Reinforcement Learning in Chess

Jason Chu, Stephen Li, Wei Min Loh
{jjchu, s575li, wmloh}@uwaterloo.ca
University of Waterloo
Waterloo, ON, Canada

Abstract

Recent endeavours to master computer chess often rely on deep reinforcement learning techniques that require extensive self-play samplings. In this report, we explore the viability of using supervised learning with a professional computer chess dataset to accelerate the learning process. Doing so would reduce the need of highly parallel hardware for sampling. Chess data obtained from an external source is used for the policy network pre-training to accelerate rule learning process. Subsequently, it is subjected to self-play using Monte Carlo tree search to generate checkmate move sequences for additional training. The results from simulations with the Stockfish chess engine reveal that our final policy network performs exceptionally well within the first 10 moves but gradually worsens later in the games. Upon empirical measures, the policy network is deemed to be at an ELO level of approximately 1100.

Introduction

Chess is a deeply complicated strategy game whose complexity extends even beyond the world's best human players in the present day. With active AI chess research starting as early as the 1950s, it was not until 1997 that a chess AI, "Deep Blue", had claimed victory over the world chess champion for the first time. Ever since, the interest in computer chess gradually declined as the performance of traditional chess engines stagnated. With the emergence of deep reinforcement learning techniques, a vastly different set of technologies is available to reinvigorate a fresh perspective of computer chess.

The methods used by many of the historical chess programs rely on search. There are an estimated 10^{43} legal positions in chess, and given this large number, it is very expensive to perform search for every move made. As such, it would be desirable for performant chess programs to avoid such enormous computations in-game. This leads to our primary research question: what percentage of wins can a deep policy network, co-trained with the Monte-Carlo tree search algorithm, obtain against modern search-based chess engines?

The deep policy network $f_\theta(s)$, which comprises of stacked convolutional and residual layers, is initially trained with data sampled from games between search-based chess engines, which is obtained from Computer Chess Rating Lists (CCRL). To strengthen its performance, the Monte-Carlo tree search (MCTS) algorithm is adopted to incorporate an element of self-play so that $f_\theta(s)$ can learn to explore and exploit different strategies in chess. With multiple iterations of training and data generation from MCTS, $f_\theta(s)$ has been shown to gradually improve in making moves that result in wins. The MCTS algorithm has been modified to fit the constraints of chess where only legal moves are considered and the reward signals are strictly derived from wins and losses only.

Performance of the network will be determined in an ELO rating system, a common metric which considers a player's rating based on other players of similar strength. Using this rating, it is possible to calculate the probability of a player winning when playing against another player (Elo 2008). $f_\theta(s)$ is subjected to 50 simulated games with StockFish[1] using an existing implementation in Python and its ranking is dictated by the conventions of the ELO rating system. These simulations determine a statistically significant evaluation of the performance of our AI against both human and computer opponents in the real world.

The performance of several chess agents were compared with one another. The primary model of choice is a deep policy network f_1 that was trained with the CCRL dataset and data generated by the MCTS. The other model, f_2, is identical to the primary model, but without training with the data from MCTS. The quantitative results suggest that in the earlier stages, both models performed at a similar level of 0.50 to 0.70 according to the Stockfish engine evaluation function. However, the addition of MCTS data greatly improved the performance of the policy network, particularly in the later stages of the matches, which enabled f_1 to win against a 800 ELO Stockfish engine on average across 50 simulations while f_2 lost. f_1 was shown to be the superior model and subsequent simulations proven that it has an average level of 1100 ELO.

This paper made some contributions to the field of artificial intelligence in games.

1) The paper explores the possibility of utilizing supervised

[1] One of the leading state-of-the-art chess engines without machine learning

learning as a means of accelerating the learning of rules in a strict environment like chess. Our model was able to easily recognize game rules within 8 hours of training on CPU. AlphaZero for Chess, while state-of-the-art, took 9 hours to train the neural network with 16 tensor processing units (TPU) and required 5,000 TPUs to generate MCTS self-play games (Silver et al. 2018). Therefore, our method would offer a more suitable alternative for artificial intelligence hobbyist to configure their own agents, especially for those who do not own vast amounts of computing resources.

2) Another major contribution is the novel design of a *single-shot* agent. Unlike other relevant machine learning-based agents which require some degree of search at inference time, our deep policy network was trained with the intention of developing an implicit probability distribution of what a model-based search would return. This results in a faster inference time and a more natural simulation of a physical world, despite a significantly poorer performance.

3) In addition to the previous point, the general training processes described by the paper are task-agnostic, i.e. they can be applied regardless of the domain as long as the environment can be represented as mathematical objects. In particular, these processes would be suitable for discrete reinforcement learning problems where the environment cannot be simulated via search at inference time.

Related Work

Researchers have contemplated many approaches to solving chess over the years. Deep Blue, the first proven computer program to reach superhuman proficiency in chess, adopted the approach of highly-parallel search systems (Campbell, Hoane Jr, and Hsu 2002). The major improvements from previous iterations of such system stem from algorithmic design and hardware upgrades. The evaluation function primarily works by estimating the state based on static and dynamic features. The development of search capability of Deep Blue focuses on hardware accelerations such as mounting 480 chess accelerator chips on a supercomputer. There were modifications to the minimax search, particularly using the null move pruning technique. This is accompanied by the algorithms being implemented on the circuitry level which gave Deep Blue a massive boost in parallel searches (approximately 200 million chess position searches per second). However, the approach outlined by the paper does not offer much relevant algorithmic insights since it heavily relies on chess databases.

After the defeat of Garry Kasparov by Deep Blue, a stagnation in computer chess persisted as researchers pursued different research areas (Greenemeier 2017). However, before the rise in popularity of deep learning, Sebastian Thrun published a pioneering paper on the application of artificial neural networks on computer chess in 1995. Prior computer chess programs utilize handcrafted features to characterize game states which may not generalize well. Thrun published the findings on adopting explanation-based neural network learning (EBNN) as a means to replace traditional evaluation functions which, albeit winning only 13% of matches against GNU-Chess, revealed potential and caveats with neural network learning (Thrun 1995). Temporal difference learning (TD) was used in conjunction with neural network to stabilize the training process in the presence of delayed rewards. Unlike other computer chess programs, the paper does not offer a sampling technique but rather relies on a database of expert chess games.

A breakthrough in effective sampling techniques was proposed by Kocsis et al. The introduction of the Monte Carlo tree search (MCTS) enabled selective sampling which helped overcome the massive sample space of a Markov Decision Process (Kocsis and Szepesvári 2006). This development stems from a bandit algorithm called UCB1 that determines which internal nodes to expand, as opposed to exploring in a systematic fashion. The underlying principle of this algorithm aligns with the trade-offs posed by a MDP: exploration versus exploitation. By introducing a randomized algorithm that gradually becomes greedy in the choice of node expansion, the expanded state spaces are shown to be sufficiently wide and deep by the end of the search. Important theoretical results from the paper include convergence in expectation of expected rewards and convergence of failure probability. The experimental result compliments the theory where the UCT algorithm requires fewer samples to achieve a particular error rate when compared to competing algorithms: ARTDP and PG-ID. In general, the Monte Carlo tree search offers a valuable algorithm as a means to explore state spaces.

