Assessing Popular Chess Variants Using Deep Reinforcement Learning

Analyse beliebter Schach-Varianten mittels Deep Reinforcement Learning
Master thesis by Maximilian Alexander Gehrke
Date of submission: July 15, 2021

1. Review: Prof. Dr. Kristian Kersting
2. Review: Prof. Constantin A. Rothkopf, Ph.D.
3. Review: M.Sc. Johannes Czech
Darmstadt

TECHNISCHE UNIVERSITÄT DARMSTADT

Fachbereich Informatik

Artificial Intelligence & Machine Learning Lab

Erklärung zur Abschlussarbeit gemäß §22 Abs. 7 und §23 Abs. 7 APB der TU Darmstadt

Hiermit versichere ich, Maximilian Alexander Gehrke, die vorliegende Masterarbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Mir ist bekannt, dass im Fall eines Plagiats (§38 Abs. 2 APB) ein Täuschungsversuch vorliegt, der dazu führt, dass die Arbeit mit 5,0 bewertet und damit ein Prüfungsversuch verbraucht wird. Abschlussarbeiten dürfen nur einmal wiederholt werden.

Bei der abgegebenen Thesis stimmen die schriftliche und die zur Archivierung eingereichte elektronische Fassung gemäß §23 Abs. 7 APB überein.

Bei einer Thesis des Fachbereichs Architektur entspricht die eingereichte elektronische Fassung dem vorgestellten Modell und den vorgelegten Plänen.

Darmstadt, July 15, 2021

Maximilian Alexander Gehrke

Abstract

Over the recent years, reinforcement learning methods have successfully been applied to board games like Go, chess and shogi. With the help of Monte Carlo tree search, neural networks could be trained to surpass the playing strength of any human in all three games. In this thesis, we extend the application of Monte Carlo tree search and create an engine that can play the chess variants 3check, Antichess, Atomic, Chess960, Crazyhouse, Horde, King of the hill and Racing kings. For each variant, we train a separate neural network, first by using expert human data and afterwards by generating millions of self-play games during reinforcement learning.

To the best of our knowledge, Monte Carlo tree search has not been applied to 3check, Antichess, Horde, King of the hill and Racing kings yet. We show that it is possible to train several chess variants with the same reinforcement learning setup and produce the very first multi-variant chess engine that utilizes Monte Carlo tree search. Our program *MultiAra* has become the worlds strongest engine in the game of Horde and second strongest - behind it's sibling *CrazyAra* - in Crazyhouse. For all other variants, except Chess960, it is on par or only slightly behind the classical evaluation of the worlds strongest multi-variant engine *Fairy-Stockfish*.

Further, we investigate whether a model trained from zero knowledge outperforms the playing strength of a model that starts with a network trained on human games, and test whether adding last moves as attention mechanism to the input representation leads to a better model in Crazyhouse. Finally, we compare game outcomes during self-play with human games and propose several ideas on how to balance certain chess variants.

Keywords: Reinforcement Learning, Chess Variants, Deep Learning, Monte Carlo Tree Search, Training From Zero Knowledge, Attention Mechanism, Game Balance

Zusammenfassung

In den letzten Jahren wurden Reinforcement Learning Methoden erfolgreich auf Brettspiele wie Go, Schach und Shogi angewendet. Mit Hilfe von *Monte Carlo tree search* konnten neuronale Netze so trainiert werden, dass sie die Spielstärke eines Menschen in allen drei Spielen übertreffen. In dieser Thesis erweitern wir das Anwendungsgebiet von *Monte Carlo tree search* und erstellen eine Engine, welche die Schach-Varianten 3check, Antichess, Atomic, Chess960, Crazyhouse, Horde, King of the hill und Racing kings spielen kann. Für jede Variante trainieren wir ein separates neuronales Netzwerk, welches wir zunächst mit Hilfe von menschlichen Spielen trainieren und anschließend durch die Erzeugung von Millionen selbst gespielter Partien während des Reinforcement Learnings verfeinern.

Nach unserem aktuellen Kenntnisstand wurde *Monte Carlo tree search* noch nicht auf 3check, Antichess, Horde, King of the hill und Racing kings angewendet und wir kreieren in dieser Arbeit die allererste *Monte Carlo tree search* Engine, welche mehrere Schach-Varianten unterstützt. Unser Programm *MultiAra* ist weltweit die stärkste Engine in Horde und die zweitstärkste Engine - hinter seinem Zwilling *CrazyAra* - in Crazyhouse. In allen anderen Varianten, außer Chess960, ist *MultiAra* gleich stark oder nur geringfügig schwächer als die weltweit stärkste Schach-Varianten Engine *Fairy-Stockfish* mit klassischer Evaluierung.

Zudem untersuchen wir, ob ein Reinforcement Learning Modell, welches ohne jegliches Vorwissen trainiert wurde, die Spielstärke eines Modells übertrifft, welches mit einem auf menschlichen Spielen trainierten Netzwerk initialisiert wurde. Außerdem testen wir, ob das Hinzufügen der zuletzt gespielten Schachzüge als eine Art Aufmerksamkeitsmechanismus fungieren und die Spielstärke in der Variante Crazyhouse erhöhen kann. Darüber hinaus analysieren wir, wie sich Spiele, welche während des Reinforcement Learnings produziert wurden, von menschlichen Spielen unterscheiden, und schlagen mehrere Ideen vor, wie man bestimmte Schach-Varianten besser ausbalancieren kann.

Stichworte: Reinforcement Learning, Schach-Varianten, Deep Learning, Monte Carlo Tree Search, Training Ohne Vorwissen, Aufmerksamkeitsmechanismen, Spiel-Balance

Contents

1. Introduction

1.1. Motivation

In 2016, the reinforcement learning method Monte Carlo tree search (MCTS) has been combined with neural networks in the algorithm *AlphaGo* to create the most powerful engine for the game of Go [27]. Subsequently, many researchers replicated and applied this technique to other board games. However, most chess variants have not been extensively explored by MCTS yet, except Crazyhouse [7, 5].

This is unfortunate, because chess variants are becoming more and more popular. For example, the open source platform *lichess.org* features 8 different chess variants since the beginning of 2016, which increase in popularity every month[1]. It would be nice to fall back not only on minimax engines with alpha-beta pruning, but also on MCTS engines to play chess variants for fun, increase your own playing strength or analyze games. Both engine types can behave very differently in move selection, playing strength or playing style. However, also if both engine types advocate the same moves and analyze games alike, it is still a worthwhile statement to get the same result from a second independently trained engine.

Nonetheless, we also expect our engine to be especially strong in the game of Horde. With 52 pieces, Horde has 1.625 times more pieces than regular chess. This is one of the reasons, why Horde has a higher branching factor than standard chess. In comparison to traditional minimax engines with alpha-beta pruning, MCTS grows a tree instead of pruning it. This is especially advantageous, if the branching factor is high.

[1] https://database.lichess.org/

1.2. Problem Formulation

To our current knowledge, no MCTS engine exists for the chess variants 3check, Antichess, Horde, King of the hill and Racing kings, and for the variants Atomic, Chess960 and Crazyhouse there are only isolated MCTS engines. If people are interested in these variants, they will need to download, install and integrate multiple MCTS engines and switch back and forth when using different variants. Currently, there is no engine that uses MCTS and is capable of playing multiple chess variants on an at least average level.

We formulate four research questions. First, we assess the practicality of using the same architecture, reinforcement learning algorithm and training schedule to train models for multiple chess variants. We reuse large parts of the *CrazyAra* engine [5, 7] and modify it to support variants other than Crazyhouse. Second, we attempt to direct the attention of the network by adding the last eight moves to the input representation, and report insights gained from a comparison with the engine sibling *CrazyAra* [5], which uses an input representation without last moves. Third, we analyze the game outcomes of self-play and evaluate the game balance of different chess variants. A better balance not only makes the game more enjoyable, but helps engines learn all aspects of the game. In the case of an unbalanced game, a good engine will nearly always win with the favoured colour and almost never see an endgame, where the not favoured colour is in the lead. Forth, we try to answer the question, whether training a model from zero knowledge, starting only with randomly initialized parameters, outperforms the playing strength of a model initialized with a network trained on human games, as proposed in *AlphaGo Zero* [28].

1.3. Outline

In the next chapter, we give some background information on the chess variants that we train in this thesis, and provide an introduction to reinforcement learning and Monte Carlo tree search. Next, we turn to related work and summarize the application of neural networks in the file of chess and chess variants, give an overview of existing multi-variant chess engines, and introduce the idea behind reinforcement learning from zero knowledge. We then present our methodology by explaining the architecture as well as the setup of supervised and reinforcement learning. The section hereon reports the results of our empirical evaluations and is followed by a discussion of our research questions. Finally, we conclude with a summary and future work.

2. Background

In this chapter, we first introduce the chess variants that we use during this work and explain their ruling. Afterwards, we give an introduction to the basic ideas of reinforcement learning and end with a description of minimax search with alpha-beta pruning and Monte Carlo tree search.

2.1. Chess Variants

In the following sections, we develop an engine that can play several different chess variants. In particular, all variants that are currently supported by the open source platform *lichess.org*[1]: 3check, Antichess, Atomic, Chess960, Crazyhouse, Horde, King of the hill and Racing kings. For the remainder of this text, we will refer to this set of variants as *lichess variants*. Every variant is played on a standard 8×8 chess board and regular chess pieces. Hereafter, we showcase each variant by explaining it's rules and the differences to standard chess.

3check. Follows the same rules as standard chess, except that a player can also win by executing a move that checks the enemy king for the third time.

Antichess. All rules of standard chess apply, except that no castling is allowed, the king can be captured, pawns may be promoted to kings and capturing is forced. If a player can take a piece, the player must; if a player can capture multiple pieces, the player can choose which one. The player who loses all pieces, wins the game. In addition, a player also wins the game when he reaches a stalemate (there are no legal moves left).

Atomic. If a player captures a piece, the capturing piece, the captured piece and all non-pawn chess pieces within a one field radius leave the board. In addition to traditional

[1] https://lichess.org/variant

checkmate, a game can be won by blowing up the enemies king or a piece next to the king. Nuking the opponent king wins the game immediately, overriding any checks and checkmates. Because of these rules, kings cannot capture other pieces and if both kings are standing next to each other, checks do not apply as it is not allowed to blow up the own king.

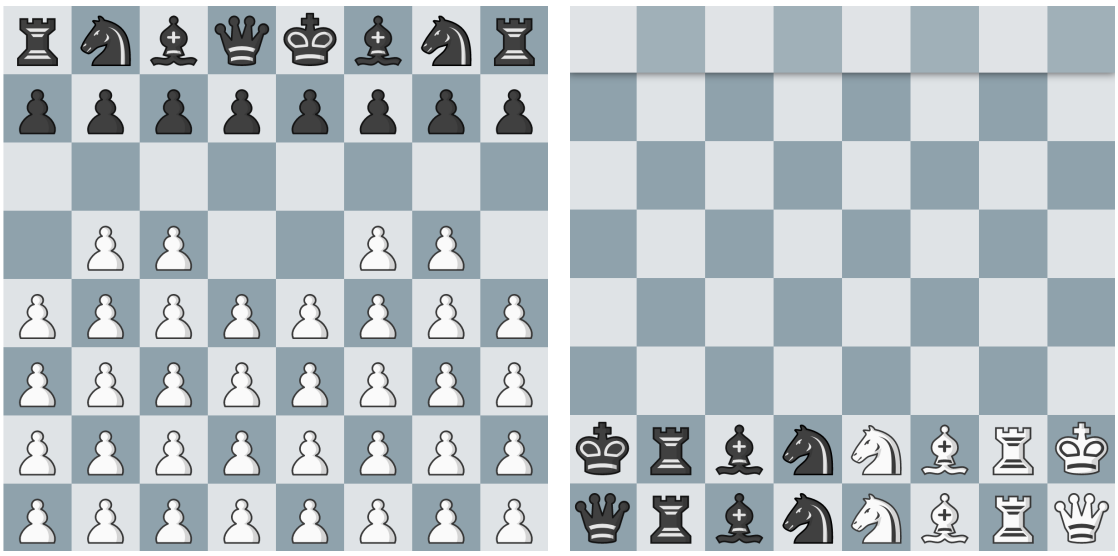
Chess960. This variant is sometimes also called *fischerandom* variant and the only difference to standard chess is a randomized home rank. The home rank is the 1^{st} row for white and the 8^{th} row for black. The only constraint applied when shuffling the home rank pieces is that one rook has to be on either side of the king. When castling, all fields between the king and the rook have to be free, and none of these fields can be in check. After castling, the king and rook always land on the fields, where they would land in standard chess. For this, the destination field of the rook has to be free. It is worth to note that castling is possible, even if the king does not move. There are exactly 960 possible home rank variations, hence the name.

Crazyhouse. The only addition to standard chess are dropping moves. When a player captures pieces from the opponent, the pieces change color and can be dropped onto an empty field of the board instead of playing a regular move. Captured pieces remain inside the *pocket* of a player, until dropped into play or the game has finished. Dropping moves can also prevent checkmates by blocking the path of pieces. Pawns cannot be dropped onto the 1^{st} or 8^{th} rank and promoted pawns that are captured revert to pawns.

Horde. The key difference to standard chess are white's chess pieces. White has 36 pawns (the *horde*) and needs to checkmate black's king according to traditional rules. In comparison, black needs to capture all pawns, including pieces promoted from pawn, to win the game. A special starting position is used, which can be seen in Figure 2.1a. Pawns of the first rank may move 2 squares, but are not allowed to be captured en passant. However, if they move one square and on a later move two squares, en passant capture is possible.

King of the hill. The first player who moves his king to one of the four central squares of the chess board, wins the game. All traditional chess rules are valid, in particular checkmating.

Racing kings. Each player does not have any pawns and both players start on the same side of the board. Blacks pieces take up the left half and whites pieces the right half of the first two ranks. Figure 2.1b shows the arrangement of the pieces in the starting position. It is not allowed to check the opponent's king and thus checkmate is not possible. The



(a) Horde (b) Racing kings

Figure 2.1.: **Starting positions for Horde and Racing kings.** Both starting positions vary significantly from the standard chess starting position.

goal is to move the king to the 8^{th} row, however if the opponent can move to the 8^{th} rank in the next turn, the game is declared a draw.

2.2. Reinforcement Learning

Reinforcement learning [31] is a group of machine learning algorithms that learn by interacting with an *environment*. An environment can be anything from the natural world to a computer simulation. In our case it is a chess board, the chess pieces and the rules of the game. The raw idea of reinforcement learning is to perform an *action*, observe how good the action was and incorporate this new information into our knowledge about the environment. This process is repeated hundreds, thousands or even millions of times until we have a more or less good idea of how our environment behaves and which action performs best in a specific situation.

The entity that executes actions is called an *agent* and the environment at a specific point in time is called a *state*. The agent is always inside a state and decides which action to

execute next. When the action is executed, the environment returns a reward and we transition to the next state. Getting a reward is a core aspect of reinforcement learning. We cannot learn without having a reward function that gives us an idea of how good or bad our actions were.

At the beginning, the agent has no idea how the environment looks like and learns by trial and error. It performs random actions and gets better the more rewards it sees. However, sometimes the agent, which in our case is a chess engine, is already given a more or less accurate model of the environment to save training time. This is often done by training a supervised model on expert data beforehand, which is also what we will do later on.

A reinforcement learning algorithm always exploits the best action it has found, but periodically tries new actions to ensure a constant amount of exploration. There may always be an even better action that we have not discovered yet. In the limit, this procedure will converge to perfect play with sufficient exploration. Determining the correct amount of exploitation and exploration is an important part of reinforcement learning and is different for each use case. If we increase exploitation, we decrease exploration and vice versa. This is called the exploitation-exploration trade-off.

Chess and it's variants are two player games that possess a finite fully observable environment with a discrete action space. They are well suited to be modeled as finite *Markov Decision Processes* (MDPs). This means that the set of states $\mathcal{S}$ (all chess board configurations), the set of actions $\mathcal{A}(s)$ (all legal moves in a specific board state) and the set of return values $\mathcal{R}$ (in our case $\{-1, 0, 1\}$) exhibit a finite number of elements. In MDPs, the future is independent of the past given the present. The agent only needs to know the last state S_{t-1} and the last action A_{t-1} to be able to chose the next action. This is called the *Markov property* and lets us define a deterministic function that completely describes the dynamics of the system:

$$p(s', r|s, a) = \Pr\{S_t = s', R_t = r|S_{t-1} = s, A_{t-1} = a\}, \tag{2.1}$$

where $s, s' \in \mathcal{S}$, $r \in \mathcal{R}$ and $a \in \mathcal{A}(s)$. The function $p : \mathcal{S} \times \mathcal{R} \times \mathcal{S} \times \mathcal{A} \to [0, 1]$ takes four arguments and returns a probability, which tells us how likely it is to transfer to a state s' and get a reward r if we are in state s and execute action a.

During reinforcement learning, we would like to create a model that tells us which actions return the highest rewards. For this, we need to know the expected reward, before making an action. To this end, we assign a value v to each state during training that describes the expected reward we would receive if we transitioned to that state. The goal is to find the

optimal value function $v_*(s)$ in order to know which state we should transition to next. This can be formalized by the Bellman equation of optimality [31]:

$$v_*(s) = \max_a \sum_{s',r} p(s', r|s, a) \left[r + \gamma \, v_*(s') \right], \tag{2.2}$$

where $p(s', r|s, a)$ and all it's variables are defined as in equation 2.1, γ functions as a discount value and lets us enact the idea that future rewards are in some scenarios not as desirable as immediate rewards, $v_*(s)$ states the optimal value at state s and $v_*(s')$ determines the optimal value at a subsequent state s' that we reach by taking action a.

If we have complete knowledge of the environment, the optimal value function can easily be estimated, for example by using *dynamic programming* [31, ch. 4]. In many cases, however, the knowledge that the agent has about it's environment is incomplete. This is also the case for chess, where it is not possible to look at every move in every position. Nonetheless, we can still get an estimate of the value function $v_*(s)$ by employing approximate methods that converge in the limit.

2.2.1. Monte Carlo Methods

When applying *Monte Carlo methods* [31, ch. 5], the agent learns from *actual* or *simulated* experience. The task must be episodic, which means that every random sequence of state-action pairs eventually terminates and returns a value. Each of these trajectories is called an episode and the idea is to sample episodes from the environment many times and continuously update the value function. The return that we get at the end of an episode is propagated back along the trajectory of states, and each state value is updated accordingly. In the limit, Monte Carlo methods create optimal value functions which in turn lead to an optimal policy if we greedily select each action. Because the values change from episode to episode, also the policy changes on an episode-by-episode basis, which is why each episode is sampled from the most recent policy.

Monte Carlo methods are applicable to chess and it's variants. The computer can play games against itself (called *self-play*) and improve it's model with every simulation. What we still miss, is a model that we can sample from and which returns a value at the end of an episode stating the outcome of the game. Luckily, we can adopt the well established tree like representation from the minimax algorithm (section 2.3.1) and combine it with Monte Carlo rollouts to create the *Monte Carlo tree search* algorithm (section 2.3.2), which we use for our engine.

2.3. Search Algorithms

2.3.1. Minimax with Alpha-Beta Pruning

Minimax is a search algorithm to learn the optimal strategy of finite and fully observable zero-sum games, like chess. If a game position is won for the active player A, minimax evaluates the position with 1. If a position is won for the opponent, player B, it is evaluated with -1, and if it is drawn, the return value is 0. This set of return values $\mathcal{R} = \{1, -1, 0\}$ is also used in our Monte Carlo tree search implementation.

If the game is relatively simple, like tic-tac-toe, minimax can look at every move in every position and determine perfect play. To do this, minimax looks at all legal moves that can be played in the current board position. It then tries out a move and lands in a new state. From there it again selects a move und repeats this process until it reaches a terminal state, which is a state where the game is either lost, won or drawn. Now the algorithm backpropagates the return value until it reaches a node, where it still has not tried every move. From there it selects a new move and repeats this process until we have seen every possible state. Minimax knows the expected return value in every state, including the root node, so it can tell you the best move.

This procedure is often displayed as a search tree and for more complex games, there is not the time to search every possible move trajectory from a given position. This is why it is common to only look until a specific depth is reached. At this depth, good positions for player A get a high value and good positions for player B a low value. These values are determined by heuristics, which are highly game specific and sometimes even dependent on the game status.

Another trick to limit the amount of nodes that has to be searched, is a minimax extension called *alpha-beta* search. The idea is simple: do not develop branches which cannot influence the result of the search. To implement this, alpha-beta stores two additional values that describe the guaranteed worst case outcomes for player A and player B. These values are typically called α and β, where α describes the minimum value for player A and β describes the maximum value for player B (player B wants to minimize the result). If we evaluate a position where player A can choose the move and the return value is higher than β, we can stop the search and prune that branch. The reason for this decision is that player B would never allow player A to go into this branch, because then player A would get a better result than if player B had chosen a different move before. This is called a β-*cutoff* and the justification for the α-*cutoff* on player B's turn is analogous.

Note that for alpha-beta pruning the order in which we look at the legal moves of a certain position is important. The vanilla minimax algorithm always has to look at all moves, so the order does not matter, but when implementing minimax with alpha-beta pruning, we want to consider first the moves that have a high priority of creating a small window between α and β. The smaller the window, the higher is the probability of pruning tree branches and saving search time.

2.3.2. Monte Carlo Tree Search

Monte Carlo tree search (MCTS) combines reinforcement learning and tree search. In comparison to minimax, we do not use pruning or heuristics to cope with the complexity of chess, but instead utilize Monte Carlo simulations.

MCTS can be divided into the phases selection, expansion, simulation and backpropagation. It starts by selecting nodes according to some internal strategy. For example: choose the best action in 95% of the time and randomly select an action otherwise. If we end up at a terminal state s_T, we skip step two and three and jump to the backpropagation step. However, when we encounter a new state s^*, we expand the search tree by creating a new child. Now, we simulate a game by performing a random rollout until we reach a terminal state s_T. We evaluate the score of the terminal state s_T and backpropagate the return value along the search path.

For MultiAra, we use a two headed neural network f_θ to facilitate the MCTS search, as introduced in *AlphaZero* [26]. The network outputs an estimate of the value and policy for a given board position. In our case, the value is a number between -1 and 1 and the policy allocates probabilities to every possible chess move. The integration of a neural network (NN) into MCTS was a mayor advancement and one of the reasons MCTS is so successful when playing complex games like chess.

If the NN is sufficiently trained, we do not have to run any search to get a policy distribution. In a tournament setting, we could play from the raw network policy without using any MCTS search at all by always selecting the move that has the highest probability. Although this weakens the playing strength, it is a feature that is not available when relzing on heuristics to evaluate board positions. But most importantly, the value head allows us to skip the simulation step. Instead of simulating a complete rollout to get a return value, we use the NN to tell us how good a board position is.

If we encounter a new state s^*, we pass it to the NN. Subsequently, we create a new child node and assign the predicted policy distribution $P(s, a_i)$ to any possible action a_i. In

addition, we use the returned value v^* to update Q-values. Every state possesses multiple Q-values that save the value of each state-action pair. When a value v^* is backpropagated, we multiply it by -1 after each step and update the Q-values along the search path by a simple moving average (SMA):

$$Q'(s_t, a_t) = Q(s_t, a_t) + \frac{1}{n_t} \left[v^* - Q(s_t, a_t) \right], \tag{2.3}$$

where s_t and a_t describe the selected state and action along the search path for every step t and n_t counts the number of simulations, which is the number of times that we selected action a_t in state s_t. If the newly encountered state is a terminal state s_T, we do not use the NN, but instead a constant evaluation of -1 (game lost), 0 (game drawn) or 1 (game won) as the state value v^*. Any states that we do not visit during search are treated as losses and get assigned a value of -1.

