Mastering Reconnaissance Blind Chess with Reinforcement Learning

Sergey Savelyev

Georgia Institute of Technology
North Avenue Northwest
Atlanta, GA 30332
April 2020

Abstract

Research within Artificial Intelligence has often set goals of being able to autonomously play games (e.g., Chess or Go) at or above human level [1, 2]. Novel machine learning-based agents have recently made advances in the state-of-the-art by achieving superhuman performance in increasingly complicated games. We believe that solving imperfect information games (i.e., games where you do not have full knowledge of the opponent's activities) should be the next goal in Artificial Intelligence research. We study Reconnaissance Blind Multi-Chess (RBMC), an imperfect information variant of Chess, which comes with a novel set of challenges that must be overcome before a computer can attain superhuman performance. Prior works have largely focused on reducing the problem to a game of standard Chess (i.e., with perfect information) by attempting to determine the true state of the Chessboard. This procedure separates the problem of acquiring and applying gathered information from the move policy, allowing existing Chess agents to be used to choose nearly optimal moves. In contrast, our method trains a triple-headed neural network through self-play reinforcement learning, handling the information-gathering process, and move process within one model. Since this agent does not attempt to solve a restricted version of the problem, the algorithm is able to execute strategies based on the imperfect information aspect of the game. We believe that such a learning method, given enough training time, should be able to outperform agents that simply reduce the problem to a standard game of Chess. In this thesis, we explore this hypothesis and algorithms for playing RBMC.

Keywords Reconnaissance Blind Multi-Chess, Reinforcement Learning.

Acknowledgments

Thanks to Rohan Paleja and Professor Matthew Gombolay, without whom this work would not be possible.

Contents

1 Introduction

Two player, zero-sum games have been studied both in game theory and artificial intelligence in an attempt to develop agents that play at or above human level [1, 3]. Many games fall under this category, such as Tic-Tac-Toe, Chess, and Go. These types of games are typically solved by searching over a set of actions and choosing one that is calculated to be more likely to lead to a winning board position. However for complicated games, such as Chess, the length of the game and the size of possible moves results in a very large search space, resulting in computational intractability when searching for an optimal strategy. For other games, such as partially observable games, the state space is partially occluded resulting in searching over belief spaces as well, where each belief in the belief space is a separate distribution over the state space. Combining a search over actions and beliefs results in a much larger state space, and therefore efficient methods for finding a high-performing policy must be developed.

1.1 Task Domain

We study the game of Reconnaissance Blind Multi-Chess (RBMC), a Chess variant in which a player cannot see their opponent's pieces. At the beginning of each of their turns, each player can pick a three-by-three region of the board to reveal, which displays the true state of the board in this region without notifying the opponent which region was scanned. After the "reconnaissance" move, the player must move one of his or her pieces (similar to normal Chess). Yet, as neither player knows what the true board looks like, neither can truly know which moves are legal or illegal or even what pieces are still in play. The information available to be extracted from the player's own moves includes whether a given move is legal or illegal and whether your piece successfully captured another piece (note this does not include which piece was captured). Formally, each turn, we receive information in the form $s_* \subset S$, where S is the true board state, which contains information regarding the result of opponent actions (e.g., did they take one of my pieces?), $s_{**} \subset S$ after the "reconnaissance" move, and partial feedback $s_{***} \subset S$ upon making the action $a \subset A$ (e.g. whether the requested action was legal or not). By combining the results of the scans, moves, and accounting for any of the player's pieces that have been taken, the player must make inferences about the true state of the board in order to defeat their opponent. This reasoning is in stark contrast to the game of normal Chess where both players always have access to the same information, and a player must only try to predict their opponent's next moves to choose the best action. Not only does RBMC require state tracking, but the game also requires a tradeoff between gathering information and playing to capture opponent pieces. We hypothesize that an agent can use machine learning to learn this balance.

1.2 Solution Approach

While state-of-the-art (SOTA) Artificial Intelligence (AI) approaches to regular Chess such as AlphaZero [4] are incredibly successful, these systems cannot be applied directly to RBMC as they require knowledge of the true state of the board. The approximate total number of game states that can be considered in a game of RBMC is approximately 10^{93} times that of a game of Chess [5], nearly the same

as in a game of Go, because each state requires keeping track of not only what the game board actually looks like, but the information that each player has acquired. For this reason, each state is much larger in RBMC than it is in Go [5], requiring more memory per state. Thus to play competitively, the player must keep track of what he or she believes that their opponent knows about their pieces, as playing to confuse their opponent could be a valid and powerful strategy. RBMC allows players to study a tradeoff between exploitation (i.e., taking actions with the goal of winning the game) and exploration (i.e., taking actions to gather more information). A balance is needed as the player cannot win the game if he or she does not know where the opponent's king is (as to capture it); however, if the player spends all of his or her time trying to figure out the true board state through information-seeking actions, the player will likely never capture the king. As Chess is an analogy for war, RBMC acts as an analogy for modern warfare, where modern technologies have made it difficult to see the opponent's movements and resources. The jump in complexity caused by the loss of perfect information makes RBMC an important milestone in the future of AI, requiring new methods to deal with this additional complexity.

Computer programs that play perfect information games, such as regular Chess, typically do so by finding a path through the game tree that leads to the desired outcome. The game tree is a representation of the possible states of a game reachable from a given state. Each of these states is connected by the move that the current player has to execute in order to reach the next state. For a simple game, such as Tic-Tac-Toe, the entire game tree can be quickly searched by a computer in order to find the optimal move to perform. However, in Chess, only a portion of the full game tree can be searched in a reasonable amount of time due to the size of the search space. Specifically, the search space of a game tends to have a size of $\mathcal{O}(b^d)$ where b is the branching factor (i.e. the number of moves there are per turn on average) and d is the depth of the game tree (i.e. the number of moves it takes for a game to reach its end). Assuming the average depth of RBMC to be 40 moves and an average branching factor of 10,000 [5], this comes out to a search space size of $10{,}000^{40} = 10^{160}$, a size far outside the capabilities of modern computers. Thus, algorithms must employ clever techniques to reduce how much of the game tree must be searched. One such algorithm is Monte Carlo Tree Search (MCTS) [6]. MCTS searches the game tree by simply picking moves it has not yet tried until the game is over. While performing this operation once will not give a good indication of which move the computer should make for a given turn, repeating this process tens of thousands of times will allow the computer to calculate which move tends to result in the most favorable outcome on average. Computer programs that play perfect information games using MCTS are only as good as the paths they find through the game tree.

A state-value heuristic, a function that quantifies the value of a state for the current player, can be used to improve the efficiency of search algorithms. AlphaZero [4], one of the state-of-the-art algorithms for playing Chess uses a heuristic learned through Deep Reinforcement Learning. This deep reinforcement learning-based approach allows the computer to learn a heuristic, a process that allows for the generation of a much more powerful heuristic than one that can be engineered by people. This relative advantage is due to the fact that engineers may accidentally overlook, not know about, or misjudge the importance of features in the heuristic.

Further, learning a heuristic ensures that any, possibly incorrect, human biases are not introduced into the heuristic [4]. The learning-based algorithm is also made to be as general as possible to reduce the amount of bias introduced in the structure of the algorithm itself. By guiding MCTS with a powerful heuristic, AlphaZero can find an optimal move while only checking eighty thousand possible future states per second, compared to Stockfish, the agent AlphaZero Chess was tested against, which evaluated seventy million possible future states per second [4, 7]. However, AlphaZero in its current form only works on perfect information games, as it requires knowledge of the true state of the game board to search the game tree.