With the rise in deep learning methods and superior hardware resources, researchers returned to the computer chess with different toolkits. Most have already abandoned dedicated hardware approaches such as Deep Blue. Instead, the recent methods approach focuses on a data-centric way to fit the evaluation function. DeepChess is a particular design that adopted this new paradigm and harnesses these recent advancements (David, Netanyahu, and Wolf 2016). Unlike previously discussed approaches, DeepChess does not rely purely on reinforcement learning and search but uses both supervised and unsupervised learning. This is possible due to the availability of vast amounts of expert chess match data. DeepChess does not work directly with the raw state representation; David et al. suggested the use of the deep belief network (DBN) to extract high level features to avoid high dimensionality and overfitting the evaluation function. The DBN is trained with 2,000,000 chess positions with 50% White-win positions and 50% Black-win positions.

A Siamese network was designed with the purpose of comparing which of the two given states is more likely to win the game. The DBN is stacked on the two heads with shared weights to serve as high level feature extractors while the subsequent layers evaluates the value function (David, Netanyahu, and Wolf 2016). The Siamese network is trained with 1,000,000 unique positions, achieving a spectacular validation accuracy of 98.0%. DeepChess, however, uses a relatively outdated variant of search: α-β search. It is used to evaluate certain positions from the root at a predefined depth. The authors recognized the issue of poor computational performance due to the massive number of pair-

wise state comparisons which was partially rectified with a smaller Siamese network without a significant decrease in accuracy.

The success of AlphaGo by Google DeepMind brought attention to the potential of deep reinforcement learning (Silver et al. 2016) in games. AlphaZero builds upon the strengths of AlphaGo and generalizes the algorithm to solve chess, Shogi and Go. AlphaZero combines the use of deep learning (like DeepChess) with an improved general-purpose MCTS (Silver et al. 2018). The key distinction from DeepChess is the complete lack of human guidance and hence, minimized bias in the policy. This is attributed to the use of MCTS which is a form of self-play. The MCTS samples games of self-play (generated from competing population of AlphaZero agents) and picks the best agents for subsequent iterations, which replaces the current agent if applicable. The two-head policy-value network is fundamentally similar to the Siamese network by David et al., but uses convolutional layers instead of fully connected layers. This neural network $f_\theta(s)$ outputs the policy vector and predicted outcome $(\mathbf{p}, v)$ where $\mathbf{p}$ is a biased estimate of the true proportion of winning state in MCTS and v is an estimate of the expected outcome of the game in a given state. The training of the network from simulated data, using only the rules of the game as input, achieved superhuman intelligence in just several hours, and was deemed to be the current state-of-the-art. However, the method requires extensive computational resources and time which may be a significant hindrance to other researchers.

Methodology

Data Parsing and Generation

A popular open-source dataset from Computer Chess Rating Lists (CCRL) will be used as initial training data for the deep policy network (obtained from `http://ccrl.chessdom.com/ccrl/4040/`). The dataset contains the moves of each recorded game between world-class chess engines and its final outcome, which is used as the probability labels for the network. This dataset was chosen for its common use among high performing learning-based chess engines such as DeepChess (David, Netanyahu, and Wolf 2016), as well as the quality of the games being played by other chess AI, many of which have abilities beyond a human grandmaster level. Within the obtained dataset, 34.4% of games led to a white win, 25.4% to a black win, and 40.1% to a draw.

The dataset consists of 1,183,411 games[2] between computer chess engines in Portable Game Notation (PGN) format. The PGN format is a plain text format for recording chess games by writing the moves played in the game. The parsing step is necessary to construct an integer-array-based data structure that is accepted by the Tensorflow API (Abadi et al. 2015).

A chess board state can be represented as an $8 \times 8 \times 13$ NumPy[3] array (Harris et al. 2020). The third dimension is attributed to the one-hot encoding of the 13 discrete piece types: 6 piece types (rook, knight, bishop, queen, king, pawn) for each colour, and an additional representing the empty tile. The manner of data generation takes into consideration the issue of overfitting. Extracting all board states in each game can lead to overfitting due to data that are highly similar since each consecutive state differs only by a single piece position. We propose a sparse sampling technique in the data generation process. Every state in every game has a 0.2 probability of being sampled and collected into the dataset $\{(s_i, (v_i, a_i))\}$.

The underlying assumption of the sampling technique is recognizing the Markovian nature of chess (Haworth, Regan, and Di Fatta 2009). An intelligent agent should only need to see the current state to make a well-informed decision, and does not require the history of moves. Therefore, the only feature of the dataset is the board representation itself with no human-constructed features. As a result, there are features external to the board state that can affect the evaluation of the game state but is not directly captured in the dataset. Such features include: castling availability, en passant availability and player turn.

For castling availability and en passant availability, these features are ignored for the purposes of training the agent as it does not require explicit knowledge of the rules of chess. Given that the training set will have games with castling and en passant moves played, the agent infers such moves probabilistically.

The agent plays the game from the perspective of the white player to avoid having the need of two distinct agents. As such, the data generation process includes transforming the board states whenever it is black's turn by flipping and swapping colours of the pieces. At this point, the game state is evaluated from white's point of view. This transformation also reduces biases of black and white perspectives which is especially crucial when the dataset is not perfectly balanced. The overall process of parsing and generating data can be summarized as follows:

Algorithm 1 Data Parsing and Generation

$G \leftarrow$ Bernoulli random number generator, $p = 0.25$
for game g in raw dataset **do**
    $v \leftarrow$ outcome of the game g ($v \in \{-1, 0, 1\}$)
    for board b in g **do**
        if black's turn **then**
            flip board and swap piece colours in b
        if `sample`(G) = 1 **then**
            $s \leftarrow$ convert b into a $8 \times 8 \times 13$ NumPy array
            $a \leftarrow$ action taken after this board state
            store tuple $(s, (v, a))$ in the dataset

Policy Network

The policy network $f_\theta(s)$ is the primary deliverable of this project as it is the direct representation of an agent. It generates the action our reinforcement learning agent will take at every turn $\boldsymbol{a}$ and evaluates the probability of winning $\boldsymbol{v}$. However, unlike Bastani et al., we do not opt for a tree

[2] As of October 18, 2020

[3] A popular mathematical and scientific computation library in Python

policy due to the massive computational overhead at inference time (Bastani, Pu, and Solar-Lezama 2018). Instead, the policy function is directly parameterized as a convolutional neural network (CNN) with weights θ under the actor-critic paradigm in reinforcement learning. A CNN specializes in tasks with data of high spatial-context information by using a parameterized convolutional kernel as a feature at each layer of the neural network. A complete feedforward operation involves applying the kernel onto every subsection of the data from the previous layers. The final output layer is typically represented as a fully-connected neuron layer which flattens the dimensions of the final convolutional layer.

