Zero Lookahead Chess Engines vs. Conventional Chess Engine Strategies: An Analysis

Owen Jaques Sadie Johansen Braeden Copeman Nick Wurzer Dhowa Husrevoglu

Faculty of Engineering and Computer Science
University of Victoria
{owjaques, sadiekayjohansen, btfcopeman, nickdwurzer, dhowa1}@gmail.com

Abstract

Chess engines have long surpassed humans in their ability to play chess, but creating one still proves to be an interesting problem. In this paper, two approaches to creating a chess engine are explored: zero lookahead chess engines, which given a board state return an evaluation of that board, and tree-based chess engines, which use search trees to look ahead into a game to predict the optimal move. An approach which combined these two approaches was also explored. Tree based chess engines were proven to be the most effective at playing chess however, some suggestions are made to improve zero lookahead chess engine methods.

1 Introduction

There are 20 possible opening moves in chess and the problem quickly becomes computationally intractable with a lower bound of 10^{120} possible sequences of moves for a game [1]. For this reason, a chess engine needs to have rules for making the next move rather than storing evaluations of all the possible board states. This creates a very interesting problem for programmers who enjoy playing chess. This study aims to test how effectively zero lookahead chess engines, which play chess without using a tree to determine the best move, can perform compared to more traditional tree-based lookahead algorithms. Zero lookahead algorithms were initially tested separately from tree-search algorithms with heuristics, before combining the two types of models to form a hybrid approach. For implementing the model used by the zero lookahead chess engines we tested an SVM and multiple neural networks. Monte Carlo Tree Search (MCTS) and MinMax tree search algorithms were implemented in a rudimentary form as a basis before testing additional features such as UCT and AlphaBeta with classical heuristics. AlphaZero's hybrid MCTS was then the inspiration in combining these tree search methods with the machine learning models we tested.

2 Problem definition

If the pieces and their positions on the chess board represent a board state, then a board's state could be classified as advantageous for white or black by some degree. All board states one move away from the current board state could be evaluated and the move yielding the most advantageous position could be chosen. The success of this algorithm depends largely on how accurately the board states are classified as advantageous or disadvantageous. One way to classify board states may be to use heuristics such as which pieces are on the board. Another way might be to train a machine learning model which when given a board state learns to predict the chance of that game being won or lost. These two algorithms may have some success but are limited because

they only look at board states one move away. Which color has the advantage can quickly change because the opponent's moves are unpredictable. The algorithm could potentially improve one of these two approaches by looking farther in advance (without evaluating all possible moves). These two concepts of evaluating board states and looking ahead to find the most successful sequence of moves were studied in an attempt to create a strong chess engine.

3 Background and related work

In Sabatelli *et al*.'s publication on evaluating chess positions with limited lookahead, they explored using multilayer perceptrons (MLP) and convolutional neural networks (CNN) to evaluate chess positions [2]. As input, each model was given a 1-hot encoded feature vector of the board, which has 768 features. The best model they developed was a three layer MLP. It was able to achieve a mean squared error (MSE) of 0.0016 on their test set [2].

In 2015, Matthew Lai developed the Giraffe chess engine, which uses a neural network to evaluate the board position [3]. While a portion of the chess engine does use a tree-based search, the paper discussing Giraffe also details some experiments without using lookahead, which were able to achieve high accuracy picking the best move. Unlike Sabatelli *et al,* Lai uses a more tailored approach for inputs to the model, where the model is given the x, y coordinates of pieces, plus a variety of other important chess characteristics, like whether or not a side can currently castle. Lai claims that by reducing the dimensionality of the data in this way, since it requires far less than 768 features, it improves the ability of the model to learn important features, and ultimately learn to play chess better [3].

Stockfish, widely considered the best open-source chess engine, uses various features such as search trees and a recently developed, efficiently updatable neural network (NNUE) for evaluating chess positions; however, its AlphaBeta search still strongly depends on human-encoded heuristics and knowledge of chess [4]. Monte Carlo tree search algorithms were initially proposed in 1949 for the statistical study of differential equations [5]. Then, in 2017, DeepMind published their work on developing AlphaZero that mastered chess using a hybrid MCTS algorithm combined with a neural network trained solely through self-play in reinforcement learning with only the basic rules of chess [6].

4 Approach

Over the duration of our project we primarily analyzed two methods, zero lookahead chess engines and tree-based chess engines. In the latter half of the project, we also had the opportunity to explore an approach which combined the two.

4.1 Zero lookahead chess engines

Given a chess board, how can a model be constructed which will output the best *legal* move? Naively, you could construct a model which, given a board state, would output a desired board state however, this neglects the second point: was the move that took the engine to that board state legal? Our approach to solve this problem changes the question: how can we construct a model which, given a board state, will output the chance of winning with that board state? This turns out to be a much simpler question to answer, so it formed the basis with how we designed our zero lookahead chess engines.

The pipeline for our zero lookahead chess engines can be broken down into three main steps. First, the engine is given the current board state. From that state it then generates a list of the possible legal moves and their corresponding board states. Next, each of those board states is fed into a board evaluation regression model which returns the chance of the model winning with that board state. Finally, the engine picks the legal move corresponding to the board state that has the highest chance of winning and returns that move.