Our goal is to create an AI approach that can successfully play RBMC and apply techniques from AlphaZero to work in an information imperfect environment. This work will build off of DeepMind's AlphaZero, many of its predecessors, and the strategies employed in other information imperfect adversarial games (e.g., poker). This approach will track probable hypotheses for the true state of the board, using scans to reduce the number of total hypotheses that are consistent with the set of information available. These hypotheses are estimates of the true state of the board currently being considered by the agent while it is determining how it should play the game. In order to maintain a set of hypotheses, this approach will look at its hypotheses from previous turns and, using all information gathered, generate consistent hypotheses for the current turn. We will then use a model learned using Deep Reinforcement Learning to combine move probability distributions across hypotheses and determine a scan policy. As we evaluate optimal moves across a set of board beliefs, an optimal move and scan can be found probabilistically without needing to know the true state of the board.

2 Literature Review

Artificial Intelligences for playing perfect information games, games where all players have access to full state information, such as Chess, typically search the game tree for a path that leads to a favorable outcome. This game tree is a set of states connected by the move that needs to be made at each state to reach the next state. As the game tree consists of both the player's moves and the opponent's, the assumption that the opponent plays rationally [8] is typically made to generate the tree. Chess programs such as DeepBlue, which beat Kasparov, the world champion in 1996, used this assumption to reduce the number of states that needed to be evaluated [9]. In addition, DeepBlue used a heuristic to attempt to predict the expected value of a given state without needing to evaluate the entirety of the game tree from that state. This estimate of the expected value given by the heuristic roughly corresponds to an estimate of how likely it is to win from that state. The better a heuristic is, the more accurately it evaluates a given state. However, the performance of an artificial intelligence agent depends on the heuristic, leading to negative bias against strategies the designer failed to consider.

A deep-learning-based state-of-the-art method for creating artificial agents for playing perfect information games comes from Silver and the DeepMind team [4, 10, 11]. Silver describes an algorithm for creating a heuristic for move and state evaluation in games such as regular Chess, Go, and Shogi. The heuristic is generated from Deep Reinforcement Learning of MCTS using the previous iteration of the heuristic to guide the MCTS algorithm. This iterative process, in effect, creates

a positive feedback loop such that if the heuristic allows MCTS to discover new strategies, the next iteration of the heuristic will incorporate the new strategies into its move policy and state valuation. Given enough of these iterations, the DeepMind team created an artificial agent that is able to play Chess at the grand-master level, learning to play from nothing but the set of legal moves.

Whereas perfect information games played by a computer only require it to efficiently search the game tree, imperfect information games have an additional set of challenges. These challenges include determining what the true state of the game is and learning to read information from the opponent's actions. Much of the research into imperfect information games is based around the card game of poker. In poker, the cards in the opponent's hand can be modeled using probabilities, allowing the player to reason over probable actions to take to win the hand. Billings et al. create a poker-playing program that learns a model of its opponent as it continues to play against the same opponent [12]. Repeated play against an opponent allows this program to use the information that the opponent gives off to its advantage, such as in which situations the opponent is likely to raise, hold, or fold and generate response strategies accordingly. By reincorporating this model back into the program's expectations of the opponent's hand, the program is able to increase its likelihood of winning, causing it to win every match it played against its opponent. While this application is slightly different than it would be in RBMC, a similar concept of learning through repeated play would be incredibly useful. In addition, as different opponents may employ different strategies, the exact approach used by Billings et al. would not generalize to all opponents. However, similar techniques could allow the program to learn to predict what kinds of strategies the opponent is likely to consider.

RBMC was proposed as a game to serve as a target for research in 2016 [13]. Chess was invented as an analogy for war, and RBMC extends that into the modern era, where one does not necessarily know what resources the opponent has at their disposal or where the opponent's units are placed. This added complexity makes RBMC an important milestone in the future of AI research. Due to its novelty, there is a lack of research papers within the scope of RBMC AI agents. One of the oldest works in this same field is Markowitz, Gardner, and Llorens's paper *On the Complexity of Reconnaissance Blind Chess*, which examines the game of RBMC under a critical lens to determine its complexity compared to other games [5]. Markowitz, Gardner, and Llorens estimate how much computation is required to compute the game tree, finding there to be approximately 10^{93} times the amount of states in a game of RBMC compared to a game of standard Chess. This is where our research will fit within the literature, serving to show others what we did to solve the problem of imperfect information in the game of RBMC. This study proposes an Artificial Intelligence to play RBMC that has been carefully designed to produce high-performing behavior.

As of the NeurIPS 2019 Reconnaissance Blind Chess Tournament, a number of RBMC bots have been released [14]. The winner of this tournament, Strangefish, is available on GitHub and can be easily set up to run locally [15]. Strangefish works by keeping track of all possible boards at any given turn. As the set of all possible boards grows to be very large, Strangefish only considers as many of them as it can within a given amount of time. Strangefish then calculates move scores for each of these possible boards based on how useful each move is expected to be using

Stockfish [7] and scan scores for each of the possible scans based on how much each scan would reduce the set of all possible boards. The Strangefish agent then picks a scan that will have the greatest possible impact on the move decision by taking into account the move scores, the scan scores, and the expected likelihood of each of the boards. After scanning, Strangefish refines its set of all possible boards. Next, to choose a move, Strangefish uses a weighted sum of its previously calculated move scores and an expected likelihood for each of the boards. Strangefish also uses a measure of how the chosen move will reduce the set of all possible boards in its decision, although this factor affects the move choice minimally.

Another baseline we benchmark against was the Trout agent [14]. The simpler Trout bot kept track of a single board state. Trout chose moves by evaluating this single board state using Stockfish. Trout chose scans semi-randomly from the set of squares that are not occupied by Trout's pieces. We now present our RBMC findings. In Section 3, we cover preliminary material. In Section 4, we detail our approach.

3 Preliminaries

Here, we discuss POMDPs to frame the RBMC problem and introduce Deep Reinforcement Learning as used by AlphaZero [4].

3.1 Partially Observable Markov Decision Processes

A Partially Observable Markov Decision Process (POMDP) can be used to model a stochastic environment where an agent does not have direct access to the state of the environment, and instead receives observations that may provide partial state information. A POMDP consists of a set of states S, a set of actions A, a set of observations O, and a real-valued reward function $R(s, a)$ [16]. At each time step, an agent navigating the POMDP's environment starts from state $s_0 \in S$, makes observation $o_0 \in O$, and chooses action $a_0 \in A$. This results in the agent receiving reward $R(s_0, a_0)$ and transitioning to s_1. However, as this is a POMDP, the agent is unaware the exact state it has transitioned to. Both the probability of ending up in state s_{i+1} from state s_i when taking action a_i, and the probability of making observation o_i when in state s_i may be unknown to the agent. The goal of the agent in the POMDP environment is to maximize the expected total reward. This expected total reward is $\mathbb{E}[\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t)]$ where γ is the discount factor for future rewards. The solution to the POMDP that maximizes the expected total reward is the optimal policy. However, since the agent does not know what state the environment is in at any given time step, this policy is not a mapping from state to action but rather a map from a probability distribution over the set of states S to an action. As the relationship between observation and state is unknown to the agent, finding a policy is extremely difficult as doing so requires dealing with state ambiguity and credit assignment simultaneously. Thus, an agent in a POMDP has to explore and exploit to produce actions that lead the agent into states with meaningful observations and high rewards. If the agent does not know what state the agent is currently in, a policy would not be able to make any useful decisions about its current state. On the other hand, if the agent only moves to acquire new information, it is not guaranteed that moving to acquire information will result in high rewards.

