Bug0: Extending the AlphaZero algorithm to a 4 player chess variant.

Devin Ardeshna
Department of Mechanical Engineering
ardeshna@stanford.edu

Robbie Selwyn
Department of Computer Science
rselwyn@stanford.edu

Abhay Singhal
Department of Computer Science
sabhay@stanford.edu

***Abstract*—Chess has been heavily studied in the field of algorithms due to its complex nature, and the ability to easily evaluate strategies against each other. In recent years – in part due to the boom of online chess – chess variants have become increasingly popular. One such variant is Bughouse, a game in which teams of two players work together over two boards to try to win, and are able to pass captured pieces to their partner to be placed anywhere on the other board. This introduces substantial complexity, as a Bughouse engine needs to coordinate chess moves on both boards and coordinate the passing of pieces between boards. In this paper, we attempt to train a Bughouse engine exclusively from self-play, and with no knowledge of game strategy other than the rules. We also apply supervised learning techniques, which have been previously attempted successfully for similar variants, to Bughouse [1].**

I. Introduction

Achieving mastery of board games with computer programs and machine learning has been of particular interest over recent years. Google's DeepMind made headlines in 2016 with its AlphaGo algorithm [1], which beat the top-ranked Go player in the world. AlphaGo's deep neural network was bootstrapped using historical game data and then further trained using Monte Carlo Tree Search (MCTS) and reinforcement learning. DeepMind has since released AlphaZero [2], a more versatile algorithm that can be trained to mastery on not only Go, but other games including Chess and Shogi. Unlike AlphaGo, AlphaZero starts only with basic knowledge of the game and trains purely through self-learning.

In addition to standard Chess, many variants exist. One better-known variant is Crazyhouse, where captured pieces switch color and may be reintroduced onto the board at any time by the capturing player (for example, if a black rook is captured, it turns into a white rook that can be dropped anywhere as a move).

Similarly, Bughouse is a 4-player extension of Crazyhouse which consists of two chess games played in parallel. Here, players work in partnerships – they are both trying to checkmate their opponent's king, but captured pieces from one board can be dropped onto the other board by the partner. Players play with a partner of opposite color (see Fig. I), so they simply hand off their captured pieces to be placed. Because pieces can be dropped anywhere on the board (often near an opponent's king), Bughouse tends to be much more focused on dynamic attacks, rather than slow maneuvering. The game concludes when there is checkmate on either board.

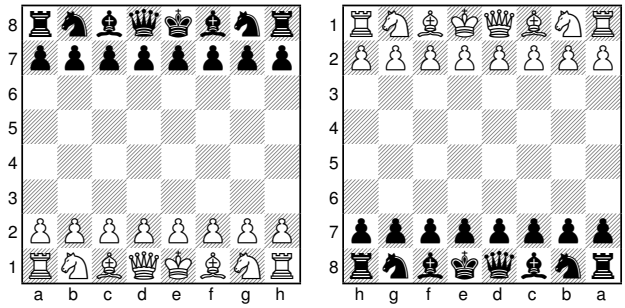

Fig. 1. Starting position of a bughouse game. Team one sits side by side and plays the white side on board one, and the black side on board two.

Our project attempts to extend the strategies of AlphaZero to Bughouse chess. To do so, we have modified the Chess board representation used in [2] to account for the additional board and reintroduction of captured pieces. Due to the complexities arising from game timing, we have also made simplifying assumptions about the order in which turns are taken. Our experimental work includes an implementation of AlphaZero for Bughouse, although we were not able to fully train this implementation through self-play due to resource constraints. In addition, we implemented several supervised learning strategies based on a large Bughouse game archive. Lastly, we also developed a minimax Bughouse engine as a baseline and used this engine to compare to our ML-based engines.

II. Related Work

One of the earliest Chess computer program successes was IBM's Deep Blue [3], which defeated then-reigning World Chess Champion Garry Kasparov in a six-game match. Deep Blue relied on alpha-beta search and many hand-crafted features based on domain expertise. Since Deep Blue, a number of other alpha-beta based chess engines such as Stockfish have emerged and dominated the Chess computer program landscape [4].

Reinforcement learning has also been applied to computer Chess in work prior to AlphaZero. KnightCap used temporal difference learning to train a neural network and achieve mastery level while playing on internet Chess servers [5].

[1]The project code can be found at https://github.com/rselwyn/bug0-cs229

DeepChess achieved grandmaster-level performance without any domain specific knowledge of Chess by training on millions of games [6]. DeepChess used unsupervised learning to extract high-level features about a chess position, followed by supervised learning to select the more favorable of two positions. However, both of these approaches still rely on alpha-beta search algorithms. Most recently, AlphaZero has shown the potential of MCTS + neural networks to yield comparable performance to top alpha-beta engines without any domain-specific knowledge other than the rules [2].