Choosing for the models to output the chance of winning, or more formally, the win-draw-loss (WDL) evaluation was based on a few factors. Primarily, this greatly simplifies the regression problem as the outputs from the models are bounded in the range [0, 1]. Conventional chess engines like Stockfish output centi-pawns, which is the measure of the amount of pawns that a player is up when position is taken into account [7]. This evaluation metric has a couple of disadvantages, firstly it is unbounded, which makes gathering uniform training data for the model more difficult, and secondly, the relationship between different centi-pawn evaluations is not linear, which makes comparing them more difficult [7]. The last factor we considered is that other machine learning based chess engines like Leela Chess Zero and Alpha Chess Zero use WDL as their evaluation metric [7].

4.1.1 Neural network based board evaluation models

Initially, we tried a variety of neural network architectures to create board evaluators which had qualitatively bad results, however, after some experimentation and research, we developed four architectures to compare using our test set. While the architectures did have some common hyperparameters, what mainly differentiated them was the type of input they expected.

The first architecture we developed is an MLP that was based off of the main input the Giraffe chess engine expects (out of three). This input is a vector which contains the positional coordinates of each piece, normalized to be in the range [0, 1], as well as the existence of each piece (0 or 1) [3]. Lai claims that these are the most important features for Giraffe [3], so it seemed intuitive to try a model using only these features as input.

The second architecture we developed is also an MLP which takes as input our most intuitive representation of a chess board. As input, the model expects a flattened 1-hot encoded 8 by 8 by 12 matrix, where the first two dimensions represent the position of a piece on a chess board, and the last dimension represents the piece type.

The third architecture we experimented with was our implementation of Giraffe's network architecture. Giraffe takes an interesting approach for its network where global features, like who's turn it is; piece-centric features, like what the first architecture we experimented with expects as input; and finally square centric features, like attack maps, are all required as separate inputs [3]. These separate inputs are then connected to separate fully connected layers, the outputs of which are concatenated then fed through another fully connected layer, which is then fed through its final output layer (Figure 1) [3]. As the paper describing the architecture does not fully explain the hyperparameters of the model, we did some experimentation to tune it to our problem which is further discussed in section 5.

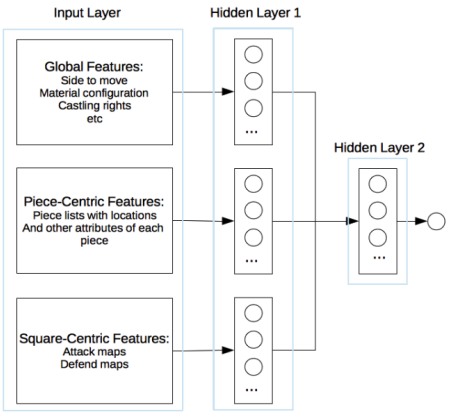

Figure 1: Giraffe network architecture [3]

The final architecture we explored is very similar to the second architecture we explored, which takes the flattened 1-hot encoded 8 by 8 by 12 matrix, except this network takes the non-flattened version. Additionally, instead of being an MLP, this architecture is a CNN, with the input passing through two convolution layers before being flattened then going through the fully connected layers. We opted not to use any pooling layers, as pooling layers downsample the feature maps [8], which we felt would not be useful for a board evaluator as the difference between a piece being on one square or the square next to it can often mean the difference between winning and losing a game.

4.1.2 SVM based board evaluation models

The support vector machine models were trained using Lichess data, which were uniformly selected based on the number of win and lose labels for the first approach and binned by Stockfish evaluation for the second approach (further on this in section 4.3.2). Our preliminary research indicated that a linear function kernel would not be suitable for chess evaluation, given the highly non-linear nature of the game. Therefore, we used a radial basis function kernel for all of our SVM models. Additionally, we chose to use support vector regression since we needed continuous evaluation to incorporate our models as a chess engine.

To further enhance the model, we incorporated additional features that provided more insight into the game The final features comprised a vector representation of mobility, attack, or defense for each piece type. For the attack map and defense map, we vectorized each piece type, paired with another piece type that was attacking or defending it. While there are other game aspects that could have been extracted, time constraints prevented us from doing so. Adding these features had a significant positive impact on the model's performance, but it still performed poorly compared to other models.

Despite improving the model's performance through the inclusion of additional features and using Stockfish evaluation as the label, the inherent complexity of chess and the unreliable labels for each board position led us to conclude that a static machine learning approach such as a SVM is unsuitable for developing a successful chess engine. As a result, we decided not to pursue a SVM further for our chess engine.

4.2 Conventional tree-based chess engine

The MinMax algorithm is a widely used technique in chess for assigning scores to each player based on heuristic metrics. Higher scores indicate that the first player is performing better, while lower scores indicate that the second player is. Using these scores, each player chooses moves that greedily maximize or minimize their score, respectively.

In our initial implementation of the MinMax algorithm, we used a rudimentary scoring system based on a similar evaluation metric to that used by Stockfish. This metric assigns values of 1 to pawns, 3 to bishops and knights, 5 to rooks, and 9 to queens. We then awarded additional points based on the number of attackers and defenders for each player. However, Stockfish has many more advanced metrics that experts, including grandmasters, have hardcoded over a decade.

AlphaBeta pruning is another technique used to cut branches from the decision tree based on the probability of a given branch leading to a win or loss relative to its sibling branches. However, we only implemented these methods in an elementary way since our focus shifted to the Monte Carlo Tree Search (MCTS) method that we hoped would work best with the neural network models we trained.

