LazyBug - A Neural Bughouse Engine

Praktikum KI - Gruppe 1

M. Heinrich, T.-K. Nguyen, V. Pham, T. Schneider, S. Thiem

15. September 2019

1 Introduction

In modern society, engines are capable of playing many of the classic board games like chess [1], checkers [2] or go [3] with superhuman strength. In 2016, AlphaGo became the first engine to beat the world champion of Go using supervised learning and deep reinforcement learning[4]. Later, AlphaZero[5] was developed as an adaptation of AlphaGo, relying solely on reinforcement learning and capable of learning a variety of different games. One of the games learned by AlphaZero is chess, where it has been shown to be capable of beating the 2016 TECE world champion Stockfish [5]. This motivated further approaches to beat more dynamic variants of chess like Crazyhouse [6]. In this work we present LazyBug - an engine for the Bughouse chess variant, based on supervised learning and deep reinforcement learning.

The game of Bughouse is played in teams of two on two boards, such that each team has one player on each board. Within each team, one player has the white pieces and the other player the black pieces. Whenever a piece is taken, it is transferred into the "pocket" of the partner of the player who captured it. Similar to Crazyhouse, instead of moving a piece any player can always "drop" a piece out of his pocket on any empty square on the board. Bughouse is played asynchronously on both boards, that is, the players of each board take turns but are never required to wait for the other board (unless they cannot make a feasible move and need material). The game ends if any player runs out of time or any player is checkmated by Bughouse rules. For further details of the Bughouse rules, please refer to the rule reference used for this work [7].

Compared to regular chess, Bughouse introduces a variety of new challenges for engines to handle. Firstly, due to the drops, the number of possible moves every turn increases significantly, making policy learning substantially harder. Furthermore, Bughouse is played by four agents instead of two. Hence, not only the opponents moves need to be predicted but also the moves of the players partner and his opponent, which makes predicting the game's outcome a hard task. The fact that the game is played asynchronously not only increases the prediction uncertainty even more, but also makes time management more complex, since waiting might be the best move at some point in the game. Finally, communication with the partner is a very important component of Bughouse, which might decide over victory and defeat if used properly. However, communicating with a (potentially human) partner is a non-trivial task and has not been used to its full potential by the existing Bughouse engines we analyzed [8, 9].

2 Policy and Value Function

Similar to AlphaZero[5], we use a single convolutional neural network (CNN) to represent both the policy $p_\sigma(a|s)$ and the value function $v_\sigma(s)$ (see subsection 2.2). As further described in subsection 2.3, the network is trained in supervised learning fashion on a large data set of human games.

2.1 Data Representation

Prior work[10] has shown that Neural Networks perform better when data is provided as a one hot vector instead of a number. Encoding the policy output as one hot allows us to learn a probability distribution over all possible moves instead of just a mapping from the board state to a single move. Hence, our moves are encoded as 87 planes of 8x8 chess boards, where each move is represented as

exactly one 1 anywhere in the tensor. As proposed by AlphaZero[5] we use a relative encoding for all possible moves by taking the start square and the direction and length (if applicable) of the move into account. The start square is always encoded in the first two dimensions of the move tensor, while the last dimension (or the plane number) gives information about the type of move. Starting with potential queen moves, there are 8 move directions with the highest possible move length being 7, we therefore require 56 planes. The next 8 planes represent all possible knight moves, followed by 9 planes for pawn underpromotions. Every pawn can either walk straight to be promoted as well as capturing a figure in an angle forward. Thus there are 3 planes for promotions to knights, 3 for bishop and 3 for rooks for each color, thus totalling 18. Promotions to queens get handled implicitly and are just represented as normal pawn move into the last or first rank, respectively. To handle drops, one additional plane for every type of figure except the king is needed. Since drops do not have a starting square, the first two dimensions are used to encode the target square of the drop.

