frontiers in Artificial Intelligence

ORIGINAL RESEARCH
published: 28 April 2020
doi: 10.3389/frai.2020.00024

Learning to Play the Chess Variant Crazyhouse Above World Champion Level With Deep Neural Networks and Human Data

Johannes Czech[1]*, *Moritz Willig*[1], *Alena Beyer*[1], *Kristian Kersting*[1,2] and *Johannes Fürnkranz*[3]

[1] Department of Computer Science, TU Darmstadt, Darmstadt, Germany, [2] Centre for Cognitive Science, TU Darmstadt, Darmstadt, Germany, [3] Department of Computer Science, JKU Linz, Linz, Austria

Deep neural networks have been successfully applied in learning the board games Go, chess, and shogi without prior knowledge by making use of reinforcement learning. Although starting from zero knowledge has been shown to yield impressive results, it is associated with high computationally costs especially for complex games. With this paper, we present *CrazyAra* which is a neural network based engine solely trained in supervised manner for the chess variant crazyhouse. Crazyhouse is a game with a higher branching factor than chess and there is only limited data of lower quality available compared to *AlphaGo*. Therefore, we focus on improving efficiency in multiple aspects while relying on low computational resources. These improvements include modifications in the neural network design and training configuration, the introduction of a data normalization step and a more sample efficient Monte-Carlo tree search which has a lower chance to blunder. After training on 569537 human games for 1.5 days we achieve a move prediction accuracy of 60.4%. During development, versions of *CrazyAra* played professional human players. Most notably, *CrazyAra* achieved a four to one win over 2017 crazyhouse world champion Justin Tan (aka *LM Jann Lee*) who is more than 400 Elo higher rated compared to the average player in our training set. Furthermore, we test the playing strength of *CrazyAra* on CPU against all participants of the second Crazyhouse Computer Championships 2017, winning against twelve of the thirteen participants. Finally, for *CrazyAraFish* we continue training our model on generated engine games. In 10 long-time control matches playing *Stockfish 10*, *CrazyAraFish* wins three games and draws one out of 10 matches.

Keywords: deep learning, chess, crazyhouse, supervised learning, Monte-Carlo tree search

OPEN ACCESS

Edited by:
Balaraman Ravindran, Indian Institute of Technology Madras, India

Reviewed by:
Michelangelo Ceci, University of Bari Aldo Moro, Italy
Ameet Soni, Swarthmore College, United States
Manan Tomar, Facebook, United States, in collaboration With Reviewer Ameet Soni

***Correspondence:**
Johannes Czech
johannes.czech@cs.tu-darmstadt.de

Specialty section:
This article was submitted to Machine Learning and Artificial Intelligence, a section of the journal Frontiers in Artificial Intelligence

Received: *20 August 2019*
Accepted: *26 March 2020*
Published: *28 April 2020*

Citation:
Czech J, Willig M, Beyer A, Kersting K and Fürnkranz J (2020) Learning to Play the Chess Variant Crazyhouse Above World Champion Level With Deep Neural Networks and Human Data. Front. Artif. Intell. 3:24. doi: 10.3389/frai.2020.00024

1. INTRODUCTION

The project *AlphaZero* (Silver et al., 2017a) with its predecessors *AlphaGoZero* (Silver et al., 2017b) and *AlphaGo* (Silver et al., 2016) marks a milestone in the field of artificial intelligence, demonstrating that the board games Go, chess, and shogi can be learned from zero human knowledge. In this article, we extend this family of games. We present the neural network based engine *CrazyAra* which learned to play the chess variant crazyhouse solely in a supervised fashion.

Crazyhouse, also known as drop chess, is the single-player version of the game bughouse and introduces the ability to re-introduce pieces that have been captured. The captured piece switches its color, and is henceforth held in the capturing so-called pocket of the respective play. Crazyhouse incorporates all classical chess rules including castling, en passant capture, and draw by three-fold repetition. In addition, instead of playing a classical move, the player has the option to drop any pocket piece onto an empty square of the board, with the exception that pawns cannot be dropped on the first or eighth rank. The element of dropping captured pieces is similar to the game of shogi, with the difference that in crazyhouse pawns can also be dropped to deliver immediate checkmate. The fact that pieces never fully leave the game makes crazyhouse a highly tactical game with a considerably larger branching factor than conventional chess. The chance of drawing and the average game length are significantly reduced because games almost always end in checkmate, and the element of the chess endgame is missing. Moreover, the ability to drop pieces to set your opponent in check often enables long forced checkmating sequences. Furthermore, crazyhouse is commonly played in short time controls, and is increasing in popularity particularly for the online community.

We approach the problem by training a deep neural network from human game data, in a similar fashion as *AlphaGo* (Silver et al., 2016). Unfortunately, *AlphaGo* is difficult, if not impossible, to directly apply to crazyhouse. First of all, there is only a significantly smaller data set of lower quality available compared to Go or chess. Second, because of the higher move complexity and the more dynamic nature of the game, several challenges had to be overcome when adapting the neural network architecture and applying Monte-Carlo Tree Search (MCTS). Specifically, our contributions are as follows:

- First, we introduce a more compact input board presentation by making the state fully Markovian and removing the history component.
- Second, we highlight the importance of input preprocessing in form of rescaling or normalization for significant better performance.
- Third, we present a new more powerful and more efficient neural network architecture based on advancements in computer vision such as grouped depthwise convolutions, pre-activation resnets, and squeeze-excitation layers.
- Fourth, we investigate several techniques to make the Monte Carlo tree search (MCTS) more sample efficient. This includes the usage of *Q*-Values for move selection, a transposition table which allows sharing evaluations across multiple nodes, and ways to decrease the chances of missing critical moves.
- Finally, we evaluate the strength of a neural network in combination with MCTS with expert human players as well as the most common crazyhouse chess engines.

We proceed as follows. We start off, in section 2, by briefly reviewing prior work in computer crazyhouse and machine learning in games. Section 3 then goes through the general scheme on how the neural network is trained and integrated with MCTS to be used as an engine. Our input representation for encoding the board state is introduced in section 4, and our output representation in section 5. Section 6 goes over the *AlphaZero* network architecture and introduces different convolutional neural network designs, which make use of pre-activation residual blocks (He et al., 2016b), depthwise separable convolutions (Howard et al., 2017), and Squeeze Excitation Layers (SE; Hu et al., 2018). Next, in section 7, we describe the training data in more detail, including statistics of the most frequent players, the win and draw percentages as well as the occurrence of different time controls. We also summarize how a computer generated data set based on *Stockfish* self play games was created. Then, the configuration of the supervised training is provided in section 8 and the performance and progress for different network architectures is visualized. Section 9 outlines the formulas for the MCTS algorithm including its hyperparameter settings. We also introduce several changes and extensions such as including *Q*-values for final move selection, a conversion of *Q*-Values to Centi-Pawn (CP), the usage of a transposition table, a parameterization for calculating the *U*-Values and a possible integration of domain knowledge to make the search more robust and sample efficient. We continue with a discussion in section 10 highlighting the benefits and disadvantages of MCTS compared to Alpha-Beta minimax search. Before concluding, we summarize the match results with human professional players and other crazyhouse engines.

2. RELATED WORK ON COMPUTER CRAZYHOUSE AND ML IN BOARD GAMES

Crazyhouse is a fairly recent chess variant, which primarily enjoys popularity in on-line chess servers such as lichess.org. Despite its youth, there are already more than a dozen engines available which are able to play this chess variant (cf. also section 11.2). The first two of these engines are *Sjeng*[1], written by Gian-Carlo Pascutto released in 1999, and *Sunsetter*[2], developed by Georg v. Zimmermann and Ben Dean-Kawamura in 2001. Later the strong open-source chess engine *Stockfish*[3] has been adapted to play crazyhouse by Daniel Dugovic, Fabian Fichter, and Niklas Fiekas. *Stockfish* won the first Crazyhouse Computer Championships 2016 and also the second Crazyhouse Computer Championships 2017 (Mosca, 2017). All these engines have in common that they follow a minimax search regime with alpha-beta pruning, as has been popularized by successful chess engines, most notably *DeepBlue* (Campbell et al., 2002). These engines are often described as having a large number of node evaluations and being strong at tactics in open positions, while sometimes having trouble in generating strategical attacks[2]. Due to the higher branching factor in crazyhouse, engines commonly reach a significantly lower search depth compared to classical chess.

Generally, machine learning in computer chess (Skiena, 1986; Fürnkranz, 1996) and in computer game playing in general has a long history (Fürnkranz, 2017), dating back to Samuel's checkers player (Samuel, 1959), which already pioneered many components of modern systems, including linear evaluation

[1]https://www.sjeng.org/indexold.html (accessed July 30, 2019).
[2]http://sunsetter.sourceforge.net/ (accessed July 30, 2019).
[3]https://github.com/ddugovic/Stockfish (accessed July 30, 2019).

functions and reinforcement learning. The original example that succeeded in tuning a neural network-based evaluation function to expert strength by playing millions of games against itself is the backgammon program *TD-Gammon* (Tesauro, 1995). Similar ideas have been carried over to other board games (e.g., the chess program *KnightCap*; Baxter et al., 2000), but the results were not as striking.

Monte-Carlo Tree Search (MCTS; Kocsis and Szepesvári, 2006) brought a substantial break-through in the game of Go (Gelly et al., 2012), featuring the idea that instead of using an exhaustive search to a limited depth, as was common in chess-like games, samples of full-depth games can be used to dynamically approximate the game outcome. While MCTS works well for Go, in games like chess, where often narrow tactical variations have to be found, MCTS is prone to fall into shallow traps (Ramanujan et al., 2010). For such games, it has thus been considered to be considerably weaker than minimax-based approaches, so that hybrid algorithms have been investigated (Baier and Winands, 2015).

Recently, *AlphaGo* has brought yet another quantum leap in performance by combining MCTS with deep neural networks, which were trained from human games and improved via self play (Silver et al., 2016). *AlphaZero* improved the architecture of *AlphaGo* to a single neural network which can be trained without prior game knowledge. Silver et al. (2017a) demonstrated the generality of this approach by successfully applying it not only to Go, but also to games like chess and shogi, which have previously been dominated by minimax-based algorithms. While *AlphaZero* relied on the computational power of large Google computation servers, for the games of Go[4] and chess[5] the *Leela* project was started with the goal to replicate *AlphaZero* in a collaborative effort using distributed computing from the crowd. Several other engines[6] built up on *Leela* or are partly based on the source code of the *Leela* project. Our work on crazyhouse started as an independent project. Generally, only little work exists on machine learning for crazyhouse chess. One exception is the work of Droste and Fürnkranz (2008), who used reinforcement learning to learn piece values and piece-square values for three chess variants including crazyhouse. In parallel to our work, Chi (2018) also started to develop a neural network to learn crazyhouse.

3. OVERVIEW OF THE *CRAZYARA* ENGINE

Our crazyhouse bot *CrazyAra* is based on a (deep) neural network that has been first trained on human games and is then optionally refined on computer-generated games. The network is used in a MCTS-based search module, which performs a variable-depth search. In the following, we first briefly sketch all components (see **Figure 1**). The details will then be described in subsequent sections.

[4]https://github.com/leela-zero/leela-zero (accessed June 10, 2019).
[5]https://github.com/LeelaChessZero/lc0 (accessed June 10, 2019).
[6]See e.g., https://github.com/manyoso/allie, https://github.com/Cscuile/BetaOne (accessed June 10, 2019).

3.1. Deep Neural Networks for Evaluating Moves

CrazyAra trains a (deep) neural network model $f_\theta(s)$ to predict the value $v \in [-1, 1]$ of a board state s and its corresponding stochastic policy distribution $\mathbf{p}$ over all available moves (section 5). Since we are learning the network based on preexisting matches, we encode the ground truth label as a one-hot vector. Specifically, depending on the final outcome of the game, each board state is assigned one of three possible outcomes {−1: lost, 0: draw, 1: win} from the point of view of the player to move. Note that the number of draws is below 1 % in human crazhouse games and thus considerably lower than in classical chess. These assignments are based on the assumption that given a considerable advantage in a particular position for a player, then it is highly unlikely that the respective player will lose the game. This assumption is, however, heavily violated in our data set partly due to the popularity of low time control games (see section 7).

Actually, the neural network is a shared network, which both predicts the value and policy (section 6) for a given board state in a single forward pass. The loss function is defines as follows

$$l = \alpha(z - v)^2 - \pi^T \log \mathbf{p} + c\|\theta\|^2 \tag{1}$$

where z is the true value, v the predicted value, π the true policy, $\mathbf{p}$ the predicted policy and c the L_2 regularization constant for the network weights θ, respectively. We set α to be 0.01 in order to avoid overfitting to the training values as suggested by Silver et al. (2016). The weights of the neural network are adjusted to reduce the loss by Stochastic Gradient Descent with Neterov's Momentum (NAG; Botev et al., 2017) (see also section 8). We keep the neural network weights fixed after training and do not apply any reinforcement learning using self-play yet.