MCTS consists of four main steps: Selection, expansion, simulation, and backpropagation. If it is left to run for long enough it will converge to the best possible move in a deterministic and perfect information game [9], such as chess. Unfortunately, because chess is intractable "running it long enough" means for longer than the universe has existed. So, a sample of simulations is taken to approximate the best move with probability proportional to the runtime.

To limit the depth of a search and prevent it focusing on branches with initial high probability of success when another branch which appears worse may become better, upper confidence bound on trees (UCT) is used (Equation 1). In the selection step of MCTS, the ratio of visits to a parent node and the current node being evaluated is included to skew the search towards sibling branches less visited.

$$UCT(node):\ \frac{value\ assigned\ to\ node}{visits\ to\ node} + \sqrt{2(\frac{\ln(visits\ to\ parent)}{visits\ to\ node + 1}} \quad (1)$$

Experiments were then conducted on different methods of expanding when terminal nodes are reached. The most successful originated from preparing the function to accept trained models as input where heuristics and rollouts were used on every child of a node being expanded to calculate an average across its children, minimum, or maximum depending on the depth. For time limits of approximately 60 seconds this proved quite successful with our first draw against Stockfish using this method. However, it has limited ability potential and was shelved to integrate the models.

4.3 Training data

Over the course of our project, we mainly trained on two types of data, data the engines generated themselves through reinforcement learning, and chess datasets we found online.

4.3.1 Reinforcement learning

For reinforcement learning we performed two main experiments to find a reward function that could label chess positions in a meaningful way. The first method was a generation-based approach where at the start the models would play random moves and the winning model was rewarded. As the models saw more positions the rate of random moves would decrease allowing the models to make more and more of the moves. We hoped the models would progress together with most games ending with a winner as one model would have found a way to exploit the other, then the loser would learn to avoid the exploit and take over as the winning model and this would continue cyclically. We however did not observe this leading us to alter our reward function. The second function incorporated the sum of pieces left on the board where pieces are given the values Pawn 1, Knight/Bishop 3, Rook 5, Queen 9 shifted in favor of the winner of the game to get reward. We hoped this would give the models a better idea of where they go wrong when they blunder pieces or when to take a piece. We again saw little improvements to learning where the majority of games ended in draws or very strange positions see Figure A8 for example where black does checkmate but has 6 knights. This prompted us to experiment with a regression model trained on a larger set of this data to evaluate the reward function.

To do this we generated random chess games, labeled the positions with our reward functions, and trained a simple regression model on this data. For the first function we observed that it was only able to predict the winner accurately at the very end of a game. The second function could tell a position was winning if one player was drastically ahead as well as in the end game. Since much of a chess game should be relatively even, these methods are not good enough to distinguish between positions where for instance white has one more pawn and thus are not going to create good chess bots. The fact that the generational models did not progress at all when the regression models did however, led us to conclude there was a mistake in our approach and that it was not feasible to fix without first having the knowledge of a good model architecture and input structure. As such we shifted focus to training on pre-labeled stockfish data to gain better understanding to later return to reinforcement learning.

4.3.2 Lichess dataset

In addition to reinforcement learning, we also wanted to train models on existing data sets. We envisioned this allowing models to learn more quickly due to the higher quality nature of the data.

The first dataset we found was provided by Lichess.org [10]. Records of the over 4 billion games played on their platform are available freely to anyone for any purpose under the Creative Commons license, which perfectly suited our purposes. In addition, roughly 6% of the positions in those games are labeled by Stockfish [10], which allowed us to create a labeled dataset of nearly limitless samples. As it turns out, the major limitation we faced gathering this data to form our dataset was the speed in which it took to process it.

As the dataset is provided in PGN format, which stores an entire game as a list of moves (Figure A1), some preprocessing is required to translate this data into a format representing a labeled board state, which the board evaluation models we developed for the zero lookahead chess engines require to train. To translate this data, you must essentially 'play' a game in the PGN format, saving the board state and its evaluation, which we use as the sample's label, as you go. In addition, the board's evaluation, marked 'eval' in Figure A1, is stored in centipawn format, whereas our models WDL format, so some preprocessing is also required to convert that as well. Fortunately, python-chess, a chess library for python, is freely available and contains all this functionality [11], so this was implemented without much difficulty.

Unfortunately, drawing samples sequentially or at random from the dataset provided from Lichess does not provide a uniform dataset (Figure 2), which as discussed during a tutorial session, is important for training high quality models. As a result, further processing is required to gather a uniform dataset. Since the labels for the dataset are continuous in the range [0, 1], it's unfeasible to create a perfectly uniform dataset; however, our method resulted in a dataset which was uniform enough to train better models than the non-uniform dataset. Our method for doing this is quite simple: we create an arbitrary amount of bins with even capacities in the range [0, 1], then we collect enough samples to fill each of those bins. Experimentally, it was found that increasing the number of bins resulted in greater execution time while decreasing the amount of bins resulted in a less uniform dataset. As a result, 21 bins were found to deliver a 'happy medium' between generating a uniform dataset and the preprocessing execution time. As shown in Figure 2, by doubling the amount of bins used to generate the histogram of the dataset some non-uniformity is still present.

