DeepChess : Learning to play chess

Emanuel Lemus-Monge
H.E.C Montréal
Montréal, Canada
elm09@protonmail.com

Paul Conerardy
H.E.C Montréal
Montréal, Canada
paul.conerardy@hec.ca

Vincent Morel
H.E.C Montréal
Montréal, Canada
vincent.2.morel@hec.ca

Abstract

In this project our team implemented a Deep Learning model able to intuitively evaluate chess positions. Chess engines are usually a collection of feature engineered techniques. The process is tricky and involves a lot of trial and error. Our goal for this project was to avoid any kind of prior knowledge and follow, as much as possible, an end-to-end learning approach. To do so, our model leverage both unsupervised and supervised training based on top level computer chess games. To test our model, we made it compete against a well known classical engine. Although its relative simplicity the model is able to achieve human level of play in a couple of hours of training. The complete implementation and game interface can be found in the following notebook.

1 Introduction and motivation

Chess is an abstract strategy game opposing two players on a eight-by-eight grid with perfect information about the game state. Each player starts with sixteen pieces: one king, one queen, two rooks, two knights, two bishops, and eight pawns. But don't let those simple premises fool you. There's a reason this apparently simple game has captivated players around the world for centuries. After each player has made a move, there are 400 different positions[1], after two moves apiece 72 084 positions, more than 9 000 000 positions after the third move of each player [2]. The total game-tree complexity was first estimated by Claude Shannon to be around 10^{120}. This astonishing number of possible choices leaves room for players to express their creativity and ability to recognize advanced patterns. For this reason, historically, chess has been treated as a hallmark for human intellect (Grabner et al., 2007). But as soon as computers were first theorized, the conquest for the mastery of chess also became a computer scientists' endeavor. In 1953, Turing proposed the first algorithm able to play chess (Turing, 1953). But we would have to wait another 40 years for computers to be able to compete at the highest level. In 1997, in New York City, Deep Blue was the first computer able to defeat a reigning human champion (Hsu, 1999).

Ever since, the chess landscape has radically changed. The board game is now dominated by various computer applied techniques. The controlled environments that board games provide have always been interesting toy-experiments for researchers. This constrained world with a clear objective function (i.e to win) is a nice playground to test new theories and the artificial intelligence field is of course no exception. Recently, in 2017, Deep Mind famously managed to achieve state-of-the-art

[1] A position describe a particular board disposition at one moment of the game
[2] At a high level, games last on average 40 moves.

performance not only in chess but also in two other oriental variants (i.e Go and Shogi) (Silver et al., 2017). Reading about this legacy, our team was ready to try to tackle the challenge of playing chess with computers. The next section will give an overview of our approach.

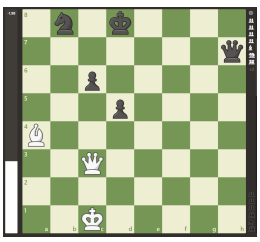
2 Scope definition

Our work draws its inspiration from DeepChess, an article published in 2016 at ICANN (David et al., 2016). At the time it was one of the first end-to-end Deep Neural Network able to achieve human Grandmaster level. In order to provide the model enough ground we will introduce, as only priors, the rules to distinguish legal moves and also an opening book to help speed up the search in early stages of the games. Our goal being to try to create a model with minimal prior human knowledge. Concerning the model itself, it will be composed of two modules, one for board position embeddings, while the second will be used to compare two given positions. At a high level, the model first learns to recognize advanced chess patterns with top level chess games, then, on the supervised phase the weights of the network are fine-tuned to be able to distinguish a winning position from a losing position. This second phase is crucial for the search of the game-tree and will be used for the pruning of the Minimax Search tree. It had been studied, notabely by J. Von Neumann, that in game theory, more specifically in sequential zero-sum games (like chess), minimizing the gains of the opponent leads to the optimal strategy of play for oneself (Neumann, 1928) (i.e. Nash's equilibrium). For this reason, the Minimax algorithm paired with Alpha-Beta pruning is usually the backbone of most chess tree searches. Our work is no exception, but the alpha-beta algorithm was slightly modified in order to compensate the lack of exact positional scoring. Instead of trying to predict an evaluation for every position, the network learns to compare two chess positions and predict intuitively the one that is most likely to win. To recapitulate, the policy of the model will be guided by a minimax search tree pruned by the alpha-beta algorithm, but instead of relying on a learned or engineered evaluation function to score every leaf of the tree, we just learn, at a high level, which position seem more favorable. This simple yet effective method can lead to surprising results as discussed in section 6.