3.2. Monte-Carlo Tree Search for Improving Performance

The trained (deep) neural network of *CrazyAra* has a move prediction accuracy of about 60%, i.e., covers most of the play-style of the average playing strength in the training set. Its performance is then improved using a variant of the Upper Confidence Bounds for Trees algorithm (Kocsis and Szepesvári, 2006), which integrates sample-based evaluations into a selective tree search. Like with all Monte-Carlo tree search algorithms (Browne et al., 2012), the key idea is to evaluate moves by drawing samples, where good moves are sampled more frequently, and less promising moves are sampled less frequently, thereby trading off exploration and exploitation. The key component is the (deep) neural network f_θ which guides the search in each step. For each iteration t at state s_t, the following UCT-like formula (Equation 2) is used for selecting the next action a_t leading to a new state s_{t+1}.

$$\begin{aligned} a_t &= \text{argmax}_a(\text{Q}(s_t, a) + U(s_t, a)) \\ \text{where} \quad U(s, a) &= c_{\text{puct}} P(s, a) \frac{\sqrt{\sum_b N(s, b)}}{1 + N(s, a)} \end{aligned} \tag{2}$$

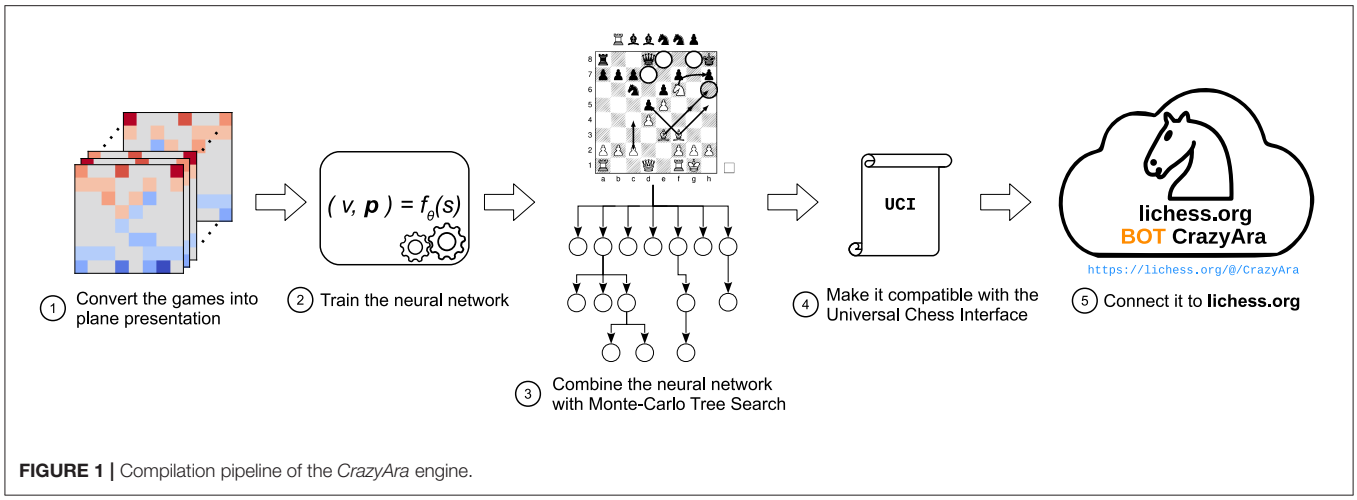

FIGURE 1 | Compilation pipeline of the *CrazyAra* engine.

This move selection continues until a new previously unexplored state s^* or a terminal node s_T is reached. The state s^* is then expanded, the corresponding position is evaluated using the learned neural network, and the received value evaluation is propagated through all nodes on the path beginning from the current position s^* back to the root position s_0. In case of a terminal node s_T the static evaluation according to the game rules being either -1, 0, or $+1$ is used instead. After enough samples have been taken, the most promising move, i.e., the node which has been visited most often, is played on the board (section 9).

3.3. Availability of *CrazyAra*

CrazyAra is compatible with the Universal Chess Interface (UCI; Kahlen and Muller, 2004) and uses the BOT API of lichess.org[7] to provide a public BOT account[8], which can be challenged to a match when online. Furthermore, the executable and full source code, including the data preprocessing, architecture definitions, training scripts[9], and MCTS search[10] is available under the terms of the GNU General Public License v3.0 (GPL-3.0; Free Software Foundation, 2017).

Let us now dive into the details of each component of *CrazyAra*.

4. INPUT REPRESENTATION OF *CRAZYARA*

The input to *CrazyAra* is a stack of 8×8 planes where each channel describes one of the feature of the current board state described in **Table 1**. The main extension compared to encodings used for classical chess is the addition of pocket pieces accounting for 10 additional channels. Furthermore, we remove the seven step history of previous board states and only consider the current board position as input. This decision has several motivations. Crazyhouse, as well as chess is a full information game. In theory, the history is not needed to find the best move. Dropping the game history also allows better compatibility to use the engine in analysis mode. Otherwise one would have to add dummy layers, which can distort the network predictions. Avoiding the history also reduces the amount of model parameters, the storage for saving and preprocessing the data set. It also allows to have fewer channels early in the network, as shown in **Figure 3**, as well as having higher batch sizes during training and inference. Furthermore, it is more consistent in combination with the use of transposition tables (see section 9.2.5). For instance a position might be reached from different transpositions and be assigned a different value and policy target during training. In case of a history free representation, the model will converge to the mean of both training samples but with history and a small data set, it might overfit to the individual targets. Furthermore, the policy is not optimized to predict the best move in a position but the same move as in corresponding game of the training set. This results in the network trying to imitate the playing behavior of the current player based on his and the opponent's past moves. As a downside, however, we lose the attention mechanism provided by the move history and lose information to detect "fortress" positions which are signified by a series of shuffling moves and more common in standard chess. Overall, this decision was based on reducing complexity and avoiding negative side effects rather than improving the validation loss.

To make the board presentation fully Markovian, we add an additional feature channel, which highlights the square for an en-passent capture if possible. In contrast to standard chess, piece promotions are a lot more common and often occur multiple times in a game. If a promoted piece gets captured, it will be demoted to a pawn again and added to the pocket of the other player. To encode this behavior, we highlight each square of a piece that has been promoted using a binary map for each player. Overall, the representation to *CrazyAra* is fully compatible with the standard Forsyth–Edwards Notation (FEN) for describing a particular board position in a compact single ASCII string,

[7]https://github.com/careless25/lichess-bot (accessed June 5, 2019).
[8]https://lichess.org/@/CrazyAra (accessed June 5, 2019).
[9]https://github.com/QueensGambit/CrazyAra (accessed June 5, 2019).
[10]https://github.com/QueensGambit/CrazyAra-Engine (accessed July 7, 2019).

TABLE 1 | Plane representation for crazyhouse.

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
Color*	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)
Total	**34**		

The features are encoded as a binary maps and features with are single values set over the entire* 8×8 *plane.*

which holds all necessary information to recover a game state[11]. However, the information of how often a particular position has already occurred gets lost when converting our representation into FEN.

In over-the-board (OTB) games, the players see their pieces usually in the first rank. We make use of symmetries by flipping the board representation on the turn of the second player, so that the starting square of the queen is always to the left of the king for both players.

4.1. Input Normalization

All input planes are scaled to the linear range of $[0, 1]$. For most computer vision tasks, each channel of an image is encoded as 8 bit per pixel resulting in a discrete value between $[0, 255]$. Commonly, these input images are divided by 255 to bring each pixel in the $[0, 1]$ range. For ImageNet (Russakovsky et al., 2015), it is also frequent practice to subtract the mean image from the train data set for every input. In the case of crazyhouse chess, we define the following constants, which act as a maximum value for each non-binary feature (see **Table 1**). We set the maximum number of pocket pieces for each piece type to 32, the maximum number of total moves to 500, and the maximum value of the no progress counter to 40[12]. Next we divide each correspond feature of an input sample with these maximum values. Note that these maximum values only describe a soft boundary and can be violated without breaking the neural network predictions. Some values such as the maximum number of pocket pieces and the maximum number of moves could have also been set to a different similar value.

To illustrate the benefits of including this step, we conducted the following experiments, learning curves shown in **Figure S1**, p. 2. We trained a small AlphaZero like network with seven residual blocks on a subset of our training data using 10000 games. For the optimizer we used ADAM (Kingma and Ba, 2015) with its default parameters: Learning-rate =0.001, $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10{-}8$ and Stochastic Gradient Descent with Neterov's Momentum (NAG; Botev et al., 2017) using our learning rate and momentum schedule (cf. section 8). When training both optimizer with and without normalization for seven epochs with a weight decay of 10^{-4} and a batch size of 1024, one can make the following observations. Both optimizers highly benefit in terms of convergence speed and final convergence when using our input pre-processing step. ADAM gains $+2, 3\%$ whereas NAG gains $+9, 2\%$ move prediction accuracy. The ADAM optimizer is much more robust when dealing with the unnormalized feature set due to its internal automatic feature rescaling, but is outperformed in terms of generalization ability when using NAG with our defined learning and momentum schedule. This agrees with research on different optimizer (Keskar and Socher, 2017).

4.2. Illustrative Example for Predictions

Essentially, we treat learning to play crazyhouse as modified computer vision problem where the neural network f_θ conducts a classification for the policy prediction combined with a regression task for the value evaluation.

Figure 2 visualizes our input representation using an exemplary board position of the first game in our test set. The pseudo color visualization, which neglects the pocket information of both players for better visualization purposes, shows how the neural network receives the board as a 8×8 multi-channel image.

Next, the activation maps of a fully trained network (section 6) for a single prediction are shown in **Figure 3**. The activation maps in **Figure 3A** are the extracted features after the first convolutional layer and in **Figure 3B** after the full residual tower. In the initial features, the pawn formations for black and white are still recognizable and the features are sparse due to the usage of ReLU activation and our compact input representation. The

[11] More information on the FEN notation can, e.g., be found at: https://en.wikipedia.org/wiki/Forsyth-Edwards_Notation.

[12] According to the 50 move rule a value of 50 is recommended instead.

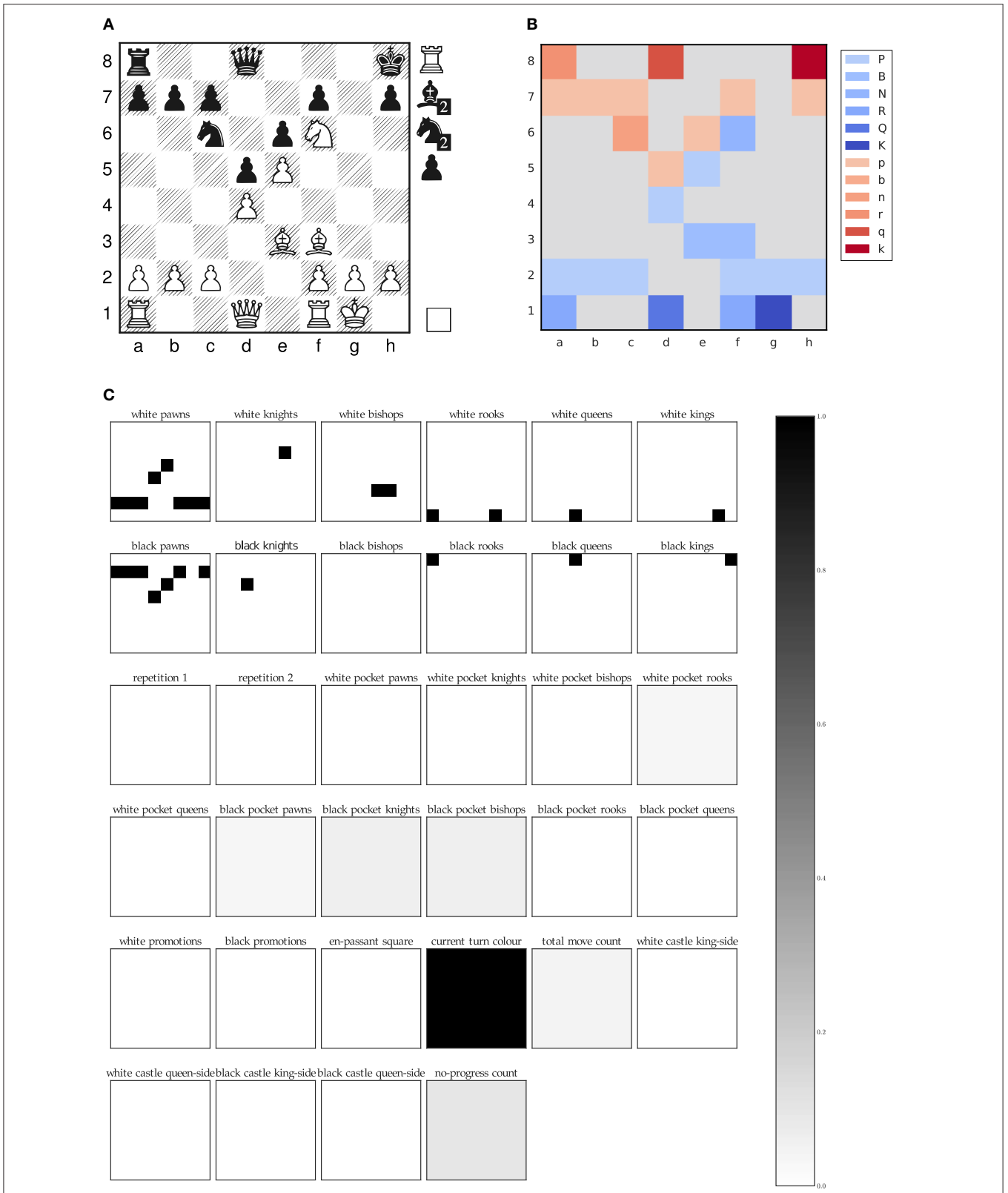

FIGURE 2 | Plane representation of an exemplary game position of our test data set, FEN: `r2q3k/ppp2p1p/2n1pN2/3pP3/3P4/4BB2/PPP2PPP/R2Q1RK1[Rbbnnp] w - - 4 22`. **(A)** Classical board representation, **(B)** Pseudo color visualization, **(C)** Plane representation.