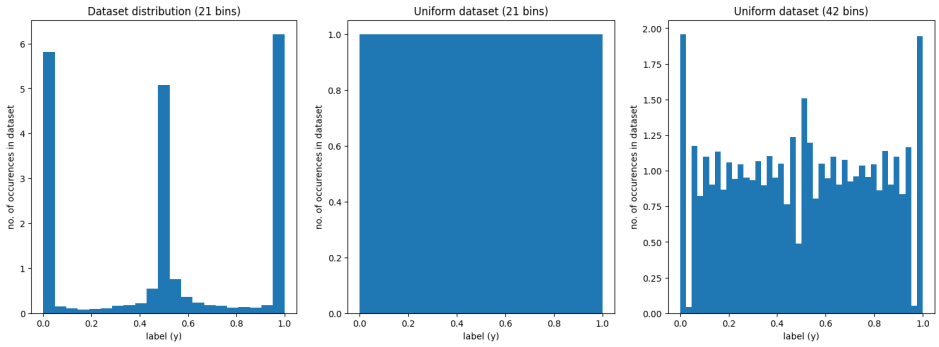

Figure 2: Lichess dataset distributions. The leftmost histogram is the dataset drawn sequentially from the Lichess data, which approximates the true distribution of the Lichess data. The middle histogram is the dataset drawn using the binning method described above with 21 bins. The rightmost histogram is that same dataset drawn using the binning method with 21 bins but with 42 bins being used to generate the histogram.

4.3 Using a zero lookahead chess engine inside a tree-based chess engine

Integrating neural networks with search trees proved to be a challenging task. Initially, we aimed to follow AlphaZero's approach by using a two-headed neural network that outputs both a value and a policy. The policy assigns probabilities to each possible move of a terminal node reached during the selection process, treating all player pieces as hybrid chess pieces that can move like

both queens and knights. The output is then filtered against legal moves and renormalized. The value head assigns a value to the same node, which is then back propagated as normal.

Traditionally, this approach would be trained through reinforcement learning, supervised learning, and self-play, rather than using pre-trained models. However, we have encountered difficulties with these methods. As a result, we conducted experiments with convolutional neural networks trained on Lichess puzzle data in an attempt to better understand how best to structure a policy network. See Figure A5 for the final structure used.

Due to time constraints, we shifted our focus to making the value models we had trained work as best we could. When a leaf node is reached during selection, we decided to keep the full expansion, where every possible legal move from the leaf node is added to the tree, and the value network assigns a new value to each. These values are used similarly to the policy, where the highest or lowest value, depending on which player is moving, is used instead of probabilities. In essence, the value network performs the work of both the policy and value heads.

Unfortunately, this approach presents several issues. The model must evaluate many board states at each expansion step, which can significantly slow down the search. Additionally, without reinforcement learning to make adjustments as games are played, the same mistakes are often repeated, resulting in poor performance against advanced engines and occasionally even the models on their own. A model evaluation may be accurate for the first expansion, but if a poor value is assigned on the next, when it is actually a good state, this will be back propagated and skew the selection towards worse outcomes.

To address these issues, we decided to combine the models we had and average the output value across them. If both models had equal evaluations, the assigned value is considered representative. If the evaluations differ, less change occurs in the back propagations. Finally, because our models are stronger and weaker respectively at different numbers of moves played, we assign weights to each model based on the search's current depth. A tournament system was introduced as well as an initial Elo rating function to evaluate the success of each method in playing actual chess games.

5 Experimental results

5.1 Zero lookahead chess engines

We were unable to get any functional chess bot through reinforcement learning however the experiments performed provided an excellent starting platform to build off of the preexisting data off Lichess.

Hyperparameters were tuned individually for each model we approached. That being said, it was experimentally determined that the Adam optimizer combined with Relu activation for each node, with exception to the last node of each model, which used a sigmoid activation function to bound the predictions in the range [0, 1], provided the best results for all of our networks. Additionally, deep, 3+ layer networks were found to be optimal for all of the models. The exact experimentally optimal network architectures for each model are shown in the Appendix (Figure A2, A3, A4).

The four models were then trained on the Lichess dataset we created with a learning rate that decreased when the learning plateaued. This made it possible to quickly train the models while still converging at a good minimum. The training was stopped when the validation loss stopped improving to prevent overfitting. The models were then tested on the same test set from the Lichess dataset. Since the models predicted continuous values, the predictions and expected values for each value in the test set had to be binned to create the confusion matrices shown below (Figure 3).

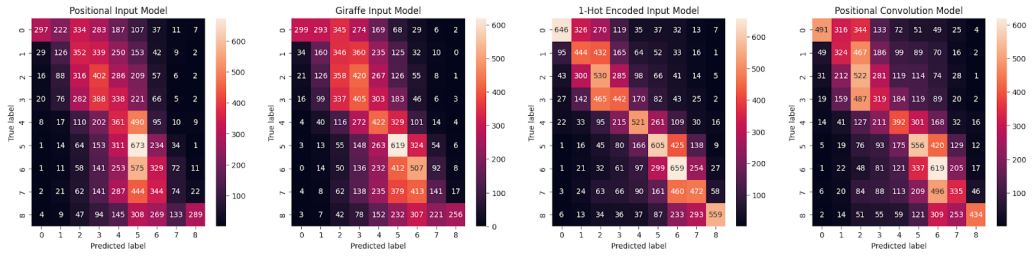

Figure 3: Binned confusion matrices for neural network board evaluators

As you can see in Figure 3 and Table 1, the MLP whose input was the one-hot encoded vector of a chess board scored the best, followed closely by the CNN whose input was the one-hot encoded matrix. Comparing these results to what was achieved by Sabatelli *et al.* and Lai in their respective papers, our models perform far worse. We currently believe this is due to our dataset's composition, which is further discussed in section 6.

