Chess AI Project Report

Eric Lykins, Jacob Seikel, and Remington Ward

Introduction:

For our final project, we chose to attempt to solve the problem of creating an AI to play chess. This problem has been tackled many times before, sometimes to great success. Some open source examples of high-level chess engines include Stockfish and Leela Chess Zero [1], [2]. Our goal was to try to replicate the success of these engines, although we knew we would not have the resources to truly compete with them. Instead, this project was meant to be more explorative; we planned to try several different approaches and take notes on which ones were most successful.

The first approach we took was to create an AI that operates using a heuristic function to evaluate board states. This approach also entails using a search tree to iterate through potential move branches. We ultimately used python-chess and Pygame for the testing environment. The other approach we used was to build a chess AI using reinforcement learning. We used many libraries for this part, including PyTorch, NumPy, CUDA, SciKit-Learn, and cuDNN.

Related Work:

There have been many studies related to chess AI, so we will just cover a sample of them here. Maharaj, Polson, and Turk measured the effectiveness of Stockfish and Leela at solving Plaskett's Puzzle, a well-known chess endgame puzzle. Stockfish is a search-based engine, while Leela uses a neural network approach. They found that Stockfish was able to more easily find forced checkmates, although Leela needed to search significantly fewer nodes once it was given enough help to find the checkmate. They noted that each approach had its benefits, so neither one truly eclipsed the other [3].

Gaessler and Piezunka studied the benefits of using AI as a learning tool and training partner for decision-makers. Specifically, they studied how very skilled chess players can use AI to help them improve, since there are few other humans who can match their skill level. They found that it was in fact a helpful self-improvement tool. However, it is not a perfect substitute for human players, since it does not ever make human mistakes. Therefore, it does not help humans learn to exploit other players' mistakes [4].

McIlroy-Young, Sen, Kleinberg, and Anderson developed a chess AI that is specifically designed to play against human opponents. It is much better at predicting human moves than most high-level chess engines, and can also take a player's skill level into consideration. In other words, it is able to predict its opponent making mistakes, which Stockfish and Leela do not do. They conclude that this approach could be generalizable for designing AI systems for human collaboration [5].

Approach:

As mentioned earlier, our project was divided into two main approaches: a heuristic-based search tree approach and a machine-learning approach. This section is divided into subsections which describe each approach and its results.

Approach #1: Search Tree

We started by doing some basic research to determine a good way to implement this approach. Eventually, we settled upon creating a minimax tree. This method begins with defining a heuristic function which evaluates a board and returns a number which estimates the advantage of one of the players. The agent evaluates the board state after each of its moves, then each of its opponent's subsequent moves, and so on. The key assumption of a minimax tree is that each player will make the optimal move at every decision (according to the heuristic function). In other words, the agent always tries to maximize the heuristic value and assumes that the opponent will always try to minimize it. In this way, I was able to create a general AI class which took a heuristic function and a search depth as input. This architecture made it relatively easy to generate new AIs once it was set up.

The first heuristic function we created simply returns a constant, mainly for testing. This results in the AI viewing every move as equal, which causes it to make random moves. After this, we created a more reasonable heuristic which sums up each side's pieces using the traditional values (pawn = 1, bishop = 3, knight = 3, rook = 5, queen = 9). For the rest of this report, we will refer to this as the basic heuristic. It then tries to maximize the difference between the two sides' material sums in its favor. This heuristic performed relatively well, so every following heuristic was created as a variant of this one. We accomplished this by first starting with the basic heuristic, which returns an integer, then adding or subtracting small decimal values depending on certain aspects of the position. In other words, these AIs followed the basic heuristic, then used a tiebreaker to select from the viable moves. A summary of all the heuristics is listed in the table below.

Heuristic Name	**Description**
Random	Makes random moves.
Basic	Prioritizes keeping its pieces while taking the opponent's.
Trader	Prioritizes taking even trades when they are available.
Attacker	Prioritizes controlling many squares and threatening enemy pieces.
Pawn pusher	Prioritizes pushing its pawns forward.
Piece pusher	Prioritizes pushing all of its pieces forward.

We ran into several difficulties while trying to implement this approach; mainly, our environment was prohibitively slow. We were able to somewhat alleviate this problem by switching from a custom chess environment to using the python-chess module, which is more optimized. We also added alpha-beta pruning, which is a method of eliminating branches from the search tree while still reaching the optimal solution. Additionally, we partially integrated C using Cython, which sped up the program execution due to using static typing instead of Python's normal dynamic typing. While these changes did markedly improve our environment speed, it remained an issue for the duration of the project. In retrospect, it would probably have been better to initially create the environment in a faster language and integrate more bit logic. Additionally, if we had set up the engine to use the universal chess interface (UCI), we could have interfaced it with many different websites for testing.

