Developing a Human-Level Chess Player with Artificial Neural Networks

Nick Van, Lawrence Lee, Joel Lee, Pavit Bath,
Muhammed Halbutogullari, Mohamed Shafi, Ezekiel Morton,
Alexander Recalde, Vivienne Chiang, Nikhila Thota, Zain Munad

Fall 2019

Abstract

The game of Chess is one of the most commonly tackled challenges in the field of machine learning. We designed and implemented several chess players, using different variants of artificial neural networks (ANNs) as the evaluation function for the alpha-beta pruning algorithm to determine the best possible next move at every juncture. We used Feed-Forward Neural Networks (FFNNs), and a combination of Deep Belief Networks (DBNs) and Siamese Networks based on the DeepChess architecture [8] to design our chess players, which are able to be trained on a personal computer. These networks were trained on publicly available game data from Lichess.org (a popular online chess server). Our fully trained models performed competently enough to reliably beat a casual human player. Our chess player modeled on the DeepChess architecture comes close to being on par with open-source chess engines StockFish and Crafty.

Introduction

The game of Chess originated around the 7^{th} century, and is played by millions of people around the world. Chess is a two player strategy board game, played on a 8x8 checkerboard. Each player has 16 game pieces of 6 unique types, each with their own rules of movement and play. The complexity of the game has made programming a chess player an attractive challenge to many. Chess gained its popularity among computer scientists in 1996 when the first supercomputer, IBM's Deep Blue, defeated then world champion Garry Kasparov in a game of chess [16]. More than two decades later, increased computational capabilities have allowed for machine learning experiments to be carried out by anyone with access to a laptop.

Our research concentrated on determining the feasibility of using various artificial neural networks to construct a competent chess player. Creating this

artificial intelligence was achieved with the support of easily accessible open-source Python libraries, such as chess interface python-chess [3] and neural network library Keras [1], which was built on top of dataflow library TensorFlow [2].

Data Processing

We sourced a publicly available data set [4] consisting of game data from popular online chess server Lichess [21], made available through Kaggle, an online machine learning community and data set repository. This data set contained over 20,000 online games between players of varying Elo ratings, including data on all game moves (in standard chess notation), opening variations and game winners.

Our objective for the data set was to manipulate it for optimal use with our models. The data was used to train a neural network to identify how likely a given chessboard is to be part of a game where the player's side wins. This likeliness is referred to as the board fitness, where a higher fitness score predicts a higher chance of victory for the player's side. For every move in a game, our chess player determines the move (of all legal moves) which results in the highest fitness score. By doing so, the model is effectively striving for maximal fitness, which equates to making the move which results in the maximal probability of securing itself a victory.

For data preparation, we removed all games from the data set that resulted in a draw, as the inclusion of such games would have no benefit to the chess player. This narrowed our data set to around 19,000 samples. We also removed all games where the Elo rating of the white player was less than 1750, which is the Elo rating of an intermediate-level tournament player. This narrowed our data set to just 5,400 games. From this reduced data set, we used the list of moves for each game to generate a sample for each move of each game. Our final data set consisted of around 300,000 samples, each labelled with which side won the game.

One caveat with this method is that all board states from a single game are given the same label. This means that meaningful board states that have a higher impact on the result of the game are given the same label as less meaningful ones. This may result in a neural network without a strong prediction for less meaningful moves. This may be a possible point of failure since meaningful moves are crucial for the neural network to concisely learn how a approach a victory.

Our approach to chessboard encoding was to convert each board state into a naïve feature vector. This is done by flattening the chessboard matrix into an array of size 64 that contains different numbers to represent the different pieces on each of the 64 squares on the chessboard. The numbers range from -6 to 6, with 0 representing an empty square, 1 to 6 representing the 6 unique pieces for the white player, and -6 to -1 representing the pieces for the black player. We increased the length of the feature vector by 5 to indicate both players'

queenside and kingside castling rights (4 inputs, either 0 or 1) and the current player's turn (1 input). This representation allowed for faster computation in implementing our neural networks as compared to using a bitboard (773 bits per chessboard).

