IDP Report: Chess & Reinforcement Learning

Guillaume Pelat
Technische Universität München
Departement of Informatics
guillaume.pelat@tum.de

Abstract

A lot of attention has been drawn on chess when Deepmind's AlphaZero beat previous computer world champion Stockfish after training only through self-play. It showed how reinforcement learning could outperform both classical and supervised-learning based engines, the later hindered by a lack of data at the highest level. However, reinforcement learning requires a lot of computational power, and is not accessible without huge backing.

In this report, I investigate the use of supervise learning to initialize a network before training it with reinforcement learning. The proposed model works without any kind of tree search, playing with full strength at all time controls. However, while this is enough to play against humans, we investigate how the move selection procedure can be use in a more classical engine to speed up its search, greatly improving its efficiency.

1. Introduction

For years after IBM's Deep Blue[4] won for the first time against a human master, chess engines mostly relied on tree exploration, evaluating positions multiple moves in the future to gain the upper hand on their opponents. However, this method results in a very inhuman-like way of playing, and ultimately relies mostly on the position evaluation function. Hence, the first application of supervised learning to chess was to compare two positions without having to craft features. The gain in design was however offset by an overall much slower decision process.

Since then, work involving Deep Learning shifted from position evaluation to movement prediction. The release of AlphaZero[14] was a huge event in the world of computer chess: a program training only with reinforcement learning was able to beat the best chess engines, after playing 44 million games against himself. Based on AlphaZero's architecture, Maia[12] was specifically trained via supervised learning to imitate human moves and predict blunders, trying to get as close as possible to a human way of playing. To achieve this goal at a given level, a Maia model learns from 12 million human games.

These approaches, while giving impressive results, also have their drawbacks: learning through self-play requires a huge computational effort, while supervised learning requires an amount of human data unobtainable at higher levels of play. However, while a policy through reinforcement learning can be greatly accelerated with a good parameter initialisation provided by supervised training, this has to my knowledge never been applied to chess outside of evaluation functions.

In addition, the current level of chess engines is now far out of reach of human master. This is partly due to the fact that computers search thousands, if not millions of positions before deciding which is the best move. Therefore, this project focused on determining the best move without using any search mechanism. However, I also want to investigate whether such model can also be used alongside a classical search-based engine to improve the efficiency of search.

To achieve these goals, the method proposed in section 3 focuses first on determining the move chosen by human experts, and then trains by itself to complete its knowledge in domains where there is a lack of data. Details on how the model is trained are provided in section 4. Section 5 will show the results of our method, as well as the interest it has when using it along a traditional search-based engine. We will then discuss some shortcomings of our methods in section 6.

2. Related Work

2.1. Tree Search

Starting with Deep Blue in 1997, early chess engines' successes were fueled by two components: a good evaluation function and a fast search strategy in the possible positions tree. The evaluation function usually takes a leaf position of the tree as an input and returns a value representing

the advantage a player has on the other, and thus how likely he is to win from this position.

The main difficulty when searching lies with the branching factor of chess, with each position having in average 35 legal moves and rendering minimax search highly impractical. The most wildly used technique to reduce the number of nodes processed seems to be alpha-beta pruning (shown in 1). Because the gains of alpha-beta pruning depends on move ordering, most strong engines also use some kind of move evaluation. For example, Sunfish classify moves using pieces values and position.

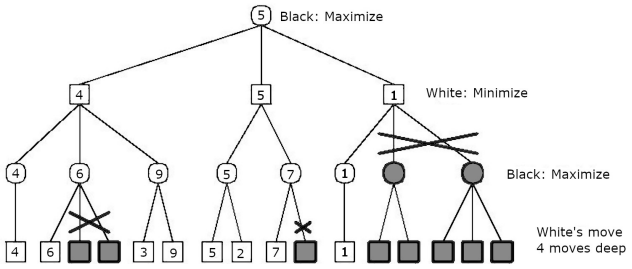

Figure 1. Alpha-Beta search illustration[1].

In traditional engines, the evaluation function relies on hundreds of hand-crafted features that require both expert knowledge and experience and are notoriously difficult to tune[2, 17]. Therefore, training an evaluation function with deep learning, allowing to derive a good function with little previous knowledge, was a hot topic of research for at least twenty years.

