are the bottle neck that must be resolved. While inefficiency does not directly impact an engine's ability to produce a move, it does impact the time it takes to produce an optimal move, thereby hindering learning.

3.1.2 Iteration 2

This version of our chess engine fixed the major drawbacks from the first iteration. For one, we now generate **ALL** moves at each board state and store them in a list for future reference. Additionally, checks are generated in a more complex yet efficient manner. While this iteration is far from perfect, this version stands as the true framework for any future developments as this model is able to generate and evaluate board states in a timely manner necessary for move prediction.

Feasibility

As previously stated, the main takeaway as far as feasibility is concerned lies in the fact that all legal moves are generated and stored at each time step. More specifically, each turn requires the generation of all legal moves, followed by storing them in a list. Consequently, a player (or AI) can only make a move so long as said move is in the legal move list. Moves are generated by looping through each square on a chess board, and so long as the square is not empty, localized moves are generated and stored.

Efficiency

Checks are the most pivotal concept in chess, not only from a competitive standpoint but also from a programmer's perspective. Considered "forcing moves", checks require immediate response from the given player. Any move that does not immediately resolve a check is illegal; any move that proceeds to put one's own king in check is illegal. As such, addressing checks needs to be handled in a very precise manner. While this complicates things in the short turn (as far as developing a working game goes), it also allows for enhanced efficiency when it comes to board evaluation.

Rather than handling checks by expanding outward from each opposing piece, a much more efficient approach is expanding outward from our king. There is no reason to generate moves that we already know will not attack our king (think a pawn that is across the board). While added board logic is necessary to properly evaluate checks, this is a necessary improvement for our engine later on.

3.2 Evaluation

3.2.1 Engine Evaluation

There are two components making up our evaluation function: A model trained on stockfish evaluation scores and an evaluation heuristic. Equally important, each one serves a distinct purpose. The stockfish trained evaluator is designed as a baseline when the heuristic fails to capture some board state. Seeing how complicated chess is as a game, it's impossible to cover all bases when devising a heuristic. Chess openings, in particular, can be problematic, while middle game trades are brilliant. On the other hand, the stockfish model lends itself towards openings and development, while being lackluster in coordinating attacks and dealing with threats itself. As such, the two compliment each other quite nicely.

3.2.2 Data and Stockfish

The data necessary for a general baseline is rather simple: Piece positions, castling rights, en passant, evaluation and turn recording (is it white or black's turn?). Castling is a pivotal aspect of chess as it directly relates to king safety. En passant is also necessary as the evaluation of certain board states is completely different with and without en passant. En passant is a more specific chess move involving a pawn capturing an enemy pawn in a different way than normal. The data source used in this case consists of roughly 35 million different board states with their respective stockfish evaluations (BingBangBoom, 2024, "Stockfish Evaluations"). That being said, for the sake of efficient testing, only 5 million data points were considered.

fen string · *lengths* 24 80	depth int64 1 245	evaluation string · *lengths* 2 7	best_move string · *lengths* 2 7	best_line string · *lengths* 2 67
7r/1p3k2/p1bPR3/5p2/2B2P1p/8/PP4P1/3K4 b - -	39	0.58	Kg7	Kg7 Re2 Rd8 Rd2 b5 Be6 Kf6 Bb3 a5 a3
8/4r3/2R2pk1/6pp/3P4/6P1/5K1P/8 b - -	58	0.00	Ra7	Ra7 Ke3 Ra3+ Ke4 Ra2 h4 gxh4 gxh4…
6k1/6p1/8/4K3/4NN2/8/8/8 w - -	87	M18	Nd6	Nd6 Kh7 Kf5 g5 Nh5 Kh6 Ng3 Kg7 Ke6…
r1b2rk1/1p2bppp/p1nppn2/q7/2P1P3/N1N5/PP2BPPP/R1BQ1RK1 w - -	25	0.24	Be3	Be3 Rd8 Rc1 d5 cxd5 exd5 exd5…
6k1/4Rppp/8/8/8/8/5PPP/6K1 w - -	99	M1	Re8#	Re8#
6k1/6p1/6N1/4K3/4N3/8/8/8 b - -	62	M27	Kh7	Kh7 Kf5 Kh6 Ng3 Kh7 Kg5 Kg8 Ne4…

Figure 1. *Sample data points from the data set* (BingBangBoom, 2024, "Stockfish Evaluations").