Search Tree Results:

As mentioned previously, our environment was significantly slower than we would have liked, which prevented us from getting a large amount of data. However, we still managed to have each of the AIs play each other ten times at depth three and twice at depth four, which was the highest depth we could reasonably manage. Note that in this context, depth four means both players each move twice. Therefore, the AIs were actually looking two moves ahead. The results of these games are shown below.

	Random	Basic	Trader	Piece pusher	Attacker	Pawn pusher
Random	Black won	White won	White won	White won	White won	White won
Basic	Black won	Black won	White won	White won	White won	Black won
Trader	Black won	White won	White won	White won	White won	Draw
Piece pusher	Black won	Black won	White won	Black won	Draw	Black won
Attacker	Black won	White won	Black won	Draw	Draw	Draw
Pawn pusher	Black won	Black won	Black won	Black won	Draw	White won

Although we were not able to collect as much data as desired, we can still analyze what we have. At depth four, the random AI was not able to win against any non-random opponent. This shows that our heuristic method is at least somewhat successful. The rest of the AIs are roughly even, although the more aggressive ones seem to have a slight edge.

Anyone who wants to examine this part of our code can look in the “code/ai” directory. All of the related files are in that folder.

Approach #2: Machine Learning

For the machine learning approach, we started by researching different deep learning algorithms. After looking into different algorithms, we settled on a Convolutional Neural Network (CNN). Due to how complex chess strategies become, we decided to use a CNN because this algorithm is known to have strong pattern recognition performance. We decided to set up our CNN so that it could take a board state as input and produce a chess move as output.

The first step we took to setting up a CNN was to set up our data environment. We sourced our dataset of chess game records from Kaggle. After acquiring the data, we used the pandas library to load and clean it. To clean the data, we filtered out games that were terminated by time forfeiture and dropped all columns with unnecessary data (player names, event details, etc.) in order to allow our CNN to focus on only the board and the moves. So our CNN could understand the data, we encoded each move as a unique integer value and each board as an 8x8 array of integers, with each number representing the piece type at that location on the board.

The training process included feeding our clean data into the CNN model and adjusting parameters and hyperparameters to optimize performance and accuracy. As we increased the data and the complexity of our model, the time taken to train the model increased dramatically. To alleviate this struggle, we opted to perform the training calculations using GPU acceleration.

After completing training, we set up an interface between our chess environment and our CNN model, allowing the model to play games.

Machine Learning Results

Our CNN model performs at the same level as the Random heuristic search tree. Our model is distinct from the Random heuristic because it demonstrates a preference for moving its king and shuffling bishops back and forth between two squares. Seeing these strategies emerge gives us valuable insight into the potential of utilizing deep learning to find strategies that are not immediately obvious to a human. In the future, we intend to supply the model with the evaluation of the heuristics mentioned in the Search Tree section. We believe this will provide enough information to our model to allow for new and better interesting strategies to emerge.

Conclusion:

In this project, we took on the task of creating an AI to play chess. We tried two approaches: a heuristic-based search tree and a machine learning model. We encountered both successes and failures, and learned valuable lessons along the way. One of the important lessons we learned was to prioritize setting up an environment which can run quickly, since that will make it much easier to work with and collect data from AIs. We also learned that keeping the machine learning code simple makes it much easier to work with. In the future, we may look into combining both of these approaches by integrating the trained model into the heuristic function of the search tree. We would also like to test the search tree approach with higher depths to further study the differences between different heuristic functions.

References:

[1] Stockfish (2024) Stockfish [source code] https://github.com/official-stockfish/Stockfish.

[2] LeelaChessZero (2024) Lc0 [source code] https://github.com/LeelaChessZero/lc0.

[3] S. Maharaj, N. Polson, and A. Turk, "Chess AI: Competing Paradigms for Machine Intelligence," *Entropy*, vol. 24, no. 4, April 2022. [Online]. Available: https://doi.org/10.3390/e24040550.

[4] F. Gaessler and H. Piezunka, "Training with AI: Evidence from chess computers," *Strategic Management Journal*, vol. 44, no. 11, pp. 2724-2750, May 2023. [Online]. Available: https://doi.org/10.1002/smj.3512.