2.2. Neural networks in Chess

The first examples of automatic evaluation function still rely on hand-crafted features, but the weights and shape of the function is trained. Mennen[11] trained different networks to evaluate a position depending on the stage of the game. Veness et al.[19] use 1812 features per position, and try different bootstrapping strategies. This is also a situation were genetic algorithms perform quite well [3, 7].

In 2016, David et al. introduced DeepChess[8], presented as the first end-to-end neural network. A position is converted in a bit string describing the occupancy of each board square, and the network learns to predict the most favorable position between two. The network is able to play at a grandmaster level, but is still too slow.

Using convolution layers instead of DeepChess' dense layers is a way to speed up calculation by making full use of GPUs, AlphaGo was able to achieve its supremacy on the game of Go by using only convolution layers[13]. Cazenave [5] showed that adding residual connections allow for faster training and higher move matching accuracy. Residual blocks formed the structure of AlphaGo's successor, AlphaZero[14], beating all other engines in Go, Chess and Shogi after training only through self-play for around 44,000,000 games.

2.3. Supervised vs Reinforcement Learning

Most chess engines relying on Deep Learning are trained through supervised learning, which is limited by data availability. While it is possible to train an evaluation function with few sample (DeepChess used 640,000 games), it is much harder to predict a move without relying on a search procedure. McIlroy-Young et al. tried with Maia[12] to replicate human moves at different level of play. They needed 12 million games per target level, a quantity which is not available at a highest level. This high requirement is likely due to the diversity of endings[2].

The ending is indeed a difficult phase of a chess game : human experts spend countless hours studying how to terminate a game in a few moves. Positions up to seven pieces have all been solved since 2012[20], but such databases are often too big to process since they take terabytes of data.

Training using Reinforcement Learning can be seen as an alternative, and AlphaZero as shown its effectiveness. It is also the key behind the success of Giraffe [10], which used a small database of position as starting points for self-play. However, the features used by Giraffe require a lot of chess knowledge, while AlphaZero needed to play millions of games. While it is well-known for achieving the highest level of play in less than four hours, this was using more than 5,000 TPUs. Leela Chess Zero, built after AlphaZero's paper, took over 3 years to get the same result.

3. Method

3.1. Position and Move Representation

A position is represented by 13 8x8 planes. 12 are encoding the pieces' position (one per color and piece type), and the last one encodes the free spaces on the board. As an input, the network uses the plane descriptions of the current board and the last two times it played (when starting a game, the two previous positions are the same as the current one), as well as a plane encoding the color (1 for white and 0 for black) and the current move number, inspired by [14].

Using 6 planes per position for encoding the pieces has been tried using 1 for white pieces and -1 for black pieces, but diminished the move-matching accuracy by around three points. On the other hand, adding a plane with the board description using pieces values (positive for white and negative for black), or planes describing all pieces each player has on the board led both to faster initial improvement, but slightly worse accuracy at the end of training.

[1] From http://www.pressibus.org/ataxx/autre/minimax/node2.html

[2] https://github.com/CSSLab/maia-chess/issues/19

The output of the network is an array of 1965 values, each referring to a move describing following the UCI protocol (source square, destination square and pawn promotion if necessary). While they do not represent the entirety of possible moves in chess, they were the only ones appearing in our dataset. During the reinforcement learning phase, only two other possible moves were identified (a7b8b and g7h8b) and can be safely ignored.

3.2. Network Architecture

The architecture of the network is fairly similar to the one used by AlphaZero, inspired by ResNet[9]. The input planes go through a convolution layer with a 1×1 kernel, outputting $256 \times 8 \times 8$ features. Then follow three residual blocks and a head. All convolution layers except the first and last ones use a 3×3 kernel, with 0-padding to keep the number of features constant through the residual tower. All layers use ReLU activation and the output is softmaxed.