Work on Crazyhouse and Bughouse engines is comparatively sparse. Certain forks of Stockfish such as Fairy-Stockfish contain support for Crazyhouse and Bughouse. Another program, CrazyAra, uses a neural network trained with supervised learning to play Crazyhouse [7]. Its successor, MultiAra, uses AlphaZero-inspired reinforcement learning to play multiple variants of Chess including Crazyhouse [8]. However, there has been little work done to apply these machine learning strategies to Bughouse. It has been suggested this may be due the complex time management tactics that can be employed in Bughouse as opposed to standard Chess.

III. Methods

In this section we describe the representation of Bughouse gameplay and board state that we use with our models. We then discuss the major components of AlphaZero and our implementation of the algorithm for Bughouse. We also briefly detail our other supervised approaches including linear regression, elastic net, decision trees, and random forest. Finally, we cover our primary baseline which is a minimax agent with heuristics borrowed from and existing chess engine.

A. Gameplay and Board Representations

One of the highlights of true Bughouse gameplay is that the two Chess games run asynchronously. In other words, the games may be played at different rates. The Bughouse game ends when any of the four players wins their board, or if both boards result in a draw. This makes Bughouse a particularly fast-paced and exciting variant of Chess, but greatly complicates the action space. In order to make the problem more tractable, we have imposed the following turn-taking order:

Team 1 Board A (White) $\rightarrow$ Team 2 Board B (White) $\rightarrow$
Team 1 Board B (Black) $\rightarrow$ Team 2 Board A (Black)

We represent the Bughouse board state s using a $60 \times 8 \times 8$ stack of planes. Unlike AlphaZero, we do not include board history (which was required for Go). Chess is a full information game, so the history is not needed to determine the best possible move. This reduces the number of model parameters and space required to store datasets. Table I details the board representation.

TABLE I
The Bughouse board state is represented as a $60 \times 8 \times 8$ stack of planes. Since the Bughouse board consists of two adjacent Chess boards, most of the planes are included twice. Board positions are encoded as bitboards, while scalar values are encoded as constant-valued planes.

Bughouse Chess			
	Feature	Planes	Type
x1	Player	1	Constant-Valued
	Active Board	1	Constant-Valued
x2	White Pieces (KQRBNP)	6	Bitboard
	Black Pieces (kqrbnp)	6	Bitboard
	White Prisoners (QRBNP)	5	Constant-Valued
	Black Prisoners (qrbnp)	5	Constant-Valued
	Promoted Pawns	1	Bitboard
	En Passant Square	1	Bitboard
	Fifty Move Counter	1	Constant-Valued
	Castling	4	Constant-Valued
	Total	60	

B. AlphaZero

Unlike many other Chess engines, AlphaZero starts only with knowledge of the game rules and learn through self-play. A deep neural network $(\mathbf{p}, v) = f_\theta(s)$ estimates move probabilities $\mathbf{p}$ and the expected outcome of the game v. Each element p_a of $\mathbf{p}$ corresponds to the move probability for action a. This neural network is used in the rollout step of a Monte Carlo Tree Search (MCTS) to select moves during self-play. The results of the MCTS and self-play are used to further refine the neural network, and the process repeats iteratively.

1) Monte Carlo Tree Search: Monte Carlo Tree Search (MCTS) is used to explore future game states during self-play. Edges of the tree (s, a) represent the possible actions a taken from state s. For each action, the prior probability $P(s, a)$, visit count $N(s, a)$, and average expected value $Q(s, a)$ are stored. Every tree simulation starts at the root node s_0 and then selects actions based on an upper confidence bound that balances exploration of new nodes with exploitation of high expected payoff. AlphaZero uses the following selection rule, where c_{puct} is a hyperparameter to adjust exploration vs exploitation, nominally set to 1:

$$a = \underset{a}{\operatorname{argmax}} \left[Q(s,a) + c_{\text{puct}} P(s,a) \frac{\sqrt{\sum_b N(s,b)}}{1 + N(s,a)} \right]$$

When a leaf node s is reached, the neural network is evaluated to produce move probabilities $\mathbf{p}$ and expected outcome v. For each possible action a and new board state s', the tree is expanded with a new edge and child node. The action's prior probability is set to the corresponding value in the move probabilities vector, $P(s, a) = p_a$. Then the algorithm works back up the tree, incrementing each action's visit count $N(s, a)$ and adjusting its average expected value $Q(s, a)$ based on the expected outcome v of the leaf node. After a fixed number of MCTS simulations, strengthened move probabilities $\boldsymbol{\pi}$ are generated for the root node using visit counts $N(s_0, a)$ and temperature parameter τ:

