

# Mastering Chess through Reinforced Deep Learning

## Cpts 434 | Project Report

### Team Members (Group 12)

(1) Reagan Kelley (ID: 11663871)

### Abstract

Chess is arguably the most studied domain of AI, with decades of experimentation, innovation, and new feats. In the last decade, chess engines have achieved new heights in performance using deep neural networks and reinforced learning. This is the focus of my project. I have chosen to partake in this project for a few reasons. Given my beginner-to-intermediate experience with neural network applications, this problem is both challenging but also researched enough that I think it will be an incredible learning opportunity. It will give me hands on experience with reinforced learning techniques and deep learning. I love chess, and I have developed my own chess application that I would like to plug an AI into. I don't expect my AI to outperform the top engines, but my goal is to create a chess AI that can beat me.

### Introduction

Chess has long been a point of interest in the machine learning community. It was first conquered by Deep Blue in 1997, when it became the first computer to beat a human opponent, world champion Gary Kasparov<sup>1</sup>. Since then, machine learning has taken off to achieve newer and more impressive feats that stretch far beyond human capabilities. Today, chess computers face off against each other rather than against human opponents in a continuous battle to see which AI is the best.



Recently, deep learning has brought chess to a new frontier in terms of precision and innovation. Past generations of chess AI has relied on transposition tables, immense databases of past games, chess opening game theory, and endgame tables. AlphaZero, Google's deep learning artificial intelligence, did not rely on any of these techniques. Rather, it learned the game of chess on its own through self-play and reinforced deep neural networks<sup>2</sup>. Deep learning now

---

<sup>1</sup> Newborn, M. (2012). p. 287. In Kasparov versus Deep Blue: Computer Chess comes of age. essay, Springer.

<sup>2</sup> Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., Lillicrap, T., Simonyan, K., & Hassabis, D. (2017, December 5). Mastering chess and shogi by self-play with a

dominates the competitive sphere of chess engines, consistently beating once superior engines like Stockfish.

In this project, I wish to use these new methods of deep neural networks and reinforced learning to build a new chess AI that can perform at a chess Elo rating of at least 2000. I have chosen this benchmark because this ranking is right above my own Elo rating, and I want to create a chess AI that can beat me. By FIDE expectations, this also would put the AI at a skill level of candidate masters (CM), the lowest awarded title in competitive chess<sup>3</sup>.

## Literary Review

Most versions of board game AI utilities some decision tree structure, where nodes represent game states and edges represent moves taken given a game state. Game states are then valued according to future game states and given a value; an AI will make the most optimal move according to that value. This works great with games with perfect information and zero randomness like tic-tac-toe, checkers, chess, and go. However, games like chess have an ungodly amount of possible game states, of which makes classic decision tree algorithms cumbersome. One of the most common decision policies, Minimax, uses future state information to minimize loss in worst case scenarios. In other words, it provides the best move assuming an opponent uses perfect play. The problem is that in games like chess with  $10^{120}$  possible game states, this is too computationally expensive. While there are many optimization methods to improve this in chess, such as alpha-beta pruning, the daunting size of the chess game tree has plagued common game AI strategies.

If we are going to look at how to apply new methods, especially ones that utilize neural networks, we must look at AlphaZero. Simply put, AlphaZero used an amalgamation of machine learning techniques to outperform chess AI that utilized optimized Minimax programs such as Stockfish. AlphaZero did utilize a game state tree, but not with Minimax rather a variation of the Monte Carlo tree search (MCTS)<sup>4</sup>. The key in AlphaZero is that it used two neural networks to do heuristic searches within the chess search space. It used a ‘value network’ to evaluate board positions and a ‘policy network to select moves. These two are neural networks, more specifically convolutional neural networks, of which when combined can create a game playing agent.