As stated, each data point consists of piece positions, castling rights, en passant, evaluation and turn. Board states are represented in Forsyth-Edwards Notation (FEN). Unintuitive at first, FEN notation is one of the easiest ways to succinctly represent piece positions, castling rights and en passant (Chess.com, n.d., "Forsyth-Edwards Notation (FEN)).

As an example, consider "7r/1p3k2/p1bPR3/5p2/2B2P1p/8/PP4P1/3K4 b - -" (BingBangBoom, 2024, "Stockfish Evaluations")**.** Starting with the top row of the chess board, "7r" represents 7 empty squares followed by a black rook (black pieces are in lower case and white pieces are upper case). "/1p3k2" represents an empty square followed by a black pawn, three empty squares, black king and 2 empty squares all on the second row from the top ("/" represents a row change). Continuing on, we see "b - -" at the end representing respectively, it's black's turn, there are no castling rights and no en passant (shown by "-").

3.2.3 Architecture

Our initial thought was to use linear layers to capture the 70 dimension tensor (64 from the 8x8 board, 1 from turn, 1 from en passant, and 4 from the four distinct castling rights). The oversight, however, lies in the fact that while there are roughly fifty possible moves at any given position, there are exponentially many more possible move classes (Oshri, B., Khandwala, N., 2015). The same move yields wildly different outcomes for even slightly different board states. Pawns, seemingly the simplest piece on the board, are in fact the most complex. One of the defining traits of a grandmaster is their pawn usage, both in terms of threats and structure. Devising attacking threats (typically on pawns) is necessary for building longstanding advantages. Even more complex are pawn breaks, backwards pawns, connected pawns, knowing when and when not to trade pawns. These aspects seem rather trivial, but the simplest avenue to success often comes down to a single pawn. All that to say even the simplest move has an avalanche of outcomes.

While linear layers are inadept at capturing the spatial dependencies of an 8x8 chess board, they are well suited to the scalar features of "turn", "castling rights" and "en passant". Instead, two convolution layers each followed by a ReLU activation function capture piece positioning prior

to flattening and feeding through two linear layers. The first linear layer combines the scalar features with the spatial board features, and the final linear layer outputs an evaluation score.

3.2.4 Heuristic

Although convolution layers are able to capture spatial features, it is less clear as to what extent the complex patterns of chess can be captured (Oshri, B., Khandwala, N., 2015). As such, even the best engine in the world, Stockfish, utilizes a combination of heuristics and neural networks to capture state evaluations. As already mentioned, our heuristic defined evaluation function lends itself more towards midgame tactics, while the stockfish trained model lends itself more towards opening development.

3.3 Devising a Ruleset

3.3.1 Checkmate

The main goal of chess is checkmating the opposing king. As such, a checkmate is awarded the greatest number of points, and our engine will hence seek out checkmate whenever possible.

3.3.2 Capturing and Defending

While checkmate is the goal, chess can be broken down into many subgoals. The highest subgoal on this totem pole is undeniably piece valuation, specifically in the context of capturing and defending pieces. Pieces are often valued according to the following: Pawns are worth 1 point, knights are worth 3 points, bishops are worth 3.xx (bishops tend to be slightly more valuable than knights), rooks are worth 5 points and queens are worth 9 points. Our heuristic defines the pieces in the same manner with the specification that bishops are worth 3.30 points. Something to note is that kings don't have a specific number value, although their value is inherently tied to checkmate.

3.3.3 Positioning and Development

Despite each piece having a specific value tied to it, this is by no means the end all be all. In the grand scheme of things, positioning is far more important than individual piece value. A rook trapped in the corner provides nowhere near the value as a centralized knight. In a similar light,

coordinated pieces have more leverage than a single attacking piece. Our heuristic takes these reasons into account in two ways: scaling the value of a centralized piece and diminishing the value of an undeveloped piece. Simply put, pieces in the central 4x4 squares have a slight edge when it comes to value, and pieces (not including pawns) on their home square have diminishing returns. This encourages our AI to develop pieces towards the center where they're most impactful.

Pawn "development" can be handled in a simpler manner. While it's important to develop the minor pieces (knights and bishops) quickly, pawns as a whole are not held under the same scrutiny. In a general sense, the 4 central pawns tend to be the most impactful. Our heuristic already applies a scaling factor to centralized pieces, and as such we can award additional points to pawns further advanced up the board.