The data $\{(s_i, (v_i, a_i))\}$ is used to pre-train $f_\theta(s)$ before the actual training process for the Monte Carlo Tree Search process because an existing $f_\theta(s)$ will be needed as the default policy for simulations. It consumes state s_i and predicts $\hat{v_i}$ and $\hat{a_i}$, and it eventually learns to optimize such that $\hat{v_i} \approx v_i$ and $\hat{a_i} \approx a_i$. The train and test dataset are split with a 80:20 ratio and a 5-fold cross validation is used during the training process. The best model, dictated by its performance in raw predictive accuracy, is chosen.

Monte Carlo Tree Search

Having a policy function trained directly with only supervised learning incurs high bias because it does not explore the domain but simply emulates player moves. Monte Carlo Tree Search utilizes a pre-trained policy network $f_\theta(s)$ to perform policy iteration through self-play with the aim of increasing the probability of action $\boldsymbol{a}_i$ that maximizes the action value function $Q(s, a)$, i.e. $\boldsymbol{a}_i = \text{argmax}_a\, Q(s, a)$. The overall process can be best described in Figure 1.

The Monte Carlo tree T starts with only node n_0 which represents the start state. Each node is characterized by a board state and the set of edges represents actions. The *select* phase involves identifying which edge to expand. The decision is based on the value of the upper confidence bound $U(s, a)$. In this project, we opt for the variant called *predictor upper confidence bound* (PUCB) which was used in AlphaGo (Rosin 2011). Legal actions a_t at timestep t are taken such that $a_t = \text{argmax}_a\, Q(s_t, a) + U(s_t, a)$ where

$$U(s, a) = \tau p(s, a) \frac{\sqrt{N(s)}}{1 + N(s, a)}$$

and τ controls the degree of exploration, $N(s, a)$ is the number of times the edge has been explored and $N(s) = \sum_a N(s, a)$.

Given the knowledge of which nodes to systematically expand, the chosen node n_t at timestep t can be expanded. During the *expansion* phase, that node has to be evaluated using Monte Carlo simulation. The policy network is used as a default rollout policy[4] to arrive at a biased estimate of true search probability π. The default rollout policies must be continued until the termination of the game. The results r of this game must be in $\{-1, 1\}$ where -1 means loss and 1 means win. The *backup* phase is simply backpropagating and updating $N(s, a)$ and $Q(s, a)$ of its ancestors. The select-expand-backup cycle repeats 50,000 times.

At the end of iterations of the MCTS, T would possess sufficient data points to estimate the search probabilities. Then we can derive the approximate search probability $\hat{\pi}_s$ of win-lose states from each state s. T allows us to generate data from the self-play since the states are given, actions represented by edges and value quantified by $Q(s, a)$. Batch sampling and updates are used to update the weights θ every 50,000 states explored to ensure that the default policy gradually improves along with the approximated tree policy. In total, 1,000,000 cumulative states are explored in the tree. The Monte Carlo Tree Search algorithm for a single cycle can be summarized as pseudocode (Silver et al. 2016).

Algorithm 2 Single-Cycle Monte Carlo Tree Search

```
T ← tree of a single node
while maximum iterations not reached do
    E ← edge that maximizes Q(s, a) + U(s, a)
    N ← expanded leaf node that is connected by E
    r ← results from complete rollout of N with f_θ(s)
    update Q(s, a) for all (s, a) that are ancestors of N with r
π̂_s ← vector of Q(s, a) for each state s
Store all tuples (s, π̂_s) to train f_θ(s)
```

Real-Time Inference

With a trained policy network, it would be possible to determine the move with the greatest probability of winning. When given a board state s, the policy network $f_\theta(s)$ produces a matrix of probabilities, with each value corresponding to a different successor state. The inference algorithm will select the state corresponding to the greatest probability in the matrix, maximizing the player's probability of winning. It will then return the selected state along with its corresponding probability of winning, allowing for reporting game state metrics to the user.

However, given that the policy network has no explicit knowledge of the rules of chess, it is possible that the move with the highest probability could result in an invalid state. In such a case, the inference algorithm would return a legal state with the highest probability dictated by $f_\theta(s)$.

Selecting a valid state requires a state validator to be available to the inference algorithm. This state validator accepts two inputs: the parent board state and the proposed board state. The validator generates all successor states of the parent board state, where a successor state is defined as a state which can be achieved in one turn from the parent board state. The proposed board state is then compared to all successor states and if there is a match, the proposed state is valid. Otherwise, the successor state is invalid.

The pseudocode would be as follows:

[4]The greedy policy function without exploration tendencies and randomization to evaluate a given node

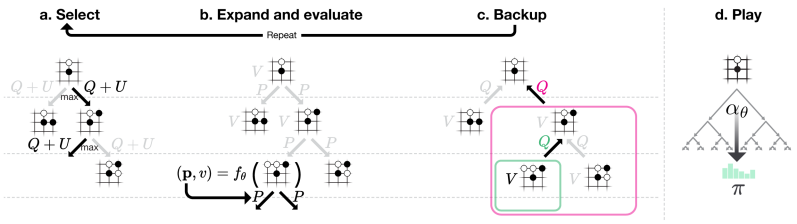

Figure 1: Monte Carlo Tree Search Processes (Silver et al. 2016)

Algorithm 3 Board State Validation

```
S ← successors of parent board state
for s ∈ S do
    if s is the proposed board state then
        return True
return False
```

In the case where there are no successors, the game has come to an end.

The inference pseudocode is shown as follows:

Algorithm 4 Inference of Policy Network

```
P ← probabilities from f_θ(s)
sort values of P in descending order
for p ∈ P do
    a ← action associated with probability p
    if a is a valid action then
        return (p, a)
return "game has ended"
```

Results[5]

The Elo rating is a popular metric used to rank human players and as such, having an Elo rating for the policy network $f_\theta(s)$ would be indicative of the calibre at which the AI plays in relation to human players. However, because $f_\theta(s)$ could be ineffective in some scenarios, it would be unfair to measure strictly in a typical ELO convention, which doesn't describe the strength of moves across the various stages of the game. Therefore, to evaluate the efficacy of $f_\theta(s)$, games are run against the Stockfish chess engine with numerous Elo rating settings while measuring the quality of moves at each board state (evaluation score) which is provided by the Stockfish chess engine. We compare the evaluation score in matches with Stockfish facing itself to the win rate of $f_\theta(s)$ as black and white in matches facing Stockfish. By using the evaluation score function, we treat our agent represented by $f_\theta(s)$ as a human player.

