Deep Reinforcement Learning for Crazyhouse

Master thesis by Johannes Czech
Date of submission: December 30, 2019

1. Review: Prof. Dr. Kristian Kersting
2. Review: Karl Stelzner
Darmstadt

TECHNISCHE UNIVERSITÄT DARMSTADT

Computer Science Department
Machine Learning Lab

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

Hiermit versichere ich, Johannes Czech, 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, 30. Dezember 2019

J. Czech

Abstract

There has been recent successes in learning the board games Go, chess and shogi most notably by the algorithm introduced as *AlphaZero*. Subsequently, independent researchers and enthusiasts partially replicated the achievement in the aforementioned domains. Moreover, different board game types have been evaluated by either exclusively using reinforcement, supervised learning or a mixture between the two. The main hindrance for achieving good performance for complex games is the data requirement for reinforcement learning and the associated hardware requirements. In this work we provide a throughout overview in applying reinforcement learning for the chess variant crazyhouse while aiming to reduce the amount of required data to achieve significant progress. Additionally, we present an extended input representation to support additional seven chess variants and evaluate whether it is advantageous to train a model for multiple variants at once. Lastly, the playing behaviour after 50^{th} model updates in the reinforcement learning loop is evaluated in 100 matches between the latest development version of the strong open source chess engine *Stockfish*. We demonstrate that *CrazyAra* surpassed *Stockfish* in crazyhouse (61 wins, 3 draws, 36 defeats) by using one million self-play games which were generated in 18 days using three V100 GPUs when starting with a network trained on human games.

Keywords: Reinforcement Learning, Crazyhouse, Chess, Deep Learning, Monte-Carlo Tree Search

Zusammenfassung

Kürzlich wurden maßgebliche Erfolge beim automatisierten Erlernen der Brettspiele Go, Schach und Shogi, durch den als *AlphaZero* eingeführten Algorithmus erzielt. Anschließend wurden die Ergebnisse von unabhängigen Forschern und Enthusiasten in den zuvor genannten Spielen teilweise repliziert. Darüber hinaus wurde, entweder unter ausschließlicher Nutzung von Reinforcement Learning oder einer Mischung mit Supervised Learning, die Anwendung auf weitere Brettspieltypen ausgeweitet. Das größte Hindernis für ein erfolgreiches Reinforcement Learning bei komplexen Spielen ist die benötigte Datenmenge und den damit verbundenen Hardwareanforderungen. In dieser Arbeit geben wir einen umfassenden Überblick zur Anwendung von Reinforcement Learning für die Schachvariante Crazyhouse mit dem Ziel, bei geringer gegebener Datenmenge, signifikante Fortschritte zu erzielen. Außerdem präsentieren wir eine erweiterte Eingabedarstellung zur Unterstützung von sieben weiteren Schachvarianten und bewerten, ob es vorteilhaft ist, ein Modell gleichzeitig für mehrere Varianten zu trainieren. Abschließend wird das Spielverhalten nach dem 50. Modell-Update des Reinforcement Learning Loops in 100 Partien zwischen der aktuellen Entwicklungsversion der starken Open-Source-Schach-engine *Stockfish* bewertet. Wir zeigen, dass *CrazyAra*, *Stockfish* in Crazyhouse aktuell überlegen ist (61 Siege, 3 Remis und 36 Niederlagen). Ausgehend von einem trainierten Netzwerk durch menschliche Partien wurden hierfür eine Millionen Partien in 18 Tagen mittels drei V100-GPUs generiert.

Stichworte: Reinforcement Learning, Crazyhouse, Chess, Deep Learning, Monte-Carlo Tree Search

Contents

1 Introduction

1.1 Motivation

The chess variant crazyhouse introduces the dropping rule into regular chess and enjoys increasing popularity in recent years, primarily on online chess servers. As the single player version of bughouse, it features a higher move complexity than regular chess. In previous work [5, 6] about supervised learning for crazyhouse it became apparent that neural networks are well suited for learning this complex variant. Now we extend the supervised learning into the reinforcement learning domain.

Crazyhouse consists of certain criteria which make it particular suitable for reinforcement learning and for training with neural networks: the games are usually much shorter in terms of game length and small mistakes can often be directly exploited. Furthermore, the element of the endgame is avoided because pieces never fully leave the board. Therefore, no crazyhouse tablebases exist or are required. Games almost always end with a decisive result in the form of a forced checkmating sequence and only rarely in a draw by perpetual check.

Additionally, we aim to improve data efficiency by validating new proposed ideas after the publications on *AlphaZero* as well as contributing new methods.

1.2 Problem Formulation

Neural networks in combination with MCTS have been utilized in many complex board games such as chess, shogi and Go successfully. The chess variant crazyhouse as a hybrid between classical chess and shogi inherits many properties from both game types. Due to the strong tactical aspect, the highly developed open source engine *Stockfish* has

surpassed the playing strength of human expert players[1]. However, the high branching factor severely limits its search space in depth.

We formulate three research questions for this work. First, we are going to approach the possibility of neural networks playing on the same or higher level than *Stockfish*. We approach this question by employing reinforcement starting from a neural network which was trained on human expert games similar to the *AlphaGo* [32] project. Second, the crazyhouse variant has not been as extensively studied as classical chess and the first move advantage by White is known to provide the first player a greater edge than for classical chess. Thus, we are going to provide some information in this area by analyzing a list of commonly played crazyhouse openings over the course of reinforcement learning. Third, in the case of *AlphaZero*, neural networks have only been trained on a single game type separately but not on multiple games at once with a single model. Having a single model for multiple game types is overall more desirable. We evaluate the possibility of reaching this goal by comparing the performance of training different models on three chess variants individually and a single model on the combined data set.

1.3 Outline

This thesis is structured as follows: first, we revisit the overall optimization objective, the core technical parts of the MCTS algorithm and the neural network architecture which will be used in the remaining work. Next, we go over related work, specifically for chess and crazyhouse, as well as publications which build upon or analyzed the *AlphaZero* algorithm. Subsequently, we extend the input output representation of the neural network to seven other commonly played chess variants and evaluate the practicability of effectively training multiple variants with the same model.

Later, in the main part of this work, we describe the reinforcement learning setup for the variant crazyhouse which includes both a summary of hyper-parameters as well as a specification of hardware and software. In an empirical evaluation we inspect the playing strength progression over the course of reinforcement learning as well as the playing behaviour in the opening stage. The thesis concludes with a 100 game match-up with *Multi-Variant-Stockfish*, the strongest classical engine for crazyhouse. Lastly, we give a brief summary and outlook for potential future work.

[1] `https://github.com/ddugovic/Stockfish/issues/147`, accessed 2019-12-30

2 Background

This chapter summarizes the technical aspect of the MCTS algorithm as introduced in [32] and later refined in [30], [29] and [6]. Next, crazyhouse is formalized as a reinforcement learning problem and compared with a supervised learning approach. Lastly, the convolutional network architecture is given which will be used in the rest of the work.

2.1 General Concept

Be it for supervised or reinforcement learning, the overall optimization objective can be defined as an actor critic loss function which consists of a combined value and policy loss:

$$l = \alpha(z - v)^2 - \pi^T \log \mathbf{p} + c\|\theta\|^2. \tag{2.1}$$

The parameter z describes the target value, v the predicted value, π the target policy distribution and $\mathbf{p}$ the predicted policy by the neural network f_θ. The hyperparameter α is used to define a custom weighting between the value and policy loss and c acts as a L_2 regularization constant of the model parameters θ.

It is typically realized by a shared neural network f_θ with two output heads and optimized by a variant of stochastic gradient descent, for instance by Stochastic Gradient Descent with Neterov's Momentum (NAG; [2]).

2.2 Monte-Carlo Tree Search

While in games like Go, Monte-Carlo Tree Search (MCTS) has been the algorithm of choice for a long time [13], for games with a more tactical nature, like chess and shogi, mini-max search with $\alpha\beta$-pruning, as popularized by *DeepBlue* [3], has dominated the engine scene

Monte-Carlo based approaches estimate the value function by averaging returns over a series of rollouts. An essential break-through was achieved in *AlphaGo* [32] by integrating a deep neural network f_θ into MCTS, in the case of the Upper Confidence Bounds for Trees algorithm (UCT; [21]). The main concept is to sample promising moves more often and to find a balance between exploration and exploitation with a variable depth search. At each state s_t for every time step t a new action a_t is selected according to the UCT-formula (2.2) until either a new unexplored state s^* or a terminal node s_T is reached.

$$a_t = \text{argmax}_a(\text{Q}(s_t, a) + U(s_t, a)) \quad \text{where} \quad U(s, a) = c_{\text{puct}} P(s, a) \frac{\sqrt{\sum_b N(s, b)}}{1 + N(s, a)} \tag{2.2}$$

The new unexplored state s^* is expanded and evaluated by the neural network f_θ. Next, the predicted policy $P(s, a_i)$ is assigned to every possible action a_i and the state evaluation v^* is back-propagated along the visited search path. In case of a terminal state s_T, the constant evaluation of either -1, $+1$ or 0 is used instead. The value evaluation v^* is multiplied by -1 after every step and updates the respective Q-values by a simple moving average (SMA):

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

Unvisited nodes are treated as losses and assigned a value of -1.

Next, we briefly go over the MCTS hyperparameters which refer to [30] and [6].

2.2.1 Default Parameter Settings

Weighting constant c_{puct} The exploration constant c_{puct} acts as a weighting parameter between Q- and U-values and is scaled according to the number of visits of a particular

node:

$$c_{\text{puct}}(s) = \log \frac{\sum_a N(s,a) + c_{\text{puct-base}} + 1}{c_{\text{puct-base}}} + c_{\text{puct-init}} \tag{2.4}$$

where we chose a $c_{\text{puct-base}}$ of $19{,}652$ and $c_{\text{puct-init}}$ of 2.5 as in [6].

Move temperature τ After a given number of simulations $N(s_0) = \sum_b N(s_0, b)$ has been reached, the new target policy $\pi(a|s_0)$ for reinforcement learning is constructed based on the number of visits for all direct child nodes. For move selection an exponential temperature scaling factor τ is applied on the policy $\pi(a|s_0)$ and the next action a is sampled from $\pi'(a|s_0)$:

$$\pi(a|s_0) = \frac{N(s_0,a)}{\sum_b N(s_0,b)}, \quad \pi'(a|s_0) = \frac{N(s_0,a)^{\frac{1}{\tau}}}{\sum_b N(s_0,b)^{\frac{1}{\tau}}}. \tag{2.5}$$

A temperature value of $\tau = 0$ corresponds to applying the argmax operator.

Exploration noise $\mathbf{Dir}(\alpha)$ To encourage the exploration of moves outside of the initial neural network policy $\mathbf{p}$ during reinforcement learning, we apply a Dirichlet noise $\mathbf{Dir}(\alpha)$ with a factor ϵ of $25\,\%$ and $\alpha = 0.2$ on the policy distribution of the root node $\mathbf{p_{s0}}$.

$$\mathbf{p'_{s0}} = (1-\epsilon)\ \mathbf{p_{s_0}} + \epsilon\ \mathbf{Dir}(\alpha) \tag{2.6}$$

We notice that for tournament play, adding noise on the neural network policy $\mathbf{p}$ usually increases the activation for blunder moves. Therefore, we instead increase overall exploration by applying an exponetial scaling with a temperature value $\tau > 1$:

$$p'(a|s) = \frac{p(s,a)^{\frac{1}{\tau}}}{\sum_b p(s,b)^{\frac{1}{\tau}}}, \tag{2.7}$$

Mini-Batch Size The mini-batch size defines how many positions are buffered and allocated for neural network inference. Larger mini-batch sizes usually improve GPU utilization at the cost of a lower node statistic update frequency. The original proposed value mini-batch size for *AlphaZero* [29] was set to eight.

Virtual Loss The node selection formula (2.2) behaves deterministic. To enable the allocation of mini-batch sizes > 1 and multi-threaded search, we temporarily affect all visited nodes by a virtual loss of 3 which emulates three total extra losses on the specific nodes. After the update of the Q-values, the effect of the virtual loss is reverted again.

Centipawn Conversion To allow a better comparison with traditional search engines, we convert the Q-value evaluation into the centipawn (cp) metric as in [6]:

$$\text{cp} = -\frac{v}{|v|} \cdot \log \frac{1 - |v|}{\log \lambda}, \tag{2.8}$$

where λ is set to 1.2.

Transposition Table We also make use of transposition tables as described in [6]: A pointer to every node within the search tree is stored in a hash-table to allow copying the value evaluation and policy prediction without requesting the neural network on the same position multiple times.

Time dependent Search Furthermore, we reuse our basic time management system from [6] in order to conduct tournament matches at arbitrary time controls and on different hardware. The system determines a default constant move time for the first 40 moves and later switches to a proportional time management.

2.2.2 Comparison with Minimax Search and $\alpha\beta$-Pruning

MCTS comprises a different set of strength and weakness compared to minimax search with $\alpha\beta$-pruning. The main advantage of MCTS when coupled with a neural network policy is its ability to select its nodes more sensibly and possibly reach higher depth than traditional engines which generate more than thousand times more nodes. However, the policy might also contain blind spots for certain moves which becomes specially apparent at tactical sequences. *Multi-Variant-Stockfish*, as a representative of minimax search, has a much lower risk at miss-evaluating low depth tactical sequences. The evaluation speed of *Stockfish* for a single position is significantly faster and relies among others techniques on piece-square tables. To handle the high move complexity of crazyhouse, it further employs multiple search heuristics such as quiesence search, razoring, futility pruning,

null move pruning, probcut, internal iterative deepening, SEE pruning and LMR history pruning. It does not contain an explicit policy function and uses the min-max operator rather than averaging for its evaluations. As a result the evaluation progression of a game is usually not as smooth as for MCTS engines. MCTS search is capable of reusing large parts of the previous search tree if the game variation follows the expected principal variation whereas $\alpha\beta$-engines need to store evaluations in the hash table for future turns.

2.3 Crazyhouse as a Reinforcement Learning Problem

Crazyhouse, as well as other two-player board games like chess, Go, shogi and Connect-Four, is a well defined environment which has a discrete action space with a fully observable discrete state space. In consequence, it is well suited to be directly converted to a Markov-Decision-Process (MDP) where each Markovian state represents the current board position, as further described in Chapter 4. Despite the higher move complexity compared to classical chess, the average game length is reduced and the game results tend to be more decisive. Additionally, the environment can be perfectly simulated on low hardware and environment interactions are simulated cheaply.

The general reinforcement learning problem can be formalized as to maximize the total discounted reward. For MDPs the goal is to find the optimal value function $v_*(s)$ or rather state action values $q_*(s, a)$ according to the Bellman equations of optimality [34]:

$$v_*(s) = \max_a \sum_{s',r} p(s', r|s, a) \left[r + \gamma \, v_*(s') \right], \tag{2.9}$$

$$q_*(s, a) = \sum_{s',r} p(s', r|s, a) \left[r + \gamma \max_{a'} q_*(s', a') \right] \tag{2.10}$$

In this work, crazyhouse is represented as a model-based, actor-critic problem with the goal to find the optimal policy and value function in a mini-max game where the value return is flipped on every ply in the search tree. No discount is used ($\gamma = 1$) and the rewards are sparse and binary in the form of terminal nodes. To allow a better approximation of the actual state value, we average the state value estimate provided by the neural network estimates and the terminal returns.

2.4 Supervised Learning vs Reinforcement Learning

Learning crazyhouse in a supervised learning and reinforcement learning setting differs in several ways which are now discussed in more detail: in supervised learning only sparse labels are available to learn the policy and value function. The policy target is a one-hot-encoded vector which replicates the move that was played in a given position. By providing different target moves at varying frequencies, the model will converge to a distribution which represents all given targets with the goal to generalize well to unseen samples. The policy is usually optimized, using the sparse version of the cross-entropy loss.

The value target is also binary or threefold, if a draw result is possible. We assign every position as winning $(+1)$, drawn (0) or losing (-1) depending on the final game result, regardless of the game progression. The value output of the neural network is clipped into the numerical range of e.g. $[-1, +1]$ using a hyperbolic tangent function or a sigmoid function if a range of $[0, 1]$ is used instead. The value loss is formalized as a mean squared error. In supervised learning there are usually a broad diverse number of different players which reduces the chance of over-fitting and missing key concepts.

In reinforcement learning we deal with a new dimension of complexity. Mainly finding the right trade-off between exploration and exploitation because the reinforcement learning agent is at risk of missing out important lines. Generating your own samples as the new optimizing targets can also have some bad recursive effect or lead to forgetting of general knowledge. For example, if only the winning progression against the previous network is observed during training, you might end up in a rock - paper - scissors scenario. In this case, the new network learns to exploit the weaknesses of the previous iteration without generalizing and increasing in overall playing strength.

In reinforcement learning the policy target is the node visit distribution after MCTS search. The cross entropy loss is therefore calculated using the full distribution. Moreover, you also gather the Q-values as an additional feature which is normally unavailable in supervised learning.

Reinforcement learning, on the other hand, also allows generating an infinite amount of training samples in theory and an automatic re-calibration of states spaces which have been previously accessed incorrectly. Nevertheless, there is no guarantee for continuous progress and due to missing any external influence there is also the risk of divergence.

2.5 Convolutional Neural Networks

Convolutional neural networks (CNNs) [12, 23] are primarily used in the computer vision or natural language processing domain and characterized by employing the space and shift invariant convolution operation. In the context of chess, the board state can be encoded as an 8×8 multi-channel image and essentially treated as a vision problem. The original *AlphaZero* network architecture [29], as shown in Table 2.2, belongs to the residual networks as popularized by [16]. Moreover, it is a shared neural network with a joint residual tower followed by a value (Table 2.3) and policy head (Table 2.4) and can be characterized by a large parameter count similar to WideResnet [42]. In contrast to common CNN architectures, local pooling layers are avoided and the spatial size is preserved.