It's important to consider that these "bonuses" should not persuade our engine into blunders. As an example, suppose a pawn was awarded 1.5 points for advancing to a square where it can be freely captured. As it currently stands, our heuristic would assign -1 points to the capture, but 1.5 points for the advancement, a net of 0.5 points. Our engine would thus mistakenly advance the pawn into a free capture by our opponent. A simple solution to this problem is to award less bonus points than the piece is worth. We've found that scaling central squares by 20%, scaling home squares by -20%, and awarding pawn advancement as follows [0, 0, .1, .15, .2, .4, .75]. Note that as the pawn approaches promotion, the number of points increases more rapidly. Another important thing to note is that the promotion square is disregarded as promoting to a piece (typically a queen) is already factored in with the additional material.

3.3.4 King Safety

King safety and castling are synonymous in this context. Castling is advantageous in the vast majority of games. According to Lichess, there is roughly a 0.3 point swing when one side is unable to castle (Lichess.org, 2024, "Analysis Board"). As such, we've allocated 0.3 points to castling. That being said, in the rare instance where an early end game is imminent (pieces, especially the queen, are traded off early), there is no benefit to castling. In fact, it's advantageous in these scenarios to have a centralized king as the king is the most powerful piece

in the end game. We thereby only allocate additional points to castling in the event that there are more than 3 opposing pieces (disregarding pawns and the king).

3.4 Minimax

Chess can be broken down as trying to play the optimal move on one's own turn while assuming your opponent will play their best move. This type of decision rule is referred to as "Minimax" (Wikipedia.org, 2024, "Minimax"). In the case of chess, each iteration revolves around recursively expanding upon the entire move pool up to a certain depth while alternating maximizing and minimizing agents. Once the set depth is reached, or the game is over (checkmate / stalemate), the board state is evaluated. The move corresponding to the optimal board state is returned, indicating the optimal move to the engine. In our case, the board state is evaluated corresponding to our stockfish trained model in conjunction with our heuristic.

Since we are expanding upon the entire move set recursively, improving the speed of move generation is pivotal to efficiency. Similarly, pruning methods have been implemented to remove the need for expanding upon unnecessary move nodes. This pruning method is known as "Alpha-beta pruning" (Wikipedia.org, 2024, "Alpha-beta pruning"). As a specific intuitive example, suppose the board state is such that there is a one move checkmate on the board. That is, if the player whose turn it is does not defend carefully, they will be checkmated. Evidently, this makes move pruning streamlined: discard any move that does not immediately resolve the checkmate. This is handled by updating alpha (beta) on the maximizing (minimizing) player's turn when a better (worse) move is presented, and breaking out of the loop if alpha (beta) is ever greater than or equal to beta (alpha).

4 Results

Chess being a complex game, there are many ways to evaluate a player's (or engine's) performance. Openings, "tactics", end games, etc. are all things that go into being a great player. Magnus Carlsen, regarded by many as the best chess player in history, happens to be the undeniable greatest end game player ever. That being said, he's also not widely regarded as the best defensively minded player, nor is he regarded as the best tactically inclined player. All that

to say that chess skill is a culmination of many aspects. Most of which are rather hard to evaluate (end games being the most complicated aspect of chess as a whole makes it very hard to evaluate), yet tactical play and opening theory are all easy to quantify from a baseline level. As such, we will be evaluating our engine in these regards.