Our configuration of the NN with a value and policy head is called an *actor-critic* approach and the loss function that we use to train the NN combines the value and policy loss in the following way:

$$l = \alpha(z - v)^2 - \beta(\pi^T \log \mathbf{p}) + c \, ||\theta||^2. \tag{2.4}$$

z and π are the true value and policy, v and $\mathbf{p}$ the predicted value and policy and c is a regularization parameter on the network parameters θ. Our goal during training is to learn the best configuration for these network parameters, so that in a tournament setting we know how good a board position is and which actions are the most promising. It is important to understand that we still run MCTS search during a tournament and ultimately select the move that had the most simulations. However, whether a move gets a lot of simulations is highly influenced by the output of the neural network, which in turn is governed by the set of parameters.

The only thing left is to choose which action a_t to take in a state s_t for time step t. As in all reinforcement learning, this is a very difficult problem and called the exploitation-exploration trade-off. As introduced by *AlphaGo* [27], we use the *Upper Confidence Bound*

Applied to Trees (UCT) formula, which selects the action that maximizes a combination of the Q-value and the weighted policy:

$$\begin{aligned} a_t &= \text{argmax}_a(Q(s_t, a) + U(s_t, a)), \text{ with} \\ U(s, a) &= c_{\text{puct}} P(s, a) \frac{\sqrt{\sum_b N(s, b)}}{1 + N(s, a)}. \end{aligned} \tag{2.5}$$

The variable b specifies each action that is available in state s, $N(s, b)$ counts the number of times we have chosen an action in state s, $N(s, a)$ counts the number of times we have chosen action a in state s, $P(s, a)$ represents the policy returned by the NN for state s and c_{puct} is a hyperparameter which controls the amount of exploration. The exact value of c_{puct} as well as all other parameter settings that we used to run MCTS can be found in section 4.4.2.

3. Related Work

The first engine playing a chess-like game, the anti-clerical chess variant, has been introduced in 1956 [24]. In 1957 followed the first engine that could play a standard game of chess [2]. In 1958 the chess engine *NSS* plays the game of chess for the first time by using the alpha-beta search extension of the minimax algorithm. Afterwards, alpha-beta search dominates the chess engine world for nearly six decades. The main focus was put on developing new and more sophisticated heuristics to select nodes and cut off tree branches. This led to the chess engines *Houdini*, *Komodo* and *Stockfish*, which dominated the chess world since 2011. It wasn't until 2017 that the reign of alpha-beta search engines was broken by *AlphaGo*, a neural network engine that uses Monte Carlo tree search [28].

In the next part, we give an overview over the history of neural networks in standard chess and for chess variants. Afterwards we introduce other engines that are able to play multiple chess variants and end with a short recap about all relevant information that explain our interest in training a model from zero knowledge.

3.1. Neural Networks in Chess

3.1.1. Standard Chess

The very first engines that used networks were *KnightCap* [1] and *Meep* [35]. They used *temporal difference learning* [31, ch. 6] to tune their network parameters, but heavily relied on manually constructed input features. Both engines can rather be seen as a linear combination of selected features instead of neural network engines, because they use mainly static evaluations.

The first neural network chess engines were *Giraffe* [20] and *Zurichess* [22] in 2015 and *DeepChess* [8] in 2016, which used fully-connected feed forward layer and heavily relied on sophisticated input features.

Following achievements in Go [27, 28], Google DeepMind released in 2017 *AlphaZero* [26], the first deep convolutional neural network (CNN) chess engine that had been trained with reinforcement learning using Monte Carlo tree search. The engine began training with a randomly initialized network and only played against itself to improve. AlphaZero achieved super human performance and was so powerful that it defeated the reigning world champion alpha-beta search engine *Stockfish 8* by winning 155 and only loosing 6 games in a 1000 game match [26].

In 2018 Gary Linscott created the open-source engine *Leela Chess Zero*[1], which reimplements the key ideas of AlphaZero. After a complete rewrite, the engine was renamed to *Lc0*[2] and won the 15$^{\text{th}}$ and 17$^{\text{th}}$ Top Chess Engine Championchip (TCEC) and placed second in the 14$^{\text{th}}$ and 18$^{\text{th}}$ - 20$^{\text{th}}$ TCEC.

Since 2018, also the proprietary engine *Komodo* [19] is able to use Monte Carlo tree search instead of minimax with alpha-beta pruning. In the same year, the chess engine *Sashimi*[3] has been introduced, which also replicates the ideas of AlphaZero, but uses a linear model for move selection and position evaluation, and has been exclusively trained as a supervised model.

At the beginning of 2021, a new engine named *Ceres*[4] has been created, which is short for *chess engine research*. Ceres also uses Monte Carlo tree search and utilizes the backends of Lc0, but includes many new ideas that could potentially increase it' playing strength beyond Lc0.

3.1.2. Chess Variants

Although neural networks, accompanied in particular with MCTS, have been used in chess over the last years, chess variants have not really been touched. One exception is Chess960, which is often trained alongside standard chess and is supported by both, Lc0 and Komodo.

[1] https://github.com/LeelaChessZero/lczero
[2] https://github.com/LeelaChessZero/lc0
[3] https://github.com/zxqfl/sashimi
[4] https://github.com/dje-dev/Ceres

The open-source engine *CrayzAra* uses MCTS and reimplements AlphaZero for the chess variant Crazyhouse [7]. It has recently been upgraded by several extension, notably a new method called *Monte Carlo graph search*, which uses an acyclic graph instead of a tree [6]. This greatly reduces memory consumption and improves playing strength. The engine we develop in this work builds on CrazyAra, it's architecture and improvements, which led to the similar name *MultiAra*.

In addition, the engine *Nebiyu*[5] also uses MCTS to play Atomic and Crazyhouse, however no pretrained neural network models are available and must be generated by the user himself.

3.1.3. Efficiently Updatable Neural Networks.

In the last years, a new type of network has made it's debut, which is called *Efficiently Updatable Neural Network* (NNUE). Introduced in 2018 by Yu Nasu for the game shogi [23], it has been adopted to chess and chess variants over the following years and increased the Elo of alpha-beta engines enormously.

Stockfish supports NNUE since August 2020 and increased it's strength for about 100 Elo only with the first version of the network [6].

Similar improvements could be achieved by training a NNUE for *Fairy-Stockfish* [10], an engine that is able to play many chess variants, including all *lichess variants*. In particular, four *lichess variants* have been improved with NNUE's at the time of writing: 3check increased by about 240 Elo, Atomic by about 500 Elo, King of the hill by about 600 Elo and Racing kings by about 280 Elo. The exact NNUE versions and model names that we used during our empirical evaluation, can be seen in Appendix D.2.

In board games, NNUEs are mainly used to evaluate the state of the board and predict the goodness of a position. NNUEs are relatively small networks that run efficiently on CPUs and are not dependent on GPUs like other neural networks. NNUEs have very few layers, for example *Stockfish* 14 uses only 4 layers at the time of writing, where most of the parameters are situated in the first layer [7]. The input representation is designed in a way, that only a few input nodes change and only a small portion of the weights needs to be updated, if a move is played or taken back. Moreover, the calculation of the other

[5] https://sites.google.com/site/dshawul/home
[6] https://stockfishchess.org/blog/2020/introducing-nnue-evaluation/
[7] https://github.com/glinscott/nnue-pytorch/blob/master/docs/nnue.md

layers can efficiently be done by using CPU specific instructions and the parameters of the network can be trained by using data from human professionals, games generated during reinforcement learning self-play or both.

3.2. Multi-Variant Chess Engines

According to our current knowledge, there are only *Fairy-Stockfish* [10] and *Multi-Variant Stockfish* [9], both alpha-beta search engines, that can play all eight *lichess variants*. However, Multi-Variant Stockfish has been archived since the beginning of 2021, which leaves Fairy-Stockfish as only choice, if you want to play all *lichess variants* against the same engine.

Additionally, there are multiple chess engines that can play individual *lichess variants*. *PyChess*[8] supports seven variants (excluding Racing kings), but has no sophisticated planning algorithm and plays rather for fun instead of focusing on raw strength. *Pulsar*[9] is able to play 3check, Antichess, Atomic, Chess960 and Crazyhouse, and *Sjaak*[10] can play Antichess, Chess960, Crazyhouse and King of the hill.

It is important to note that all of these engines use minimax with alpha-beta search. According to our current knowledge, there is no engine that uses MCTS in conjunction with a neural network which is able to play all or a subset of the *lichess variants*.

3.3. Training From Zero Knowledge

In 2016 Google DeepMind released *AlphaGo* [27], a program that uses convolutional neural networks and Monte Carlo tree search to play the game Go at a superhuman level. One year later, DeepMind trained it's engine again with an improved network structure, but more importantly decided to refrain from any human expert data and only let the engine play against itself [28]. In other words, DeepMind initialized their new engine *AlphaGo Zero* with random weights instead of training it in a supervised fashion with expert human data before starting reinforcement learning.

[8] https://github.com/pychess/pychess
[9] http://www.lanternchess.com/pulsar/
[10] http://www.eglebbk.dds.nl/program/chess-index.html

In addition, DeepMind trained another prototype, *AlphaGo Master*, which uses the same architecture and algorithm as *AlphaGo Zero*, but was pretrained on human expert data and used the same hand-crafted features as *AlphaGo*. *AlphaGo Zero* performed better than *AlphaGo Master* with an Elo rating of $5,185$ and $4,858$ respectively. In a 100 game tournament, *AlphaGo Zero* defeated *AlphaGo Master* with 89 wins and 11 losses. DeepMind summarizes that "a pure reinforcement learning approach requires just a few more hours to train, and achieves much better asymptotic performance, compared to training on human expert data" [28].

Following these findings, in 2017 DeepMind trained it's new algorithm *AlphaZero* using only reinforcement learning self-play and no human data [26]. The engine is able to play Go, chess and shogi and achieves superhuman performance in all three games.

As far as we can tell, there are no other studies that examine whether an initial model, trained on human expert data, produces a weaker MCTS engine, in comparison to a pure reinforcement learning approach. To shed light on this question, we execute an ablation study, where we train the chess variant King of the hill twice with the same network architecture and the same settings, but one time with and one time without human expert knowledge.

4. Methodology

The basis for our experiments is the pipeline programmed in [5, 7] by Czech et al. Our mayor contributions to the code are multi-variant changes to the binary, the dynamic loading of different chess variants, unit tests to ensure that all chess variants work as intended and a refactoring of the reinforcement learning setup to simplify the procedure.

In the first part of this chapter, Section 4.1, we describe the computational layout that we used to train our models. The preprocessing steps and model architecture that we utilized for each chess variant are displayed in Section 4.2. We started model development by training each model on human data in a supervised fashion, and Section 4.3 describes the data set and training settings used for this purpose. After training on human data, we refined the supervised models by employing reinforcement learning. The goal we have pursued in our reinforcement learning setup as well as the engine and training settings can be looked up in Section 4.4. Additionally, we describe the setting changes we made for training an additional model from zero knowledge. Lastly, in Section 4.5, we specify the tournament setup that we used to test MultiAra and end with the availability of MultiAra.

4.1. Computational Layout

Hardware. All training and testing has been executed on two *NVIDIA DGX-2* server instances. Each server features 16 *NVIDIA Tesla V100* GPUs with 32GB memory and dedicated CUDA and tensor cores. Each server also contains 96 *Intel Xeon Platinum* 8174 CPUs with 3.1 gigahertz and 2 threads per core. The effective number of GPUs used to generate games and train models varied from 1 to 16, depending on the amount of traffic on the servers. For tests, we either used 1 GPU and 3 CPU threads (MultiAra and CrazyAra) or 4 CPU threads (Fairy-Stockfish).

Software. For all code that we ran on the *DGX-2* servers, we used a docker container that has been build upon the official *MXNet-NVIDIA-Docker*[1] image (release 20.09). It uses Ubuntu 18.04, Python 3.6, NVIDIA CUDA 11.0.3, cuDNN 7.5.1.10 and version 418.126.02 of the NVIDIA GPU driver. As a back end we use *TensorRT*, but version 7.2.1 instead of version 7.1.3, which is the standard of the 20.09 release of the MXNet image. Further, we installed several C++ dependencies needed to build and run the binary: Boost 1.7.0[2], XTL 0.6.5[3], XTensor 0.20.8[4] and z5 2.0.5[5].

The MultiAra version that we used for testing can be downloaded from the *CrazyAra* repository[6] using commit 6690a514e1f7319b555a39c9ce0808f82b7ba7d9.

4.2. Architecture

4.2.1. Preprocessing

Our input representation is uniform, in the sense that it can represent any position of any chess variant, which is played on an 8×8 chess board. To achieve this, our input representation features 63 planes with a size of 8×8, where the 8×8 structure displays the 64 fields of the chess board while preserving it's layout (Table 4.1).

The first 12 planes are used to describe where the chess pieces of each player are located. Next, we have 2 planes that indicate how often the board position has already occurred (a player can claim a draw after 3 repetitions); 5 planes for each player to display the pocket pieces if we are playing Crazyhouse; 2 planes to indicate pieces that have been promoted from pawns; 1 plane to indicate whether an en passant capture is possible; 1 plane to indicate whether black or white has to move next; 1 plane to indicate the total move count; 4 planes to track castling options; 1 plane to represent the no progress counter; 4 planes to indicate how often a player has already been checked (needed for the 3check variant); 1 plane to indicate whether the game started from a random position; and 8 planes that display the active variant and are one hot encoded .

[1] https://docs.nvidia.com/deeplearning/frameworks/mxnet-release-notes/running.html
[2] https://www.boost.org/
[3] https://github.com/xtensor-stack/xtl
[4] https://github.com/xtensor-stack/xtensor
[5] https://github.com/constantinpape/z5
[6] https://github.com/QueensGambit/CrazyAra/

Feature	Planes	Type	Comment
P1 pieces	6	bool	order: {PAWN, KNIGHT, BISHOP, ROOK, QUEEN, KING}
P2 pieces	6	bool	order: {PAWN, KNIGHT, BISHOP, ROOK, QUEEN, KING}
Repetitions*	2	bool	indicates how often the board positions has occurred
P1 pocket count*	5	int	order: {PAWN, KNIGHT, BISHOP, ROOK, QUEEN}
P2 pocket count*	5	int	order: {PAWN, KNIGHT, BISHOP, ROOK, QUEEN}
P1 Promoted Pawns	1	bool	indicates pieces which have been promoted
P2 Promoted Pawns	1	bool	indicates pieces which have been promoted
En-passant square	1	bool	indicates the square where en-passant capture is possible
Colour*	1	bool	all zeros for black or all ones for white
Total move count	1	int	sets the full move count (FEN notation)
P1 castling*	2	bool	order: {KING_SIDE, QUEEN_SIDE}
P2 castling*	2	bool	order: {KING_SIDE, QUEEN_SIDE}
No-progress count*	1	int	sets the no progress counter (FEN halfmove clock)
P1 remaining-checks*	2	bool	tracks the number of checks (needed for 3check variant)
P2 remaining-checks*	2	bool	tracks the number of checks (needed for 3check variant)
is960*	1	bool	indicates whether the game uses a random starting position
Variant*	8	bool	one-hot encoded; order: {chess, crazyhouse, kingofthehill, 3check, antichess, atomic, horde, racingkings}
Last 8 moves	**16**	bool	indicates the origin and destination square of the last eight moves, starting with the most recent move
Total	**63**		

Table 4.1.: **Neural network input representation**. Each input plane contains 8×8 values, however planes with $*$ are binary planes, where each of the 64 values is the same. P1 stands for the active player and P2 for the opponent. The values of a plane with the type bool can only be 0 or 1.

Lastly, we added 16 new planes to the input representation used in [5], which indicate the origin and destination squares of the last eight moves, starting with the most recent move. The idea is that last moves function as a mechanism that guides the attention of the network.

All layers are normalized to the linear range of $[0, 1]$, as described in [7]. Additionally we want to note that the *is960* layer and the eight *Variant* layers are not really needed for our experiments, because we train a separate network for each variant. However, we kept the planes to ensure comparability to other runs with the same network structure.

4.2.2. Convolutional Neural Network

As neural network architecture, we used the *RISEv2 mobile* architecture from [5] with 13 residual blocks. Instead of fully connected layers, this architecture uses convolutions [21] to process the input. Although convolutional neural networks (CNNs) are mainly used in computer vision and natural language processing, they can also be used for chess, if we encode the board state as an 8×8 multi-channel image, which we did. However, we did not use any pooling layers and preserved the spatial size, which is unusual for CNNs, but works well when processing the position of a board game.

The core of the network consists of a residual tower (Table 4.2) followed by a value (Table 4.3) and policy head (Table 4.4). We used 13 residual blocks, where each block uses convolutions, batch normalization and rectified linear unit (ReLU) activation. Before the first residual block, the input is processed by 256 convolutional layers with a size of 3×3.

Residual networks have been introduced in [13] and were also used in the *AlphaZero* network architecture [26], which was the starting point for the RISEv2 mobile network architecture. The common residual connections have been replaced by inverted residual connections, as introduced in the *MobileNetV2* architecture [25], and the convolutions have been replaced by depthwise separable convolutions as shown in the *MobileNetV1* architecture [15]. Both increase the memory efficiency of the network.

Another improvement over the *AlphaZero* architecture is the size of the residual blocks, which follows a linear increasing pyramid structure [12], growing from 512 layers in the first to 2048 layers in the last block. In addition, the last 5 blocks use a squeeze and excitation mechanism [16], where the network weights each feature channel based on it's importance.

Czech et al. showed in previous work [5, 6, 7] that the *RISEv2 mobile* network architecture works well in conjunction with Monte Carlo tree search and achieves similar performance as *AlphaZero*, while evaluating nodes twice as fast on GPU and three times as fast on CPU [5]. The average nodes per second (NPS) typically correlates with the strength of an engine.

At the end of the residual tower, the network is split into a value and a policy head. The value head uses a *tanh* activation function to predict a single value between -1 and 1 (Table 4.3) and the policy head uses a *softmax* activation function to output a vector of size 5184, where each entry encodes a different chess move (Table 4.4). The single value is used to determine the goodness of the current position and the policy map functions as a guideline to predict the moves the engine should look at.

Layer Name	Output Size	RISEv2 Mobile with 13 Residual Blocks
conv0 batchnorm0 relu0	$256 \times 8 \times 8$	conv 3×3, 256
res_conv0_x res_batchnorm0_x res_relu0_x res_conv1_x res_batchnorm1_x res_relu1_x res_conv2_x res_batchnorm2_x shortcut + output	$256 \times 8 \times 8$	$\begin{bmatrix} \text{conv } 1 \times 1, 128 + 64x \\ \text{dconv } 3 \times 3, 128 + 64x \\ \text{conv } 1 \times 1, 256 \end{bmatrix} \times 8$
res_conv0_x res_batchnorm0_x res_relu0_x res_conv1_x res_batchnorm1_x res_relu1_x res_conv2_x res_batchnorm2_x shortcut + output	$256 \times 8 \times 8$	$\begin{bmatrix} \text{(SE-Block, } r = 2) \\ \text{conv } 1 \times 1, 128 + 64x \\ \text{dconv } 3 \times 3, 128 + 64x \\ \text{conv } 1 \times 1, 256 \end{bmatrix} \times 5$
value head policy head	1 5184	Table 4.3 Table 4.4

Table 4.2.: **RISEv2 mobile architecture with 13 residual blocks.**

Layer Name	Output Size	Value Head 8 Channels
conv0 batchnorm0 relu0	$8 \times 8 \times 8$	conv 1×1, 8
flatten0 fully_connected0 relu1	256	fc, 256
fully_connected1 tanh0	1	fc, 1

Table 4.3.: **Value head as shown in [7].**

Layer Name	Output Size	Value Head 8 Channels
conv0 batchnorm0 relu0	$256 \times 8 \times 8$	conv 3×3, 256
conv1 flatten0 softmax0	5184	conv 3×3, 81

Table 4.4.: **Policy head as shown in [7].**

4.3. Supervised Learning

Before starting the reinforcement learning process, we trained a supervised model on human expert data for each chess variant to initialize the reinforcement learning run.

4.3.1. Data Set

For each variant, we downloaded all available games from the *lichess.org open database* [32] that have been played between August 2013 and July 2020. We selected only games played between players of the highest 10^{th} percentile. The resulting minimum Elo threshold lies at 2000 for Crazyhouse, 1950 for Chess960, 1925 for King of the hill, and 1900 for 3check, Atomic, Horde and Racing kings. For Antichess we chose a cutoff of 2200.

The games from April 2018 function as test and the games from August 2018 as validation data, and are not used during training. The amount of data available for supervised training varied from 90.2 thousand games for Racing kings to 956.2 thousand games for Crazyhouse (see Figure 4.1).

4.3.2. Training Configuration

For training the neural network, we decided to use the same settings as in [5], which originated from [7]. After several tryouts, we realized that the training configuration from previous work functions well and results in a constantly improving loss and the absences of too many spikes. In addition, we decided to keep the batch size at 512 to increase comparability with it's sister project *CrazyAra* [5].

Every chess variant was trained for 7 epochs, without a reward discount factor and without using dropout. During training we utilized a spike recovery mode, which reverts the network to the last model state, if a spike greater than 1.5 has been experienced. If 20 or more spikes occurred, the training run was aborted, which did not happen. The loss has been calculated as in equation 2.4 with weighting parameters $\alpha = 0.01$ and $\beta = 0.99$ as suggested by [27] to avoid overfitting. In addition, we used $L2$ regularization with a weight decay of 0.0001 on the network parameters to avoid overfitting even further.

As optimizer we used gradient descent with *Nesterov momentum* [3] and applied a learning rate and momentum scheduler as proposed in [29] with a maximum learning rate of of 0.35 and a minimum learning rate of 0.00001.

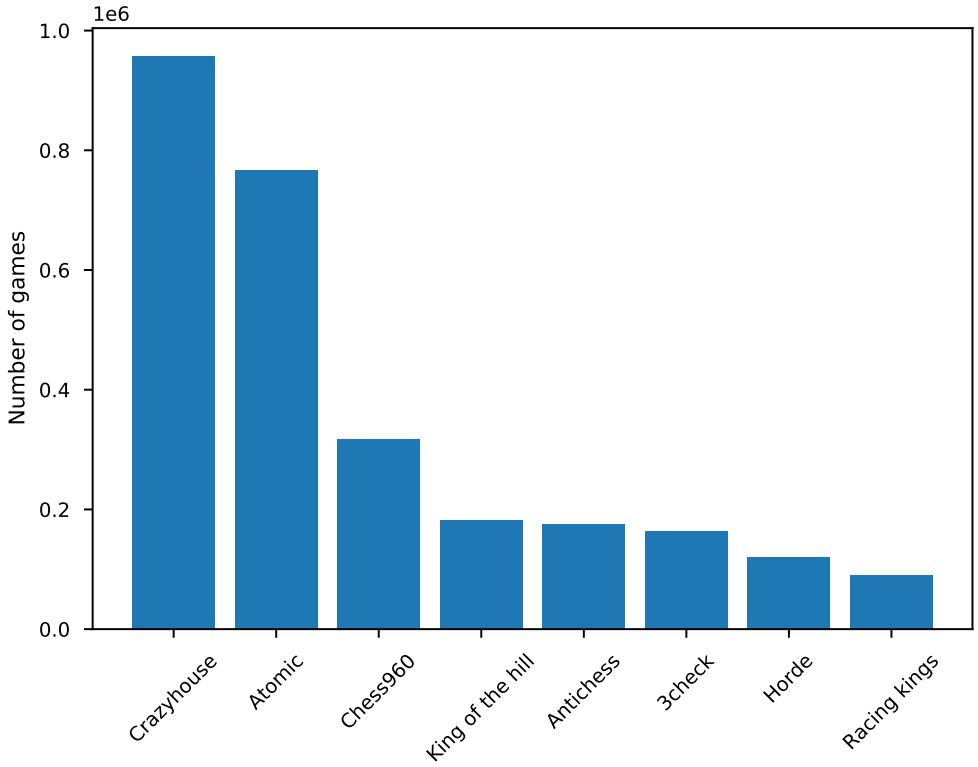
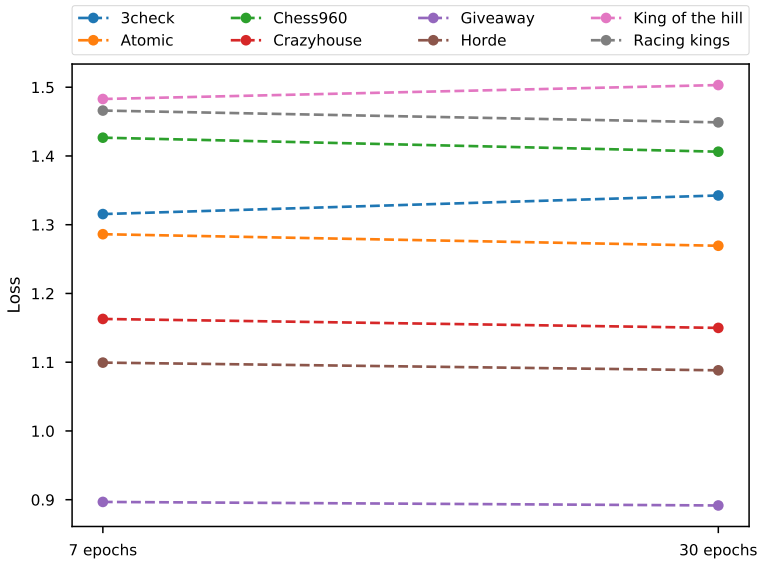
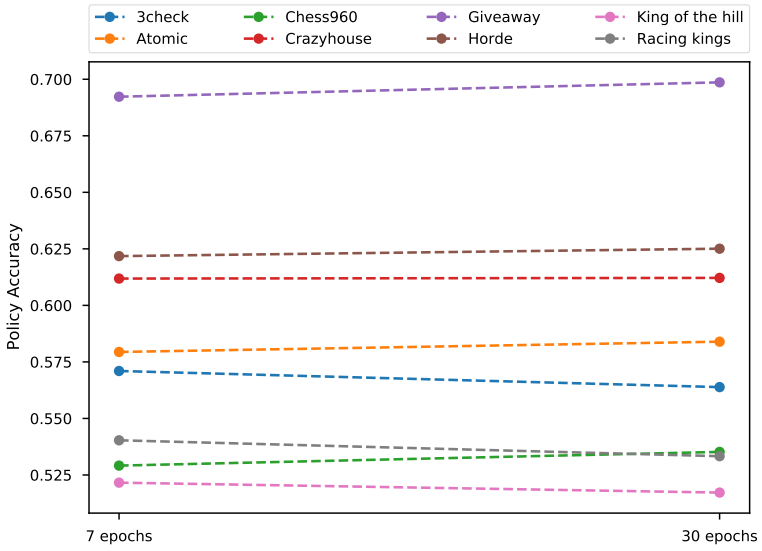


Figure 4.1.: **Human games available for supervised training**. Only games between players of the highest 10^{th} percentile from the *lichess.org open database* [32] have been selected.