The final playing strength of the model coupled with MCTS is determined by multiple factors which include the achieved performance on the validation set, the inference speed and memory consumption. For the subsequent chapters we prefer the RISEv2 mobile architecture (Table 2.1) as introduced in [6] over the original *AlphaZero* architecture. RISEv2 mobile combines the residual block of MobileNet v2 [28] with a linearly increasing number of filter like in the pyramid-architecture [15] as well as global squeeze excitation layers (SE; [17]). According to Table 2.5, RISEv2 mobile achieved similar performance on the lichess.org crazyhouse data set while having a fifth of the number of parameters and yielding more than two times evaluations per second on GPU and three times more evaluations on CPU (Figure 2.1). The median number of evaluation position per second, commonly referred as nodes per seconds (NPS), has been measured for three seconds on fifteen benchmark positions on three different backends.

Table 2.1: RISEv2 mobile / 8-value-policy-map-mobile architecture: 13 $\times$ 256 as presented in [6]

Layer Name	Output Size	RISEv2 mobile 40-Layer

`conv0`
`batchnorm0`
`relu0`
	$256 \times 8 \times 8$	conv $3 \times 3,\ 256$

`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`
	$256 \times 8 \times 8$	$\begin{bmatrix} \text{(SE-Block, } r = 2) \\ \text{conv } 1 \times 1, 128 + 64\,x \\ \text{dconv } 3 \times 3, 128 + 64\,x \\ \text{conv } 1 \times 1,\ 256 \end{bmatrix} \times 13$

value head *policy head*	1 5184	Table 2.3 Table 2.4

Table 2.2: AlphaZero's network architecture: 19 $\times$ 256 as shown in [6]

Layer Name	Output Size	AlphaZero Resnet 39-layer

`conv0`
`batchnorm0`
`relu0`
	$256 \times 8 \times 8$	conv $3 \times 3,\ 256$

`res_conv0_x`
`res_batchnorm0_x`
`res_relu0_x`
`res_conv1_x`
`res_batchnorm1_x`
`shortcut + output`
`res_relu1_x`
	$256 \times 8 \times 8$	$\begin{bmatrix} \text{conv } 3 \times 3,\ 256 \\ \text{conv } 3 \times 3,\ 256 \end{bmatrix} \times 19$

value head *policy head*	1 5184	Table 2.3 Table 2.4

Table 2.3: Value head for different architectures with n-channels as shown in [6]

Layer Name	Output Size	Value Head N-channels

`conv0`
`batchnorm0`
`relu0`
	$n \times 8 \times 8$	conv 1×1, n

`flatten0`
`fully_connected0`
`relu1`
	256	fc, 256

`fully_connected1` `tanh0`	1	fc, 1

Table 2.4: Policy head type policy-map as shown in [6]

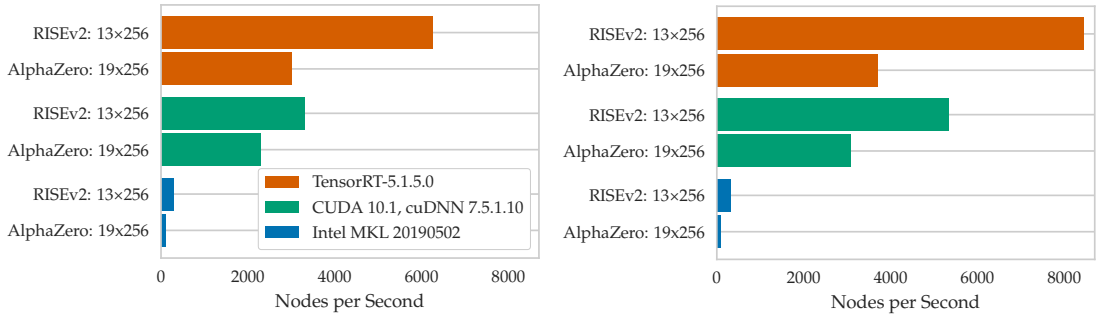
Layer Name	Output Size	Policy Map

`conv0`
`batchnorm0`
`relu0`
	$256 \times 8 \times 8$	conv 3×3, 256

`conv1`
`flatten0`
`softmax0`
	5184	conv 3×3, 81

Table 2.5: Performance metrics for different models on the lichess.org crazyhouse validation set, as shown in [6]

Validation Metrics	**AlphaZero 19 $\times$ 256 8-value-policy-map**	**RISEv2 13 $\times$ 256 8-value-policy-map-mobile**
Combined Loss	1.1964	**1.1925**
Policy Loss	1.2008	**1.1968**
Value Loss	**0.7577**	0.7619
Policy Accuracy	0.6023	**0.6032**
Value Accuracy Sign	**0.6899**	0.6889
Model Parameters	23,288,576	**4,820,113**
Disc Space	95.1 MB	**19.4 MB**



(a) Batch size 8

(b) Batch size 16

Figure 2.1: Seed comparison between AlphaZero's 19x256 network architecture and the RISEv2 13x256 architecture, measured using CrazyAra 0.7.0 (MXNet 1.6.1, GTX 1080ti + AMD® Ryzen 7 1700 eight-core processor and for IntelMKL: Intel® Core™ i5-8250U CPU @ 1.60GHz) on 15 benchmark positions

3 Related Work

This section provides an overview of relevant work in the field of chess as well as achievements in other board games before and after the publications on *AlphaZero* [30].

An initial success of neural network based evaluation functions was achieved by Gerald Tesauro and his Gammon playing program called *TD-Gammon* [35] which learnt to play slightly below top level human players after generating $1{,}500{,}000$ self-play Gammon games.

At a similar time as *AlphaGo* [31], the Go playing program named *darkforest* [37] was developed by Facebook and used a deep convolutional neural network with a pure pattern-matching approach. Later for *darkfmcts3* [37], a MCTS search engine was added and the program achieved the playing strength of an advanced amateur Go player. Next, followed the publications by DeepMind on *AlphaGo* [31], and its successors *AlphaGoZero* [32] and *AlphaZero* [30] which learnt the board games Go, chess and shogi without prior human knowledge.

3.1 Neural Networks in Chess

The first successful attempts of employing neural networks in the game of chess, as in the programs *Giraffe* [22] and *DeepChess* [7], have been using classical fully connected feed forward networks and usually made use of supplementary hand-crafted feature inputs. If additionally reinforcement learning was used, then programs like *KnightCap* [1] and *Meep* [39] relied on Temporal-Difference (TD) learning for adjusting their set of parameters.

For the game variant crazyhouse as well classical chess, suicide chess and atomic chess, in the work by Droste and Fürnkranz [8] the piece value tables were learnt by applying TD learning in a reinforcement learning setting. They adapted the open-source $\alpha\beta$-search

engine *Sunsetter*[1] for their research and demonstrated an improvement over the original values both for the *Sunsetter* and *Deep Sjeng*[2] engine (version 11.2). The open source engine *Stockfish* had not been extended to *Multi-Variant-Stockfish* at the time of publication yet. In contrast to learning specific heuristic value parameters, *CrazyAra* assesses a board position exclusively through a deep convolutional neural network. This facilitates capturing the element of initiative which plays a major role in the crayzhouse variant.

As a collaborative open source project with the goal to replicate *AlphaZero* for Go, Gian-Carlo Pascutto founded the *Leela Zero*[3] software project in 2017. The project was later forked by Gary Linscott in 2018 and renamed into *Leela Chess Zero*[4] to address the game of chess in a similar manner. Subsequently, Alexander Lyashuk and others fully rewrote the engine and abbreviated the program name into *Lc0*[5]. *Lc0* won the Superfinal against *Stockfish* in season 15[6] of the Top Chess Engine Championship (TCEC).

In 2019, Gordon Chi evaluated the performance of deep residual networks in a crazyhouse playing program *SixtyFourEngine* [5]. Moreover, Czech et al. demonstrated with *CrazyAra 0.6.0* [6] the possibility of reaching a playing level above human expert strength while only relying on human games in a supervised learning setting.

3.2 Neural Networks in other Bord Games

The game of Go as well as simple game types such as Connect-Four and Othello attracted most attention in academic research. The publications in this area focused on either replicating the results by DeepMind, studying the effects for different hyper-parameter values or proposing modification to the default setup.

The work by Wang et al. [40] studied the effect of twelve different *AlphaZero* hyper-parameters (e. g. learning rate, c_{puct}, MCTS rollouts . . .) under the three objectives: training loss, time-cost and playing strength. Their experiments were done on the 6×6 Othello game and used the python-only open-source implementation *AlphaZeroGeneral*[7].

[1] `http://sunsetter.sourceforge.net/`, accessed 2019-12-29
[2] `https://www.sjeng.org/indexold.html`, accessed 2019-12-29
[3] `https://zero.sjeng.org/`, accessed 2019-12-30
[4] `https://github.com/glinscott/leela-chess`, accessed 2019-12-30
[5] `https://github.com/LeelaChessZero/lc0`, accessed 2019-12-30
[6] `https://tcec-chess.com/articles/Sufi_15_-_Sadler.pdf`, accessed 2019-12-30
[7] `https://github.com/suragnair/alpha-zero-general`, accessed 2019-12-30

During a successful attempt of replicating *AlphaZero* for the game Connect-Four, the researchers Abrams et al. proposed, inter alia, using a mixture of the Q-value and the final game result z as the new value target[8]. The new technique introduces a mixture parameter, called the *Q-value-ratio*, as a new hyper-parameter. This can be seen as combining a boots-trap sample with a Monte-Carlo return and has been shown to potentially speed-up learning by reducing variance at the cost of introducing bias [34].

Morandin et al. developed the Go playing program, *Sensible Artificial Intelligence* (SAI) [26], with the goal to improve the playing behaviour under disadvantageous positions. They changed the default value target into modelling the winrate as a sigmoid function of the bonus points in respect thereof. Their evaluation was conducted on a smaller 9×9 Go board.

The main run for Go by DeepMind [29] lasted several days and used $5{,}000$ Tensor processing units (TPUs) resulting in about 41 TPU-years. A Facebook research group replicated this performance by using $2{,}000$ V100 GPUs for about 14 days. The corresponding program *ELF OpenGo* [38] was open-sourced and its playing strength was verified with a $20 : 0$ score against global top professional Go players. Later in 2019, David J. Wu claimed to have achieved a 50x reduction in computation and to have surpassed the playing strength of both *ELF OpenGo* and *Leela Zero*. His open-sourced playing program *KataGo* was trained on 27 V100 GPUs for 19 days, totalling in about 1.4 GPU-years. His proposed improvements include non-domain-specific as well as domain-specific techniques: *Playout Cap Randomization*, *Forced Playouts with Policy Target Pruning*, *Global Pooling*, *Auxiliary Policy Targets* and *Auxiliary Ownership and Score Targets*.

His game randomization configuration which samples the first moves directly from the raw network policy, influenced the reinforcement learning randomization setting of this work as described in Section 5.2.

[8] https://medium.com/oracledevs/lessons-from-alphazero-part-4-improving-the-training-target-6efba2e71628, accessed 2019-12-30

4 Multi Chess Variant Training

This chapter introduces several chess variants and their main features with the idea of training a single neural network on multiple variants at once. Consequently, the former input and output representation for crazyhouse has to be extended.

Many chess variants have been developed for the game of chess. While some fundamentally modify the rules of chess and the overall objective, others keep most of the classical chess rules. To guarantee the best conditions for multi variant training, the three variants chess960, King of the Hill and Three-check were chosen which stay close to classical chess rules, share the same legal move generation and a similar starting position.

4.1 Chess Variants

Chess960 also known as Fischer Random Chess, invented by former world champion Bobby Fischer, is not a different variant in itself but a meta-variant which can be applied to almost any chess variant. It follows the same rules as the defined chess variant, for instance classical chess, but is defined by using one of 960 unique random starting positions where the bishop pair remains on opposite-color squares and the king is placed in between the two rooks. Castling remains possible on either king or queen side if the intercepting pieces have been moved.

The King of the Hill (koth) chess variant retains all regular chess rules with the addition of winning the game if the king enters one of the four central squares.

Three-Check uses the same rules set as classical chess but allows winning either by checkmate or putting the king of the opponent into check for the third time. Giving a check with multiple pieces at once, for instance due to a discovered check, is counted as a single check.

4.1.1 Data Set

As in previous work [6], the training data originates from the lichess.org database [36]. The Elo distributions are slightly different among each variant, therefore we chose all available games from August 2013 until August 2018 in which both players belong to the top ten percentile of the respective cumulative Elo distribution. This results in an Elo threshold of 1950 Elo for Chess960, 1925 Elo for King of the Hill and 1900 Elo for Three-Check. The games from April 2018 and August 2018 were withhold for validation and testing.

4.2 Unified Representation for Multiple Chess Variants

The extended input and output representation for multiple variants is based on [6] and does not encode a move history. All features are re-scaled to be in the range $[0, 1]$ by using the following soft maximum boundary constants: 16 maximum pocket pieces for a single piece-type, 500 maximum moves and 50 as the maximum no progress counter. Table 4.1 summarizes the input representation which supports all nine chess variants on lichess.org. Four additional 8×8 planes are used to describe the remaining checks to win in the Three-check variant and eight 8×8 planes present the current active variant in one-hot encoded form. A single plane provides information whether the game originated from a 960 starting position and hence uses modified castling rules.

For the policy description, as shown in Table 2.4, we reuse the policy map representation of crazyhouse from [6] and add three additional planes to allow king promotion moves. King promotion is legal and commonly played in the antichess variant. We favor the policy map over the plane vector representation due to generally resulting in a lower policy loss [6].

To ensure that the information of the current active variant is sufficiently recognized, the last nine planes of the input representation was given as an additional input to each residual block and added to the final 256 feature plane representation before the value and policy head.

Table 4.1: Multi variant plane representation. The features are encoded as a binary maps and features with $*$ are single values set over the entire $8\times$ 8 plane. 13 additional planes account for describing the current variant as well as the amount of checks given. Entries in **bold** vary from the default crazyhouse representation.

Feature	Planes	Type	Comment
P1 piece	6	bool	order: {`PAWN`, `KNIGHT`, `BISHOP`, `ROOK`, `QUEEN`, `KING`}
P2 piece	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 and all ones for white
Total move count*	1	int	sets the full move count (FEN notation)
P1 castling*	2	bool	binary plane, order: {`KING_SIDE`, `QUEEN_SIDE`}
P2 castling*	2	bool	binary plane, order: {`KING_SIDE`, `QUEEN_SIDE`}
No-progress count*	1	int	sets the no progress counter (FEN halfmove clock)
P1 remaining-checks*	**2**	bool	3check variant: After 3 checks by one player the game ends
P2 remaining-checks*	**2**	bool	3check variant: After 3 checks by one player the game ends
is960	**1**	bool	indicates if game uses a randomized starting position
Variant	**8**	bool	one-hot, order: {`chess`, `crazyhouse`, `kingofthehill`, `3check`, `giveaway`, `atomic`, `horde`, `racingkings`}
Total	**47**		

Table 4.2: Policy map representation for multi variants. Entries in **bold** vary from the default crazyhouse representation

Feature	Planes	Comment
Queen moves	56	direction order: {N, NE, E, SE, S, SW, W, NW} with 7 lengths per direction
Knight moves	8	move order: {2N1E, 1N2E, 1S2E, 2S1E, 2S1W, 1S2W, 1N2W, 2N1W}
Promotions	**15**	piece order: {KNIGHT, BISHOP, ROOK, QUEEN, **KING**}
Drop moves	5	piece order: {PAWN, KNIGHT, BISHOP, ROOK, QUEEN}
Total	**84**	

4.3 Supervised Training

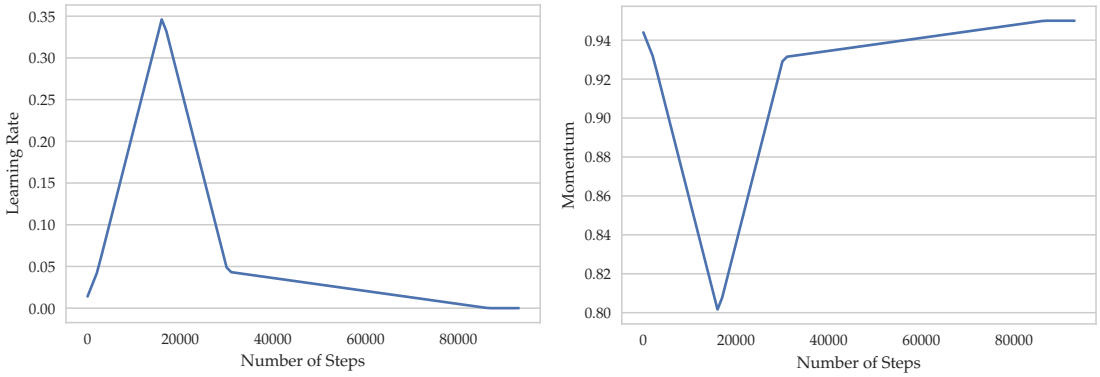
To allow a direct comparison with previous results, the supervised learning is performed under the same setup as in [6]: For the convolutional neural network architecture the RISEv2 architecture (Table 2.1) was used. The model parameters were initialized according to Xavier weight initialisation [14] with a magnitude of 2.24 and updated by stochastic gradient descent with Nesterov momentum [2]. The learning rate and the momentum parameter were modified by a single one cycle linear schedule as proposed in [33]. The corresponding maximum and minimum learning rate were set to 0.35 and 0.00001, as shown in Figure 5.1.

