Chess Neural Network Trained on Algorithmically Generated Data

University of California, Merced

Math 180

Professor: Harish Bhat

TA: Derek Sollberger

Author: Philip Felizarta

Math 180 Course Project

December 11, 2020

Objective

As a chess enthusiast, I am interested in using the classification and regression techniques I learned in class to create a chess engine. Traditionally, modern chess engines use Alpha-Beta search to produce move suggestions. However, this project will follow in the steps of DeepMind's AlphaZero[8] and create a model f_θ that produces a policy vector $\vec{p}$ (representing a distribution over chess moves) and a predicted outcome of the game $v \in (-1, 1)$ for a given chess board s.

$$f_\theta(s) = (\vec{p}, v)$$

The model will be trained via Supervised Learning on generated data set.

Generating Data

The state-space of chess positions is extremely large with an upwards of 10^{46} possible legal positions. As a result, the training data must be carefully justified and/or massive. Silver et al.[8] and Lai et al. [7] both use Reinforcement Learning frameworks where positions are generated by the model via self-play. Though these methods are attractive for their tabula rasa approach, reinforcement learning requires the generation of millions or billions of positions and hundreds of epochs of training to produce high performing models. In the case of AlphaZero, 48 million games (full games, not positions) were generated during its training process. The cost to replicate such an experiment would amount to millions of dollars! As a student researcher that looks to circumvent these tremendous costs, I use an approach inspired by the Go-Explore algorithm [3] and a famous quote by former chess world champion Gary Kasparov.

"By strictly observing Botvinnik's rule regarding the thorough analysis of one's own games, with the years I have come to realize that this provides the foundation for the continuous development of chess mastery." - Gary Kasparov

To solve hard exploration problems like Montezuma's Revenge, Ecoffet, et al. [3] periodically uses an expert policy to explore promising states and then trains off of the trajectories produced by the expert. In chess, converting a winning position is extremely complicated as there are often numerous tactical resources for your opponent to punish inaccurate play. Moreover, there are often very deep reasons for not playing intuitive moves. As a result, I hypothesize training a model on self-play games of Stockfish alone would not produce a model that performs well tactically. Instead, I propose thorough analysis of a 2,500 Stockfish 12 self-play games.

Base Trajectories

To generate the training data for this project, Stockfish 12 was used to create 2,500 base trajectories of 20 ply in length. To convert Stockfish's deterministic centipawn values into a policy that can be sampled from stochastically, I apply the softmax function to the set of the normalized centipawn evaluations p_a of the action-space $\mathcal{A}$.

$$\vec{\pi}_{\text{base}} = \text{softmax}(\{\frac{p_a}{10} \mid a \in \mathcal{A}\})$$

The base policy $\vec{\pi}_{\text{base}}$ is calculated at each position in the self-play games and sampled from to generate the next move in the base trajectory. I look to a stochastic process that may not play the best moves to generate the base trajectories because I want the model to be able to convert winning and defend losing positions that are “close” to positions Stockfish encounters in a normal game.

Expert Trajectories

For each state in each base trajectory I will generate an expert trajectory. This process is similar to Go-Explore [3]; however, these expert trajectories are played out until a goal state is reached. Instead of sampling from stochastic process, the expert trajectories are formed using the best move suggestion from Stockfish at each state. The best move suggestions are converted to one-hot vectors to be used as the labels for the classifier, while the value-head labels are formed using the following equation derived from the Deterministic Bellman Equation:

$$V(s) = \max_a(R(s,a) + \gamma V(s'))$$

Since $R(s,a) = 0$ unless a goal state is reached, the regression labels are formed via the simplified form:

$$V(s_t) = \gamma^{t-1} R$$

Where $R \in \{-1, 0, 1\}$ is the result of the self-play game and t is the number of ply between the current state and the goal state. This project uses $\gamma = 0.99$ during the generation of the data set. $\gamma = 0.99$ rather than 1 to preserve a sense of distance from the goal state. Moreover, the expert policy used in this project is Stockfish’s best move suggestion after 100ms of search on a single CPU core. Though Stockfish 12 at this very fast evaluation is superhuman, it is nowhere near full strength and can make potential inaccuracies that distort the true value of a position. γ should be very close to 1 since chess is deterministic, and the value of a given position should be equal to the next if the best move is played.