Table 1: Test set results for each neural network board evaluator

Model architecture	Test loss (mean squared error)	Test mean absolute error
MLP (1-hot input)	0.0323	0.1183
CNN (1-hot input)	0.0440	0.1490
Giraffe implementation	0.0496	0.1709
MLP (positional input)	0.0557	0.1829

5.2 Conventional tree-based chess engine

Our conventional tree-based method was initially quite successful. We achieved 2 draws against Stockfish set to low Elo ratings. In testing, this model was able to beat all of our other models as they currently are. However, expanding on this method would require many further hours spent getting better at playing chess rather than designing and developing machine learning options that we feel show greater promise because of the success of AlphaZero and Leela.

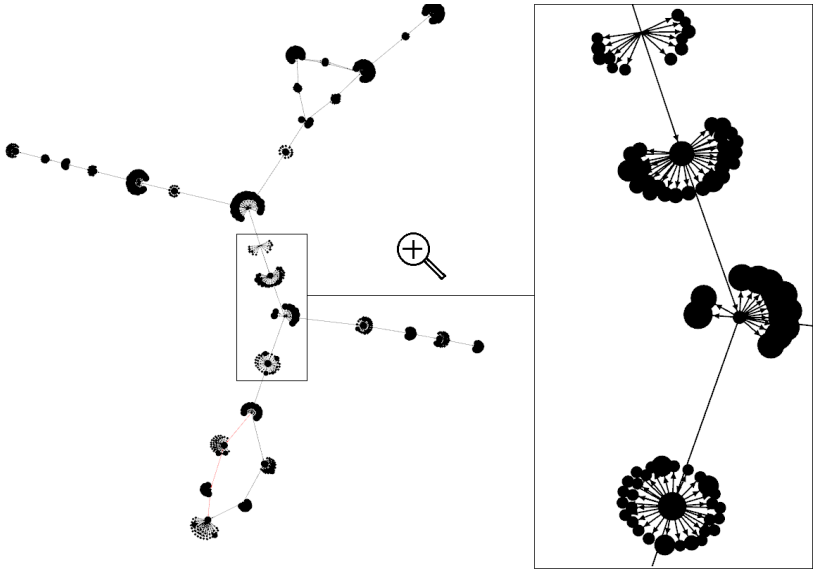

Figure 4: Graphical representation of MCTS with models created using Pydot and Gephi.

5.3 Using a zero lookahead chess engine inside a tree-based chess engine

Experiments using the Elo rating system created skewed results as most of our models were unable to defeat the base engines resulting in continuously dropping Elo scores. Scoring the models against each other in the tournament system produced comparative results useful for further changes to the tree search. However, iterative changes and long game play times resulted in very little data for final comparisons. The most important of which was changing how the models assign different weights for a board state based on which players turn it was and their piece color.

While compiling this report and the policy network graph, we made alterations to the network. Although it's too late to integrate our working CNN policy network with the MCTS for this report. With a fluctuating learning rate of 0.01,0.001, depending on training progress, using Adam optimization and cross entropy loss. Our test accuracy has reached approximately 30% in selecting the best possible move before filtering for legal moves on less than ideal data. This adjustment was trained on only a small sample of our data so we look forward to seeing its end result.

In summary, integrating neural networks with search trees posed challenges, and our experiments with different approaches had mixed results. Ultimately, we found that combining our trained models and carefully weighing their outputs helped to mitigate some of the limitations of our approach. Figure 4 shows graphical data created from searches using this method to check how trees were developing. Such graphs were useful when making adjustments based on visuals.

6 Future work

The data we used off Lichess was undersampled and preprocessed a lot but there are still imbalances as seen on the right of Figure 3. Additionally most of our models were trained on data that was never checked for uniqueness and only balanced on label. In the future we hope to further experiment with data checked for label and piece number balance to get a nearly continuous unique distribution as well as expanding this to a much larger dataset as one million positions are still a very small portion of possible chess positions.

We had to pause experiments for reinforcement learning due to time constraints and understanding of the reward function and models. Focusing on preexisting data allowed us to learn what model architecture and input structures work for chess, as well as giving a better understanding on how positions should be labeled/rewarded. With this new knowledge we hope to return to a reinforcement learning based engine in the future.

7 Conclusion

Chess is a formidable challenge, presenting complex data that even in 1,000,000 board states sampled from games played on Lichess.org, most board states only appeared once. To further improve our zero-lookahead models, improvements to preprocessing and sampling of training data must be made. Although there is still ample room for improvement in refining our approaches, we are delighted with the results we have achieved so far and look forward to tackling similar challenges in the future. The most promising model to date is MCTS with UCT selection, where move values are assigned by a combination of our neural networks and logic checks that keep the search within the rule space; avoiding obvious blunders.

Acknowledgement

We would like to thank Nishant Mehta for teaching the SENG 474 offering at UVic this term, as well as Jonas Buro, Andrea Nguyen, and Quan Nguyen for providing guidance during and after the tutorials they delivered. For any future students looking at taking this course at UVic, we highly recommend it as it was both interesting and highly pertinent to the current technological issues of today.

References

[1] C. E. Shannon, “XXII. Programming a computer for playing chess,” *Lond. Edinb. Dublin Philos. Mag. J. Sci.*, vol. 41, no. 314, pp. 256–275, Mar. 1950, doi: 10.1080/14786445008521796.