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

When $\tau = 1$, the move probabilities are proportional to their respective visit counts. As $\tau \to 0$, $\boldsymbol{\pi}$ becomes a one-hot vector representing the best move. In practice, τ is switched from 1 to 0 after a set number of moves to encourage exploration early in the game. Using the strengthened move probabilities $\boldsymbol{\pi}(a|s_0)$, an action is randomly selected and the game proceeds. A subtree of the existing search tree is used to start the next set of MCTS simulations.

2) Neural Network: During self-play, the board state s and strengthened move probabilities $\boldsymbol{\pi}$ from MCTS are recorded before each action is taken. At the end of the game, a series of training examples of the form $(s, \boldsymbol{\pi}, z)$ are generated with this data. Each training example also includes the final outcome of the game z.

This self-play data is uniformly sampled to further train the neural network. AlphaZero uses the following loss function to minimize the MSE of expected outcome and cross entropy of move probabilities, along with a L2 regularization term:

$$l = (z - v)^2 - \boldsymbol{\pi}^T \log \mathbf{p} + C||\theta||^2$$

AlphaZero uses a single neural network with a policy head for move probabilities $\mathbf{p}$ and a value head for expected outcome v. The core of the network is a common residual tower with fully connected layers on either head. DeepMind showed that this "single-res" architecture performs better than two individual networks or a network that uses just convolutional layers. It has been suggested that using the network for two objectives promotes regularization of the residual tower, which combined with computational efficiency, increases performance.

Our network's output representation differs slightly from AlphaZero's convention. The output of the policy head is a 2272×1 vector where each entry represents a legal move in Bughouse. This contrasts with AlphaZero's plane representation, where more than half the entries are invalid moves (i.e. off of the board). Finally, the values in our compressed output vector are masked to limit selection to legal moves.

C. Supervised Learning

To explore alternate training methods, we implemented a variety of regression models to predict $V(s)$ for any state s (similar to the CrazyAra engine).

1) Data Processing: Raw game data from the Free Internet Chess Server (FICS) Bughouse database [9] were parsed to create a sequence of moves aligned with a win or loss result from the perspective of team 1. Each move was then played and the resulting state s stored as an array representation of the board, along with the value of the state represented, as calculated by

$$V(s) = \gamma^{n-i} R$$

where $\gamma = 0.99$ represents the discount rate, n represents the number of moves in the game, i represents numbers of moves hitherto played, and $R = \pm 1$ represents the game result from the perspective of team 1 (1 for victory). It was further set that $V(s_0) = 0$ for starting state s_0.

2) Models: Models were trained on the data with states as input and expected value as output. All of the following were tested: linear regression, ElasticNet, decision tree, and random forest (with 10 estimators). Given that many overfit the train data, PCA was run on the states to reduce them to 512 principle components that captured over 99% of information, and another set of the aforementioned models was trained on this dataset.

a) Linear Regression: We fit a linear regression model $h_\theta(s) = \sum_{j=1}^{m} \theta_i s_i$ to optimize mean squared error and solve the following optimization problem:

$$\min_\theta \frac{1}{2} \sum_{i=1}^{n} (h_\theta(s^{[i]}) - V(s^{[i]}))$$

b) ElasticNet: ElasticNet adds a linear combination of L_1 and L_2 penalties $||\theta||_1$ and $||\theta||_2^2$ to linear regression to prevent overfitting.

c) Decision Trees: We also fit a decision tree classifier. This is a non-parametric model that builds a tree of decisions using the input that best classifies the data at a given node.

d) Random Forest: Random forest is an ensemble model that sub samples the data and fits several decision trees to improve the performance of the model and help prevent overfitting.

D. Baselines

1) Minimax: As our primary baseline, we implemented a Bughouse engine using a conventional minimax approach. To make the tree-search more tractable, several optimization techniques were used, including alpha-beta pruning and transposition tables.

Minimax is governed by the equation

$$u = \min_{-i} \max_{i} u_i(a_i, a_{-i})$$

where u is the overall utility, a_i are the actions for the maximizing player (you), and a_{-i} are the actions for the minimizing player (the opponent). Alpha-beta pruning works by creating a feasibility region (α, β) for the results. If a branch (a move) is outside of the feasibility region, it isn't explored because we know that move in particular would never work.