The engineers at DeepMind, the creators of AlphaZero, were able to train the chess AI using reinforced deep learning (Deep RL). There are many forms of Deep RL, but all utilize a neural network agent that learns to make decisions through trial and error. AlphaZero was the first AI to be able to teach itself the game of chess through self-play. In training, on every move AlphaZero would simulate a hyperparameter number of possible games forward. In each of these mini games the AI would choose the most productive move, meaning it strikes the best balance

---

general reinforcement learning algorithm. arXiv.org. Retrieved February 18, 2023, from <https://arxiv.org/abs/1712.01815>

<sup>3</sup> Elo, 1978, p. 18

<sup>4</sup> Silver, D., Huang, A., Maddison, C. et al. Mastering the game of Go with deep neural networks and tree search. Nature 529, 484–489 (2016). <https://doi.org/10.1038/nature16961>

between exploration and exploitation (a heuristic model set by the value and policy network). Exploration is AlphaZero's main strategy for discovering new game states and expanding its dataset of training data. Eventually, these simulations will end in a win, loss, or draw. These become the labels of the game state, of which will be the features of a new data point. With this, the algorithm can backpropagate through the two networks and fine tune them. Through enough simulations, the AI will uncover similar game states of which it can measure moves for high exploitation, that being it is a promising move for victory.

## **Technical Plan**

I intend to create a chess AI that utilizes Deep RL and the utilization of MCTS. Immediately there is some early steps that need to be taken to avoid problems in the future. First, I would like to use my chess application that I have coded in C#. I want to use this mainly for ad hoc testing purposes and as a graphical aid, of which I can use to either test the neural network directly or replay previous simulated games. A majority of the machine learning implementation will be done through python as it is the most supported for machine learning.

Already, there are some impending limitations to my project. First, I lack the hardware used by DeepMind to simulate its high volume of games for training. I expect a significant portion of this project will be spent on optimization of these machine learning strategies on normal hardware. I also expect there to be many complications with hardware that may lead to unexpected crashes during training sessions. Thus, I will need to implement a safeguard feature that can consistently benchmark progress and maintain working models.

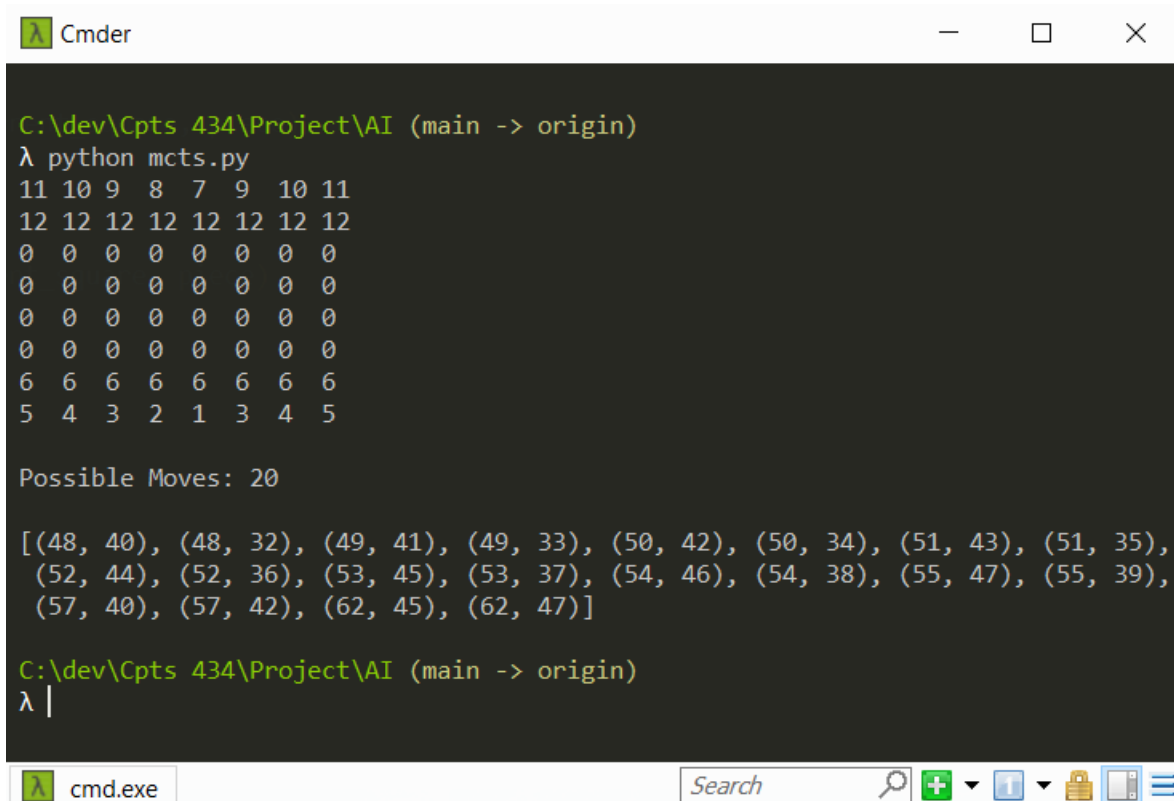
The plan is to use the steps by AlphaZero and the community to get started in building a stable network for chess learning. As the project advances, I will add my own hyperparameters or tertiary machine learning techniques to boost performance or fine-tune problems that may arise from my unique implementation.