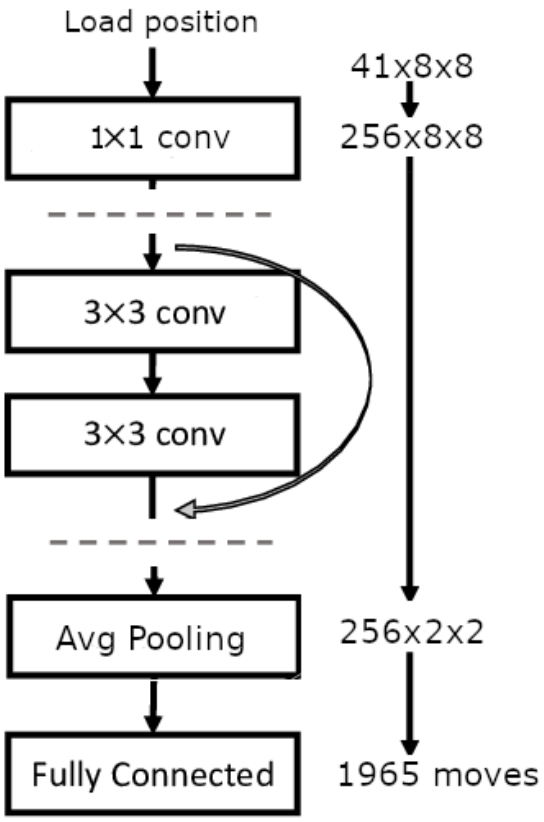

Figure 2. Architecture of the network for move matching. The residual block is repeated three times, and all layers have ReLU activation.

Outputs referring to illegal moves are masked, and the remaining probabilities are re-normalized linearly. When training, moves are selected according to their probabilities to ensure some kind of exploration, but during play we choose only among the two best propositions.

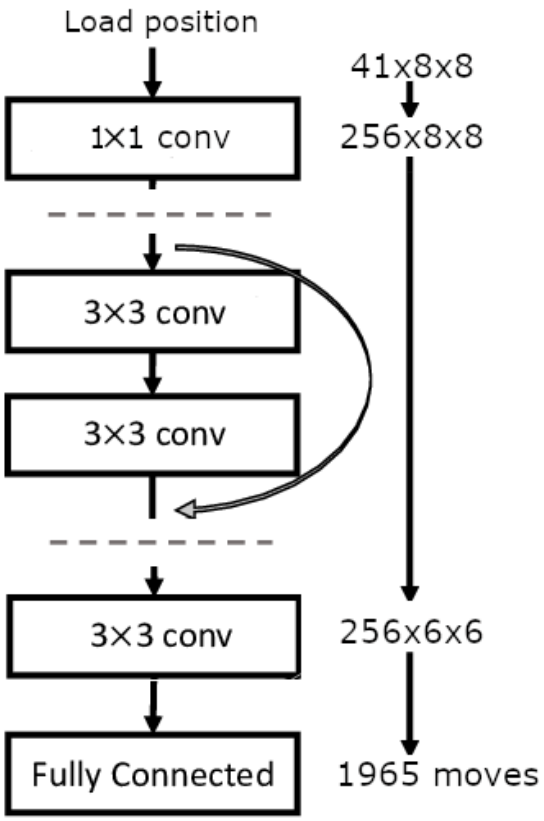

Figure 3. Architecture of the network for self-play. The residual block is repeated three times, and all layers have ReLU activation.

We used two heads during our training: a simple one comprised of a pooling and a dense layer for move matching showed in figure 2, and a second with a convolution layer and a dense layer for self-play as in 3. The idea behind these two heads is that the supervised learning allows for training the convolution layers to extract features useful for move generation.

3.3. Two-step Training

As previously stated, training the model happens in two phases. First, we perform a step of supervised learning, were we try to match moves made by humans in the same position. It is by performing this task that we devised the position representation and network architecture explained in the previous section; details on the data used and training parameters can be found in section 4.

Once the network is able to predict human moves with enough confidence, we proceed to the reinforcement learn-

ing phase. In a first experiment, a checkpoint of the network was used as an opponent; however, convergence was much faster by making the network play against itself. Therefore, we only maintain a fixed version of the network in order to check for improvements, and replace it when the current state of the network beats its previous self over 200 games.

In order to ensure exploration and to speed up training, the network does not play a full game at each step; instead, games ending with a checkmate were selected from the database and the network plays starting from a random position of the game. The reason for this is twofold: first, it ensures that the visited position are actually found in games. Second, it allows us to take the human outcome of the game in consideration when computing the reward (see section 4.3 for more details).