Remarks. We did not only train supervised models for 7 epochs, but also for 30 epochs. We wanted to test whether more epochs result in a lower loss or higher policy accuracy. However, neither the loss nor the policy accuracy significantly improved when training the models for 30 instead of 7 epochs (Figure 4.2). Models that have trained with fewer epochs generalize better to unseen data, which is why we decided to use the models trained for 7 epochs as the starting point of our reinforcement learning runs.



(a) Loss



(b) Policy accuracy

Figure 4.2.: **Comparing networks trained on human games.** For each variant, we trained a network for 7 and for 30 epochs and compare the loss and policy accuracy.

4.4. Reinforcement Learning

4.4.1. Goal

Our main goal in this work is to create an MCTS engine that can play eight different chess variants. However, we also wanted to know how viable a single reinforcement learning setup is for training different games. Therefore, we designed the reinforcement learning setup as simple as possible. Please note that this approach may not result in the best networks, but provides us with valuable insights about how each variant behaves. To achieve the highest possible playing strength, more tests would be needed to finely adjust the hyperparameters of each chess variant.

The reinforcement learning process consists of two different steps: the game creation step, also called self-play, and the training step. During game creation the engine plays against itself to create new games and during training the network's parameters are updated based on the created games. We will first describe the settings for the game creation step and afterwards the settings for the training step. For a detailed description of how the settings are implemented, please see [5] and [7]. Afterwards we will briefly go over the changes we made to train a second King of the hill model from zero knowledge.

4.4.2. Engine Settings

The engine settings determine how games are played during self-play. The settings especially take care of the amount of exploration and the amount of nodes that are evaluated.

Evaluation depth. We set the number of *simulations* to 3200, the number of *nodes* to 800 and the *batch size* to 8. Per position, 800 nodes of the tree are searched or 3200 simulations are executed. If the maximum number of nodes or simulations is reached, the search stops and the best move gets returned. The batch size determines the number of threads that are used to execute the search, which means that in our case each thread only searches 100 nodes or 400 simulations. To enable multi-threaded search, we temporarily affected all visited nodes by a *virtual loss* of 3. Further, we set the *node random factor* to 10, so that the amount of nodes that are evaluated are sampled from $[760, 840]$ instead of always searching for exactly 800 nodes. In addition, we used a *fixed move time* of 1 second to force a move if search takes too long.

Exploration. Game generation will only improve our network if we do not play the same moves every game. Because engines work deterministic, we describe in the next four paragraphs several options to ensure sufficient exploration.

We start by applying *Dirichlet noise* to the root node s_0 of our tree. In our case, we chose $\alpha = 0.2$ for the Dirichlet distribution with a factor of $\epsilon = 0.25$, which increases the probability of moves that have a low policy probability.

$$\mathbf{p}'_{\mathbf{s_0}} = (1 - \epsilon) \times \mathbf{p}_{\mathbf{s_0}} + \epsilon \times \text{Dir}(\alpha) \tag{4.1}$$

Additionally, we sampled a random number of starting moves from the policy. The exact number varies from game to game, but is determined by an exponential distribution that has a mean at 8. Because the exponential distribution has a very long tail, we cut back samples that are higher than 30. These moves count as book moves and are not used when training the neural network. During this phase, we applied in 5% of the cases a temperature scaling of at least 2 to the policy distribution.

After playing book moves from the policy, the next 15 moves are sampled with a temperature of 0.8. The temperature is an exponential scaling that is applied to the policy vector after the board position has been evaluated. The policy vector is sharpened, which makes moves with a high probability even more likely. This enhancement is increased after every move by black by applying an exponential decay of 0.92 to the temperature. We want the engine to play games that exhibit exploration, but on the same time minimize the chance to play moves that have a low probability, because low probability moves often end in a blunder which turns around the game result. This would not only influence the moves after the blunder, but also confound the evaluation of the moves before the blunder.

Another exploration parameter is the constant c_{puct}, which weights the Q- and U-values in the UCT formula (Equation 2.5). We tuned it according to the following formula:

$$c_{\text{puct}}(s) = \log \frac{\sum_a N(s,a) + c_{\text{puct-base}} + 1}{c_{\text{puct-base}}} + c_{\text{puct-init}}, \tag{4.2}$$

where we used a $c_{\text{puct-base}}$ of $19,652$ and $c_{\text{puct-init}}$ of 2.5

Centipawn conversion. The centipawn conversion transforms the Q-values of our network into the widely used centipawn (cp) metric, which is used by traditional chess engines. We reused the formula of [7] with $\lambda = 1.2$:

$$\text{cp} = -\frac{v}{|v|} \times \log \frac{1 - |v|}{\log \lambda}. \tag{4.3}$$

Q-Veto Delta. A new mechanic recently introduced by Johannes Czech into the *CrazyAra* universe, is a comparison between simulations and Q-values to determine the best move. Usually the move that gets the most visits during MCTS search will be picked as next move. However, if the second best move had a Q-value that was at least 0.4 centipawns better than the move with the highest visit count, we selected the second most visited move instead. During this process, we had the option to select a weighting factor that would consider the mixture of Q-value and number of simulations, however, we decided to only use the Q-value.

Search algorithm. As search algorithm we used Monte Carlo graph search (MSGS), which has been developed by Czech, Korus, and Kersting. Only for Atomic we used Monte Carlo tree search. MCGS uses a directed acyclic graph instead of a tree to enable information flow between subtrees and reduce memory consumption [6]. In comparison to vanilla MCTS, we used a mixture of return value and Q-value as our target. Each target value still used 85% of the value given back by the neural network, but also incorporated 15% of the Q-value.

Arena games. To determine whether the new model is stronger than the old model, we performed a 100 game arena tournament after every model update. We used the same aforementioned engine settings and only changed the temperature of the first 15 moves, which we reduced from 0.8 to 0.6 to get an even sharper distribution. This results in a reduced exploration and a higher playing strength.

Further, we used transposition hash tables to reuse the value of nodes that have already been computed, but all other parameters were turned off. In particular, we did not use any kind of time management system or tablebases, did not reuse the search tree and did not use ponder, a feature of chess engines to analyze the game while the opponent thinks. A listing of all engine settings can be seen in Appendix C.1.

4.4.3. Training Settings

When enough games had been generated during self-play, the network was retrained in a supervised fashion, by using the newly generated games. This is why we reused the supervised training settings from section 4.3.2 with the following changes.

We lowered the maximum learning rate from 0.35 to 0.05 and the minimum learning rate from 0.0001 to 0.00005. During self-play we explore the domain to look for new moves that might be stronger than the currently best move. This procedure does not always select the best move, which is why we chose a learning rate that is a lot lower than the learning rate for supervised training when we use human games.

Further, we only trained for 1 epoch instead of 7. Tests have shown that our network starts to overfit, if we select 2 or more training epochs. We also changed the portion of value loss and policy loss used to calculate the overall loss. We increased the value loss factor α from 0.01 to 0.5 and the policy loss factor β from 0.99 to 1.0, resulting in the same loss function that is used in *AlphaZero* [26].

A complete enumeration of the training settings can be seen in Appendix C.2. However, we also designed a reinforcement learning schedule to ensure better convergence. Table 4.5 shows the three changes which we conducted over the course of a reinforcement learning run. At update 10 we increased the number of packages that were needed to retrain the network parameters from 10 to 15. Each package is created by a single GPU and contains 81920 samples. At update 20 we decreased the maximum learning rate from 0.05 to 0.005 and again at update 30 from 0.005 to 0.0005 (Table 4.5).

Update	Option	Change
10	Packages needed per model update	$10 \rightarrow 15$
20	Maximum learning rate	$0.05 \rightarrow 0.005$
30	Maximum learning rate	$0.005 \rightarrow 0.0005$

Table 4.5.: **Option changes during reinforcement learning.** Each export package is created by a single GPU and contains exactly 81920 new samples.

4.4.4. Training From Zero Knowledge

As an ablation study, we wanted to compare how reinforcement learning differs, if we start training with and without knowledge. We chose the chess variant King of the hill and trained it a second time, starting with random weights. We used the same reinforcement learning settings as for the other training runs (Section 4.4.2 - 4.4.3), but the model stopped training after 30 updates, exactly at the point where we reduced the learning rate for the second time.

We plotted the playing strength of each model update against the playing strength of the supervised King of the hill model (Figure 4.3). We wanted to see, how long the training run from zero knowledge needed to compete with the supervised model that we used to initialize the other King of the hill training run. It appeared that after 20 updates the model could match the supervised model. This is why we returned to update 20 and basically started the reinforcement learning run from there again. The maximum learning rate schedule (Table 4.5) subsequently shifted by 20 updates with the first reduction at update 40 from 0.05 to 0.005 and the second reduction at update 50 from 0.005 to 0.0005. The complete training period subsequently incorporated 80 updates to match the 60 updates for the original King of the hill model.

4.4.5. Remarks

Atomic. When trying to train Atomic with Monte Carlo graph search (MCGS), the binary regularly generated `NaN` during self-play. We were not able to fix the error, which is probably related to the explosions in Atomic, which is why we decided to train Atomic with MCTS in this work.

Chess960. We could not improve the Chess960 model that we had trained on human games by reinforcement learning. We tried to reduce learning rate by a factor of 10 or 100, and virtually switched off any kind of exploration, but without success.

Racing kings. We disabled mirroring for Racing kings. Mirroring is a feature that is enabled in most MCTS engines by default. It is useful for games that use a mirrored setup, because the engine only has to learn the value and policy of a board state from the point of view of the active player. For a two player game, it reduces the number of board states that the engine must be able to identify by half. Additionally, in the case of chess, the network also learns that pawns only move forward. However, Racing kings has no pawns and both players start from the same row. Horizontal mirroring would not help

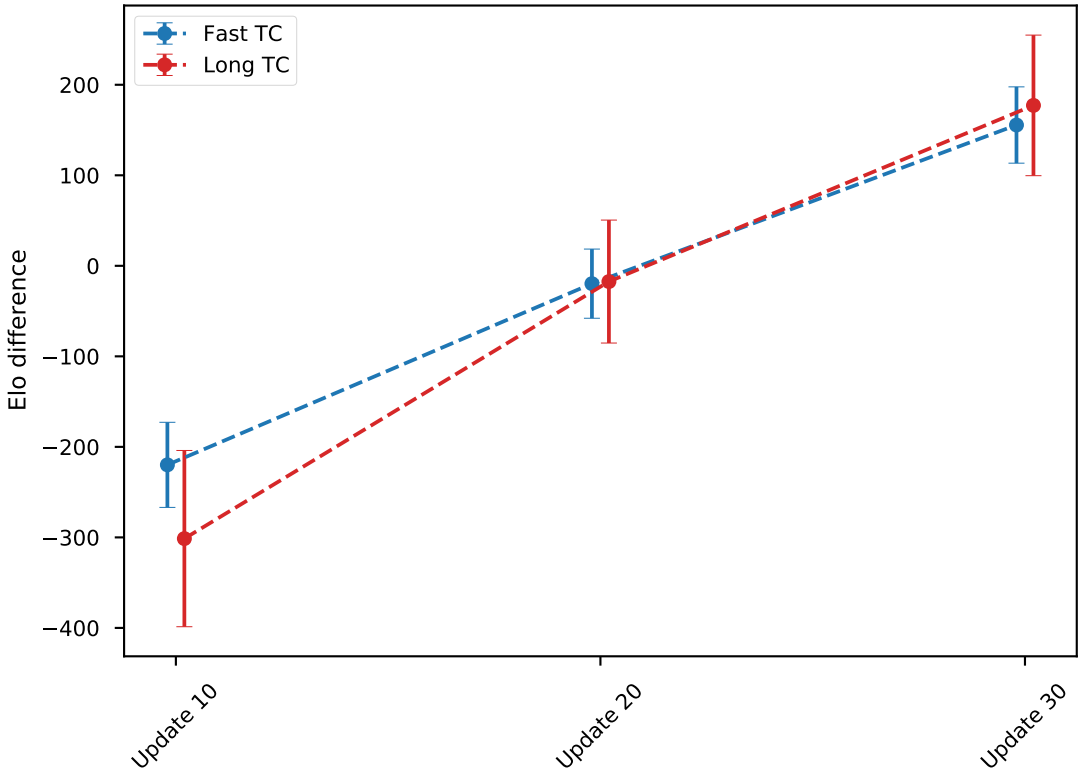


Figure 4.3.: **Model from zero knowledge vs. model trained on human games.** We show the Elo difference between every 10 model updates of the King of the hill model that we trained from zero knowledge and a supervised model that we trained using only human data. During the fast/long time control (TC) each player had $10/60$ seconds for the game plus $0.1/0.6$ seconds per move.

the neural network, but rather make it more complicated. Racing kings has a relatively small branching factor in comparison to chess or other chess variants, so mirroring is not really needed to decrease the amount of positions the network needs to identify. We tested both setups and found that turning off mirroring increases the Elo substantially (see Appending D.3).

4.5. Tournament Settings

Software. To evaluate the goodness of our models, we ran more than 250 tournaments, where two engines played against each other. To administer a tournament, we used version $1.0.0$ of the *Cute Chess* command line interface[7], which we built from source using Qt $5.9.5$.

Time Control. Each tournament was executed with the same settings, except that we ran each tournament twice, with a fast and a long time control (TC). When executing a tournament with a fast TC, each engine had 10 seconds thinking time per game, which increased by 0.1 seconds for every move ($10 + 0.1s$ TC). In comparison, when playing with a long TC, each engine had 60 seconds thinking time, with a 0.6 increment per move ($60 + 0.6s$ TC).

Opening books. To avoid starting every game from the same position, we used *opening books*. These text documents contain the most common starting positions for each variant and at the start of a game, the board is randomly set to one of these positions with a maximum depth of 5 half moves (plies). We used the standard opening books (named '*chess-variant.epd*') from Fabian Fichter[8]. Because each opening is played twice and the engines swap colors every game, each engine played each opening one time as white and another time as black.

Game settings. Each tournament was played in a round-robin fashion with 2 games per round. During the fast TC the engines played 150 rounds (= 300 games) and during the long TC 50 rounds (= 100 games). Each game was played until the end with no resign and no draw threshold. We also did not employ any tablebases, did not allow thinking on opponent's time (called *ponder*) and recovered crashed engines, which we did not need to make use of.

[7] https://github.com/cutechess/cutechess
[8] https://github.com/ianfab/books

Engine settings. During testing, MultiAra and CrazyAra used the same settings. They were allowed to use 3 threads and a single GPU and otherwise used standard settings. In particular, they used a batch size of 16 as well as a node temperature of 1.7; employed the MCTS solver and time manager, and reused the search tree during MCTS. They did not use any Dirichlet noise nor a cap on MCTS rollouts.

To test the playing strength of MultiAra, we conducted games against *Fairy-Stockfish* [10], which is the strongest actively maintained engine for *lichess variants*[9]. We compiled Fairy-Stockfish version 13.1 from source, with `bmi2` support for better performance and assigned it 4 threads. Otherwise, we used it's standard settings, except turning off ponder and setting the hash size to 2048. We did not use any efficiently updatable neural networks (NNUEs), unless explicitly specified.

4.6. Availability of MultiAra

All code of *MultiAra* is available online[10] under the terms of the GNU General Public Licence v3.0 [11]. The models trained for each variant, in conjunction with binaries of the engine, will be released soon. In particular, the *MultiAra* engine supports the Universal Chess Interface (UCI, [18]) and can be integrated in any chess GUI or program that supports UCI. Using code that bridges the gap between the Lichess API and UCI engines[11], we created a public BOT account[12], which can be played against when online.

[9] https://github.com/ianfab/Fairy-Stockfish/wiki/Playing-strength#lichess-variants
[10] https://github.com/QueensGambit/CrazyAra
[11] https://github.com/ShailChoksi/lichess-bot
[12] https://lichess.org/@/MultiAra

5. Empirical Evaluation

In this chapter, we evaluate the training process of our new engine MultiAra for each chess variant and compare it with the open source engine *Fairy-Stockfish*. Additonally, we match MultiAra versus *CrazyAra* playing Crazyhouse and analyze how game outcomes during self-play change in comparison to human games. Lastly, we report the results of a pure reinforcement learning run for King of the hill, which did not use any expert human data and compare it with the respective King of the hill model that had been initialized with a network trained on human games.

5.1. Elo Development Over Time

Every 10 model updates, we took the best model and ran a tournament with a fast and a long time control against the best model from 10 updates before. For update 10 we used the supervised model, that had only been trained on human games, as opponent instead. For more details about the tournament settings, see section 4.5. A model update does not necessarily mean model improvement. There might be some model updates that did not improve the model and did not replace the old model. However, over the course of training the models improved, otherwise we would have stopped the training process.

Training a model for the first 10 updates on 4 GPUs took 2 days. Every subsequent 10 model updates took 3 days on 4 GPUs, which amounts to around 17 days for a training run of 60 epochs.

Chess960. We were able to refine all supervised models by reinforcement leraning, except Chess960. Although games were generated, the refined Chess960 models were not able to beat the supervised model. For this reason, Chess960 is not present in any model evaluations and strength comparisons.

3check, Antichess, Crazyhouse, Horde & King of the hill. We trained 3check and Horde for 50 model updates and Antichess, Crazyhouse as well as King of the hill for 60 model updates. In total, we generated about $61 - 75$ million training samples and $4 - 5$ million validation samples for each variant (Table 5.1). In terms of games, these sum up to $0.5 - 1.2$ million games for training and $31 - 80$ thousand games for validation.

All four variants steadily improved over the course of the training process and started to converge after around 30 model updates (Figure 5.1). The Elo gain for the two different time controls remained closely connected, only diverging by a maximum of 67 Elo during the training process of King of the hill. Compared to the supervised model, the overall playing strength increased for Horde by 300, for Antichess by 340, for 3check by 400, for Crazyhouse by 480 and for King of the hill by 690 Elo. It should be noted that Antichess started to marginally decrease between update 40 and 60 in the long time control.

Atomic & Racing kings. The models of Atomic and Racing kings improved at the beginning, but stopped after the 26^{th} and 4^{th} model update, respectively. Therefore, we only executed 30 model updates for Atomic and 10 model updates for Racing kings.

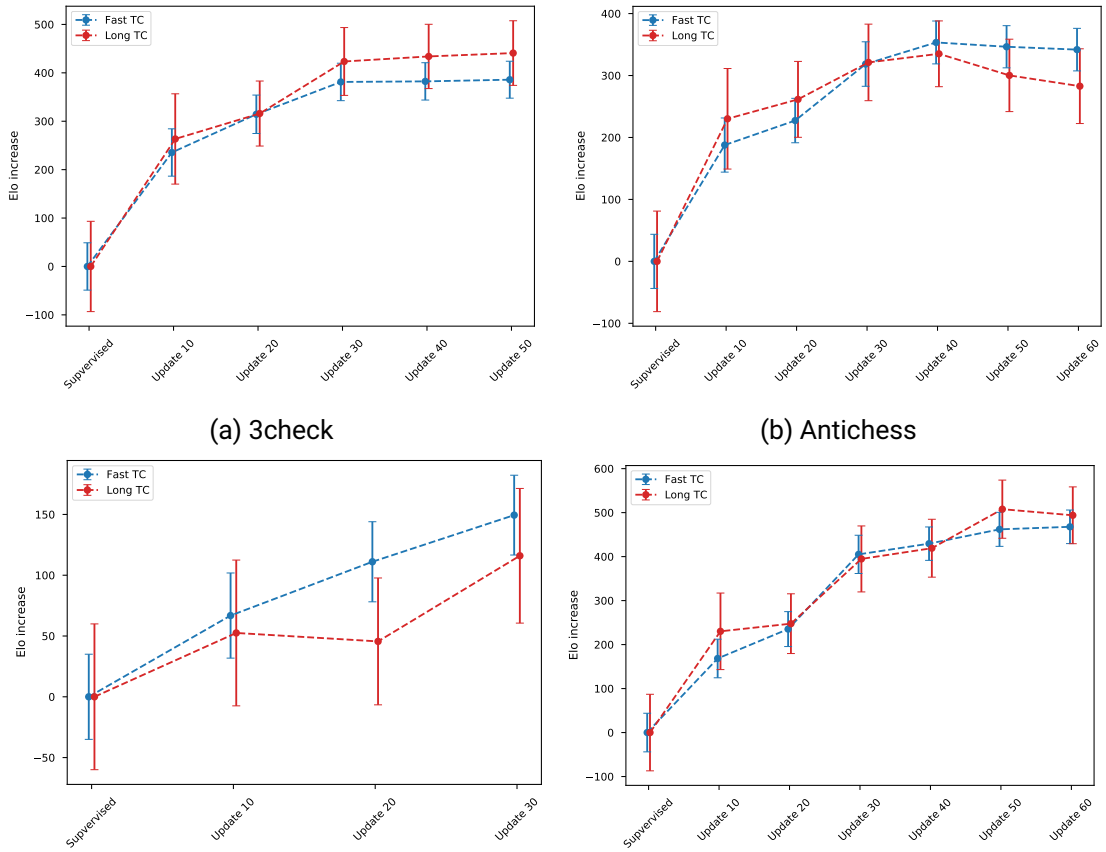
For Atomic, we generated 36 million training and 2.5 million validation samples, which translate to about 530 thousand training and 37 thousand validation games (Table 5.1). In Figure 5.1c we can see a linear Elo increase for both time controls with a downward spike at update 20 for the long time control and an overall Elo gain of 150 and 115 Elo for the fast and long time control, respectively.