The α value in the combined loss (2.1) which controls the weighting of the value loss was reduced to 0.01 in order to avoid overfitting to the value target. Supervised learning proceeded with a batch-size of $1{,}024$ and a weight-decay of 10^{-4} for seven epochs. Each of the three variants was trained in a separate run as well as in a combined training run where the training data for all variants is mixed. As can be seen in Figure 4.2 the variance for the performance metrics in each variant was significantly higher in the case of multi-variant training and failed to decrease over the course of training.

The overall best results were obtained by learning all variants at once rather than individually. However, the model was unfortunately unable to capture all variants within a single model state but oscillated between different local optima (Table 4.3, column "Combi-single"). Compared to the performance metrics in crazyhouse (Table 2.5), the model converged to a higher policy loss but a lower value loss for the chess960 variant. Although the inferior policy performance seems surprising, because generally there are

Table 4.3: Performance metrics for different traing setups on the lichess.org validation set

Metric	Variant	Only	Combi-best	Combi-single
Policy Loss	chess960	1.4939	**1.4835**	1.4835
Value Loss	chess960	0.6519	**0.6504**	0.6504
Policy Accuracy	chess960	0.5136	**0.515**3	0.5153
Value Accuracy Sign	chess960	0.7204	**0.7220**	0.7220
Policy Loss	King of the Hill	1.5345	**1.4910**	2.4136
Value Loss	King of the Hill	0.7995	**0.7957**	1.3589
Policy Accuracy	King of the Hill	0.5090	**0.5184**	0.3249
Value Accuracy Sign	King of the Hill	0.6593	**0.6638**	0.5121
Policy Loss	Three-Check	1.3772	**1.3185**	2.0552
Value Loss	Three-Check	0.7598	**0.7537**	0.8746
Policy Accuracy	Three-Check	0.5588	**0.5703**	0.3672
Value Accuracy Sign	Three-Check	0.6998	**0.7037**	0.6367



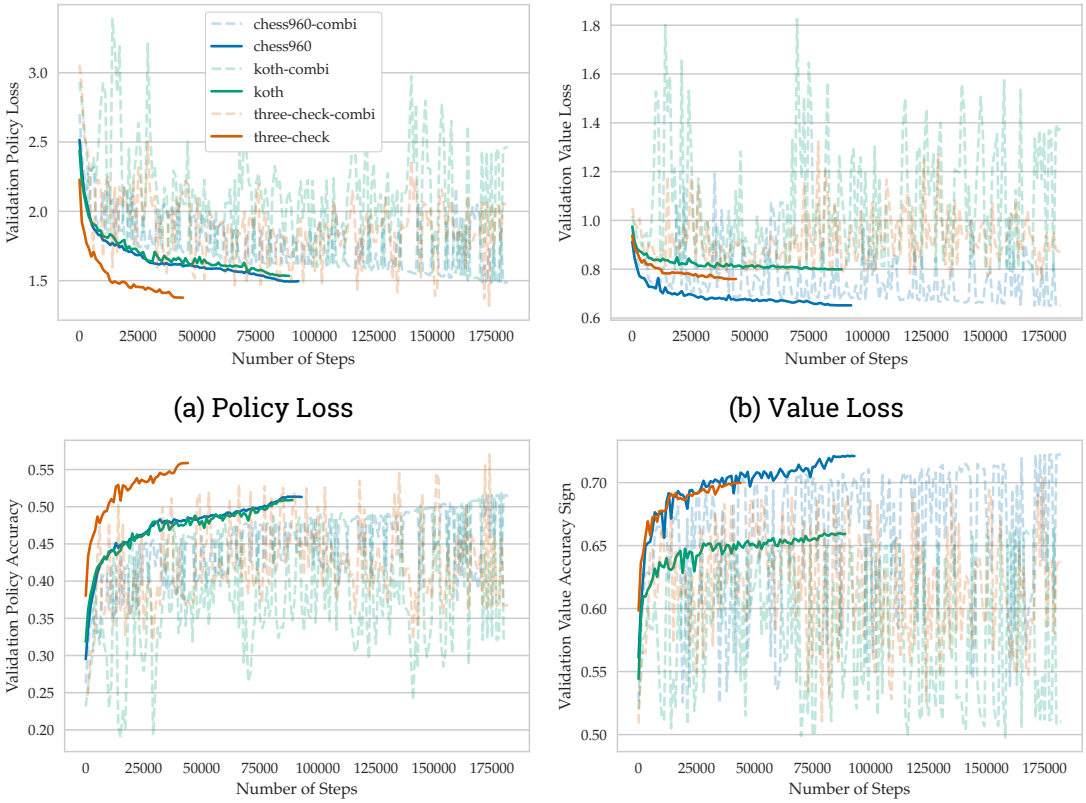
(a) Learning rate schedule

(b) Momentum schedule

Figure 4.1: Schedules used for modifying the parameters of Nesterov's optimizer, here for learning **chess960**, same as in [6]

more moves available in crazyhouse, in practice, the lines in crazyhouse are more forcing, involve more checking moves and allow fewer transpositions.

The results of this experiment suggest that training a neural network with multiple objectives at once or using transfer learning for initialization can be beneficial in cases where the training data remains similar and is limited. However, effectively learning many variants within a single model state and in a combined reinforcement learning loop appears to be difficult.



(a) Policy Loss

(b) Value Loss

(c) Policy Accuracy

(d) Value Accuracy Sign

Figure 4.2: Learning progress for the variants **chess960**, **King of the Hill** and **Three-Check** using the `lichess.org` data set. Dotted lines show the progress for learning all variants with a single model. Each training lasted for seven epochs using a batch size of $1{,}024$. The metric "Value Accuracy Sign" defines the amount of cases where the neural network predicted the correct game outcome while neglecting draws.

5 Reinforcement Learning for Crazyhouse

This chapter describes the information for adapting the engine *CrazyAra* to learn crazyhouse in a reinforcement learning setting.

5.1 Computational Layouts

Although certain formulations can be conveniently represented mathematically, it is not necessarily the best way for an algorithmic form.

Moreover, MCTS search tends to scale well with increasing number of nodes and generating self-play games accounts for the major part of computational requirements during reinforcement learning. Consequently, code optimization plays an important role at large scale reinforcement learning projects and there are a few important distinct design decision to make.

5.1.1 Monte-Carlo Layouts

The rollouts of a single MCTS batch-request can be either executed and allocated on a single thread or with as many threads as the size of the mini-batch. Using a single thread for each batch is usually preferable because it reduces the management overhead between different threads, allows a more deterministic controllable behaviour and a better scaling to high batch-sizes. In a desirable setup, two threads are sufficient to prepare the next batch for neural network inference and fully utilize the graphics processing unit (GPU).

The board state for an individual node can be stored directly in each node or locally copied from the initial root node. In the later case, the selected action must be applied after every step but less memory is required at the cost of runtime.

The UCT-formula (2.2) for node selection can be solved in vectorized form or through single values. Either way, a loop over all singular values at all times should be avoided. Instead we can abuse the deterministic and monotonous behaviour of node selection: after a node has been evaluated by the neural network, its policy distribution is sorted in descending order. For all future node selection at this node, the UCT-formula needs only to be calculated on all already visited nodes as well as the first non-visited node with the highest policy activation. For additional optimization, the sorting of the policy distribution can be delayed until the first node visit after the initial node expansion.

5.1.2 Software

An automated setup of generating hundreds of thousand self-play games with integrated learning, demands high software quality standards: preferably a 100 % crash free program without any memory leak, and a fluid switching between training and replacing the currently generating network.

The source code of *CrazyAra*[1] is published under the terms of the GNU General Public License v3.0 (GPL-3.0; [11]) and the engine is compatible with the Universal Chess Interface (UCI; [20]). Several third parties libraries are used in different areas. Neural network training and inference is executed via the deep learning MXNet library [4]. At the time of writing a single GPU back-end for TensorRT[2] and CUDA[3]/cuDNN[4] as well as two CPU back-ends for OpenCL[5] and IntelMKL[6] are supported. Representing chess boards in python is done in the python-chess library [10]. The generated training data is compressed in the Zarr [25] and z5 [27] library using the lz4[7] compression format. Vectorized operations in C++ are implemented with the Blaze library [19, 18] and move generation routines have been integrated from *Multi-Variant Stockfish* [9].

[1] `https://github.com/QueensGambit/CrazyAra`, accessed 2019-12-28
[2] `https://developer.nvidia.com/tensorrt`, accessed 2019-12-29
[3] `https://developer.nvidia.com/cuda-zone`, accessed 2019-12-29
[4] `https://developer.nvidia.com/cudnn`, accessed 2019-12-29
[5] `https://www.khronos.org/opencl/`, accessed 2019-12-29
[6] `https://software.intel.com/en-us/mkl`, accessed 2019-12-29
[7] `https://github.com/lz4/lz4`, accessed 2019-12-29

5.2 Reinforcement Learning Setting

The self-play game generation and neural network update was executed on three NVIDIA Tesla V100 on a DGX2 server instance.[8]

Two GPUs were exclusively used for selfplay game generation and one GPU was used both for game generation and updating the neural network as soon as sufficiently many samples have been acquired. On a single GPU about 45 games were generated per minute which corresponded to $1{,}000$ training samples per minute. Each training sample was produced by an average of 800 MCTS rollouts with a batch-size of 8.

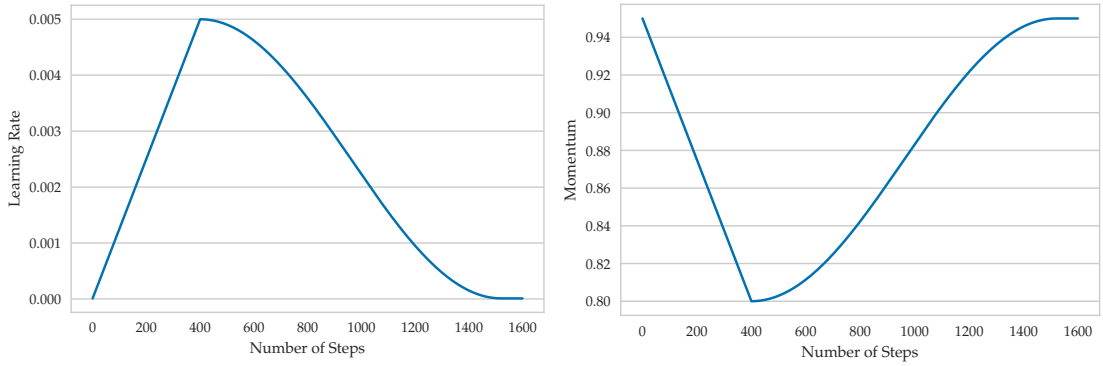
Not all of the game moves were exported as training samples and exported as further described in Section 5.2.1. A new neural network was generated every $819{,}200$ ($= 640 \cdot 128 \cdot 10$) newly generated samples. After the 10^{th} model update, the number of required samples was increased to $1{,}228{,}800$ ($= 640 \cdot 128 \cdot 15$) samples. $81{,}920$ ($= 640 \cdot 128$) of these samples were used for validation and the rest for training. Furthermore, $409{,}600$ ($= 640 \cdot 128 \cdot 5$) samples were added and randomly chosen from 5 % of the most recent replay memory data. The training proceeded for one epoch and afterwards, all samples, except validation samples, were added to the replay memory.

Games were played until completion without a resign threshold because games in crazyhouse are generally shorter than in classical chess and practically always end in checkmate or three-fold repetition. The search tree was reused during game generation, for the benefit of higher speed, but at the cost of a small reduction in exploration.

Besides this, a Q-value-ratio of 0.15 was used to include a small amount of boots-trapping. Generally, the higher the Q-value ratio, the lower the value loss became and the tendency to converge to both extremes -1 and $+1$ decreased. The value loss weighting parameter α (2.1) was set to 0.5 because of a defined value range from -1 to $+1$ instead of 0 to 1.

Rather than relying on a fixed learning rate and momentum, a Cosine-Annealing-Schedule [24] was used. Additionally, a 25 % warm-up period was pre-prended to improve the assimilation to context drifts in the given training data.

[8] TensorRT support and float16 backend was disabled due to a temporary driver incompatibility, otherwise the speed would have likely been doubled.



(a) Learning rate schedule

(b) Momentum schedule

Figure 5.1: Schedules used for modifying the parameters of Nesterov's optimizer, here for training the first 10 model updates

5.2.1 Randomization

Finding an appropriate amount of exploration appeared to be the most critical aspect for stable learning. If almost no exploration takes place, then the danger increases that the agent to skip important lines. In consequence it only learns to exploit its former generator network while losing overall generality.

According to the default *AlphaZero* algorithm [30], the move is sampled from the MCTS policy based on 800 rollouts with a temperature value of 1.0 for the first 15 moves (30 plys). This appeared to be problematic in the case of crazyhouse in which many moves are blunders and can change the final game outcome. Due to sampling for 30 plys, the final game result which corresponds to the binary value target for training became severely distorted and noisy. Therefore a move temperature value τ of 0.8 with an exponential move decay of 0.93 was used instead on every second ply. To still allow sufficient exploration and to limit the probability of sampling poor moves from the MCTS policy, the first r plys were directly sampled from the raw neural network policy with a temperature of $\tau = 1$, similar to [41]. The value r was determined by an exponential distribution with a mean of 8 plys. Because the exponential distribution has a long tail and can potentially return a high number of plys, in cases where a ply > 30 was returned it was uniformly sampled from the range $[0, 30]$ instead. The initialization was stopped either when the number of given plys has been sampled or when the next move would lead to a terminal state, making it a mate in one training sample.

(a) Policy Loss

(b) Value Loss

(c) Policy Accuracy

(d) Value Accuracy Sign

Figure 5.2: Learning progress during reinforcement learning with batch size 512

Sampling from the raw network policy not only requires little computational cost, but also helps to reduce the amount of duplicate training samples. In contrast to supervised learning, the first opening samples are almost identical for the policy target and only differ in the value target when generating games with a single set of weights.

Moreover, in order to increase exploration, in $\omega = 5\,\%$ of all cases during initialization from the raw policy, a temperature value of $\tau = 2$ was used with probability of $75\,\%$, $\tau = 5$ with probability $20\,\%$ and $\tau = 10$ for the remaining 5%.

Lastly, the number of rollouts $N = 800$ for every training sample was slightly randomized on every position:

$$N' = N + n_{\text{factor}} \cdot N, \tag{5.1}$$

where $n_{\text{factor}} \sim [-0.05, +0.05]$.

5.2.2 Changes in Configuration during the Run

The maximum learning rate started with a value of 0.005. It was then reduced by a factor 10 after the 10$^{\text{th}}$ model update and again reduced to 0.0005 after the 20$^{\text{th}}$ model update index. The number of required files for training was increased from 10 to 15 after the 10$^{\text{th}}$ model update. Besides this, the probability ω for increasing the temperature during initialisation was increased from 5% to 25% after the 40$^{\text{th}}$ model update index.

5.3 Observations

The number of MCTS rollouts is an essential hyperparameter. MCTS tends to converge a single node over time while other moves tend to get a lower visit proportion of the full visit distribution. Only increasing the rollouts without changing other parameters has the effect that exploration is reduced over time in a recursive fashion. The batch-size for neural network network inference plays a major role as well. Higher batch-sizes are usually preferable for achieving a higher number of board evaluations per second (Figure 2.1). However, at a fixed number of evaluations, lower values achieve higher playing strength because of updating the search tree at a higher frequency and achieving higher search depth.

Figure 5.2 visualizes the performance on the withould validation set after each model update. The policy loss is unable to reach a value of zero during reinforcement learning

due to not learning against sparse labels, but overall both the value and policy loss decreased compared to the initial supervised trained network. However, because the model generates its own data, a high policy accuracy was to be expected. The policy metrics almost behave flat which is mainly because of a continuous change in the training data.

Similar to [41] a gating test of 100 games (drawn games exluded) was conducted to measure the playing strength between the previously generating network and the updated version. For these games 800 MCTS rollouts with a starting move temperature of 0.6 were used. If the contender network achieved a positive score, it replaced the current generating network. Over the course of one million generated games, the new network always managed to replace its old version.

5.4 Additional remarks

Policy pruning [41], in the form of setting all policy entries < 0.03 after move sampling to 0, was disabled after the 11^{th} model update index, to ensure a higher exploration rate. Future evaluation will be needed to judge if it is beneficial in the case of crazyhouse.

Playout cap random [41] has been implemented and briefly evaluated: reaching a high amount of games seems not be an issue for crazyhouse. *Playout cap random* both reduced the generation speed for training samples and likely increased the noise for the value target due to an increasing blunder rate. It appears to be more beneficial in less tactical game types with a higher average game length.

The *integration of Q-Values for final move selection* [6] has been shown to improve playing strength at low node count. However, treating the new modified policy distribution as the new policy target led to a diminishing exploration rate. Using the information on Q-values for move selection, but not as the policy target might increase efficiency for future tests.

6 Empirical Evaluation

In this chapter we evaluate the Elo progression during selfplay, investigate the opening preferences and conclude with a final 100 match comparison between *Multi-Variant-Stockfish* and *CrazyAra*.

6.1 Elo-Development

The results are taken after one million generated self-play games ($973{,}044$ training games / $62{,}013{,}440$ training samples, $77{,}180$ validation games / $4{,}915{,}200$ validation samples) and 18 days of training[1] [2].

Besides that, due to opening sampling and the short general game length the effective cost for generating a single game was reduced. Because analyzing playing strength in direct comparison between two proceeding model updates can be misleading, the Elo was measured in an independent tournament of $2{,}750$ games using every fifth model checkpoint on random opening positions. Figure 6.1 visualizes the Elo progression extracted from the tournament results in Table 6.1. Compared to the original model based on supervised training on human games an improvement by roughly 370 Elo was achieved.

The graph also suggests that the model has not reached its final strength yet.