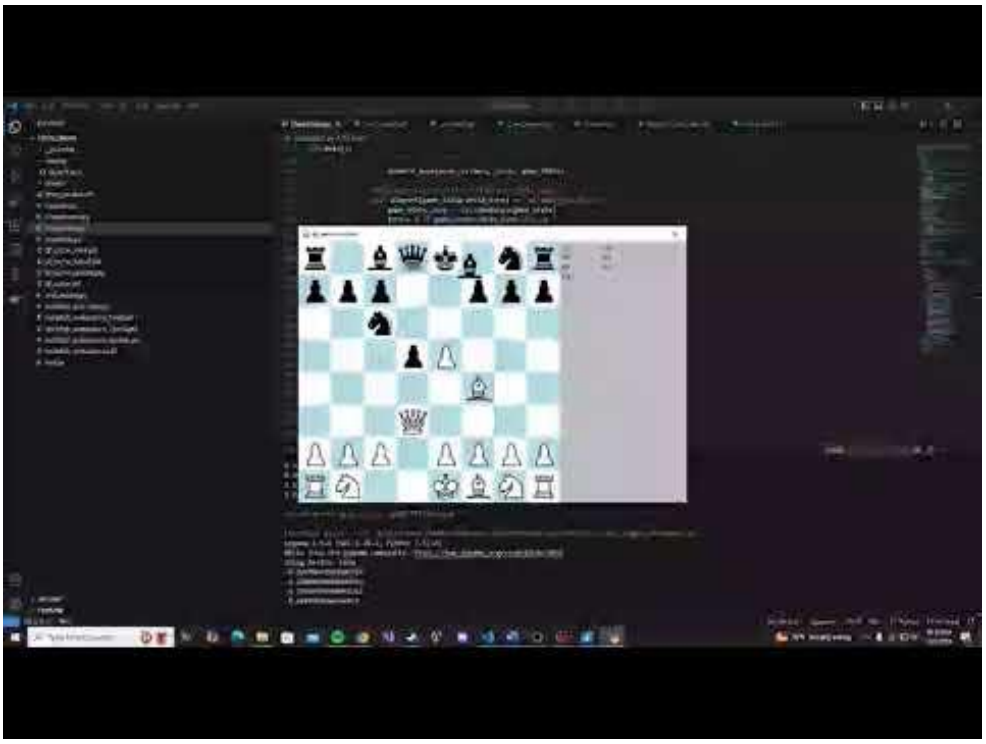
4.1 Defining Metrics

"Tactics" is a simple term used to describe all of the more complicated moves in chess (forks, pins, skewers, etc.). We will be evaluating our engine based on its capability to properly defend a piece, along with its ability to properly attack a piece.

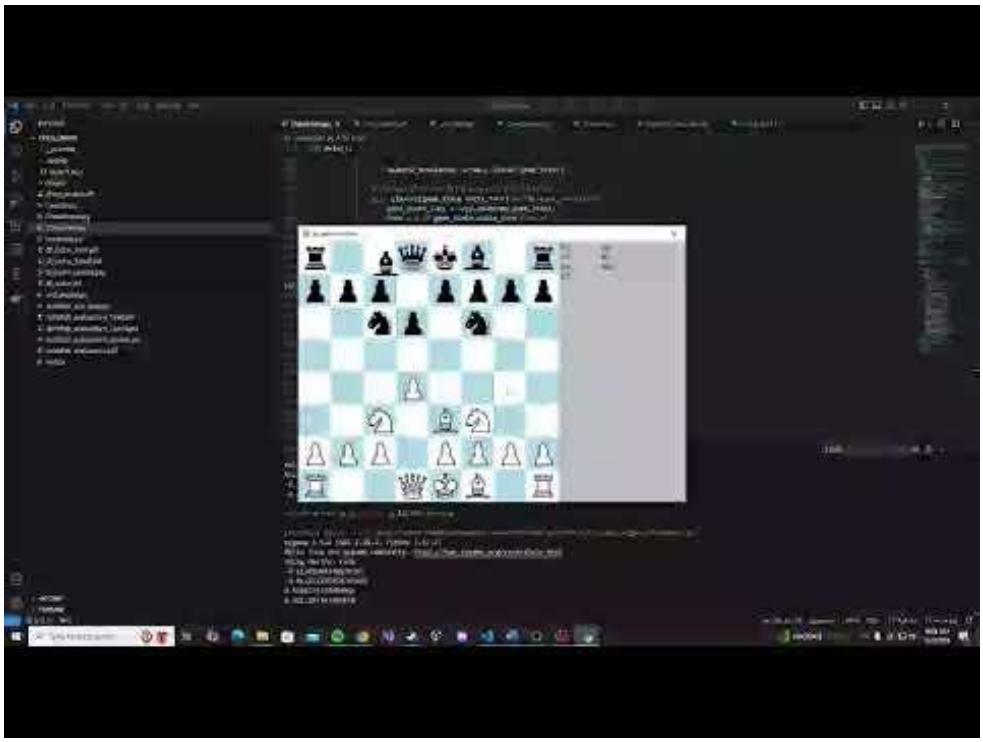
Opening theory is arguably the simplest aspect of chess to understand: Develop your pieces on strong squares.

4.2 Evaluation

After testing our engine over numerous games, it's apparent that our engine will never tactically blunder up to a certain depth. While our engine is certainly capable of calculating moves to high depth, it's unrealistic if it takes hours to make a move. Hence, we testify that our engine will not blunder up to a depth of 4. Additionally, our engine develops its pieces in a *mostly* logical manner, prioritizing the center.