[2] M. Sabatelli, F. Bidoia, V. Codreanu, and M. Wiering, “Learning to Evaluate Chess Positions with Deep Neural Networks and Limited Lookahead:,” in *Proceedings of the 7th International Conference on Pattern Recognition Applications and Methods*, Funchal, Madeira, Portugal: SCITEPRESS - Science and Technology Publications, 2018, pp. 276–283. doi: 10.5220/0006535502760283.

[3] M. Lai, “Giraffe: Using Deep Reinforcement Learning to Play Chess.” arXiv, Sep. 14, 2015. doi: 10.48550/arXiv.1509.01549.

[4] S. Maharaj, N. Polson, and C. Turk, “Gambits: Theory and evidence,” *Appl. Stoch. Models Bus. Ind.*, vol. 38, no. 4, pp. 572–589, 2022, doi: 10.1002/asmb.2684.

[5] N. Metropolis and S. Ulam, “The Monte Carlo Method,” *J. Am. Stat. Assoc.*, vol. 44, no. 247, pp. 335–341, Sep. 1949, doi: 10.1080/01621459.1949.10483310.

[6] D. Silver *et al.*, “Mastering the game of Go without human knowledge,” *Nature*, vol. 550, no. 7676, Art. no. 7676, Oct. 2017, doi: 10.1038/nature24270.

[7] “Win-Draw-Loss evaluation - Leela Chess Zero.” https://lczero.org/blog/2020/04/wdl-head/ (accessed Apr. 08, 2023).

[8] J. Brownlee, “A Gentle Introduction to Pooling Layers for Convolutional Neural Networks,” *MachineLearningMastery.com*, Apr. 21, 2019. https://machinelearningmastery.com/pooling-layers-for-convolutional-neural-networks/ (accessed Apr. 08, 2023).

[9] S. Gelly *et al.*, “The grand challenge of computer Go: Monte Carlo tree search and extensions,” *Commun. ACM*, vol. 55, no. 3, pp. 106–113, Mar. 2012, doi: 10.1145/2093548.2093574.

[10] “lichess.org open database.” https://database.lichess.org/ (accessed Apr. 08, 2023).

[11] “python-chess: a chess library for Python — python-chess 1.9.4 documentation.” https://python-chess.readthedocs.io/en/latest/ (accessed Apr. 08, 2023).

Appendix

The appendix is composed of additional figures we wished to show that were not strictly necessary for the main body of the report.

```
[Event "Rated Blitz game"]
[Site "https://lichess.org/ctS1L3KE"]
[Date "2023.02.01"]
[Round "-"]
[White "Tasyra"]
[Black "Anatoly_zharkov_1984"]
[Result "0-1"]
[UTCDate "2023.02.01"]
[UTCTime "00:00:12"]
[WhiteElo "1524"]
[BlackElo "1536"]
[WhiteRatingDiff "-6"]
[BlackRatingDiff "+5"]
[ECO "B01"]
[Opening "Scandinavian Defense: Mieses-Kotroc Variation"]
[TimeControl "300+0"]
[Termination "Normal"]

1. e4 { [%eval 0.36] [%clk 0:05:00] } 1... d5 { [%eval 0.82] [%clk 0:05:00] }
2. exd5 { [%eval 0.66] [%clk 0:04:59] } 2... Qxd5 { [%eval 0.75] [%clk 0:04:59
```

Figure A1: Sample game in PGN format (first four moves) [10]

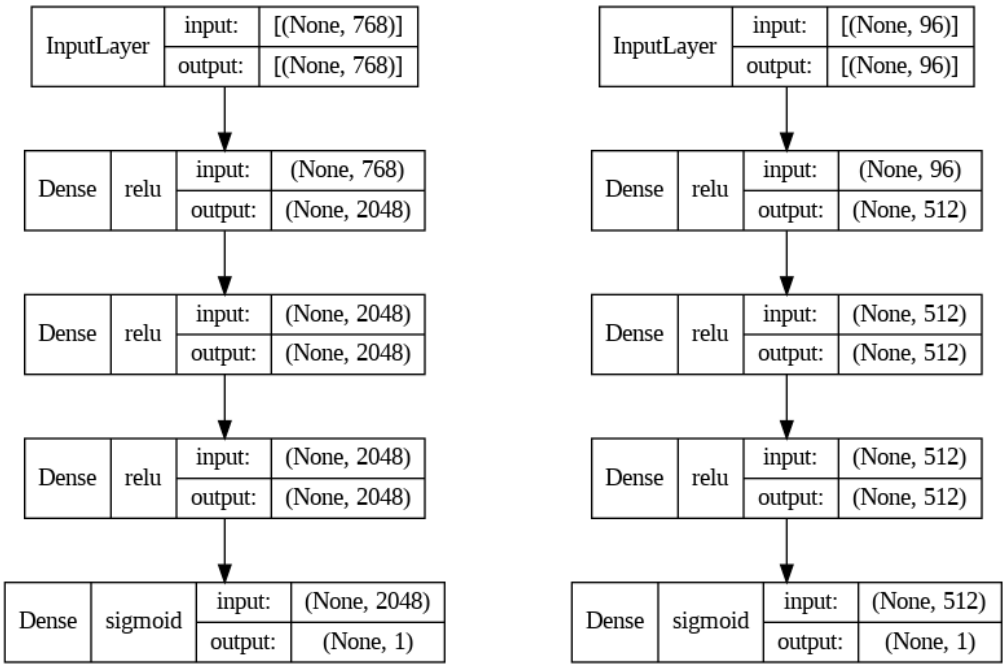

Figure A2: MLP model architectures. To the left is the model whose input is a one-hot encoded board whereas to the right is the model whose input is the piece-centric features from the Giraffe chess engine [3].