Approach

Convolutional Neural Network Architecture

Initially our plans involved implementing a Convolutional Neural Network (CNN) to model an effective chess player. Ultimately, our research indicated that this approach had been taken in the past with limited success due to challenges inherent to using a CNN approach for chess [14]. As detailed by authors Oshri and Khandwala, each move on a chess board correlates to a significant change in board state, especially compared to a single move in the game Go where CNNs have had proven success. As such, it may not be possible to sufficiently represent the logical decision making process used in chess (based mostly on domain knowledge) in the activation layers of a CNN. The performance of the CNN detailed in the paper against the Sunfish chess engine proved these concerns to be valid. Out of 100 games, 26 were drawn while the remaining games were lost. While the authors suggested some methods to address the aforementioned shortcomings (evaluation functions that apply domain-specific knowledge), we ultimately decided that the use of the next two architectures would likely lead to superior results than any hurried attempt to adapt a CNN to be competitive.

Feed-Forward Neural Network Architecture

The first part of our methodology focused on training a feed-forward neural network to perform pseudo-classification with our first prepared data set, using 64 input nodes (corresponding to the board squares of the feature vector) and 2 output nodes (corresponding to the probability of each player winning). Between using a sigmoid and using a softmax function, we elected to use a softmax function for the output activation function, in order to force the sum of the outputs to be equal to 1. We chose a categorical cross-entropy function as the loss function for this model, which is in alignment with our goal of classification.

We performed a parameter sweep on the number of hidden layers and the number of hidden nodes per layer. To keep the parameter sweep simple, we tested combinations of limited sets of parameters, namely 2, 4, 6, and 8 hidden layers and 4, 8, 16, 32, and 64 hidden nodes per layer, for a total of 20 combinations of parameters. We created models for all combinations and trained them while keeping all other parameters static. With 200,000 sample board states in the training set and a batch size of 100, we decided to train the models for 5 epochs each. Following training, we evaluated the models on a test set containing 10,000 sample board states. We plotted the training and test losses of all 20 models as functions of the epoch, one for each different activation function

for the hidden layers:

The first hidden-layer activation function we tested was the sigmoid function. This provided us with promising training results. However, the testing error plateaued around 0.68 and did not improve over iterations.

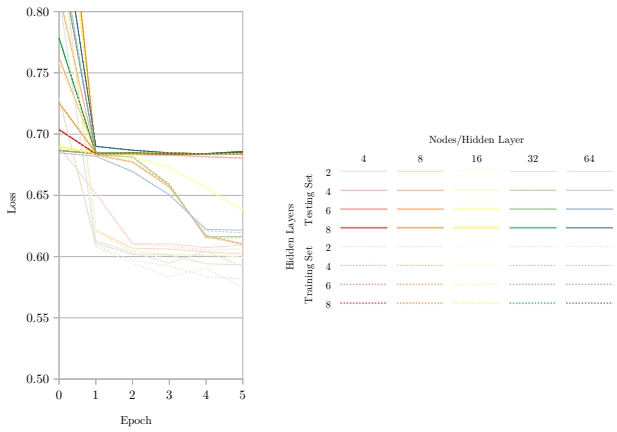


Figure 1: Sigmoid Hidden Layer Activation Function

The next hidden-layer activation function tested was the Rectified Linear Unit (ReLU) function. Using a ReLU activation function improved our training results remarkably.