[1] The full compressed self-play training data resulted in 29 GB disc space, including the game logs and all intermediate model checkpoints.

[2] The generation was not performed continuously over 18 days, but sporadically interrupted due to changing hyperparameters such as the maximum learning rate.

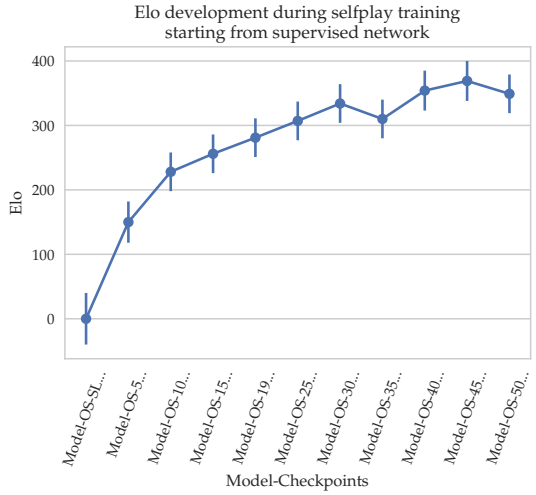


Figure 6.1: Self Elo progression during reinforcement learning

Table 6.1: Match result for different selfplay model checkpoints[a]

Rank	Model-Update-Idx	Elo	+/-	Games	Score	Draws
1	45	369	31	500	63.7%	5.0%
2	40	354	31	500	61.7%	3.8%
3	50	349	30	500	61.0%	4.8%
4	30	334	30	500	59.0%	4.8%
5	35	310	30	500	55.6%	4.0%
6	25	307	30	500	55.1%	4.2%
7	19	281	30	500	51.5%	5.0%
8	15	256	30	500	47.9%	5.0%
9	10	228	30	500	43.8%	3.6%
10	5	150	32	500	33.3%	2.6%
11	Supervised	0	40	500	17.4%	0.8%

[a] 800 MCTS rollouts, no Dirichlet noise, zero move temperature and a node policy temperature of 1.3 was used.

6.2 Opening statistics

This section gives a systematic overview of the popularity trend for 42 crazyhouse opening position during reinforcement learning. Each position is also accompanied by the white-winrate as well as the draw ratio. The openings in crazyhouse are overall similar to classical chess openings, however there are crazyhouse specific gambit opening positions like the **B01 Scandinavian: Gambit** and **B02 Alekhine: Crosky-Gambit**. Moreover, in contrast to chess, openings which involve moving the c-pawn or f-pawn (e. g English Opening, Caro-Cann Defense, Dutch Defense,...) are not considered to be viable.

6.2.1 Observations

The two most popular human openings **C50 Italian Game: Giuoco Piano** and **C50 Italian Game: Hungarian Defense** rapidly lost popularity, mainly because of an overwhelming white win rate. The **C47 Four Knights Scotch** was reached most often, presumably due to a high chance of a transposition from different opening lines. Surprisingly, the **B20 Sicilian Defense** was relatively popular; maybe this was due to a low prior knowledge in Sicilian lines from human games and because other main openings, such as the Scandinavian Defense and French Defense emerged as highly white favored also. The high proportion of the **A00 Grob Opening** and **A00 Polish Opening** are sampling side effects from the raw network policy.

Overall it can be stated that no concrete line was found for Black which equalizes White's opening advantage. Nonetheless, in openings such as the **C55 Italian: Two Knights Defence**, **C47 Four Knights Scotch**, **B00 Nimzowitsch Defense** and **B02 Alekhine** it found sufficiently many lines to limit White's win rate under the given node and randomization settings.

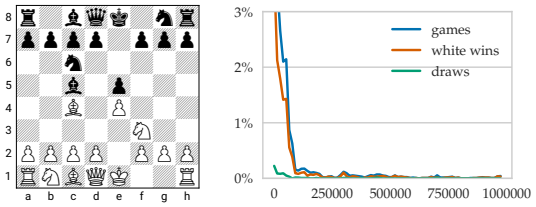
6.3 Strength comparison with Multi-Variant-Stockfish

In previous evaluations [6], we compared the playing strength of the initial network for reinforcement learning which was trained supervised on human expert games with thirteen crazyhouse engines on CPU. Further, we conducted ten games in time control 30 min + 30 s between *CrazyAraFish*, a neural network trained on *Stockfish* crazyhouse self-play games, and the latest official release (2018-11-29) of *Multi-Variant-Stockfish* at that time. *Stockfish* won the match by six wins, one draw and three losses.

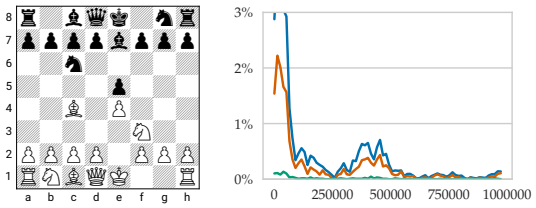
For this evaluation the same hardware and the latest development version (2019-12-03) of *Multi-Variant-Stockfish* was used which appeared to have gained ≈ 89 Elo[3] for crazyhouse in self-play. *Stockfish 10-Dev* was running on an AMD® Ryzen 7, 1700 eight-core processor × 16 with a hash-size of $4{,}096$ MB and 8 threads, resulting in 6.7 million NPS. *CrazyAra 0.7.0 (Model-OS-45)* was using the same processor as *Stockfish* on two threads and additionally a NVIDIA® GTX1080ti with a batch-size of 16 for neural network inference, yielding approximately seven up to ten thousand NPS.

[3] According to GitHub user `Matuiss2`, *Multi-Variant-Stockfish* (2019-12-03) scored 58 wins, 33 loses and 3 draws vs *Multi-Variant-Stockfish* (2018-11-29) in crazyhouse (TC: 1+1)

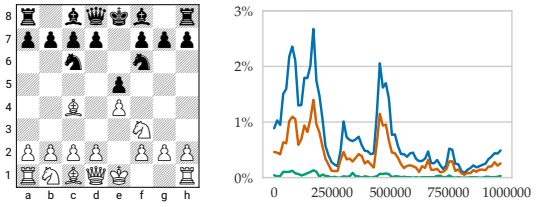
C50 Italian Game: Giuoco Piano



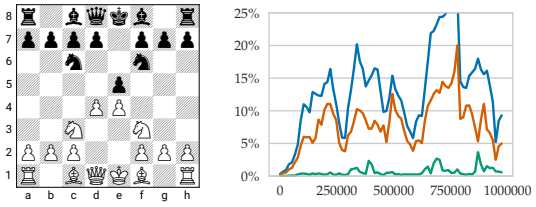
C50 Italian Game: Hungarian Defense



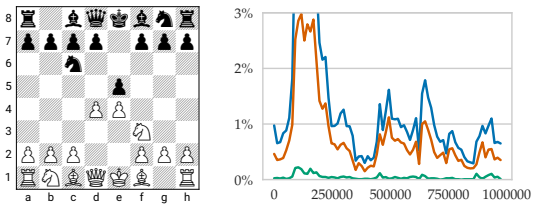
C55 Italian: Two Knights Defence



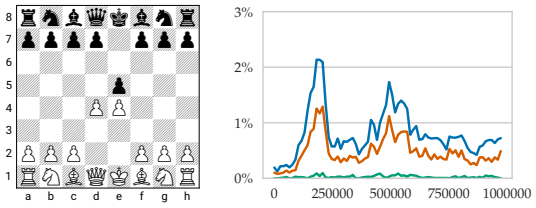
C47 Four Knights Scotch



C44 Scotch Game



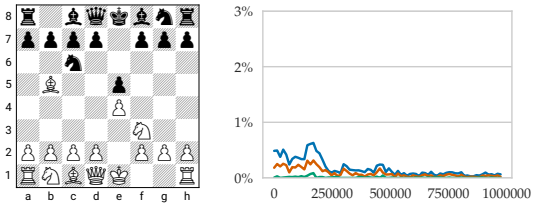
C21 Center Game



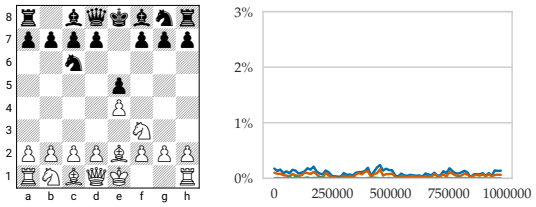
C60 Ruy Lopez

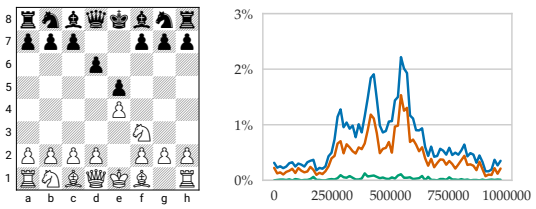
C44 Tayler Opening

C41 Philidor Defense



C00 Werewolf Gambit





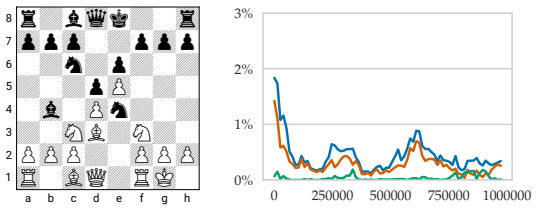
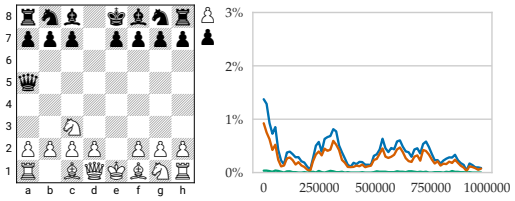
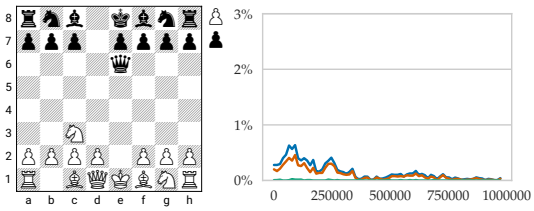


Figure 6.2: Opening statistics 1-10, x-axis shows the number of generated games, y-axis shows the amount of games, **C47 Four Knights Scotch** uses a different y-axis scaling

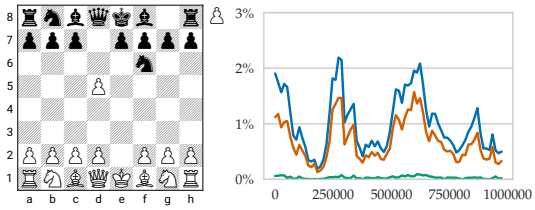
B01 Scandinavian Defense: Main-Line



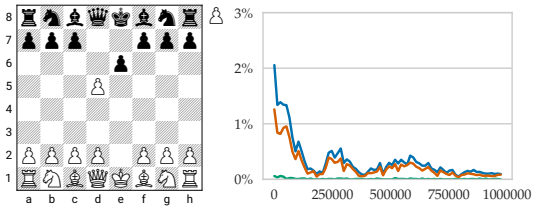
B01 Scandinavian Defense: Mieses-Kotroc Variation



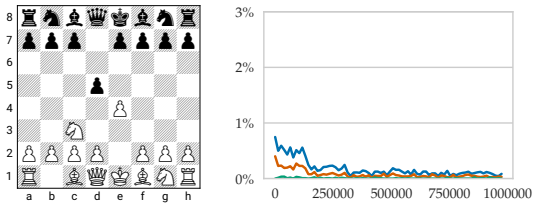
B01 Scandinavian Defense: Modern variation



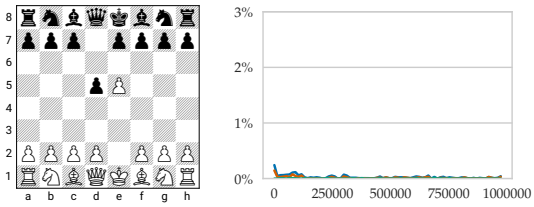
B01 Scandinavian Defense: Gambit



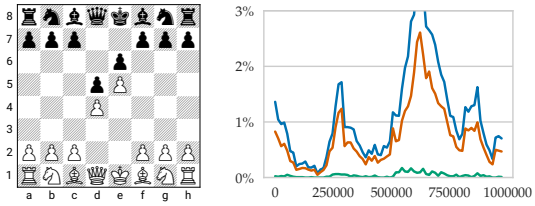
B01 Anti-Scandinavian: Nc3



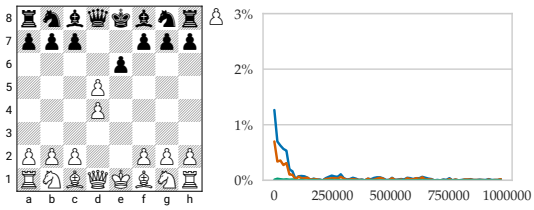
B01 Anti-Scandinavian: e5



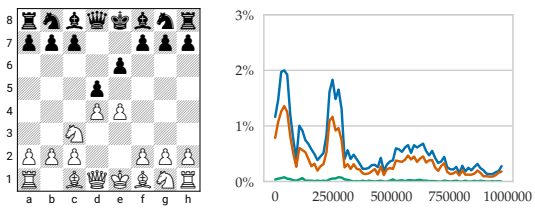
C02 French Defense: Advanced



C01 French Defense: Exchanged



C10 French Defense: Paulsen



C03 French Defense: Tarrasch

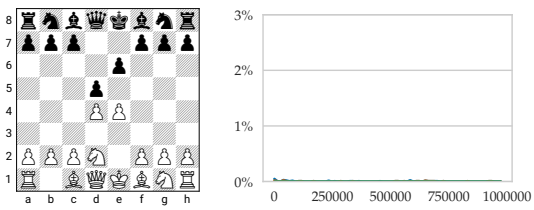
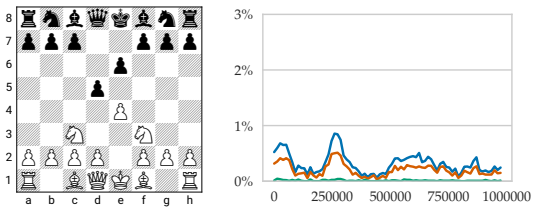
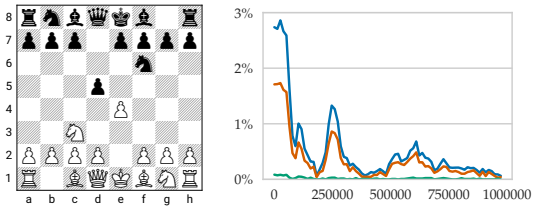


Figure 6.3: Opening statistics 11-20

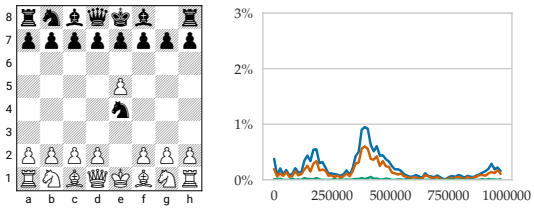
C00 French Defense: Two Knights



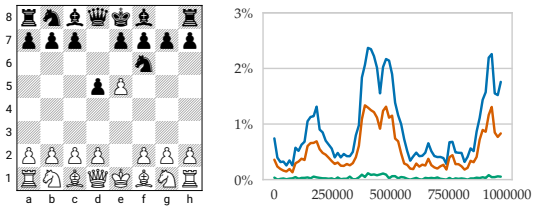
B02 Alekhine Defense: Scandinavian Variation



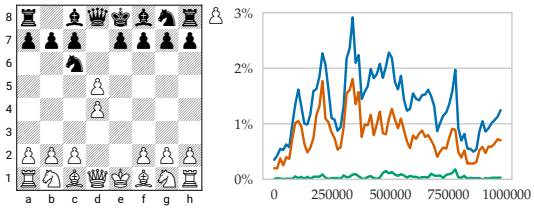
B02 Alekhine Defense: Mokele Mbembe



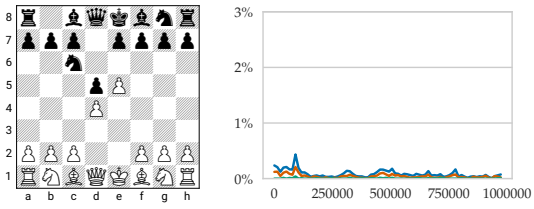
B02 Alekhine: Crosky-Gambit



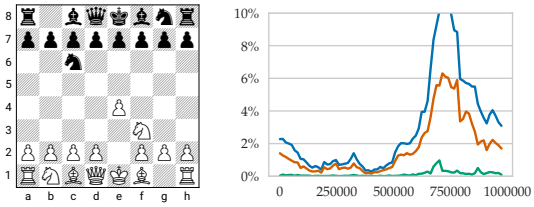
B00 Nimzowitsch Defense: Exchanged



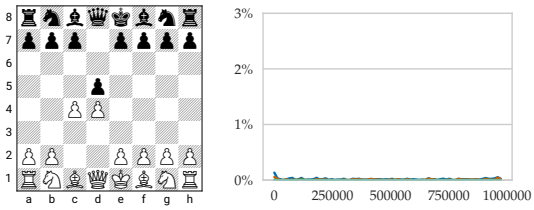
B00 Nimzowitsch Defense: Advanced



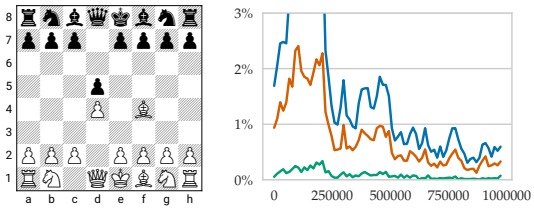
B00 Nimzowitsch Defense: Declined



D06 Queen's Gambit



D00 Queen's Pawn Game: London system



D00 Queen's Pawn Game: Levitsky Attack

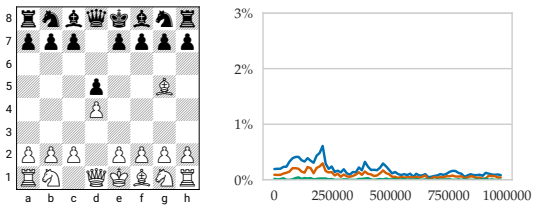
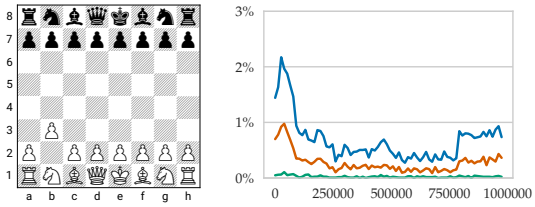
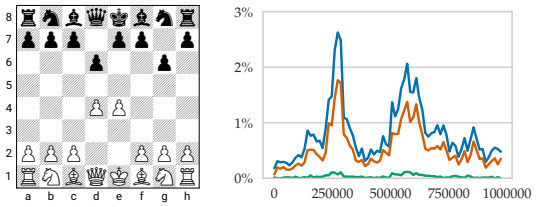