Parallelization

A great advantage of generating the data using base and expert trajectories is its potential for parallelization. The algorithm used to generate the data set creates a new process for each expert trajectory after the main process gives a new position within the base trajectory. However, despite these advantages, generating the data set took my machine (16 core AMD Ryzen Threadripper) $\sim$ 3 hours. The runtime could be shortened if you reduce the evaluation speed of the base trajectory, but this comes at the cost of reducing sample relevancy (Note the evaluation at the expert trajectories affects the labels, while the evaluation at the base trajectory controls what positions the algorithm samples self-play games from).

Analysis of Data

Base Trajectory Analysis

Due to the sheer amount of positions that are generated in this project, meaningful analysis of what positions are played is limited to the positions produced in the base trajectories. To analyze the "quality" of the self-play games and the number of repeated base trajectory positions, I'll use the free database software SCID. The software creates a tree of positions played in the set of 2500 trajectories formed allowing easy access to what openings are played.

ECO	Frequency
B20 (The Sicilian)	248
A10 (The English)	256
A04 (Reti Opening)	216
C20 (King's Pawn)	170
D00a (Queen's Pawn)	138
A45a (Indian Game)	104
C00a (The French)	98
B10a (The Caro Kahn)	64

In my analysis, 75.9 percent of base trajectories started with 1.e4, 1.d4, 1.c4, and 1.Nf3. To avoid repeated examples, positions seen in the base trajectories only start an expert trajectory if they are unique.

Structure of the Data

Model Input

To take advantage of the spatial patterns that occur in chess, this project uses convolutions in the model. As a result, the model inputs are three dimensional volumes (8x8x17). Each 8x8 plane describes a feature on the chess board. The first 12 planes (6 planes for white pieces, 6 planes for black pieces) encode the location of the pieces (1 for if a piece is on that square, 0 if not). The last 5 planes encode some extra rules, such as: white or black to move (plane of all 1's or 0's), kingside castling rights (1 plane for white, 1 plane for black), and queenside castling rights (1 plane for white, 1 plane for black). My research differs from AlphaZero [8] because it does not include previous moves in its input (making the input volume much smaller).

Classifier

Each class in the classifier output of the model $\vec{p}$ represents a UCI (Universal Chess Interface) command. UCI commands represent chess moves by denoting the transfer of a piece from one square to another. Moreover, there is a command for each type of promotion in chess (knight, bishop, rook, and queen). As a result, the model ends up with 1968 different classes.

At run-time, the model will need an environment to produce a mask over the legal moves and re-normalize the policy distribution accordingly.

Value-head

The model uses a tanh activation function to scale its value output $v \in (-1, 1)$.

Analysis of Data Set

Though the data set generated by the algorithm is composed of 4,029,624 examples, some moves in chess are extremely rare. In fact, all classes not found in the data set are promotions to knights and bishops. This is because promotions of these kind are very rarely justified in game. As a result, the model is expected to never predict those classes. Performance as a whole should not be affected by the model's incompetence at identifying these particular promotions. Conversely, there are extremely frequent classes in the data set due to their utility in each game. For example, the moves e2e4 and d2d4 are often thematic moves that show up once per chess game. Figure 1 displays the actual frequency of each class in the data set.

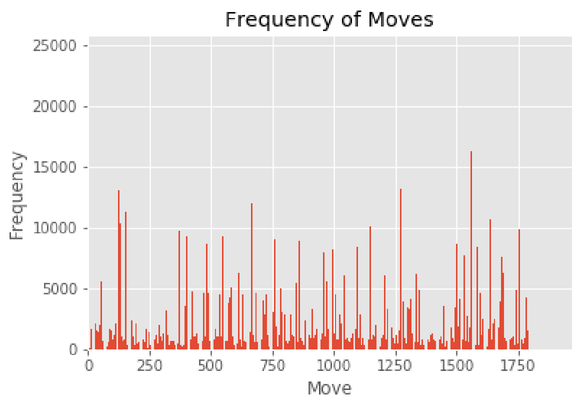

Figure 1: The figure displays the frequency of each class in the data set. Some moves (like bishop and knight promotions) are never seen due to their rarity in high-level chess. Other moves like e2e4 e7e5 have very high frequency due to their utility in game.