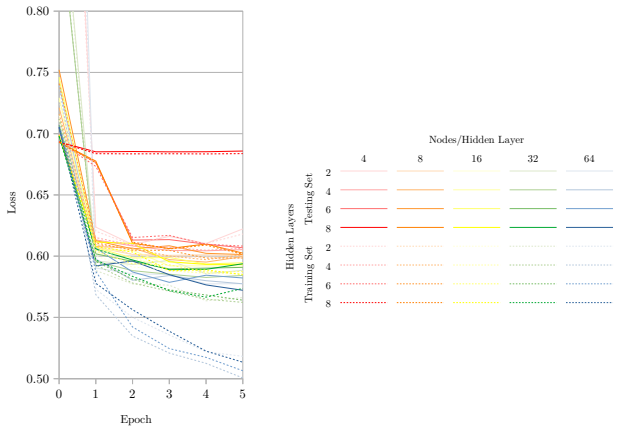


Figure 2: ReLU Hidden Layer Activation Function

The final hidden-layer activation function tested was a leaky ReLU function, which had similar performance to the ReLU function, except for the results being

slightly more uniform. We noted a clear plateau past 3 epochs that was simply not improving the results as rapidly as the ReLU activation had.

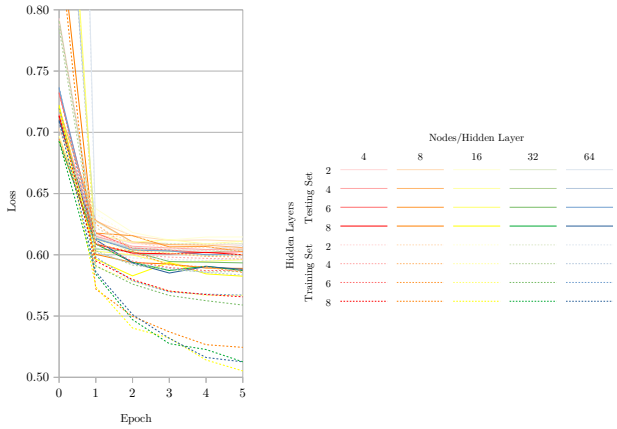


Figure 3: Leaky ReLU Hidden Layer Activation Function

From these results, we decided to use ReLU for the hidden-layer activation function.

In general, the models also exhibited a negative correlation between the number of nodes per hidden layer and the training and test losses. That is, an increase in the number of nodes per layer resulted in a decrease in the training and test losses. An important observation is the discrepancy between the training and test losses. The difference is higher for for the models with a greater number of nodes per layer. This is especially apparent when each layer has 64 nodes. We interpret the discrepancy as a precursor to over-fitting. The number of hidden layers, however, had a weaker impact on the losses. For any given number of nodes per layer, the number of hidden layers did not result in large changes to the losses. It is interesting to note the effect of deep networks on models with 4 nodes per layer. Losses remained comparatively high for the models consisting of 8 hidden layers with 4 nodes per layer. Due to the negative correlation between loss and the number of nodes per layer, we concluded that it was best to use a greater number of nodes. We omit the 64 node option out of fear of over-fitting, so we chose 32 nodes. As for the number of hidden layers, we decided on 8. We speculate that a deeper network will be better after more training without causing over-fitting.

In summary, we training our first data set on a feed-forward neural network constructed with 8 hidden layers, with 32 nodes per layer, using ReLU hidden-layer activation functions and a softmax output activation function. These hyperparameters were selected for their ability to handle complexity, and also because of their impressive training and testing results. Training and testing errors dropped under 0.60, showing continued signs of improvement with the epochs, unlike the other functions that were tested.

DeepChess Architecture

The second part of our methodology was derived from an architecture called DeepChess [8] that relies on deep neural networks. First, unsupervised training occurs on a deep neural network in order to extract the best features from a given position. Then, using supervised training, we train the chess player to choose the best position out of two options through a version of the alpha-beta algorithm.

In order to implement the DeepChess architecture, we further processed the data to remove the first five moves from each game and any moves that were captures. This was done to improve the model, as the opening moves can be randomly selected and does not crucially contribute to the final win or loss of the white player. The moves that were captures were removed as they are usually transient in nature and also do not always determine the winner. [8] While the DeepChess architecture uses the bitboard representation for its board states, we decided to use our 69-input feature vector to allow for faster computing.