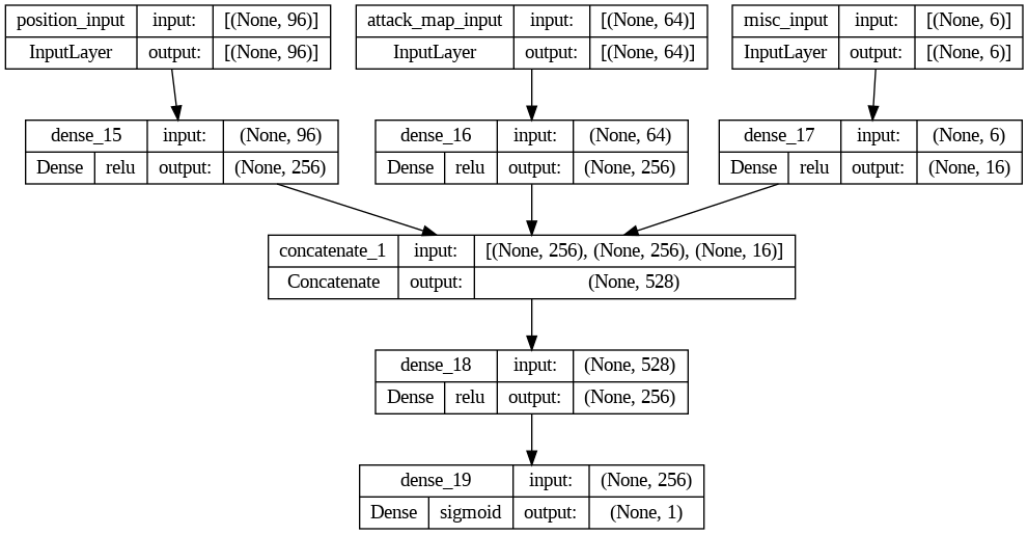

Figure A3: Giraffe model architecture whose hyperparameters were tuned experimentally. Since the paper on Giraffe did not provide hyperparameters for the network, optimal hyperparameters had to be determined experimentally for this network.

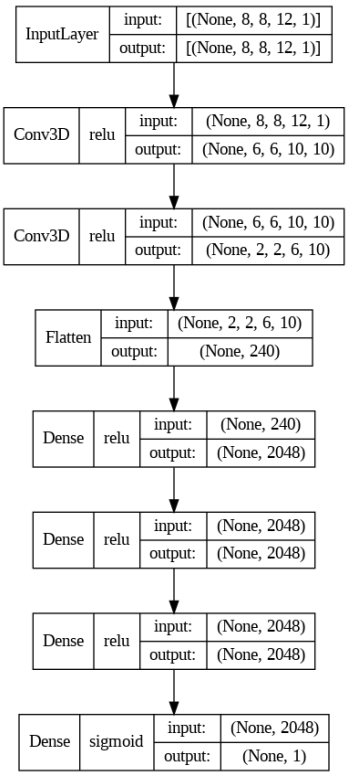

Figure A4: Convolutional neural network model whose input is a one-hot encoded matrix representing the board.

