Mastering the Game of Chess using Monte Carlo Tree Search

Cooper Golemme cooper.golemme@tufts.edu
Department of Computer Science
Tufts University
Medford, MA, USA

Gabe Schwartz gabriel.schwartz@tufts.edu
Department of Data Science
Tufts University
Medford, MA, USA

Editor: Gabe Schwartz, Cooper Golemme

Abstract

The emergence of Monte Carlo Tree Search (MCTS) as a highly effective learning strategy has significantly advanced the development of artificial intelligence in complex decision-making domains, notably in strategic games like chess, Go and Atari games. MCTS enhances decision-making by simulating numerous potential future scenarios, effectively navigating the vast search space inherent in games like chess. This paper proposes the development of a reinforcement learning that learns to play the complex game of chess using MCTS. We leverage the powerful decision making ability of MCTS and domain knowledge of chess to create a comprehensive chess agent, capable of highly intelligent chess play.

Keywords: Monte Carlo Tree Search, Reinforcement Learning, Chess, Artificial Intelligence

1 Introduction

In recent years, planning algorithms rooted in lookahead search features have achieved remarkable successes in artificial intelligence. Indeed, if the goal of AI is to achieve human like decision making ability, lookahead search is a key feature of human intelligence that an AI should try to replicate. Human chess grandmasters, a rank given to the best chess players in the world, are often thinking many moves ahead to balance choosing the best move for the current game state and setting up future powerful moves. Planning ahead is not limited to the game of chess. Almost all complex games, puzzles and tasks, require some form of planning ahead to achieve success. It is no wonder why in recent years, there has been a great push to integrate lookahead search features into AI agents.

One such lookahead search method is Monte Carlo Tree Search (MCTS). MCTS is renowned for its efficacy in decision-making processes. It excels in exploring vast search spaces by simulating numerous potential future scenarios. By strategically balancing exploration and exploitation of moves, MCTS efficiently narrows down promising paths within complex decision spaces, such as those found in chess.

In chess, the combinatorial explosion of possible moves presents significant challenges for traditional algorithms. However, recent advancements – particularly those leveraging

deep reinforcement learning – have demonstrated promising results by integrating MCTS with neural networks to achieve exceptional performance, as exemplified by AlphaZero (Silver et al., 2017) and MuZero (Schrittwieser et al., 2020). These methods have surpassed traditional chess engines with reinforcement learning driven entirely by self-play.

Along the lines of past work in the area, this paper proposes the creation of a chess-playing reinforcement learning agent utilizing MCTS, optimizing it to play the game of chess.

2 Gameplay and Rules

Chess is a complex game that has been played for centuries. The goal of the game is to take the other player's king while defending your own.

Chess is played on an 8x8 board with alternating square colors. Each player has 16 pieces to start the game: 8 pawns across the penultimate row, two rooks in the corners, two knights inside the rooks, two bishops inside the knights, and a king and a queen on the last two squares in the back row, opposite of the opposing king and queen. The pawns can move forward one square at a time, with 3 exceptions:

1. The first time the pawn is moved, it can be moved two squares instead of 1.
2. If an opposing piece is diagonal, the pawn can take it. It cannot take pieces directly in front of it.
3. If a pawn reaches the back row on the opposite side of the board, it promotes to the player's choice of knight, rook, bishop, or queen.

The knights move in an "L" shape, meaning in any direction, it goes 2 spaces forward and one to either side. It is the only piece that can leap over other pieces. The bishops move diagonally, and one is always on light squares while the other is always on dark squares. They can move as many squares as are available to move in one chosen direction. The rooks move horizontally and vertically, and can similarly move as many squares in a chosen direction as are available. The queen can move both diagonally and horizontally, granting it every move that the rook or the bishop could make. Finally, the king can move in any direction, but only one square at a time.

Additionally, the king can "castle" with one of its rooks before either piece moves. To do this, the knight and bishop (and queen if on long-side) must have already moved. The rook moves two squares towards the center and the king jumps over it to the column directly on the outside. For example, if the back row is: [rook, empty, empty, king, ...], castling would result in [empty, king, rook, empty, ...].