Figure 6.4: Opening statistics 21-30, **B00 Nimzowitsch Defense: Declined** uses a different y-axis scaling

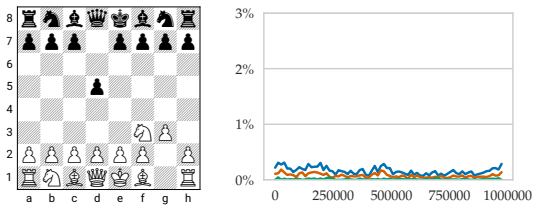
A01 Nimzo-Larsen Attack



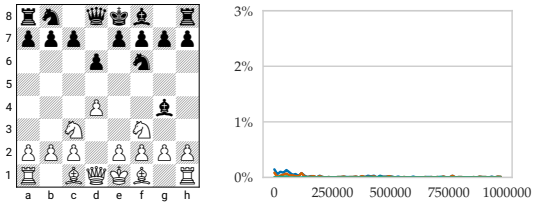
B07 Pirc Defense



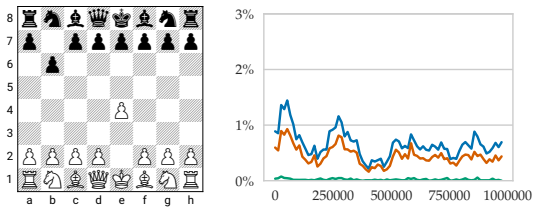
A07 King's Indian Attack



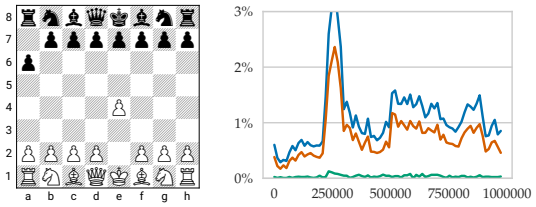
A46 Indian Game



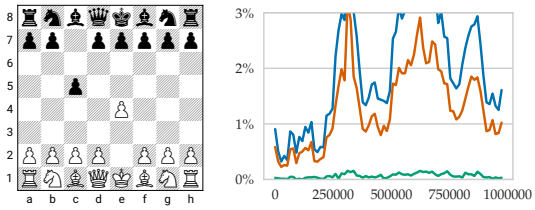
B00 Owen Defense



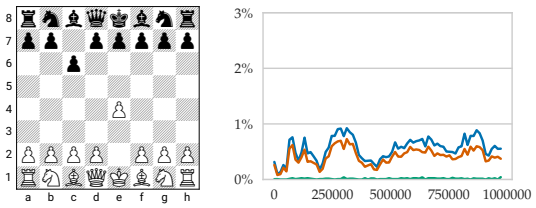
B00 St. George Defense



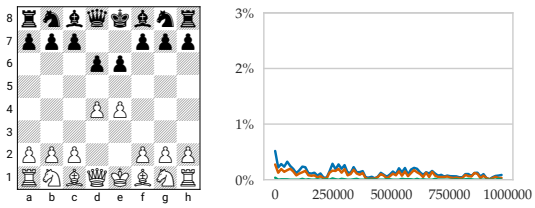
B20 Sicilian Defense



B10 Caro-Kann Defense



C00 Wolf / Rat Defense



B07 Pirc Defense: Bg4

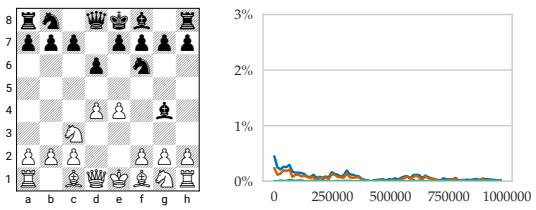
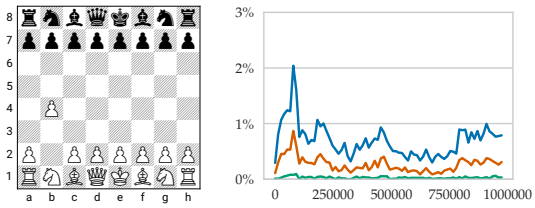


Figure 6.5: Opening statistics 31-40

A00 Polish Opening



A00 Grob Opening

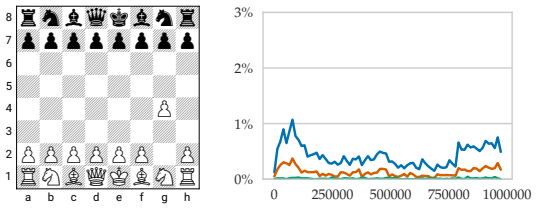


Figure 6.6: Opening statistics 41-42

The favorable evaluation speed for *CrazyAra* is primarily due to the replacement of the *AlphaZero* 19 $\times$ 256 architecture by the RISEv2 13 $\times$ 256 architecture. If *CrazyAra* were using the original *AlphaZero* 19 $\times$ 256 architecture instead, it would have achieved around $3{,}800$ NPS (Figure 2.1).

The manual enhancements of checking moves was disabled in the case of *CrazyAra*. Although it seemed to have helped in finding and defending forced mating sequences quicker, it generally degraded playing performance. Rather than applying Dirichlet noise on the prior root node policy, a temperature value of two was used on every expanded node in order to increase the activation of low candidate moves while still respecting the given policy of the neural network. Dirichlet noise with a weighting factor of $25\,\%$ usually increased the policy activation of blunder moves and made the search results less reproduceable, especially at low count. To avoid adding further noise, the u_{divisor} (9.2.6 U-Value exploration factor [6]) was kept at constant 1.0. *CrazyAra* reused the search tree of the proceeding search and a temperature of zero for move selection. Pondering was disabled for both engines.

Because the draw rate is minimal in crazyhouse and most common crazyhouse opening positions provide the first player an advantage of more than one hundred centipawns, we instead started the games from 50 randomly selected openings out of 81 opening positions which are intended to be more equal. The opening positions have been gathered by crazyhouse expert **FM** `optilink`. Each opening position was played twice with reversed colors in the second case.

Table 6.2 summarizes the match results: the match ended with a positive score in favor of *CrazyAra* and 28 points ahead of *Stockfish*.

Hereinafter eight example games are given in which the same engine won with both the

Table 6.2: Match result of CrazyAra 0.7.0 playing Multi-Variant-Stockfish-Dev in a time control of 15 min + 10 s

Engine Name	Version	Elo Rating	NPS	Wapc	Lapc	+	=	-
CrazyAra	0.7.0, Model-OS-45	?	8K	124 ± 29	83 ± 16	61	3	36
Stockfish	10 (2019-12-03)	$\approx$4,000	6,700K	83 ± 16	124 ± 29	36	3	61

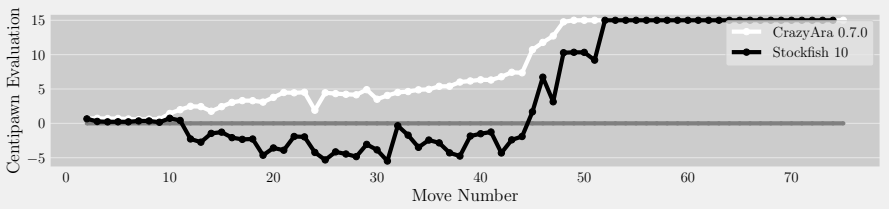
white and black pieces from the identical starting position.[4]

6.3.1 Observations

The games reveal the different nature of the two engines: *CrazyAra* is generally less materialistic and more willing to use long-term strategic sacrifices to gain initiative and activate its pieces (e. g. Figure 6.7a). Furthermore, due to focusing on the most promising lines, *CrazyAra* was able to reach higher depth in its principal variation than *Stockfish*. It seems also more effective in utilizing pawn-pushes such as rook pawn attacks to its advantage. Furthermore, it appears to employ a certain crazyhouse specific maneuver regularly after castling: `Kh1, Rg1` / `Kh8, Rg8` (e. g. Figure 6.7a, 6.10a, 6.11a, 6.12a). Overall, the majority of *CrazyAra's* wins were based on *Stockfish* misevaluating certain middle-game positions.

Moreover, some of *CrazyAra's* lost matches (e. g. Figure 6.13a, 6.14a) revealed the major drawback of the vanilla UCT algorithm which is its inability to share knowledge in between different branches, especially in cases where the policy and value function missed an important line during search. As can be seen in the columns win average plys (**Wapc**) and lost average plys (**Lapc**) of Table 6.2, *CrazyAra* also required more plys than *Stockfish* in converting winning games and was inferior at delaying lost games. The lower efficiency at finding forced mating sequences was also a consequence of not enhancing checking moves.

[4] All 100 games are available at the repository `www.github.com/QueensGambit/CrazyAra`

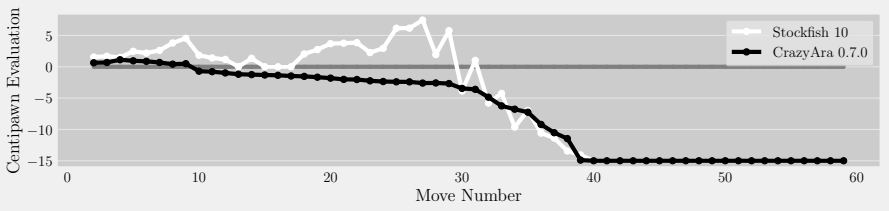


(a) Evaluation progression for both engines

[Event "RL-Eval"]
[Site "Darmstadt, GER"]
[Date "2019.12.18"]
[White "CrazyAra-0.7.0-Model-OS-45"]
[Black "stockfish-x86_64-modern 2019-12-03"]
[Result "1-0"]
[TimeControl "900+10"]
[Variant "crazyhouse"]

1. e4 {book} d5 {book} 2. e5 {book} Bf5 {book} 3. d4 {+0.70/26 26s} e6 {-0.66/24 27s} 4. Nf3 {+0.73/30 26s} Ne7 {-0.28/24 13s} 5. Bg5 {+0.70/52 26s} Nbc6 {-0.22/27 39s} 6. Nc3 {+0.71/41 26s} h6 {-0.23/25 8.1s} 7. Be3 {+0.64/39 26s} Bg4 {-0.22/27 21s} 8. Bd3 {+0.55/34 27s} Nf5 {-0.35/26 23s} 9. Bxf5 {+0.61/63 26s} Bxf5 {-0.35/26 14s} 10. O-O {+0.59/61 26s} Be7 {-0.16/28 29s} 11. N@h5 {+1.44/42 28s} Rg8 {-0.74/29 60s} 12. Ng3 {+1.99/28 27s} Bh7 {-0.42/26 17s} 13. Kh1 {+2.48/29 27s} Qd7 {+2.27/24 14s} 14. Rg1 {+2.44/31 27s} O-O-O {+2.74/23 13s} 15. a4 {+1.76/38 27s} Kb8 {+1.46/28 136s} 16. Nb5 {+2.43/41 27s} Ka8 {+1.28/28 77s} 17. Nxa7 {+3.05/41 27s} Nxa7 {+2.08/25 12s} 18. a5 {+3.31/38 27s} B@a6 {+2.33/26 14s} 19. P@b6 {+3.30/37 27s} Nc8 {+2.28/28 100s} 20. c4 {+3.09/24 27s} dxc4 {+4.64/24 7.2s} 21. d5 {+3.78/35 27s} exd5 {+3.58/26 33s} 22. bxc7 {+4.49/39 27s} Qxc7 {+3.91/27 25s} 23. Bb6 {+4.45/37 27s} Qd7 {+1.91/28 76s} 24. P@d6 {+4.52/38 27s} Rde8 {+1.97/26 146s} 25. dxe7 {+1.91/37 43s} P@e4 {+4.23/23 15s} 26. e6 {+4.45/27 27s} Qxe6 {+5.32/23 5.7s} 27. Nd4 {+4.33/31 27s} Qxe7 {+4.15/26 25s} 28. B@g4 {+4.23/33 27s} P@e6 {+4.46/25 13s} 29. Nxe4 {+4.16/52 28s} Bxe4 {+4.84/25 9.6s} 30. P@b5 {+4.90/31 27s} P@d7 {+3.06/27 55s} 31. bxa6 {+3.51/33 42s} bxa6 {+3.86/26 10s} 32. b4 {+4.06/32 27s} P@c6 {+5.48/25 37s} 33. b5 {+4.50/36 27s} P@b7 {+0.35/26 28s} 34. bxa6 {+4.63/39 28s} bxa6 {+1.72/25 7.3s} 35. Bf3 {+4.88/25 27s} Bxf3 {+3.49/25 50s} 36. Qxf3 {+4.95/32 28s} B@d6 {+2.43/26 48s} 37. P@b5 {+5.39/40 28s} cxb5 {+2.84/20 3.3s} 38. Nxb5 {+5.40/30 28s} axb5 {+4.29/24 21s} 39. P@a6 {+6.01/41 29s} P@b7 {+4.76/23 12s} 40. axb7+ {+6.17/32 28s} Kxb7 {+1.85/26 22s} 41. P@c5 {+6.34/34 18s} N@a6 {+1.51/24 13s} 42. cxd6 {+6.31/32 17s} Qxd6 {+1.26/22 10s} 43. B@a4 {+6.78/35 19s} P@c6 {+4.31/20 4.4s} 44. B@c5 {+7.43/24 17s} Nxc5 {+2.40/22 16s} 45. Bxb5 {+7.36/27 16s} Nxb6 {+1.91/21 10s} 46. axb6 {+10.72/21 16s} N@e4 {-1.68/21 10s} 47. B@c7 {+11.78/26 16s} Qxc7 {-6.72/19 10.0s} 48. bxc7 {+12.72/30 15s} B@b6 {-3.14/19 4.3s} 49. Bxc6+ {+14.77/19 15s} Kxc6 {-10.28/20 16s} 50. N@e5+ {+15.15/17 15s} Kb7 {-10.34/16 9.9s} 51. P@a5 {+15.74/19 15s} B@a7 {-10.33/17 10s} 52. axb6 {+26.03/16 15s} Bxb6 {-9.20/18 6.0s} 53. B@d8 {+30.43/15 14s} Rxd8 {-47.15/18 12s} 54. cxd8=Q {+25.23/14 14s} Rxd8 {-M36/23 12s} 55. R@b5 {+25.22/13 14s} B@a7 {-17.95/17 10s} 56. Rxa7+ {+26.51/9 14s} Kxa7 {-29.24/17 8.7s} 57. P@a5 {+27.27/12 13s} R@a2 {-28.40/16 11s} 58. axb6+ {+29.97/9 13s} Ka8 {-36.43/17 9.7s} 59. B@e3 {+32.38/9 13s} P@d4 {-26.76/16 10s} 60. Rxc5 {+34.24/12 13s} B@b8 {-M22/19 10.0s} 61. b7+ {+33.16/12 13s} Kxb7 {-M14/22 9.7s} 62. Rb5+ {+35.83/11 13s} B@b6 {-M18/26 4.4s} 63. Rxb6+ {+38.19/13 13s} Kxb6 {-M16/32 4.7s} 64. Bxd4+ {+34.77/13 12s} N@c5 {-M26/20 5.2s} 65. Bxc5+ {+35.79/11 12s} Nxc5 {-33.28/17 6.6s} 66. B@d4 {+38.88/12 12s} R@b5 {-M18/29 26s} 67. Bxc5+ {+37.14/11 12s} Kb7 {-M14/33 3.9s} 68. P@e7 {+40.97/11 12s} B@c7 {-M16/27 5.4s} 69. exd8=N+ {+41.12/9 12s} Bxd8 {-M14/33 5.6s} 70. Nxd7 {+45.60/7 12s} Ra6 {-M12/27 5.0s} 71. Nxb8 {+44.56/9 12s} Rxc5 {-M10/37 6.2s} 72. Nxa6 {+56.67/7 12s} Kxa6 {-M8/47 5.7s} 73. N@b4+ {+67.07/5 11s} Kb6 {-M6/245 5.6s} 74. Q@a6+ {+75.38/5 11s} Kc7 {-M4/1 0.001s} 75. R@b7+ {+99.99/3 11s} Kc8 {-M2/1 0.001s} 76. R@b8# {+99.99/1 0.025s, White mates} 1-0