3.4. Probability-based search

Lai introduced in Giraffe[10] the concept of probability search, illustrated in figure 4. When the model outputs a probability $p(t, i)$ for each possible move, we can compute a probability of appearance for each node encountered in the search:

$$p(t+k) = \prod_{j=1}^{k} p(t+j, i)$$

were the succession of moves $(i, i+1, ..., i+k)$ leads from position t to position $t+k$.

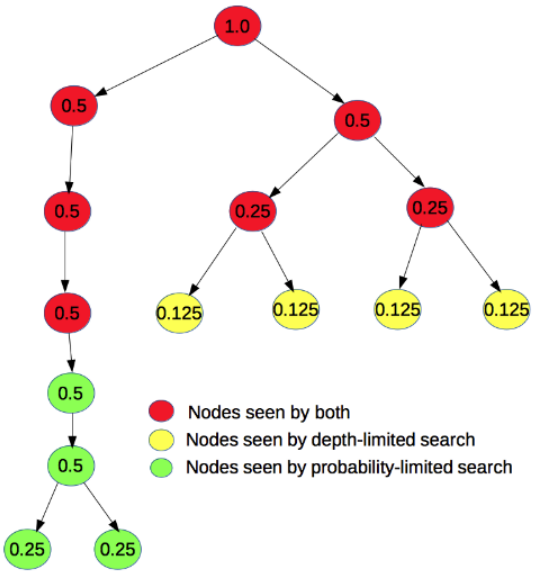

Figure 4. Comparison between alpha-beta search and probability search. From [10].

Therefore, we can replace the depth threshold by a probability threshold in the search function. This is especially useful in situations were there are a lot of possible moves, but one is significantly better than the others; a situation often found during the endgame, were a few pieces have large amplitude of movement.

3.5. Monte-Carlo Tree Search

In addition to alpha-beta and probability, there is a third kind of search: the Monte-Carlo Tree Search (MCTS). This is a kind of search used in particular by AlphaZero that is also suited to models that output a probability distribution.

During an MCTS search, a sequence of moves is simulated by the engine according to the model's move probabilities. The resulting position is evaluated, and after simulating a certain number of sequences, the move leading to the most favorable positions is selected. Another possibility, used in this work, is to play a few games from the current position and select the move leading to the most wins. Whether it is better than Alpha-Beta or not seems to depend on how good the evaluation function is[16], although it can be used to prune the space tree and thus improve the search efficiency[6].

4. Experimental Settings

4.1. Data

Most of this project was performed using games from the Lichess Elite Database[3], comprised of 3,819,130 games played on Lichess by players rated above 2200 (which represents the top 1% of human players on Lichess) between September 2013 and May 2020. 10% of them were used to generate validation data. Additionally, 93,103 games played by engines in the Computer Chess Raking List of 2020[4] were used to test the move prediction performance of the model at the highest level.

During the initial training phase, the network is trained for 150,000 steps, each of them comprised of 100 positions randomly selected in the dataset. As suggested in [12], positions occurring during the first 10 moves of a game are avoided.

4.2. Training Parameters

The network was trained using Binary Cross Entropy loss and Adam optimizer on an Asus ROG Zephyrus-M-GM501, using an Intel Core i7-8750H CPU cadenced at 2.20 GHz and a Nvidia Geforce GTX 1070.

During the move matching phase, the loss is computed only on legal moves, as opposed to all predictable moves. While this intuitively limits the ability of the network to generalize a move to similar positions, computing the loss on all moves implies that moves rarely seen but with a high importance, such a promoting a pawn, will not be ignored

[3] https://database.nikonoel.fr/

[4] http://ccrl.chessdom.com/ccrl/4040/games.html

by the network; experiments showed that computing the loss only on legal moves led to a consistent 2-point increase in accuracy for a variety of board representations and network architectures.