The game state is encoded as two 14x8x8 tensors (one for each board) and a vector containing scalar information about the game, like the pockets, the clocks, castling rights and whose turn it is. The tensors each consist of one plane for each figure type of each color, having a 1 at each square occupied by the respective piece type and a 0 elsewhere. As we are using a CNN architecture, these tensors are concatenated on their last axis before being passed into the network. Additionally we have a plane indicating which pawns are currently allowed to take en passants and one plane indicating which pieces are promoted pawns. The en passant capture moves and the promoted pawns could be represented by a slim representation, however we chose here to use the entire board to make the convolutions work similar to regular pawn capture moves. For the pockets we use a 2x5 input with a 1 dimensional input for each figure containing the number of pieces a player has, normalized to a value between 0 and 1 with 1 being the maximum number possible. Considering pawns, 1 would be 16 pawns available and 0 none. The clocks are scaled to minutes before being passed into the network. Castling is represented by 2 bits for each color, 1 if a player can castle left or right respectively. To make training simpler, we mirror the board, such that the moving player is always on top. While it is also an option to simply rotate the board, mirroring yields the advantage that it makes the starting positions of black and white appear equal.

2.2 Neural Network Architecture

Similar to AlphaZero [5], we use a single neural network with a policy head and value head. Our first layer is a mixture of a convolutional layer and a dense layer to process the spatial and scalar data into an 8x16 layer with depth 256. Each filter of the first layer has one additional non-spatial parameter for each scalar input value. The sum of these scalar values, weighted by their respective parameters, is added to the filter output before the activation function. This layer is followed by several residual blocks which contain convolutional layers, batch normalization, and ReLU activations accompanied by an averaging of the skip connection and the output. Currently we are using 13 residual blocks, but the number of blocks can be configured by a hyperparameter.

The policy header is a convolutional layer with 87 filters that represent each possible action as a one-hot-vector. Here we use a softmax activation function to get the probabilities of each move to played next (see subsection 2.1). The value head consists of a convolutional layer with a Kernel of 1x1 and one filter, which is then flattened. This is followed by a dense layer of 256 Neurons followed by a dense layer with a tangent hyperbolic activation function for the value. Output of the value head is number between -1 and 1, where 1 indicates a victory for the own team and -1 a defeat, respectively. Unless otherwise mentioned we use a kernel of 3x3, a stride of one, zero padding, no dropout and ReLU activation functions.

2.3 Supervised Learning

As stated before, the neural network is trained in supervised learning fashion on human games. Hence, the policy head is trained to predict the next move played given a board position and the value head is trained to predict if the moving player's team is winning (1) or the opposing team is winning (-1). Both heads are trained jointly, with the loss function being the sum of a cross-entropy loss for the policy head and mean-squared-error loss for the