We first created a type of Feed Forward Neural Network called a deep belief network (DBN). The layers of the DBN consist of stacked autoencoders that are trained layer by layer using unsupervised training. The purpose of an autoencoder is to reduce dimensionality and noise in the data and extract the most important features. [9] This is done by finding the weights that closely map the input layer to the output layer, making it as near to an identity function as possible. For our network, we chose to build five connected autoencoders of sizes: 64-64-64-60-40. Therefore, the first autoencoder layer is trained with the 69-input feature vector as the input layer and the output layer (69:64:69). Those weights are fixed and then the second autoencoder layer is trained with the first autoencoder layer as its input and output (64:64:64), and so on. The DBN was trained for 150 epochs with a batch size of 256 and a ReLU activation function.

Using this trained DBN, we then constructed a Siamese Network with supervised training (the DeepChess architecture). [10] This type of neural network allows the model to learn what positions result in a win or loss for the white player. Our Siamese network uses two disjoint copies of the DBN that are in turn jointly connected to four more hidden layers of sizes 60-40-20-2. During the training, the network will take in an input pair that consists of a move from a game in which the white player eventually wins and a move from a game in which the black player eventually wins. By using two disjoint copies of the DBN, we are able to increase the dataset from which the network can obtain a training pair. For example, if there are 1000 positions from games that are eventually won by the white player and 1000 positions from games that were eventually lost by the white player, we can obtain 2×10^6 training pairs, since each pair can be used twice as [win, loss] and [loss, win].

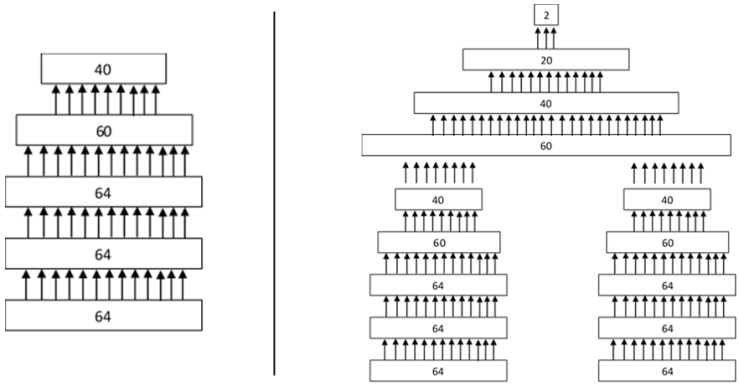


Figure 4: *(left)* Our Deep Belief Network *(right)* Our Siamese Network (modeled on the DeepChess architecture)

The four hidden layers on top of the two disjoint copies are used to compare the positions to determine the best one. The layers take an input from each of the DBNs with the shared weights from the prior training and finally output two values. The first three hidden layers use a ReLU activation function and the final output layer uses a softmax function; the network was trained for 550 epochs. We implemented a Python generator function [11] and passed it into the fit generator function of our model. This allowed us to generate a new data set for every epoch, reducing the possibility of overfitting the model. [8]

Testing

Before examining real-world performance, we verify the accuracy of the neural networks. The dense model has two outputs, but any single output can be calculated from the other output. Therefore, only one output needs to be examined to determine the classification accuracy of the dense network. We used 5-fold validation to see if the network is improving its accuracy at predicting a white win. Roughly half of the board states in the data set corresponds to a white win, so we expect a randomly initialized network to begin with an accuracy around 0.5. The figure below confirms this. Accuracy grew quickly after the first few epochs of training. It plateaued after a few dozen epochs and seems to converge to 0.57.

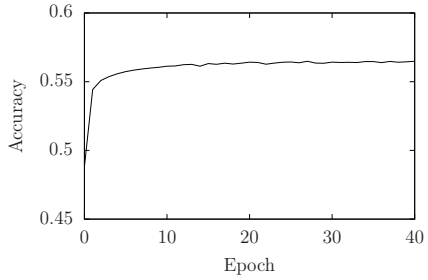


Figure 5: 5-fold validation classification accuracy during learning

Validation of the DeepChess model was not as straightforward. During training, we noticed the model's accuracy stayed relatively fixed at 50 percent. We turned to an alternate method.