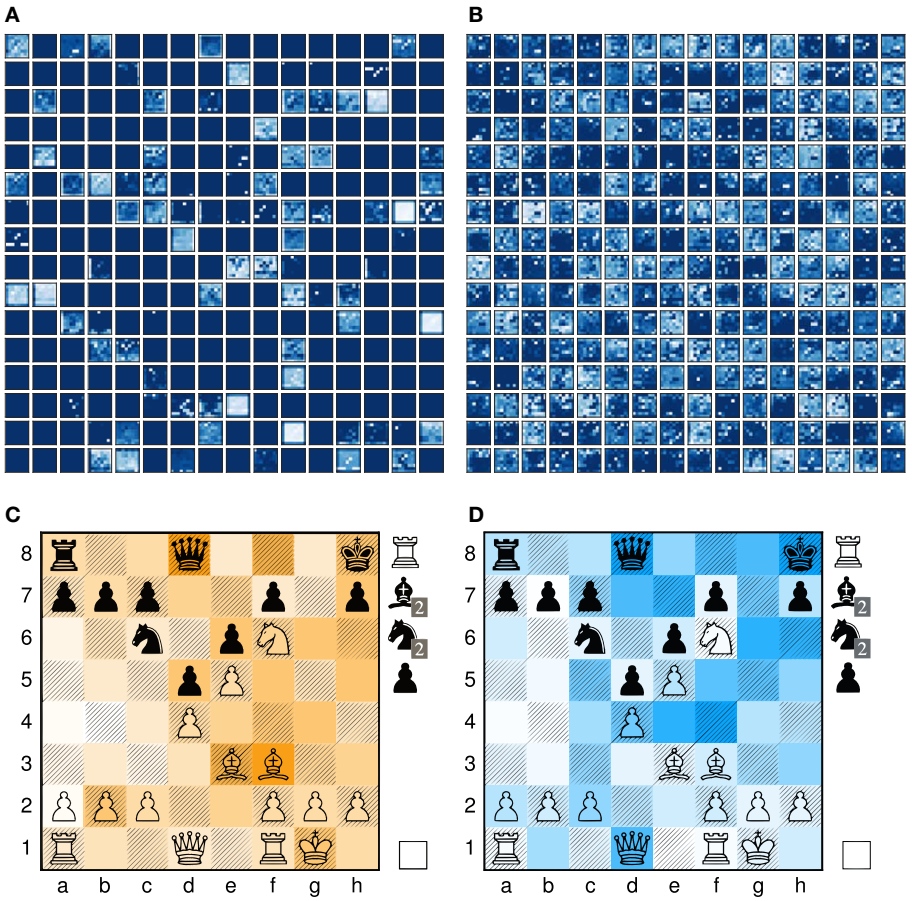

FIGURE 3 | Activation maps of model 4-value-8-policy when processing input sample (**Figure 2**) from the test data set. **(A)** Features after conv0-batchnorm0-relu0, **(B)** Features after the full residual tower, **(C)** Final value activation maps, **(D)** Final policy activation maps.

final features visually exhibit a high contrast and are fed to both the value and policy evaluation head.

The final spatial activation map of the value head are summed along the first axis and overlayed on top of the given board position in **Figure 3C**. Dark orange squares represent high activation regions and unsaturated those of low importance. The value evaluation fundamentally differs from those used in other crazyhouse engines and fully relies on the estimated winning chances by the neural network for a particular position. Formulating a handcrafted value evaluation for crazyhouse is challenging because game positions often require high depth in order to reach a relatively stable state and the element of initiative and king safety is intensified and hard to encode as a linear function. In the case of *Stockfish*, its specialist list of chess hyper parameters including the values for pieces on the board and in the pocket, `KingAttackWeights`, `MobilityBonus`, and `QueenThreats` has been carefully fine-tuned for the crazyhouse variant with the help of a SPSA Tuner (Kiiski, 2014; Fichter, 2018).

Based on the games of *CrazyAra* with other crazyhouse engines and human players, it seems to attach high importance to initiative and often seemingly, intuitively sacrifices pieces in tactical or strategic positions in order to increase the activity of its pieces. This often leads to strong diverging position evaluations compared to other engines (section 11.2) because the value evaluation of most engines is fundamentally based on material. *CrazyAra* seems to have a higher risks of choosing a sacrifice which was not sufficiently explored or missing a tactical sequence rather than slowly losing due to material disadvantage. The tendency for an active play-style is primarily due to a prominent proportion of human crazyhouse players in our data set which are known to play aggressively. Moreover, most players are more prone to make a mistake when being under attack and in time trouble which influences the value evaluation. On the other hand, there are also players which prefer consolidation moves instead of risking a premature attack. During the course of training, the network converges to the best fit limited by its model expressiveness which captures the play style of the majority of the games in our data set.

5. OUTPUT REPRESENTATION OF *CRAZYARA*

The value output, which represents the winning chances for a game position, represents as a single floating point value in the range $[-1, +1]$ as described in section 3.1.

For the policy output we investigate two conceptually different representations. First, the policy head consists of a single convolutional layer with a variable amount of feature channels followed by a fully connected layer storing all 2272 theoretical plausible moves as described in UCI-notation. Each entry in this vector stands for a particular predefined move and queen promoting moves are treated as individual moves, denoted with a suffix q, e.g., `e7e8q`. This form of policy representation was also used by[13].

In the second version, the policy head representation directly encodes each of the 2272 moves on a particular square on 81 channels of 8×8 planes: first come the queen moves in all compass direction, followed by all knight moves for each square, promoting moves, and finally all dropping moves. Queen moves also include all pawn, bishop, rook, and king moves. We spent three additional channels for queen promotion moves, to be consistent with the UCI-move-representation[14]. However, please keep in mind that most squares of the 5184 values describe illegal moves. This is due to the fact that the corresponding move would lead outside of the board and that promoting moves are only legal from the second last row on.

In the UCI-move-representation, en-passant moves and castling moves do not have separate vector indices and king side and queen side castling is denoted as the move `e1g1` and `e1c1`, respectively. Treating these as special move or as *king captures rook*, which would ensure a better compatibility with the chess960—also known as Fischer random variant, is a valuable alternative for the network.

6. DEEP NETWORK ARCHITECTURE OF *CRAZYARA*

Finding a (deep) neural network for playing board games covers three criteria, which determine the playing strength when using the learnt model in MCTS. First, the performance, i.e., the policy and value loss on the test data set is the main component for predicting the best moves at each roll-out. Second, a better performance is often associated with a longer training and inference time leading to less evaluations per second during search. Third, the memory consumption per prediction specifies the maximum achievable batch size.

To address these points, *CrazyAra* makes use of a dual architecture design with a tower of residual blocks followed by a value and policy head as recommended by Silver et al. (2017a). The originally proposed *AlphaZero* architecture differs from most common computer vision architectures in several ways: there are no down-sampling operation such as max-pooling, average pooling, or strided convolution to preserve the spatial size and no increase in feature channels per layer. Consequently, the number of parameters is rather high and most comparable with WideResnet (Zagoruyko and Komodakis, 2016). Moreover, in the final reduction step for selecting the classes, convolutional layers are used instead of a global average pooling layer due to the spatial importance of defining moves.

Residual connections (He et al., 2016a) play an important role for training computer vision architectures effectively. Later the original version has been revisited in ResneXt (Xie et al., 2017) making use of branching in form of group convolutions.

We train several different architectures on the same training set with the same optimizer settings[15].

Specifically, model *4-value-8-policy*, *8-value-16-policy*, and *8-value-policy-map* essentially follow the original *AlphaZero* network architecture (see **Table S2**) but use different value and policy heads. Specifically, *4-value-8-policy* means that four channels in the value head and eight channels in the policy head are used. *8-value-policy-map* has a policy head has a predefined mapping of move to squares.

First we tried training the original *AlphaGoZero* network architecture *1-value-2-policy* (Silver et al., 2017b), which has one channel for the value and two channels for the policy head. This unfortunately, did not work for crazyhouse and led to massive gradient problems, especially for deeper networks. The reason is that the policy for crazyhouse is much more complex than in the game of Go. In Go you can only drop a single piece type or pass a move, but in crazyhouse you can do any regular chess move and additionally drop up to five different piece types. When only relying on two channels, these layers turn into a critical bottleneck, and the network learns to encode semantic information on squares, which are rarely used in play. Based on our analysis of the policy activation maps, similar to **Figure 3D**, we observed that for these networks, usually squares on the queen-side hold information such as the piece type to drop. In cases where these squares are used in actual play, we encountered massive gradient problems during training. We found that at least eight channels are necessary to achieve relatively stable gradients.

RISEv2-mobile/*8-value-policy-map-mobile* (**Table 2**) is a new network design which replaces the default residual block with the inverted residual block of MobileNet v2 (Sandler et al., 2018) making use of group depthwise convolutions. Moreover, it follows the concept of the Pyramid-Architecture (Han et al., 2017): due to our more compact input representation only about half activation maps are used after the first convolution layer (**Figure 3A**). Therefore, the number of channels for the 3×3 convolutional layer of the first block start with 128 channels and is increased by 64 for each residual block reaching 896 channels in the last block. We call this block type an operating bottleneck block due to either reducing or expanding the number of channels.

It also uses Squeeze Excitation Layers (SE; Hu et al., 2018) which enables the network to individually enhance channels activation maps and based the winning entry of the ImageNet classification challenge ILSVRC in 2017 (Russakovsky et al., 2015). For our network we use a ration r of two and apply SE-Layer to the last five residual blocks. The name RISE originates from the Resnet architecture (He et al., 2016a; Xie et al., 2017), Inception model (Szegedy et al., 2016, 2017), and SE-Layers (Hu et al., 2018).

[13]https://github.com/Zeta36/chess-alpha-zero (accessed June 8, 2019)
[14]Details can be found in **Table S1**, p. 3.

[15]**Figure 4** was generated with the tool https://github.com/HarisIqbal88/PlotNeuralNet (accessed August 8, 2019).

TABLE 2 | RISEv2 mobile/*8-value-policy-map-mobile architecture*: 13 × 256.

Layer name		Output size		RISEv2 mobile 40-layer
conv0 batchnorm0 relu0		256 × 8 × 8		conv 3 × 3, 256
res_conv0_x res_batchnorm0_x res_relu0_x res_conv1_x res_batchnorm1_x res_relu1_x res_conv2_x res_batchnorm2_x shortcut + output		256 × 8 × 8		$\begin{bmatrix} \text{(SE-Block, } r=2) \\ \text{conv } 1\times1, 128+64x \\ \text{dconv } 3\times3, 128+64x \\ \text{conv } 1\times1, 256 \end{bmatrix} \times 13$
Value head	Policy head	1	2272/ 5184	see **Tables S4–S6**, p. 5–6.

Model *8-value-policy-map-preAct-relu+bn* (see **Table S3**) replaces the default residual block with a preactivation block (He et al., 2016b) and adds an additional batchnormalization layer to each residual block as suggested by Han et al. (2017).