The training and validation set for the policy network pretraining were constructed from splitting the complete data, which consists of 5,000,000 board states, action and outcomes, into a 80:20 ratio. The training was done under a 5-fold stratified cross validation. The test set was obtained from some 50,000 unseen games where 500,000 board states were sampled from and eventually based on how well $f_\theta(s)$ emulated the actions prescribed by the dataset.

The hyperparameters of $f_\theta(s)$ are overall CNN architecture, optimizers, loss functions, epochs and regularization parameters. Given the black-box nature of deep neural networks, our architecture was inspired by AlphaGo Zero, where we use 22 *residual towers* which each comprises of 2 convolutional layers with 64 filters of 3×3 kernel, 2 layers of batch normalization (with ReLU activation function) and a skip input connection (Silver et al. 2016). This choice is also motivated by the performance and efficiency of DenseNet (Huang et al. 2017) which popularized extensive residual connections. The output of $f_\theta(s)$ necessitates a two-head policy network where it has been demonstrated that shared weight multiple-head outputs grant convergence stability and improved performance (Silver et al. 2016) compared to separate value and policy networks. The policy head will use the softmax activation function since it is a decent representation of multi-class probability distributions and the value head will use the hyperbolic tangent function. f_θ is trained under the loss function ℓ where

$$\ell = (v - \hat{v})^2 - \boldsymbol{\pi}^\top \log \boldsymbol{p} + c\|\theta\|^2$$

and c is set to 0.001 to control the degree of L2 weight regularization. The optimizer selected is the Adam optimizer which typically offers advantages in a highly nonlinear surface (Silver et al. 2016).

Implementation

The implementation can be divided into four sections: data processing/generation, policy network training, Monte-Carlo tree search and game simulations as can be seen in Figure 2. The overall implementation strongly integrates a popular Chess library on Python called *python-chess*.

The PGN parsing is primarily handled by *python-chess*. Each game contains approximately 140 board states of which a subset of board states are sampled from sparsely. As games progress, the number of possible moves increases exponentially which results in more similar board states among moves at the beginning than the rest of the moves. To evenly distribute sampled data among variations of games, the first six moves are sampled with a probability that is inversely proportional to the probability density function of an exponential distribution with a rate parameter of $\lambda = 33$ which is the average number of actions at each state (Fainshtein and HaCohen-Kerner 2006). All subsequent board states are

[5]The complete implementation can be found at `https://github.com/wmloh/ChessAI`

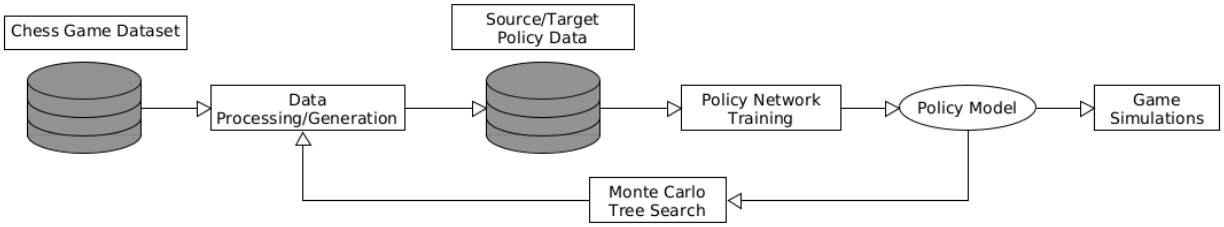

Figure 2: Implementation structure

sampled with a Bernoulli distribution with a probability of 0.25.

The policy network is encapsulated in a custom-defined class `PolicyModel` which abstracts the complexity of working with Tensorflow as well as automatically reformatting the output from the `PolicyModel` into one that fits easily into existing function signatures. For instance, since actions in chess involves moving a piece from one spot to another, we need two separate policies: *source policy* (location of piece to move) and *target policy* (destination of the move). Another feature of the `PolicyModel` is the pre-defined construction on the neural architecture based on the general DenseNet structure. The only two hyperparameters that are tunable is the number of *feature blocks* (residual blocks between the input and the source policy output) and the *sandwich blocks* (residual blocks between the source policy output and the target policy output). As described by the AlphaGo paper (Silver et al. 2016), each block is comprised of, in order (illustrated in Figure 7):

1. Input layer
2. Convolutional (2D) layer with 64 filters, (3, 3) kernel size, stride of 1 with dimension-preserving padding and initialized with Xavier uniform initializer
3. Batch normalization layer
4. Parametrized rectified linear unit (PReLU)
5. Convolutional (2D) layer with 64 filters, (3, 3) kernel size, stride of 1 with dimension-preserving padding and initialized with Xavier uniform initializer
6. Batch normalization layer
7. Double-input add layer from the previous layer and from the input layer
8. Parametrized rectified linear unit (PReLU)

The optimizations needed for this component is minimal since Tensorflow provides a highly optimized system with GPU integration.

The Monte-Carlo tree search (MCTS) is the most implementation-intensive component due to the complexity of the algorithm as well as the scalability and optimizations needed. The main class `MCTS` holds the root `Node` and implements the high-level functions such as simulating multiple games from the root using the algorithm discussed in the methodology. The primary function here is the function to generate data that results in a checkmate. This is done by searching for games in the dataset that are forfeited and are not in a checkmate status. At every board state starting from the most recent board state, it performs 40 MCTS cycles with `PolicyModel` as its rollout policy and determines the best move. After the best move is taken, it is repeated until the game ends with a checkmate move. All sequences of moves leading up to checkmates are collected as additional training data for the model. `Node` contains the details of the algorithm where functions to complete the four stages of MCTS (as described in Algorithm 2) are implemented. The primary technical challenge encountered in this component is the difficulty in parallelizing the search operations and optimizing the algorithm. The algorithm prescribed by Silver et al. in AlphaGo is sequential and our attempts to parallelize the algorithm resulted in poor and unintelligible results. Hence, the implementation presented in this project remains sequential.

The simulations of chess games are done against Stockfish with various ELO settings. The ELO setting of Stockfish is configurable to predetermined skill levels, and setting a lower ELO causes Stockfish to purposefully choose sub-optimal moves with limited searching time and depth. To simulate games using a trained model, the output policy from `PolicyModel` goes through an inference algorithm that selects legal moves which maximize the probabilities given by the model. The Stockfish engine is also used for move analysis, and is called every time for each Stockfish and `PolicyModel` configuration. This analysis function returns the expected outcome of the game (win, draw, loss) given the current board state.