4.3. Self-Play Reward Attribution

During the self-play phase, the network plays a game either against himself, or against a previous version of himself, storing the network output for each position. At the end of the game, a label is created for each prediction p and position t as follows:

$$y(t,i) = \begin{cases} r & \text{if move } i \text{ was selected} \\ p(t,i) & \text{else.} \end{cases}$$

The self-play starts from 100,000 positions extracted from games with the only condition that the game does not end in a draw (50,000 were won by white and 50,000 by black). This allows for adapting the reward signal when the outcome reached through self-play is different from the actual result of the game, thus taking into account the advantage gained before the starting position in the reward. If the game is not finished after too many moves have been played, the game is aborted automatically. The self-play reward is the same for all moves played by a color c and depends on the winning human color c' and the number of moves n_t available in each position:

$$r(t,i) = \begin{cases} 1 & \text{if } c \text{ won} \\ 1 & \text{if } c \neq c' \text{ and draw} \\ 0 & \text{if } c = c' \text{ and } c \text{ lost} \\ 1/n_t & \text{if } c = c' \text{ and draw} \\ 1/n_t & \text{if game is aborted} \end{cases}$$

If the color losing is the same as with human players ($c \neq c'$ and c lost), the moves played by this color are ignored. Additionally, all human moves played between the beginning of the game and the self-play starting position are considered as the best moves and rewarded as such.

This strategy was also tried while ignoring the human results, using the following rewards:

$$r(t,i) = \begin{cases} 1 & \text{if won} \\ 1/n_t & \text{if draw} \\ 0 & \text{if lost or aborted} \end{cases}$$

While I believe this approach can also work, after 20,000 training games this model lost consistently against the one using human results during training. As time was the biggest constraint, this approach was therefore given up.

5. Results

5.1. Move-matching performance

During the supervised learning phase, we try to match the moves made by human in a given position. Different architectures were tried:

1. A network consisting only of 4 dense layers;
2. The network proposed in figure 2;
3. A similar network, but without the residual connections;
4. A similar network with a smaller pooling layer;
5. The proposed network, but computing the loss on all moves instead of considering only the legal moves;
6. Maia[12], given as a reference.

Method	Accuracy
Dense layers	0,24
Ours	**0,38**
Convolution Layers	0,34
Ours + pooling	0,36
Ours + Loss on all	0,36
Maia	0,35

Table 1. Accuracy of different networks after 90k steps.

The different networks have been trained for 90k steps (with batches of 100 positions). Maia has been included because it has been introduced as explicitly attempting to match human performance. It is worth noting that while it has a lower score than the proposed model, it is likely due to the short training: it indeed consists of many more weights than our network and would be the one benefiting the most from longer training.

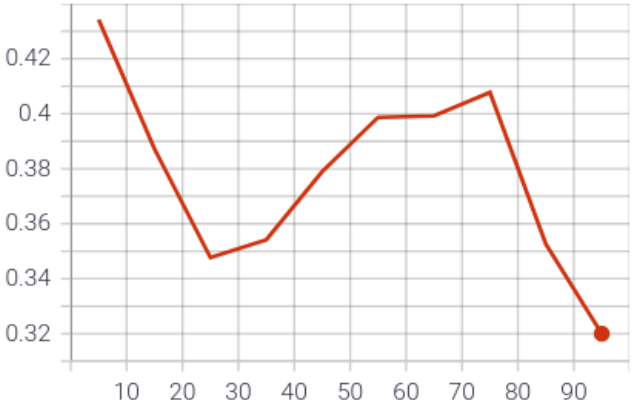

Figure 5. Move-matching accuracy relative to game progression.

Figure 5 gives a sense of how the model fares in different stages of the game. Despite having never been trained on opening moves, this is how it is most capable of mimicking humans, showing a capacity of generalization. However, it

does not seem to replicate well moves performed just after the opening. This can be justified by the sheer diversity of human players: since this is the moment where the number of pieces and possible moves is the highest, it is likely that the training time is simply too short to replicate this part well, which is why I decided during the self-play phase to use moves played between the beginning of the game and the self-play starting position as additional supervised training data.