Chess is theorized to be a drawn game, meaning one should expect the outcomes of the expert trajectories to be a majority of drawn games. Figure 2 showcases that the data set contains roughly half of its data from drawn games. Interestingly, when the game is decisive, white seems to be more favorable (this coincides with common chess theory that white has a first move advantage). For the purposes of the data set, the number of decisive games is beneficial since the model will need concrete examples of converting wins for the network to perform well in an actual game.

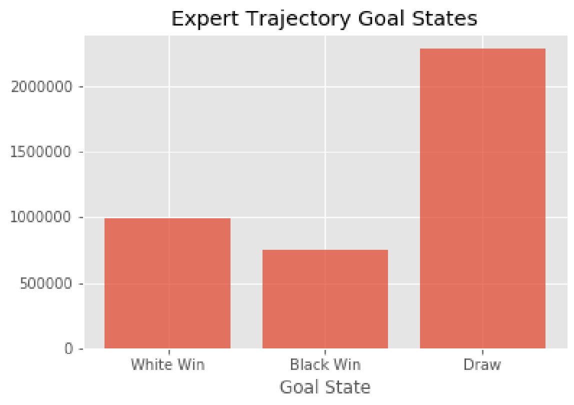

Figure 2: Figure shows the outcomes the expert trajectories. One would expect chess to be drawish and white to have some advantage in decisive games.

The effects of $\gamma = 0.99$ in the Bellmen Equation to generate label for the value-head can be observed in Figure 3. Since γ is so close to 1, the algorithm reaps the benefit of labeling positions based on how far they are from checkmate while maintaining accuracy in evaluating the winning chances of a given position.

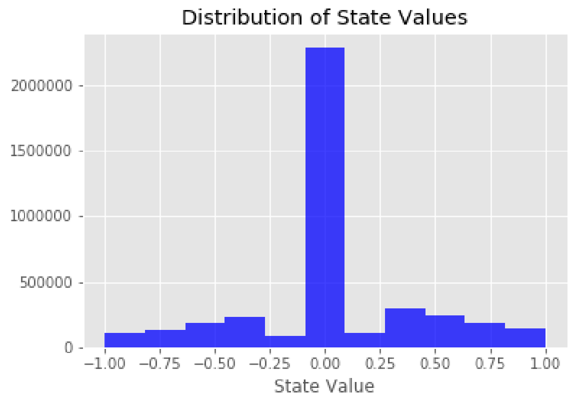

Figure 3: Figure displays the actual distribution of value-head targets in the data set. $\gamma = 0.99$ seems keep the absolute values of decisive positions between 0.25 and 1. To positive tail is probably fatter due to white's first move advantage.

The Model

Model Architecture

Due to computational constraints, I use a relatively small 64 filter by 6 block Residual Convolutional Neural Network [4]. Each residual block uses Batch Normalization [6] and Squeeze Excitation [5] to speed up the learning process.

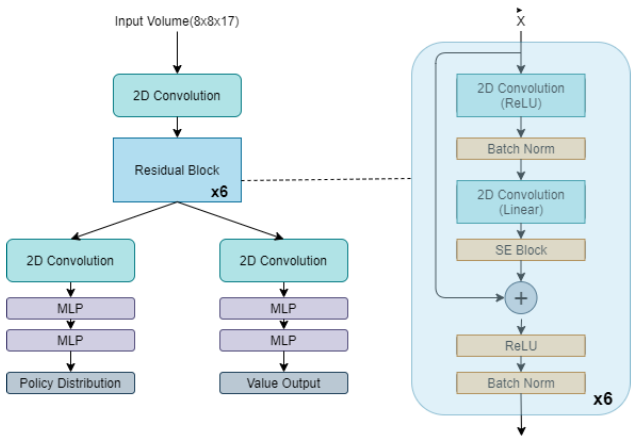

Figure 4: This model has two output heads for better efficiency during tree search. Batch Normalization and Squeeze Excitation blocks are used to speed up learning.