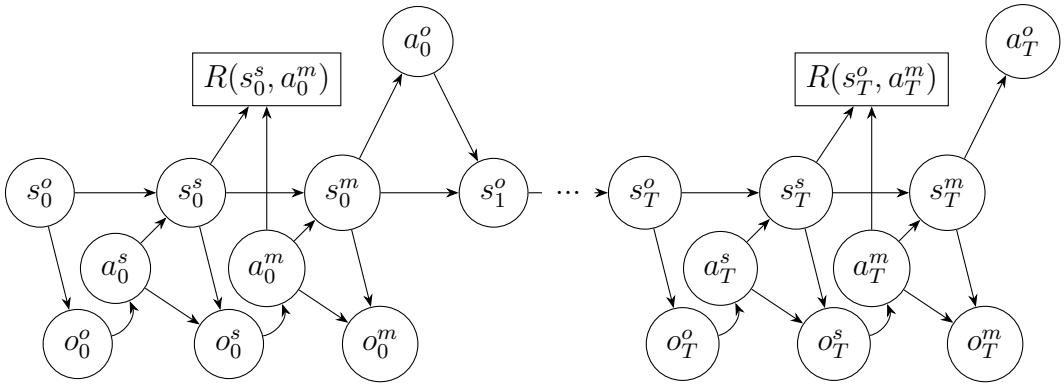


Figure 1: This figure shows the state transitions, observations, actions, and rewards for the Reconnaissance Blind Chess POMDP model. Figure derived from Artificial Intelligence Foundations of Computational Agents [17].

In RBMC, we define S as the set of all possible Chess boards. A is the set of all possible Chess moves and all possible scan actions. O is the set of all possible observations. O includes all possible scan results, gathered opponent move information, and move attempt response information. Figure 1 shows the transformation of states and the flow of information throughout a game of RBMC. The state $s_t^o \in S$ represents the state at the start of the player's t-th turn (right after the opponent turn has ended). This state provides the player with observation $o_t^o \in O$ containing gathered opponent move information from the move the opponent had just recently completed, such as whether the opponent took one of the player's pieces. The player then performs scan action $a_t^s \in A$. This takes the player into state $s_t^s \in S$. This state, s_t^s, has the same Chessboard configuration as s_t^o. The player is also provided with observation $o_t^s \in O$ containing the scan result information for scan a_t^s. Next, the player makes move action $a_t^m \in A$ resulting in state $s_t^m \in S$. State s_t^m is the result of performing the Chess move represented by action a_t^m from state s_t^s. From state s_t^m, the player then receives observation $o_t^m \in O$ containing information about the success or failure of move a_t^m. At this point, the player's turn is over and the opponent makes a move. This move action $a_t^o \in A$ is hidden from the player, but it results in state $s_{t+1}^o \in S$ representing the state at the start of the player's next turn. The reward function $R(s_t^s, a_t^m)$ is 1 if action a_t^m caused the agent to win the game, -1 if the agent's king had already been captured in state s_t^s, and 0 otherwise (i.e. if the game has not yet been decided or if it results in a draw). Thus, an agent that follows a policy that optimizes the expected total reward will follow a policy that favors it to win.

3.2 AlphaZero

AlphaZero is a general framework for creating super-human agents for adversarial, two-player, fully observable turn-based games [4]. In the paper, Silver et al. create state-of-the-art agents for Chess, Go, and Shogi using Deep Reinforcement Learning and Monte Carlo Tree Search. These agents would pick moves using MCTS guided by a model trained using Deep Reinforcement Learning.

3.2.1 Deep Reinforcement Learning

Deep Reinforcement Learning is a type of Machine Learning where a model with millions of parameters is trained to take actions in an environment in order to maximize a given reward function. The resultant of Deep RL is a learned policy that can predict high-reward actions given a state. This has been used to solve a wide variety of tasks such as Chess, Dota 2, and Atari [4, 18]. Although Deep Reinforcement Learning allows for many different problems to be solved by neural networks, Deep Reinforcement Learning takes a large amount of computational power. The agent needs to spend a lot of time trying actions that lead to high/low reward and increase/decrease their likelihood in future trajectories. Thus, training takes a long time because a single play of the game only provides so much information about the value of the actions taken during that game. This is especially prevalent in long-term sequential games with sparse rewards such as Chess. In order to get a strong estimate of the expected reward when using a policy, states must be visited many times to decrease the variance in the estimated value.

In AlphaZero [4], the reward function is only based on the terminal state and is 1 when the agent wins the game, -1 when the agent loses the game, and 0 when the agent ties. The agents in this work were trained using self-play reinforcement learning. Each agent plays its respective game against itself, recording the inputs at each time step, the moves taken at each time step, and the final result of the game. In parallel, the agent would be trained on this information, learning to better predict what moves should be taken and what the expected result of the game should be. As the moves taken by the agent are not directly based upon the output of the neural network but the output of MCTS, the network slowly improves its predictions of the game result and the probabilities with which moves should be taken. This, in effect, creates a positive feedback loop that results in the super-human state-of-the-art agents demonstrated by Silver et al. for Chess, Go, and Shogi.

While the inputs and outputs to AlphaZero's neural networks depend on the specific game, the main structure of the networks used in Chess, Go, and Shogi are the same. These networks take as input a representation of the state of the game from the last eight time steps and output a move policy as well as an expected value for the current state. For Chess, the neural network used by AlphaZero has an input consisting of 119 8×8 images. Seven of these images are used to provide the neural network with the team color, total move count, castling legality for both teams, and a no progress count for the current time step. The remaining 112 are grouped into 8 sets of 14 based on the time step that it is representing. These 14 images represent player 1's pieces (using 6 images), player 2's pieces (using 6 images), and the current repetition count (using 2 images). In total this uses a one-hot encoding to represent the state of the Chessboard for the current time step and the 7 previous ones (including when it was the opponent's turn to move). The expected value for the current state is outputted as a single real number. The move policy, on the other hand, is returned as a set of move probabilities for all possible types of moves for all 64 tiles in the Chessboard where the move could start. Although not all of these moves are even possible because of the bounds of the Chessboard, this leads to a very easy and fast way to get the probability for a given move.

input : The current state B, number of iterations n, neural network M, and exploration parameter c_{exp}

output: A move policy

/* $N(s,a) =$ Node.visits */
/* $P(s,a) =$ Node.prior */
root $\leftarrow$ Node(state $= B$, prior $= 1$, children $= \{\}$, visits $= 0$) ;
for $i \leftarrow 1$ **to** n **do**
    node $\leftarrow$ root;
    states $\leftarrow [B]$;
    /* Step 1: Selection */
    while node .children $\neq \{\}$ **do**
        node $\leftarrow \arg\max_{\text{c}\in\text{node.children}}(\text{c.value} + c_{exp}\text{c.prior}\frac{\sqrt{\text{node.visits}}}{\text{c.visits}+1})$;
        states $\leftarrow$ Append(states, node.state);
    end
    /* Steps 2 and 3: Expansion and Evaluation */
    MovePolicy, value $\leftarrow M$(states);
    node.children $\leftarrow$ {node(state $=$ ApplyAction(node.state, action), prior $=$ MovePolicy(action), children $= \{\}$, visits $= 0$) : action $\in$ LegalActions(node.state)};
    node.children.prior $\leftarrow$ Softmax(node.children.prior);
    /* Step 4: Backup */
    v $\leftarrow$ node.value;
    while node $\neq$ root **do**
        node.visits $\leftarrow$ node.visits $+1$;
        node.total $\leftarrow$ node.total $+$ v;
        node.value $\leftarrow \frac{\text{node.total}}{\text{node.visits}}$;
        node $\leftarrow$ node.parent;
    end
end
policy : $\pi(a \mid B) \propto x$.visits where $x \in$ root.children $\wedge\, x$.action $= a$;