Figure 6.7: Game 7 / 100

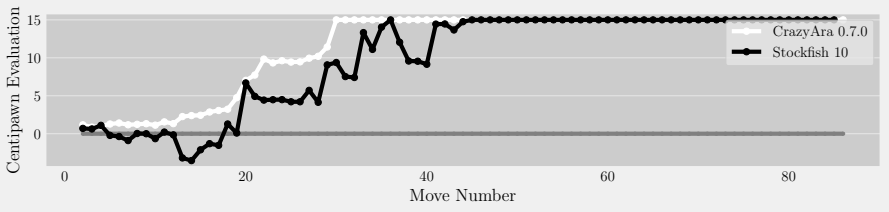


(a) Evaluation progression for both engines

[Event "RL-Eval"]
[Site "Darmstadt, GER"]
[Date "2019.12.18"]
[White "stockfish-x86_64-modern 2019-12-03"]
[Black "CrazyAra-0.7.0-Model-OS-45"]
[Result "0-1"]
[TimeControl "900+10"]
[Variant "crazyhouse"]

1. e4 {book} d5 {book} 2. e5 {book} Bf5 {book} 3. d4 {+1.56/26 52s} e6 {-0.64/42 26s} 4. Nc3 {+1.68/28 93s} Ne7 {-0.69/40 26s} 5. Bb5+ {+1.49/25 15s} Nbc6 {-1.12/37 26s} 6. Nge2 {+2.47/25 9.8s} a6 {-0.95/36 26s} 7. Ba4 {+2.19/27 28s} h5 {-0.87/34 26s} 8. Bg5 {+2.62/25 27s} h4 {-0.69/39 27s} 9. h3 {+3.80/26 38s} Rh5 {-0.42/45 27s} 10. Be3 {+4.55/24 9.2s} Ng6 {-0.49/41 26s} 11. O-O {+1.84/28 81s} Rg5 {+0.70/39 27s} 12. Bxc6+ {+1.41/30 110s} bxc6 {+0.78/56 26s} 13. Bxg5 {+1.18/29 18s} Qxg5 {+0.98/65 26s} 14. N@f3 {+0.02/30 28s} Qd8 {+1.19/63 26s} 15. R@g5 {+1.35/31 152s} B@e4 {+1.25/67 27s} 16. Rxf5 {0.00/30 20s} exf5 {+1.31/75 27s} 17. B@h5 {0.00/30 12s} Bxf3 {+1.36/73 27s} 18. Bxf3 {0.00/31 16s} N@g5 {+1.47/71 27s} 19. B@h5 {+2.05/29 12s} B@e4 {+1.53/69 27s} 20. Bxe4 {+2.74/27 50s} dxe4 {+1.66/61 28s} 21. Nxe4 {+3.69/26 11s} fxe4 {+1.81/59 27s} 22. B@g4 {+3.77/26 12s} B@d7 {+2.01/57 28s} 23. e6 {+3.85/26 22s} Bxe6 {+2.03/54 27s} 24. Bxe6 {+2.27/27 41s} Bd6 {+2.24/54 27s} 25. P@e5 {+2.95/25 9.6s} N@f3+ {+2.35/52 28s} 26. gxf3 {+6.15/22 12s} Nxf3+ {+2.40/50 27s} 27. Bxf3 {+6.17/22 15s} exf3 {+2.41/48 28s} 28. B@e4 {+7.46/22 17s} Qg5+ {+2.60/46 29s} 29. N@g3 {+1.95/27 100s} R@g2+ {+2.58/45 31s} 30. Kh1 {+5.76/1 0s} hxg3 {+2.68/56 28s} 31. Bxc6+ {-3.87/23 84s} P@d7 {+3.45/53 29s} 32. Nxg3 {+1.02/20 8.8s} Rxg3 {+3.60/51 28s} 33. Bexd7+ {-5.79/21 31s} Kd8 {+4.85/65 28s} 34. P@e7+ {-4.25/21 29s} Kxe7 {+6.24/33 30s} 35. P@g2 {-9.63/23 41s} fxg2+ {+6.77/44 29s} 36. Bxg2 {-6.97/21 5.6s} P@f3 {+7.25/27 29s} 37. Qxf3 {-10.58/22 26s} Rxf3 {+9.20/25 30s} 38. exd6+ {-11.38/21 14s} cxd6 {+10.52/23 29s} 39. P@g3 {-13.44/20 9.9s} B@e4 {+11.46/17 30s} 40. N@c8+ {-14.07/18 10s} Rxc8 {+14.89/17 31s} 41. B@c6 {-23.42/19 10s} Rxc6 {+19.15/17 19s} 42. Bxc6 {-25.48/18 10.0s} Bxc6 {+19.02/15 18s} 43. R@c7+ {-25.85/17 10s} B@d7 {+20.29/11 18s} 44. P@b7 {-27.67/17 9.8s} B@b8 {+21.48/11 18s} 45. Rxd7+ {-31.34/17 10s} Kxd7 {+22.17/11 17s} 46. d5 {-32.76/17 10.0s} Bxd5 {+24.53/11 17s} 47. P@e6+ {-37.16/17 10s} fxe6 {+29.45/9 16s} 48. Rae1 {-51.24/17 10.0s} Rxg3 {+36.86/5 16s} 49. B@e8+ {-M12/40 7.6s} Kxe8 {+32.54/13 16s} 50. Rxe6+ {-M14/29 4.0s} Bxe6 {+43.67/11 15s} 51. P@f7+ {-M12/40 5.9s} Kxf7 {+42.94/9 15s} 52. fxg3+ {-M16/27 4.8s} P@f4 {+37.89/7 15s} 53. Rxf4+ {-M12/44 5.6s} Qxf4 {+39.78/5 15s} 54. R@f8+ {-M12/43 5.0s} Nxf8 {+51.29/7 14s} 55. P@g6+ {-M10/51 5.8s} Nxg6 {+50.29/5 14s} 56. gxf4 {-M8/62 5.9s} N@g3+ {+44.89/7 14s} 57. Kg1 {-M6/245 2.5s} Q@e1+ {+66.74/7 14s} 58. Q@f1 {-M6/245 1.6s} N@e2+ {+64.09/5 13s} 59. Kh2 {-M4/1 0s} Nxf1+ {+99.99/3 13s} 60. Bxf1 {-M2/245 0.013s} Q@g1# {+99.99/1 0.021s, Black mates} 0-1

Figure 6.8: Game 8 / 100

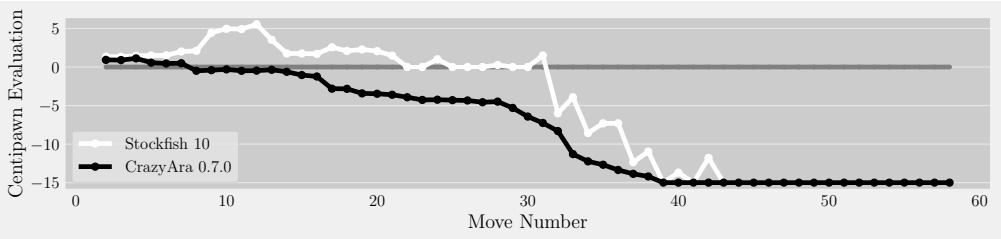


(a) Evaluation progression for both engines

[Event "RL-Eval"]
[Site "Darmstadt, GER"]
[Date "2019.12.18"]
[White "CrazyAra-0.7.0-Model-OS-45"]
[Black "stockfish-x86_64-modern 2019-12-03"]
[Result "1-0"]
[TimeControl "900+10"]
[Variant "crazyhouse"]

1. d4 {book} Nf6 {book} 2. Bf4 {book} e6 {book} 3. e3 {+1.17/31 26s} Be7 {-0.70/28 68s} 4. Nf3 {+0.87/28 26s} Nc6 {-0.63/27 42s} 5. Nc3 {+1.03/33 26s} O-O {-1.10/28 66s} 6. Bc4 {+1.28/38 26s} Nh5 {+0.22/29 25s} 7. d5 {+1.41/55 27s} exd5 {+0.37/24 7.3s} 8. Nxd5 {+1.18/49 26s} d6 {+0.90/25 12s} 9. P@h6 {+1.24/61 26s} Nxf4 {-0.03/28 35s} 10. exf4 {+1.32/66 26s} gxh6 {0.00/27 15s} 11. O-O {+1.13/64 26s} B@f6 {+0.65/28 79s} 12. Nxe7+ {+1.56/48 29s} Bxe7 {-0.22/29 42s} 13. B@g5 {+1.33/35 26s} P@h3 {+0.15/28 28s} 14. Bxh6 {+2.26/35 27s} hxg2 {+3.19/23 12s} 15. P@g7 {+2.39/53 27s} gxf1=Q+ {+3.54/26 42s} 16. Qxf1 {+2.43/45 27s} R@g4+ {+2.12/29 165s} 17. P@g2 {+2.86/48 27s} Rxg7 {+1.31/28 28s} 18. Bxg7 {+3.06/50 27s} Kxg7 {+1.53/23 8.8s} 19. Re1 {+3.23/57 27s} B@g4 {-1.28/28 140s} 20. N@d5 {+4.75/27 29s} Bf6 {-0.09/27 30s} 21. Nxf6 {+7.03/32 27s} Qxf6 {-6.69/24 19s} 22. B@h4 {+7.73/32 27s} Qxb2 {-4.92/26 60s} 23. R@g5+ {+9.83/34 28s} N@g6 {-4.43/24 7.3s} 24. Rxg4 {+9.33/38 28s} Bxg4 {-4.48/26 12s} 25. B@f6+ {+9.62/37 27s} Qxf6 {-4.49/28 38s} 26. Bxf6+ {+9.44/52 27s} Kxf6 {-4.21/25 15s} 27. Q@g5+ {+9.47/50 28s} Kg7 {-4.21/1 0s} 28. Qxg4 {+9.95/48 28s} B@c3 {-5.71/25 15s} 29. f5 {+10.20/33 28s} N@h6 {-4.14/26 24s} 30. Qh5 {+11.43/31 28s} Nf4 {-9.07/24 81s} 31. Qg5+ {+16.44/26 31s} Kh8 {-9.38/23 16s} 32. f6 {+16.70/26 28s} Bxf6 {-7.53/23 8.2s} 33. Qxf6+ {+16.67/23 28s} P@g7 {-7.41/24 14s} 34. Qxh6 {+17.44/23 28s} gxh6 {-13.32/24 42s} 35. B@f6+ {+17.36/22 29s} Kg8 {-11.11/22 3.8s} 36. B@g7 {+17.85/21 29s} B@h8 {-14.04/23 27s} 37. Bxf8 {+17.99/23 30s} Bxf6 {-15.57/19 15s} 38. Bxh6 {+18.47/23 30s} P@e2 {-12.05/18 6.7s} 39. Rxe2 {+17.98/21 30s} Nxe2+ {-9.58/18 3.8s} 40. Bxe2 {+18.10/16 31s} B@d4 {-9.55/21 19s} 41. P@g7 {+18.66/20 20s} Bxg7 {-9.15/21 3.1s} 42. Bxg7 {+19.21/20 18s} Bxg7 {-14.44/22 15s} 43. P@f6 {+19.55/13 18s} B@f8 {-14.46/21 12s} 44. fxg7 {+20.03/21 18s} Bxg7 {-13.67/22 3.5s} 45. B@f6 {+21.74/24 17s} Q@f8 {-14.76/21 16s} 46. Bxg7 {+28.17/11 17s} Qxg7 {-15.46/22 10s} 47. N@f5 {+25.60/12 16s} B@f8 {-16.03/23 10s} 48. Nxg7 {+35.52/11 16s} Bxg7 {-15.47/19 4.8s} 49. B@d5 {+29.03/10 16s} R@f6 {-16.66/20 15s} 50. Bxf7+ {+27.59/12 15s} Kh8 {-18.17/20 10s} 51. B@e8 {+31.98/11 15s} R@f8 {-17.51/19 10.0s} 52. P@g6 {+31.98/9 15s} Raxe8 {-15.93/19 10s} 53. Bxe8 {+28.79/9 15s} B@d5 {-15.49/19 10s} 54. gxh7 {+28.39/13 14s} Rxe8 {-21.28/18 10s} 55. P@d7 {+29.84/11 14s} Ref8 {-16.82/17 3.4s} 56. R@e8 {+28.28/11 14s} B@g6 {-22.43/18 16s} 57. R@g8+ {+26.81/14 14s} Bxg8 {-23.81/19 10s} 58. hxg8=Q+ {+26.99/15 13s} Rxg8 {-24.75/19 10s} 59. Rxg8+ {+29.53/11 13s} Kxg8 {-31.07/17 10.0s} 60. B@d5+ {+31.83/9 13s} P@e6 {-29.19/18 10.0s} 61. R@e8+ {+30.32/14 13s} R@f8 {-27.08/18 8.8s} 62. Bxe6+ {+31.39/11 13s} Rxe6 {-24.69/18 11s} 63. Rxe6 {+30.58/11 13s} B@h7 {-18.63/16 10s} 64. P@e7 {+34.63/11 13s} Nxe7 {-27.00/18 10s} 65. Rxe7 {+38.41/12 12s} N@f7 {-29.17/16 10.0s} 66. R@e8 {+32.82/7 12s} R@a8 {-32.31/17 9.9s} 67. Rxa8 {+30.24/18 12s} Rxa8 {-32.10/18 7.3s} 68. R@c8+ {+28.63/17 12s} R@f8 {-46.22/18 12s} 69. Rxa8 {+28.14/10 12s} Rxa8 {-58.31/18 10s} 70. R@c8+ {+25.95/9 12s} R@f8 {-M22/20 9.0s} 71. Q@e8 {+36.19/9 12s} Kh8 {-62.88/15 11s} 72. Rxa8 {+46.02/19 12s} Bg8 {-M16/25 7.6s} 73. Qxf8 {+53.41/8 12s} Bxf8 {-M14/34 4.8s} 74. Rxf8 {+45.17/7 11s} Q@g7 {-M12/38 9.7s} 75. Rexf7 {+47.65/7 11s} Bxf7 {-45.33/14 3.0s} 76. d8=Q {+42.70/7 11s} R@g6 {-M32/17 23s} 77. B@f6 {+46.34/8 11s} Rxf6 {-M18/26 4.3s} 78. Qxf6 {+46.81/9 11s} Qxf6 {-M20/16 4.7s} 79. N@g4 {+43.92/10 11s} Qf5 {-M20/18 14s} 80. R@h6+ {+39.74/13 11s} Kg7 {-M10/42 4.4s} 81. Rxg8+ {+47.08/7 11s} Bxg8 {-M10/47 5.3s} 82. B@f6+ {+49.18/5 11s} Kf8 {-M8/77 5.2s} 83. R@d8+ {+44.04/3 11s} B@e8 {-M6/245 0.83s} 84. Rxe8+ {+52.24/7 11s} Kxe8 {-M6/245 1.5s} 85. R@d8+ {+62.49/5 11s} Kf7 {-M4/1 0s} 86. N@g5+ {+99.99/3 11s} Qxg5 {-M2/1 0.001s} 87. Nxg5# {+99.99/1 0.017s, White mates} 1-0

Figure 6.9: Game 23 / 100

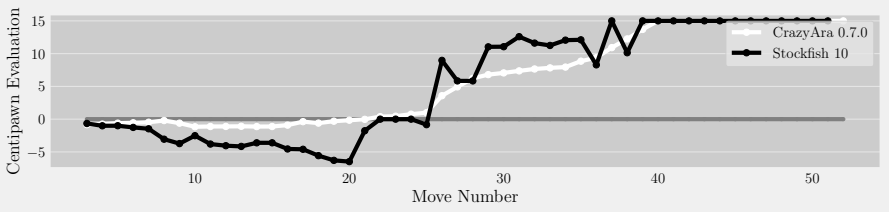


(a) Evaluation progression for both engines

[Event "RL-Eval"]
[Site "Darmstadt, GER"]
[Date "2019.12.18"]
[White "stockfish-x86_64-modern 2019-12-03"]
[Black "CrazyAra-0.7.0-Model-OS-45"]
[Result "0-1"]
[TimeControl "900+10"]
[Variant "crazyhouse"]

