Reinforcement Learning in Chess

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

Abstract

Complete this section for D4.

The *Abstract* should be at most 150 words long, and should summarize briefly what your project is about. This includes the motivation for the problem (2-3 sentences), the problem you tackled (2-3 sentences), and your main results (1-2 sentences).

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 are 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. This leads to our primary research question: what percentage of wins can a deep policy network when co-trained with a Monte-Carlo tree search algorithm obtain when competing with 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 extracted from games between search-based chess engines, which is obtained from Computer Chess Rating Lists (CCRL). To strengthen its performance, 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 iteration of training and data generation from MCTS, $f_\theta(s)$ has been shown to gradually improve in making moves that results 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 loses 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. $f_\theta(s)$ is subjected to 1000 simulated games with StockFish[1] using a given implementation in Python and its ranking is dictated by the conventions of the ELO rating system. Applying these techniques, we aim to evaluate and investigate the performance of our AI against both human and computer opponents in the real world.

- **For D1, describe the problem and your methodologies only. Then, complete this part for D4.**

  Describe, at a high-level, the problem you tackled and your main results. What research questions are you trying to answer? What methodologies did you used to answer the question? What are the performance measures that you used to evaluate the methodologies? Describe your key findings in 2-3 sentences. (2-3 paragraphs)

- **Complete this bullet point for D4.**

  Emphasize your contributions. How should we interpret the results? Why should people care about this work? Does this project introduce any novel techniques or reveal any unexpected findings? In bullet point forms, list 3-4 key contributions of your project.

Copyright © 2020, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

[1] One of the leading state-of-the-art chess engine

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 pairwise 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 ← Bernoulli random number generator with p = 0.2
for game g in raw dataset do
    v ← outcome of the game g (v ∈ {−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 ← convert b into a 8 × 8 × 13 NumPy array
            a ← 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 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

[2] As of October 18, 2020

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

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.

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:

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

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. To evaluate the efficacy of $f_\theta(s)$, games are run against the Stockfish chess engine with a setting of 2000 as its Elo rating. This corresponds to the competency of a chess *Expert* as defined by the United States Chess Federation (Just and Burg 2003). We compare the win rate of black and white in matches with Stockfish facing itself to the win rate of $f_\theta(s)$ as black and white in matches facing Stockfish. Given this, if $f_\theta(s)$ has

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

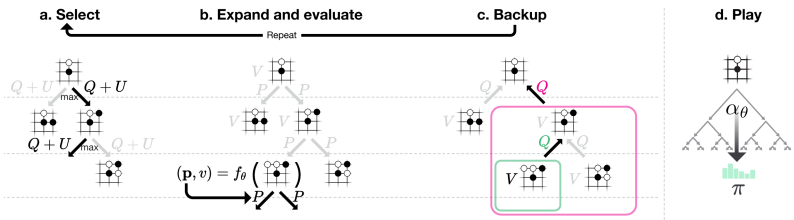

Figure 1: Monte Carlo Tree Search Processes (Silver et al. 2016)

more wins as black and more wins as white than Stockfish does when facing itself, $f_\theta(s)$ has achieved an Elo rating of 2000 or higher. Otherwise, it has not achieved a rating of 2000 or higher. By repeating this process with different Elo ratings for Stockfish, it is possible to estimate the Elo rating of $f_\theta(s)$. Evaluating in this manner provides a quantitative albeit relativistic measure of the performance of $f_\theta(s)$. By using Elo rating, we treat our agent represented by $f_\theta(s)$ as a human player.

The training and validation set for the policy network pre-training were constructed from splitting the complete data, which consists of 5,000,000 board states, action and outcomes, into 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 12 *residual towers* which each comprises of 2 convolutional layers with 256 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 5 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).

Complete the following two paragraphs for D3.

Describe the findings from your evaluation. Describe both (a) how well your techniques worked, and (b) what you learned about the problem through these techniques.

Prepare figures (e.g., Figure 2) and tables (e.g., Table 1) to describe your results clearly. Make sure to label your figures and tables and explain them in the text. If you are comparing the performance of algorithms, include statistical tests to assess whether the differences are statistically significant. If possible, describe how your techniques compare to prior work.

Techniques	F-1 Score
Baseline	0.80
Another Baseline	0.76
My Awesome Algorithm	**0.95**

Table 1: example of a table summarizing the results

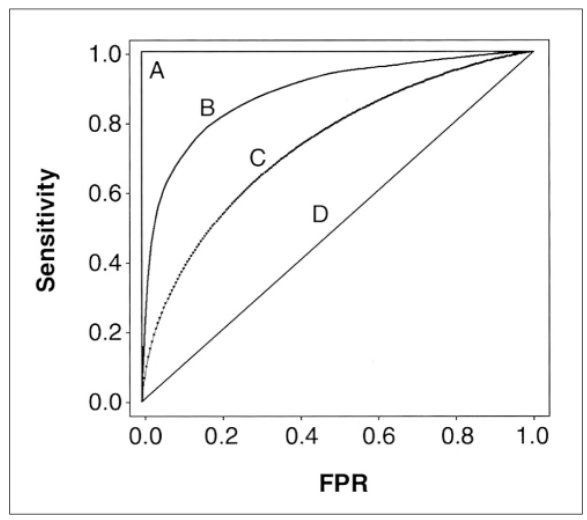

Figure 2: ROC curve of my awesome algorithms

Discussion

Complete this section for D4.

The *Discussion* section (∼1 pages) describes (a) the implications of your results, and (b) the impact and the limitations of your approach.

For the results, describe how a reader should interpret them. Try to form concise take-away messages for the reader. For your approach, describe the extent to which your approach helps to solve the problem. Describe any limitations of your approach. If possible, compare your results and your approach to that of prior work.

Conclusion

Complete this section for D4.

The *Conclusion* section (∼0.5 pages) provides a brief summary of the entire paper. In this section, describe

- the motivation, the problem, and your results, and
- 3-4 promising future directions.

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.

[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.