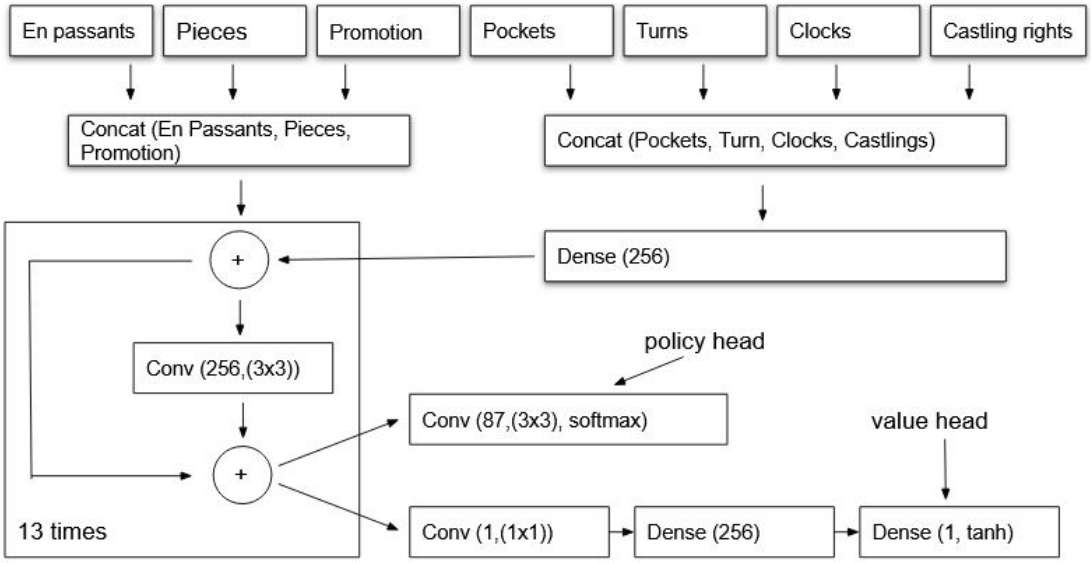

Figure 1: The architecture of the neural network. The number in the dense layer represents its number of neurons. In a convolutional layer it represents the amount of 8x8 boards, followed by its Kernel Size.

value head. To ensure that the losses are balanced, the policy loss is multiplied with a factor of 0.01. Currently, we use a batch size of 256 and a static learning rate of 0.02, but these hyperparameters are likely to change in the future. We obtain our data from a large database of Bughouse games[11], which were played on `freechess.org` [12]. The database contains over 3.7 million games, comprised of roughly 380 million positions and is thus sufficiently large to train a reasonably sized network as ours. To ensure the quality of our data, we only consider games played by the top 10% of the players in terms of their elo rating.

3 Search Algorithm

To evaluate the quality of a move in a given position, we use an adapted version of a Monte Carlo Tree Search (MCTS) variant developed in prior work [5]. MCTS is a heuristic search technique, which we use to evaluate the quality of moves in a given board state. Instead of traversing the entire game tree from our current position, like Minimax does, MCTS traverses only a small selection of interesting moves. If these moves are well chosen, this gives an approximate idea of how well a move will perform. To find interesting moves, we query the neural network described in section 2, which is also used to evaluate positions at the bottom of the search tree. Since moves on the other board can greatly affect the position on the own board, it is necessary to consider these when evaluating potential own moves. Thus, we developed a MCTS variant that simulates moves on both boards, picking the player to move next at random.

The results of these simulations are stored in a tree structure, where nodes represent board states and edges represent moves. This tree is initialized containing only a root node, which corresponds to the current board state. Each simulation is composed of four steps: selection, expansion, evaluation and backpropagation, on which we elaborate in the following.

Selection Starting from the root node, MCTS utilizes the results from the policy and value network to select an action a to take from the respective game state s. In the following, we denote the policy output from the perspective of player p as $\pi_p(a|s)$ and the value as $v_p(s)$, respectively. To simplify the notation we denote the associated player of an action a by $p(a)$. The game state consists of the state of both boards and that the value function always determines the value of the team and not of a single player. Each edge (s, a) contains an action value $Q(s, a)$, visit count $N(s, a)$ and prior proba-

bility $P(s,a)$, which are defined as follows:

$$Q_p(s,a) = \frac{1}{N(s,a)} \sum_{i=1}^{n} 1(s,a,i) v_p(s_L^i)$$
$$N(s,a) = \sum_{i=1}^{n} 1(s,a,i)$$
$$P_p(s,a) = \pi_p(a|s)$$

After selecting the player p to move next, MCTS picks the edge with the highest value of $Q_p(s,a)$ plus a bonus $u_p(s,a)$:

$$u_p(s,a) \propto \frac{C(s)\sqrt{N(s)}P_p(s,a)}{1+N(s,a)}$$

where $N(s)$ is the total visit count of state s and $C(s)$ is the exploration rate, given by

$$C(s) = c_{init} + \log\left(\frac{1+N(s)+c_{base}}{c_{base}}\right) + c_{init}$$

During the search $C(s)$ is essentially constant but slowly increasing, encouraging exploration over exploitation. As we can see, $u_p(s,a)$ is inversely proportional with the visit count $N(s,a)$, therefore encourages diversity in selecting moves, while the action value $Q_p(s,a)$ supports exploitation of good moves. This procedure is repeated until an edge (s_P, a_P), that has not been visited before, has been selected or a node that ends the game decisively (e.g. win/loss/draw) has been reached. In the latter case, we skip the expansion phase and continue with the evaluation with $s_L = s_P$.