When one player moves a piece such that the next move could take the king, the opponent's king has been put in "check." By rule, that player must react to the check or else the game is over. They can react by taking the opponent's attacking piece, by blocking the path of the attack, or by moving their king. A player may not move their king into check, or they will lose the game. If a player puts the opposing king in check and the opponent has no

possible moves that result in them escaping check, then they are said to be in "checkmate," which indicates the end of the game.

3 Environment

The environment that we will use will be a chess engine implemented on top of Open AI Gymnasium (Gymnasium). Gymnasium is an open-source toolkit for developing, comparing, and benchmarking reinforcement learning algorithms. It provides a simple API to generate steps, episodes, rewards, and states. This will vastly simplify our learning environment.

The Gymnasium implementation gives us an 8x8 board where black and white pieces are labeled by 1 thru 6 for white, and -1 thru -6 for black, with each numerical value corresponding to a piece type. With this, the environment will keep track of all legal moves for each piece such that any move requested by an agent must pass a legality check.

4 AI Agent

4.1 Monte Carlo Tree Search

Monte Carlo Tree Search(MCTS) is a popular algorithm to make optimal decisions in games where future outcomes can be uncertain and complex, but can be simulated (Sutton and Barto, 1995). It attempts to balance exploration of new moves and exploitation of known moves. It has previously been used in state of the art reinforcement learning algorithms to play complex games like Chess, Go and Atari (Schrittwieser et al., 2020).

MCTS is executed when the agent encounters new state to select a new action for the given state. Below, we show the four phases of MCTS and explain them in detail.

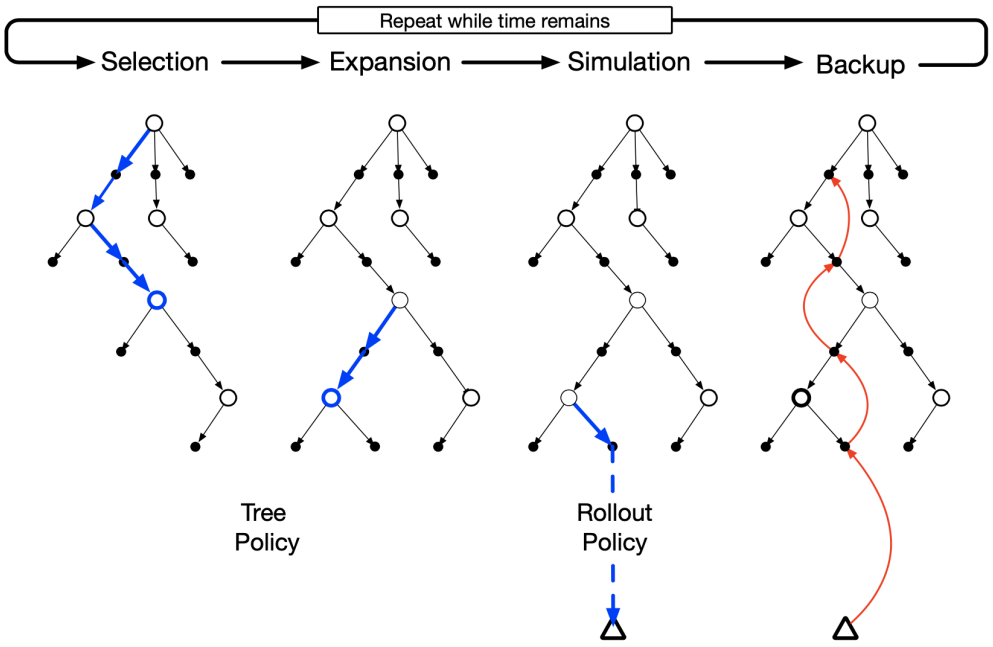

Figure 1: Figure showing the phases of Monte Carlo Tree Search: Selection, Expansion, Simulation, and Backup. We select actions according to a *tree policy*, expand the tree with new actions, follow a *rollout policy* when we reach a leaf, and update our action value estimates up propagating returns back up the tree.