Dong et al. (2017) and Zhao et al. (2017) discovered that the common 1:1 ratio between the number of convolutional layers and ReLU activations is suboptimal and that removing the final activation in a residual block or the first activation in the case of a pre-activation can result in improvements. Model *8-value-policy-map-preAct-relu+bn* and *RISEv2-mobile*/model8-value-policy-map-mobile follow a 2:1 ratio.

Furthermore, motivated by the empirical findings[16],[17],[18] we also tried flipping the conventional order of Batchnorm-ReLU into ReLU-Batchnorm. Here, we observed a faster convergence during training, but also witnessed NaN-values. The model continued to converge to NaN-values even when relying on checkpoints fall-backs of a healthy model state.

Our proposed model[19] *8-value-policy-map-mobile* is up to three times faster on CPU and 1.4 times faster on GPU. The reason why the model does not scale as efficiently on GPU like on CPU is because group convolution and Squeeze Excitation layers are not as suited for GPU computation because they cause memory fraction (Hu et al., 2018; Ma et al., 2018).

7. TRAINING DATA

Now that the architectures are in place, let us turn toward the data used for training. As training data we mainly used 569537 human games[20] played by lichess.org users from January 2016 to June 2018 in which both players had an Elo $\geq$ 2000. The majority of the games used for training have been played by a small group of active players: 20 players participated in 46.03% of all games[21].

In crazyhouse, the opening advantage for the first player is even more pronounced and the chance to draw is significantly reduced compared to chess. We used all games ending in checkmate, resignation, a draw or time forfeit except aborted games. All games and moves in our data set are equally weighted in the loss function (1).

The average Elo rating for a single game is 2279.61 and very short time controls are the most popular.

One minute games make up 45.15% of all games. As a consequence the chances of blundering is increased and the quality of moves usually declines over a course of a game. Some games are also won by winning on time in a lost position in which one person gives as many possible checks.

Additionally, however, we also generated a data set based on 121571 *Stockfish* self play games for training a version called *CrazyAraFish*. For each game position *Stockfish* used one million ±100000 nodes with a hash size of 512 mb. The opening book was based on a set of neural network weights[22] which was trained on the lichess.org database. Consequently the opening book corresponds to the most popular human crazyhouse openings. The opening suite features 1000 unique opening positions where each opening had 5.84 ± 2.77 plies. One ply describes a half-move in chess notation.

[16]https://github.com/pudae/tensorflow-densenet/issues/1 (accessed July 30, 2019).
[17]https://github.com/keras-team/keras/issues/1802 (accessed July 30, 2019).
[18]https://www.reddit.com/r/MachineLearning/comments/67gonq/d_batch_normalization_before_or_after_relu/ (accessed July 30, 07).
[19]Inference time comparison can be found in **Table S9**, p. 7

[20]https://database.lichess.org/ (accessed July 30, 2019).
[21]Statistics about the data set can be found in **Figures S3–S5**, p. 3–4.
[22]The model used was based on our RISEv1 architecture: https://github.com/QueensGambit/CrazyAra/wiki/Model-architecture

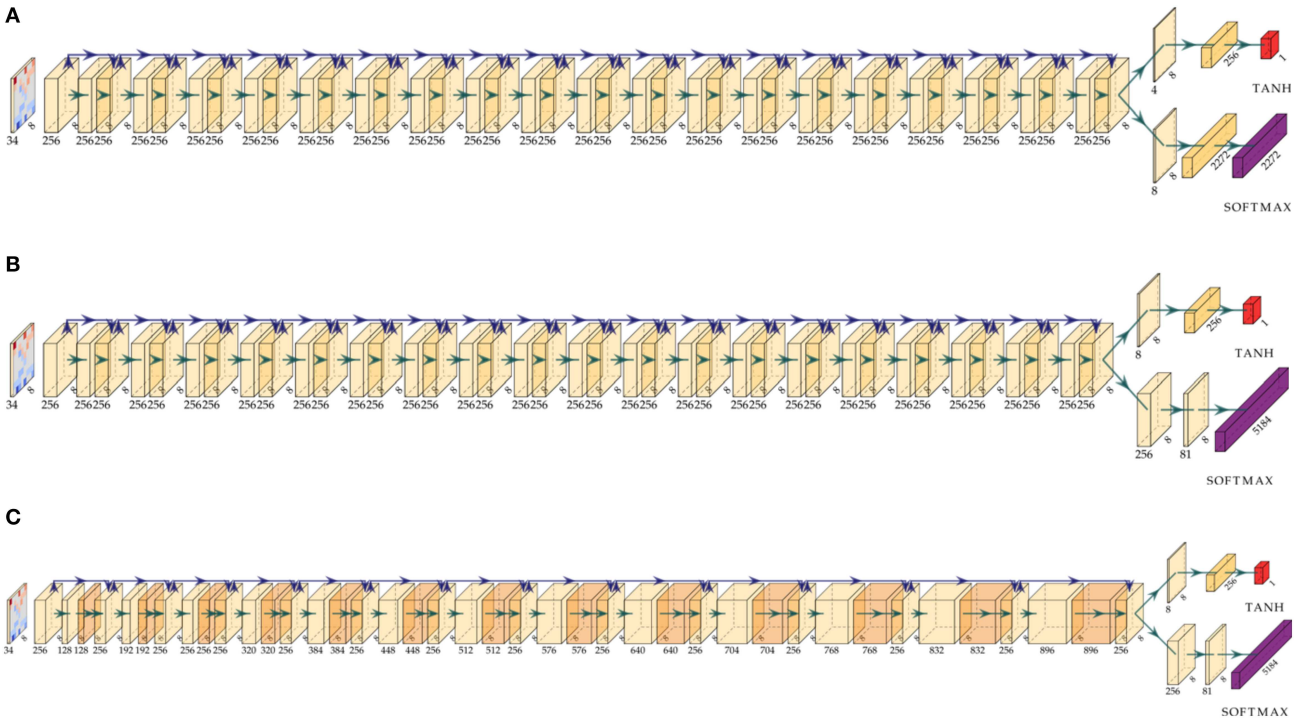

FIGURE 4 | Comparison of the activation map shapes for different architecture models. All models process the same input representation via a sequence of residual blocks followed by a value and policy head. **(A)** Model architecture 4-value-8-policy: 19 residual blocks followed by a value head with 4 channels and a policy head with 8 channels. **(B)** Model 8-value-policy-map: 19 residual blocks followed by a value head with 8 channels and a policy map representation. **(C)** Model RISEv2-mobile/8-value-policy-map-mobile architecture: 13 incrementally increasing operating bottleneck blocks.

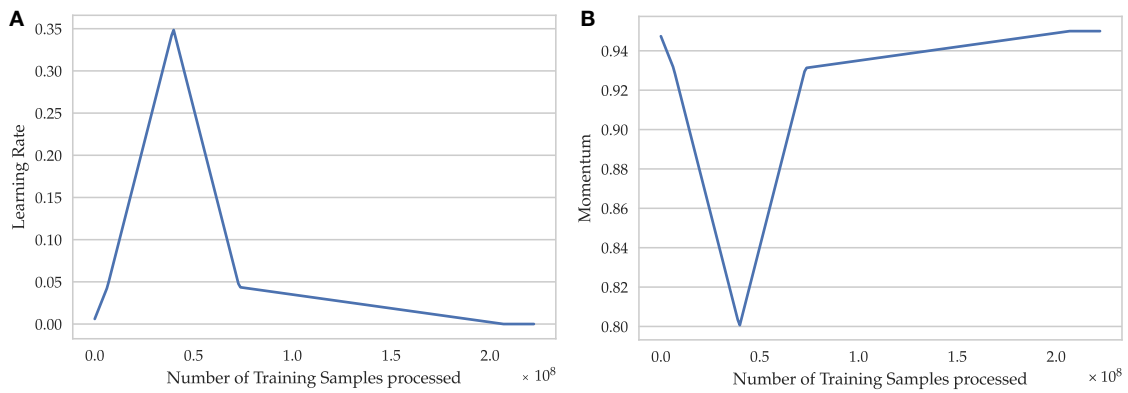

FIGURE 5 | Schedules used for modifying the parameters of Nesterov's stochastic gradient descent optimizer. **(A)** Learning rate schedule, **(B)** Momentum schedule.

8. SUPERVISED LEARNING TO PLAY CRAZYHOUSE

We trained the resulting models for seven epochs, using a one cycle learning rate schedule combined with a momentum schedule (Smith and Topin, 2019) and updating the weights of the neural network by Stochastic Gradient Descent with Neterov's Momentum (NAG; Botev et al., 2017). For the batch size we chose the highest value possible on our hardware, which is 1024, and a weight-decay of 10^{-4} for regularization. The first iterations were treated as a warm-up period followed by a linear long cool-down period. The maximum and minimum learning rate (**Figure 5**) were set to 0.35 and 0.00001, respectively, and were determined using a learning rate range test (Smith, 2018). The maximum learning rate of 0.35 is higher than the typical learning rates and acts as an additional source of regularization

(Smith and Topin, 2019). The momentum schedule was built based on the learning rate schedule (**Figure 5**) with a maximum value of 0.95 and minimum value of 0.85. Linear schedules, in contrast to a step-wise learning rate reduction by a fixed factor, greatly reduced the number of training iterations needed and also yielded a higher generalization ability. The advantages of linear schedules have also been verified on ImageNet by Mishkin et al. (2017). A summary of the hyperparameter configuration for supervised learning can be found in **Table S7**, p. 6.

As a new metric, we define the "*value accuracy sign*" metric, which determines if the network predicts the correct winner given a random game position (**Figures 6**, **7**). Drawn games are neglected for this metric. Generally, improving the performance on the value loss is harder because it has a lower influence on the combined loss. Nonetheless, the gap between validation and train loss is still higher for value prediction, which confirms that reducing its influence is necessary to avoid overfitting on this small data set (**Figure 6**).

Furthermore, the value loss decreased over the course of a game. After training, we evaluated our models using a mate in one data set (**Table 3**), which was generated using 1000 positions from our test set. On average there are 115.817 legal moves, from which 1.623 lead to direct mate. As our best mate in one accuracy, we achieved 0.955 with a value loss of 0.047. Because the first candidate moves sometimes lead to a mate in #2 or mate in #3 instead, we also provide the top five mate in one accuracy where we achieved a value of 0.999.

Model *8-value-policy-map-preAct-relu+bn* and *RISEv2-mobile/8-value-policy-map-mobile*, which both use a 2:1 convolution-ReLU ratio, performed best regarding the combined loss, but worse for the value loss compared to other models.

For training on the *Stockfish* self play data, we reused the above-mentioned neural network that we have also used for *Stockfish's* opening book (cf. footnote [22]) and the same supervised learning configuration. We employed transfer learning for parameter initialization which avoids relearning the move generation based on a smaller data set and enabled an initial move prediction accuracy of 46.29%. Alternatively, one could freeze the first layers of the network and only fine-tune the last layers or retrain the network from scratch. We changed the maximum learning rate to 0.01 and set the minimum learning rate 0.00001. After convergence the move prediction increased to 56.6% on the validation set. We also achieved a significant lower value loss of 0.4407 on this set. This is primarily because the games by *Stockfish* do not contain as many back and forth blunders as human games and the prior win-probability for White is higher.

9. CONFIGURATION OF THE MONTE-CARLO TREE SEARCH

Only relying on the initial prior policy $\mathbf{p}$ provided by the supervised model f_θ for playing will remain unsatisfactory in a complex environment such as crazyhouse. Therefore, the policy is now improved using Monte Carlo Tree Search (MCTS). An overview of the MCTS hyperparameters and their respective values is available in **Table S8**, p. 6.

9.1. Default Parameter Settings

For the MCTS we used the most recent version of the PUCT algorithm due (Silver et al., 2017a). The current game position which is subject to search is the root node of the search tree. If this position has already been expanded in the previous search then the respective subtree becomes the new search tree. Next, the statistics of the root node are updated through a continuous sequence of rollouts. A rollout or simulation is defined as the process of traversing the tree until a new unexplored node is encountered. Then, this node is expanded and evaluated by the (deep) neural network, returning the value prediction, and distribution over all possible moves. This value prediction is backpropagated through the tree for every node that has been visited along its path and the process restarts. No random rollouts are used and all predictions are provided by a single shared neural network.