## **Intermediate Results**

### **Building the Infrastructure:**

In order for to create a model to play chess, we need to build an environment for our AI to learn. Our program is going to need a few things before it can begin training. Since we are going to be utilizing a Monte Carlo tree structure, we need to incorporate the rules of chess into the nodes, which represent the game state. Each node will need to hold various sorts of data. It will need to know the current state of the board such as the coordinates for each piece. It will need to know whose turn it is, and finally, it will need to know special parameters unique to the game of chess. These are parameters such as whether or not the king is in check, if a piece is in a royal pin (therefore it cannot move), and whether or not the king has moved yet in the game, because if so, would not be able to castle. Thus, in addition to a grid, it will need store these extra parameters.

Each MCTS node will need to know the state, but it will also need to know what are the legal moves in that position, because this will dictate the current node's possible children. So, for each node there must be a way to compute all legal moves from a given position. This will give the AI a list of options to choose from and through the policy network, will choose the most optimal move.



```
C:\dev\Cpts 434\Project\AI (main -> origin)
λ python mcts.py
11 10 9 8 7 9 10 11
12 12 12 12 12 12 12 12
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
6 6 6 6 6 6 6 6
5 4 3 2 1 3 4 5

Possible Moves: 20

[(48, 40), (48, 32), (49, 41), (49, 33), (50, 42), (50, 34), (51, 43), (51, 35),
(52, 44), (52, 36), (53, 45), (53, 37), (54, 46), (54, 38), (55, 47), (55, 39),
(57, 40), (57, 42), (62, 45), (62, 47)]

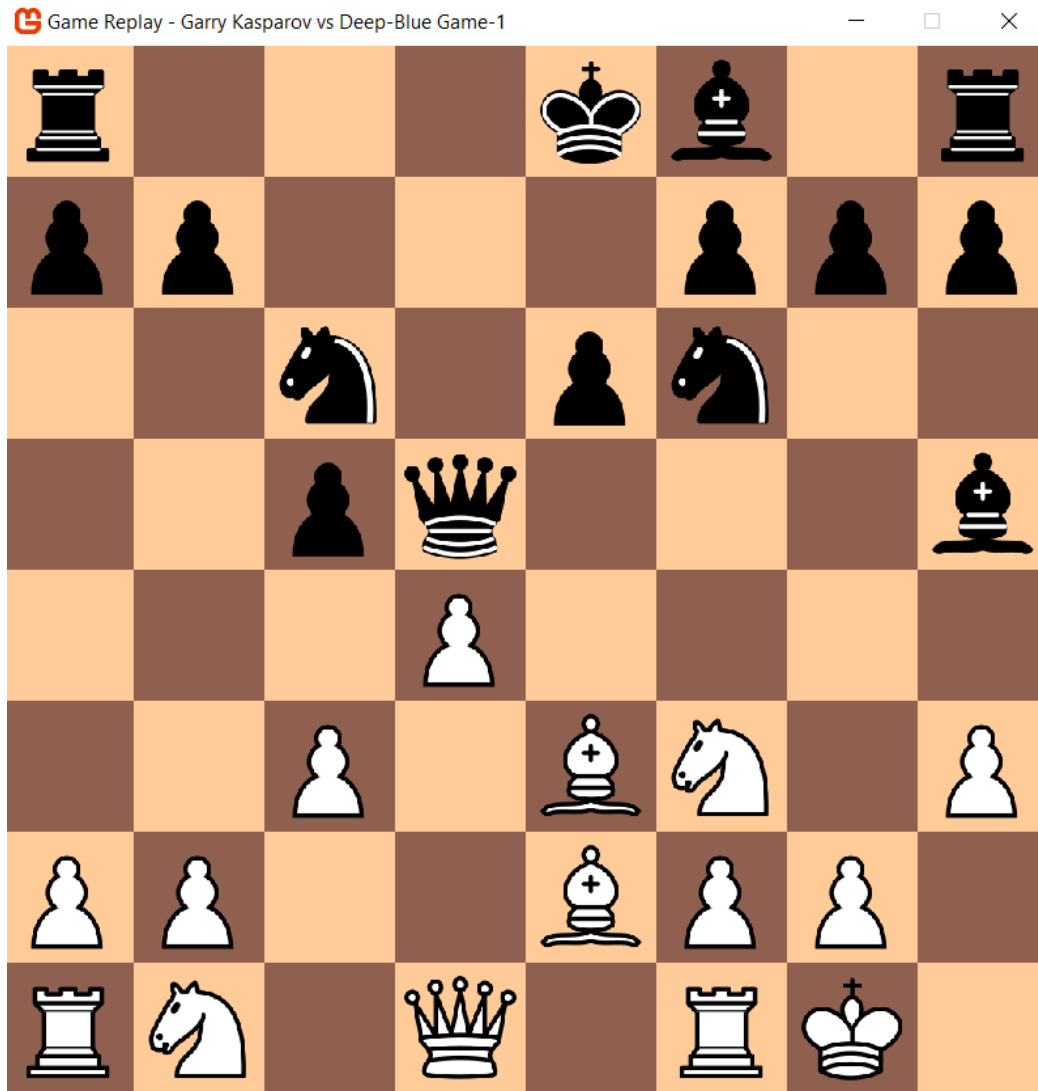
C:\dev\Cpts 434\Project\AI (main -> origin)
λ |
```