2.1 Core components of a chess engine

Before diving into specifics, we'll take a moment to recall what is needed in order for a computer to play chess. Chess engines are complex, but they can be broken down into three core concepts. First, is the **search tree** needed in order to explore all the possible moves and game scenarios. Then, we need to give all those possible moves a score, this second part is called the **evaluation function**. This evaluation function will serve as guide for the **pruning strategy** of the search tree. Although zero-sum games, like chess, have been demonstrated to be optimally solved by minimizing the gain of the opponent, this strategy needs to rely on a perfect evaluation function, which as we will see in a second is hard.

2.1.1 Position evaluation



The most important component of chess engines is undoubtedly the evaluation function. It is responsible of selecting the best move based on the score it granted to each possible scenario of a game. But evaluating a move and the resulting board position is tricky. The naive approach would be to simply score a position for both players based on the remaining pieces on the board. For example, if Black has two pawns, a knight and its queen (1) while White only has a bishop and its queen on the board (2) (see Figure), we could calculate the absolute difference between White and Black positions as follows ;

$$Black\ evaluation : 2\ pawns{*}1\ point{+}1\ knight{*}3\ points{+}1\ Queen{*}9\ points = 14\ points \quad (1)$$

$$White\ evaluation :\ 1\ bishop * 3\ points + 1\ Queen * 9\ points = 12\ points \quad (2)$$

Absolute value of the difference between Black and White positions would yield a value of 2 which could then be normalized in order to give an evaluation ranging from -1 (Black won) to +1 (White won).

But as one can imagine, this simple evaluation only considers what is called the *Material* value but overlook more advanced concepts such as *Pawn Structure*, *Center Control*, *King Safety*, *Pins*, ... A more exhaustive list of those concepts can be found in the Appendix 1. Ever since its debut, Chess Theory has grown immensely. To give a rough estimate, the largest private collection of chess books, Schmid's library, is said to contain between 20 000 and 50 000 books (Geuzendam, 2010). The question is, how can we condensate all this accumulated knowledge into useful features to evaluate positions for our computer engine ?

3 Related works

As previously touched, the literature surrounding machines playing chess has a long history. Today, two opposing paradigms are now dominating the pack, with one steadily overthrowing reigning uncontested champion since the start of Computer Chess Championship. Let's explore these two approaches.

3.1 Linear evaluation approximation

As it would be practically impossible to program all the high-level concepts that human masters have come up over the years, the evaluation function typically consists of a collection of sophisticated domain-specific feature engineering trying to condensate human knowledge into usable features. In order to avoid discrete optimization, evaluations functions are usually linear combination of independent features with their associated weights;

$$Evaluation = \sum_{i=1}^{n} F_i * W_i \quad where\ f(a+b) = f(a) + f(b) \tag{3}$$

To be more precise, advanced chess engines use a collection of additive evaluations functions in order to achieve top performance. But picking features that are truly independent with no hidden dependencies is tedious and requires patience in addition of domain knowledge. Let's consider a chess program using this kind of approach.

The most successful chess engine in recent memory is unquestionably Stockfish. Since 2015 Stockfish has won the 8 T.C.E.C [3] on the last 12 competitions that have been held. Originally developed by a team of three devoted programmers, the project has now expanded and is constantly receiving updates by the chess programming community. To this day, the repository has been forked more than 1 700 times (Github, 2021). As mentioned, feature engineering is heavily used for the evaluation component of the engine but the search and data representations have also been ponderously tuned over the years in order to achieve top performance. A list of Stockfish's engines features is available in the Appendix 2.