The training run for Racing kings resulted in 8 million training and 0.8 million validation samples, which consist of 230 thousand training games and 23 thousand validation games (Table 5.1). Despite only four model improvements, Racing kings gained 282 Elo in the long and 349 Elo in the fast time control, when comparing it with the supervised model (Figure 5.1g).

Remarks. Although Atomic continuously improved for the first 26 model updates, we were not able to create a model that could defeat update 25. Even with a decreased learning rate, it did not manage to improve. For Racing kings, we tried to reanimate the reinforcement learning run by decreasing the learning rate by a factor of 10 and a factor of 100, without success. We also started a new training run, where we decreased the Dirichlet exploration by setting α to 0.6 instead of 0.2. This new network also stagnated after 4 model improvements and matched the original Racing kings network both in the long as well as in the fast time control (see Appendix D.3).

Variant	Model updates	Training		Validation	
		Games	Samples	Games	Samples
3check	50	1,161,458	61,276,160	78,163	4,096,000
Antichess	60	1,155,661	71,761,920	80,236	4,915,200
Atomic	30	527,772	35,799,040	36,902	2,457,600
Crazyhouse	60	994,685	74,219,520	65,994	4,915,200
Horde	50	487,411	64,143,360	31,076	4,096,000
King of the hill	60	1,125,604	73,728,000	74,962	4,915,200
Racing kings	10	229,664	8,028,160	23,245	819,200

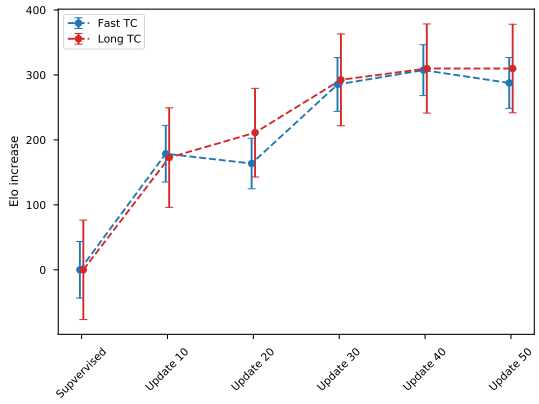
Table 5.1.: **Amount of generated games and samples.** The table displays the number of reinforcement learning model updates, training games, training samples, validation games and validation samples for each chess variant.



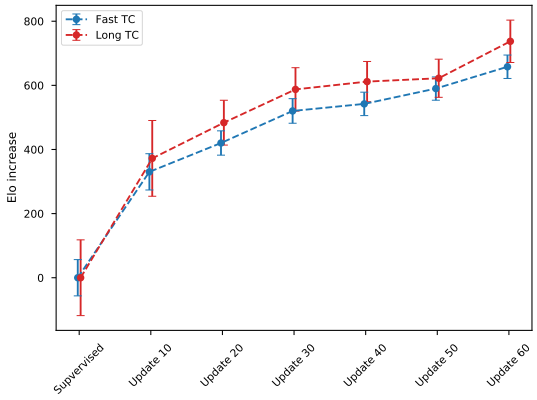
(a) 3check (b) Antichess

(c) Atomic (d) Crazyhouse

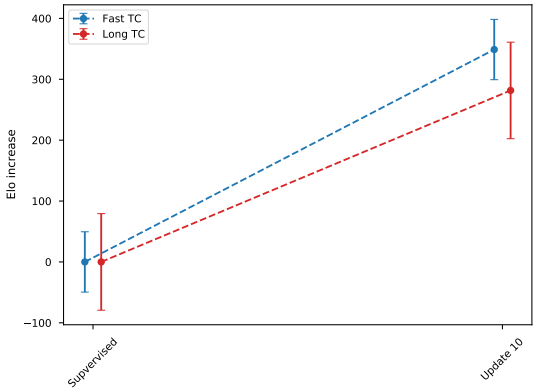
Figure 5.1.: **Chess variants Elo development**. We matched every 10^{th} model update with the model 10 updates earlier and reported the cumulative Elo gain. The Elo for the initial network, which we trained using only human games, was set to zero. During the fast/long time control (TC) each player had $10/60$ seconds for the game plus $0.1/0.6$ seconds per move.



(e) Horde



(f) King of the hill



(g) Racing kings

Figure 5.1.: **Chess variants Elo development**. We matched every 10^{th} model update with the model 10 updates earlier and reported the cumulative Elo gain. The Elo for the initial network, which we trained using only human games, was set to zero. During the fast/long time control (TC) each player had $10/60$ seconds for the game plus $0.1/0.6$ seconds per move.

5.2. Strength Comparison with Fairy-Stockfish

For each variant, we executed tournaments against *Fairy-Stockfish* 13.1 every 10 model updates. The Elo progression for the unique variants is similar to the Elo development between the individual MultiAra updates described in the last section. However, MultiAra won significantly more games against Fairy-Stockfish in the fast time control than in the long time control, when playing Horde, Crazyhouse and Atomic. Please turn to Appendix B.2 and B.3 for more details.

In total, the playing strength of MultiAra against Fairy-Stockfish (with classical evaluation) increased by around 330 Elo for 3check, 370 Elo for Antichess, 40 Elo for Atomic, 250 Elo for Crazyhouse, 210 Elo for Horde, 490 Elo for King of the hill and 226 Elo for Racing kings (Figure 5.2).

The final Elo difference between MultiAra and Fairy-Stockfish (with classical evaluation) is shown in Figure 5.3. MultiAra was able to beat Fairy-Stockfish in Crazyhouse by around 350 Elo and in Horde by circa 300 Elo. In 3check, Atomic and King of the Hill both engines were about even, but in Antichess, Chess960 and Racing kings MultiAra lost with an Elo difference of around 130, 500 and 150 Elo, respectively.

In addition, we challenged Fairy-Stockfish with NNUE evaluation. As written in Section 3.1.3, Fairy-Stockfish could improve it's playing strength for 3check, Atomic, King of the hill and Racing kings by employing *efficiently updatable neural networks* (NNUEs). Thus, when Fairy-Stockfish with it's latest NNUEs matched MultiAra, the Elo discrepancy in Racing kings increased from 130 to roughly 230. Additionally, Fairy-Stockfish now won in 3check, Atomic and King of the hill by about 65, 300 and 320 Elo, respectively. See Appendix D.2 for the exact NNUE model names that we used during testing.

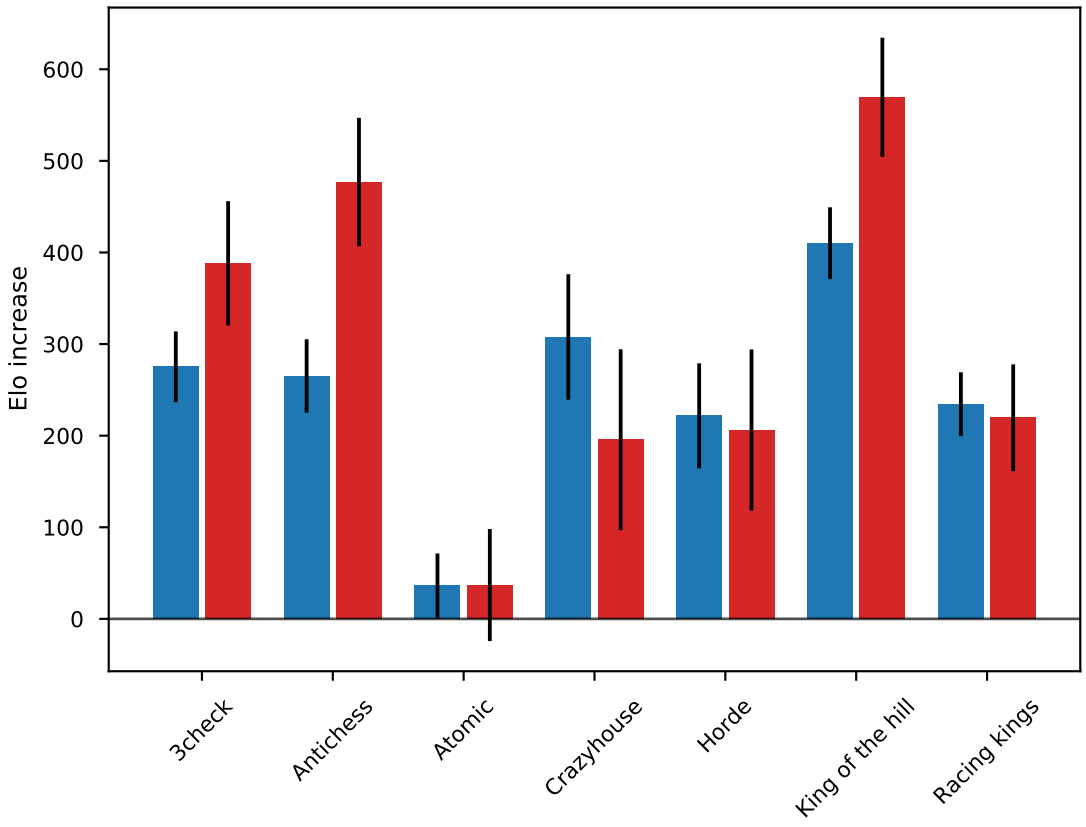
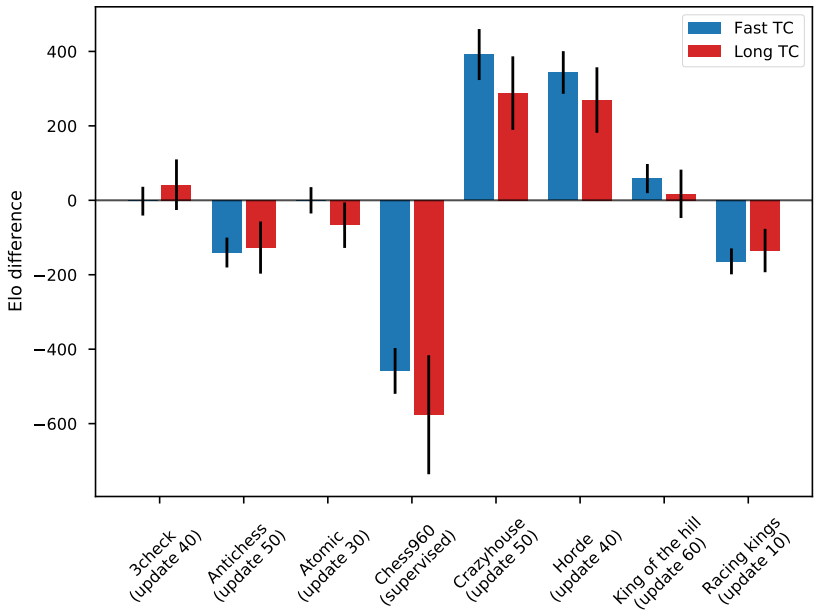
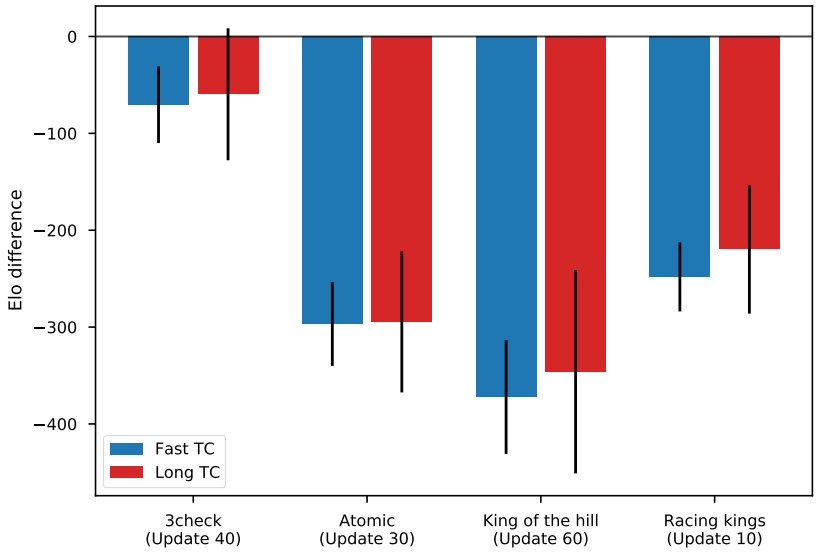


Figure 5.2.: **Elo gain of MultiAra against Fairy-Stockfish 13.1.** The Elo gain refers to Fairy-Stockfish with classical evaluation. During the fast/long time control (TC) each player had $10/60$ seconds plus $0.1/0.6$ seconds per move.



(a) Classical evaluation



(b) NNUE evaluation

Figure 5.3.: **Elo difference of MultiAra against Fairy-Stockfish 13.1.** We used the strongest models for MultiAra. During the fast/long time control (TC) each player had $10/60$ seconds for the game plus $0.1/0.6$ seconds per move.

5.3. Strength Comparison with CrazyAra

In addition to challenging *Fairy-Stockfish*, we also ran Crazyhouse tournaments against *CrazyAra* [5]. We compared every 10^{th} update of the MultiAra Crazyhouse model against the strongest CrazyAra model. The latter can be accessed online by downloading the $0.9.0$ release [4], has been trained for 96 reinforcement learning updates and constitutes a refinement of the Crazyhouse model from [5], which had been trained for 60 updates. The strength difference between the two model versions is about 60 Elo [4], which amounts to a net increase of around 430 Elo in comparison to the supervised model.

Figure 5.4 shows that our model did not manage to become as strong as the latest CrazyAra model. CrazyAra beat MultiAra with an Elo difference of 60 Elo in the fast and 80 Elo in the long time control. However, our model has a lower value and policy loss, as well as a higher value sign accuracy and policy accuracy (Table 5.2, top).

We performed an additional test between MultiAra and CrazyAra, where each binary used it's Crazyhouse supervised model, which had exclusively been trained on human games from the *lichess.org open database* [32]. MultiAra lost again with the exact same Elo difference of 60 Elo in the fast and 80 Elo in the long time control, although the supervised model exhibited a better loss and policy accuracy (Table 5.2, bottom).

Engine	Model	Value loss	Policy loss	Value sign acc.	Policy acc.
CrazyAra	Update 96	0.44052	1.08962	0.777 %	0.768 %
MultiAra	Update 49	0.33402	0.99867	0.835 %	0.793 %
CrazyAra	Supervised	1.19246*		-	0.603 %
MultiAra	Supervised	1.16293*		-	0.612 %

Table 5.2.: **Model comparison between MultiAra and CrazyAra.** Models refer to the chess variant Crazyhouse and the metric *value sign accuracy* counts the fraction of correctly predicted game outcomes. *Supervised models have a combined loss of 1% value and 99% policy loss.

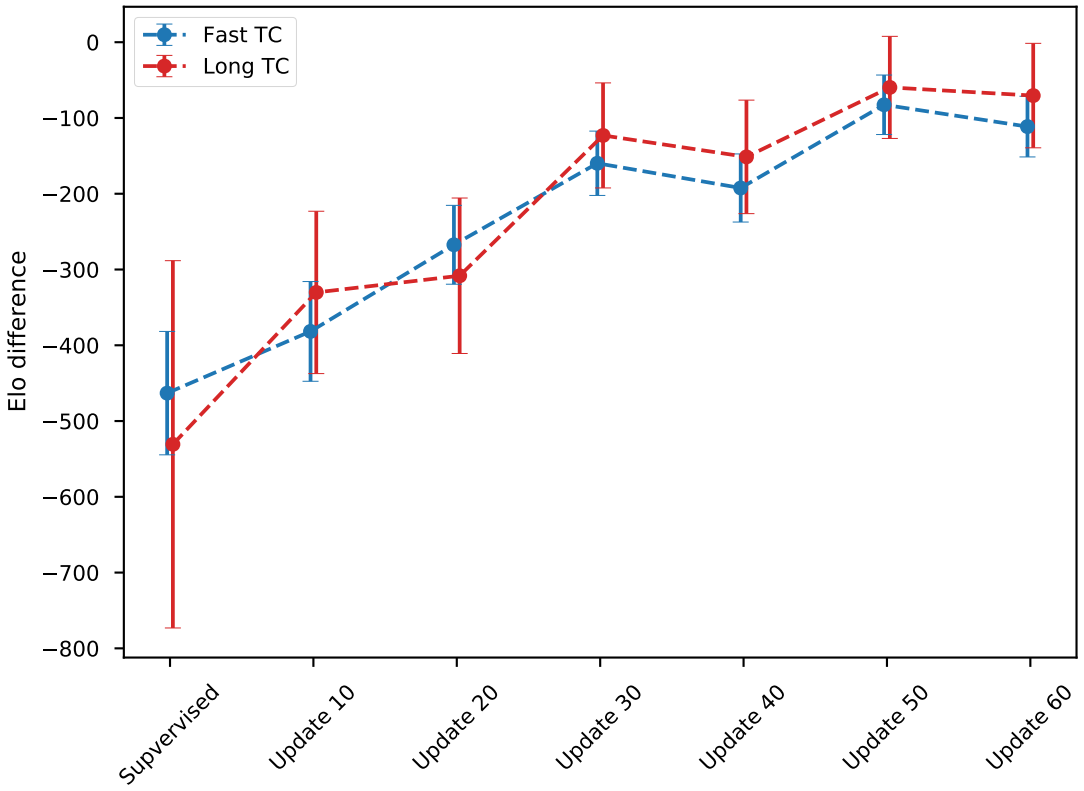


Figure 5.4.: **Elo difference between MultiAra and CrazyAra.** Calculated for every 10^{th} model update of MultiAra. CrazyAra uses it's strongest model for Crazyhouse. Both engines are build with the same commit and use the same settings. During the fast/long time control (TC) each player had $10/60$ seconds for the game plus $0.1/0.6$ seconds per move.

5.4. Self-Play Game Outcomes

In this section we elaborate the difference between human games and games generated during self-play. The statistics for human games have been calculated by using the data from the *lichess.org open database* [32] as described in Section 4.3.1. For self-play statistics, we used the last 50 packages generated during each variant's reinforcement learning run. Each package contains 81920 samples.

Game length. Because every variant has (slightly) different rules, the average game length varies. For Crazyhouse, an average human game has a length of 55 plies (half moves), which increases to around 100 plies for Horde and decreases to 30 plies for Racing kings. Figure 5.5 shows the average game length for each variant, both for human games as well as for self-play games. We can see that the order of the chess variants is largely the same. Only Atomic seems to take longer than Antichess during self-play, whereas it is shorter than Antichess for human games. Further, the games for each variant are on average 24 plies longer during self-play than during human play.

Game outcomes. In Figure 5.6, we compare the average percentage of games that end with a draw, white win or black win for human games (top) and for self-play games (bottom). In human games, draws are very unlikely for all *lichess variants*. Crazyhouse, King of the hill and 3check nearly never end in a draw, and Chess960, Atomic, Horde and Antichess are drawn with around $4 - 6\%$. Only Racing kings regularly ends in a draw with about 15%, due to it's special draw rule. During self-play, the draw rate is amplified for all variants, except Horde. Especially Racing kings ends most of the time in a draw with approximately 55%.

Nearly all *lichess variants* are white favored, with the exception of Horde. For human games, 3check, Atomic and Crazyhouse have the highest percentages with a difference of 16%, 15% and 10% between white and black wins, which shrinks to 7% for King of the hill and $4 - 5\%$ for Antichess, Chess960 and Racing kings. Horde is a special case, as both colours have different task allocations and black is clearly in favor with 54% to 42%. When compared with the game outcome of self-play data, the differences between white winning and black winning get intensified. This is particularly noticeable for 3check, where white wins in 73% of the games and for Horde, which ends in 82% of the games with a black win.

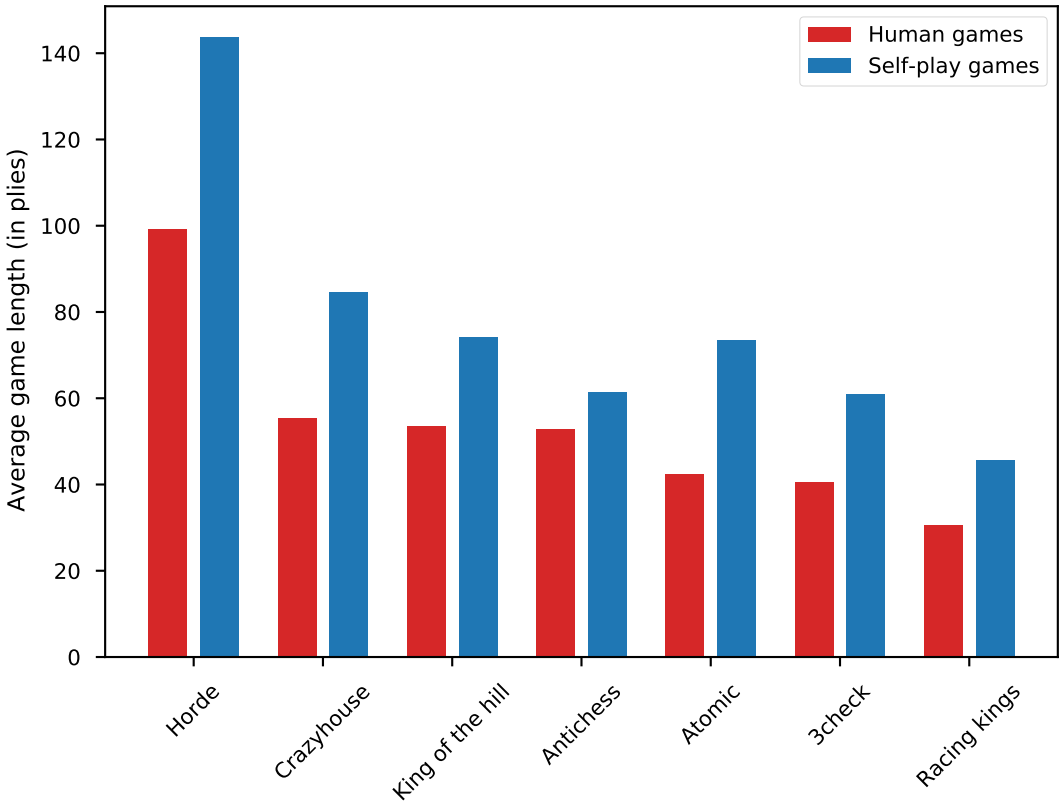
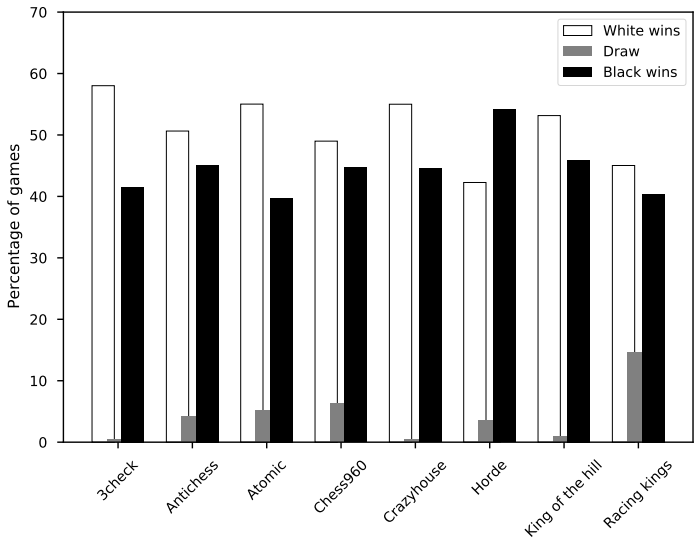
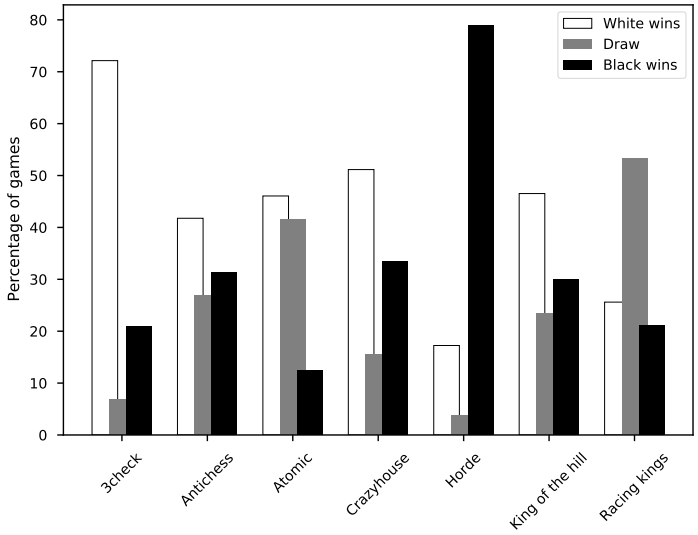


Figure 5.5.: **Average game length of human vs self-play games.** Self-play games were generated during reinforcement learning and we took the last 50 packages (each containing 81920 samples) produced for each variant. The statistics for human games have been calculated by using the data from the *lichess.org open database* [32] as described in section 4.3.1. The average game length is measured in half moves (plies).