Each iteration of MCTS consists of four steps:

1. **Selection.** During selection, we start from the current position at the root node of the tree. We then traverse the tree by repeatedly selecting child nodes until we reach a leaf node. We traverse by following a *tree policy*. In our application, we will select nodes according to the Upper Confidence Bound (UCB) formula. We need to shape the UCB formula for this context. The general formula for a given node i looks like:

$$UCB_i = \text{win rate of node } i + C\sqrt{\frac{\ln(\text{parent visits})}{\text{visits to node } i}}$$

2. **Expansion.** During expansion, the tree is expanded from the selected leaf node by adding one or more child nodes. This corresponds to adding new actions for the agent to explore in the future.

3. **Simulation.** From the selected node or the nodes added in expansion, we simulate a complete episode according to the rollout policy. From this, we get a Monte Carlo trial, where we first select actions in the tree according to the UCB policy and then generate an episode outside of the tree by the roll out policy.

4. **Backup.** From this generated trial in the simulation step, we have some reward. In the backup step, the reward from this trial is backed up the tree to update (in the

case where we simulated from selected node in step 1) or initialize (from expanded node in step 2) the action values in the tree. For the actions generated beyond the tree, no values are updated or stored. A visualization of this is shown in Figure 1.

With MCTS, we continue these 4 steps starting at the tree's root to some time horizon T or until some other threshold is reached. When MCTS finishes, an action from the root node is selected which either: (1) has the highest visit count, (2) has the highest action value estimate, or (3) some combination of the two. After selecting this action, the agent the proceeds to the next action, performing MCTS over again to select a new action. We will experiment with different ways to select this action to best fit our application to playing chess.

4.2 Brief Demonstration

To visualize how MCTS will be applied in the chess context, we will breifly outline what each step in the algorithm will look like in terms of a chess game.

1. Selection: MCTS first selects a promising sequence of moves (e.g., pawn move, then knight move).
2. Expansion: It then tries a new move from that point, expanding its knowledge (the tree).
3. Simulation: It randomly simulates playing out the game until the end (e.g., win or loss), getting a quick idea for how promising the new move is.
4. Backup: If the simulated game ends in a win, the path of moves leading there becomes more attractive, influencing future searches.

With this approach, we hope to build a competent chess agent.

5 Evaluation

To test and evaluate our agent, we will first set up the environment using OpenAI Gymnasium described in the environment section. We then will initialize two agents to play against each other and train them using MCTS. Finally, we will evaluate their performance against other known chess agents.

5.1 Setup

We will define and create the chess environment as outlined in the environment section. We will initialize agent parameters using the starting state of the board and starting values for its learned actions.

5.2 Training

We will train the agents using MCTS, performing selection from the current board state when it is the agent's turn, expanding moves, simulating an episode of the game, and finally backup to evaluate the performance of that move. After a training game is complete, we

will pin the new trained agents against each other again in a new chess game and repeat this process.

5.3 Testing

We will save agent versions at certain training iterations to benchmark model performance to ensure we are learning in our environment. We will play games with new versions of agents versus older versions. We will compare our agent against other known chess agents to evaluate our training methods.

6 Timeline and Responsibilities

6.1 Timeline

- **March 24**: Set up chess environment using the packages we outlined
- **End of March**: Implement MCTS and run a few training simulations
- **Mid April**: Fine tune model and train on many episodes
- **End of April**: Evaluate model performance through benchmarking and comparisons
- **May**: Write up report on our efforts.

6.2 Responsibilities

- **Cooper Golemme**
  - Build MCTS agent
  - Tune agent parameters
  - Run simulations
  - Contribute to report
- **Gabe Schwartz**
  - Build chess environment
  - Make sure agent interfaces with environment
  - Tune agent parameters
  - Run simulations
  - Contribute to report

References

OpenAI Gymnasium. OpenAI gymnasium. https://gymnasium.farama.org/index.html. Accessed: 2025-03-14.

Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, Timothy Lillicrap, and David Silver. Mastering atari, go, chess and shogi by planning with a learned model. *Nature*, 588(7839):604–609, December 2020. ISSN 1476-4687. doi: 10.1038/s41586-020-03051-4. URL http://dx.doi.org/10.1038/s41586-020-03051-4.

David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy P. Lillicrap, Karen Simonyan, and Demis Hassabis. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *CoRR*, abs/1712.01815, 2017. URL http://arxiv.org/abs/1712.01815.

Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, 1995. ISBN 9780262039246. URL http://creativecommons.org/licenses/by-nc-nd/2.0/.