Algorithm 1: Monte Carlo Tree Search

3.2.2 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) is an algorithm often used to solve Markov Decision Processes (MDPs). In an MDP, unlike in a POMDP, there are no observations and the agent is aware of what state it is currently in. MCTS assumes that the value of an action is the average reward received when randomly selecting game endpoints reachable from the state you get to when you take that action. AlphaZero uses a slightly modified version of MCTS that uses a neural network to guide its search of future game states. This modified MCTS explores the game tree, starting from the current state as the root, through the following algorithm. First, the algorithm descends down the game tree until it reaches a leaf node through a process called selection (Step 1 in Algorithm 1). This process reaches a leaf node by picking the possible action at each given node in the game tree that maximizes the value of the upper confidence bound $Q(s,a) + U(s,a)$ where s is the node and a is the action. $Q(s,a)$ is the expected value of the game once this action is taken (represented as c.value in Algorithm 1), calculated as a simple average of the value found each time the algorithm chose to take this particular action from this node. $U(s,a)$ is an exploration term to allow the algorithm to consider paths that it has not yet been to. $U(s,a) = c_{exp}P(s,a)\frac{\sqrt{\sum_b N(s,b)}}{1+N(s,a)}$, where c_{exp} is a constant used to control how much exploration the algorithm should do, $P(s,a)$ is the prior probability of taking this action (c.prior in Algorithm 1) determined by the neural network's move policy head, and $N(s,a)$ is the amount of times this algorithm has taken action a from node s (represented as c.visits and node.visits in Algorithm 1). Once a leaf node is reached, expansion and evaluation occur (Steps 2 and 3 in Algorithm 1). In expansion, the possible actions at the leaf node are considered and child nodes are created for each of the possible actions. The neural network is then evaluated on the node. The prior probabilities for each of the children nodes are set using the softmax of the move policy head's output and the node's value is determined from the neural network's value head's output. Finally, the backup step (Step 4 in Algorithm 1) occurs where the visit counts and the expected values are updated for each of the nodes this iteration of MCTS passed through. AlphaZero's agents then repeat these four steps $n = 800$ times. As MCTS is iterated more and more times, the depth of the search tree considered increases occasionally as MCTS expands more nodes. A final move can then be picked by the agents, where they choose an action for the current state from a distribution where the probability of choosing a given action is proportional to the number of times MCTS chose to take that action raised to the $\frac{1}{\tau}$ power, where τ is a temperature parameter used to control whether the algorithm should attempt to explore new strategies or exploit known strategies. In conclusion, the final policy returned by MCTS for AlphaZero's agents is $\pi(a \mid s_0) \propto N(s_0,a)^{\frac{1}{\tau}}$.

4 Our Approach

Here, we describe the elements of our RBMC agent shown in Figure 2. We use a residual neural network to output a joint move policy, state valuation, and scan policy. The scan policy determines the probability of each potential scan. The agent chooses how to scan based on these probabilities. The state valuation and move policies are used in the execution of the MCTS algorithm to pick a move. This model determines these policies using the 100 most probable beliefs of the true state

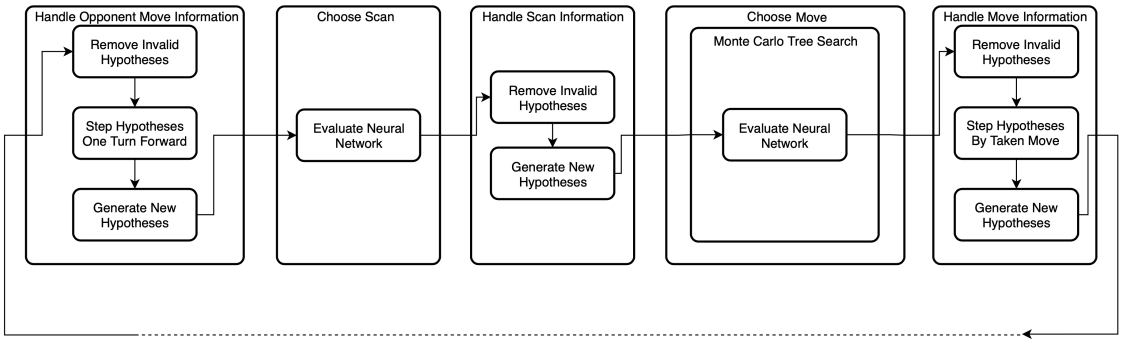


Figure 2: This figure shows an overview of our agent, broken into five stages. The dotted line at the bottom represents the opponent's turn.

currently being tracked as well as the strength of those beliefs.

In order to track the belief of the board state, the agent maintains 10,000 separate, non-unique hypotheses of the true state of the board. The number 10,000 was chosen as it was large enough to prevent the agent from losing the true state of the board in most circumstances, yet is small enough to avoid taking exceedingly long to make decisions. When the agent receives information, the system removes hypotheses that are not consistent with the information that the agent received. As this causes the total number of hypotheses considered by the agent to decrease, the agent then generates new hypotheses so that it stays at the same total number of hypotheses. These hypotheses are generated in such a way as to explore the belief space and consider possible hypotheses that were previously ignored.

Our agent works in five stages, as represented in Figure 2. The first stage deals with handling information received from the opponent's previous move. At the start of this stage, the agent's set of hypotheses are all possibilities of what the true state of the board was before the opponent moved. First, this stage removes any of the 10,000 hypotheses that are not consistent with the information that the agent received. For instance, if the agent was tracking hypotheses where it was impossible to take the captured piece, those hypotheses would be removed. Next, the agent steps the hypotheses one turn forward. It does so by modifying each hypothesis by a random opponent move that is consistent with the information that has been received. Now all the hypotheses still being considered are updated to represent a current hypothesis set. Since some hypotheses were removed, new hypotheses must be generated for the agent to remain at the 10,000 number before continuing to the second stage.

In the second stage represented in Figure 2, a location to scan is chosen by our agent. This is simply done by evaluating the neural network given the agent's beliefs.

The input to our neural network is an $N \times N \times (M \times T \times B + L)$ image stack that represents the beliefs of the state of the board through M planes for each of its B beliefs for the latest T time steps. As a Chessboard is 8×8, $N = 8$. $L = 7$ of these 8×8 images are used to represent the player color, total move count, castling legality, and the number of moves where progress was not made. The other $M \times T \times B$ of these images are used to encode the $B = 100$ most probable beliefs over the last $T = 8$ time steps. For each of these time steps, B beliefs are encoded using M images each. These $M = 13$ images are used to represent player 1's pieces (6 images), player 2's pieces (6 images), and the probability of this belief (1 image). This input encodes

the beliefs of the agent along with how the beliefs have changed in the last couple turns.

The network has three different output heads for the scan, movement, and value estimation. The move policy output head encodes move probability distributions using 73 $N \times N$ images. Each of the $N \times N$ locations in the images represents a starting position for the move and each image represents a different type of move. 56 of these images are used to represent queen style moves. These 56 are further grouped into 8 sets of 7 images where each set represents a different direction and the 7 images in each set represent a distance. These queen style moves are able to represent all possible moves that queens, kings, rooks, and bishops can perform as well as most possibilities for the moves pawns can perform. Although more moves are encoded than can be performed, moves that are impossible (i.e., invalid) are ignored. Another 8 of the 73 images are used to represent knight style moves. Finally the last 9 images are used to encode pawn underpromotion moves. When a pawn reaches the last row it is promoted into any piece that the player chooses except for a king. These 9 underpromotion images represent the possibility for the pawn to become either a knight, rook, or bishop when moving in any of the three possible directions for a pawn (attack left, move forward one, or attack right). All together each value present in these images represents a single unique move that can be performed in a game of Chess. The state representation is extremely large, requiring a very large number of parameters to accurately process this information through a neural network.