3.2 Non linear evaluation approximation

What if we could model a learning agent to learn itself all those features instead of hand-picking them? This is precisely the question DeepMind tried to answer in their 2017 paper, Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm (Silver et al., 2017). The aim was to find a general approach able to learn to play chess (also Shogi and Go) at a super-human level without adding any domain knowledge (except for the rules of the game). At a high level, Alpha Zero is composed of three main stages:

- Self-play: Generate data from the current best neural network.
- Training: Using data generated by self-play in order to tune the network
- Evaluation: Assess if the trained network is better or worse than the current best model

[3] Top Chess Engines Competition.

Alpha Zero starts from random play, knowing only the basic rule of the game it is playing. After the training phase, the model outputs a vector of move probabilities (i.e. The policy of the agent) and an evaluation of each position. Instead of relying on a hand-crafted linear evaluation function, the deep neural network uses a non-linear function approximation. This allows the model to learn powerful representations and get a deeper understanding of chess positions thanks to more context-dependent estimates. A Monte-Carlo Tree Search using the learned policy to narrow searches only to high probability moves. Instead of using what has been considered for the last decade as the hallmark of chess search heuristic (i.e. alpha-beta search), DeepMind's team pivoted and went for a MCTS approach. This simulation-based approach was often proved unsuccessful as it also introduces the risk of approximation errors by taking the average of a subtree. But thanks to the huge amount of computation power[4] used to generate quick games and knowing that each MCTS used 800 simulations (guided by the learned policy) approximation errors tend to be canceled out. Although seemingly anecdotal, using MCTS paired with DeepMind's neural network allowed the model to learn previously unachieved levels of chess representations. For detailed figures describing the CNN-based architectural choices of Alpha Zero for its body and heads, refer to the Appendix 3. And although Google subbranch, Deep Mind, decided not to release publicly Alpha Zero, it reportedly crushed Stockfish in a new match of 1000 games, losing only 6 games and winning 155 games (Other games were draws) in a peer-reviewed article released a couple of months after the original publication. Alpha Zero also asserted its dominance by displaying a much more efficient search, dominating Stockfish even with a 10-to-1 time ratio (Silver et al., 2018) handicap for every move. Those results resonated like a shockwave in the Chess programming community. Since then Stockfish has been forced to add a Neural Network component into their heuristic search in order to keep up with the new competition. A figure representing Stockfish's network and how it works in pair with the classical engine is shown in the Appendix 4. This addition allowed Stockfish to reach new ELO heights at an unprecedented speed [5].

4 Datasets

In this section, we will describe the data used and the processing applied on the original dataset. Chess is one of the most studied games in history. Earliest records date back to the 15^{th} century (Castellvi, 1475). Publicly available high-quality games are easy to find. We opted for the Computer Chess games dataset (ComputerChess, 2021), keeping only decisive games [6]. Wins and losses will be our labels for the supervised part of the training. From those games we only kept 10 randomly chosen positions throughout the game. We believe this number of sample allow enough granularity in order to learn various games stages (i.e. Beginning, Mid and End games). We also excluded positions within the first five moves of the game and preceding a capture. The very early stages will usually be taken care by the opening book we use to speed-up the search. And positions preceding captures don't yield any structural information as to how it lead to this position. The positions were then transformed into board represented by arrays of binary variables for efficiency. Each board consist of 773 bits (2 players x 6 type of pieces x 64 squares on the board + 5 bits containing castling rights and which player turn it is). Those 'bitboard' representations were then transformed into tensors that could be fed into the encoder. See figure 1.

5 Methodology

Alpha Zero's approach is attractive to say the least. But the computational resources needed to reproduce a similar experiment aren't realistic in this context. So instead, we opted for a simpler neural network approach inspired by DeepChess's article (David et al., 2016), leveraging both unsupervised and supervised learning methods. The former will be pre-trained to initialize the weights of the latter. The final pipeline will take two board positions as input and output a prediction of the position that is most likely to win. Let's go through each specific component step by step.

[4] 5000 TPUs to generate games and 64 TPUs to train the model.