Specifically, a node is selected at each step t by taking $a_t = \text{argmax}_a(\text{Q}(s_t, a) + U(s_t, a))$. The Q-values $\text{Q}(s_t, a)$ of state s_t for every available action a are calculated by a simple moving average (SMA) over all node evaluations and terminal visits of their respective sub-trees. Each newly acquired state evaluation is multiplied by -1 after every step along its back-propagated visited search path. The U-values are based on a combination of the current number of node visits and the predicted probability distribution $P(s, a)$ by the neural network: $U(s, a) = c_{\text{puct}} P(s, a) \sqrt{\sum_b N_r(s, b)} / (1 + N(s, a))$. We choose $c_{\text{puct-init}} = 2.5$ as our U-Value weighting constant, also called exploration constant, which is scaled over the search by

$$c_{\text{puct}}(s) = \log \frac{\sum_a N(s, a) + c_{\text{puct-base}} + 1}{c_{\text{puct-base}}} + c_{\text{puct-init}} \quad (3)$$

with a $c_{\text{puct-base}}$ of 19652. We apply dirichlet noise with a factor of 25% to the probability distribution $P(s, a)$ of the root node with α of 0.2 to encourage exploration. To avoid that all threads traverse the same path during a concurrent rollout we use a virtual loss of 3 which temporarily reduces the chance of visiting the same current node. We initialized the Q-values to -1 for all newly unvisited nodes. This value caused a lot of misunderstandings and confusion because in the initial *AlphaZero* papers the network was described to return a value range output of $[-1, +1]$, but the initialization of Q-Values to be zero assumed the values to be in the range of $[0, 1]$. When treating unvisited nodes as draws, it often led to bad search artifacts and over-exploration, especially when explored positions have negative values because unvisited and low visited nodes have a higher value compared to more visited nodes. As an intermediate solution when initializing unvisited nodes as draws, we introduced a pruning technique in which nodes are clipped for the rest of the search that (1) did not return a better value prediction than their parent node and (2) have a prior policy visit percentage of below 0.1 %. This pruning was used in the matches with JannLee (section 11.1)

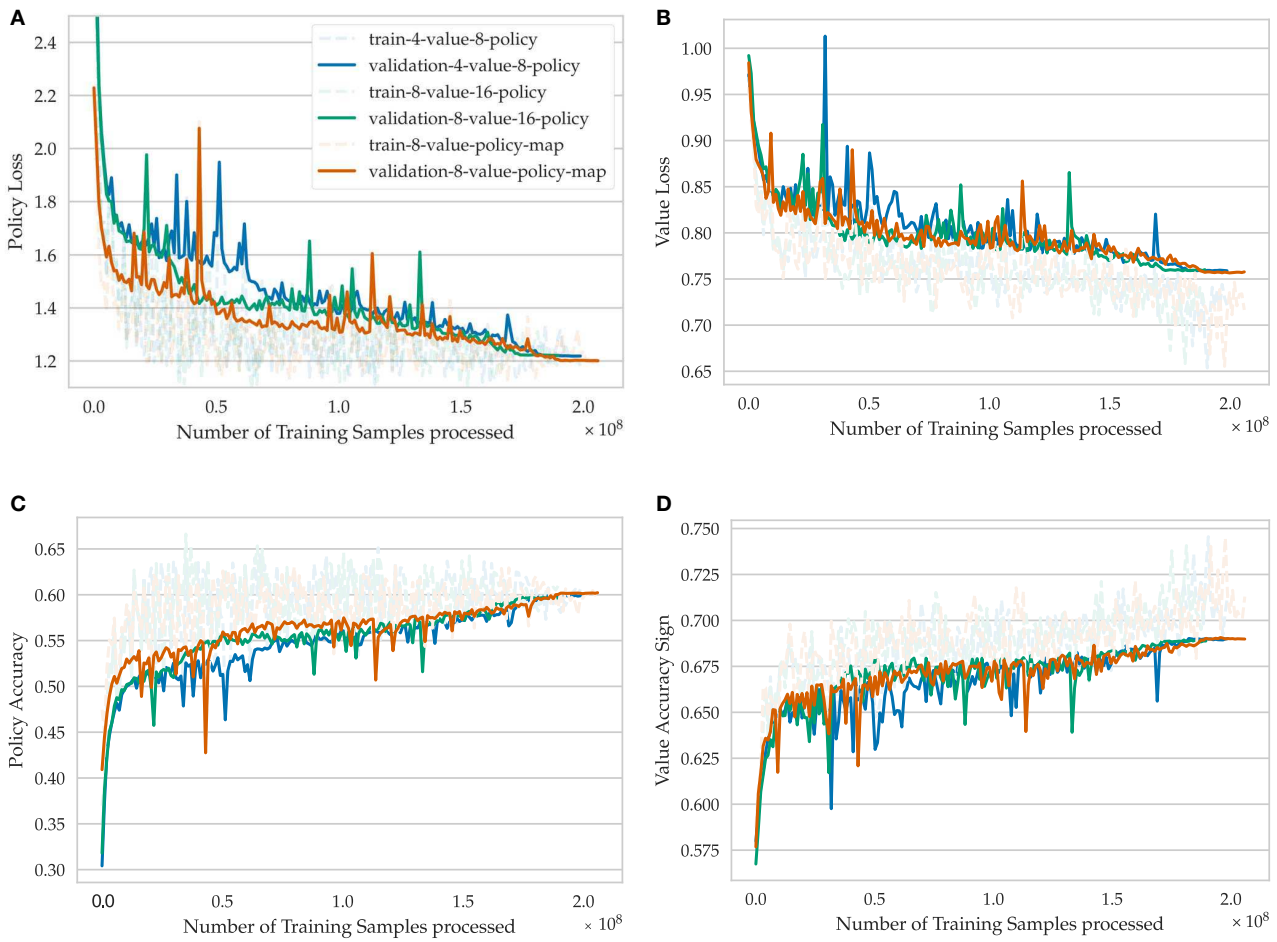

FIGURE 6 | Learning progress of training for seven epochs on the lichess.org crazyhouse data set for different model architectures. *4-value-8-policy* means that four value channels and eight policy channels are used in the respective network heads. *8-value-policy-map* means that eight value channels are used in the value head and the policy is encoded in a direct move to square mapping. **(A)** Policy loss, **(B)** value loss, **(C)** policy accuracy, **(D)** value accuracy sign.

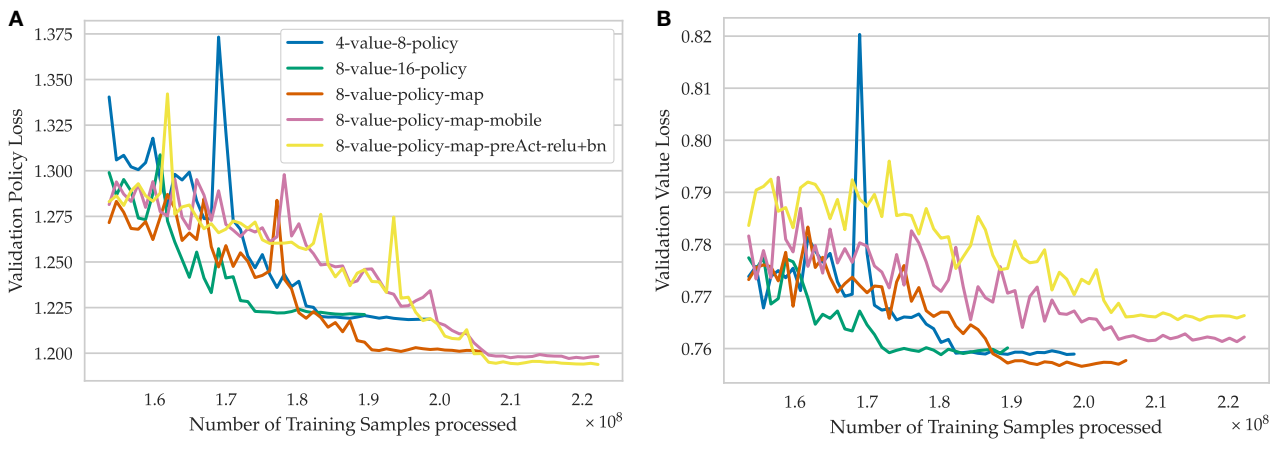

FIGURE 7 | Detailed view on the last iterations of training for different model architectures. **(A)** Validation policy loss, **(B)** Validation value loss.

TABLE 3 | Performance metrics for different models on the lichess.org crazyhouse validation set.

Evaluation metrics on the validation set	4-value-8-policy	8-value-16-policy	8-value-policy-map	8-value- policy-map-mobile	8-value- policy-map-preAct -relu+bn
Combined loss	1.2138	1.2166	1.1964	1.1925	**1.1896**
Policy loss	1.2184	1.2212	1.2008	1.1968	**1.1938**
Value loss	0.7596	0.7601	**0.7577**	0.7619	0.7663
Policy accuracy	0.5986	0.5965	0.6023	0.6032	**0.6042**
Value accuracy sign	0.6894	0.6888	**0.6899**	0.6889	0.6868
Mate-in-one-accuracy	0.954	0.942	**0.955**	0.945	**0.955**
Mate-in-top-5-accuracy	**0.999**	0.998	**0.999**	0.998	**0.999**
Mate-in-one-value-loss	0.0532	**0.0474**	0.0560	0.0567	0.0624

Bold entries indicate the best metric values: lowest value for loss and highest for accuracy.

and caused a major problem in long time control settings: a key move, which would have defended a certain mate threat, has been clipped.

9.2. Changes to Monte-Carlo Tree Search

However, to really master the game of crazyhouse at the level of a world champion, we also had to modify standard MTCS in several ways that we now discuss.

9.2.1. Integration of Q-Values for Final Move Selection

For selecting the final move after search, the vanilla version uses an equation only based on the number of visits for each direct child node

$$\pi(a|s_0) = \frac{N(s_0,a)^{\frac{1}{\tau}}}{\sum_b N(s_0,b)^{\frac{1}{\tau}}}, \tag{4}$$

where τ is a temperature parameter which controls the exploration. In tournament play against other engines, τ is set to 0 which results in choosing the move with the highest number of visits.

We investigated taking information about the Q-values into account, which do not require additional search time and are updated for every rollout. Using Q-values for move selection is motivated by the fact that the most frequently visited node is not necessarily the best one, but the quality for each move is in principle described by its Q-value. Usually there is a strong correlation between the most visited move and the move with the highest Q-value. However, in cases when a strong counter reply was only discovered late during search, the Q-value converges quickly, but the respective node still remains at most visits for several samples.

Silver et al. (2018) also acknowledged that deviating from the most visited node for move selection can yield better results: *"When we forced AlphaZero to play with greater diversity (by softmax sampling with a temperature of 10.0 among moves for which the value was no more than 1% away from the best move for the first 30 plies) the winning rate increased from 5.8 to 14%."* When naively picking the node with the highest Q-value or directly combining Q-values with number of visits, we encounter the problem that Q-values of nodes with low visit counts can be poorly calibrated and can exhibit an overestimation of its actual value. Therefore, we apply the following procedure. First, we re-scale the Q-values into the range $[0,1]$ and set all Q-values with a visit count $<$ $\mathrm{Q}_{\mathrm{thresh}}\max_a(N(s_0,a))$ to 0. We set $\mathrm{Q}_{\mathrm{thresh}}$ to 0.33 and denote these updated Q-values as Q'(s_0,a).

The Q-values are then integrated as a linear combination of visits and Q-values:

$$\pi(a|s_0) = (1-\mathrm{Q}_{\mathrm{factor}})\frac{N(s_0,a)}{\sum_b N(s_0,b)} + \mathrm{Q}_{\mathrm{factor}}\,\mathrm{Q'}(s_0,a). \tag{5}$$

9.2.2. Q-Values With Principal Variation

The Q-values can be further adjusted by updating them taking the information of the Principal Variation (PV) for each move candidate into account:

$$\mathrm{Q}(s_0,a) = \min(\mathrm{Q}(s_0,a), \mathrm{Q}(s_t,x)), \tag{6}$$

where $t = 5$ and x is the selected move at each depth along the rollout. The PV-line describes the suggested optimal line of play for both players and is constructed by iteratively choosing the next move according to Equation (5) until s_t or a terminal node s_T has been reached. As can be seen in **Figure 8** and **Table 4**, the relative increase in Elo[23] is more drastic compared to the scalability of *AlphaZero* for classical chess and more similar to the Elo increase of *AlphaZero* for shogi (Silver et al., 2017a). We think this is due to the lower chances for draws and higher move complexity of crazyhouse which statistically increases the chance to blunder. This makes crazyhouse an excellent testing environment for both MCTS and Alpha Beta Search because effects of changes in the search are reinforced while the number of draws and average game length is vastly reduced.

Move selection which also makes use of Q-values outperformed the vanilla version for a node count $<$ 1600. To improve its behavior also for higher node counts, we applied

[23]Matches are available in **`Data Sheet 2.ZIP`**. The starting opening positions are available at: https://sites.google.com/site/zhassociation/download: `ccva-50start.pgn` (accessed July 30, 2019).

a dynamic adaption of Q_{thresh}, similar to Equation (3), and keep Q_{factor} fixed at 0.7:

$$Q_{\text{thresh}}(s) = Q_{\text{thresh-max}} - \exp\left(-\frac{\sum_a N(s,a)}{Q_{\text{thresh-base}}}\right)(Q_{\text{thresh-max}} - Q_{\text{thresh-init}}), \quad (7)$$

where $Q_{\text{thresh-init}}$ is 0.5, $Q_{\text{thresh-max}}$ is 0.9, and $Q_{\text{thresh-base}}$ is 1965.