(a) Human games



(b) Self-play games

Figure 5.6.: **Game outcome of human vs self-play games.** The statistics for human games have been calculated by using the data from the *lichess.org open database* [32] as described in section 4.3.1, and for self-play games we used the last 50 packages (each containing 81920 samples) generated during reinforcement learning.

5.5. Training From Zero knowledge

Besides the King of the hill model that used a supervised model as starting point (*SL-model*), we also trained another King of the hill model, but this time from zero knowledge (*model-from-zero*). We trained the model for 80 Updates, which took 26 days on 4 GPUs. In total, we produced $1,587,943$ training games and $102,727,680$ training samples, as well as $104,574$ validation games and $6,717,440$ validation samples.

Over the course of the reinforcement learning process, the randomly initialized model gained 800 Elo in the fast and 1000 Elo in the long time control (Figure 5.7). At the beginning, the fast and long time controls stayed close together, but at update 30 the model gained 400 Elo in the long, but only 200 in the fast time control. This is similar to the *SL-model*, which gained 40 Elo less at update 10 in the fast time control than in the long time control (Figure 5.1f).

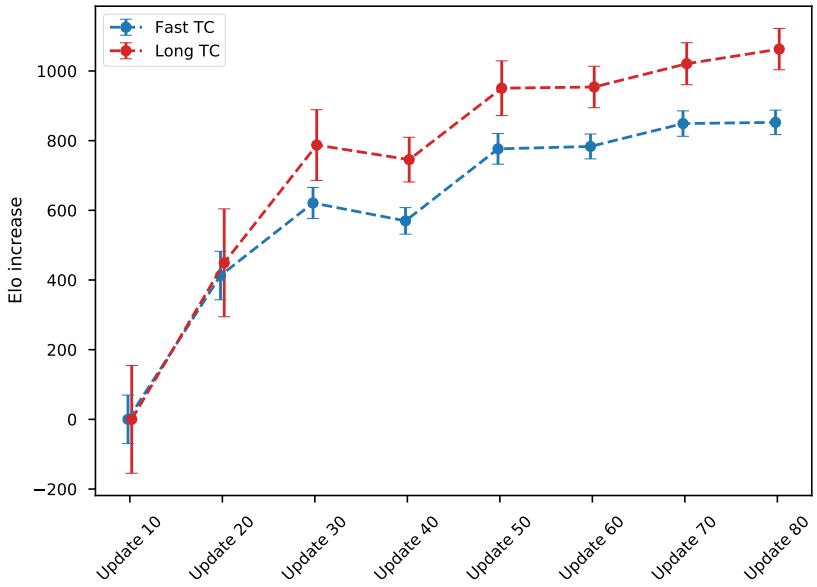


Figure 5.7.: **Elo development for King of the hill from zero knowledge.** The model started with random parameters and the plot shows the cumulative Elo gain. During the fast/long time control (TC) each player had $10/60$ seconds for the game plus $0.1/0.6$ seconds per move.

Further, we measured the Elo difference between each 10^{th} model update of the *SL-model* and the *model-from-zero*. To clarify, we ran a tournament between update 10 of the *SL-model* and update 10 of the *model-from-zero* and repeated this procedure every 10 updates. We can see in Figure 5.8 that the *SL-model* beats the *model-from-zero* by 600 Elo at update 10. Over the next 50 updates, the Elo difference between the two models shrank, until the *model-from-zero* was only 190 Elo weaker than the *SL-model* at update 50. This means that the *model-from-zero* had a higher net Elo increase per update than the *SL-model*. Only between update 50 and 60 the *SL-model* gained more Elo than the *model-from-zero*, because the latter virtually stagnated due to an insufficient amount of training samples (for more details, see section 4.4.4).

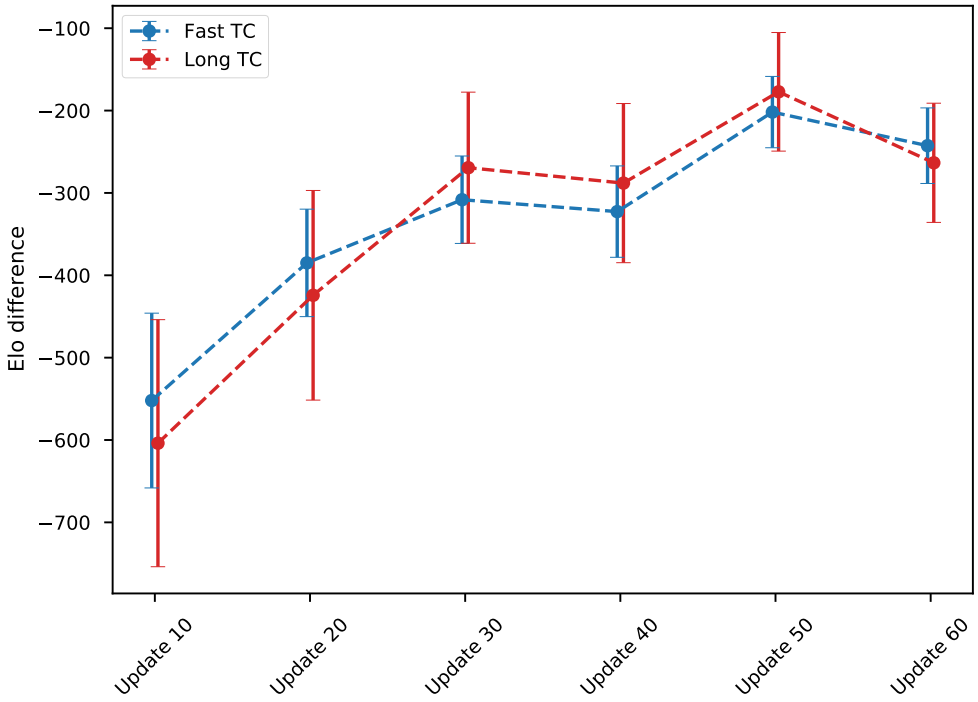


Figure 5.8.: **Zero knowledge vs initialization with human games.** Elo difference between every 10^{th} model update of the King of the hill *SL-model* and *model-from-zero*. The former has been initialized with a network trained on human games, and the latter with random weights. During the fast/long time control (TC) each player had $10/60$ seconds for the game plus $0.1/0.6$ seconds per move.

When we compare every 10^{th} model update of the *model-from-zero* with the strongest checkpoint of the *SL-model*, which was at update 60, we notice that the *model-from-zero* has difficulties becoming as strong as the *SL-model* (Figure 5.9). Although the *model-from-zero* had enormous Elo gains during the first 50 updates, it more or less stagnated afterwards and remained around 220 Elo weaker than the *SL-model* with only marginal Elo growth.

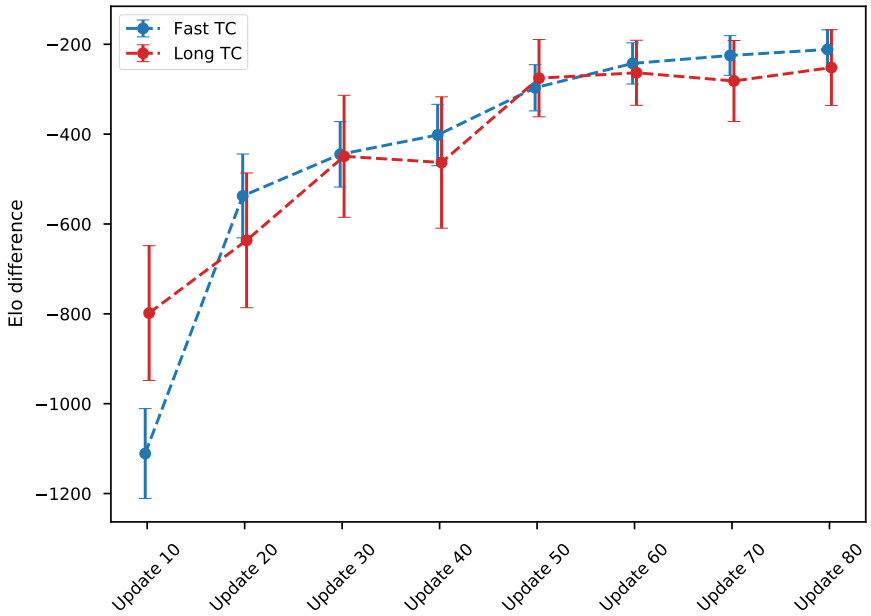


Figure 5.9.: **Playing strength of King of the hill from zero knowledge.** We compared the *model-from-zero* with the best performing *SL-model* checkpoint. The former has been initialized randomly and the latter by a network trained on human games. For all tournaments we used update 60 of the *SL-model* as opponent. During the fast/long time control (TC) each player had $10/60$ seconds for the game plus $0.1/0.6$ seconds per move.

Furthermore, we calculated the game outcome during self-play for the King of the hill *model-from-zero*. The average game length is exactly the same as in self-play of the *SL-model*, with 74 half moves (plies). In addition, the draw rate is with 23% only one percent lower as for the *SL-model*, but the discrepancy between white and black wins is not as amplified with only 12% instead of 19%. As a reminder, human King of the hill games have virtually no draws and white wins only 7% more games than black.

6. Discussion

In this chapter we discuss how practical a single reinforcement learning setup is to train multiple chess variants, whether including last moves in the input representation yields an advantageous attention mechanism, how self-play game outcomes can help us balance certain chess variants and whether training from zero knowledge improves the playing strength of chess variants.

6.1. General Reinforcement Learning Setup

The empirical evaluation in section 5 clearly shows, that the RISEv2 mobile network architecture and the reinforcement learning setup from Czech [5] is suitable to train different chess variants. This supports Google DeepMind's insight that one architecture can be used to train multiple games [26].

However, the results also show that it is not trivial to train eight different chess variants using a fixed set of parameters and a predefined learning rate annealing. Section 5.1 revealed that we could not apply reinforcement learning to Chess960 at all and that training stopped for Atomic after update 25 and for Racing kings after update 3. The Elo of Antichess even slightly decreased during the last 30 model updates. We noticed that different chess variants need different learning rate intervals. Sometimes we reduced the learning rate, even though the variant was making huge progressions with the old learning rate. On the other hand, sometimes the model did not improve for several updates, until it finally reached the point, where we reduced the learning rate. We think that Atomic, Chess960 and Racing kings converged prematurely and even the other variants might benefit from individualized training configurations. We are almost certain that the playing strength of several variants would increase even further if we train each variant with individual hyperparameters again.

This is in contrast with DeepMind's findings, who used the same reinforcement learning setup to train a separate network for Go, chess and shogi [26]. But DeepMind's algorithm differs in two significant ways: it starts from zero knowledge and scales the applied exploration by the number of legal moves. Especially the latter is worth studying in the future. If we think about chess variants like Antichess, where there are often only one, two or three legal moves, or Racing kings, which only has half of the regular pieces, a scaling of the exploration noise is definitely worth testing.

In the future, we would like to see especially Chess960 get up and running. Chess960 needs less or even no exploration due to the 960 different starting positions. We tried setting all parameters in a way that provided as less exploration as possible, but were not able to get it to run. For MultiAra's sibling *ClassicAra*, which is build on the same architecture, but only plays standard chess, we tried to start reinforcement learning self-play for standard chess as well, but also failed. Both cases might very well be related. For chess and for Chess960 the amount of human games available is much higher than for chess variants. It is possible that the initial supervised models, which have only been trained with human games, are already so powerful, that they always beat the reinforcement learning models. Maybe the remedy would be to use more than 820 thousand training samples per update. While training King of the hill from zero (Section 5.5), the model stagnated between the 50^{th} and 60th model update, but could be reanimated by using more training samples for a model update. Usually 1.3 million samples are used at this stage of the training process, which we increased to at least 1.7 million training samples. Another idea would be to start training from zero and skip training a model with expert data all together.

An extension that we highly recommend, is to run multiple reinforcement learning self-play processes on one GPU. At the moment, we only use less than 950 Megabits of 32 Gigabit GPU memory. We also recommend using a bigger batch size than eight, maybe even connecting both approaches. As reported in section 5.1, the current setup needs roughly 17 days to create around 75 Million samples with 4 GPUs. In comparison, DeepMind trained the chess model of *AlphaZero* for 2.8 billion samples in 9 hours and the model only started to converge after 820 million samples [26]. A bigger throughput would save precious training time and on the other hand increase GPU utilization.

6.2. Last Moves as Attention Mechanism

The motivation to include the last eight moves in the input representation, was to add an attention mechanism. Although chess and it's variants can be formulated as a *Markov*

Decision Process, which means that in theory the engine does not need the move history to calculate the best move, we still wanted to evaluate, whether last moves might be helpful to find better strategies. Many chess GUIs, for example *lichess.org* or *tcec-chess.com*, have already implemented this feature by visually highlighting the last move, which seems to help human players focus their attention.

MultiAra and *CrazyAra* [5] both use the same RISEv2 architecture, with the exception that MultiAra uses a different input representation. This means that we can compare the playing strength of the two networks and draw a conclusion about the usefulness of last moves. In comparison to CrazyAra's 34 input planes, MultiAra uses 63 planes which include variant specific information, but also 16 planes representing the last eight moves.

The parameter settings to train models for MultiAra and CrazyAra were largely the same, with negligible differences in the reinforcement learning schedule and game generation configurations (e.g. turning off draw adjudication). Both engines roughly used the same human expert data to generate a supervised model as a starting point for reinforcement learning and the binaries that we used during the tournaments have been created with the same up-to-date commit[1]. Both binaries share large portions of their code, with the exception that MultiAra needs to take care of all lichess variants. This results in dozens of queries to determine which variant is currently active, when processing rules or variant specific conditions.

We ran performance tests between CrazyAra and MultiAra with the conclusion that both engines process virtually the same number of nodes per seconds (Appendix D.1). In section 5.3 we have seen that the Crazyhouse model for MultiAra exhibits a lower loss and a higher policy accuracy than CrayzAra's model. From our experience, a model that has a better loss and policy accuracy, while the nodes per second remain the same, should be stronger. Therefore, it is very surprising that MultiAra performs worse than CrazyAra.

However, we still advocate that the new input representation for Crazyhouse, which includes the last eight moves, is not necessarily worse. It is intriguing, that the Elo difference between both engines stayed constant, when comparing the supervised models and best performing models of each engine. In particular, it is possible that the Elo difference is caused by an unobserved artifact that has not been found yet.

In conclusion, we think that attention mechanisms, provided by including last moves, might be advantageous for chess variants. Especially, if we look at the lower loss and increased policy accuracy, as well as take into account that MultiAra has only trained it's Crazyhouse model for 60 updates in comparison to CrazyAra's 96 updates.

[1] https://github.com/QueensGambit/CrazyAra/commit/1ea250b20b54f3eb5db3615e3dc450ae4319fb82

6.3. Self-Play Game Outcomes

In section 5.4 we saw a comparison between game statistics of human games and games created during reinforcement learning self-play. The average length of a game in self-play is higher than in human games, which can be explained by the missing of draw and resign adjudications during self-play. This is also a possible explanation, why so many games in Atomic are drawn in self-play games compared to human games. Humans often resign, if they see that a game is in favor of the opponent, especially on online platforms where the next game can be started immediately.

In addition, the incurred amplification of game outcomes shows the imbalance of chess variants. It is not easy to balance games and chess variants are no exception. In this context, engines can be very beneficial, because they try to master games irresistibly. We can see that small differences between the win ratio of black and white grow to huge advantages, if engines play the game (Figure 5.6). The small advantage that a player has, for example by moving first or by having slightly better pieces, is inexorably exploited by engines.

Especially 3check and Horde are heavily imbalanced and favoured for white and black, respectively. Racing kings is approximately balanced, but ends in a draw in half of the games. Hence, Racing kings can quickly become boring. However, we can use this information to know which variants we need to change in which direction to achieve better game balance. DeepMind recently did exactly that for standard chess [33]. For 3check, one idea would be, that black only has to give chess twice to win the game and for Horde we could try to give white two more pawns. In Racing kings, we could remove the addional draw rule and give black an edge by placing a single piece, maybe a knight, further forward. It should be fairly easy to implement these variants in MultiAra's ecosystem and retrain the variant via reinforcement learning.

The extremely favoured game outcomes are also an explanation why MultiAra tends to play weaker in endgames, like in Horde when winning as white (see Appendix A.2 for example games). During self-play, MultiAra wins 82% of the Horde games as black and simply does not generate enough examples to learn how to play in white favoured endgames. There are several fixes for this problem. One idea is to implement tablebases and incorporate the knowledge whether we have found a tablebase position during the search in our decision on the next move. Likewise, it would also be feasible to train the network on a different loss, where it is beneficial to win the game as soon as possible.

A different solution would be to use expert networks that are trained separately regarding the number of pieces that are left on the board. During a tournament, the engine would need to switch seamlessly between the networks. Also, we could use larger convolutions for the endgame expert network, maybe even 8×8 convolutions, to be able to recognize structures that take up the whole board.

Another idea would be to use opening books during self-play. This way, the engine does not always start from the same position and learns to play a game starting from an unusual position. This addition would almost certainly boost MultiAra's performance against other engines, as tournament games are nearly always started from an opening book position.

6.4. Training From Zero Knowledge

We can see from our results in section 5.5, that the King of the hill model that we trained from zero knowledge (*model-from-zero*) stagnated during the last 30 model updates, while being around 220 Elo weaker than the model that had been initialized with a network trained on human games (*SL-model*). Even when adding exactly the amount of training time that the *model-from-zero* needed to beat a supervised model that had been purely trained on human games, it did not get as strong as the *SL-model*. We can conclude, that at least for our setup, training a model from zero knowledge is not trivial and does not gain the same playing strength than a *SL-model* in a reasonable amount of time.

Of course the term *reasonable amount of time* is open for interpretation. We trained the *model-from-zero* for 80 model updates, which took 26 days on 4 NVIDIA Tesla V100 GPUs. Depending on the amount of resources, a model trained on human games might be advantageous, especially because training such a model only takes several hours on a single GPU. However, our results do not give us a definitive answer, whether training from zero knowledge cannot yield networks that have an increased playing strength than networks that are initialized with a supervised model, as suggested by DeepMind [26].

Interestingly, the inferiority of the *model-from-zero* to the *SL-model* is also reflected by the black and white win rates during self-play. White only wins 12% more games than black when looking at the self-play games for the *model-from-zero*, instead of 17% during self-play of the *SL-model*. This means that the *SL-model* can exploit the advantage that white has in King of the hill much better than the *model-from-zero*.

7. Conclusion

7.1. Summary

In this thesis we trained a single Monte Carlo tree search engine that can play eight chess variants. Our program *MultiAra* can be downloaded and used with traditional chess GUIs to play against, train your own strength or analyze games. To the best of our knowledge, it is the only multi-variant chess engine that uses Monte Carlo tree search and the strongest engine for the chess variant Horde.

We showed that it is possible to train seven out of eight chess variants from *lichess.org* by using the exact same input representation, neural network architecture and reinforcement learning setup. Only the variant Chess960 could not be trained by reinforcement learning. Further, we confirmed that last moves as attention mechanism can indeed be advantageous for the chess variant Crazyhouse and suggested some changes to improve the game balance of 3check, Horde and Racing kings.

Lastly, we studied whether a pure reinforcement learning model, started from zero knowledge, can surpass the playing strength of a model, which used a network trained on human games as a starting point, and concluded that the net Elo gain per update is higher, but the effective playing strength lower. However, this topic clearly requires more research to find a definitive answer.

7.2. Future Work

Implement hyperparameter tuning. In Section 6.1 we discussed the advantages of individual hyperparameters for each chess variant. Therefore, we suggest exploring the parameter space of each chess variant with *Bayesian Optimization* [30] and selecting the parameters that perform best. Another good idea is the introduction of *population based training* [17], which selects the best values of certain parameters each time the neural network is optimized from a predefined range of values.

Explore attention mechanisms. To be able to make a save statement, whether attention mechanisms are useful for chess or chess variants, more research is needed. However, attention mechanisms cannot only be included by adding last moves to the input representation, but also by using methods like *Long short-term memory* [14] or *Transformer* [34] networks. An additional advantage of the latter is it's generality to focus the attention on any aspect that helps to make better decisions about the next move.

Upgrade to newer architecture. In this thesis, we used the RISEv2 mobile architecture [5], which has recently been upgraded to RISEv3. The improved architecture is around 150 Elo stronger[1], but has only been tested with a supervised model in standard chess yet. However, we expect similar improvements for chess variants, refined by reinforcement learning.

Include a WDLP head. Similar to *Leela chess zero*[2], Johannes Czech recently added a win, draw, loss and plies-to-end (WDLP) head to the CrazyAra repository[3]. This head can be used to replace the value head for both, the RISEv2 and RISEv3 architecture. A WDLP head estimates the probability of winning, drawing or losing, and outputs an estimate of how many plies are needed to finish the game. Both, the probability of winning, losing or drawing, as well as the estimated amount of plies that are needed to end the game, are great attention mechanisms, which could lead to an improvement in playing strength. Especially the latter might lead to a significant Elo gain due to our observations that MultiAra is for some variants significantly worse in the late game than in the early game (see Appendix A for example games). A plies-to-end counter, could signalize the neural network, whether it is in the endgame. First test results on a supervised model for standard chess show an approximate gain of 50 Elo.

[1] https://github.com/QueensGambit/CrazyAra/releases/tag/0.9.3
[2] https://lczero.org/blog/2020/04/wdl-head/
[3] https://github.com/QueensGambit/CrazyAra/pull/123

Add variant specific features. In this work we used the same input representation for all chess variants. Although this yielded good results, we think that an improved, variant specific input representation might help the network to determine what is important. For example, the number of checks, which are currently present in the input representation, but only really needed by 3check, can be deleted for other chess variants to avoid misguided attention. Features to add could be the position of the castling rooks for Chess960, a plane highlighting the center for King of the hill and a plane highlighting the last row for Racing kings, among others.

Train separate Horde networks. Horde is a chess variant with different pieces and goals for white and black. It might be beneficial to train two separate networks, one playing only the white and the other playing only the black pieces. If we look at Horde endgames, when MultiAra is playing white (Appendix A.2), we can see that MultiAra is not sure what to do. In particular, when the white pawns promote, MultiAra is not sure how to use them. Training two separate networks should clarify the roles of black and white and lead to a higher playing strength.

Aknowledgements: The author thanks Johannes Czech for providing, sharing, and explaining the *CrazyAra* code. His supervision of this thesis was extraordinary and without his continuous input, help and expertise, this work would not have been possible. The author also thanks Thibault Duplessis for publishing the *lichess.org open database*, and Fabian Fichter and Daniel Dugovic for the freely available chess engines *Fairy-Stockfish* and *Multi-Variant Stockfish*. He also appreciates the support of the TU Darmstadt Machine Learning Lab by overseeing this thesis and granting access to two DGX-2 server instances, and is grateful for the permission of Jannis Blüml to use his homemade logo. Lastly, the author thanks his family for the perfectly splendid moral and professional assistance.

Bibliography

[1] Jonathan Baxter, Andrew Tridgell, and Lex Weaver. “Learning to play chess using temporal differences”. In: *Machine Learning* 40.3 (2000), pp. 243–263.

[2] Alan Bernstein et al. “A chess playing program for the IBM 704”. In: *Proceedings of the May 6-8, 1958, western joint computer conference: contrasts in computers*. 1958, pp. 157–159.