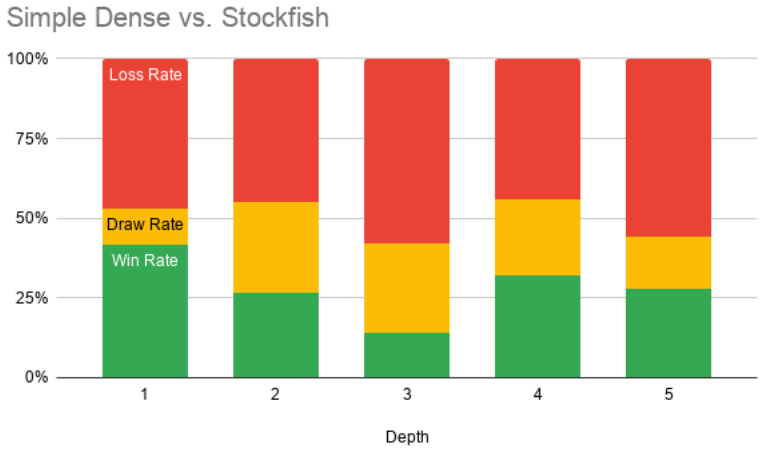
In order to further understand the proficiency of our two models based on dense neural networks and Siamese networks (modeled on the DeepChess architecture), we passed these models as an evaluation function for the alpha-beta pruning algorithm to determine the best move from the given board state. The alpha-beta pruning algorithm compares a new node's value with α and β and either replaces our current node (if it is larger than α) or prunes that branch (if it is larger than β). [12] Our version of the alpha-beta pruning algorithm, which was heavily influenced by the DeepChess architecture implementation, compares each position with an α_{pos} and β_{pos} that store positions. [8] Each new possible position is predicted and compared with DeepChess and if it is better than α_{pos}, it becomes the new α_{pos}, and if the position is better than β_{pos}, we prune the branch. Our alpha-beta pruning algorithm allows us to compare the models at deeper depths of decision making between positions.

To accurately assess both models, we tested each against two open source chess engines, StockFish [17] and Crafty [18]. These engines are highly complex and have been optimized over decades. Although both play remarkably, StockFish is commonly regarded as the best chess engine in the world. A common input to both engines is depth, which determines how many steps ahead the engine can look when making a decision, effectively allowing it to plan ahead. Both of our models were tested against each engine at depths of 1 through 5.