Performance

Exploratory data analysis was performed on the games parsed. The distribution of win/loss games are fairly balanced: 2,264,611 game states that led to wins and 2,251,325 game states that led of loses. However, it was observed that in each game, there is a severe lack of sequences of moves that results in checkmates. Upon investigation, it was revealed that the players in the game typically forfeit a few moves prior to a forced checkmate due to unfavourable chances of winning. As a result, the initial training of the model would most likely be unable to learn how to checkmate the opponent. This issue is partially addressed in a later section.

The extracted board states, action and values from the PGN files were used to conduct the initial training of the

`PolicyNetwork`. The goal of the initial training is to optimize the first few layers of the residual neural networks to extract spatial features from the sparse three-dimensional board states, as opposed to a end-to-end classification. Nevertheless, the loss and accuracy metrics grant insightful information during the hyper-parameter selection process. Four candidate hyper-parameters were proposed and were subsequently trained over the course of 24 hours.

Figures 3 and 4 illustrate the cross-entropy loss (full line) and accuracy (dotted line) during the training process. The legend in the graphs (x, y) indicate that x is the number of feature blocks and y is the number of sandwich blocks. An important characteristic of the plots is the monotonically decreasing loss functions in both the training and validation. This implies that the model is underfitting[6] and fails to sufficiently converge because the accuracy stagnates at approximately 50% and the loss stagnates at 1.4 cross-entropy loss for source policy and 35% accuracy and 2.1 cross-entropy loss for target policy.

This is, however, not unexpected since the training data is extremely noisy due to seemingly arbitrary moves made over 150,000 games. Also, the action space for source policies is smaller than the action space for target policies because source policies include only the agent's pieces while target policies include the empty spaces and the opponent's pieces. Therefore, the difference in variance manifests as better source policy prediction. In both figures, the training loss and validation loss is minimized by the `PolicyModel` with the deepest network, i.e. 12 feature blocks and 10 sandwich blocks. Hence, this model is chosen the main model. Figure 8 illustrates the overall architecture[7]. The other hyper-parameters were separately chosen:

- L2-regularization constant – 0.001
- Adam optimizer learning rate – 0.01
- Batch size – 512
- Epochs – 22
- Steps per epoch – 400

For the subsequent order of model performance from best to worst, it is in the decreasing order of number of feature blocks for source policy and the decreasing order of total number of blocks for target policy. This revelation is indeed unsurprising for both cases. A major indicator is the fact that the model underfits. So at this stage, more complex models would perform better. From the model architecture, we can see that the feature blocks precede the source policy head so the more blocks there are, the better the performance in source policy predictions. This is also the case for the target policy head since all feature and sandwich blocks precede the target policy head.

As explained during the exploratory data analysis, there is a lack of moves that result in a checkmate. This means that our proposed model can gain an advantage over the opponent but is unable to successfully end the game. Therefore, the Monte-Carlo tree search is used as a data generation process to get the required data. It was used to generate 5,000 board states, with their corresponding actions, starting from incomplete games in the dataset that are forfeited but could have resulted in a checkmate. The number of states generated is significantly below what is needed for proper training of such a deep neural network. However, due to the severe lack of optimization in the `MCTS` implementation, this is unavoidable for our case. Generating the board states of 30 games with 40 rollouts per move took 16 hours. Nevertheless, the nominated (12,10)-`PolicyModel` was retrained on the original dataset with these additional 5,000 board states. Since the sample size is too small for a proper evaluation, we cannot measure it as we had done previously. Instead, we measure its performance through game simulations in the next phase.

The sequences of moves in Figure 5 are sampled from 50 simulated games between 800-ELO and 2000-ELO Stockfish respectively against various models that we created. Expected outcome is used to measure the performance of the models. An expected outcome is defined in $[0, 1]$, where 0 represents a loss from the perspective of the model, 0.5 represents a draw, and 1 represents a win. Thus, a higher expected outcome corresponds to a better performance from the model at the given move number. A uniform random policy is used as a control in this experiment to provide a baseline (lower bound of performance) of our models. The 800-ELO Stockfish also acts as a control in our experiment as a rather intelligent agent with minimal search depth capabilities.

Against the 800-ELO Stockfish (left subplot of Figure 5), the random policy performs the worst, with all games decisively lost in 40 moves or less. Denote M_1 to be the model trained with MCTS and M_2 to be the model trained without MCTS. M_2 performs better initially but deviates from M_1 at the 40-move mark, where M_2 continues to worsen its expected outcome while M_1 recovers performance-wise. This signifies that applying MCTS on the policy considerably improved the performance of the model, particularly at later move numbers. Games between two 800-ELO Stockfish engines are also simulated, and it is observed that the expected outcome in these games oscillate at 0.5, and the game lengths of these simulations are considerably longer than those of the other models; both of these factors indicate the expected outcome to be a draw, which is expected from two players of similar skill.

Against 2000-ELO Stockfish (right subplot of Figure 5), all models perform significantly worse; none of the models are able to draw/win in any of these simulations. However, M_1 and M_2 outperform the other policies simply in terms of delaying their eventual defeat. M_1 and M_2 differs in terms of the phase in which they excel in, which grants a slightly unexpected result. M_1 was earlier conjectured to perform better in later moves but this plot seems to disagree with that due to being the only model that surpasses the 0.5 mark albeit for a brief moment at the start. M_2, which was trained under fewer moves from the later stages of games, seems

[6]The primary reason the experiment led to models that underfit is because of the lack of computational resources to train extremely complex models. Training our most complex model took approximately 8 hours on CPU.

[7]Only 6 blocks are shown for clarity and concision

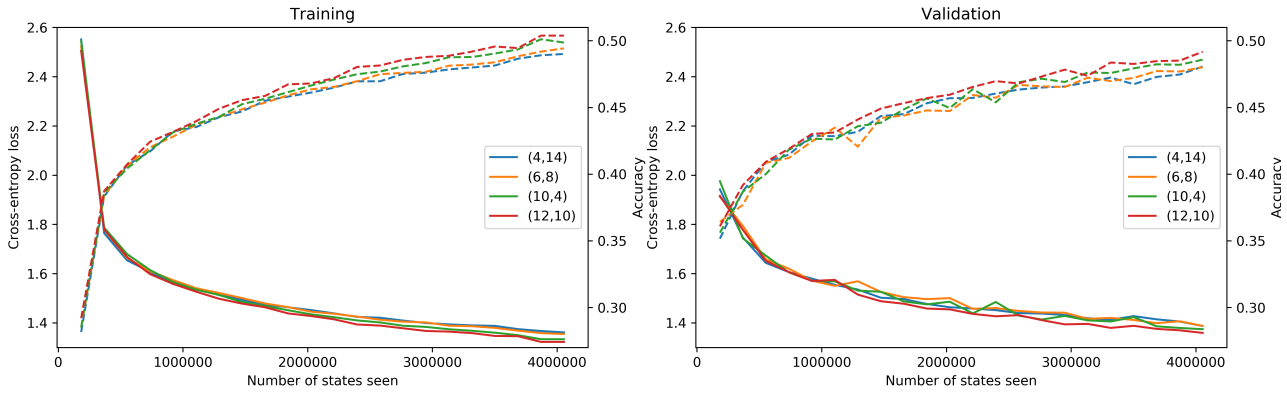

Figure 3: Loss (complete lines) and accuracy (dotted lines) for source policy during training and validation