[3] Aleksandar Botev, Guy Lever, and David Barber. “Nesterov’s accelerated gradient and momentum as approximations to regularised update descent”. In: *2017 International Joint Conference on Neural Networks (IJCNN)*. IEEE. 2017, pp. 1899–1903.

[4] Johannes Czech. *CrazyAra & ClassicAra 0.9.0*. URL: `https://github.com/QueensGambit/CrazyAra/releases/tag/0.9.0` (visited on 07/07/2021).

[5] Johannes Czech. “Deep Reinforcement Learning for Crazyhouse”. M.Sc. TU Darmstadt, Dec. 2019, p. 54.

[6] Johannes Czech, Patrick Korus, and Kristian Kersting. “Monte-Carlo Graph Search for AlphaZero”. In: *arXiv preprint arXiv:2012.11045* (2020).

[7] Johannes Czech et al. “Learning to Play the Chess Variant Crazyhouse Above World Champion Level With Deep Neural Networks and Human Data”. In: *Frontiers in artificial intelligence* 3 (2020), p. 24.

[8] Omid E David, Nathan S Netanyahu, and Lior Wolf. “Deepchess: End-to-end deep neural network for automatic learning in chess”. In: *International Conference on Artificial Neural Networks*. Springer. 2016, pp. 88–96.

[9] Daniel Dugovic. *Multi-Variant Stockfish*. URL: `https://github.com/ddugovic/stockfish` (visited on 07/08/2021).

[10] Fabian Fichter. *Fairy-Stockfish*. URL: `https://github.com/ianfab/Fairy-Stockfish` (visited on 07/08/2021).

[11] Free Software Foundation. *gnu.org*. 2007. URL: https://www.gnu.org/licenses/gpl-3.0.en.html (visited on 07/07/2021).

[12] Dongyoon Han, Jiwhan Kim, and Junmo Kim. "Deep pyramidal residual networks". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2017, pp. 5927–5935.

[13] Kaiming He et al. "Deep residual learning for image recognition". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778.

[14] Sepp Hochreiter and Jürgen Schmidhuber. "Long short-term memory". In: *Neural computation* 9.8 (1997), pp. 1735–1780.

[15] Andrew G Howard et al. "Mobilenets: Efficient convolutional neural networks for mobile vision applications". In: *arXiv preprint arXiv:1704.04861* (2017).

[16] Jie Hu, Li Shen, and Gang Sun. "Squeeze-and-excitation networks". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2018, pp. 7132–7141.

[17] Max Jaderberg et al. "Population based training of neural networks". In: *arXiv preprint arXiv:1711.09846* (2017).

[18] Stefan-Meyer Kahlen and Harm Geert Muller. *UCI protocol*. 2004. URL: http://wbec-ridderkerk.nl/html/UCIProtocol.html (visited on 07/07/2021).

[19] Larry Kaufman. *Komodo*. URL: https://komodochess.com/ (visited on 07/07/2021).

[20] Matthew Lai. "Giraffe: Using deep reinforcement learning to play chess". In: *arXiv preprint arXiv:1509.01549* (2015).

[21] Yann LeCun, Yoshua Bengio, et al. "Convolutional networks for images, speech, and time series". In: *The handbook of brain theory and neural networks* 3361.10 (1995), p. 1995.

[22] Alexandru Moșoi. *Zurichess*. URL: https://bitbucket.org/brtzsnr/zurichess/src/master/ (visited on 07/07/2021).

[23] Yu Nasu. "Efficiently updatable neural-network-based evaluation functions for computer shogi". In: *The 28th World Computer Shogi Championship Appeal Document* (2018).

[24] David Brine Pritchard. *The encyclopedia of chess variants*. Games & Puzzles Publications, 1994.

[25] Mark Sandler et al. “Mobilenetv2: Inverted residuals and linear bottlenecks”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2018, pp. 4510–4520.

[26] David Silver et al. “A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play”. In: *Science* 362.6419 (2018), pp. 1140–1144.

[27] David Silver et al. “Mastering the game of Go with deep neural networks and tree search”. In: *nature* 529.7587 (2016), pp. 484–489.

[28] David Silver et al. “Mastering the game of go without human knowledge”. In: *nature* 550.7676 (2017), pp. 354–359.

[29] Leslie N Smith and Nicholay Topin. “Super-convergence: Very fast training of neural networks using large learning rates”. In: *Artificial Intelligence and Machine Learning for Multi-Domain Operations Applications*. Vol. 11006. International Society for Optics and Photonics. 2019, p. 1100612.

[30] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. “Practical bayesian optimization of machine learning algorithms”. In: *Advances in neural information processing systems* 25 (2012).

[31] Richard S Sutton, Andrew G Barto, et al. *Introduction to reinforcement learning*. Vol. 135. MIT press Cambridge, 1998.

[32] Thibault Thibault. *lichess.org open database*. Jan. 2013. URL: `https://database.lichess.org/` (visited on 07/07/2021).

[33] Nenad Tomašev et al. “Assessing game balance with AlphaZero: Exploring alternative rule sets in chess”. In: *arXiv preprint arXiv:2009.04374* (2020).

[34] Ashish Vaswani et al. “Attention is all you need”. In: *Advances in neural information processing systems*. 2017, pp. 5998–6008.

[35] Joel Veness et al. “Bootstrapping from Game Tree Search.” In: *NIPS*. Vol. 19. 2009, pp. 1937–1945.

Appendices

A. Example Games

A.1. Atomic

[Event "Casual Atomic game"]
[Site "https://lichess.org/bU8oIApH"]
[Date "2021.07.11"]
[White "MultiAra"]
[Black "Ilmare"]
[Result "1-0"]
[Variant "Atomic"]
[TimeControl "180+2"]

1. Nh3 f6 2. e3 d5 3. Ng5 Bg4 4. f3 fxg5 5. Na3 b5 6. d4 Nd7 7. g3 Ndf6 8. h4 e6 9. Bh3 Ng4 10. Bxg4 h5 11. e4 g6 12. O-O Bh6 13. Bg5 Nf6 14. Qd2 O-O 15. Kh1 Kh8 16. Bxh6 Ng8 17. Qa5 c5 18. Qa6 g5 19. Qd6 Qxd6 20. exd5 gxh4 21. Rad1 Nh6 22. dxc5 Nf5 23. c4 h4 24. Rd7 Rf7 25. Rd8+ Rf8 26. c5 Kh7 27. Rfd1 hxg3 28. Rb8 Ng3+ 29. Kg1 Kg6 30. Rdd8 Ne2+ 31. Kh2 a5 32. Nc2 Ra7 33. Rd7 Kg5 34. Rxf8 Ra8 35. Rd8 Ra7 36. Rg8+ Kf4 37. Rg7 Ra8 38. Rg4+ Kg3 39. Rg8 Ra7 40. Rg7 Ra8 41. c6 Rh8 42. Rh7 Rg8 43. c7 Kf2 44. Kg2 Nf4 45. c8=Q Ke2+ 46. Kf2 Kd2 47. Rg7 Rh8 48. Ke1 Rh2 49. Rg2 Rh1 50. Rg1 Rh2 51. Ke2 Rf2 52. Kd3 Re2 53. a4 Re3 54. Nd4 Ke2 55. Rf1 Kd2 56. b4 e5 57. Kc2 Rc3 58. Qb8 Ke3+ 59. Kd2 Ke2 60. Nxb5 Rc2 61. b5 Kd1 62. b6 Kc1 63. b7 Kb2+ 64. Kc3 Nd3 65. Qd8 Nc5 66. Qd5 e4 67. f4 e3 68. Re1 e2 69. f5 Nd3 70. Rxe2 Kc1+ 71. Kb2 Kb1 72. f6 Ka2 73. f7 Ka1 74. Ka2 Rb2 75. Kb1 Rc2 76. Kb2 Rc3 77. Ka2 Rc2 78. Qc5 Rb2 79. b8=Q Rb3 80. Kb1 Rb2 81. Kc1 Rc2+ 82. Kb2 Rc3 83. Kb1 Rc1 84. Kb2 Rc3 85. Qd4 Rc2 86. Ka3+ Rb2 87. Qd1+ Ka2 88. Kb3 Rc2 89. Kb2 Rc3 90. f8=Q Ra3 91. Kb1 Ka1 92. Qb2 Rc3 93. Qxc3 Kb2 94. Qa3 Ka2 95. Qb2 Ka1 96. Qdc1 Ka2 97. Ka1 Kb1 98. Qa2 Kb2 99. Qcb1 Kc3 100. Qab2+ Kc4 101. Qd4# 1-0

Figure A.1.: **Atomic game 1.** Online against Bot Ilmare. MultiAra plays white and has a clear winning position after move 46 with a promoted queen. However, it needs two more queen promotions to checkmate.

[Event "Casual Atomic game"]
[Site "https://lichess.org/YSGnvtnv"]
[Date "2021.07.11"]
[White "MultiAra"]
[Black "Ilmare"]
[Result "1-0"]
[Variant "Atomic"]
[TimeControl "180+2"]

1. e4 e6 2. d4 Nh6 3. f3 Na6 4. Nh3 Nb4 5. Ng5 f5 6. Qe2 Nd3+ 7. Qxd3 fxe4 8. Na3 Bb4+ 9. c3 b5 10. Nxh7 Qh4+ 11. g3 Qh7 12. h4 Ba5 13. b4 Bb6 14. h5 Ba6 15. Bg5 Qf5 16. g4 Qf8 17. Nxb5 c5 18. Bd2 cxd4 19. Bf4 e5 20. Rd1 d5 21. Bc1 O-O-O 22. h6 g5 23. h7 Qh8 24. Ba6+ Kb8 25. Rh6 Qxh7 26. Rd2 d4 27. Rh2 e4 28. Rh8 d3 29. Bxd3 e3 30. Rxd8 e2 31. Be3 Kc7 32. Bf2 Kd6 33. Bg1 Ke5 34. Bh2+ Kd5 35. f4 gxf4 36. Bg1 Ke4 37. b5 Kd3 38. c4 Kd2 39. c5 Kd1 40. a4 Kd2 41. a5 Kd1 42. g5 Kd2 43. Bf2 Kd1 44. a6 Kd2 45. Be3 Kd1 46. Bf4 Kd2 47. Be5 Kd1 48. Bd4 Kd2 49. Bc3 Kd1 50. Bb2 Kd2 51. Ba1 Kd1 52. Bf6 Kd2 53. Be5 Kd1 54. b6 Kd2 55. Bf4 Kd1 56. Bg3 Kd2 57. Bh4 Kd1 58. Bg3 Kd2 59. Be5 Kd1 60. Bd4 Kd2 61. Bc3 Kd1 62. Bb2 Kd2 63. bxa7 Ke3 64. Bc3 Kd2 65. Bd4 Kd1 66. Bf2 Kd2 67. Be3 Kd1 68. Bf4 Kd2 69. Bg3 Kd1 70. Bh4 Kd2 71. Bg3 Kd1 72. c6 Kd2 73. Bh4 Kd1 74. Bg3 Kd2 75. Bd6 Kd1 76. Bc5 Kd2 77. Ba3 Kd1 78. Bb2 Kd2 79. Bc3 Kd1 80. Bb4 Kd2 81. Bc3 Kd1 82. g6 Kd2 83. Bd4 Kd1 84. Be5 Kd2 85. Bf4 Kd1 86. Be5 Kd2 87. Bc3 Kd1 88. Bb4 Kd2 89. Ba3 Kd1 90. Be7 Kd2 91. Bd6 Kd1 92. Be7 Kd2 93. Bb4 Kd1 94. Bc3 Kd2 95. Be5 Kd1 96. Bd4 Kc2 97. Bf2 Kd2 98. Be3 Kd1 99. c7 Kc2 100. c8=Q+ Kd1 101. Qc2 Kd2 102. Qd1 Kc3 103. Qd2+ Kc4 104. Qd3+ Kb4 105. Qb3+ Ka5 106. Qb5# 1-0

Figure A.2.: **Atomic game 2.** Online against Bot Ilmare. MultiAra plays white and has a clear winning position after move 30, but needs 76 more moves to win the game. It does not utilize pawn promotion until move 100.

A.2. Horde

[Event "Ara40-FairySF-long"]
[Site "TU-Darmstadt"]
[Date "2021.07.05"]
[Round "16"]
[White "MultiAra40"]
[Black "FairyStockfish"]
[Result "1/2-1/2"]
[FEN "rnbqkbnr/pppp2pp/3Ppp2/1PP1PPP1/PPP2PPP/PPPPPPPP/PPPPPPPP/PPPPPPPP b kq - 0 1"]
[GameDuration "00:04:49"]
[TimeControl "60+0.6"]
[Variant "horde"]

1... exf5 {+0.94/22 1.9s} 2. gxf5 {-6.73/24 2.0s} h6 {+0.85/22 0.77s} 3. gxf6 {-3.80/21 2.0s} gxf6 {+1.01/23 0.85s} 4. h5 {-3.97/19 2.0s} fxe5 {+0.71/23 1.4s} 5. fxe5 {-1.62/37 2.0s} Bxd6 {+0.67/26 2.2s} 6. cxd6 {+0.01/31 2.0s} cxd6 {+0.86/23 1.3s} 7. exd6 {+0.08/29 2.0s} Qf6 {+0.59/23 2.9s} 8. c5 {+0.05/27 2.0s} Qxf5 {+0.63/23 0.87s} 9. g4 {+0.04/29 2.0s} Qg5 {+0.51/24 0.95s} 10. g3 {+1.00/23 2.1s} Nf6 {-0.17/22 3.1s} 11. g2 {+1.24/23 2.1s} Nxh5 {+0.13/21 1.6s} 12. gxh5 {+1.32/22 0.25s} Qxh5 {-0.01/22 4.2s} 13. h4 {+2.09/18 0.75s} b6 {-0.06/18 4.5s} 14. d4 {+2.42/30 2.2s} Bb7 {-0.12/21 12s} 15. a5 {+3.18/28 1.8s} bxa5 {+0.13/20 0.98s} 16. bxa5 {+2.41/24 2.2s} Rf8 {+0.13/22 3.3s} 17. f4 {+3.36/27 0.75s} Rf6 {+0.05/23 1.2s} 18. a6 {+8.20/14 1.0s} Bxa6 {+0.09/20 6.4s} 19. bxa6 {+4.01/30 2.4s} Nxa6 {+0.05/21 0.54s} 20. b4 {+3.64/26 2.3s} Nb8 {+0.09/19 0.99s} 21. c4 {+9.58/15 1.3s} Nc6 {+0.13/22 1.0s} 22. c3 {+8.43/20 2.5s} Nd8 {+0.13/21 1.9s} 23. a4 {+7.08/24 2.5s} a5 {+0.59/21 2.5s} 24. b3 {+8.26/16 0.75s} axb4 {+0.44/21 0.75s} 25. cxb4 {+7.89/18 2.7s} Nb7 {+0.28/22 1.2s} 26. b2 {+8.03/17 0.25s} Nxd6 {+0.44/21 2.7s} 27. cxd6 {+7.10/23 2.9s} Rxd6 {+0.36/18 0.73s} 28. a3 {+7.36/20 0.25s} O-O-O {+0.13/21 11s} 29. d3 {+9.33/14 2.5s} Kb7 {+0.25/18 1.5s} 30. h3 {+13.40/10 2.3s} Qf7 {+0.47/15 0.22s} 31. h2 {+14.20/12 2.5s} Rf6 {+0.24/17 1.4s} 32. d5 {+10.94/16 3.7s} Kc7 {+0.09/17 0.97s} 33. d2 {+12.82/15 2.8s} Qh5 {-0.05/16 1.2s} 34. a5 {+13.54/12 2.8s} Ra6 {-0.02/15 0.33s} 35. e4 {+17.29/11 4.3s} Rf8 {+0.05/16 0.36s} 36. f5 {+17.94/9 1.3s} Kb7 {+0.05/15 0.49s} 37. f3 {+19.00/8 1.2s} Qe8 {-0.01/18 1.2s} 38. e3 {+19.12/8 1.2s} Qh5 {-0.28/18 0.59s} 39. e2 {+20.34/8 1.1s} d6 {+0.13/16 0.24s} 40. d4 {+19.69/9 1.0s} Kc7 {-0.41/16 1.0s} 41. c3 {+19.83/8 1.1s} Raa8 {-0.66/17 0.68s} 42. d3 {+19.21/7 1.0s} Ra6 {-0.94/15 0.62s} 43. c5 {+19.95/8 0.99s} Re8 {-1.01/15 0.34s} 44. c4 {+19.54/9 0.96s} Rf8 {-1.27/15 0.78s} 45. f4 {+19.59/8 0.94s} Kd8 {-1.01/16 0.46s} 46. f3 {+19.27/8 0.91s} Kc8 {-1.48/15 0.31s} 47. e5 {+20.12/8 0.89s} Rxf5 {-1.80/17 1.0s} 48. e4 {+20.09/10 0.87s} Rxf4 {-1.12/16 0.42s} 49. gxf4 {+22.20/10 0.85s} Qxh4 {-1.17/14 0.12s} 50. g3 {+25.01/10 0.75s} Qxh3 {-3.17/15 0.35s} 51. exd6 {+24.79/9 0.82s} Qxh2 {-3.56/17 1.4s} 52. b5 {+24.55/8 0.81s} Rxa5 {-7.14/18 0.76s} 53. c6 {+22.23/11 0.79s} Qf2 {-9.86/20 0.63s} 54. b4 {+25.81/8 0.75s} Rxb5 {-3.78/14 0.13s} 55. cxb5 {+21.02/12 0.77s} Qxd4 {-3.33/15 0.086s} 56. d7+ {+20.53/16 0.50s} Kd8 {-5.71/20 0.75s} 57. a4 {+20.91/12 0.75s} Ke7 {-17.01/19 1.3s} 58. f5 {+23.71/7 0.75s} Kd8 {-4.54/20 0.59s} 59. g4 {+24.15/7 0.74s} h5 {-5.15/17 0.40s} 60. gxh5 {+21.79/16 0.73s} Ke7 {-16.27/20 0.91s} 61. d8=B+ {+25.76/8 0.72s} Kxd8 {-7.51/14 0.051s} 62. a3 {+18.93/10 0.71s} Qxb2 {-13.42/18 1.1s}

Figure A.3.: **Horde game 1.** Game $31/100$ of the long time control tournament against Fairy-Stockfish. Although Horde is heavily black favoured, after move 80 MultiAra has a clear winning position as white. However, MultiAra does not manage to win and settles for a draw by insufficient mating material.

63. a5 {+23.19/17 0.50s} Qxe2 {-18.59/17 0.70s} 64. a6 {+20.16/7 0.50s} Qe3 {-22.51/17 0.62s} 65. a4 {+19.60/9 0.72s} Qd4 {-13.78/19 0.56s} 66. h6 {+19.39/6 0.71s} Kc7 {-25.19/17 0.63s} 67. f6 {+19.08/7 0.50s} Qxb4 {-20.75/17 0.61s} 68. h7 {+17.75/13 0.71s} Qf8 {-11.06/16 0.13s} 69. h8=N {+16.94/8 0.70s} Kb6 {-18.21/18 0.98s} 70. e5 {+15.37/12 0.69s} Qxh8 {-22.28/17 0.67s} 71. d6 {+16.40/11 0.69s} Kc5 {-25.03/19 0.45s} 72. f4 {+19.50/11 0.68s} Kd5 {-24.94/17 0.73s} 73. c7 {+19.13/7 0.68s} Qc8 {-33.39/17 0.63s} 74. a7 {+17.49/13 0.67s} Ke6 {-29.46/17 0.60s} 75. b6 {+14.01/15 0.67s} Qb7 {-33.36/16 0.60s} 76. d4 {+15.28/10 0.50s} Qc8 {-32.16/18 0.60s} 77. a5 {+14.22/17 0.67s} Qa8 {-37.02/18 0.60s} 78. a6 {+14.93/8 0.66s} Kf5 {-52.34/17 0.60s} 79. d5 {+20.65/8 0.66s} Ke4 {-32.72/14 0.10s} 80. d7 {+23.19/9 0.50s} Qf8 {-58.31/17 1.0s} 81. d6 {+22.93/6 0.66s} Kd4 {-62.01/16 0.68s} 82. f5 {+23.02/6 0.66s} Qg8 {-58.80/15 0.63s} 83. d8=B {+26.05/6 0.65s} Qc4 {-51.23/18 0.59s} 84. e6 {+25.70/7 0.65s} Qxa6 {-46.49/16 0.60s} 85. f7 {+22.96/7 0.65s} Ke3 {-53.65/16 0.60s} 86. f6 {+23.38/6 0.64s} Kd3 {-40.07/14 0.61s} 87. e7 {+22.82/5 0.64s} Qb7 {-55.24/17 0.60s} 88. f8=N {+22.09/7 0.64s} Qc6 {-41.48/14 0.29s} 89. f7 {+20.94/6 0.63s} Kc3 {-39.63/16 0.70s} 90. Nh7 {+21.17/6 0.63s} Kd4 {-61.12/14 0.75s} 91. Nf6 {+20.46/7 0.63s} Kc5 {-59.37/13 0.63s} 92. b7 {+18.46/12 0.63s} Qxb7 {-25.40/13 0.23s} 93. e8=B {+16.56/9 0.63s} Kc4 {-51.78/15 0.94s} 94. d7 {+17.55/7 0.62s} Kc3 {-67.48/11 0.63s} 95. Ng4 {+17.68/6 0.50s} Qa6 {-57.85/13 0.60s} 96. Nh2 {+17.17/7 0.63s} Kb3 {-53.55/12 0.61s} 97. Ng4 {+15.93/6 0.63s} Qd6 {-36.48/12 0.33s} 98. Nf6 {+16.53/12 0.63s} Qa3 {-58.95/15 0.81s} 99. Nh7 {+15.74/7 0.50s} Qxa7 {-49.86/11 0.65s} 100. c8=B {+13.45/9 0.50s} Qd4 {-37.16/12 0.61s} 101. Bc7 {+12.43/8 0.64s} Qf2 {-43.09/13 0.61s} 102. Nf6 {+12.95/6 0.50s} Qxf6 {-28.25/10 0.11s} 103. Bb7 {+8.60/12 0.50s} Kc4 {-M16/19 0.98s} 104. Bg2 {+10.53/9 0.50s} Qf2 {-M14/17 0.53s} 105. Bh1 {+11.00/6 0.50s} Qf5 {-M16/16 0.39s} 106. Bg2 {+10.80/7 0.50s} Kd3 {-M14/18 0.18s} 107. d8=Q+ {+10.41/11 0.50s} Kc2 {-M12/35 0.17s} 108. Bd7 {+10.26/9 0.50s} Qxf7 {-M12/24 0.20s} 109. Bd6 {+8.81/11 0.68s} Qb3 {-M14/21 0.18s} 110. Qc8+ {+8.08/20 0.68s} Kd2 {-M14/35 0.23s} 111. Qe8 {+8.51/8 0.50s} Qc4 {-M14/28 0.53s} 112. Qe5 {+8.31/5 0.50s} Kc2 {-M12/31 0.18s} 113. Qe1 {+8.58/15 0.69s} Kb3 {-M10/52 0.29s} 114. Qb1+ {+8.85/10 0.68s} Kc3 {-M12/42 0.32s} 115. Ba3 {+8.72/13 0.50s} Kd4 {-M14/39 0.34s} 116. Qb2+ {+8.96/14 0.68s} Qc3 {-M14/43 0.36s} 117. Qb6+ {+8.63/12 0.50s} Kd3 {-M14/42 0.44s} 118. Bf1+ {+8.57/17 0.50s} Ke4 {-M12/45 0.40s} 119. Be6 {+8.43/10 0.50s} Qxa3 {-7.27/14 0.47s} 120. Bfc4 {+7.69/13 0.50s} Qc3 {-M70/24 1.8s} 121. Bed5+ {+6.77/15 0.70s} Kf4 {-7.65/19 0.39s} 122. Qf2+ {+6.92/11 0.50s} Kg5 {-7.20/18 0.33s} 123. Be6 {+6.50/12 0.50s} Qf6 {-M40/36 1.2s} 124. Qg3+ {+5.63/16 0.71s} Kh6 {-M34/45 0.35s} 125. Qh3+ {+5.53/13 0.70s} Kg7 {-M38/52 0.39s} 126. Bcd5 {+5.39/15 0.50s} Qd4 {-M44/30 1.0s} 127. Qh5 {+5.49/16 0.50s} Qf4 {-37.81/19 0.22s} 128. Bc4 {+5.28/12 0.50s} Qf6 {-M40/50 1.2s} 129. Bed5 {+4.50/18 0.50s} Qg6 {-M48/31 0.53s} 130. Qh3 {+4.10/12 0.50s} Qd6 {-25.87/20 0.82s} 131. Be6 {+4.45/11 0.50s} Qf4 {-M38/53 1.0s} 132. Bcd5 {+4.12/13 0.50s} Kf6 {-14.40/20 0.28s} 133. Qh8+ {+4.36/12 0.50s} Kg6 {-M48/44 0.81s} 134. Qg8+ {+3.90/18 0.75s} Kh6 {-M40/44 0.36s} 135. Qh8+ {+4.01/15 0.25s} Kg6 {-14.78/19 0.27s} 136. Qg8+ {+3.36/15 0.77s} Kh6 {-M40/48 0.91s} 137. Bh3 {+3.34/11 0.50s} Qg5 {-8.28/23 0.61s} 138. Qh8+ {+2.76/21 0.75s} Kg6 {-6.26/22 0.40s} 139. Be4+ {+2.80/19 0.75s} Kf7 {-18.48/23 0.66s} 140. Bd7 {+2.83/15 0.25s} Ke7 {-6.24/17 0.38s} 141. Qe8+ {+2.80/18 0.76s} Kd6 {-5.19/24 0.42s} 142. Qe6+ {+2.64/14 0.75s} Kc5 {-4.22/17 0.39s} 143. Bdc6 {+2.51/13 0.50s} Kb4 {-4.10/17 0.41s} 144. Bed5 {+2.41/9 0.50s} Kc5 {-4.19/20 1.7s} 145. Qd7 {+2.35/10 0.50s} Qf6 {-5.87/26 2.0s} 146. Be6 {+2.21/20 0.75s} Qe5 {-4.91/21 1.3s} 147. Bf7 {+2.01/11 0.50s} Qd6 {-3.59/17 0.17s} 148. Qxd6+ {+1.95/13 0.50s} Kxd6 {-3.57/17 0.11s, Draw by insufficient mating material} 1/2-1/2

Figure A.3.: **Horde game 1.** Game $31/100$ of the long time control tournament against Fairy-Stockfish. Although Horde is heavily black favoured, after move 80 MultiAra has a clear winning position as white. However, MultiAra does not manage to win and settles for a draw by insufficient mating material.