[5] R. McIlroy-Young, S. Sen, J. Kleinberg, and A. Anderson, "Aligning Superhuman AI with Human Behavior: Chess as a Model System," *ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*

Appendix A: Jacob Seikel

In this project, I primarily worked on the search tree approach to our problem. This includes creating the AI class and all of the heuristics, as well as a file to test all them. I also took several approaches to optimizing the process of searching through the search tree; my most successful optimization was implementing alpha-beta pruning.

My biggest regret with this project is not optimizing the environment further. As previously mentioned, we made this project in Python for its ease of use. However, this decision caused it to be much slower than it could have been. We did find examples of chess environments that had been written in C++ and used bit logic to represent board states. These types of environments have significantly less overhead and can therefore run much faster. However, by the time we considered this as an option, it was too late to switch.

Despite encountering difficulties with the speed of our environment, I still feel that we had positive results. At the highest depth we tested, my custom AIs beat an opponent making random moves 100% of the time. Additionally, when I played against some of the AIs for testing purposes, I struggled to beat them. I am not a particularly strong chess player, but I was still a little surprised that it was actually somewhat difficult to win against them. Although they did not necessarily have a good high-level understanding of the game, it was still difficult to beat them in tactics since they were viewing every possible outcome at once.

I think I learned a lot from this project. First, I had not done much coding in Python before, so this project has helped me improve my skills in this language. I also learned a good way to structure an AI of this type. Creating the general AI class proved to be a good architectural decision, since it meant that creating new AI instances was very simple. Additionally, I learned several different optimization techniques and tools, including alpha-beta pruning and Cython.

Appendix B: Remington Ward

This project was a very informative experience for me. I was responsible for setting up the environment for us to use. I developed the chess game logic in python before anyone else touched the project. I set up the way the AIs would interface with the game and I created the display and interaction using Pygame.

I jumped into creating the chess logic very early. I chose python as the programming language because I am familiar with it and it is easy to work with. The environment worked well manually playing games, but when we created a basic AI, it became apparent that the environment was too slow. I spent a lot of time optimizing the chess logic to make it faster. I managed to get chess moves generating about 5 times faster than before, but that was still very slow.

Instead of creating an environment in python, I should have used C++ to create a basic interface using UCI. UCI stands for the Universal Chess Interface and is a standard way for chess engines to communicate with GUIs. If I had instead of creating a chess environment I had set up UCI, we could have used a pre-existing chess GUI and even had our engine play other ones online. By the time I had realized we should use UCI and C++, it was too late to redo our code. We had to run our AI's at a lower depth in order to achieve a realistic runtime.

Outside of creating the environment, I helped both Jacob and Eric when they encountered trouble. I found the datasets for the machine learning section of the project and I set up the code to download them. I assisted Jacob with creating and optimizing the AIs.

This project taught me not to jump into coding too early. I should have done more research early on. That would have helped me decide on a better environment for our engines to run in. Python is a very helpful tool and was the correct call for the machine learning section, but it proved to be too slow for chess AIs to run at a good depth.

Appendix C: Eric Lykins

For this project, I mostly worked on the machine learning approach to our problem. I extracted and cleaned the data, set up GPU acceleration, created the convolution neural network, and tuned its parameters.

I learned a lot from this project. From preparing the data, I learned that it is very important to carefully consider which parts of the data are important and which are not. When working with a large dataset, every time we changed our minds about what data we wanted to include, we had to wait for a significant amount of time for the raw data to be processed according to our new standards.

Setting up a very simple neural network was originally quite easy. After investigating several python packages for machine learning, most of the difficulty in setting up a neural network was abstracted away.

Before setting up GPU acceleration, I had no knowledge of how to do so. One of the hardest lessons I learned from teaching myself how to use GPU acceleration was to stop and look ahead. Trying to set up the GPU acceleration involved downloading several different programs from different locations, which each had many different versions that only worked with certain other packages, some hardware, and with some GPU hardware. Several days were spent trying to determine where out of memory errors were coming from, and it was due to a mismatch in program versions.

When training the model, I discovered that changing any of the input parameters was made very difficult due to how the data was being packaged and sent to the GPU. Most error messages were hidden by the GPU, making debugging very difficult. Due to this, I learned how to handle errors in python in a much more robust way than I knew how to do before.

Tuning the parameters gave me a deeper understanding of what exactly is happening under the hood of machine learning models. This project led to me learning about how batch normalization, back propagation, batch size, and learning rate all impact a model’s loss and accuracy values.