Additionally to the moves proposed by the policy, a "sitting" move is added with a constant probability of 0.01. The semantic of this move is that the player decides to wait for the other board to move before making an own move. Thus, the value of this move is the value of the best move of the other player. Note, that the other player also has the option to sit instead of move, in which case the player with the lowest clock loses. Hence, for a rational agent, sitting is only an option if the partner is on the move and not currently sitting or the opponent is on the move and has a lower clock. One possible scenario in which the "sitting" move might be utilized is when the partner's opponent is low on time and one move from being checkmated. Another common scenario is when the partner is on the move and is capable of capturing a piece that can be utilized to checkmate the own opponent.

Expansion In the expansion phase, we expand s_P with a child node s_L. The child node s_L, being a new leaf node, is processed once by the supervised-learning policy network p_σ, outputting prior probabilities $P(s,a)$ for all edges (s_L, a) from s_L. Each of the edges is initialized as follows:

$$P_p(s,a) = \pi_p(s|a)$$
$$Q_p(s,a) = -1$$
$$N(s,a) = 0$$

Evaluation In the evaluation phase, the new leaf node s_L is evaluated by the neural network. The resulting value output is stored in $v_p(s_L)$ and the policy outputs for each move in $\pi_p(a|s_L)$.

Backpropagation After the evaluation, the action value $Q_p(s,a)$ is updated with $v_p(s_L)$ and the visit count $N(s,a)$ is incremented for all the traversed edges.

Choosing the best move After the final evaluation has been backpropagated, the move with the highest visit count is selected and played on the board. If considerably more moves have been visited on the other board than on the own board or a "sitting" move has the most visits, we wait for the board to change instead of making a move. In that case "sitting" is communicated to the partner together with an explanation of the reason. This explanation usually contains a suggestion of a move that is better than all moves LazyBug could play in this position.

4 Time Management

In regular chess, time management consists solely on the question of how much time shall be allocated to think about the next move. Bughouse chess on the other hand adds a new level of complexity to this problem, as the clocks of each player might influence their decision making directly. For example, if the partner's opponent is low on time, it might be beneficial to play safe moves since your partner is likely going to win his board on time anyway. Thus, we broke down the problem of time management into time allocation and time-aware decision making.

4.1 Time allocation

While using a constant time to think about each move is the most straightforward way to tackle this problem, it comes with two mayor drawbacks. Firstly, on easy positions where the choice of the next move is clear, a lot of time might be wasted unnecessarily reconsidering an obvious choice. Secondly, on difficult positions with a lot of possible options, the time might be to short to consider all of them properly, leading to poor move choices. To deal with these problems, we developed a dynamic move time allocation, where after a minimum search time we continue generating nodes until one move has been visited in at least 40% simulations. This enables LazyBug to allocate more time on hard positions and waste less time on easy positions.

4.2 Time-aware decision making

To take the clocks into account when generating move proposals, we feed the clocks into the neural network. However, during the MCTS simulations, we do not have the exact clocks, as we do not know how long the players would take to move in the respective positions. Hence, the move times during MCTS simulations need to be estimated. While one option is to simply learn the move times from the human generated data, we decided that this might lead to problems, since engines approach chess differently than human players, leading to imprecise time estimations. Instead we simply estimate the move times using the average of the previous 5 moves of each player. To avoid a horizon effect, we do not consider a player that ran out of time in simulation to lose the game. Instead the simulated players move faster as they approach a clock of zero.

5 Implementation

Our implementation is purely python based and using `keras` [13] with `tensorflow` [14] as backend. For the move generation, our team developed a fork [15] of `python-chess` [16], which also came to use in the chess server [17] of the tournaments. While python is great for prototyping, it is suboptimal in terms of efficiency. To use the GPU as efficiently as possible, we select and buffer 16 MCTS-nodes and evaluate them at once. Additionally, we utilize multithreading to continue collecting nodes while the GPU is evaluating the previous set of nodes. Using these two techniques, we were able to reduce the time consumption of the node selection enough to make the network evaluation performed by `keras` the bottleneck, rendering any further optimization useless. On average, we achive a rate of approximately 300 nodes per second. LazyBug's code can be found on github [18].