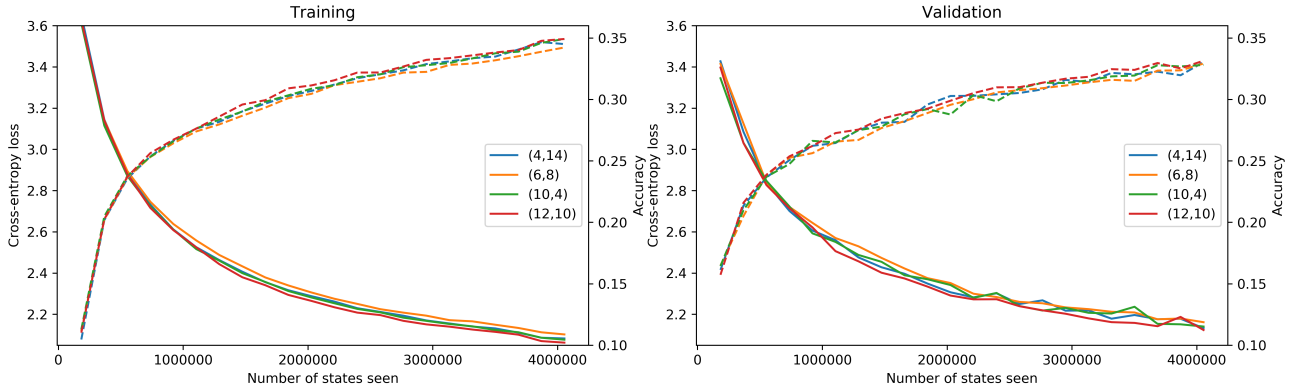

Figure 4: Loss (complete lines) and accuracy (dotted lines) for target policy during training and validation

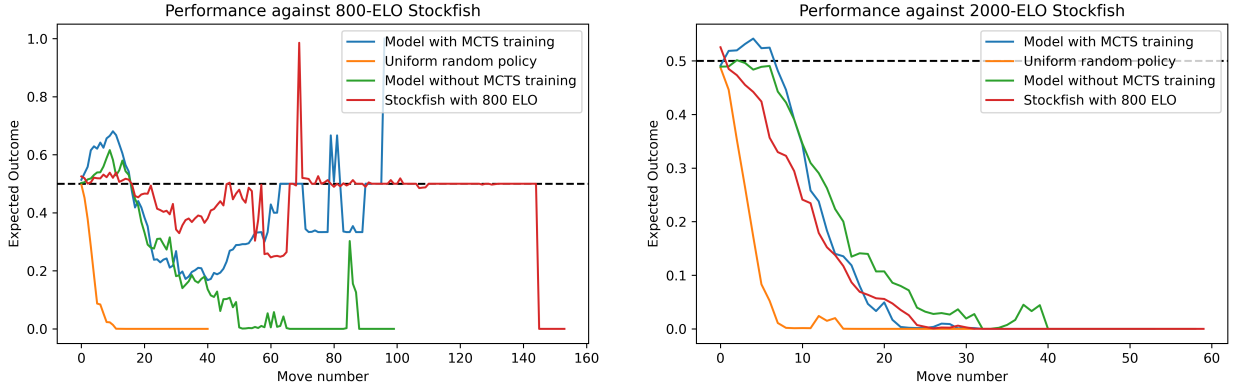

Figure 5: Average of expected outcome graphs in 50 simulations for comparison of model performance against two levels of Stockfish strength. 1.0 indicates that our proposed model wins, 0.5 indicates a draw, and 0.0 indicates that our proposed model loses.

to be able to slightly prolong its survivability. Nevertheless, such manifestations could be attributed to some systematic bias in the Stockfish engine that exploits a particular weakness in M_1 but not M_2.

The data for Figure 6 is sampled by taking the average expected outcomes from 50 simulations between the MCTS-trained model and multiple ELO-rating settings. A characteristic "hump" at the start of the game is observed across

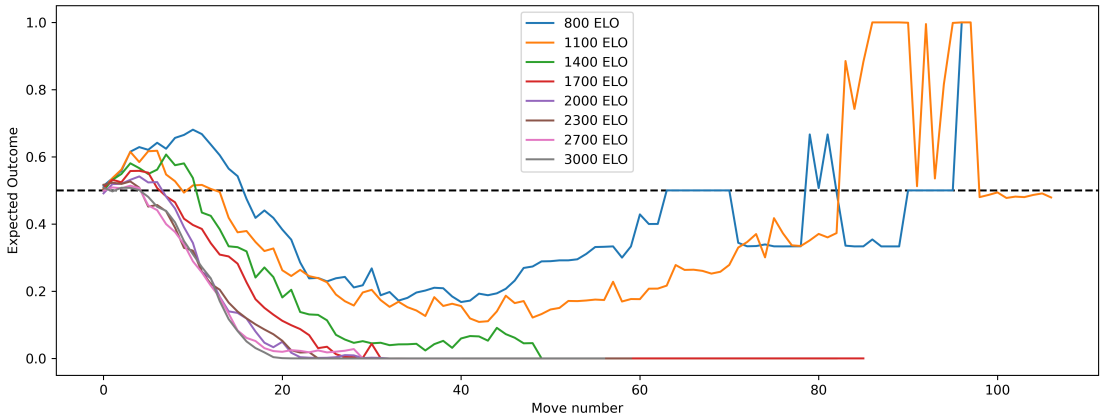

Figure 6: Graph of expected outcome graphs between MCTS-trained model and various Stockfish ELO settings from 800 ELO to 3000 ELO. 1.0 indicates that our proposed model wins, 0.5 indicates that it draws and 0.0 indicates that it loses

all opponent ratings. This is consistent with the prior explanation that the model is heavily trained on opening-game data, where the variability of the moves is at its lowest. Thus, the model has better performance in the game opening, leading to a greater initial expected outcome. In all the games, the model suffers a drop in performance and could only recover against Stockfish with low ELO. This can be interpreted as the MCTS-trained model being capable of challenging Stockfish at settings of up to 1100 ELO.

Lessons Learned

Through the process of the project, we learned how to use the Tensorflow Functional API and the complexities of the application of the Monte Carlo Tree Search algorithm.

In this project, we used Tensorflow to handle the training and implementation of our deep neural network. We specifically chose to use the Functional API because of the flexibility it provides allows, making it suitable for the complexity of our network. Our network produces multiple outputs and has layers which are not connected sequentially, ruling out the possibility of using the more common Tensorflow Sequential API.