[5] An historical progress of Stockfish is presented in the Appendix 5

[6] Games that lead to draws do not provide labels and are less optimal for learning decisive features

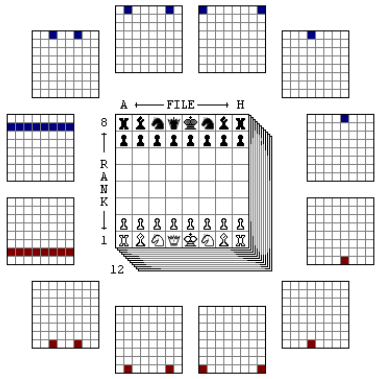


Figure 1: Bitboard representaion : A one-bit inside a bitboard implies the existence of a piece of this piece-type on a certain square.

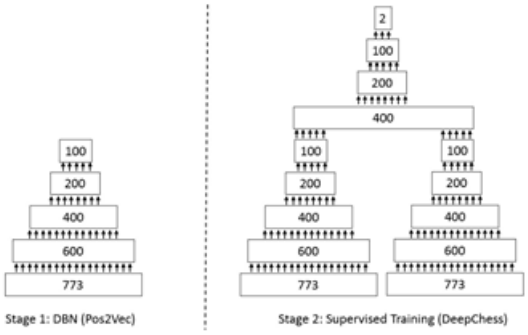


Figure 2: Original architecture of the positional encoder (left) and the positional comparator (right).

5.1 Positional autoencoder (Pos2Vec)

This module will learn to transform board positions into an embedded vector. In order to learn representations in an unsupervised manner, we first encode the boards and try to decode them into their original size by minimizing the mean squared error between the two. For the encoding, the network is composed of five fully-connected layers of size: 773 – 600 – 400 – 200 – 100 and inversely the decoding part: 100 – 200 – 400 – 600 – 773. Concerning the data, we randomly selected 1 000 000 winning positions, as pre-training on losses would impact negatively pre-trained weights. We used Adam paired with a learning rate of 0.0005 and trained the model during 50 epochs for this part. The pre-training was performed during overnight (lasted approximately 12 hours). For the following stage, only the encoding part was used in conjunction with the second module of the pipeline.

5.2 Positional comparator (DeepChess)

This module is made-up of a Siamese network using two copies of the previously pre-trained encoder with the addition of linear layers added at the end. The model can now take two boards as input and compare which one is the most likely to win. Those two disjointed copies will be responsible of learning high-level features of given positions. The five next layers will be in charge of comparing the learned features in order to determine the best position between the two position inputed. An ablation study was performed on this part of the network and lead to a modification of the proposed architecture. We opted for larger linear layers of size: 500 – 1048 – 500 – 100 – 2 adding more capacity and flexibility to our model. The two modules (i.e. Siamese encoder and DeepChess) are updated jointly throughout the supervised process. Note that the weights of both positional encoder are tied together to ensure consistency in the encoding of both board positions. During the training, the network takes random pairs of positions. Each pair is composed of one winning position and one

losing position. ReLU activations were used here. To calculate the loss, we went for a cross entropy loss. Finally, Adam and a learning rate of 0.0005 were also used here. This supervised training was a shorter, lasting around 3 hours. The full network was then used as the evaluation function backing a modified version of the classical Alpha-Beta Search used in traditional chess engines.

5.3 Modified Alpha-Beta algorithm

As discussed earlier, building a good approximation function for the evaluation of chess positions is tricky. But it is crucial in order to reduce/prune the tree complexity in a full game of chess. But instead of trying to approximate reliably a specific value for each position, here, the problem is alleviated by taking a more intuitive approach. Instead of having to rely on a precise evaluation of every leaf investigated in the search tree, the network learns to rule-out less interesting position while retaining in memory the best one. This organic approach of intuitively comparing board positions is similar to the way of human players explore possible scenarios in a chess game.

5.3.1 Description of the original algorithm and modifications

Alpha-Beta is a pruning technique used for the depth-first search tree of the Minimax algorithm. It is an adversarial search algorithm meaning that one refutation (i.e a move worse than the current best move) is enough to avoid further exploration of the subbranch. This dramatically reduces the complexity of the original tree. Also, this pruning technique does not overlook any optimal value and yields the same result as a standard minimax search tree would.