1. d4 {book} Nf6 {book} 2. Bf4 {book} e6 {book} 3. e3 {+1.34/28 65s} d6 {-0.93/24 26s} 4. Nf3 {+1.33/29 83s} Be7 {-0.90/32 26s} 5. Nc3 {+1.46/28 26s} Nc6 {-1.12/38 26s} 6. h3 {+1.50/25 14s} O-O {-0.57/36 27s} 7. Be2 {+1.51/25 11s} Kh8 {-0.46/43 26s} 8. O-O {+2.01/23 11s} Rg8 {-0.49/41 26s} 9. e4 {+2.10/24 11s} e5 {+0.49/60 27s} 10. dxe5 {+4.45/25 12s} dxe5 {+0.41/58 26s} 11. Qxd8 {+4.96/29 42s} Bxd8 {+0.31/52 26s} 12. Nxe5 {+4.92/27 10s} Nxe5 {+0.49/55 26s} 13. Bxe5 {+5.54/26 21s} Bxh3 {+0.47/58 26s} 14. N@g5 {+3.53/28 44s} N@h6 {+0.36/56 26s} 15. Nxh3 {+1.76/27 32s} P@g4 {+0.62/57 27s} 16. Q@g3 {+1.75/28 40s} gxh3 {+1.04/54 29s} 17. Qxh3 {+1.70/27 11s} P@g4 {+1.23/67 27s} 18. Qh2 {+2.56/26 73s} Q@h5 {+2.81/41 27s} 19. Qxh5 {+2.09/27 80s} Nxh5 {+2.83/43 27s} 20. P@g3 {+2.28/29 119s} Bf6 {+3.42/40 27s} 21. Bxf6 {+2.06/29 17s} Nxf6 {+3.47/55 27s} 22. Q@h1 {+1.49/30 77s} B@h3 {+3.60/43 28s} 23. gxh3 {0.00/32 26s} gxh3 {+3.92/45 27s} 24. Qxh3 {0.00/34 14s} P@g4 {+4.27/43 27s} 25. Qh2 {+1.00/27 64s} N@f3+ {+4.24/37 27s} 26. Bxf3 {0.00/29 17s} gxf3 {+4.30/35 27s} 27. B@h3 {0.00/29 27s} B@g4 {+4.34/45 28s} 28. B@f5 {0.00/31 27s} Bxh3 {+4.57/51 29s} 29. Bxh3 {+0.26/31 12s} B@g4 {+4.49/44 28s} 30. B@f5 {0.00/35 50s} Q@h5 {+5.31/41 30s} 31. B@h4 {0.00/27 22s} Bxh3 {+6.43/27 29s} 32. Bxf6 {+1.50/22 14s} gxf6 {+7.26/23 28s} 33. Bxh3 {-6.00/24 77s} B@e5 {+8.31/25 29s} 34. B@d4 {-3.93/21 6.9s} Bxd4 {+11.30/32 29s} 35. N@f4 {-8.56/23 19s} Bxc3 {+12.24/34 29s} 36. Nxh5 {-7.31/22 21s} N@e2+ {+12.68/32 29s} 37. Kh1 {-7.31/1 0s} B@g4 {+13.36/26 29s} 38. N@e3 {-12.35/21 40s} Bxh3 {+13.86/29 30s} 39. Qxh3 {-10.97/20 4.8s} B@g4 {+14.20/32 29s} 40. B@f8 {-15.77/20 25s} Raxf8 {+16.08/33 32s} 41. P@e7 {-13.70/20 8.1s} B@g7 {+16.65/27 19s} 42. Nxg4 {-17.58/21 15s} B@g2+ {+16.77/25 18s} 43. Qxg2 {-11.79/19 2.9s} fxg2+ {+16.96/23 17s} 44. Kxg2 {-22.83/21 17s} Nxg4 {+17.66/21 17s} 45. Q@h3 {-24.47/19 11s} Q@h2+ {+18.04/27 17s} 46. Qxh2 {-25.74/20 10s} Nxh2 {+18.27/28 16s} 47. Q@h4 {-27.08/19 10s} N@f3 {+19.85/21 17s} 48. Qxh2 {-22.63/16 10s} Nxh2 {+20.59/24 16s} 49. f3 {-M40/17 10.0s} Nxf1 {+42.78/11 16s} 50. Rxf1 {-M12/41 4.6s} Nxg3 {+34.52/9 15s} 51. N@g6+ {-M10/69 4.7s} hxg6 {+42.24/7 15s} 52. N@f4 {-M8/78 4.8s} Nxf1 {+41.63/7 15s} 53. B@g3 {-M8/55 4.9s} N@e3+ {+40.56/9 14s} 54. Kg1 {-M8/79 4.9s} P@h2+ {+37.59/7 14s} 55. Bxh2 {-M6/245 0.47s} Nxh2 {+47.11/5 14s} 56. Nxg6+ {-M8/245 1.5s} fxg6 {+81.41/3 14s} 57. B@f1 {-M6/245 0.11s} Nxf3+ {+73.58/5 14s} 58. Kf2 {-M4/245 0.060s} Q@e1+ {+99.99/3 13s} 59. Kxf3 {-M2/1 0s} Q@g4# {+99.99/1 0.014s, Black mates} 0-1

Figure 6.10: Game 24 / 100

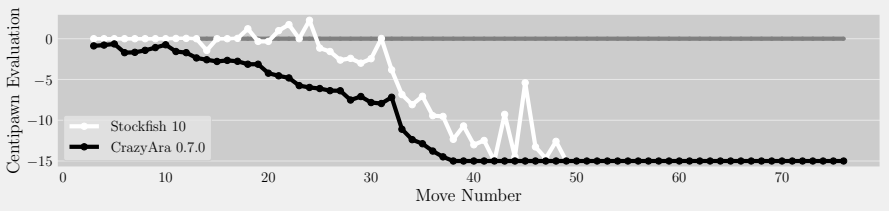


(a) Evaluation progression for both engines

[Event "RL-Eval"]
[Site "Darmstadt, GER"]
[Date "2019.12.18"]
[White "CrazyAra-0.7.0-Model-OS-45"]
[Black "stockfish-x86_64-modern 2019-12-03"]
[Result "1-0"]
[TimeControl "900+10"]
[Variant "crazyhouse"]

1. e4 {book} e5 {book} 2. Bc4 {book} Nc6 {book} 3. c3 {book} Nf6 {book} 4. d3 {-0.84/32 26s} Be7 {+0.66/27 30s} 5. Nf3 {-0.75/36 26s} d6 {+1.01/28 61s} 6. O-O {-0.61/32 27s} O-O {+1.00/24 12s} 7. Nbd2 {-0.55/32 26s} Na5 {+1.26/24 15s} 8. Kh1 {-0.47/33 27s} Bg4 {+1.47/27 56s} 9. Rg1 {-0.21/40 28s} Qd7 {+3.06/26 40s} 10. Qf1 {-0.61/40 27s} Nh5 {+3.73/25 10s} 11. h3 {-1.16/42 27s} Bxf3 {+2.52/31 69s} 12. gxf3 {-1.12/56 27s} Bh4 {+3.80/29 38s} 13. B@f5 {-1.11/65 27s} Nxc4 {+4.05/24 7.5s} 14. Nxc4 {-1.09/33 28s} Qd8 {+4.16/28 44s} 15. Be3 {-1.13/53 27s} B@f4 {+3.61/28 32s} 16. N@e2 {-1.12/47 27s} Bxe3 {+3.61/25 13s} 17. Nxe3 {-0.91/43 28s} Bxf2 {+4.55/25 18s} 18. Qxf2 {-0.38/48 28s} B@h4 {+4.60/25 18s} 19. Bxh7+ {-0.60/51 27s} Kxh7 {+5.57/25 13s} 20. P@g3 {-0.31/48 28s} Nxg3+ {+6.29/25 23s} 21. Rxg3 {-0.16/43 27s} B@g5 {+6.48/25 29s} 22. B@c4 {-0.03/42 28s} Bxg3 {+1.77/25 28s} 23. Qxg3 {+0.39/60 28s} P@f4 {0.00/27 42s} 24. Qg2 {+0.45/34 28s} N@f2+ {0.00/29 25s} 25. Qxf2 {+0.76/39 28s} fxe3 {0.00/30 13s} 26. B@f5+ {+0.99/45 28s} Kh8 {+0.84/28 28s} 27. Bxf7 {+3.58/53 29s} P@h7 {-8.97/26 43s} 28. Bxh7 {+4.93/42 28s} Rxf7 {-5.84/25 19s} 29. N@g6+ {+6.23/40 28s} Kxh7 {-5.84/1 0s} 30. B@f5 {+6.80/38 28s} R@h6 {-11.04/28 116s} 31. Nxe5+ {+7.06/50 29s} g6 {-11.06/26 54s} 32. Bxg6+ {+7.37/42 29s} Rxg6 {-12.59/25 24s} 33. P@f5 {+7.66/40 29s} dxe5 {-11.61/23 13s} 34. fxg6+ {+7.83/31 29s} Kxg6 {-11.25/21 11s} 35. P@f5+ {+7.95/29 29s} Kh6 {-12.06/25 139s} 36. N@g4+ {+8.85/27 32s} Kg7 {-12.11/21 28s} 37. Qg2 {+9.38/29 29s} B@h5 {-8.29/20 33s} 38. P@g6 {+10.92/25 31s} B@h7 {-15.23/23 33s} 39. gxf7 {+12.25/21 30s} Bxg4 {-10.14/16 6.5s} 40. R@e8 {+13.70/19 30s} N@f8 {-23.93/20 50s} 41. Rxd8 {+15.66/17 19s} N@f2+ {-39.92/14 27s} 42. Qxf2 {+26.65/7 18s} Bxf3+ {-M16/30 8.3s} 43. Qxf3 {+43.34/11 18s} P@g2+ {-M14/36 4.8s} 44. Qxg2 {+47.75/9 17s} N@f2+ {-M12/59 4.8s} 45. Qxf2 {+43.99/5 17s} P@g2+ {-M10/59 5.3s} 46. Qxg2 {+53.53/7 17s} B@f6 {-M8/68 5.3s} 47. Rxa8 {+47.96/7 16s} Kh6 {-M8/53 5.2s} 48. N@g4+ {+52.90/5 16s} Kg7 {-M6/245 1.8s} 49. R@g8+ {+58.33/5 16s} Bxg8 {-M4/245 0.053s} 50. fxg8=Q+ {+55.35/7 15s} Kxg8 {-54.39/1 0.001s} 51. Nxf6+ {+61.78/5 15s} Kf7 {-M4/245 0.064s} 52. Q@g8+ {+99.99/3 15s} Kxf6 {-M2/245 0.015s} 53. Q2xg5# {+99.99/1 0.063s, White mates} 1-0

Figure 6.11: Game 31 / 100

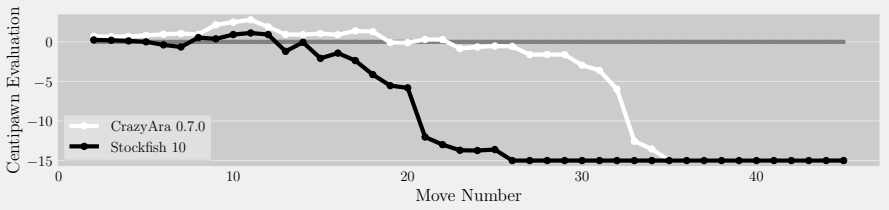


(a) Evaluation progression for both engines

[Event "RL-Eval"]
[Site "Darmstadt, GER"]
[Date "2019.12.18"]
[White "stockfish-x86_64-modern 2019-12-03"]
[Black "CrazyAra-0.7.0-Model-OS-45"]
[Result "0-1"]
[TimeControl "900+10"]
[Variant "crazyhouse"]

1. e4 {book} e5 {book} 2. Bc4 {book} Nc6 {book} 3. c3 {book} Nf6 {book} 4. d3 {0.00/29 24s} Be7 {+0.88/31 26s} 5. Nf3 {0.00/31 19s} d6 {+0.78/35 26s} 6. O-O {0.00/33 32s} O-O {+0.64/31 27s} 7. Bb3 {0.00/35 11s} Na5 {+1.71/29 27s} 8. Bc2 {0.00/34 12s} Nc6 {+1.67/31 27s} 9. Bb3 {0.00/33 15s} Na5 {+1.43/47 27s} 10. Bc2 {0.00/36 19s} Qd7 {+1.09/45 27s} 11. Nbd2 {0.00/31 143s} Qg4 {+0.74/26 27s} 12. Re1 {+0.03/29 9.2s} Nc6 {+1.57/36 28s} 13. Nf1 {+0.04/29 13s} Kh8 {+1.71/41 27s} 14. Ne3 {0.00/30 51s} Qh5 {+2.36/35 27s} 15. Qe2 {-1.44/24 26s} Rg8 {+2.58/42 27s} 16. Bd1 {0.00/28 24s} g5 {+2.79/41 27s} 17. Nd2 {0.00/25 12s} Qxe2 {+2.66/54 27s} 18. Bxe2 {+0.04/25 11s} Be6 {+2.77/43 27s} 19. Ndf1 {+1.22/24 23s} Rg6 {+3.13/35 29s} 20. Ng3 {-0.33/29 187s} Rag8 {+3.13/43 28s} 21. Q@b5 {-0.33/25 30s} Rh6 {+4.23/47 28s} 22. Qxb7 {+0.99/23 15s} Nd4 {+4.55/40 27s} 23. cxd4 {+1.73/25 57s} exd4 {+4.81/44 28s} 24. Qa8 {0.00/29 67s} Q@g7 {+5.75/54 28s} 25. Qxg8+ {+2.26/25 18s} Nxg8 {+5.98/52 28s} 26. N@f5 {-1.15/28 138s} dxe3 {+6.11/54 29s} 27. Bxe3 {-1.57/26 17s} P@f4 {+6.37/47 28s} 28. Nxg7 {-2.62/25 42s} fxe3 {+6.38/50 28s} 29. Nxe6 {-2.41/23 7.5s} exf2+ {+7.52/25 30s} 30. Kh1 {-3.00/25 22s} B@e5 {+7.09/37 29s} 31. Q@f3 {-2.45/24 44s} Rxh2+ {+7.83/28 29s} 32. Kxh2 {+0.03/1 0s} fxe1=Q {+7.95/33 29s} 33. Rxe1 {-3.82/22 18s} P@f4 {+7.18/33 29s} 34. R@h3 {-6.85/25 60s} fxg3+ {+11.10/27 30s} 35. Qxg3 {-8.09/23 34s} Bxg3+ {+12.38/23 29s} 36. Rxg3 {-7.06/21 4.1s} Q@h6+ {+12.88/29 29s} 37. R@h5 {-9.44/21 22s} P@h4 {+13.78/19 30s} 38. B@f2 {-9.53/21 12s} hxg3+ {+14.49/28 31s} 39. Bxg3 {-12.34/20 10s} R@h4+ {+15.54/26 30s} 40. P@h3 {-10.70/19 7.1s} Rxh5 {+15.78/22 30s} 41. B@c3+ {-13.01/19 13s} Bf6 {+16.50/23 20s} 42. Bxh5 {-12.48/21 9.9s} Bxc3 {+17.36/21 18s} 43. bxc3 {-16.07/21 10s} B@f4 {+16.89/21 18s} 44. B@d4+ {-9.28/19 4.9s} R@g7 {+17.29/19 18s} 45. P@f2 {-14.48/20 15s} Bxg3+ {+18.25/17 17s} 46. fxg3 {-5.40/19 4.0s} B@e5 {+19.21/20 17s} 47. Bxe5 {-13.27/22 16s} dxe5 {+19.04/16 17s} 48. P@f2 {-14.67/23 10s} Q@d2 {+20.61/20 17s} 49. R@f1 {-12.59/20 4.2s} Qxf2 {+22.74/17 16s} 50. Rxf2 {-20.44/20 16s} P@h4 {+22.95/15 15s} 51. B@g1 {-24.38/19 10s} hxg3+ {+22.95/15 16s} 52. Kh1 {-25.99/20 10s} P@h2 {+25.88/13 15s} 53. Bxh2 {-26.43/20 10.0s} gxf2 {+25.10/14 15s} 54. Q@e3 {-29.60/19 10s} fxe1=Q+ {+24.31/13 15s} 55. Qxe1 {-32.37/20 9.7s} Qxe6 {+22.90/11 14s} 56. B@f8 {-20.27/20 4.7s} B@h6 {+21.71/9 14s} 57. P@f2 {-20.30/22 16s} N@c2 {+24.32/21 14s} 58. Bxg7+ {-28.94/18 10.0s} Bxg7 {+25.24/21 14s} 59. Qe2 {-31.08/19 10s} R@e1+ {+26.76/12 13s} 60. R@g1 {-33.11/19 10.0s} Rxe2 {+27.03/14 13s} 61. Bxe2 {-26.22/20 8.1s} R@e1 {+23.03/15 13s} 62. R@f1 {-26.07/20 12s} Rxe2 {+31.81/10 13s} 63. P@d5 {-28.77/18 10.0s} Qf6 {+31.88/9 13s} 64. P@f5 {-30.70/20 10s} B@f4 {+35.08/7 13s} 65. P@g3 {-32.53/19 10.0s} N@d2 {+35.34/9 12s} 66. gxf4 {-31.29/18 3.3s} exf4 {+32.90/8 12s} 67. B@d4 {-35.38/19 17s} Nxd4 {+37.32/9 12s} 68. cxd4 {-43.40/17 9.9s} Nxf1 {+34.16/9 12s} 69. Rxf1 {-M16/29 7.5s} P@f3 {+49.65/7 12s} 70. N@e1 {-M14/36 5.7s} fxg2+ {+48.23/3 12s} 71. Nxg2 {-M14/37 4.8s} P@f3 {+49.80/5 12s} 72. N@h4 {-M12/43 4.9s} fxg2+ {+49.22/11 12s} 73. Nxg2 {-M10/53 5.6s} B@f3 {+54.50/9 11s} 74. Bxf4 {-M8/60 5.7s} Bxg2+ {+54.83/5 11s} 75. Kh2 {-M6/245 1.6s} N@f3+ {+61.18/5 11s} 76. Kg3 {-M4/245 0.052s} Q@h2+ {+99.99/3 11s} 77. Kg4 {-M2/1 0s} R@h4# {+99.99/1 0.021s, Black mates} 0-1

Figure 6.12: Game 32 / 100

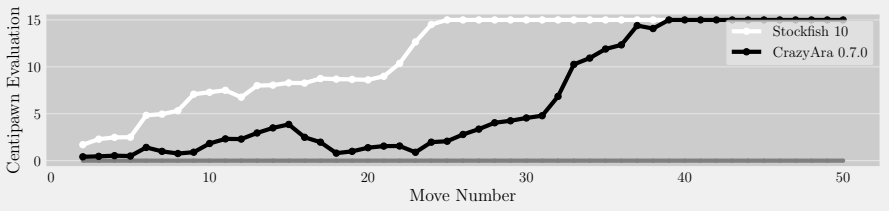


(a) Evaluation progression for both engines

[Event "RL-Eval"]
[Site "Darmstadt, GER"]
[Date "2019.12.19"]
[White "CrazyAra-0.7.0-Model-OS-45"]
[Black "stockfish-x86_64-modern 2019-12-03"]
[Result "0-1"]
[PlyCount "92"]
[TimeControl "900+10"]
[Variant "crazyhouse"]