Objective Functions and Regularization

The objective function used to train the model is a weighted sum of the individual losses for each head and the l_2 norm of the model parameters θ. The weights of the value head c_1 and l_2 regularization term c_2 are determined by an active learning algorithm. $\vec{\pi}_{\text{expert}}$ and z are the move and value labels, respectively.

$$\mathcal{L}(\theta, \vec{\pi}_{\text{expert}}, z) = \mathcal{L}_{\text{CE}}(\theta, \vec{\pi}_{\text{expert}}) + c_1 \mathcal{L}_{\text{MSE}}(\theta, z) + c_2 ||\theta||_2^2$$

$$\mathcal{L}_{\text{CE}}(\theta, \vec{\pi}_{\text{expert}}) = -(\vec{\pi}_{\text{expert}})^T \log(\vec{p})$$

$$\mathcal{L}_{\text{MSE}}(\theta, z) = \mathbb{E}[(v - z)^2]$$

Tuning Hyperparameters with a Gaussian Process

If the training of the model is framed as an extremely expensive function (one that takes 3 hours to call) $V(\vec{\lambda})$ where $\vec{\lambda}_i$ is a hyperparameter of the model we wish to tune, then function approximation can be used to quickly find the global optimum of $V(\cdot)$ over a finite set of hyperparameters $\mathcal{X}$. A Gaussian Process paired with an active learning algorithm was used to approximate $V(\cdot)$. This process was inspired by AlphaGo's own tuning process [2].

Active Learning Algorithm

To accurately approximate the global minimum of $V(\cdot)$ in very few function calls ($\sim$15), one must be selective of what points they sample from $\mathcal{X}$. This is precisely why I use a Gaussian Process. Not only do they produce strong approximations with very few points, they also provide us with a measure of uncertainty. With this measure of uncertainty, it is possible to algorithmically sample points that are potentially better. Given a Gaussian Process:

$$GP(\lambda) \sim (\mu(\vec{\lambda}), \sigma(\vec{\lambda}))$$

and the Expected Improvement acquisition function [1]:

$$a_{\text{EI}}(\vec{\lambda}) = (V^* - \mu(\vec{\lambda}))\Phi(\frac{V^* - \mu(\vec{\lambda})}{\sigma(\vec{\lambda})}) + \sigma(\vec{\lambda})\phi(\frac{V^* - \mu(\vec{\lambda})}{\sigma(\vec{\lambda})})$$

Where V^* is the best loss found so far, Φ and ϕ are the cdf and pdf of the normal distribution, respectively. Sampling the next point in the algorithm is done via the maximization of the acquisition function $a_{\text{EI}}(\cdot)$.

$$\vec{\lambda}_{\text{next}} = \arg\max_{\vec{\lambda} \in \mathcal{X}} a_{\text{EI}}(\vec{\lambda})$$

After calling $V(\vec{\lambda}_{\text{next}})$, the algorithm retrains the Gaussian Process on the new data, and maximizes the new Expected Improvement recursively. After 15 iterations of active learning, I take the best performing (lowest Validation Loss $V(\cdot)$) hyperparameters from the set of hyperparameters I sampled from. In total, this process took 45 hours on my machine. Perhaps with parallelization of function calls one could shorten this time significantly, as the main bottleneck is the time it takes to train a network. Attempts to use much smaller data sets did not produce good results, so this algorithm trains a 6x64 Residual Network on 90% of the Training Set and computes the Validation Loss on the other 10% of the Training Set. Figure 5 lists the hyperparameters determined by the active learning algorithm for the 6x64 ResNet architecture.

Selection of $\mathcal{X}$

The utility of this algorithm crutches on the selection of $\mathcal{X}$. $\mathcal{X}$ is a finite set that you have to take $\arg\max$ and $\arg\min$ over, thus there is a significance to selecting relevant points. A set $\mathcal{X}$ of 50,625 hyperparameter combinations was formed by sampling 15 points of interest for each hyperparameter (learning rate, l2 regularization constant and value head importance) and forming a grid-space over them. Due to the nature of the learning rate and l2 regularization constant, the 15 points are exponentially spaced between a specified minimum and maximum. Figure 6 displays the final step of the active learning algorithm.

Hyperparameter	Value
Learning Rate	2.49735e-1
L2 Regularization Constant	2.68270e-4
Value Loss Coefficient	2.68270e-1
Momentum	5.28571e-1

Figure 5: The table displays the hyperparameters determined by the active learning algorithm.