The algorithm maintains two values, alpha and beta, representing the minimum score that the maximizing player is assured to have and the maximum score that the minimizing player is assured to have respectively. Alpha is initialized at negative infinity and beta at positive infinity (i.e. Both players start with the worst case scenario). Whenever the max score of the min player (Side to move player or Alpha player) is assured to become less than the min score of the max player (Opponent or Min player) is assured (i.e. $\beta < \alpha$), this subtree is pruned. Inversely, alpha and beta values are updated. Because of the lack of exact value for each consider position, we used instead the likelihood of winning for each position in our values of α_{pos} and β_{pos}. The logic remains unchanged for the rest. The pseudo-code for the Alpha-Beta algorithm is shown below. We used a publicly available implementation of the Alpha-Beta algorithm [7] that we modified to fit our problem.

```
function alphabeta(node, depth, α, β, maximizingPlayer) is
    if depth = 0 or node is a terminal node then
        return the heuristic value of node
    if maximizingPlayer then
        value := −∞
        for each child of node do
            value := max(value, alphabeta(child, depth − 1, α, β, FALSE))
            α := max(α, value)
            if α ≥ β then
                break (* β cutoff *)
        return value
    else
        value := +∞
        for each child of node do
            value := min(value, alphabeta(child, depth − 1, α, β, TRUE))
            β := min(β, value)
            if β ≤ α then
                break (* α cutoff *)
        return value
```

6 Empirical Results

6.1 Implementation of the testing framework

In order to automate the testing and result generation of our model, we implemented a way to simulate playing multiple games against opponents of various strengths. The model is also able to play chess puzzles. To simulate different opponents we implemented the Stockfish chess engine[8] for which it is possible to set a particular level of play. Playing a game imply randomly setting the

[7] Owen PS repository : https://github.com/owenps/RhinoChess

[8] https://stockfishchess.org/

color (i.e. White or Black) and let Stockfish play the opposing color. We then iteratively ask each engine to find the best move to play given the current state of the chess board. To further test our architecture, we wanted to ask our model to solve specific chess situations called *Chess puzzles*. Those kind of puzzles are easily available to online chess players[9]. This requires to set the board in a specific state and ask the model to perform a prediction. To do so we relied on the FEN notation which is a way of representing a chess board as a string. With this FEN string we would then recreate a board and ask our model to perform a move prediction.

6.2 Elo estimation

Players' performance in chess is usually represented using the Elo score. But getting an estimate of our model's performance can be difficult without a large enough pool of players for whom we know their exact ranking. Instead we opted for a strategy to make our model play a lot of games against a regular engine for which we can set an exact Elo score in advance. The comparative nature of the Elo scoring system makes it simple to derive an approximation of the rating of our model from its win-rate ratio against Stockfish at different levels of play.

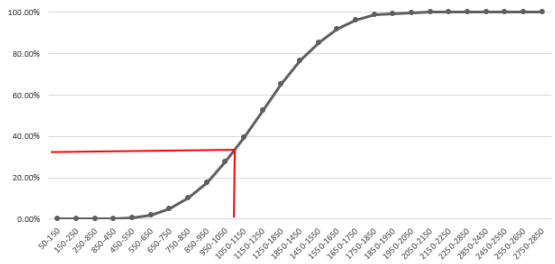


Figure 3: Distribution of players' Elo scores on chess.com.

Getting an idea of the distribution of players' Elo scores is a bit more difficult as it is usually recorded during official tournaments. But there aren't a lot of records of chess at amateur level so it is hard to infer beginner and middle-level players' elo distribution. To try to capture the Elo score of players of all skill levels we pulled the Elo score of players on the chess.com website. Looking at statistics, chess.com is currently the most popular website to play online chess. This allowed us to consider all level of play. The median of the elo distribution hovers around 1200 elo. 50% of players are situated between 1000 and 1300 elo. Our model was able to reach an elo rating of around 1100, placing it above 30 to 35% of all players on chess.com (See Fig.3).