Monte Carlo Tree Search is much more difficult to optimize than was originally thought. At each iteration of the algorithm, a full game must be simulated and additionally, the direction of the search is determined by the result of the previous iteration. As such, it is difficult, if not impossible, to simultaneously traverse multiple branches of the tree effectively and efficiently. Also, the enormous number of possible moves makes it computationally expensive to have a reasonably deep tree.

Discussion

Interpretation of Results

The direct outcome from the simulations reveals that our deep policy network when co-trained with the Monte-Carlo tree search algorithm can be estimated to have an ELO of approximately 1100, which is relatively low. Using the probability of winning prescribed by the ELO rating system (Just and Burg 2003), it has a probability of 0.8531 to win against a 800 ELO Stockfish chess engine and a probability of 0.0008 to win against a 2000 ELO Stockfish chess engine. This implies that our research question can be answered quantitatively with these numbers.

Given the constraints of computational resources and time, our policy network, which could be trained from scratch in 24 hours, outperforms our expectations in terms the ability to implicitly recognize game rules, strategy and anticipating player moves. Upon qualitative measures, the agent has proven to be a formidable adversary during human-versus-AI games which is especially good in the first half of the game. The quantitative results from Figure 6 also emphasize the unexpected strength of our agent in the first 10 moves.

With this result, it offers some evidence that the supervised learning component (i.e. the policy pre-training process) performed excellently in quickly learning important starting moves in typical chess matches. It also suggests some evidence that the Monte-Carlo tree search data generation is either not producing enough data or sufficiently "good" moves. This would explain the weakness of the agent in the later parts of the game.

The MCTS co-training seems to also improve the performance in the early moves, as could be seen in Figure 5. The data generation by MCTS was exclusively for generating sequences of moves that result in checkmates since the CCRL dataset lacks those. However, that process inadvertently improved the performance of other moves as well. One plausible interpretation is simply because the policy model is subjected to more training with a more diverse dataset. This could only be verified with another experiment with control to determine if the policy model simply improved regardless how the dataset was derived.

Implications of Results

The closest relevant research paper on the same problem is on DeepChess where they constructed a Siamese network with 1,000,000 chess positions. In contrast, our policy network, whose architecture is similar except for the triple-head output and the deep belief network, was trained with 5,000,000 chess positions. Despite a clear advantage in terms of data availability, our network only achieved a mediocre validation accuracy of 50% and 35% for source and target policy respectively whereas DeepChess had achieved 98%. This could be attributed to the lack of effort in extensive hyperparameter tuning due to the lack of computing resources. As earlier elaborated, the overall architecture of the policy model was inspired by AlphaGo (Silver et al. 2016) which certainly lies in another domain. In DeepChess, it was determined that overfitting is guaranteed to not occur so no form of regularization methods was adopted by their Siamese network (David, Netanyahu, and Wolf 2016). Our policy network, however, had a small degree of regularization in the loss function.

Nevertheless, this comparison is not fully justified as DeepChess utilizes α-β search at inference time while our policy network performs a single-shot prediction. Such form of search greatly improves the prediction due to the existence of some degree of simulation, allowing it to explore with much greater depth than our policy network. Considering that the policy network technically has a search depth of 0, we are convinced that the integration of a search algorithm at inference time would greatly improve the results. Because our policy network does not require search, the short inference time would offer a massive advantage in large-scale deployments. Most online chess engines that cater to millions of users perform client-side predictions which may be slow. Having a single-shot inference without search like our design would greatly speed up prediction times and be more practical since it is not always desirable for amateur chess players to play with 2000 ELO chess engines and above.

Limitations

Our framework for developing a chess AI formalizes our understanding on certain issues that manifest as a result of our choices. The design choice to train the policy $f_\theta(s)$ such that it approximates the search probability π is a common design in other reinforcement learning agents such as AlphaGo (Silver et al. 2016) and DeepChess (David, Netanyahu, and Wolf 2016). However, AlphaGo and DeepChess reinforce the neural network output with additional simulations at run time to improve the predictions since neural networks may not be able to fully encapsulate all complexities without simulations. Our policy network, especially when it underfits, demonstrates this issue when encountering states that are vastly different from what it was trained on. In this case, $f_\theta(s)$ may output irrelevant actions which could have been improved with a MCTS search at run time.

Additionally, our training pipeline is severely lacking optimization. The bottleneck at the MCTS data generation step limited the final policy network to sufficiently explore other strategies besides those that are learned from the CCRL dataset. As a consequence, the policy network is primarily a supervised function approximator of agents in the dataset with limited novelty in terms of its style of actions. However, this is not the limitation of the design but rather a flaw in optimization.

Conclusion

Historically, chess engines have relied on searching for viable and strategic moves. As a result, large amounts of compute power are required to determine favourable moves. By using a deep policy network for determining favourable moves, these large computations can be compressed into smaller constant time computations. Our model accomplishes this by pre-training on sampled data obtained from a dataset of games played between chess engines, and reinforcing itself with the data generated from Monte Carlo tree search. This boost in time performance could possibly have a trade-off with the efficacy of moves, leading us to determine what percentage of wins can a deep policy network, co-trained with the Monte Carlo tree search algorithm, obtain against modern search-based chess engines. Our policy network performs well in the first few moves of the game, but quickly loses its advantage after those moves. This can be explained by an abundance of early-game data and a lack of later-game data. By playing against many different skill levels of the Stockfish chess engine, it is determined that our model has an approximate ELO rating of 1100. With this metric, we can determine that our model has a probability of 0.8531 in winning against a 800 ELO Stockfish chess engine and a probability of 0.0008 in winning against a 2000 ELO Stockfish chess engine.

There must be more work on the current implementation to fully evaluate the efficacy of our design. As of now, the policy network is facing a lack of data from MCTS which is the result of a lack of optimization. Therefore, the immediate next step is to optimize the implementation, preferably incorporating asynchronous multiprocessing.

In addition to improving upon the current implementation, there are countless similar methods we can explore. A k-iteration MCTS can be performed at inference time. By doing so, we can determine the improvement it yields over to our existing design, which would allow us to evaluate the performance-speed trade-off with different values of k. A hybrid of the current method and this method can be formed where k controls the degree of search. This addition could be useful if there is a demand for the *ponder*[8] option as is typically used in Stockfish. The agent can perform MCTS search while a human player is thinking which will not sacrifice speed and could improve performance.

Another perspective of the design is to look at a value-centric approach. While the current policy network does output a value v indicating the probability of winning, the results were extremely inconsistent and hence did not serve useful in the current scope. However, future endeavour can focus on improving the regression output v which can be used as a MCTS search heuristic.