To evaluate a board position after a series of moves, we define a heuristic

$$u_b = \sum_{x=1}^{8} \sum_{y=1}^{8} (V(x, y) + P(x, y))$$

where $V(x, y)$ is the value of the piece at position (x, y) and $P(x, y)$ is an added value of that given piece being specifically at position (x, y).

A transposition table is also setup to ensure that the same position is never processed twice. In chess, this happens frequently because a board can end up in the same state if the move order of two moves is switched.

TABLE II
COMPARISON OF HYPERPARAMETERS BETWEEN THE ORIGINAL ALPHAZERO IMPLEMENTATION AND OUR BUGHOUSE IMPLEMENTATION.

	AlphaZero	Bughouse
Mini-Batch Size	4096	64
Mini-Batches per Checkpoint	1000	6250
Learning Rate	0.2 (annealed)	0.002
Games per Checkpoint	63K	10
MCTS Simulations per Move	800	5
Resnet Depth	39	7
Hours Trained	9	24

TABLE III
HEAD-TO-HEAD PERFORMANCE (20 GAMES). NUMBER OF WINS BY ROW AGAINST COLUMN ARE REPORTED. DRAWS ARE IN PARENTHESES.

	RM	MM	LR	EN	DT	RF
Random Move		0	0 (1)	17 (3)	16 (4)	7 (2)
MinMax Baseline	20		20	20	20	20
Linear Regression	19 (1)	0		20	20	20
Elastic Net	0 (3)	0	0		0 (20)	0 (20)
Decision Tree	0 (4)	0	0	0 (20)		0
Random Forest	11 (2)	0	0	0 (20)	20	
A0 Reinforcement	1	0				
A0 Supervised	19	4				

2) Random Move Baseline: As a secondary baseline, we implemented a Bughouse engine that picks a random legal move (with no strategic or material considerations).

IV. EXPERIMENTS AND RESULTS

Using the AlphaZero algorithm (implementation based on [10]), we attempted to learn Bughouse chess through reinforcement learning but ran into several resource constraints. We then tried to train the AlphaZero network on Bughouse games played on FICS (Free Internet Chess Server) with better results. Finally, we constructed the other supervised learning models using the same dataset. All of these approaches were compared to a minimax agent and random agent as baselines.

A. Training Pipeline and Hyperparameters

1) Reinforcement Learning: In the AlphaZero paper, the self-play and network training steps happen asynchronously. Each self-play game queries the latest neural network weights, and the network is trained on examples sampled randomly from buffered self-play history. To simplify our implementation, we alternate between both steps on a single thread.

An additional consideration was the extensive computational power that DeepMind was able to employ when learning standard Chess. Per the AlphaZero paper, they used 64 second-generation TPUs for network training and 5000 first-generation TPUs for self-play. This hardware allowed them to generate nearly 80 network checkpoints and 5 million self-play games per hour. With significantly less games available, we have to use a smaller mini-batch size and lower learning rate. Additionally, because DeepMind can generate so many self-play games, they only sample about 20% of their buffered self-play history per checkpoint. In our implementation however, we run 10 epochs of our buffer (6250 mini batches total) per checkpoint. A comparison with our derated hyperparameters for single GPU training can be found in Table II.

2) Supervised Learning: For supervised learning with the AlphaZero network, we weighted the MSE (expected outcome) loss term by a factor of 0.01 as was done in the AlphaZero paper. This biases the model to learn the best moves from the dataset rather than predicting outcome perfectly. We used mini batches of 1024 samples and ran training for 8 hours, processing roughly 2400 mini batches.

B. Evaluation Metrics

1) Head-to-Head W/L/D: Each engine was tested against the other engines in a head to head Bughouse competition. Colors were switched at the halfway mark to eliminate bias.

2) Elo: Elo is a standard rating system for games such as Chess. The premise is that rating is gained or lost based on game result, in addition to the strength of the opponent (the better the opponent, the more elo per win). Elo is also probabilistically backed – at a 100 elo point elo difference, the higher rated player should win 64% of the time. At a difference of 200, that goes to 76%.

Typically, elo is given absolutely. However, because we have a closed pool of engines that can't play any players with established ratings, it is easier to instead compute an expected difference in elo between two engines given the head to head score [2]. This can be calculated as follows using wins w, losses l and draws d:

$$\delta_{\text{elo}} = -400 \log\left(\frac{1}{p} - 1\right)$$

$$p = \frac{w + d}{w + d + l}$$

C. Evaluation

To evaluate the performance of the different engines, we ran them head to head in 20 matches. The engines were not given a time limit per move, but rather were configured to come up with their moves relatively quickly. When using the AlphaZero neural network and MCTS, we performed 25 simulations. Because the minimax engine was slower to compute, its search depth was set to 1 (the immediate response by the opponent on the same board). The results of these matchups are listed in Table III and Table IV.