[Event "Ara40-FairySF-long"]
[Site "TU-Darmstadt"]
[Date "2021.07.06"]
[Round "45"]
[White "MultiAra40"]
[Black "FairyStockfish"]
[Result "1/2-1/2"]
[FEN "rnbqkbnr/ppp1pp1p/3p2p1/1PPP1PPP/PPPPPPP1/PPP1PPPP/PPPPPPPP/PPPPPPPP b kq - 0 1"]
[GameDuration "00:04:45"]
[TimeControl "60+0.6"]
[Variant "horde"]

1... gxh5 {+0.71/24 3.3s} 2. gxh5 {-6.02/25 2.0s} e6 {+0.90/23 0.65s} 3. fxe6 {-6.01/21 2.0s} fxe6 {+0.85/22 0.72s} 4. h6 {-5.97/21 0.25s} dxc5 {+1.13/22 3.6s} 5. dxc5 {-5.99/22 2.1s} a6 {+1.36/23 2.2s} 6. h4 {-5.33/23 2.1s} Bd7 {+1.36/21 4.8s} 7. d4 {-4.98/24 2.1s} Nxh6 {+1.28/23 3.1s} 8. gxh6 {-4.68/21 2.1s} Bxh6 {+1.25/21 0.80s} 9. c6 {-4.70/23 2.1s} bxc6 {+1.53/21 1.2s} 10. bxc6 {-4.75/21 2.1s} Nxc6 {+1.53/21 1.2s} 11. dxc6 {-5.03/17 2.1s} Bxc6 {+1.59/20 0.81s} 12. h5 {-5.21/16 2.1s} Kd7 {+1.48/19 1.4s} 13. d2 {-4.02/17 2.0s} Bb7 {+1.20/22 13s} 14. c5 {-4.12/15 2.1s} e5 {+1.53/21 1.3s} 15. fxe5 {-2.90/19 2.2s} Bg7 {+1.08/21 3.6s} 16. f4 {-1.23/18 1.0s} Bxe4 {+0.85/16 1.7s} 17. c4 {-1.29/23 2.2s} Rg8 {+1.38/18 3.9s} 18. h3 {-1.67/21 2.0s} Bh6 {+1.31/17 3.4s} 19. b5 {-2.01/38 2.3s} Ke8 {+1.65/20 6.0s} 20. c6 {-1.37/30 0.75s} Kf7 {+1.59/18 0.84s} 21. d3 {+0.84/27 1.0s} Bxf4 {+1.67/20 0.71s} 22. gxf4 {+4.07/20 2.5s} Bf5 {+1.67/21 1.0s} 23. c5 {+3.45/36 2.5s} Bxh3 {+1.23/20 3.9s} 24. c4 {+4.47/22 2.5s} Be6 {+1.38/20 3.3s} 25. h2 {+4.87/22 1.3s} Bg4 {+1.25/22 6.1s} 26. b4 {+5.77/21 1.3s} Bxe2 {+1.75/16 0.14s} 27. fxe2 {+5.04/25 2.7s} Rxg2 {+0.78/20 1.8s} 28. d5 {+9.68/15 2.7s} Rxg1 {+0.95/18 0.81s} 29. d6 {+10.31/18 2.8s} axb5 {+0.76/18 0.83s} 30. axb5 {+11.86/16 2.8s} Ra7 {+0.85/15 0.26s} 31. b3 {+15.04/14 1.5s} Rxe1 {+0.17/15 0.90s} 32. a4 {+17.36/13 3.0s} Rxc1 {+0.54/13 0.063s} 33. a5 {+17.88/11 3.1s} Qe8 {+0.54/14 0.20s} 34. d4 {+19.32/11 3.1s} Rxb1 {-2.73/15 1.6s} 35. d5 {+18.98/11 3.1s} Rg1 {-4.28/15 0.80s} 36. b6 {+20.84/8 1.0s} Qa8 {-3.59/15 0.40s} 37. b5 {+23.23/8 0.98s} cxd6 {-1.65/15 0.43s} 38. cxd6 {+22.64/9 0.95s} Rb7 {-3.55/16 0.79s} 39. b4 {+23.94/9 0.92s} Rg8 {-6.90/14 0.79s} 40. c5 {+24.86/6 0.90s} Rbb8 {-8.90/15 0.64s} 41. e4 {+25.61/10 0.88s} Rgc8 {-9.94/14 0.61s} 42. e3 {+23.51/6 0.75s} Ke8 {-7.65/15 0.42s} 43. d7+ {+21.93/8 0.75s} Kf7 {-10.10/17 0.75s} 44. a4 {+23.75/6 0.75s} Kg8 {-12.50/17 0.63s} 45. a6 {+24.31/6 0.83s} Rd8 {-17.24/16 0.60s} 46. a5 {+24.04/12 0.75s} Rf8 {-17.29/17 0.60s} 47. e6 {+25.28/6 0.80s} Kg7 {-22.59/19 0.60s} 48. e5 {+27.10/7 0.79s} Rfd8 {-19.77/15 0.60s} 49. e4 {+26.59/7 0.77s} Rh8 {-20.35/16 0.60s} 50. b7 {+27.35/7 0.76s} Qa7 {-28.31/16 0.60s} 51. a3 {+26.11/8 0.75s} Rhf8 {-23.40/17 0.60s} 52. a4 {+25.25/6 0.74s} Rxf4 {-26.69/15 0.60s} 53. f3 {+22.89/6 0.73s} Rff8 {-17.72/17 0.40s} 54. f4 {+23.82/6 0.72s} Rfd8 {-25.92/18 0.76s} 55. f5 {+23.28/7 0.71s} Kh6 {-24.86/18 0.55s} 56. f6 {+23.19/7 0.70s} Kg5 {-27.13/16 0.69s} 57. e7 {+26.30/6 0.69s} Kh4 {-31.48/15 0.60s} 58. e6 {+27.61/6 0.69s} Rh8 {-35.17/15 0.60s} 59. e5 {+27.25/6 0.68s} h6 {-23.94/16 0.30s} 60. h3 {+25.61/7 0.68s} Kg3 {-35.26/15 0.84s} 61. h4 {+26.11/6 0.67s} Kg4 {-31.49/16 0.65s} 62. d6 {+24.90/8 0.67s} Kf5 {-34.19/17 0.61s} 63. c7 {+22.26/6 0.66s} Ra8 {-36.71/14 0.61s} 64. b6 {+23.31/7 0.66s} Qxa6 {-26.17/13 0.089s} 65. c6 {+21.62/7 0.65s} Kxe6 {-38.46/14 1.0s} 66. b5 {+19.08/8 0.65s} Qxb7 {-35.17/17 0.25s} 67. d8=Q {+17.15/10 0.65s} Qb8 {-30.82/16 0.66s} 68. b7 {+19.01/7 0.64s} Qxd8 {-40.06/16 0.92s} 69. exd8=N+ {+21.29/7 0.50s} Raxd8 {-38.63/17 0.12s} 70. a6 {+20.73/8 0.65s} Rde8 {-26.21/17 0.75s} 71. b6 {+21.21/7 0.50s} Kxe5 {-30.24/15 0.93s}

Figure A.4.: **Horde game 2.** Game $89/100$ of the long time control tournament against Fairy-Stockfish. Although Horde is heavily black favoured, after move 75 MultiAra has a clear winning position as white. However, MultiAra does not manage to win and settles for a draw by fifty moves rule.

72. d7 {+18.87/9 0.50s} Ref8 {-31.98/15 0.66s} 73. c8=Q {+20.18/7 0.50s} Ke4 {-45.51/15 0.61s} 74. a5 {+21.19/8 0.66s} Rxc8 {-40.63/18 0.60s} 75. dxc8=R {+23.05/6 0.50s} Rh7 {-32.11/12 0.053s} 76. a7 {+24.11/6 0.67s} Kd3 {-46.92/16 1.0s} 77. c7 {+22.46/6 0.66s} Kd4 {-68.46/16 0.69s} 78. b8=N {+23.05/7 0.66s} Ke5 {-37.98/15 0.36s} 79. a6 {+21.82/8 0.65s} Kf4 {-43.94/17 0.81s} 80. b7 {+23.60/7 0.65s} Rf7 {-31.12/14 0.24s} 81. a8=N {+23.09/8 0.65s} Rxf6 {-28.48/14 0.14s} 82. a7 {+21.11/7 0.64s} Kg3 {-33.51/16 1.3s} 83. Rd8 {+19.53/7 0.64s} Rf7 {-30.18/13 0.21s} 84. Nc6 {+20.46/7 0.64s} Rg7 {-72.68/15 1.1s} 85. Nb4 {+18.86/10 0.64s} Rd7 {-M14/16 0.61s} 86. Nd5 {+18.26/10 0.63s} Rxc7 {-34.09/16 0.11s} 87. Rh8 {+14.78/10 0.63s} Rxb7 {-8.71/12 0.074s} 88. Rxh6 {+9.68/14 0.50s} Rxa7 {-8.37/20 0.24s} 89. Rh8 {+8.81/18 0.50s} Kxh4 {-8.22/21 1.2s} 90. Nac7 {+8.17/13 0.25s} Kg5 {-8.21/18 0.21s} 91. Rh7 {+7.79/14 0.25s} Ra1 {-7.10/14 0.21s} 92. Rh8 {+7.57/16 0.50s} Rh1 {-8.25/23 1.9s} 93. Ne6+ {+6.55/15 0.50s} Kf5 {-7.46/15 0.062s} 94. Nec7 {+6.68/16 0.25s} Rh3 {-8.31/22 0.93s} 95. Rh6 {+6.83/11 0.50s} Kg5 {-5.75/20 0.18s} 96. Rb6 {+5.73/13 0.50s} Rxh5 {-4.54/13 0.16s} 97. Rc6 {+4.29/13 0.50s} Kf5 {-4.21/17 0.86s} 98. Ne7+ {+3.30/18 0.75s} Ke4 {-4.20/14 0.11s} 99. Na6 {+2.91/26 0.50s} Rh7 {-3.95/16 0.57s} 100. Ng6 {+2.64/18 0.25s} Kf5 {-3.82/15 0.20s} 101. Nc5 {+2.34/20 0.50s} Rg7 {-3.75/15 0.28s} 102. Nh4+ {+1.90/16 0.78s} Ke5 {-3.62/15 0.47s} 103. Ra6 {+1.88/10 0.25s} Rh7 {-3.41/16 2.0s} 104. Nf3+ {+1.68/21 0.79s} Kd5 {-3.30/16 0.39s} 105. Ne6 {+1.65/17 0.50s} Rh1 {-3.31/13 0.22s} 106. Nc7+ {+1.60/15 0.79s} Ke4 {-3.10/14 0.22s} 107. Ng5+ {+1.36/14 0.77s} Kf5 {-2.96/15 0.50s} 108. Ra5+ {+1.35/17 0.50s} Kf6 {-2.94/14 0.22s} 109. Ne4+ {+1.27/13 0.77s} Ke7 {-2.79/16 0.47s} 110. Ra6 {+1.49/17 0.50s} Rd1 {-2.82/16 0.33s} 111. Rh6 {+1.48/12 0.50s} Kd7 {-2.64/16 0.36s} 112. Nb5 {+1.34/23 0.77s} Ke7 {-2.50/19 2.1s} 113. Rh7+ {+1.26/14 0.75s} Ke6 {-2.39/13 0.17s} 114. Rh6+ {+1.17/11 0.50s} Ke5 {-2.37/15 0.31s} 115. Nc5 {+1.16/11 0.50s} Rd8 {-2.35/15 0.28s} 116. Rh5+ {+0.61/12 0.76s} Kf6 {-2.20/15 0.36s} 117. Rh6+ {+0.60/13 0.75s} Ke5 {-2.13/17 0.36s} 118. Ne6 {+0.64/9 0.50s} Rd1 {-2.00/16 0.72s} 119. Ng7 {+0.80/11 0.50s} Rd7 {-1.94/14 0.27s} 120. Rg6 {+0.63/14 0.50s} Rd1 {-1.77/17 1.2s} 121. Re6+ {+0.95/13 0.50s} Kf4 {-1.80/14 0.23s} 122. Rf6+ {+0.81/11 0.50s} Kg5 {-1.63/14 0.39s} 123. Rf7 {+0.72/11 0.50s} Kg6 {-1.33/15 1.6s} 124. Rc7 {+0.55/18 0.50s} Kf6 {-1.27/15 0.55s} 125. Nh5+ {+0.50/12 0.79s} Ke6 {-1.18/16 0.97s} 126. Rc6+ {+0.68/13 0.50s} Kd7 {-1.06/16 1.9s} 127. Ra6 {+0.65/13 0.50s} Ke7 {-1.14/16 0.52s} 128. Ra7+ {+0.65/11 0.50s} Ke6 {-0.99/12 0.12s} 129. Ra6+ {+0.68/15 0.50s} Kf5 {-0.96/14 0.28s} 130. Ra8 {+0.61/10 0.80s} Kg6 {-0.85/14 0.27s} 131. Nc3 {+0.61/14 0.50s} Re1 {-0.65/16 1.4s} 132. Nf4+ {+0.48/23 0.79s} Kf6 {-0.68/14 0.16s} 133. Ra6+ {+0.61/21 0.50s} Kf5 {-0.48/14 0.84s} 134. Ng2 {+0.65/10 0.50s} Rh1 {-0.43/13 0.13s} 135. Ne3+ {+0.28/22 0.80s} Ke5 {-0.36/15 0.94s} 136. Nc4+ {+0.31/22 0.78s} Kd4 {-0.30/13 0.19s} 137. Ne2+ {+0.44/11 0.50s} Kc5 {-0.13/15 1.7s} 138. Ne5 {+0.31/14 0.77s} Kb5 {-0.15/13 0.37s} 139. Re6 {+0.36/12 0.50s} Kc5 {-0.04/14 0.42s} 140. Rc6+ {+0.34/16 0.50s} Kd5 {0.00/12 0.12s} 141. Rb6 {+0.29/12 0.75s} Kc5 {0.00/16 0.22s} 142. Rc6+ {+0.30/10 0.50s} Kb5 {0.00/25 0.25s} 143. Nc3+ {+0.18/8 0.77s} Kb4 {0.00/31 0.27s} 144. Ra6 {+0.23/6 0.50s} Rh3 {0.00/38 0.29s} 145. Na4 {+0.16/4 0.75s} Rh1 {0.00/54 0.32s} 146. Nb6 {0.00/2 0.76s} Ra1 {0.00/245 0.010s, Draw by fifty moves rule} 1/2-1/2

Figure A.4.: **Horde game 2.** Game $89/100$ of the long time control tournament against Fairy-Stockfish. Although Horde is heavily black favoured, after move 75 MultiAra has a clear winning position as white. However, MultiAra does not manage to win and settles for a draw by fifty moves rule.

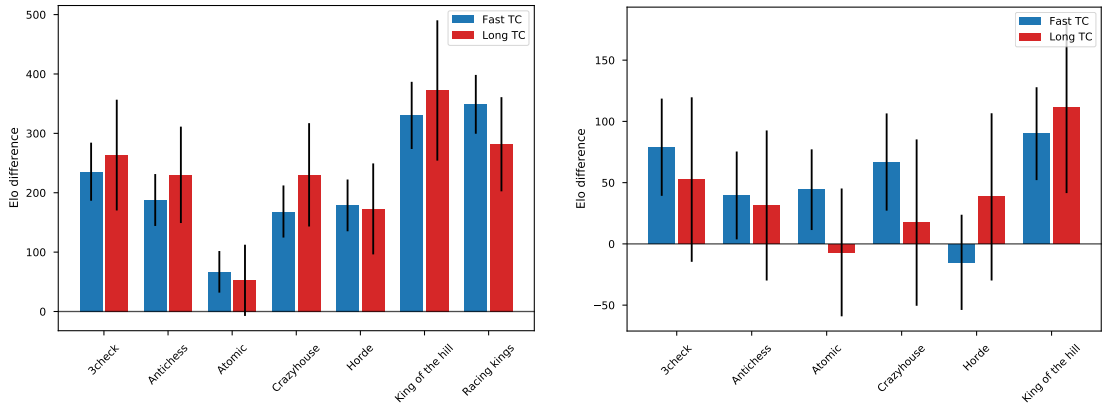
[Event "Casual Horde game"]
[Site "https://lichess.org/qXpGxjRp"]
[Date "2021.07.10"]
[White "MultiAra"]
[Black "Fairy-Stockfish"]
[Result "1/2-1/2"]
[Variant "Horde"]
[TimeControl "30+1"]
[FEN "rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1"]

1. h5 a6 2. e5 d6 3. e4 e6 4. fxe6 fxe6 5. cxd6 cxd6 6. h4 Bd7 7. e3 Kf7 8. f5 exf5 9. gxf5 Ne7 10. g4 dxe5 11. dxe5 g6 12. e6+ Bxe6 13. fxe6+ Kxe6 14. h6 Kf7 15. g3 Ng8 16. d4 Be7 17. f4 Nxh6 18. gxh6 Bxh4 19. h3 Be7 20. g2 g5 21. d5 Bf8 22. fxg5 Qxg5 23. f4 Qxh6 24. d4 Nd7 25. f3 axb5 26. axb5 Nf6 27. e5 Nxg4 28. fxg4 Rg8 29. c5 Rxg4 30. hxg4 Qxh1 31. c4 Qxe1 32. c3 Qxe3 33. c6 Qe4 34. a4 Rc8 35. cxb7 Rb8 36. a5 Rxb7 37. b6 Bxb4 38. cxb4 Qxd4 39. c3 Qe4 40. d6 Rb8 41. c5 Qc6 42. d3 h6 43. d4 h5 44. gxh5 Rd8 45. g4 Ra8 46. a4 Qb7 47. a2 Kg7 48. e6 Kf6 49. f5 Rb8 50. e7 Kg5 51. b5 Qd5 52. a6 Rg8 53. a7 Kf6 54. a5 Kg5 55. a4 Kxg4 56. h6 Kxf5 57. b4 Kf6 58. b3 Qa8 59. c4 Ke6 60. g3 Rh8 61. a6 Kd7 62. g4 Re8 63. c6+ Kxd6 64. g5 Rxe7 65. g6 Re4 66. c5+ Ke7 67. d5 Kf6 68. c7 Kxg6 69. d6 Rh4 70. c6 Rf4 71. b7 Qxa7 72. a5 Kh7 73. b6 Qxa6 74. b8=Q Rxb4 75. b2 Qxb6 76. axb6 Kg6 77. b7 Kh5 78. Qa8 Re4 79. b4 Re7 80. b5 Kh4 81. b6 Rxc7 82. b3 Kg4 83. b4 Rd7 84. b5 Rc7 85. b8=N Rf7 86. b7 Kg5 87. b6 Kg6 88. d7 Rf8 89. d8=N Kh5 90. Ne6 Re8 91. Nc5 Re6 92. Nca6 Kg6 93. Nb4 Re8 94. N4a6 Kh7 95. Nb4 Kxh6 96. N4a6 Rg8 97. Nc5 Rh8 98. Nca6 Kh5 99. c7 Re8 100. Nb4 Re5 101. Nc2 Kg6 102. Na1 Rh5 103. Nc2 Rh1 104. Na3 Rh2 105. Nb5 Rh1 106. Nc3 Rh2 107. Na4 Kh7 108. Nc5 Rh1 109. Ncd7 Rh2 110. c8=N Rf2 111. Nc5 Rf4 112. Nca6 Rf5 113. Na7 Rf8 114. Nac6 Rh8 115. Ncb4 Kg7 116. Qa7 Kg6 117. N8c6 Kg5 118. Nc2 Kh5 119. Nc6b4 Kg4 120. Nd4 Kh3 121. Nbc2 Kg4 122. Ne6 Kg3 123. Nec5 Kf2 124. Nd7 Kf3 125. Ndc5 Rf8 126. Ne6 Ke2 127. b8=N Rf3 128. Nd7 Kf1 129. Nec5 Ke2 130. b7 Rf1 131. Ncb4 Rf5 132. Nc2 Kf3 133. Nb6 Rf7 134. Nca4 Kg4 135. N4c5 Kg5 136. Ncd7 Rf1 137. Ndc5 Rf7 138. Na3 Kh5 139. Nca4 Rf8 140. Nb2 Rf7 141. N2c4 Rf8 142. Nd6 Kh4 143. Ndb5 Rf7 144. Nbc7 Rf8 145. b8=N Rf4 146. N8d7 Kg3 147. Ndb8 Rf7 148. Nc6 Rf3 149. Ncb8 Rxa3 150. Ne6 Ra5 151. Nd4 Rg5 152. N8d7 Kf2 153. Ne6 Rg6 154. Nec5 Kf1 155. Nca4 Rc6 156. Nab8 Re6 157. Na6 Rg6 158. N4c5 Rg8 159. Ne6 Rg4 160. Qb8 Rg2 161. Na4 Re2 162. N4c5 Kg1 163. Nb4 Rd2 164. Nca6 Kf1 165. Ng5 Rf2 166. Ne4 Rb2 167. Nc6 Rxb8 168. Ncxb8 Ke2 169. Nc3+ Kd3 170. Na4 Kc2 171. Nab6 Kb1 172. Ne5 Kc1 173. N8d7 Kd2 174. Nc7 Ke3 175. Nb8 Kd4 176. Nca6 Kxe5 177. Nc6+ Kd6 178. Ncb4 Ke5 179. Na4 Ke4 180. N4c5+ Kd4 181. Ne6+ Kc4 182. Nf4 Kd4 183. Nh5 Ke5 184. Nd3+ Kd4 185. Ndf4 Ke4 186. Nb4 Ke5 187. Nfd5 Kd6 188. Ndf4 Ke5 189. Nfd5 Kd6 190. Ne3 Ke5 191. Nc4+ Kd4 192. Na5 Kc5 193. Nd3+ Kb5 194. Ndf4 Kxa5 195. Nf6 Kb4 196. Nh3 Kc4 197. Ng5 Kd3 198. Ne6 Kc4 199. Ng5 Kd3 200. Ne6 Kc4 201. Nf4 Kd4 202. Ng6 Kc5 203. Ne5 Kd4 204. Neg4 Kc4 205. Ne4 Kd5 206. Ng5 Kd4 207. Nf7 Kd3 208. Nd6 Kd4 209. Nc8 Kd5 210. Na7 Kc4 211. Nc6 Kc5 212. Ne7 Kd4 213. Ng6 Ke4 214. Nh6 Kd5 215. Nf5 Ke6 216. Nd4+ Kd5 217. Nb3 Kc4 218. Nd2+ Kd3 219. Nf3 Ke4 220. Nfh4 Kd4 221. Ne7 Ke4 222. Nhg6 Kd4 223. Nh4 Ke4 224. Neg6 Kd4 225. Ne7 { The game is a draw. } 1/2-1/2

Figure A.5.: **Horde game 3.** Online against Fairy-Stockfish. Although Horde is heavily black favoured, after move 75 MultiAra has a clear winning position as white. However, MultiAra does not manage to win and settles for draw by threefold repetition.