6.3 Puzzles

Puzzles are a very common way to learn and test your abilities in chess. They set the player in a specific situation where they are given a chess board and are asked to find the best move possible. This allows us to test our model against various scenarios and get a better intuition about our model's strengths and weaknesses. This second test allowed us to better grasp our model's understanding of chess.

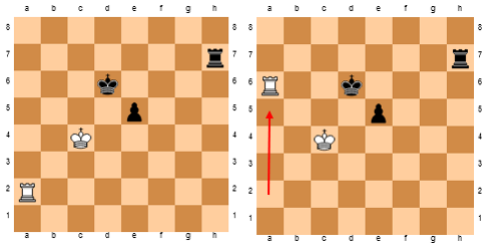


Figure 4: Optimal solution found by the model: a2a6.

[9] https://lichess.org/training

The model seems to display a capacity at finding strategic positions and checkmates, which is consistent with its capacity to complete full games. But it seems to have a hard time understanding the concept of piece value and appears to consider a player's advantage mostly on the number of pieces possessed.

We explicitly see this situation when proposing a board to the model where trading a piece would be the right move (i.e. willingly giving up a piece to force the opponent in a disadvantageous position), but instead our model plays conservatively and proposing a subpar move.

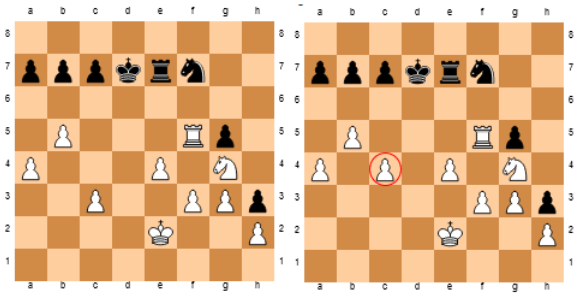


Figure 5: Wrong and conservative move found by the model: c3c4.

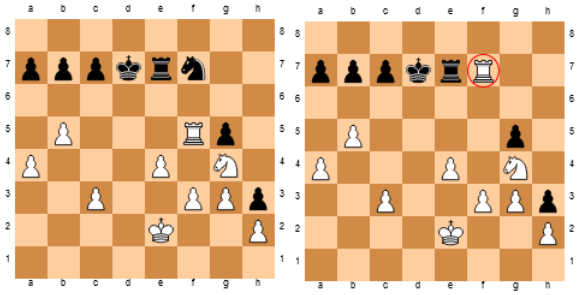


Figure 6: Expected move: f5f7.

7 Further discussion

7.1 The use of FFNs instead of CNNs for chess

At the early stages of our project, there was a lot of hesitation in the kind of architecture we should use. But a study comparing both MLPs and CNNs discussed the superiority of the former over the latter in chess. They highlighted the strength of MLPs in classification tasks and added that the full potential of CNNs wasn't fully leveraged by the 8x8 board used in the game (Sabatelli et al., 2018).

7.2 Qualitative predictions and Time efficiency trade-off

The quality of exploration of the full game tree is computationally expensive as stated in the introduction. In order to make predictions in a reasonable amount of time (i.e Around 3 minutes) we had to constrained the search tree. Two points limit the quality of our predictions in order to save time.

- The depth of the search tree. We used a depth of 3, but a deeper look-ahead could increase the level of play of our model.
- Book openings. We used one book opening in order to speed up the early search of the game tree. This limited library of openings handicaped our model if it doesn't recognize the opening played by the opponent.

References

Castellvi (1475). https://www.chessgames.com/perl/chessgame?gid=1259987.

ComputerChess (2021). http://www.computerchess.org.uk/ccrl.

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

Geuzendam, D. (2010). The finest chess collection in the world. *New in Chess Magazine*.

Github (2021). https://github.com/official-stockfish/stockfish.

Grabner, R. H., Stern, E., and Neubauer, A. C. (2007). Individual differences in chess expertise: A psychometric investigation. *Acta psychologica*, 124(3):398–420.

Hsu, F.-h. (1999). Ibm's deep blue chess grandmaster chips. *IEEE Micro*, 19(2):70–81.