9.2.3. Centi-Pawn Conversion

To achieve a better comparability with other crazyhouse engines, we convert our final Q-value of the suggested move after search to the centi-pawn (cp) metric with

$$\text{cp} = -\frac{v}{|v|} \cdot \log \frac{1-|v|}{\log \lambda}, \quad (8)$$

where λ is 1.2.

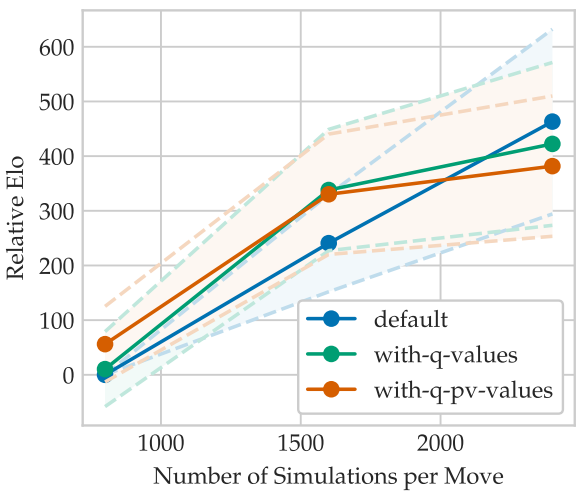

FIGURE 8 | Self Elo increase with respect to nodes.

9.2.4. Time Dependent Search

We also integrate a basic time management system, which does not search moves based on a fixed number of nodes, but on a variable move time t_{current}. This makes the engine applicable on different hardware and better suited for playing other engines. There are two main time settings for engine games. In basic one, a predefined constant time is assigned for a given number of moves (e.g., 40). In sudden death mode a total game time is given to both players for all moves. Optionally, the time can be increased by a certain amount on each move, also called increment, in order to reduce the risk of losing on time in long games.

Our time management uses in principle a fixed move time, which depends on the expected game length, remaining time, and increment. We allocate a constant move time and add 70% of the increment time per move. For sudden death games, we assume a game length of 50 and switch to proportional based system at move 40. In the proportional system, we allocate 5% of the available move time and therefore assume that the game will last 20 moves longer. This formula models the empirical distribution of the expected number of moves to the end of the game as presented by Vučković and Šolak (2009).

Moreover, we stop the search immediately if there is only a single legal move available (e.g., the king is in check with only one escape square) or prematurely at half the allocated time for *easier* positions in order to save time for the rest of the game. We consider a position to be *easy* if the first candidate move of the prior policy has a likelihood >90% and remains the move with the highest Q-value. We extend the search-time by half of the additional preassigned time for *critical* positions. A position is considered *critical* if the Q-value of the current candidate move is 0.1 smaller than the Q-value of the last played move of the previous state. The concept of prematurely stopping MCTS search for the game of Go has been investigated by Baier and Winands (2016).

Last, for games with human players, we also adjust the allocated time per move

$$t_{\text{current}} = t_{\text{current}} + t_{\text{factor}} t_{\text{current}}, \quad (9)$$

where $t_{\text{factor}} \sim [-0.1, +0.1]$ to increase playing variety.

TABLE 4 | Match results of different MCTS move selection types playing each setting against 800 simulations per move using only the most visited node.

Type	Simulations	+	–	=	Elo difference
Default (visits only)	800	–	–	–	0
Default (visits only)	1600	80	20	0	240.82 ± 88.88
Default (visits only)	2400	93	6	1	**463.16** ± 168.96
Visits & *q*-values	800	51	48	1	10.43 ± 68.57
Visits & *q*-values	1600	87	12	1	**338.04** ± 110.92
Visits & *q*-values	2400	91	8	0	422.38 ± 148.92
Visits & q-pv-values	800	57	41	2	**56.07** ± 69.14
Visits & q-pv-values	1600	87	13	0	330.23 ± 110.06
Visits & q-pv-values	2400	90	10	0	381.70 ± 128.31

Matches were generated with CrazyAra 0.4.1 using model 4-value-8-policy. *Games start from 50 unique CCVA opening positions with a temperature value of zero for all settings. Bold entries indicate the best setting that produced the highest Elo difference for a fixed number of simulations.*

9.2.5. Transposition Table

Furthermore, we introduce a transposition table, which stores a pointer to all unique nodes in the tree. Since the number of nodes is several magnitudes lower than for alpha beta engines, its memory size is negligible. Transposition tables are used in most modern chess engines as a look-up table to store and reuse the value evaluation for already explored nodes. In the case of MCTS, we can reuse already existing policy prediction vectors as well as value evaluations. Additionally, one might copy the move generation instead of recomputing it. Transpositions occur frequently during search in chess games and can increase the evaluated nodes per second by a factor of two or more on certain position. Because our input representation depends on the movement counter as well as the no-progress counter, we only take transpositions into account where these counters share the same value. f the node being in the set of its parent nodes could even result in a better performance.

9.2.6. *U*-Value Exploration Factor

We also notice that for certain positions, if a node was found with a Q-value $\gg 0$, then the node exploration of unvisited nodes is sharply reduced. This is because all nodes are initialized with a Q-value of -1 and represent losing positions. We make the division factor for calculating the U-values parameterizable:

$$U(s,a) = c_{\text{puct}} P(s,a) \frac{\sqrt{\sum_b N(s,b)}}{u_{\text{divisor}} + N(s,a)}. \tag{10}$$

This increases the chance of exploring unvisited nodes at least once, according to the principle

> *"When you see a good move, look for a better one"*—Emanuel Lasker.

A $u_{\text{divisor}} < 1$ increases the need of exploring unvisited nodes and can help to reduce the chance of missing key moves, but comes at the cost of losing search depth. To avoid over-exploration at nodes with a low visits count, we reduce u_{divisor} over time, similar to Equation (11):

$$u_{\text{divisor}}(s) = u_{\text{min}} - \exp\left(-\frac{\sum_a N(s,a)}{u_{\text{base}}}\right)(u_{\text{min}} - u_{\text{init}}), \tag{11}$$

where u_{min} is 0.25, u_{init} is 1, and u_{base} is 1965.

9.2.7. Integration of Domain Knowledge

Checks are usually important moves in crazyhouse and it can have detrimental effects if these are missed during search. To ensure that checks have been sufficiently explored, we add the option to enhance the prior probability for all checking moves $P_{check'}(s,a) < check_{\text{tresh}}$ by

$$P_{check'}(s,a) = P_{check'}(s,a) + check_{\text{factor}} \max_a(P(s,a)), \tag{12}$$

where we set $check_{\text{tresh}}$ to 0.1, $check_{\text{factor}}$ to 0.5, and renormalize the distribution afterwards. This modification has the following motivations: the preassigned order for checking moves should be preserved, but checking moves with a low probability are preferred over low confidence non-checking moves. The top-candidate non-checking moves should remain as the move with the highest prior probability.

Including this step might not be beneficial as soon as our network f_θ reaches a certain level of play, but it provides guarantees and was found to greatly increase the efficiency in positions where a forced mate is possible or in which the opponent is threatening a forced mating sequence. Further, we disable any exploration for a particular node as soon as a move was found which leads to a winning terminal node. A direct checkmate is a case which is known to be the best move for all available moves, so additional exploration is unneeded an can only distort the value evaluation.

10. DISCUSSION

Before moving on to our empirical evaluation, let us discuss the pros and cons of the techniques used in *CrazyAra* compared to alternatives as well as provide an illustrative example for our MCTS approach.

10.1. The Pros and Cons of MCTS for Crazyhouse

As mentioned, alpha-beta engines are strongest at open tactical positions. This holds particularly true for finding forced sequences such as a long series of checks. For example, it is common for *Stockfish* to find a forced mate of 20 or up to 40 half-moves in under 5 s in crazyhouse games given sufficient computing power.

In contrast, MCTS shows the opposite behavior and shares similar strength and weakness when compared to human play. It exhibits a significantly lower number of node evaluation and is generally inferior in solving tactical positions quickly if the tactic does not follow its current pattern recognition. On the other hand, it is often better at executing long term strategies and sacrifices because its search is guided by a non-linear policy and is able to explore promising paths more deeply. Alpha-beta engines commonly purely rely on handcrafted linear value evaluations and there is no flow of information in a proposed principal evaluation. The non-linear, more costly value evaluation function can also allow it to vary between small nuances in similar positions. Additionally, (deep) neural networks are able to implicitly build an opening book based on supervised training data or self-play, whereas traditional alpha-beta engines need to search a repeating position from scratch or have to store evaluations in a look-up table which is linearly increasing in size.

In crazyhouse the importance of tactics is increased compared to classical chess and generally when a certain tactic has been missed, the game is essentially decided in engine vs. engine games. Stronger strategic skills result in long grinding games, building up a small advantage move by move, and usually take longer to execute.

MCTS is naturally parallelizable and can make use of every single node evaluation while minimax-search needs to explore a full depth to update its evaluation and is harder to

parallelize. Also the computational effort increases exponentially for each depth in minimax-search. Besides that, MCTS returns a distribution over all legal moves which can be used for sampling like in reinforcement learning. As a downside, this version of MCTS highly depends on the probability distribution $a \sim P(s, a)$ of the network f_θ which can have certain blind spots of missing critical moves. Minimax search with a alpha-beta pruning is known to have the problem of the horizon effect where a miss-leading value evaluation is given because certain tactics have not been resolved. To counteract this, minimax-based algorithms employ quiescence search to explore certain moves at greater depth (Kaindl, 1982). For MCTS search taking the average over all future value evaluation for all expansions of a node can be misleading if there is only a single winning line and in the worst case to divergence.

10.2. Exemplary MCTS Search

Figure 9 shows a possible board position in a crazyhouse game in which $P(s, a)$ misses the correct move in its first 10 candidate moves. The white player has a high material advantage, but black is threatening direct mate by `Qxf2#` as well as `Qxb1` followed by `R@h1#`. White has to find the following sequence of moves `24. N@e6!! fxe6! 25. Bxf6! Ke8 26. P@f7!! Kxf7 27. N@g5! Ke8 28. Nxh3!` to defend both mate threats and to keep a high winning advantage. Intermediate captures such as `Rxe4` or `N@c6` or a different move ordering are also losing.

There are 73 moves available and $P(s, a)$ of the used model assigns a low softmax-activation of 2.69e-05 for the move `N@e6` resulting as the 53rd candidate move. **Figure 9** shows the progression of the number of visits and Q-value for the move `N@e6` using our introduced MCTS adaptations. To disable the effect of randomness and fluctuations due to thread scheduling, we make the search fully deterministic for this experiment: we set the number of threads to one with a batch size of eight and replace the Dirichlet noise distribution by a constant distribution which uniformly increases the prior probability for all moves in the root node.

The convergence for the node visits behaves linearly as soon as the move `N@e6` emerged as the most promising move. If the move remains unexplored its corresponding Q-value stays at a value of -1. After the move is visited for only a few samples the event is associated with a spike. These high initial Q-value degrade over time because simulations chose the wrong follow-up lines and consequently back-propagate a miss-leading value evaluation. In cases where the move `N@e6` remains unvisited for a period, the Q-value behaves flat. As soon as all of white only moves have been found in response to different attempts by black, the Q-value quickly recovers and converges to a value of approximately $+0.6$.

When comparing our changes to MCTS, we notice the following: in the *default* MCTS version the move `N@e6` remains unvisited for more than 10000 simulations due to a very low prior probability. In the case when exploration is disabled as soon as a mate in one has been found (*fix-checkmates*) it requires slightly fewer samples. If the prior probability for all checking moves is uniformly increased (*enhance-checks*, section 9.2.7) the convergence is overall the fastest. For the parameterized *u-value-exploration-factor*, the initial visit and convergence of the Q-value is pulled forward. The last setting (*all*) combines all of our pre-mentioned search adaptions. The initial node exploration occurs similarly to *enhance-checks*, and after the Q-value degraded the node remains unexplored for 8000 simulations. However, after the correct lines for white have been found for the first few depths, the Q-value convergences as quickly as *enhance-checks*.

11. EXPERIMENTAL EVALUATION

Our intention here is to evaluate the playing strength of *CrazyAra*. To this end, we let *CrazyAra* play matches against other crazyhouse engines as well as the human player Justin Tan, the 2017 crazyhouse world champion.