The value output head outputs a single number representing the expected value of the game given the current beliefs. This number in the range $[-1, 1]$ describes whether a win or a loss is expected by the network. It is used in conjunction with the move policy during MCTS to improve the network's move policy by looking at future time steps.

The scan policy output head encodes scan probability distributions using a single 6×6 image. Each of these 36 values represents the probability of scanning a single one of the inner 36 tiles of the Chessboard. The outer ring of tiles is ignored here as they provide no additional utility over scanning one of the inner 36. On the contrary, by scanning a tile in the outer ring, the agent would receive less information because part of the scan region would be out of bounds. When the neural network is evaluated in the second stage (as represented in Figure 2), these 36 values are normalized using softmax and then a scan is chosen randomly according to that distribution.

The third stage shown in Figure 2 handles the information received by the agent after scanning. First the agent removes any hypotheses that were invalidated by the information revealed by the scan. New hypotheses are then generated to keep the total amount at 10,000.

During the fourth stage (Figure 2), the agent chooses what move it should perform. To choose this move, MCTS (Algorithm 1) is used. Our MCTS is nearly identical to AlphaZero's [4]. Since we are playing an imperfect information game, our state is our 100 most likely beliefs rather than a single state. Our neural network also has an additional output head for the scanning policy. Apart from these differences, our MCTS algorithm is the same. Similarly, we choose our movement action for the current time step from a distribution where the probability of choosing a given action is proportional to the number of times MCTS chose to take that action raised to the $\frac{1}{\tau}$ power, where τ is a temperature parameter used to control how much exploration the algorithm performs.

Finally during the fifth and last stage (Figure 2), our agent handles the information received by the agent after making its move. The agent removes any invalidated hypotheses and then steps the other hypotheses forward by one time step so that the move that the agent took is represented in all of the remaining hypotheses. Finally, new hypotheses are generated leaving the agent with 10,000 hypotheses, all representing Chess boards where it is now the opponent's move.

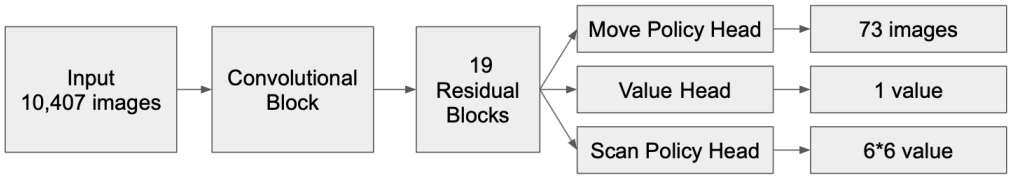
The neural network's architecture is similar to that of AlphaZero [4]. Figure 3a is a representation of the overall structure of this neural network. The main structure in this architecture is the residual tower which starts with a single convolutional block (Figure 3b) followed by 19 residual blocks (Figure 3c). The convolutional block has three parts to it: 256 padded convolutional filters using 3×3 kernels with a stride of 1, a batch normalization layer, and a rectified linear unit (ReLU) layer. The residual block is similar consisting of 256 padded convolutional filters using 3×3 kernels with a stride of 1, a batch normalization layer, a ReLU layer, 256 padded convolutional filters using 3×3 kernels with a stride of 1, a batch normalization layer, a skip connection to the input of the block, and a ReLU layer. The three output heads are then built on top of this residual tower. The move policy head (Figure 3d) has the structure: 2 convolutional filters using 1×1 kernels with a stride of 1, a batch normalization layer, a ReLU layer, and then a fully connected layer to the move policy head outputs. The value head (Figure 3e) has the structure: 1 convolutional filter using a 1×1 kernel with a stride of 1, a batch normalization layer, a ReLU layer, a fully connected layer to 256 nodes, a ReLU layer, a fully connected layer to a single node, and a layer. The scan policy head (Figure 3f) has the structure: 2 convolutional filters using 1×1 kernels with a stride of 1, a batch normalization layer, a ReLU layer, and then a fully connected layer to the 36 scan policy head outputs.

This model was trained using self-play reinforcement learning. Simply put, the model would play against itself, recording its inputs, the actions taken, the scans taken and the final result of the game. The model is then trained on these recordings such that the model is trained to predict the move that was made after many iterations of MCTS, the result of the game, and the scan that was made given the inputs the model has at the time. The variation in the input data is increased by removing some of the input beliefs before training so that the network learns to be more robust to missing information, similar to dropout. This is meant to allow the network to be able to function and return optimal outputs without needing to have the true state of the board as an input. The model's training and self-play happen simultaneously and the model being used for self-play is updated as games are finished.

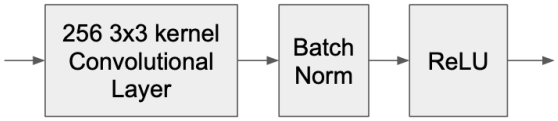
This methodology allows us to test our framework by playing against other agents. We evaluate our model this way, using existing RBMC agents, in the upcoming section.

5 Results

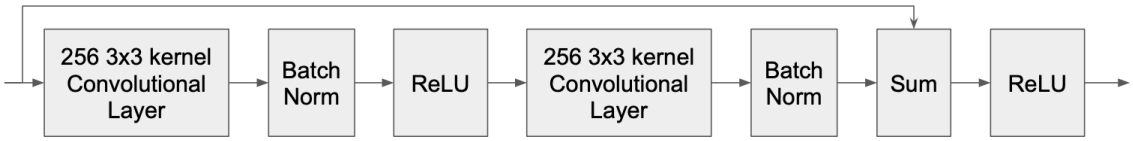
Throughout this section, we will first discuss our baseline agent and then move into the improvements that led to our final agent.



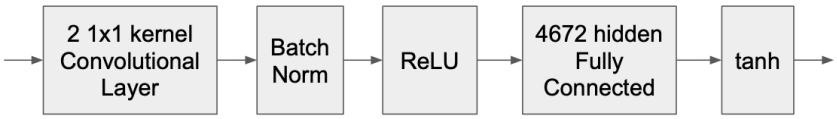
(a) This figure shows the overall structure of the neural network used by our agent built out of the pieces shown in figures 3b, 3c, 3d, 3e, and 3f.



(b) This figure shows the structure of the convolutional block.

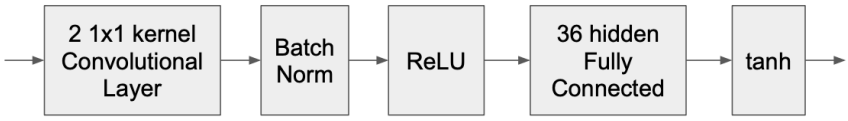


(c) This figure shows the structure of the residual block.



(d) This figure shows the structure of the move policy output head.

(e) This figure shows the structure of the expected value output head.



(f) This figure shows the structure of the scan policy output head.

Figure 3: This figure shows the various parts that make up our agent's neural network. The full structure is shown in figure 3a with sub-pieces shown in figures 3b, 3c, 3d, 3e, and 3f.