Neumann, J. v. (1928). Zur theorie der gesellschaftsspiele. *Mathematische annalen*, 100(1):295–320.

Sabatelli, M., Bidoia, F., Codreanu, V., and Wiering, M. (2018). Learning to evaluate chess positions with deep neural networks and limited lookahead. In *ICPRAM*, pages 276–283.

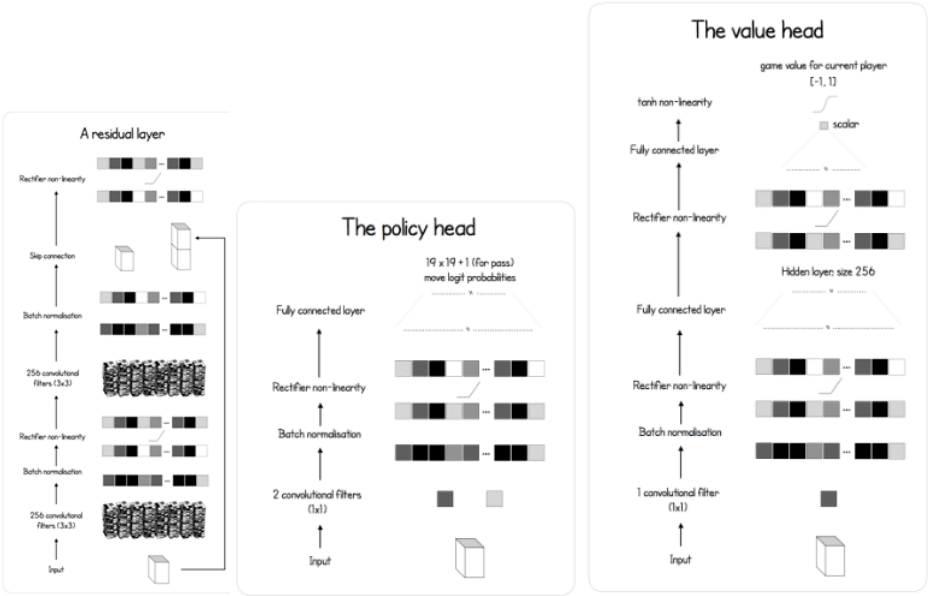
Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., et al. (2017). Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815*.

Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., et al. (2018). A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144.

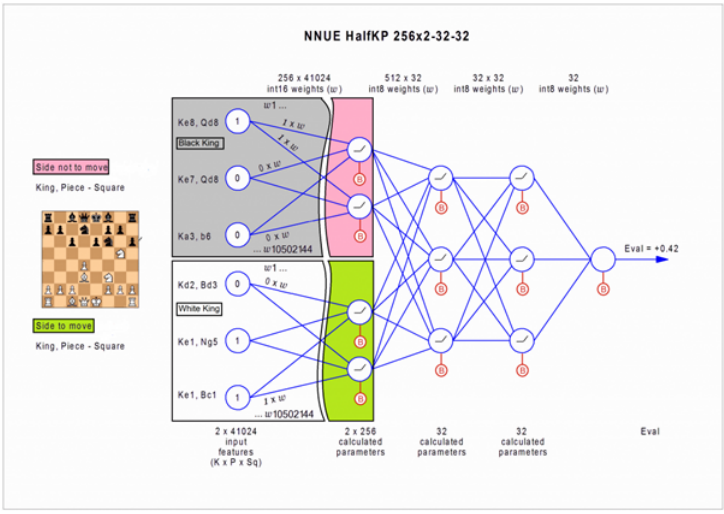
Turing, A. M. (1953). Digital computers applied to games. *Faster than thought*.

Appendix

[1] Advanced Concepts in Evaluation : https://www.chessprogramming.org/Evaluation#General_Aspects
[2] Stockfish features in Search, Pruning Strategy and Evaluation : https://www.chessprogramming.org/Stockfish#cite_ref-25
[3] Alpha Zero Architecture :



[4] Linear Layers used in the updated version of Stockfish NNUE.



[5] Elo progess over time. Stockfish NNUE describe the integration of a neural net in the engine.