1. e4 {book} c5 {book} 2. Nc3 {book} Nc6 {book} 3. f4 {book} e6 {-0.24/29 141s} 4. Nf3 {+0.69/28 26s} Be7 {-0.20/26 15s} 5. Be2 {+0.67/40 26s} Nh6 {-0.12/26 28s} 6. O-O {+0.68/38 26s} d5 {0.00/28 16s} 7. d3 {+0.82/38 27s} O-O {+0.38/29 34s} 8. Kh1 {+0.96/36 27s} d4 {+0.65/25 17s} 9. Nb5 {+1.05/45 27s} a6 {-0.54/29 170s} 10. Na3 {+0.95/48 27s} f6 {-0.38/26 27s} 11. Qe1 {+2.15/30 27s} Ra7 {-0.92/30 125s} 12. Qg3 {+2.49/25 27s} b5 {-1.12/27 36s} 13. Bd2 {+2.78/30 27s} Nf7 {-0.92/27 46s} 14. c4 {+1.90/36 41s} dxc3 {+1.20/23 5.3s} 15. Bxc3 {+0.94/44 40s} b4 {+0.06/27 87s} 16. P@h5 {+0.91/50 26s} Nh6 {+2.08/25 23s} 17. Nc4 {+1.05/44 28s} bxc3 {+1.42/25 15s} 18. bxc3 {+0.89/41 27s} P@g4 {+2.37/24 7.1s} 19. P@g6 {+1.39/31 27s} gxf3 {+4.14/23 9.0s} 20. Rxf3 {+1.29/36 26s} hxg6 {+5.53/25 24s} 21. hxg6 {-0.07/70 41s} P@g4 {+5.81/27 27s} 22. P@g5 {-0.10/69 26s} fxg5 {+12.02/24 15s} 23. fxg5 {+0.30/66 27s} gxf3 {+12.99/24 15s} 24. gxh6 {+0.28/78 26s} fxe2 {+13.70/24 13s} 25. P@h7+ {-0.84/62 41s} Kh8 {+13.73/1 0s} 26. hxg7+ {-0.65/59 26s} Kxg7 {+13.61/1 0s} 27. P@h6+ {-0.53/59 26s} Kxh6 {+18.46/21 7.1s} 28. N@g8+ {-0.58/57 26s} Kg7 {+19.15/24 23s} 29. P@h6+ {-1.62/59 41s} Kh8 {+19.15/1 0s} 30. Nxe7 {-1.60/57 26s} Qxe7 {+19.88/23 15s} 31. g7+ {-1.61/27 27s} Kxh7 {+21.13/23 13s} 32. B@g6+ {-2.95/30 39s} Kg8 {+21.87/23 11s} 33. gxf8=N {-3.61/50 26s} R@f1+ {+29.02/23 34s} 34. R@g1 {-5.98/29 39s} Rxf8 {+29.15/21 7.1s} 35. Bf7+ {-12.55/20 39s} Kxf7 {+32.22/23 35s} 36. Rae1 {-13.54/23 25s} B@c7 {+33.76/24 21s} 37. Qg7+ {-16.64/17 25s} Ke8 {+34.61/1 0s} 38. Rxe2 {-17.61/26 25s} P@f2 {+46.67/20 9.0s} 39. Rxf2 {-17.87/14 25s} Qxg7 {+M29/22 12s} 40. hxg7 {-17.35/14 26s} N@g3+ {+M15/51 9.1s} 41. hxg3 {-18.45/12 0.25s} Q@h2+ {+M13/62 10s} 42. Kxh2 {-19.29/10 0.001s} N@g4+ {+M11/69 9.2s} 43. Kh3 {-22.67/8 17s} Nxf2+ {+M9/91 9.7s} 44. Kh4 {-24.29/6 17s} Bxg3+ {+M7/188 9.2s} 45. Kxg3 {-29.63/4 16s} R@g4+ {+M3/245 0.046s} 46. Kh2 {-99.99/2 0.048s} P@g3# {+M1/245 0.020s, Black mates} 0-1

Figure 6.13: Game 63 / 100



(a) Evaluation progression for both engines

[Event "RL-Eval"]
[Site "Darmstadt, GER"]
[Date "2019.12.19"]
[White "stockfish-x86_64-modern 2019-12-03"]
[Black "CrazyAra-0.7.0-Model-OS-45"]
[Result "1-0"]
[PlyCount "103"]
[TimeControl "900+10"]
[Variant "crazyhouse"]

1. e4 {book} c5 {book} 2. Nc3 {book} Nc6 {book} 3. f4 {book} d6 {-0.41/36 26s} 4. Nf3 {+1.70/28 74s} Nf6 {-0.46/41 26s} 5. Bb5 {+2.29/24 8.1s} Bg4 {-0.53/38 26s} 6. O-O {+2.48/25 12s} g6 {-0.49/45 26s} 7. e5 {+2.50/23 12s} Nh5 {-1.41/39 40s} 8. exd6 {+4.83/25 17s} Qxd6 {-0.99/45 27s} 9. Ne4 {+4.96/25 13s} Qc7 {-0.76/50 26s} 10. Bxc6+ {+5.32/25 11s} Qxc6 {-0.90/55 26s} 11. Ne5 {+7.08/23 11s} B@d4+ {-1.83/50 40s} 12. P@e3 {+7.29/25 18s} Bxe5 {-2.32/51 26s} 13. Qxg4 {+7.50/25 24s} N@f6 {-2.30/49 26s} 14. Nxf6+ {+6.76/27 44s} Nxf6 {-2.95/46 27s} 15. Qe2 {+8.00/27 16s} Bxf4 {-3.49/48 26s} 16. Rxf4 {+8.05/28 35s} Bg7 {-3.86/46 26s} 17. N@e5 {+8.29/29 57s} O-O {-2.49/39 28s} 18. Nxc6 {+8.27/26 9.7s} bxc6 {-1.97/50 26s} 19. B@h4 {+8.75/27 17s} N@h5 {-0.80/49 27s} 20. Rf1 {+8.69/27 24s} N@f5 {-0.99/38 26s} 21. Rxf5 {+8.66/26 22s} gxf5 {-1.38/36 27s} 22. B@e5 {+8.61/27 38s} Kh8 {-1.55/47 27s} 23. d3 {+8.99/24 19s} R@g6 {-1.56/43 27s} 24. Kh1 {+10.37/25 27s} P@g4 {-0.89/36 28s} 25. Q@f1 {+12.65/25 15s} Rad8 {-1.97/41 40s} 26. Qxf5 {+14.51/25 14s} Rd5 {-2.07/47 27s} 27. N@c4 {+15.08/25 19s} P@f3 {-2.79/34 27s} 28. gxf3 {+17.07/26 27s} gxf3 {-3.36/38 27s} 29. Qexf3 {+17.07/27 33s} P@g2+ {-4.04/36 27s} 30. Kg1 {+18.18/27 31s} Ng4 {-4.25/34 27s} 31. Qxg6 {+16.60/29 60s} hxg6 {-4.55/45 27s} 32. Bxg7+ {+18.65/25 15s} Kxg7 {-4.79/32 27s} 33. Qxg2 {+19.76/23 25s} B@f6 {-6.85/28 43s} 34. Qxg4 {+25.05/23 20s} Bxh4 {-10.26/27 40s} 35. Qxh4 {+26.40/24 41s} Q@g5+ {-10.93/24 26s} 36. B@g3 {+29.89/22 21s} Qxh4 {-11.90/28 26s} 37. Bxh4 {+31.09/24 30s} Rf5 {-12.34/25 26s} 38. B@e5+ {+47.84/20 33s} B@f6 {-14.40/28 28s} 39. Bhxf6+ {+M31/26 12s} exf6 {-14.08/34 27s} 40. N@e6+ {+M29/30 21s} fxe6 {-15.48/32 27s} 41. Q@e7+ {+M27/35 15s} Q@f7 {-16.02/30 17s} 42. P@h6+ {+M21/38 14s} Kh7 {-16.66/28 17s} 43. B@g8+ {+M19/44 14s} Kxg8 {-17.42/26 16s} 44. Qxf7+ {+M17/50 13s} Rxf7 {-18.12/24 16s} 45. Q@e8+ {+M15/53 16s} Q@f8 {-19.07/22 16s} 46. N@e7+ {+M13/59 18s} Rxe7 {-19.53/12 16s} 47. R@h8+ {+M11/66 13s} Kxh8 {-20.99/10 0.041s} 48. Qxf8+ {+M9/98 12s} R@g8 {-23.48/8 16s} 49. P@g7+ {+M7/245 7.5s} Nxg7 {-27.91/6 15s} 50. hxg7+ {+M5/245 0.047s} Rxg7 {-32.77/4 15s} 51. Q@h6+ {+M3/245 0.046s} B@h7 {-99.99/2 15s} 52. Qhxg7# {+M1/245 0.027s, White mates} 1-0

Figure 6.14: Game 64 / 100

7 Conclusion

In the final chapter we summarize all results and give an outlook for potential future work.

7.1 Summary

In this work we presented a throrough reinforcement learning setup for a convolutional neural network to learn the game variant crazyhouse which was initalized by training on human expert games.

We demonstrated that the learnt model coupled with a MCTS search regime was able to surpass the playing level of the strongest $\alpha\beta$-search engine *Stockfish* under the presented time and hardware conditions. We welcome independent third party evaluations under different hardware and time controls to verify this statement. Neural networks appear to outscale traditional search engines in particular in highly complex domains for which the formulation of a handcrafted value function is challenging. The higher move complexitiy and branching factor within the search tree also seems to favor neural networks with MCTS because of being able to focus the search on the most promising lines thanks to a learnt policy function. Despite the increased importance of tactics in crazyhouse, MCTS search appears to be a valid choice.

By adapting the reinforcement learning setting, only one million games were needed to surpass *Stockfish*, the strongest competitor for this variant. The computational cost amounted for 0.14 GPU years which is significantly lower than for instance *KataGo* [41] with a cost of 1.4 GPU years. The creation of *KataGo* is arguably the most efficient reinforcement learning setup for the game of Go at the time of writing.

7.2 Future work

A replication for classical chess or other chess variants might be helpful to put the achievement into perspective. The experiment could also be repeated when starting from zero knowledge to investigate how many self-play games are needed to reach the playing strength of the initial supervised network. Moreover, the performance was evaluated as an intermediate result. Future assessment will describe for how long the model continues to improve in the presented setup. It is also highly probable that the reinforcement learning setting can be further accelerated by e. g. validating the mentioned approaches in Section 5.4.

Additionally, the time management system could be further improved by allocating less time on obvious and more time on complicated positions. Neural network architecture search continues to be an active research field and upgrading the convolutional neural network architecture is likely the most promising area for future progress. Additionally, the support for CPU usage could be optimized to allow future competitions on the same hardware as traditional engines. Furthermore, the MCTS could be extended to different domains including continuous and state and/or action spaces. At last the major weak spots of the MCTS search engine could be addressed, which is its vulnerability to forced tactical sequences and the inability to share knowledge between different subtrees.

The rapid development of *Stockfish* for chess after version 8 showed that it is able to keep up with new neural network engine competitors. It is highly likely that *Multi-Variant-Stockfish* will gain significant playing strength in crazyhouse as well in the near future.

Acknowledgments The author thanks the crazyhouse expert **FM** `optilink` for providing a crazyhouse opening suite with equal starting positions. He is also grateful to the `TheFinnisher` for sharing his list of popular crazyhouse openings. Further, he appreciates the work of the core *Multi-Variant-Stockfish* and *Fairy-Stockfish* developers, Daniel Dugovic, Fabian Fichter and Niklas Fiekas as well as the the *Stockfish* developer team. He thanks Patrick Schramowski and the Machine Learning Lab of TU Darmstadt for supporting this work by granting access to the DGX2 server instance. Moreover, he appreciates the help of GitHub user `Matuiss2` for evaluating the progression of *Multi-Variant-Stockfish* in crazyhouse. Finally, the author thanks Thibault Duplessis and the lichess.org team for maintaining the lichess.org database.

Bibliography

[1] Jonathan Baxter, Andrew Tridgell, and Lex Weaver. “Knightcap: a chess program that learns by combining td (lambda) with game-tree search”. In: *arXiv preprint cs/9901002* (1999).

[2] 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.

[3] Murray Campbell, A Joseph Hoane, and Feng-Hsiung Hsu. “Deep blue”. In: *Artificial intelligence* 134.1 (2002), pp. 57–83.

[4] Tianqi Chen et al. “Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems”. In: *arXiv preprint arXiv:1512.01274* (2015).

[5] Sun-Yu Gordon Chi. “Exploring the Performance of Deep Residual Networks in Crazyhouse Chess”. In: *arXiv preprint arXiv:1908.09296* (2019).

[6] Johannes Czech et al. “Learning to play the Chess Variant Crazyhouse above World Champion Level with Deep Neural Networks and Human Data”. In: *arXiv preprint arXiv:1908.06660* (2019).

[7] 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.

[8] Sacha Droste and Johannes Fürnkranz. “Learning the piece values for three chess variants”. In: *ICGA Journal* 31.4 (2008), pp. 209–233.

[9] Daniel Dugovic, Fabian Fichter, and Niklas Fiekas. *Multi-variant fork of popular UCI chess engine*. original-date: 2014-07-31T23:11:27Z. July 2019. URL: `https://github.com/ddugovic/Stockfish` (visited on 07/29/2019).

[10] Niklas Fiekas. *niklasf/python-chess*. original-date: 2012-10-03T01:55:50Z. Dec. 2019. URL: `https://github.com/niklasf/python-chess` (visited on 12/28/2019).

[11] Free Software Foundation. *gnu.org*. https://www.gnu.org/licenses/gpl-3.0.en.html. [accessed 2019-07-30]. 2017. URL: https://www.gnu.org/licenses/gpl-3.0.en.html (visited on 07/18/2019).

[12] Kunihiko Fukushima. "Neocognitron: A hierarchical neural network capable of visual pattern recognition". In: *Neural networks* 1.2 (1988), pp. 119–130.

[13] Sylvain Gelly et al. "The Grand Challenge of Computer Go: Monte Carlo Tree Search and Extensions". In: *Communications of the ACM* 55.3 (2012), pp. 106–113. URL: http://hal.inria.fr/hal-00695370/PDF/CACM-MCTS.pdf.

[14] Xavier Glorot and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks". In: *Proceedings of the thirteenth international conference on artificial intelligence and statistics*. 2010, pp. 249–256.

[15] 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.

[16] 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.

[17] 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.

[18] Klaus Iglberger et al. "Expression templates revisited: a performance analysis of current methodologies". In: *SIAM Journal on Scientific Computing* 34.2 (2012), pp. C42–C69.

[19] Klaus Iglberger et al. "High performance smart expression template math libraries". In: *2012 International Conference on High Performance Computing & Simulation (HPCS)*. IEEE. 2012, pp. 367–373.

[20] Stefan-Meyer Kahlen and G. Harm Muller. *UCI protocol*. http://wbec-ridderkerk.nl/html/UCIProtocol.html. [accessed 2019-06-05]. Jan. 2004. URL: http://wbec-ridderkerk.nl/html/UCIProtocol.html (visited on 06/05/2019).

[21] Levente Kocsis and Csaba Szepesvári. "Bandit Based Monte-carlo Planning". In: *Proceedings of the 17th European Conference on Machine Learning (ECML'06)*. Berlin, Germany: Springer-Verlag, 2006, pp. 282–293. ISBN: 3-540-45375-X, 978-3-540-45375-8. DOI: 10.1007/11871842_29.

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

[23] 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.

[24] Ilya Loshchilov and Frank Hutter. “Sgdr: Stochastic gradient descent with warm restarts”. In: *arXiv preprint arXiv:1608.03983* (2016).

[25] Alistair Miles. *zarr-developers/zarr-python*. original-date: 2015-12-15T14:49:40Z. Dec. 2019. URL: https://github.com/zarr-developers/zarr-python (visited on 12/19/2019).

[26] Francesco Morandin et al. “SAI: a Sensible Artificial Intelligence that plays with handicap and targets high scores in 9x9 Go (extended version)”. In: *arXiv preprint arXiv:1905.10863* (2019).

[27] Constantin Pape. *constantinpape/z5*. 2019. DOI: 10.5281/ZENODO.3585752. URL: https://zenodo.org/record/3585752.

[28] Mark Sandler et al. “Mobilenetv2: Inverted residuals and linear bottlenecks”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2018, pp. 4510–4520.

[29] 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.

[30] David Silver et al. “Mastering chess and shogi by self-play with a general reinforcement learning algorithm”. In: *arXiv preprint arXiv:1712.01815* (2017).

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

[32] David Silver et al. “Mastering the game of Go without human knowledge”. In: *Nature* 550 (2017), pp. 354–359.

[33] 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.

[34] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[35] Gerald Tesauro. “Temporal difference learning and TD-Gammon”. In: *Communications of the ACM* 38.3 (1995), pp. 58–68.

[36] Thibault Thibault. *lichess.org game database*. Jan. 2013. URL: https://database.lichess.org/ (visited on 06/08/2019).

[37] Yuandong Tian and Yan Zhu. “Better computer go player with neural network and long-term prediction”. In: *arXiv preprint arXiv:1511.06410* (2015).

[38] Yuandong Tian et al. “Elf opengo: An analysis and open reimplementation of alphazero”. In: *arXiv preprint arXiv:1902.04522* (2019).

[39] Joel Veness et al. “Bootstrapping from game tree search”. In: *Advances in neural information processing systems*. 2009, pp. 1937–1945.

[40] Hui Wang et al. “Hyper-Parameter Sweep on AlphaZero General”. In: *arXiv preprint arXiv:1903.08129* (2019).

[41] David J Wu. “Accelerating Self-Play Learning in Go”. In: *arXiv preprint arXiv:1902.10565* (2019).

[42] Sergey Zagoruyko and Nikos Komodakis. “Wide residual networks”. In: *Proceedings of the British Machine Vision Conference (BMVC)*. 2016.