6 Evaluation

Figure 2 shows the accuracy of the winner prediction over the course of the optimization. Unsurprisingly, the accuracy increases significantly when drawing closer to a mate. However, as these mates are not necessarily forced, it is impossible to reach 100% here. Parts of the error might also come from the fact that even forced mates can be evaded by sitting or if the partner mates his opponent quicker. Figure 3 draws a similar picture for the mean absolute error of the value prediction. While a mean absolute error of 0.87 seems fairly bad at the first glance, it is important to note that the data also includes many early game positions, in which it is nearly impossible to predict the winner. Hence, the data sets containing positions close to a mate draw a much clearer picture on the actual accuracy of the value network. In Figure 4 we see the accuracy of the policy predicting the move that was actually played in the presented position.

We evaluated our model against conventional engines (Figure 6) and in multiple tournaments against the other groups (Figure 6). However, the results of the tournaments are not very conclusive as all teams reported significant performance issues on the tournament servers. Thus, instead of traversing the MCTS search tree we were forced to play using only the policy and always select the move with the highest probability. When we used our MCTS on a local machine[1] we are able to beat our standalone policy about 70% of the time. The fact that the standalone policy won against our MCTS in 30% of all cases indicates that much of the playing strength comes from the policy and the value function seems to be inaccurate in certain cases.

As visible in Figure 6, the conventional engines we evaluated LazyBug against were Sjeng [8] and Sunsetter [9]. Both use variants of quiescence search to evaluate the board position and potential