5.1 Baseline Agent

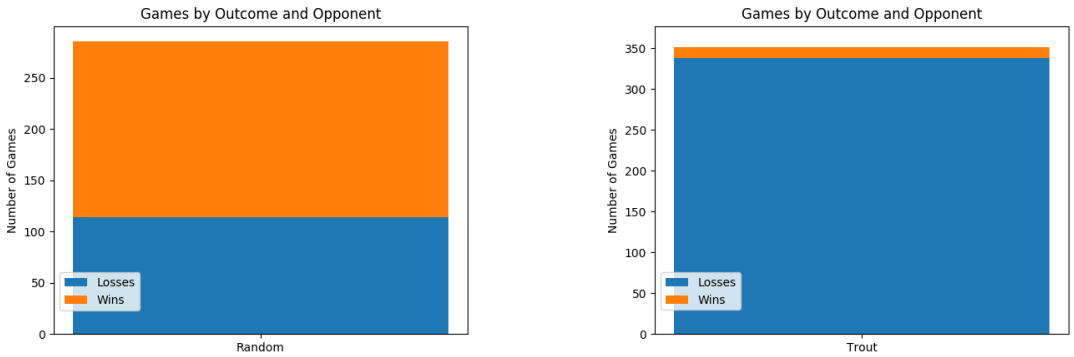
Prior to our modified agent, we created a simpler version that would use existing Chess agents in combination with a belief system. This system would scan in regions that were expected to have the greatest decrease in the number of possible hypotheses. This simpler agent first evaluated move probabilities on each of the hypotheses and then combined the different move probabilities to result in a policy weighted according to the probability of that hypothesis being the true state of the board. This agent used Leela Chess Zero's Neural Network as its joint move policy and value-estimation model [19]. This mechanism allowed us to avoid training a move policy and value-estimation model to play Chess and focus on a system to track the board state hypotheses efficiently.

In the baseline agent, scans were chosen to try and remove the most possible unique hypotheses from the agent's beliefs. Each possible scan position was evaluated and the scan position that had the most configurations of the 3×3 scanned region in the set of hypotheses was chosen to be used as the scan. Since only up to one of the configurations for a given scan position could be the configuration present in the true board, scanning at the scan position with the most configurations would cause the most possible unique hypotheses to be invalidated. In other words, a scan was chosen to maximize the size of the set $\{h_s | h \in H\}$ for scan s where H is the set of hypotheses tracked by the agent and h_s is a representation of the 3×3 region that would be revealed if h was the true state of the board and scan s was performed. Later, when scan s will be performed, the agent will receive information h^*_s, where h^* is the true board state. This information will invalidate any $h \in H$, where h_s does not match h^*_s.

As the baseline agent had evaluated up to 10,000 hypotheses each turn, each hypothesis is evaluated using a MCTS search using 800 iterations spread between unique hypotheses, or 1 iteration per unique hypothesis if there were more than 800 unique hypotheses. As we were constricted within the five-minute time limit used by the NeurIPS Reconnaissance Blind Chess Tournament [14], 10,000 was chosen empirically. In order to speed up the agent, the hypothesis tracking system had to assume that the system was correct about the state of the board in recent time steps. This is opposed to utilizing information to update previous sets of knowledge, and then regenerating current hypotheses based on information across turns (a much more computationally intensive task).

However, this assumption leads to the possibility that the agent may lose track of the true state of the board and disprove all of its hypotheses. This loss of awareness could happen if the agent decided to track different courses of action than the opponent took, propagating a mistake in its tracking turns ago. In the case where the agent begins to run out of time, the agent lowers the amount of tracked hypotheses and the total iterations of MCTS performed in order to not lose due to a timeout. This increases the likelihood the agent will lose track of the true state as the game proceeds. After having zero hypotheses that are consistent with the current information, the agent would switch into playing completely randomly rather than attempt to regenerate a new set of beliefs by backtracking, as it did not have the time to do so.

When running our baseline agent against an agent that moves purely randomly for the entirety of the game we see that we win more often than not. Figure 4 depicts the performance of our baseline agent against a random agent (Figure 4a) and the



(a) Random Agent

(b) Trout Agent

Figure 4: This figure depicts the performance of our baseline agent against a random agent (Figure 4a) and the NeurIPS Trout agent (Figure 4b).

NeurIPS Trout agent (Figure 4b). Our baseline agent won 60% of the time because even though at some point our agent would switch to playing randomly, the non-random play at the start of the game confers enough of an advantage that tends to last even when playing randomly. We found that a random agent was hard to play against because the belief tracking system was unable to make any assumptions about how the opponent would move. This scenario led our agent to have a tracking accuracy of around 10% against a random agent, as measured by the proportion of tracked hypotheses that corresponded to the true state of the board.

However, against a more powerful opponent our agent would tend to lose. The Trout agent works by keeping track of a single belief of the board which it updates very simply using almost random scans and choosing moves using Stockfish, a powerful heuristic-based Chess engine [7]. As the Trout agent did not move randomly, our agent was able to have a tracking accuracy (Figure 5) over 35% for much of the early portion of the game, later falling to 15% for the middle of the game and finally below 10% for the rest of the game as our agent would make sacrifices to its performance to stick within the time limit. Unfortunately, the cutbacks that had to be made to our agent to allow it to play a full-length game within the time limit meant that it cannot play well against a more powerful opponent, such as Trout.

The fact that our baseline agent needed to evaluate up to 10,000 hypotheses every turn in order to decide what move should be chosen by the agent for enough turns to finish the game, typically somewhere from 20 to 50 turns, means that our agent had to be able to evaluate a single hypothesis in 1.2 milliseconds assuming it spends all of its available time evaluating hypotheses for 50 turns. However, our agent also needs to keep track of its set of hypotheses, further reducing the amount of time it has available to spend evaluating hypotheses. Thus, our agent was usually unable to perform MCTS on all of its hypotheses and even then would typically run out of time before reaching 50 turns. The time constraint meant that the agent had to rely on learned knowledge of the future of the game rather than being able to evaluate possible future states. This limitation severely crippled this agent and forced us to switch to making purely random moves in order to prevent it from losing due to the time limit. Another drawback to this agent was that it used a heuristic trained to play standard Chess. This heuristic is likely to be sub-optimal for RBMC. Overall,

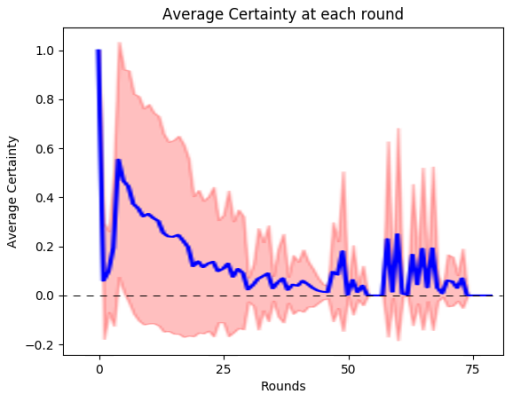


Figure 5: This figure depicts the ratio of the baseline agent's tracked hypotheses which are equal to the true state of the board for each time step against the NeurIPS Trout Agent. The blue line shows the average value of the ratio at each round. The red region shows one standard deviation above and below the blue line for each round.

our baseline agent highlighted that the speed at which we handle information and perform MCTS must be greatly improved before we are able to produce a high-performing agent.

5.2 Modified Agent

Our new agent avoids the computational bottleneck caused by having to evaluate 10,000 hypotheses every turn by using a single move policy, value-estimation, and scan policy network that can evaluate many of them simultaneously. The new agent takes the 100 most probable hypotheses and uses them and their respective probabilities in its evaluation of game states during MCTS. In doing so, the new agent mitigates this drawback by using a single neural network to evaluate a combined move policy over the most probable hypotheses. This new model also has input into what is scanned next. Utilizing a triple-headed neural network allows us to learn scanning criteria that are specific to its current strategy, rather than having to make do with scans that are only trying to reduce the number of hypotheses. While this modified agent does require us to train our neural network, the model then has more time available to run MCTS because it only has to do MCTS once per turn. Additionally, by having a combined scan and move policy, the agent is able to choose scans that have a larger impact on its future choice of move.