5.2. Reinforcement Learning

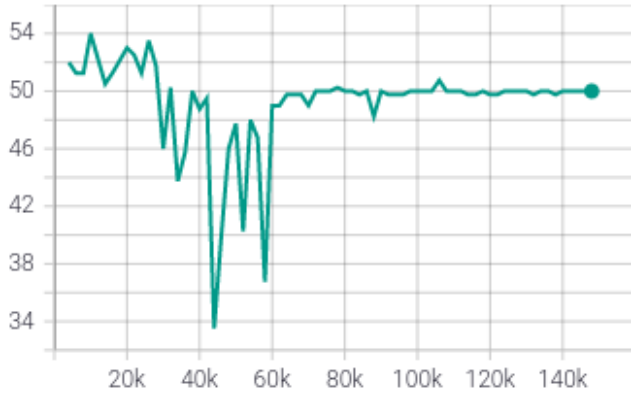

Figure 6. Score against the previous iteration when playing as black.

The question of whether White has an advantage in chess has been opened since at least 1889. While statistical analysis seems to support this idea[18], it is still debated both by game theorists and psychologists.

In the case of our model, however, it seems that Black is much easier to learn than White (winrate during training of both colors are shown in figures 6 and 7), perhaps because at low level reacting is easier than taking the initiative. After 30,000 games though, this advantage disappears as White improves, and both players stabilize with around 50% winrate after 70,000 matches.

However, this not because the winrate does not improve anymore that the model converged. Indeed, because playing a match until the end takes time (one to four seconds per iteration depending on the length of the game), I decided to end the game in a draw if it lasts longer than 350 moves (the average length game is usually around 80 moves).

5.3. Playing along another engine

In order to see if the model was actually better than a basic search-based engine, I performed a tournament between three different engines:

1. AB: a simple engine using an alpha-beta search of depth 4. Its evaluation function uses piece values and piece position as features.

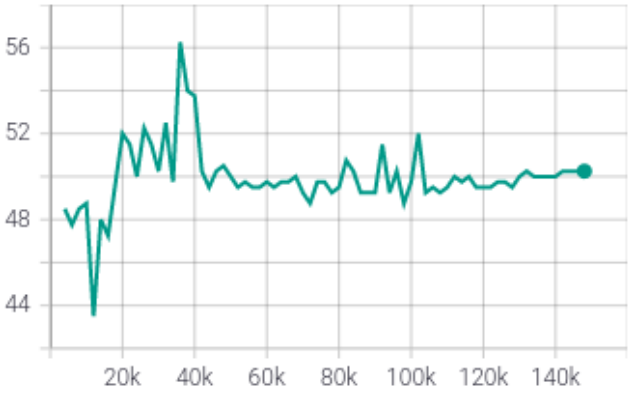

Figure 7. Score against the previous iteration when playing as white.

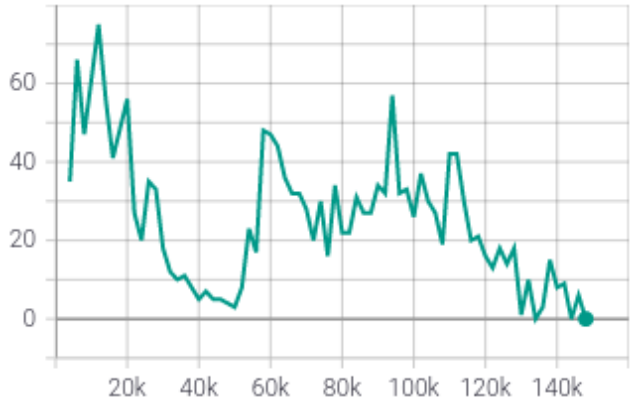

Figure 8. Number of aborted games during evaluation.

2. Ours: The proposed model, choosing between its two best moves according to their respective probabilities

3. AB with prob: A mix of the two, using the model output to guide the search. This engine is still limited to a maximum depth of 8, but with a probability threshold of 0.001.

Models played against each other for 100 matches, 50 as white and 50 as black.

-	AB	Ours
AB	-	32
Ours	68	-
AB with prob	73.5	57

Table 2. Winrate over 100 matches. A draw counts as 0.5 points for both players.

Against our model, the search-based engine achieved only 24 victories and 16 draws out of a 100 matches, indicating that our model is around 130 Elo stronger. When adding the probability-based search, the difference is about

175 Elo. This gain is also reflected against the model without search, with 47 victories and 20 draws in favor of the probability-based engine.