11.1. Matches With Human Professional Players

Over the course of its development, *CrazyAra* was hosted several times on lichess.org playing among others the strongest human crazyhouse players from whom it has learnt to play the game. On December 21st 2018 at 18:00 GMT *CrazyAra 0.3.1* played multiple world crazyhouse champion Justin Tan, also known as *LM JannLee* and won four of five informal games[24]. The match has been streamed live and commented by Justin Tan. in the time control of 5 min + 15 s. For the settings, we used a temperature value τ of 0.07 for the first four moves, thinking during opponents turn called "Ponder" was disabled and we achieved 250 nodes per second (NPS). Our recommended changes to MCTS in sections 9.2.6, 9.2.7 had not been integrated in this version.

11.2. Strength Evaluation With Other Crazyhouse Engines

We also evaluate the playing strength of *CrazyAra 0.6.0* using the network architecture *8-value-policy-map-mobile* (see **Table 2**) on an Intel® Core™ i5-8250U CPU @ 1.60GHz × 8, Ubuntu 18.04.2 LTS against all participants of the second CCVA Computer Championships (Mosca, 2017) or their respective updated version.

All engines including *CrazyAra* were run on the same hardware in a time control of 15 min + 5 s per move. The number of threads was set to 8 and the hash size, which allows storing board evaluations, to 1024 mb, if the engine provides an option for it. If only a Windows executable is available for an engine, we made it compatible with the help of Wine (Julliard, 1994), which reduces the original NPS of an engine by about 10 %. We refer to the term NPS as the number of position evaluations per second and in the context of MCTS as the number of MCTS simulations per second. "Ponder" was turned off and we also allowed other engines to use opening books or position learning. As the deep learning framework for *CrazyAra*, we used MXNet 1.4.1 (Chen et al., 2015) with the Intel-MKL backend (Wang et al., 2014) and also enabled the network subgraph optimization by Intel. Arithmetic, vectorized operations in the MCTS search were formulated in the Blaze 3.6 library (Iglberger

[24] All games can be found in the **Supplementary Material** (section 1.1, p. 1). The match has been streamed live and commented by Justin Tan.

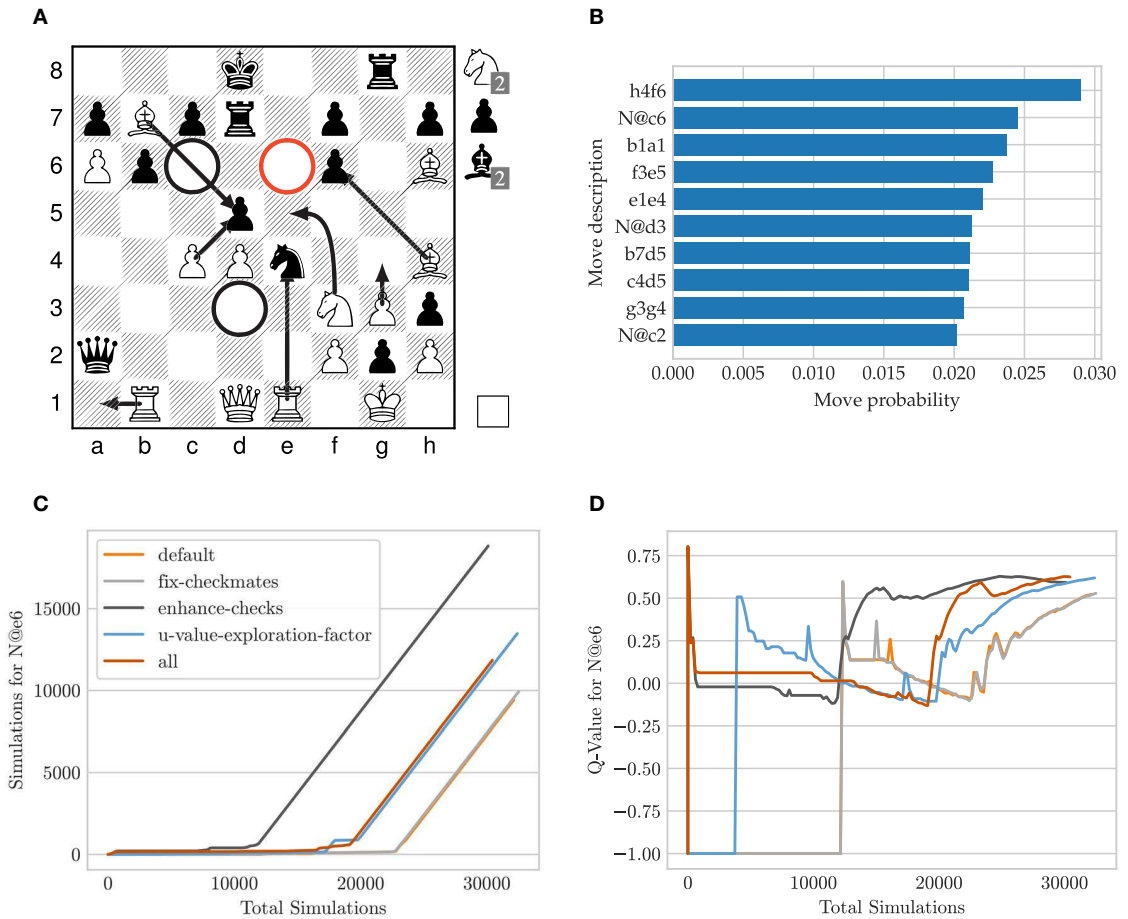

FIGURE 9 | Search tree statistics over time for the Move N@e6 using different MCTS modifications[25]. **(A)** Exemplary tactical board position, **(B)** sorted policy distribution of the first 10 candidate move, **(C)** simulation progression, **(D)** *Q*-value progression.

et al., 2012a,b) and move generation routines were integrated from *multi-variant Stockfish*[26].

To get a reference point, DeepMind's *AlphaZero* "*used 44 processor cores and four of Google's first-generation TPUs generating about 60000 positions per second in chess compared to Stockfish's roughly 60 million*" yielding a ratio of 1:1000. *AlphaZero* started to outperform the search efficiency of *Stockfish 8* in their setup after 30000 node evaluations (Silver et al., 2017a). We achieved 330 NPS for *CrazyAra 0.6.0* on the aforementioned CPU, resulting in approximate 7260 nodes per move in this time control compared to an average NPS between 1 million nodes for most alpha-beta engines and 4.6 million for multi-variant *Stockfish 10*, resulting in a ratio between 1:3000 and 1:14000. Despite this large gap in number of evaluated nodes, *CrazyAra* often achieved a higher depth compared to other engines. This is partly due to the forcing nature of crazyhouse in which a majority of the moves are losing outright and filtered out by the neural network. However, we use the term depth for *CrazyAra* as the length of the explored principal variation after the search, and alpha-beta engines usually explore the full tree or much larger parts of it. The depth and centipawn evaluation of the current board position is denoted as `{<centipawn>/<depth> <time spent>}` after each move where a positive centipawn value describes an advantage in the view of the respective engine.

We played 10 matches with each engine starting from five common crazyhouse opening positions. Each position was played twice, one in which *CrazyAra* played the white and one in which it played the black pieces. The matches were played without a resign threshold and always ended in a mate position or a draw. We enabled all of our proposed changes for the MCTS search (section 9.2) and used a temperature value of zero to make the moves for *CrazyAra* relatively deterministic and to give all other engines the same chances.

The results of the matches (see **Table 5**) demonstrate that *CrazyAra 0.6.0* clearly won against 12 of the 13 participants with either 10 or 9 wins out of 10 matches. All matches with the respective engine evaluations and their depth on each move are

[25]The FEN for this position is: 3k2r1/pBpr1p1p/Pp3p1B/3p4/2PPn2B/5NPp/q4PpP/1R1QR1K1/NNbpw--123

[26]https://github.com/ddugovic/Stockfish (accessed July 30, 2019).

TABLE 5 | Match results of *CrazyAra 0.6.0* on CPU playing twelve different crazyhouse engines.

Engine name	Version	Elo rating	NPS (million)	Wapc	Lapc	+	=	–
CrazyAra	0.6.0	–	0.00033	62 ± 18	74 ± 24	118	0	12
PyChess	1.1	> 1566.25*	0.012	–	41 ± 11	0	0	10
KKFChess†	2.6.7b	1849.50	2.9	–	57 ± 10	0	0	10
TSCP ZH	1.1	1888.47	0.5	–	54 ± 12	0	0	10
Pulsar	2009-9b	1982.07	?	–	61 ± 13	0	0	10
Feuerstein†	0.4.6.1	2205.74	0.1	–	57 ± 8	0	0	10
Nebiyu†	1.45a	2244.39	1.5	–	42 ± 8	0	0	10
SjaakII	1.4.1	2245.56	0.425	–	61 ± 10	0	0	10
Sjeng	11.2	2300.00	0.7	–	66 ± 15	0	0	10
CrazyWa	1.0.1	2500.00	1.4	–	73 ± 10	0	0	10
Sunsetter	9	2703.39	1.5	–	74 ± 19	0	0	10
TjChess	1.37	2732.58	1.37	53 ± 0	85 ± 17	1	0	9
Imortal†	4.3	> 2997.33*	0.9	134 ± 0	77 ± 9	1	0	9
Stockfish	10 (2018-11-29)	> 3946.06*	4.6	70 ± 16	–	10	0	0

Wapc *and* ***Lapc*** *means "win average ply count" and "loss average ply count" and describe the average game length. Engine denoted with† use Wine for emulation.*
**PyChess', rating of rating of 1566.25 corresponds to version 0.12.4, Immortal's rating of 2997.33 corresponds to version 3.04, Stockfish's rating of 3946.06 corresponds to version 2017-09-23 using a single thread instead of eight.*

TABLE 6 | Match results of *CrazyAraFish 0.6.0* playing *Stockfish 10* in a time control of 30 min + 30 s.

Engine name	Version	Elo rating	NPS (million)	Wapc	Lapc	+	=	–
CrazyAraFish	0.6.0	–	0.0014	118 ± 22	98 ± 34	3	1	6
Stockfish	10 (2018-11-29)	> 3,946.06	6.7	98 ± 34	118 ± 22	6	1	3

available in the **Supplementary Material**[27]. However, *CrazyAra* lost all games to *Stockfish*. Despite this, it was able to generate +2.62, +6.13, +6.44 centipawn positions in three separate games as white according to the evaluation of *Stockfish*. To reduce the effect of the opening advantage for the first player in crazyhouse and also the fact that *CrazyAra* can make use of an implicitly built opening-book, we choose five opening positions for the final evaluation, which are more balanced and less popular in human play.

The games between *CrazyAraFish 0.6.0* and *Stockfish 10* were generated on an AMD® Ryzen 7, 1700 eight-core processor ×-16 for both engines and also a GTX1080ti for *CrazyAraFish 0.6.0*. *Stockfish* achieves 6.7 million NPS on our setup. The hash size for *Stockfish* was set to 1,024 mb[28]. We used a batch size of eight and two threads for traversing the search tree in the case of *CrazyAraFish*. In this setting, *CrazyAraFish 0.6.0* achieved a NPS of 1400 resulting in a node ratio of about 1:4700. In positions where many transpositions and terminal nodes were visited, the NPS increased to 4000. The matches were played in a long time control format of 30 min + 30 s. Here, *CrazyAraFish 0.6.0* won three games and drew one game out of 10 games (see **Table 6**).

[27]**`Data Sheet 1.ZIP`**; for more information about the engines and their corresponding authors please refer to the *Crazyhouse Alpha List* maintained by Simon Guenther http://rwbc-chess.de/chronology.htm (accessed July 7, 2019).

[28]We also tried choosing a higher hash size for *Stockfish* e.g., 4,096 mb, but found it to be unstable resulting in game crashes for the *Stockfish* executable *x86_64-modern 2018-11-29*. These crashes have been reported to the corresponding developers.

The better performance of *Stockfish* in this evaluation can be attributed to multiple factors. First, *CrazyAraFish* had a higher chance of missing important lines during search due to a lower amount of position evaluations. Second, the data set generated by *Stockfish* contained five times fewer games compared to the lichess data set and could benefit from additional measures to counteract the deterministic playing behavior of alpha-beta engines. Third, the value evaluation was sub-optimal for certain positions partially because the value loss is only weighted by 1% in the combined loss during training in order to avoid overfitting. All games can be found in **Figures S8–S17**, p. 8–17.

12. CONCLUSION

In this work we have developed a crazyhouse chess program, called *CrazyAra*, based on a combination of deep neural networks and tree search, that plays at the level of the strongest human players. Despite the highly tactical game-style of crazyhouse and a relatively small and low quality data set, it is possible to achieve a remarkable performance when only using supervised learning. We demonstrated that MCTS is fairly successful at low samples, when powered by a (deep) neural network, and is able to drastically increase the quality of moves over time for the crazyhouse variant. Most importantly, we demonstrated that the scheme proposed by Silver et al. (2017a) can be and has to be improved and accelerated in different areas for crazyhouse: this includes achieving a better training performance

and the prediction of higher quality moves with the same number of MCTS simulations. Indeed, several optimizations are achievable in future work. A faster generation of rollouts e.g., by using low precision inference like float16 or int8 and potential future improvements in network design and MCTS will boost performance. Additionally, applying reinforcement learning can help to increase the playing strength of *CrazyAra* further.