[1] Intel i7 and NVIDIA GTX 1070

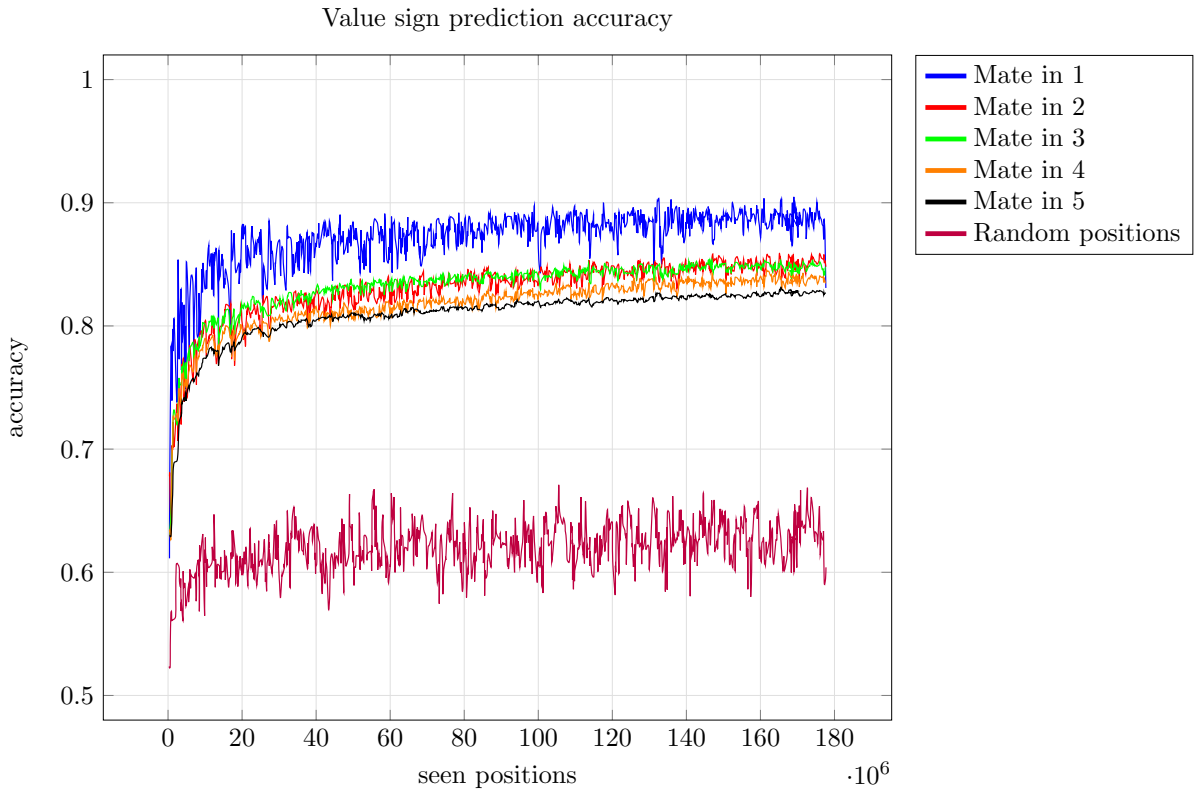

Figure 2: Winner prediction accuracy on different validation data sets over the course of the optimization. The validation sets consist of positions taken from real human games. A prediction is considered correct if the sign of the true game value was predicted correctly. As the name suggests, the first 5 data sets contain only positions in which one player was mated within 1 to 5 moves.

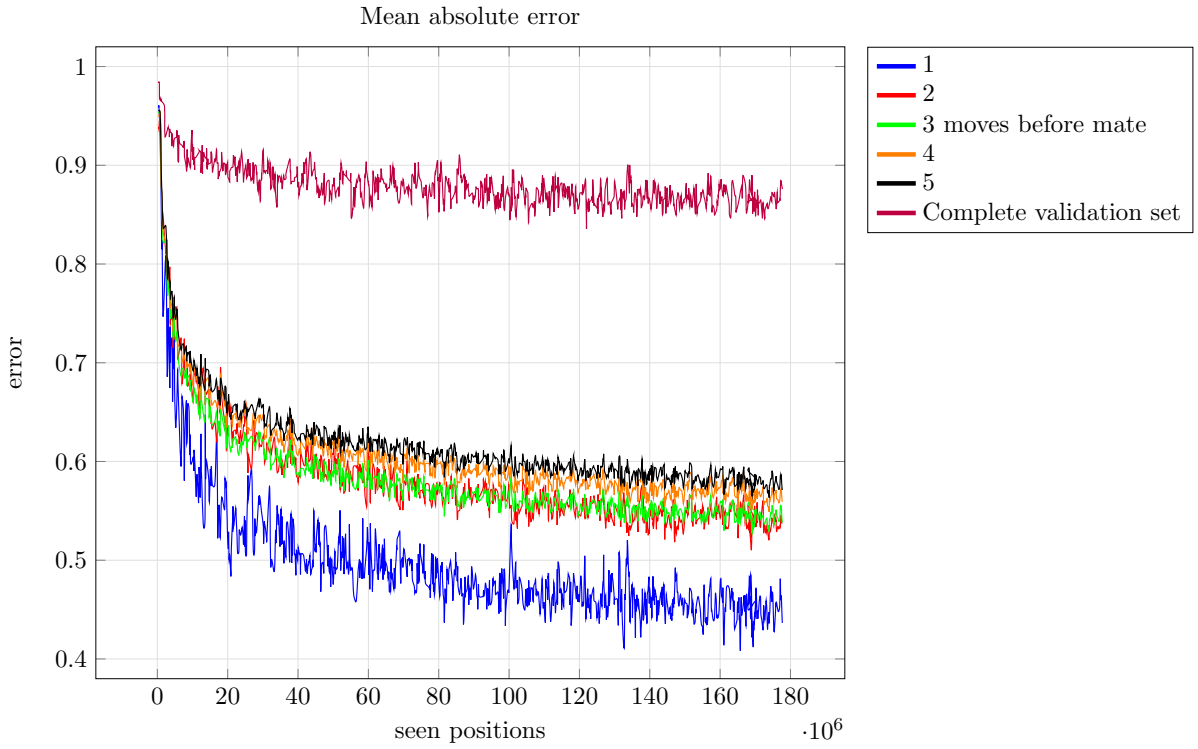

Figure 3: Mean absolute error of the value function on different validation data sets over the course of the optimization.