Unfortunately, this new agent is not without its faults. In order to train our agent using Self-Play Reinforcement Learning, hundreds of thousands of self-play games need to be recorded, storing inputs to the neural network, MCTS outputs, game results, and scans for every single turn. All 10,000 hypotheses for the last 8 turns, probabilities for all 4,672 possible outputs, the game result, and probabilities for all 36 possible for each turn are saved to a file using python pickling, averaging 3.5 megabytes in size. These files must be saved for each turn throughout the game, limited to a maximum horizon length of 100 to prevent the self-play from going indefinitely without generating results. As the agent is both used for white and black to generate self-play data for both sides, a full game played ends up generating

approximately 700 megabytes of data. This save file structure is a huge improvement on earlier versions of the training data saves. Originally the save files, which were saved using the .json format, averaged 10 gigabytes in size, meaning that each game required approximately 2 terabytes of data.

Besides changing the encoding from text-based .json formatted files to pickled python objects, any extraneous information, such as extra hypotheses from time steps not needed, present in these save files was also removed. Generating and storing this data takes approximately 16 hours of computation, most of it spent trying to generate more hypotheses to reach the 10,000 number after it disproved hypotheses in its scans. Generating these new hypotheses is an expensive operation because the agent cannot tell, while it is generating new hypotheses, whether a given state of the board is invalidated because of information collected at future time steps causes all states reachable from that state to be invalid. The agent either has to track the entire exponentially growing set of boards or it has to continue to spend computing power to generate new legal hypotheses. The time complexity of the algorithm that the agent uses to generate these new hypotheses is $\mathcal{O}(-nNbi\frac{\ln N}{\ln f})$, where n is the number of hypotheses to generate, N is the total number of hypotheses, b is the branching factor of the game, i is the number of information items available for each turn, and $f \in (0, 1)$ is a parameter to control how far back into the past the algorithm should look in order to generate these new hypotheses. The time complexity can be somewhat simplified as b averages 35, the branching factor of Chess, and i averages 2.4 to $\mathcal{O}(-nN\frac{\ln N}{\ln f})$. This time complexity shows that the amount of time that it takes the algorithm to generate new hypotheses is determined by the number of hypotheses the agent is tracking. The more it is tracking, the more it will end up invalidating each time it receives information, and the more it will have to generate.

Together, the massive amount of data handled combined with the amount of time that the hypothesis generation takes means that training the network using self-play reinforcement learning is incredibly slow. Once the neural network is trained, the agent can be run much faster than it is while training. Not only would a fully trained neural network cause the agent to be better at tracking the true state of the board by picking better scans, but the hypothesis generation algorithm would have less work to do every turn.

However even though the training was very slow and very few self-play games were completed, the agent had learned something. Figure 6 shows a game between our agent and Strangefish. We chose to test the modified agent against Strangefish because we were able to play against Strangefish locally and could remove the time constraint that had been present earlier. Although our agent lost this game, the game reveals that in just a couple of self-play games our agent was able to learn a Chess opening. In our agent's first two moves (Figures 6a and 6c), our agent executes the King's Head Opening, a variant of the Barnes Opening. The King's Head Opening is not particularly strong as it reveals the king to a large portion of the Chessboard, allowing for a direct attack from the opponent's bishops and queen. The use of this opening shows that the agent has learned a very basic understanding of Chess, although it appears to have not yet learned protective behavior. However, it is apparent this understanding does not last very long into the game. In its third move (Figure 6e), our agent tries to perform an illegal move, so nothing happens. Ultimately, the agent loses because it did not detect the presence of the black team's bishop after Strangefish's fifth move (Figure 6j). Except for its first move (when the

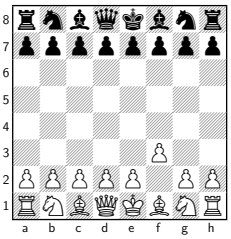
scan cannot provide any information), the agent repeatedly scanned around g7. This choice meant that the agent was unable to see how the board was developing. Although it should have been able to infer that it was put in check during its sixth move (Figure 6k) because it saw that the bishop had gone missing, we deduce that not enough training iterations had passed to learn protective behavior. Although the agent's moves in this game were not very impressive, they show a basic understanding of the start of the game which is impressive considering the agent had completed less than 20 self-play games before this match.

6 Discussion

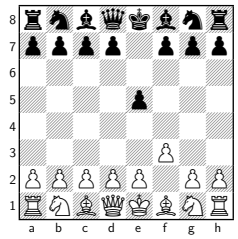
As the computational load of our training framework was extremely high, we provide analysis for the modified agent (as it is continuing to self-play). We expect to be able to defeat Strangefish, the state-of-the-art RBMC bot, with our agent because our algorithm should be able to learn and find a better state-value heuristic than the one Strangefish relies upon [15]. As Strangefish is heavily based on Stockfish, the heuristic function will suffer from a domain shift and perform suboptimality. Our algorithm learns a joint move and scan policy. Thus, our agent should be able to learn to make informed moves and to plan its scans in such a way to gain the most advantages over its opponent. Manually programming to scan for most information vs. scanning to inform upcoming moves was shown to be suboptimal in the Neurips 2019 RBMC competition [14]. Similarly to how AlphaZero was able to learn to defeat Stockfish, our algorithm should be able to learn how to defeat Strangefish, not only because our algorithm should be able to learn how to play RBMC better, but our algorithm should also be able to learn how to extrapolate information and reason about the true board state. Since our algorithm is designed for RBMC rather than adapted from regular Chess, our algorithm will be able to better play towards the intricacies of Reconnaissance Blind Multi-Chess.

Another clear flaw of Strangefish is that it is based on and requires Stockfish [15]. While basing Strangefish off of Stockfish does lend a huge benefit to Strangefish, as its designers don't have to worry about estimating a state-value heuristic, the algorithm cannot be generalized to games without a state-of-the-art agent to bootstrap upon. Unlike RBMC, most imperfect information games do not have variant perfect information games with existing state-of-the-art agents. In order to use Strangefish's method for a different game, both a reduction from the imperfect information to the perfect information game needs to be developed as well as an agent to play the perfect information game. Our framework, on the other hand, requires minimal changes before training can begin to create a state-of-the-art agent for the new game.

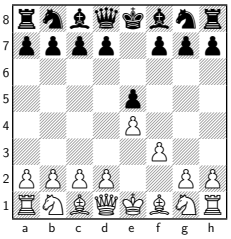
Unlike a hand-engineered method, which can miss out on certain features, a learning-based method will autonomously learn to recognize features and strategies as it collects more data. Our agent will continue to improve as it trains, including strategies more intelligent than can be manually defined by people. Additionally, a learning-based method can avoid accidental biases that might be introduced into it by the people who created the method, such as exaggerating the importance of given features or playing towards very specific strategies. Thus, by reducing biases in the agent and allowing for the agent to learn from its own past mistakes and successes the resulting agent should able to defeat an engineered agent if the agent is given enough training time.