5.4. MCTS

The biggest improvement however was brought by using MCTS. Table 3 sums up the result achieved with different number of games played using each legal move in a given position, against the same model without search.

-	Ours
MCTS 3	71
MCTS 5	76
MCTS 10	77
MCTS 40	78

Table 3. Winrate over 100 matches. A draw counts as 0.5 points for both players.

The gains achieved are between 156 and 220 Elo, for 3 and 40 searches respectively. In addition, it is way faster than the Alpha-Beta and probability search, with around 4 seconds per game played against a minute per position for the previous ones.

Because the time taken is linear relative to the number of simulations, it is also way easier to influence the search time compared with Alpha-Beta and Probabilistic searches, where the number of evaluated positions is dependant on move-ordering and depth threshold in a non-linear way.

6. Possible improvements

6.1. Use MCTS for training

During the reinforcement learning phase, the network plays games against itself and is updated according to the result of the game. This is a very similar behaviour to the one of MCTS, except that the later plays thousands of simulations in a single game. Therefore, it could be interesting to try using the result of a Monte-Carlo tree search to update the network, as it aggregates much more game experience than a single normal game and can be used to update predictions for all legal moves, instead of the one played. However, I am not sure this would be an improvement in terms of total training time and chose to pursue different approaches in this project.

A second problem is that the result of the search is not a probability distribution, but rather a confidence value that each move is the good one, this can not be related directly to the policy network. Although there are ways to artificially make it a probability distribution, this would require an amount of experiments that I did not have the time to perform.

6.2. Time Control

Used in its current state, the model performs equally well in any time control: indeed, since it does not perform a search, the computation is done in a single pass faster than any human player. However, if I want my model to compete with top engines, I need to implement some kind of search: although I could not find relevant values for chess, AlphaGo Zero gains more than 2000 Elo by using MCTS (and a significantly better hardware) compared to the raw output of the net[15]. To put it in perspective, the winrate vs Elo difference curve follows a logistic distribution, leading to a 0.9998 win probability for a 1000 point difference, and AlphaGo Zero's rating without search would put it in the top-500 of human players.

Therefore, implementing a search is essential to reach a competitive level. However, this takes time to compute, which is limited in chess and was DeepChess' main drawback[8]. There are some strategies to implement a time management feature in engines[1], but it seems to be a much less active research field than move-selection algorithms. An effective time-management strategy can also improve the algorithm design phase by limiting time spend on testing: on my hardware, playing a game without search takes up to 4 seconds, while the alpha-beta engine use up to 8 seconds to select one move, which has to be done 40 times per game and per color in average.

6.3. Dying ReLU and training speed

Experiments during the move-matching phase indicated that a learning rate of 0.001 seemed optimal for learning, as shown in figure 9.

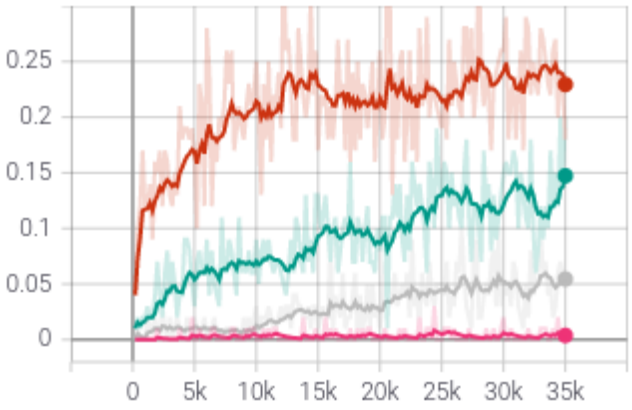

Figure 9. Training accuracy with different learning rates. From top to bottom: 10^{-3}, 10^{-4}, 10^{-5}, 10^{-2}.

However, during the reinforcement learning phase Dying ReLUs tend to happen quite a lot. I tried to mitigate this by resetting the weights of dead neurons, but this measure seemed to hinder the learning process to much. As this

problem happened with an already pretrained network at a late point of my research, trying different countermeasures would prove too long and I had to decrease the learning rate, which was not satisfying either.