D. Discussion

1) Reinforcement Learning: The AlphaZero reinforcement learning produced an agent that was considerably worse than the random baseline. We believe this was due to the way we had to configure our parameters due to the computational cost of self-play game generation. By using a short MCTS search and training repeatedly on the same games, we most likely caused the model to overfit a small set of possible moves.

[2]Elo difference is not well defined for matches where one side has no wins, but we report it as 700 point difference because that is approximately what it is

TABLE IV
RELATIVE ELO BETWEEN ENGINES.

	RM	MM	LR	EN	DT	RF
Random Move		-700	-636	+436	+382	-70
MinMax Baseline	+700		+700	+700	+700	+700
Linear Regression	+636	-700		+700	+700	+700
Elastic Net	-436	-700	-700		0	0
Decision Tree	-382	-700	-700	0		0
Random Forest	+70	-700	-700	0	+700	
A0 Reinforcement	-512	-700				
A0 Supervised	+512	-241				

2) Supervised Learning: Supervised learning with the AlphaZero network produced the best results and was even able to beat the minimax agent with very little training. We believe this is because we had a much larger dataset, so we could run many more batches without reusing the same game data. In addition, this data was much higher quality, rather than the near random play being produced in reinforcement learning.

V. FUTURE WORK

A. Agent Structure

To simplify the problem, we structured our engine as a single agent making a move on both boards for the Bughouse team. An alternative is to have two teams, each comprised of two collaborating agents. Structuring the agents to play like this can be challenging (because the engines need to figure out how to be aware of the needs of each other), but searching one board space massively reduces the dimension of the problem and might make it more tractable. Additionally, individual agents may allow the Bughouse timing problem to be more easily addressed.

B. Improved Training

One of the cornerstones of the AlphaZero algorithm was the raw computational power available to DeepMind. This allowed them to generate vast datasets and always train on fresh data played with the newest model. We chose a game with a much higher branching factor than even the games used by DeepMind, but had very little computing power in comparison. In addition to speeding up training, faster hardware could be used to run several training sessions and pick optimal hyperparameters, something we could not investigate.

C. Real-World Testing

To test all of our engines, we mainly played them against each other. If we were to continue this project, we could release our engines onto common game platforms such as chess.com, and obtain a concrete engine elo from performance against human opponents with established elos.

VI. CONTRIBUTIONS

1) Devin: Set up the Bughouse game representation, self-play based engine, and re-built the NN structure for the self-play engine. Modified the testing arena to handle input from any generalized Bughouse engine.

2) Robbie: Developed minimax-based baseline engine. Performed self-play experiments. Performed head-to-head engine testing. Provided general guidance on the game of Bughouse.

3) Abhay: Generated (state, value) dataset from Bughouse-DB BPGN files. Trained all supervised models. Performed gameplay experiments for supervised models against each other and against baselines.

REFERENCES

[1] D. Silver, A. Huang, C. Maddison, A. Guez, L. Sifre, G. Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, "Mastering the game of go with deep neural networks and tree search," *Nature*, vol. 529, pp. 484–489, 01 2016.

[2] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan, and D. Hassabis, "Mastering chess and shogi by self-play with a general reinforcement learning algorithm," 2017.

[3] M. Campbell, A. Hoane, and F. hsiung Hsu, "Deep blue," *Artificial Intelligence*, vol. 134, no. 1, pp. 57–83, 2002.

[4] "Stockfish." https://stockfishchess.org/.

[5] A. Tridgell and L. Weaver, "Learning to play chess using temporal differences," *Machine Learning*, vol. 40, pp. 243–263, 09 2000.

[6] O. E. David, N. S. Netanyahu, and L. Wolf, "Deepchess: End-to-end deep neural network for automatic learning in chess," *Lecture Notes in Computer Science*, p. 88–96, 2016.

[7] M. A. Gehrke, "Assessing popular chess variants using deep reinforcement learning," m.sc., TU Darmstadt, jul 2021.

[8] J. Czech, M. Willig, A. Beyer, K. Kersting, and J. Fürnkranz, "Learning to play the chess variant crazyhouse above world champion level with deep neural networks and human data," *Frontiers in Artificial Intelligence*, vol. 3, p. 24, 2020.

[9] "FICS Bughouse Database." https://www.bughouse-db.org/dl/, 2021.

[10] "Alpha Zero General for Chess and Battlesnake." https://github.com/Unimax/alpha-zero-general-chess-and-battlesnake, 2021.