Video 1. *Showcasing the engine's (white) ability to spot tactics and win pieces.*



Video 2. *Showcasing the engine's (white) early prioritization on development and opening theory.*

4.3 Improvements

There are many avenues already being considered to both improve the functionality of the engine, but also improve the efficiency. For one, the utilization of bitboards to represent a chess board would greatly improve the speed in generating moves. Bitboards represent each square of a chess board as a single bit of the 64-bit word encompassing the entire board (Wikipedia.org, 2022, “Bitboards”). Move calculations then involve shifting bits. As stated previously, increasing the efficiency of an engine increases the strength of the engine by allowing calculations up to a greater depth.

Aside from changing the chess functionality itself, modifications to the evaluation of a board state are already being considered. Simplest of all is increasing the utilization of data to improve our chess model. Only 5 million data points were considered in the development of our engine; this number can be greatly improved, as well as being more specialized. For instance, models can be developed with the intention of a specific aspect of chess. Openings in particular can be retrieved from a database and trained upon, just as end games can.

Certainly, the model architecture can be more fine-tuned, which is an ever-growing process. From the hyperparameters, to the input and outputs of the convolution layers, to the layers themselves, the model architecture requires immense testing to perfect.

In a similar sense, our heuristic can use added improvements that will both increase precision while also improving speed. Algorithmically, our alpha-beta pruning method can be further enhanced by considering stronger moves first. While it’s impossible to know definitively which moves are stronger, we can probabilistically assume which moves will be strongest in a similar manner as to over the board chess. For instance, it’s often stated that the first things to consider are checking / attacking moves. These moves tend to lend themselves towards more favorable positions. Hence, by also storing whether a move is checking / attacking, we can prioritize our search towards these moves first in hopes of pruning unnecessary trees. In a similar sense, our evaluation function can be improved by providing a slight edge to pieces based on the number of pieces they’re directly attacking / defending. For instance, a bishop that is attacking two pieces and defending two pieces, is likely stronger than a bishop that is only defending a single piece.

Evidently, this is collinear with wanting pieces on central squares. A knight in the center of the board has up to eight squares it can jump to while a knight on the rim only has up to four squares it can jump to.

Aside from functionality changes, one of the goals of this engine was to use it as a learning mechanism. In particular, training an engine to categorize weaknesses of a player based on their respective games would streamline learning. For instance, consider a chess player who happens to be particularly strong in the mid game, but struggles in the end game; or a player that is tactically adept, but struggles with pawn breaks. It can be hard to evaluate your own play and suggest which types of puzzles to train (tactics, openings, end games, etc.), but an engine capable of identifying one's level of play could act as a recommendation system in regard to which puzzles to train. Engines are already able to classify puzzles into the categories mentioned above, but none are catered towards one individual player.

5 Conclusion

Fine-tuning the various aspects of the chess program itself, the heuristic, as well as the convolution neural network were more cumbersome than expected. While the engine is unable to play at the level we had hoped, it is certainly capable of competing against an average 1,000 elo opponent. Future work will be targeted at the strength of the engine, as well as improved functionality and efficiency.

6 References

Chess.com. (2009). *A bust to the King's Gambit… Bobby Fischer.* Retrieved from https://www.chess.com/clubs/forum/view/a-bust-to-the-kings-gambit-bobby-fischer

Chess.com. (n.d.). *Chess Engine.* Retrieved from https://www.chess.com/terms/chess-engine

BingBangBoom. (2024). *Stockfish Evaluations* [Data set]. Hugging Face. Retrieved from https://huggingface.co/datasets/bingbangboom/stockfish-evaluations

Chess.com. (n.d.). *Forsyth-Edwards Notation (FEN)*. Retrieved from https://www.chess.com/terms/fen-chess

Oshri, B., Khandwala, N. (2015). Predicting Moves in Chess using Convolutional Neural Networks. *ConvChess.* Retrieved from vision.stanford.edu/teaching/cs231n/reports/2015/pdfs/ConvChess.pdf

Lichess.org/analysis. (2024). *Analysis Board.* Retrieved from https://lichess.org/analysis

Wikipedia.org. (2024). *Minimax*. Retrieved from https://en.wikipedia.org/wiki/Minimax

Wikipedia.org. (2024). *Alpha-beta pruning*. Retrieved from https://en.wikipedia.org/wiki/Alpha-beta_pruning

ChessProgramming.org. (2022). *Bitboards.* Retrieved from https://www.chessprogramming.org/Bitboards