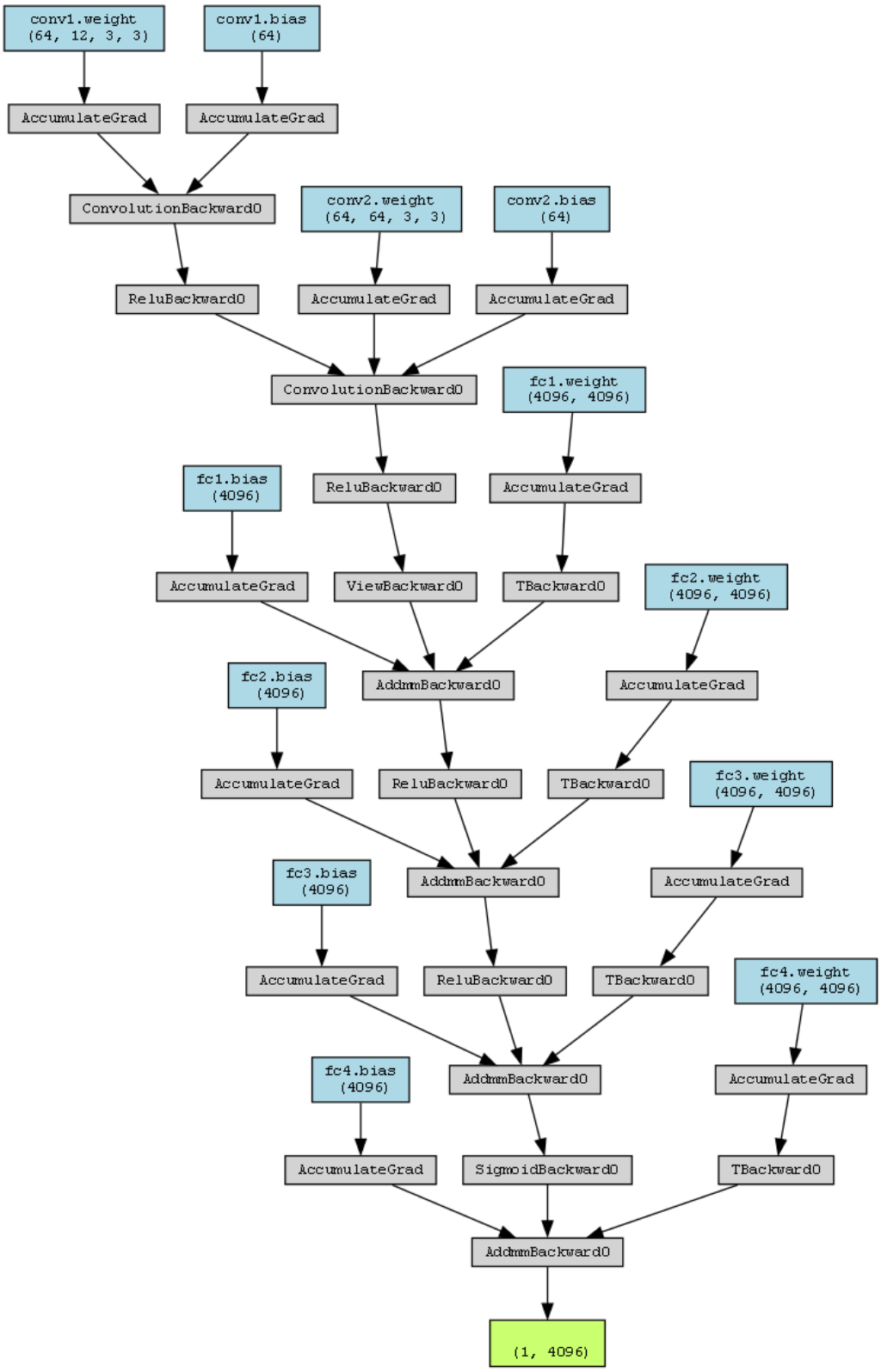

Figure A5: Convolutional neural network for the policy model attempt that was to be used in search tree implementation using PyTorch

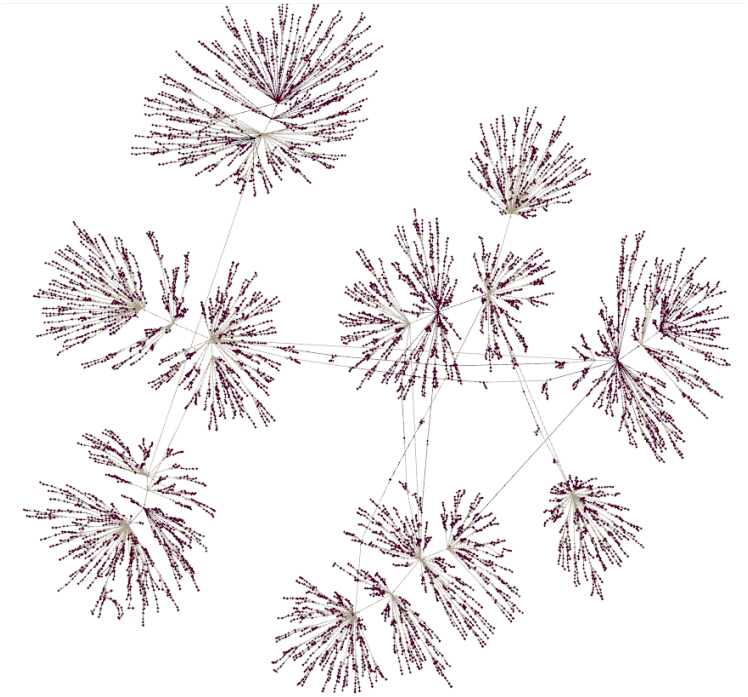

Figure A6: A Pydot->Gephi image of the traditional MCTS for a chess move

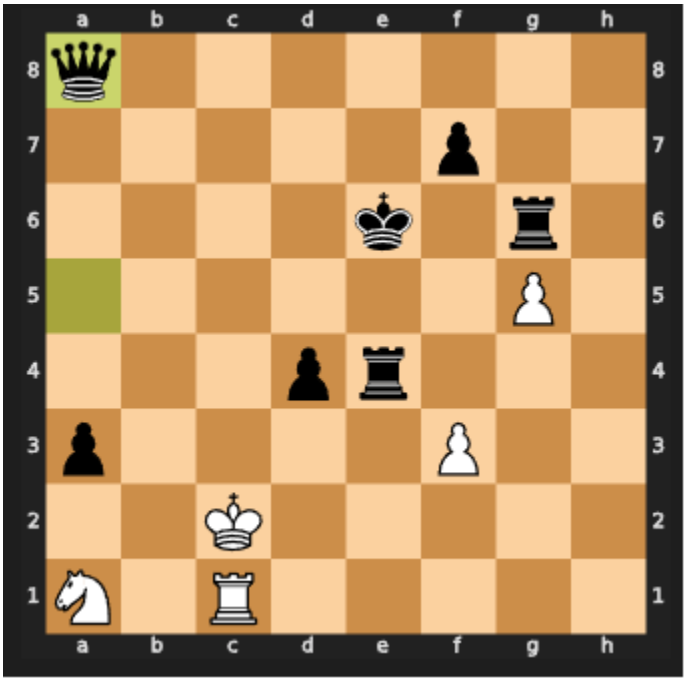

Figure A7: A python-chess board representation of the search space used for the Pydot images

Figure A8: Example board position achieved between two reinforcement agents