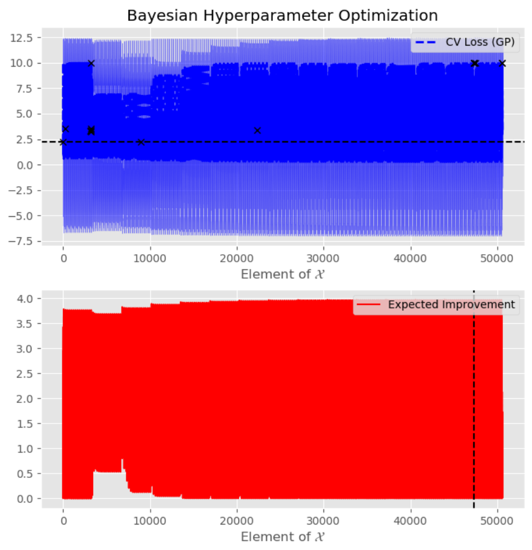

Figure 6: The figure displays the estimated performance of hyperparameter combinations using the trained Gaussian Process and the resulting expected improvement values. The x-axis denotes the i^{th} element of $\mathcal{X}$. (Note the graph displays the objective function as though it has a one dimensional input for visual purposes only.)

Tree Search

Due to the very tactical nature of chess, tree search is required for accurate play. Similarly to AlphaZero[8], this paper will use Monte-Carlo Tree Search with a modified PUCT leaf policy to generate move suggestions at test time.

$$PUCT(s, a) = Q(s, a) \pm p_i C_{\text{puct}} U(s, a)$$

Where $Q(s, a)$ is the average value of the node's child (after action a), $U(s, a)$ is the upper confidence bound of the node's child (after action a), p_i is the neural network's corresponding classification prediction for that action, and C_{puct} is a constant controlling the exploration of the Monte-Carlo Tree Search. The child action $\arg\max_{a \in \mathcal{A}} PUCT(s, a)$ is iteratively selected to retrieve a leaf-node per simulation.

Results

After determining the best set of hyperparameters using the active learning algorithm, the 6x64 ResNet was trained for 25 epochs on the entirety of the Training Set. The network's accuracy and respective losses are then evaluated on the Test Set. Figure 7 displays these results, while Figure 8 showcases how the model's performance evolves over training.

6x64 ResNet Performance	
Training Cross Entropy Loss ($\vec{p}$)	1.7709
Test Cross Entropy Loss ($\vec{p}$)	2.1505
Training Accuracy ($\vec{p}$)	44.84%
Test Accuracy ($\vec{p}$)	34.24%
Training Mean Squared Error (v)	0.0394
Test Mean Squared Error (v)	0.0405

Figure 7: The table denoting the final model's performance on the Training and Test Sets. The policy head of the model is not able to learn the data set very well. This fact is reflected in the accuracy of both the Training and Test sets.

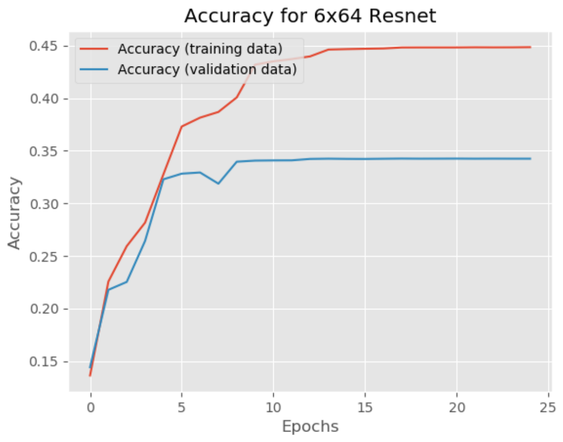

Figure 8: The figure displays training and test accuracy of the final model over its 25 epochs of training. It seems the model needs significantly more capacity to perform better on the data set. Unfortunately, it already takes 3 hours for a single training run for a network of this size with the GPU (RTX Titan) used in this experiment. Making the network any larger would significantly increase the time it takes to conduct the model training and hyperparameter search portions of the paper.