DATA AVAILABILITY STATEMENT

All games of the presented results in this article as well as supplementary figures and tables are included in the article/**Supplementary Material**. The supervised training dataset is freely available at: https://database.lichess.org/ and the generated Stockfish self-play dataset can be found at: https://github.com/QueensGambit/CrazyAra/wiki/Stockfish-10:-Crazyhouse-Self-Play. All data preprocessing scripts can be accessed at: https://github.com/QueensGambit/CrazyAra.

AUTHOR CONTRIBUTIONS

The project was initially started by JC, MW, and AB as part of the course Deep Learning: Architectures & Methods held by KK and JF in summer 2018. This paper was mainly written by JC with the continuous support and revision of KK and JF.

ACKNOWLEDGMENTS

The authors thank the main *Stockfish* developers of crazyhouse, Fabian Fichter, Daniel Dugovic, Niklas Fiekas for valuable discussions. They also thank other crazyhouse-engine programmers including Bajusz Tamás, Harm Geert Muller, and Ferdinand Mosca for providing their latest chess engine executable. The authors are grateful to the lichess.org crazyhouse community *LM JannLee* (Justin Tan), *IM gsvc*, *TheFinnisher*, *IM opperwezen* (IM Vincent Rothuis), *FM WinnerOleg* (FM Oleg Papayan), and *okei* among others for playing *CrazyAra* on lichess.org and providing valuable feedback. They thank github users *@noelben* for the help in creating a time-management system and *@Matuiss2* for improving the coding style and frequently testing the engine. They appreciate the constructive feedback from Karl Stelzner when writing the paper. In particular, the authors are thankful to the users *crazyhorse*, *ObiWanBenoni*, *Pichau*, and *varvarakh* for helping in generating the *Stockfish* self-play data set. Finally, the authors thank Thibault Duplessis and the lichess.org developer team for creating and maintaining lichess.org, providing a BOT API and the lichess.org database.

SUPPLEMENTARY MATERIAL

The Supplementary Material for this article can be found online at: https://www.frontiersin.org/articles/10.3389/frai.2020.00024/full#supplementary-material

REFERENCES

Baier, H., and Winands, M. (2016). Time management for Monte Carlo tree search. *IEEE Trans. Comput. Intell. AI Games* 8, 301–314. doi: 10.1109/TCIAIG.2015.2443123

Baier, H., and Winands, M. H. M. (2015). MCTS-Minimax hybrids. *IEEE Trans. Comput. Intell. AI Games* 7, 167–179. doi: 10.1109/TCIAIG.2014.2366555

Baxter, J., Tridgell, A., and Weaver, L. (2000). Learning to play chess using temporal differences. *Mach. Learn.* 40, 243–263. doi: 10.1023/A:1007634325138

Botev, A., Lever, G., and Barber, D. (2017). "Nesterov's accelerated gradient and momentum as approximations to regularised update descent," in *2017 International Joint Conference on Neural Networks (IJCNN)* (Anchorage, AK), 1899–1903. doi: 10.1109/IJCNN.2017.7966082

Browne, C. B., Powley, E., Whitehouse, D., Lucas, S. M., Cowling, P. I., Rohlfshagen, P., et al. (2012). A survey of Monte Carlo tree search methods. *Trans. Comput. Intell. AI Games* 4, 1–43. doi: 10.1109/TCIAIG.2012.2186810

Campbell, M., Hoane, A. J., and Hsu, F.-H. (2002). Deep blue. *Artif. Intell.* 134, 57–83. doi: 10.1016/S0004-3702(01)00129-1

Chen, T., Li, M., Li, Y., Lin, M., Wang, N., Wang, M., et al. (2015). MxNet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv [Preprint]. arXiv:1512.01274*. Available online at: https://www.cs.cmu.edu/~muli/#pub

Chi, G. (2018). *A Deep Learning Crazyhouse Chess Program That Uses a Monte Carlo Tree Search (MCTS) Based Evaluation System and Reinforcement to Enhance Its Play Style: FTdiscovery/64crazyhousedeeplearning*, Stanford University, California, United States.

Dong, X., Kang, G., Zhan, K., and Yang, Y. (2017). EraseReLU: a simple way to ease the training of deep convolution neural networks. *arXiv [Preprint]. arXiv:1709.07634*.

Droste, S., and Fürnkranz, J. (2008). Learning the piece values for three chess variants. *ICGA J.* 31, 209–233. doi: 10.3233/ICG-2008-31403

Fichter, F. (2018). *SPSA Tuner for Multi-Variant Fork of Stockfish Chess Engine: ianfab/spsa*. Available online at: https://github.com/ianfab/spsa (accessed July 21, 2019).

Free Software Foundation (2017). Available online at: https://www.gnu.org/licenses/gpl-3.0.en.html (accessed July 30, 2019).

Fürnkranz, J. (1996). Machine learning in computer chess: the next generation. *Int. Comput. Chess Assoc. J.* 19, 147–161. doi: 10.3233/ICG-1996-19302

Fürnkranz, J. (2017). "Machine learning and game playing," in *Encyclopedia of Machine Learning and Data Mining*, eds C. Sammut and G. I. Webb (Boston, MA: Springer), 783–788. doi: 10.1007/978-1-4899-7687-1_509

Gelly, S., Schoenauer, M., Sebag, M., Teytaud, O., Kocsis, L., Silver, D., et al. (2012). The grand challenge of computer GO: Monte Carlo tree search and extensions. *Commun. ACM* 55, 106–113. doi: 10.1145/2093548.2093574

Han, D., Kim, J., and Kim, J. (2017). "Deep pyramidal residual networks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (Honolulu, HI), 5927–5935. doi: 10.1109/CVPR.2017.668

He, K., Zhang, X., Ren, S., and Sun, J. (2016a). "Deep residual learning for image recognition," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (Las Vegas, NV), 770–778. doi: 10.1109/CVPR.2016.90

He, K., Zhang, X., Ren, S., and Sun, J. (2016b). "Identity mappings in deep residual networks," in *European Conference on Computer Vision* (Amsterdam: Springer), 630–645. doi: 10.1007/978-3-319-46493-0_38

Howard, A. G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., et al. (2017). Mobilenets: efficient convolutional neural networks for mobile vision applications. *arXiv [Preprint]. arXiv:1704.04861*.

Hu, J., Shen, L., and Sun, G. (2018). "Squeeze-and-excitation networks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (Salt Lake City, UT), 7132–7141. doi: 10.1109/CVPR.2018.00745

Iglberger, K., Hager, G., Treibig, J., and Rüde, U. (2012a). Expression templates revisited: a performance analysis of current methodologies. *SIAM J. Sci. Comput.* 34, C42-C69. doi: 10.1137/110830125

Iglberger, K., Hager, G., Treibig, J., and Rüde, U. (2012b). "High performance smart expression template math libraries," in *2012 International Conference on High Performance Computing & Simulation (HPCS)* (Madrid), 367–373. doi: 10.1109/HPCSim.2012.6266939

Julliard, A. (1994). *WineHQ - Run Windows Applications on Linux, BSD, Solaris and macOS*. Available online at: https://www.winehq.org/ (accessed June 3, 2019).

Kahlen, S.-M., and Muller, G. H. (2004). *UCI Protocol*. Available online at: http://wbec-ridderkerk.nl/html/UCIProtocol.html (accessed June 05, 2019).

Kaindl, H. (1982). Quiescence search in computer chess. *SIGART Newslett*. 80, 124–131.

Keskar, N. S., and Socher, R. (2017). Improving generalization performance by switching from Adam to SGD. *arXiv [Preprint]. arXiv:1712.07628*.

Kiiski, J. (2014). *SPSA Tuner for Stockfish Chess Engine*. Available online at: https://github.com/zamar/spsa (accessed June 3, 2019).

Kingma, D. P., and Ba, J. (2015). "Adam: A method for stochastic optimization," in *Proceedings of the 3rd International Conference on Learning Representations (ICLR)* (San Diego, CA).

Kocsis, L., and Szepesvári, C. (2006). "Bandit based Monte-Carlo planning," in *Proceedings of the 17th European Conference on Machine Learning (ECML)* (Berlin: Springer), 282–293. doi: 10.1007/11871842_29

Ma, N., Zhang, X., Zheng, H.-T., and Sun, J. (2018). "Shufflenet v2: Practical guidelines for efficient cnn architecture design," in *Proceedings of the European Conference on Computer Vision (ECCV)* (Munich), 116–131. doi: 10.1007/978-3-030-01264-9_8

Mishkin, D., Sergievskiy, N., and Matas, J. (2017). Systematic evaluation of convolution neural network advances on the Imagenet. *Comput. Vis. Image Understand.* 161, 11–19. doi: 10.1016/j.cviu.2017.05.007

Mosca, F. (2017). *2nd CCVA Computer Championships - Crazyhouse Chess Variant Association*. Available online at: https://sites.google.com/site/zhassociation/computers/tournaments/2nd-ccva-computer-championships (accessed June 5, 2019).

Ramanujan, R., Sabharwal, A., and Selman, B. (2010). "On adversarial search spaces and sampling-based planning," in *Proceedings of the 20th International Conference on Automated Planning and Scheduling (ICAPS)*, eds R. I. Brafman, H. Geffner, J. Hoffmann, and H. A. Kautz (Toronto, ON: AAAI), 242–245.

Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., et al. (2015). Imagenet large scale visual recognition challenge. *Int. J. Comput. Vis.* 115, 211–252. doi: 10.1007/s11263-015-0816-y

Samuel, A. L. (1959). Some studies in machine learning using the game of checkers. *IBM J. Res. Dev.* 3, 210–229. doi: 10.1147/rd.33.0210

Sandler, M., Howard, A., Zhu, M., Zhmoginov, A., and Chen, L.-C. (2018). "Mobilenetv2: Inverted residuals and linear bottlenecks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (Salt Lake City, UT), 4510–4520. doi: 10.1109/CVPR.2018.00474

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., et al. (2016). Mastering the game of GO with deep neural networks and tree search. *Nature* 529, 484–489. doi: 10.1038/nature16961

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

Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., et al. (2018). A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science* 362, 1140–1144. doi: 10.1126/science.aar6404

Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., et al. (2017b). Mastering the game of Go without human knowledge. *Nature* 550, 354–359. doi: 10.1038/nature24270

Skiena, S. S. (1986). An overview of machine learning in computer chess. *Int. Comput. Chess Assoc. J.* 9, 20–28. doi: 10.3233/ICG-1986-9103

Smith, L. N. (2018). A disciplined approach to neural network hyper-parameters: part 1-learning rate, batch size, momentum, and weight decay. *arXiv [Preprint]. arXiv:1803.09820*.

Smith, L. N., and Topin, N. (2019). "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* (Baltimore, MA: International Society for Optics and Photonics), 1100612. doi: 10.1117/12.2520589

Szegedy, C., Ioffe, S., Vanhoucke, V., and Alemi, A. A. (2017). "Inception-v4, inception-resnet and the impact of residual connections on learning," in *Proceedings of the 31st AAAI Conference on Artificial Intelligence* (San Francisco, CA).

Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., and Wojna, Z. (2016). "Rethinking the inception architecture for computer vision," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (Las Vegas, NV), 2818–2826. doi: 10.1109/CVPR.2016.308

Tesauro, G. (1995). Temporal difference learning and TD-gammon. *Commun. ACM* 38, 58–68. doi: 10.1145/203330.203343

Vučković, V., and Šolak, R. (2009). Time management procedure in computer chess. *Fact Univer. Ser.* 8, 75–87.

Wang, E., Zhang, Q., Shen, B., Zhang, G., Lu, X., Wu, Q., et al. (2014). "Intel math kernel library," in *High-Performance Computing on the Intel®Xeon Phi™* (Cham: Springer), 167–188. doi: 10.1007/978-3-319-06486-4_7

Xie, S., Girshick, R., Dollár, P., Tu, Z., and He, K. (2017). "Aggregated residual transformations for deep neural networks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (Honolulu, HI), 1492–1500. doi: 10.1109/CVPR.2017.634

Zagoruyko, S., and Komodakis, N. (2016). "Wide residual networks," in *Proceedings of the British Machine Vision Conference (BMVC)* (York). doi: 10.5244/C.30.87

Zhao, G., Zhang, Z., Guan, H., Tang, P., and Wang, J. (2017). Rethink ReLU to training better CNNs. *arXiv:1709.06247 [cs]*. doi: 10.1109/ICPR.2018.8545612

Conflict of Interest: The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.