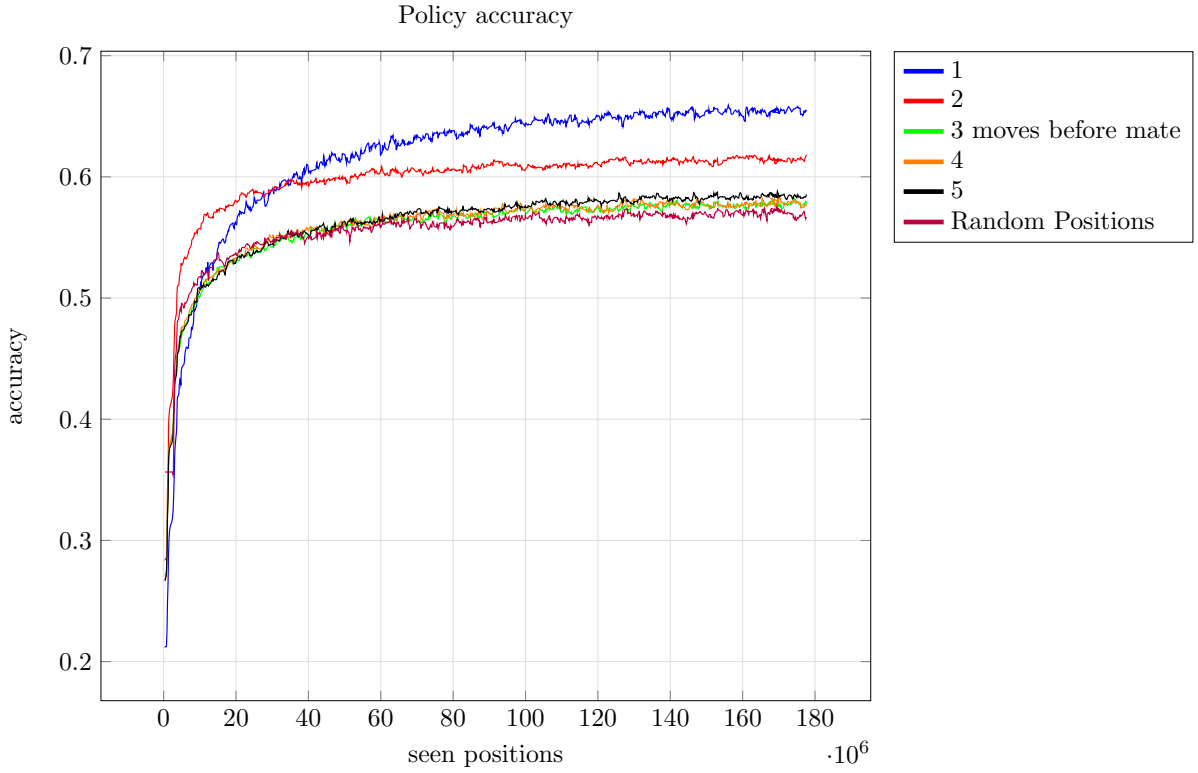

Figure 4: Accuracy of the move prediction on different validation data sets over the course of the optimization.

Opponent	Wins	Draws	Losses
Sjeng	47	0	3
Sunsetter	27	1	22
Standalone Policy	33	0	17

Figure 5: Results of LazyBug against Sjeng, Sunsetter and the Standalone Policy. Time control was 5 minutes without increment.

Group	Wins	Draws	Losses
1	157	12	36
2	58	12	139
3	143	12	49
4	34	12	168

Figure 6: Results of the latest tournament on 25th of August 2019. The groups played 5 games in each possible configuration with a time control of 2 minutes without increment. LazyBug is group 1.

moves. To the best of our knowledge, Sunsetter is the strongest open source Bughouse engine while Sjeng seems to have a lower playing strength. While we were able to beat Sjeng quite consistently, Sunsetter currently seems to be an even match for LazyBug. The interested reader is referred to our GitHub repository [18] for the .bpgn files off all our evaluation games played.