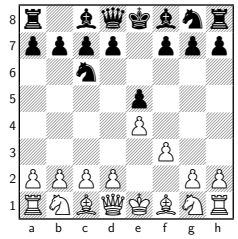
(a) Our First Move



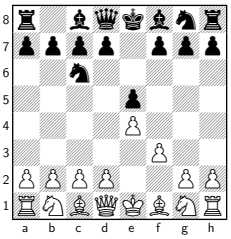
(b) Strangefish's First Move



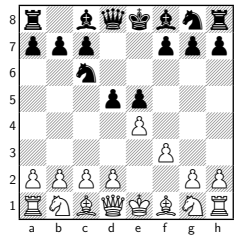
(c) Our Second Move



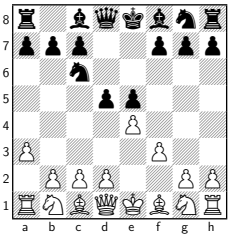
(d) Strangefish's Second Move



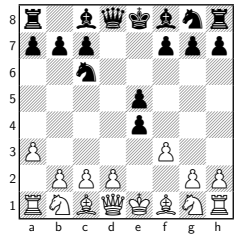
(e) Our Third Move



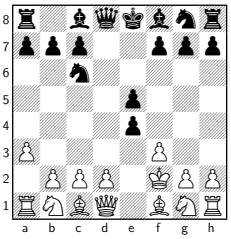
(f) Strangefish's Third Move



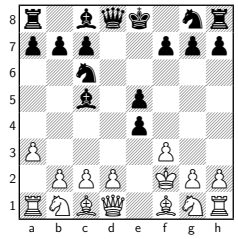
(g) Our Fourth Move



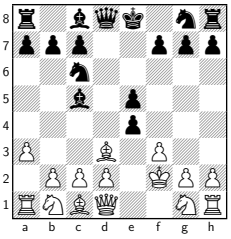
(h) Strangefish's Fourth Move



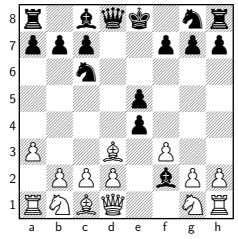
(i) Our Fifth Move



(j) Strangefish's Fifth Move



(k) Our Sixth Move



(l) Strangefish's Sixth Move

Figure 6: This figure shows a game played by the modified agent (white) against Strangefish (black). Chessboard figures created using skak [20].

Although our agent ultimately was unable to defeat Strangefish with the current amount of training time, we have taken many steps to get it to the point where is it today. Our triple-headed neural network architecture, used because our agent needs to choose both moves and scans, allows the network to learn more efficiently and more accurately as it is able to intertwine its move and scan policy. This intertwined nature of the policies allows the network to choose informed moves as well as allowing the network to choose scans that will have the greatest impact on its future moves. This nature also allows the move policy and scan policy to share computations, increasing the rate at which the network learns. In order to facilitate the training of this huge network, we had to design an efficient space-saving structure to store network inputs and training targets. As mentioned in the previous section, we managed to reduce our stored file sizes over 2,000 times. This reduction was done by storing only the exact information required by the neural network during training using a compressed binary file format rather than in a text-based file format. This results in more efficient use of disk space as well as a faster algorithm for self-play and training.

7 Conclusion

Much of the literature for artificial intelligence agents for two-player, adversarial, zero-sum games has revolved around perfect information games. Imperfect information games require a different set of strategies and present different problems to consider, especially for learning-based frameworks. Previous techniques for playing imperfect information games such as RBMC have shown to be extremely specific and rely upon a state-of-the-art Chess agent. While these approaches allow for the creation of a high-performing agent with ease, the underlying assumptions cause many drawbacks and result in a domain-specific agent. In this paper, we take a step at providing a generalized learning algorithm to play imperfect information games such as Reconnaissance Blind Multi-Chess, focusing on tracking states from partial information and learning scanning and movement policies. We apply this method to Reconnaissance Blind Multi Chess, creating an agent that has learned to play RBMC without any outside influence.

While we were unable to provide the breakthrough in performance that we would have hoped for in the study of imperfect information games, we have made many strides towards a general learning-based framework for these games. These strides include our triple-headed neural network architecture to allow the same neural network to make both the movement and scanning decisions, allowing the network to learn the balance between information gathering and taking actions to win the game. Another advancement we have made is directly using our set of beliefs of the state of the board as input into the neural network. This advancement allows the network to learn relationships between beliefs and possible true board states that may be responsible. While these changes did make training the network very inefficient, they have created an agent that can learn to play RBMC given no prior information about the game.

8 Future Work

In order to further improve our method it should be possible to use Recurrent Neural Networks (RNNs), such as Long Short Term Memory networks (LSTMs), in order to encode the sequence of information received into a compact state. By using machine learning-based methods to create an encoder for the gathered information, a compact representation of the current state of the board, based on only the information gathered, can be created and then used in place of our current representation of the beliefs of the board. A compact state representation would allow the policy network to be trained much faster while learning its own representation of the board state. This improvement would allow the agent to better translate the sequence of information received to the current move policy. A learned compact state representation would make it possible for the agent to learn to reconcile past information with newer information and, hopefully, combine them to have new inferred information that was never explicitly given to the agent. This change would effectively compress the current board belief state and remove the need for the hypothesis tracking algorithm, removing the two main obstacles the current agent has towards its training.

References

[1] Stuart J. Russell and Peter Norvig. *Artificial intelligence a modern approach*. 3rd ed. Prentice Hall, 2010.

[2] Volodymyr Mnih et al. "Human-level control through deep reinforcement learning". In: *Nature* 518.7540 (2015), pp. 529–533.

[3] Michael L. Littman. "Friend-or-Foe Q-learning in General-Sum Games". In: *ICML*. 2001.

[4] David Silver et al. *Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm*. 2017. arXiv: 1712.01815 [cs.AI].

[5] Jared Markowitz, Ryan W. Gardner, and Ashley J. Llorens. *On the Complexity of Reconnaissance Blind Chess*. 2018. arXiv: 1811.03119 [cs.AI].

[6] Guillaume Chaslot et al. "Monte-Carlo Tree Search: A New Framework for Game AI." In: *AIIDE*. 2008.

[7] *Stockfish*. 2020. URL: https://stockfishchess.org.

[8] Herbert A Simon. "Theories of bounded rationality". In: *Decision and organization* 1.1 (1972), pp. 161–176.

[9] Murray Campbell, A.Joseph Hoane, and Feng-hsiung Hsu. "Deep Blue". In: *Artificial Intelligence* 134.1 (2002), pp. 57–83. ISSN: 0004-3702. DOI: https://doi.org/10.1016/S0004-3702(01)00129-1. URL: http://www.sciencedirect.com/science/article/pii/S0004370201001291.

[10] David Silver et al. "Mastering the game of go without human knowledge". In: *Nature* 550.7676 (2017), pp. 354–359.

[11] David Silver et al. "Mastering the game of Go with deep neural networks and tree search". In: *nature* 529.7587 (2016), p. 484.

[12] Darse Billings et al. “Game-Tree Search with Adaptation in Stochastic Imperfect-Information Games”. In: *Computers and Games*. Ed. by H. Jaap van den Herik, Yngvi Björnsson, and Nathan S. Netanyahu. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 21–34. ISBN: 978-3-540-32489-8.

[13] Andrew J. Newman et al. “Reconnaissance blind multi-chess: an experimentation platform for ISR sensor fusion and resource management”. In: *Signal Processing, Sensor/Information Fusion, and Target Recognition XXV*. Ed. by Ivan Kadar. Vol. 9842. International Society for Optics and Photonics. SPIE, 2016, pp. 62–81. DOI: 10.1117/12.2228127. URL: https://doi.org/10.1117/12.2228127.

[14] *Reconnaissance Blind Chess*. 2019. URL: https://rbc.jhuapl.edu/.

[15] Gino Perrotta, Robert Perrotta, and tmacksmyers. *StrangeFish*. 2019. URL: https://github.com/ginop/reconchess-strangefish.

[16] Hanna Kurniawati, David Hsu, and Wee Sun Lee. “Sarsop: Efficient point-based pomdp planning by approximating optimally reachable belief spaces.” In: *Robotics: Science and systems*. Vol. 2008. Zurich, Switzerland. 2008.

[17] *Artificial Intelligence*. 2010. URL: https://artint.info/html/ArtInt_230.html.

[18] Julian Schrittwieser et al. *Mastering Atari, Go, Chess and Shogi by Planning with a Learned Model*. 2019. arXiv: 1911.08265 [cs.LG].

[19] *Leela Chess Zero*. 2020. URL: https://lczero.org/.

[20] *skak*. 2020. URL: https://ctan.org/pkg/skak.