B. Extended Model Analyses

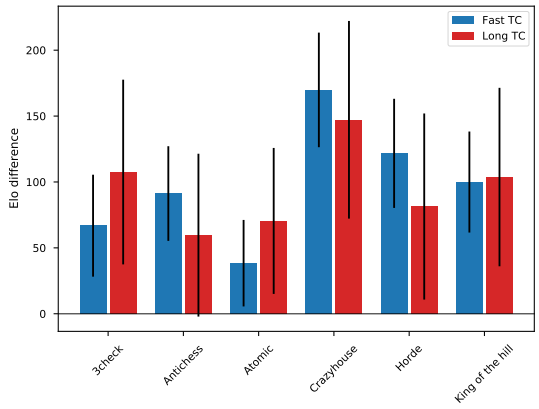
B.1. Development of MultiAra's Elo



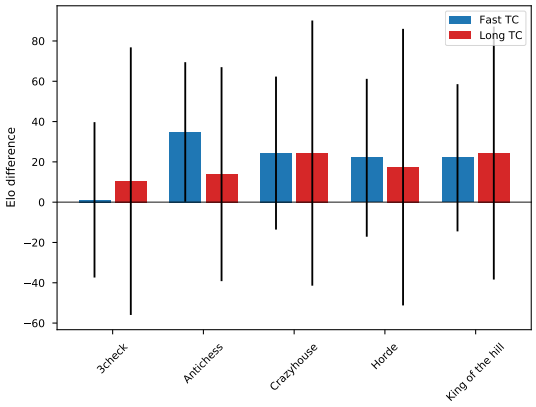
(a) Only human games $\rightarrow$ Update 10

(b) Update $10 \rightarrow 20$

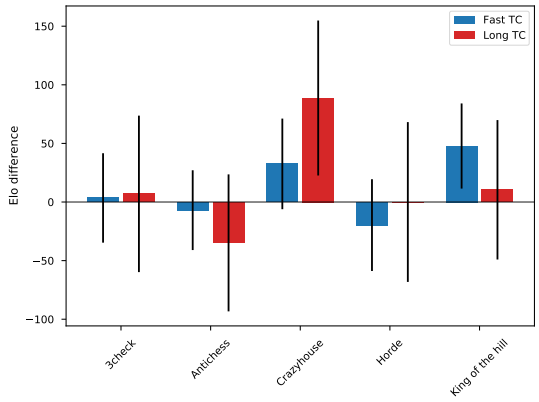
Figure B.1.: **Development of MultiAra's Elo.** Elo difference between every 10^{th} model update for each chess variant. During the fast/long time control (TC) each player had $10/60$ seconds for the game plus $0.1/0.6$ seconds per move.



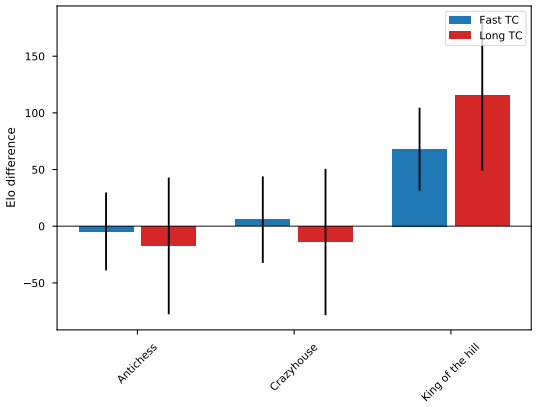
(c) Update $20 \rightarrow 30$



(d) Update $30 \rightarrow 40$



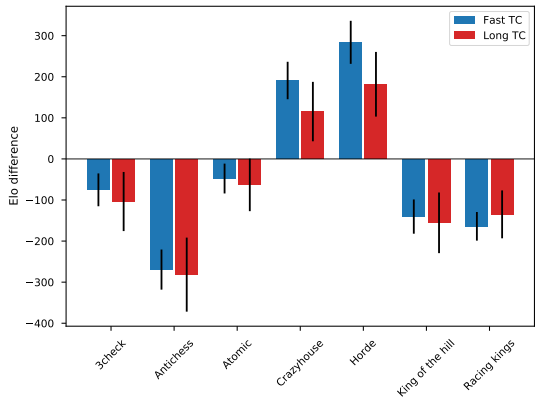
(e) Update $40 \rightarrow 50$



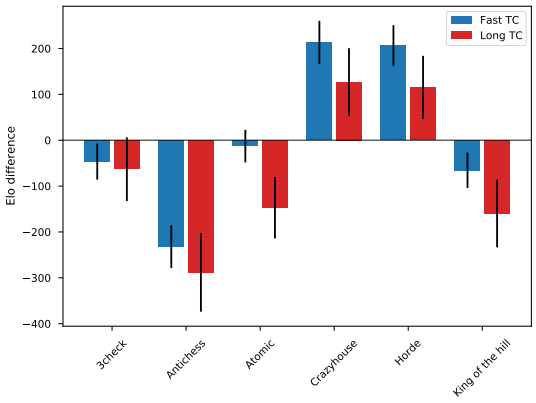
(f) Update $50 \rightarrow 60$

Figure B.1.: **Development of MultiAra's Elo.** Elo difference between every 10^{th} model update for each chess variant. During the fast/long time control (TC) each player had $10/60$ seconds for the game plus $0.1/0.6$ seconds per move.

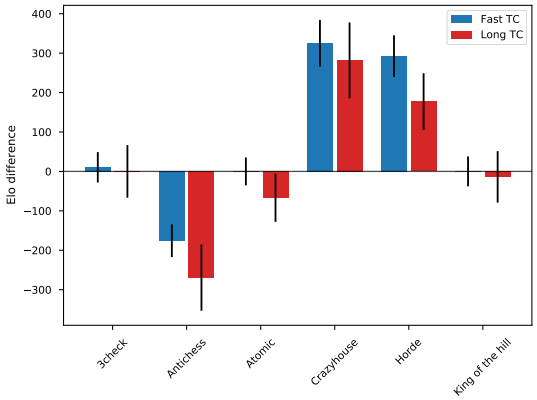
B.2. Development of MultiAra's playing strength



(a) Supervised 10

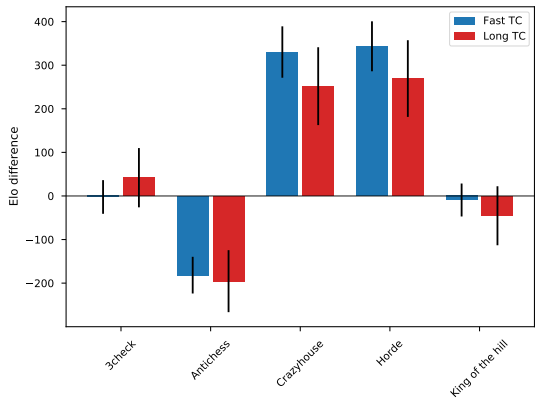


(b) Update 20

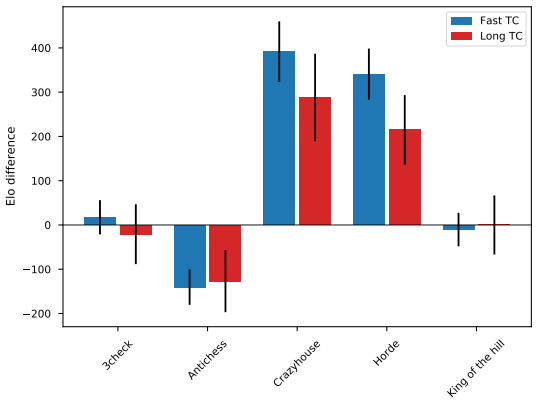


(c) Update 30

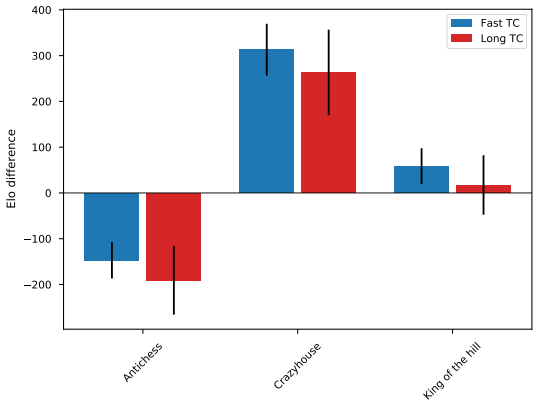
Figure B.2.: **Development of MultiAra's playing strength.** Elo difference between every 10^{th} model update of MultiAra and Fairy-Stockfish 13.1 (classical evaluation). During the fast/long time control (TC) each player had $10/60$ seconds for the game plus $0.1/0.6$ seconds per move.



(d) Update 40



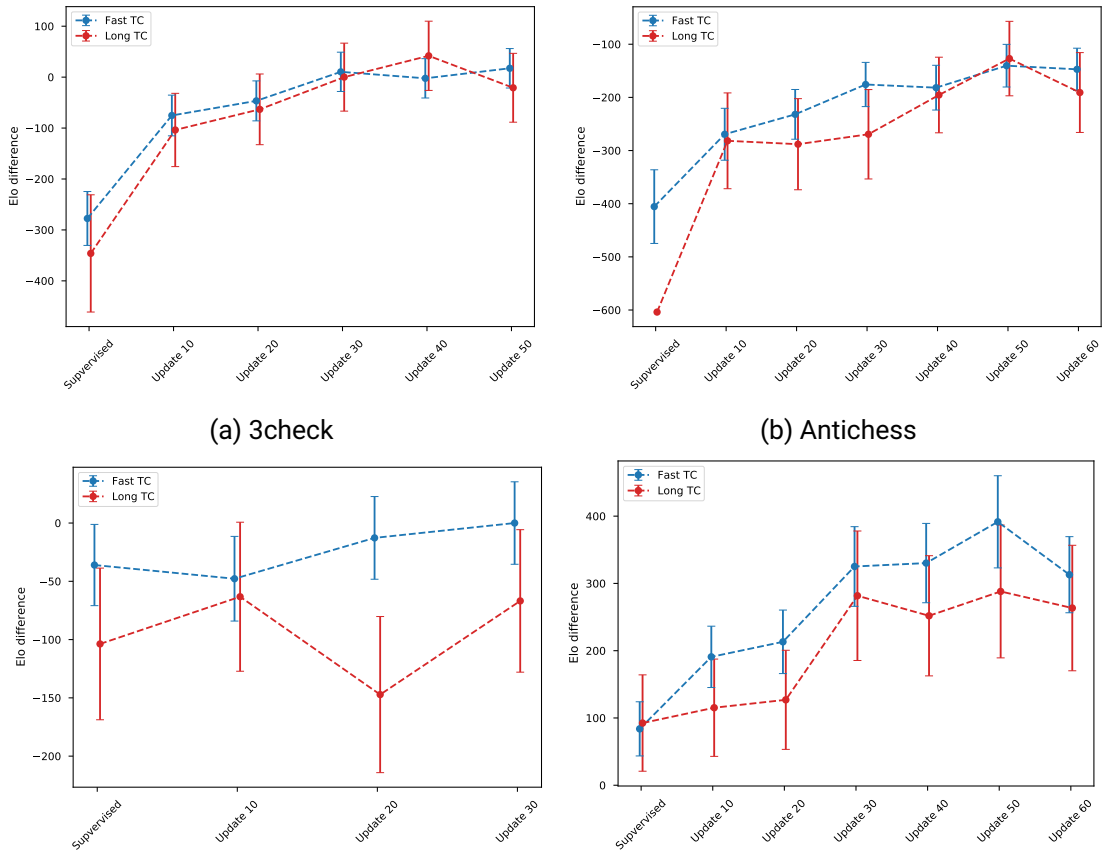
(e) Update 50



(f) Update 60

Figure B.2.: **Development of MultiAra's playing strength.** Elo difference between every 10^{th} model update of MultiAra and Fairy-Stockfish 13.1 (classical evaluation). During the fast/long time control (TC) each player had $10/60$ seconds for the game plus $0.1/0.6$ seconds per move.

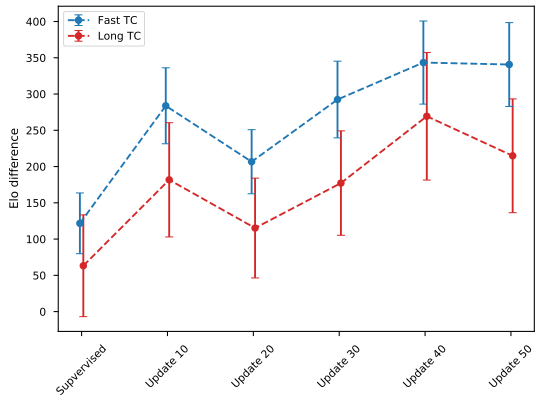
B.3. Development of each variant's playing strength



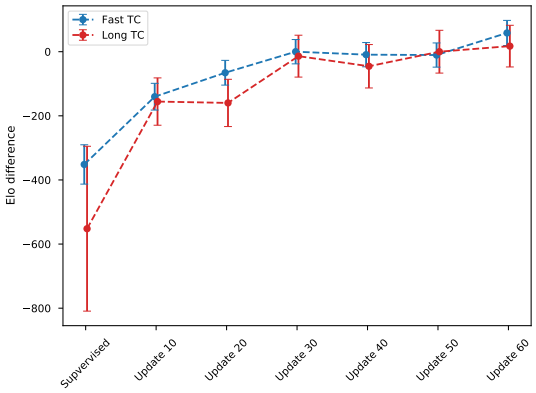
(a) 3check (b) Antichess

(c) Atomic (d) Crazyhouse

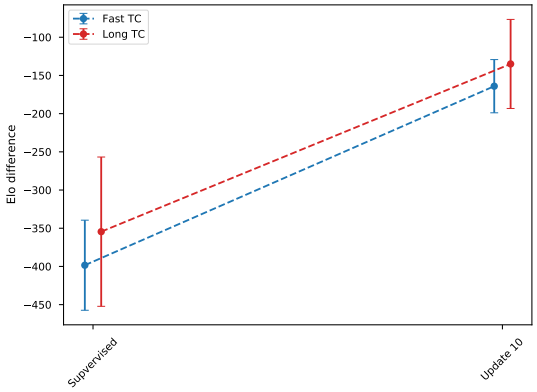
Figure B.3.: **Development of each variant's playing strength.** Comparison between Fairy-Stockfish 13.1 (classical evaluation) and the individual MultiAra model checkpoints for each variant. During the fast/long time control (TC) each player had $10/60$ seconds for the game plus $0.1/0.6$ seconds per move.



(e) Horde



(f) King of the hill



(g) Racing kings

Figure B.3.: **Development of each variant's playing strength.** Comparison between Fairy-Stockfish 13.1 (classical evaluation) and the individual MultiAra model checkpoints for each variant. During the fast/long time control (TC) each player had $10/60$ seconds for the game plus $0.1/0.6$ seconds per move.

C. Reinforcement Learning Settings

C.1. Engine Configuration

```
# Python options
arena_games: int = 100
nn_update_files: int = 10
precision: str = f'float16'
rm_nb_files: int = 5
rm_fraction_for_selection: float = 0.05

# Binary Options
Allow_Early_Stopping: bool = False
Batch_Size: int = 8
Centi_CPuct_Init: int = 250
Centi_Dirichlet_Alpha: int = 20
Centi_Dirichlet_Epsilon: int = 25
Centi_Epsilon_Checks: int = 0
Centi_Epsilon_Greedy: int = 0
Centi_Node_Temperature: int = 100
Centi_Q_Value_Weight: int = 100
Centi_Q_Veto_Delta: int = 40
Centi_Quick_Dirichlet_Epsilon: int = 0
Centi_Quick_Probability: int = 0
Centi_Quick_Q_Value_Weight: int = 70
Quick_Nodes: int = 100
Centi_Random_Move_Factor: int = 0
Centi_Resign_Probability: int = 0
Centi_Resign_Threshold: int = 0
```

```
Centi_U_Init_Divisor: int = 100
Centi_Virtual_Loss: int = 100
MCTS_Solver: bool = True
Milli_Policy_Clip_Thresh: int = 0
Move_Overhead: int = 0
Nodes: int = 800
Centi_Node_Random_Factor: int = 10
Reuse_Tree: str = False
Search_Type: str = f'mcgs'
Selfplay_Chunk_Size: int = 128
Selfplay_Number_Chunks: int = 640
Simulations: int = 3200
SyzygyPath: str = f''
Timeout_MS: int = 0
Use_NPS_Time_Manager: bool = False
MaxInitPly: int = 30
MeanInitPly: int = 8
Centi_Raw_Prob_Temperature: int = 5
Centi_Temperature: int = 80
Centi_Temperature_Decay: int = 92
Centi_Quantile_Clipping: int = 0
Temperature_Moves: int = 15

# Binary option changes for arena tournaments
Centi_Temperature: int = 60
```

C.2. Training Configuration

```
div_factor: int = 2
batch_size: int = int(1024 / div_factor)
batch_steps: int = 100 * div_factor
context: str = "gpu"
cpu_count: int = 4
device_id: int = 0
discount: float = 1.0
```

```
dropout_rate: float = 0
export_dir: str = "./"
export_weights: bool = True
export_grad_histograms: bool = True
is_policy_from_plane_data: bool = False
log_metrics_to_tensorboard: bool = True
k_steps_initial: int = 0
symbol_file: str = ''
params_file: str = ''

# optimization parameters
optimizer_name: str = "nag"
max_lr: float = 0.1 / div_factor
min_lr: float = 0.00001 / div_factor
max_momentum: float = 0.95
min_momentum: float = 0.8
max_spikes: int = 20

name_initials: str = "MAG"
nb_parts: int = None
normalize: bool = True
nb_training_epochs: int = 1
policy_loss_factor: float = 1
q_value_ratio: float = 0.15
seed: int = 42
select_policy_from_plane: bool = True
spike_thresh: float = 1.5
sparse_policy_label: bool = False
total_it: int = None
use_mxnet_style: bool = False
use_spike_recovery: bool = True
val_loss_factor: float = 0.5
wd: float = 1e-4
```

D. Miscellaneous

D.1. Engine Performance

Time per position	Engine	NPS (avg)	NPS (median)	PV-Depth
3000 ms	CrazyAra	22279	22903	35
3000 ms	MultiAra	21668	22377	32
100 ms	CrazyAra	21722	22238	18
100 ms	MultiAra	21554	22250	20

Table D.1.: **Nodes per second.** Evaluated positions per second, also known as nodes per second (NPS), for CrazyAra and MultiAra. The NPS has been measured for 3 seconds and another time for 100 milliseconds on 15 Crazyhouse benchmark positions.

D.2. Fairy-Stockfish NNUE Models

Variant	Version	NNUE Model Name	Elo gain
3check	4	3check-bba0afd355bd.nnue	236.02
Atomic	6	atomic-6afe873e5ac3.nnue	501.99
King of the hill	5	kingofthehill-581cd1c0b2e5.nnue	609.37
Racing kings	5	racingkings-9ef6dd7cfdfb.nnue	290.88

Table D.2.: **NNUE models.** Displaying the Fairy-Stockfish NNUE models that we used for testing. The Elo gain refers to Fairy-Stockfish with classical evaluation.

D.3. Setup Tests for Racing Kings

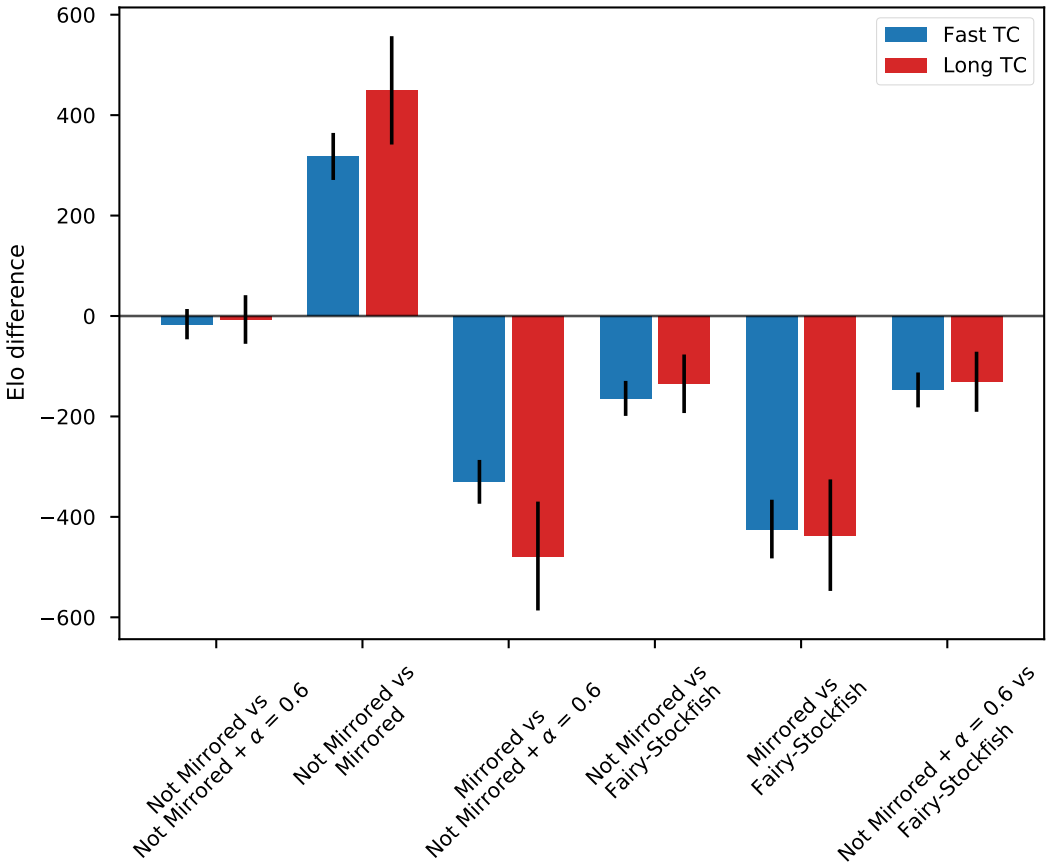


Figure D.1.: **Testing different setups for Racing kings.** For each setup we trained a supervised model with human games and refined it with 10 reinforcement learning updates. Afterwards, we compared the models in a round robin tournament. The model *Mirrored* has mirroring turned on and *Not Mirrored* has mirroring turned off. In addition, the model *Not Mirrored +* $\alpha = 0.6$ has decreased Dirichlet exploration. During the fast/long time control (TC) each player had $10/60$ seconds for the game plus $0.1/0.6$ seconds per move.