7 Conclusion

In our work we developed a neural network engine, capable of playing Bughouse chess. To overcome many of the challenges Bughouse introduces over regular chess, we develop an adaptation of a MCTS variant proposed in prior work[3, 5]. We show that our engine is capable of winning against conventional state-of-the-art Bughouse engines like Sjeng [8] and Sunsetter [9].

However, as discussed in section 6, a lot of the playing strength seems to come from the move proposals generated by the neural network policy. This indicates that there is probably potential for improvement in the value function. A reason for this could be the quality of the human generated data, which is likely to contain blunders and misplays, adding high variance to the result of each game given a position. This problem could

be tackled in the future by filtering the data more strictly or utilizing self play similar to Silver et al. [3] to generate high quality games.

Additionally, the means of communication with our engine are currently very limited. This is due to the fact that we found learning communication from the data to be a highly non-trivial task. One option to include a richer communication in LazyBug is to define a set of commands (e.g. asking for pieces or not to move) and reward fulfilling these tasks in the MCTS search. However, this would introduce a huge set of new hyperparameters, since the correlation between the value of a position and the value of fulfilling a command needs to be defined in some way. Thus, this solution would require a huge amount of hand-engineering and seems not very clean from a machine learning perspective. Another option is to use reinforcement learning and learn to use communication, for example by playing with another engine with hardcoded commands. Yet this would require large amounts of computation power and time, since reinforcement learning is usually not very data efficient and was thus not done in this work.

References

[1] Murray Campbell, A Joseph Hoane Jr, and Feng-hsiung Hsu. Deep blue. *Artificial intelligence*, 134(1-2):57–83, 2002.

[2] Jonathan Schaeffer, Neil Burch, Yngvi Björnsson, Akihiro Kishimoto, Martin Müller, Robert Lake, Paul Lu, and Steve Sutphen. Checkers is solved. *science*, 317(5844): 1518–1522, 2007.

[3] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815*, 2017.

[4] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–486, 2016.

[5] 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`.

[6] Alena Beyer Johannes Czech, Moritz Willig. Crazyara - deep learning for crazyhouse, (accessed on 12.06.2019). `https://github.com/QueensGambit/CrazyAra`.

[7] Laws of bughouse chess, (accessed on 15.06.2019). `http://bughousechess.wz.cz/CompleteBughouseChessRules.htm`.

[8] Sjeng : a chess-and-variants playing program, (accessed on 15.06.2019). `https://sjeng.org/indexold.html`.

[9] Sunsetter, (accessed on 15.06.2019). `http://sunsetter.sourceforge.net/`.

[10] Kedar Potdar, Taher Pardawala, and Chinmay Pai. A comparative study of categorical variable encoding techniques for neural network classifiers. *International Journal of Computer Applications*, 175:7–9, 10 2017. doi: 10.5120/ijca2017915495.

[11] Fics bughouse database, (accessed on 13.06.2019). `https://www.bughouse-db.org/`.

[12] Free internet chess server, (accessed on 13.06.2019). `https://www.freechess.org/`.

[13] Keras: The python deep learning library, (accessed on 15.09.2019). `https://keras.io/`.

[14] Tensorflow: An end-to-end open source machine learning platform, (accessed on 15.09.2019). `https://www.tensorflow.org/`.

[15] Fork of python chess, (accessed on 13.06.2019). `https://github.com/TimSchneider42/python-chess`.

[16] Python chess, (accessed on 13.06.2019). `https://github.com/niklasf/python-chess`.

[17] tinychessserver, (accessed on 15.09.2019). https://github.com/MoritzWillig/tinyChessServer.

[18] Lazybug - a neural bughouse engine, (accessed on 15.09.2019). https://github.com/TimSchneider42/lazybug.