In a future version, I would like to test different measures against this problem, such as different weight initialization or modified activation functions like Leaky ReLU. While this is not related to Chess engines anymore, I believe this knowledge is necessary when working with long training times (the reinforcement learning phase presented in this paper took 70 hours).

7. Conclusion

In this work, I propose a new framework for training a chess engine. Inspired both by AlphaZero's and Leela's performance of learning chess only through self-play, and by Maia's human-like way of playing, I designed a network that can be trained both via supervised and reinforcement learning. The proposed model does not learn an evaluation function, but only a policy dictating which move to choose.

In addition to outperforming a search-based engine, I showed how the model can be used to enhance any basic engine by using either a probability-based search or a Monte-Carlo tree search instead of the classic depth-limited search, leading to an improvement of up to $+220$ Elo. Additionally, the performances achieved by using MCTS are good enough for all kinds of time control parameters, which makes it a good training partner for a human in all situations.

References

[1] Hendrik Baier and Mark Winands. Time management for monte carlo tree search. *IEEE Transactions on Computational Intelligence and AI in Games*, 8:301–314, 09 2016.

[2] Michael Buro. Statistical feature combination for the evaluation of game positions. *J. Artif. Int. Res.*, 3(1):373–382, Dec. 1995.

[3] B. Bŏkovíc. Tuning chess evaluation function parameters using differential evolution algorithm. 35:283–284, 01 2011.

[4] Murray Campbell, A.Joseph Hoane, and Feng hsiung Hsu. Deep blue. *Artificial Intelligence*, 134(1):57–83, 2002.

[5] Tristan Cazenave. Residual networks for computer go. *IEEE Transactions on Games*, 10(1):107–110, 2018.

[6] Yen-Chi Chen, Chih-Hung Chen, and Shun-Shii Lin. Exact-win strategy for overcoming alphazero. In *Proceedings of the 2018 International Conference on Computational Intelligence and Intelligent Systems*, CIIS 2018, page 26–31, New York, NY, USA, 2018. Association for Computing Machinery.

[7] Eli David, H. Herik, Moshe Koppel, and Nathan Netanyahu. Genetic algorithms for evolving computer chess programs. *IEEE Transactions on Evolutionary Computation*, 18, 10 2014.

[8] 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.

[9] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016.

[10] Matthew Lai. Giraffe: Using deep reinforcement learning to play chess, 2015.

[11] Henk Mannen. Learning to play chess using reinforcement learning with database games, 2003.

[12] Reid McIlroy-Young, Siddhartha Sen, Jon Kleinberg, and Ashton Anderson. Aligning superhuman ai with human behavior: Chess as a model system. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD '20, page 1677–1687, New York, NY, USA, 2020. Association for Computing Machinery.

[13] David Silver, Aja Huang, Christopher Maddison, Arthur Guez, Laurent Sifre, George 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*, 529:484–489, 01 2016.

[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.

[15] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy P. Lillicrap, Fan Hui, L. Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of go without human knowledge. *Nature*, 550:354–359, 2017.

[16] Shogo Takeuchi, Tomoyuki Kaneko, and Kazunori Yamaguchi. Evaluation of monte carlo tree search and the application to go. pages 191 – 198, 01 2009.

[17] Shogo Takeuchi, Tomoyuki Kaneko, Kazunori Yamaguchi, and Satoru Kawai. Visualization and adjustment of evaluation functions based on evaluation values and win probability. pages 858–863, 01 2007.

[18] Haroldo Valentin Ribeiro, Renio Mendes, Ervin Lenzi, Marcelo del Castillo, and Luís Amaral. Move-by-move dynamics of the advantage in chess matches reveals population-level learning of the game. *PloS one*, 8:e54165, 01 2013.

[19] Joel Veness, David Silver, Alan Blair, and William Uther. Bootstrapping from game tree search. In Y. Bengio, D. Schuurmans, J. Lafferty, C. Williams, and A. Culotta, editors, *Advances in Neural Information Processing Systems*, volume 22. Curran Associates, Inc., 2009.

[20] V. Zakharov, M. G. Mal'kovskii, and A. I. Mostyaev. On solving the problem of 7-piece chess endgames. *Programming and Computer Software*, 45:96–98, 2019.