To test the network's playing strength, 25 games were played (800 Simulations per Move) vs Stockfish 12 (at 100ms) for varying values of C_{puct}. The games were then analyzed using Lichess.org's Stockfish 12 cloud engine to retrieve centipawn losses. Table 9 displays these mean centipawn losses per move. The implementation of Monte-Carlo Tree Search used (written by me) only utilizes a single thread; thus, the resulting experiments each take around an hour to complete.

Engine Performance at 800 Nodes per Move	
C_{puct} Constant	Mean Centipawn Loss
0.0	50.849
0.6	44.052
1.2	48.407
2.5	44.186
5.0	39.720
1000.0	49.682

Figure 9: Mean Centipawn Losses per move (calculated using Lichess.org cloud analysis Stockfish 12) for varying C_{puct} constants calculated over 25 games vs Stockfish 12 at 100ms. Centipawn loss per move measures how far in quality an agent's average move is from Stockfish 12.

Discussion

This study uses centipawn loss to evaluate the performance of the model when combined with tree search, but centipawn loss seems to be an inconsistent metric at evaluating performance in general. For instance, centipawn loss does not reflect the difficulty of the positions encountered. To avoid any of the inconsistencies, the agents were tested against Stockfish 12 at 100 ms per move (the same settings used to create the labels of the network) to ensure critical positions are reached in these evaluated games. The results are enough to conclude, however, that the engine does not play at superhuman strength at 800 simulations of tree search. In the 2018 World Chess Championship, Magnus Carlsen (the world champion) and Fabiano Caruana (the challenger) both played with less than 8 centipawn loss per move.

Qualitative analysis of each game (done by me) seems to indicate that the engine blunders quickly in tactical positions but finds good positional moves in quiet positions. This is most likely due to the PUCT selection policy's tendency to search much deeper than needed (rather than investigating shallow tactics) during tree search. Though the p_i term in the selection policy calculated by the model should circumvent this problem, the low performance of the policy head on the training data probably results in "bad" values for p_i. Perhaps a network with a higher capacity to learn the data set could result in a much stronger engine.

Though the data set is composed of 4 million examples, the complexity of chess most likely calls for a data set that is orders of magnitude larger. Alphazero [8] trained on 48 million games which is presumably billions of examples. This study was limited by the memory available on my system, but a modified implementation (that did not load the data set into RAM) of the training procedure could definitely allow for a data set in the billions. However, a data set that is 100 times larger than the one used in this study would take 100 times longer to generate, find good hyperparameters for, and train!

Conclusion

The methods presented in the paper fall short of producing a superhuman chess engine, but I hypothesize that scaling or slightly modifying the experiment may produce a much stronger network. A notable detail about the engine's play is its deviations from openings that it has trained on. This may significantly impact its performance as any good chess player knows you cannot play openings and their resulting middle game positions if you have not thoroughly prepared for them. Reinforcement Learning methods like Alphazero [8] solve these issues, as the network continually trains on positions that it selects. By scaling this experiment to have 100 times more base trajectories, it may be reasonable to assume the network's opening selection would be much less of an issue. Another idea would be to replace the base trajectory policy with the network's policy (instead of the modified Stockfish 12 used in this experiment) and then iteratively train the network like in Reinforcement Learning. Both of these extensions to the research would require much more computational resources to achieve in a timely manner.

References

[1] Eric Brochu, Vlad M. Cora, and Nando de Freitas. A tutorial on bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning, 2010.

[2] Yutian Chen, Aja Huang, Ziyu Wang, Ioannis Antonoglou, Julian Schrittwieser, David Silver, and Nando de Freitas. Bayesian optimization in alphago, 2018.

[3] Adrien Ecoffet, Joost Huizinga, Joel Lehman, Kenneth O. Stanley, and Jeff Clune. Go-explore: a new approach for hard-exploration problems, 2019.

[4] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015.

[5] Jie Hu, Li Shen, Samuel Albanie, Gang Sun, and Enhua Wu. Squeeze-and-excitation networks, 2019.

[6] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift, 2015.

[7] Matthew Lai. Giraffe: Using deep reinforcement learning to play chess, 2015.

[8] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. Mastering chess and shogi by self-play with a general reinforcement learning algorithm, 2017.