The code execution above shows some of the contents of a node holding the current game state. It shows an 8x8 grid, where each value corresponds to a piece ID. For this given state, it also shows all the legal moves for the position. In this example, the game has just started and it is white's turn. The code returns a list of pairs indicating (from\_index, to\_index), or in other words, a piece to move from an index in the grid to a different index in the grid.

Visualize the Model:

We have coded the rules of chess into our Monte Carlo search tree, but we still need a way to evaluate how the model is improving as it trains. In the future, we will utilize various chess statistics. We will also likely import other chess computers to test our current models of which in turn can provide us an accurate ELO rating. However in these early stages, we can use our knowledge of chess to get a sense of improvement by simply watching its simulated games. To do this, we will need to replay the game through a graphical application. We coded a chess application in C#, which can import games with portable-game-notation (PGN) files. When our

model simulates a game, we can save that game through a PGN file. We can then launch our chess GUI and replay the game and judge its performance.



The graphic above shows a demo of our chess GUI. We imported Kasparov's game against Deep Blue and the application above allows us to rewatch through our project.

## Future Work

We are just about finished with the foundation for the project. We have created an environment that understands the rules of chess, with a Monte Carlo search tree. Our next step is to use our MCTS structure, and create two neural networks, a policy and value network, which will be used as a heuristic to traverse and build the tree. We think that the best option is to use a CNN for the value network, but a residual network for the policy network. This is because the value network's job is to evaluate the state of the board and provide a score, which means that it receives game state input such as the 8x8 grid. We can use this grid as we have done with

pictures, where the CNN can recognize patterns in this grid as if it were pixel data. For the policy network, we decided to use a ResNet structure for two reasons. The first is because we are not taking grid-like data from the game state for input, but the available moves and determining the best option. Secondly, inspired by AlphaGo, who used a residual network with their model to play Go, we think it would be wise to follow in their direction to utilize a ResNet to play chess.

As we connect our chess program and the MCTS structure to the two networks, we will continue to run and train the model throughout the remainder of the semester, periodically saving the model, and observing its progression in play through its past games. If we encounter poor performance, we will fine-tune our approach and report it in our final report.

## References

Elo, 1978, p. 18

Newborn, M. (2012). p. 287. In Kasparov versus Deep Blue: Computer Chess comes of age. essay, Springer.

Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., Lillicrap, T., Simonyan, K., & Hassabis, D. (2017, December 5). Mastering chess and shogi by self-play with a general reinforcement learning algorithm. arXiv.org. Retrieved February 18, 2023, from <https://arxiv.org/abs/1712.01815>

Khovanskiy, M. (2022, August 25). AlphaZero chess: How it works, what sets it apart, and what it can tell us. Medium. Retrieved February 19, 2023, from <https://towardsdatascience.com/alphazero-chess-how-it-works-what-sets-it-apart-and-what-it-can-tell-us-4ab3d2d08867#:~:text=In%20short%2C%20AlphaZero%20is%20a,the%20rules%20of%20said%20games.>