[8]Ponder in the Stockfish engine enables Stockfish to perform search while the opponent is thinking

References

[Abadi et al. 2015] Abadi, M.; Agarwal, A.; Barham, P.; Brevdo, E.; Chen, Z.; Citro, C.; Corrado, G. S.; Davis, A.; Dean, J.; Devin, M.; Ghemawat, S.; Goodfellow, I.; Harp, A.; Irving, G.; Isard, M.; Jia, Y.; Jozefowicz, R.; Kaiser, L.; Kudlur, M.; Levenberg, J.; Mané, D.; Monga, R.; Moore, S.; Murray, D.; Olah, C.; Schuster, M.; Shlens, J.; Steiner, B.; Sutskever, I.; Talwar, K.; Tucker, P.; Vanhoucke, V.; Vasudevan, V.; Viégas, F.; Vinyals, O.; Warden, P.; Wattenberg, M.; Wicke, M.; Yu, Y.; and Zheng, X. 2015. TensorFlow: Large-scale machine learning on heterogeneous systems. Software available from tensorflow.org.

[Bastani, Pu, and Solar-Lezama 2018] Bastani, O.; Pu, Y.; and Solar-Lezama, A. 2018. Verifiable reinforcement learning via policy extraction. In *Advances in neural information processing systems*, 2494–2504.

[Campbell, Hoane Jr, and Hsu 2002] Campbell, M.; Hoane Jr, A. J.; and Hsu, F.-h. 2002. Deep blue. *Artificial intelligence* 134(1-2):57–83.

[David, Netanyahu, and Wolf 2016] David, O. E.; Netanyahu, N. S.; and Wolf, L. 2016. Deepchess: End-to-end deep neural network for automatic learning in chess. In *International Conference on Artificial Neural Networks*, 88–96. Springer.

[Elo 2008] Elo, A. E. 2008. 8.4 logistic probability as a rating basis. *The Rating of Chessplayers, Past&Present. Bronx NY* 10453.

[Fainshtein and HaCohen-Kerner 2006] Fainshtein, F., and HaCohen-Kerner, Y. 2006. A chess composer of two-move mate problems. *ICGA Journal* 29(1):3–23.

[Greenemeier 2017] Greenemeier, L. 2017. How ai has advanced since conquering chess. https://www.scientificamerican.com/article/20-years-after-deep-blue-how-ai-has-advanced-since-conquering-chess/.

[Harris et al. 2020] Harris, C. R.; Millman, K. J.; van der Walt, S. J.; Gommers, R.; Virtanen, P.; Cournapeau, D.; Wieser, E.; Taylor, J.; Berg, S.; Smith, N. J.; Kern, R.; Picus, M.; Hoyer, S.; van Kerkwijk, M. H.; Brett, M.; Haldane, A.; del R'ıo, J. F.; Wiebe, M.; Peterson, P.; G'erard-Marchant, P.; Sheppard, K.; Reddy, T.; Weckesser, W.; Abbasi, H.; Gohlke, C.; and Oliphant, T. E. 2020. Array programming with NumPy. *Nature* 585(7825):357–362.

[Haworth, Regan, and Di Fatta 2009] Haworth, G.; Regan, K.; and Di Fatta, G. 2009. Performance and prediction: Bayesian modelling of fallible choice in chess. In *Advances in computer games*, 99–110. Springer.

[Huang et al. 2017] Huang, G.; Liu, Z.; Van Der Maaten, L.; and Weinberger, K. Q. 2017. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 4700–4708.

[Just and Burg 2003] Just, T., and Burg, D. B. 2003. *United States Chess Federation's Official Rules of Chess*. Random House Incorporated.

[Kocsis and Szepesvári 2006] Kocsis, L., and Szepesvári, C. 2006. Bandit based monte-carlo planning. In *European conference on machine learning*, 282–293. Springer.

[Rosin 2011] Rosin, C. D. 2011. Multi-armed bandits with episode context. *Annals of Mathematics and Artificial Intelligence* 61(3):203–230.

[Silver et al. 2016] Silver, D.; Huang, A.; Maddison, C. J.; Guez, A.; Sifre, L.; Van Den Driessche, G.; Schrittwieser, J.; Antonoglou, I.; Panneershelvam, V.; Lanctot, M.; et al. 2016. Mastering the game of go with deep neural networks and tree search. *nature* 529(7587):484–489.

[Silver et al. 2018] Silver, D.; Hubert, T.; Schrittwieser, J.; Antonoglou, I.; Lai, M.; Guez, A.; Lanctot, M.; Sifre, L.; Kumaran, D.; Graepel, T.; et al. 2018. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science* 362(6419):1140–1144.

[Thrun 1995] Thrun, S. 1995. Learning to play the game of chess. *Advances in neural information processing systems* 1069–1076.

Appendix

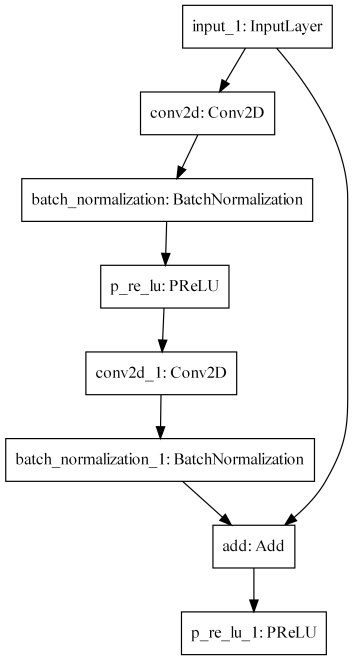

Figure 7: Model Architecture for Feature and Sandwich Blocks

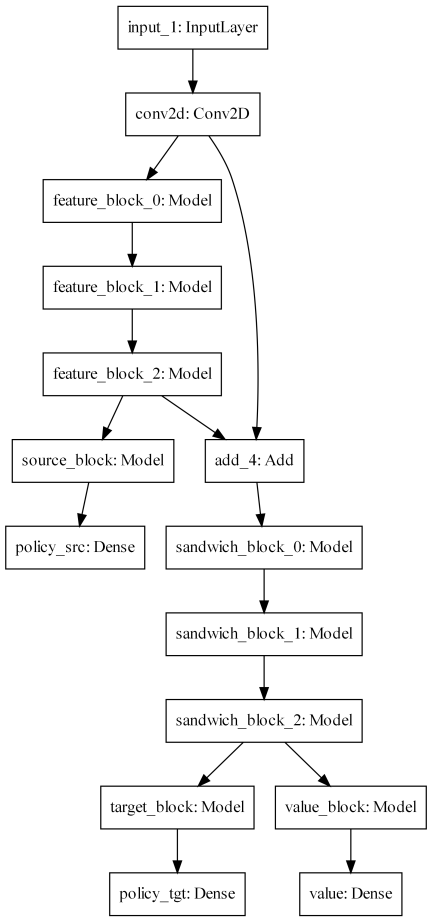

Figure 8: `PolicyModel` Architecture