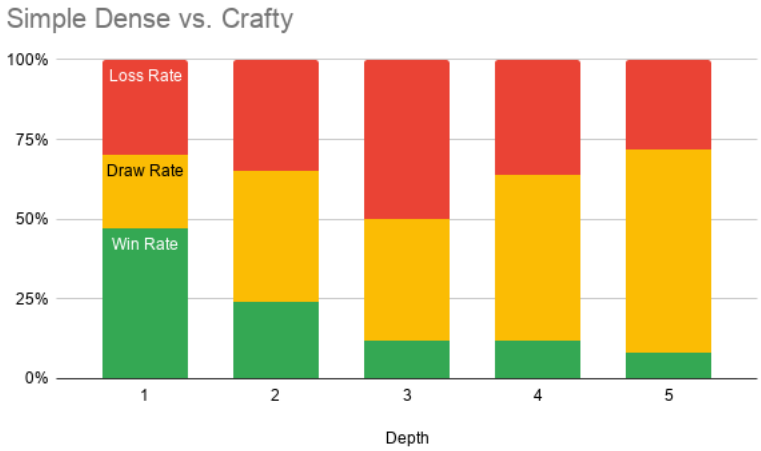
Results

SimpleDense and DeepChess against StockFish and Crafty

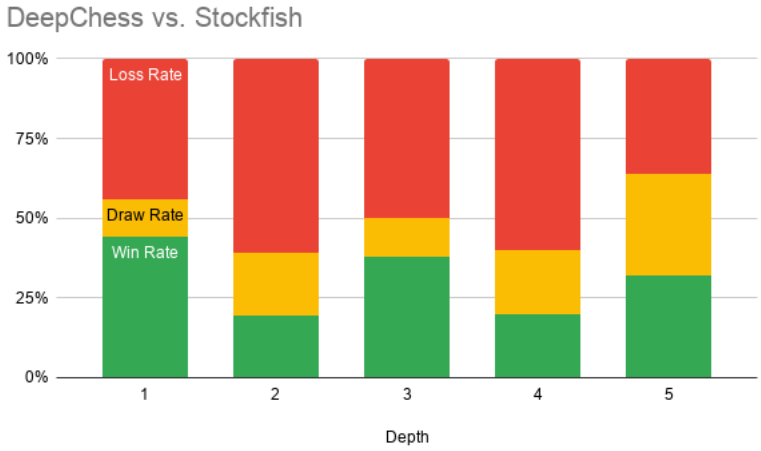
At its best (at a depth of 1), our dense model beat StockFish in 42% of the games. At a depth of 3, StockFish wins 58% of matches, which makes it the least ideal setup.



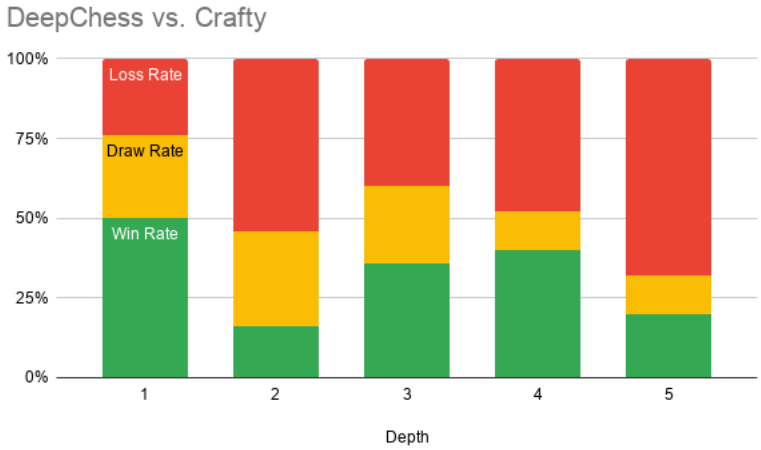
At a depth of 1, Crafty lost 47% of the games against our dense model. With increased depth, our win rate falls, but unlike the other match ups, our draw rate is steadily increasing with depth. At a depth of 5, we draw an astounding 64% of the games. The most likely explanation for this is the dense network is stalling causing a draw.



The DeepChess model performed slightly better against StockFish in comparison to the the dense model. The DeepChess model won 30% or more of the games played against StockFish at depths 1,3, and 5. For both depths 2 and 4, our win rate was 20%.



At a depth of 1, DeepChess won 50% of the games against Crafty, the highest win rate in among all our tests. Wins generally decrease with depth, with the sudden dip to 16% at depth = 2 being the outlier.



Our results show that we consistently win most frequently when the chess engine is set to a depth of 1. Generally, the number of games won decreased with increased depth, which was expected. Both models lost to StockFish more frequently than they did to Crafty, possibly confirming our hypothesis that StockFish is a superior engine. It is important to notice that while none of our models are beating the engines over 50% of the time, these are far superior algorithms, and our algorithms are able to hold up.

Comparing our two models, we see that our DeepChess model won more frequently than our dense model given the same opponent setup. On the other

hand, our dense model was better at ending a game with a draw. This meant that while we lost more games with the DeepChess model, we also won more games, both relative to the dense model. This indicates that our DeepChess model is superior at securing a win, while the dense model is superior at preventing the opponent from winning.

Discussion

Our work focused on training and comparing the performance of different neural networks which we designed to play chess using a dataset and a personal computer. From our results, we see that it was difficult to implement a chess player with any model that outperformed either the StockFish or Crafty chess engines in more than 50% of the games played.

Our testing of the SimpleDense architecture revealed that an increase in depth did not lead to an increased win rate, contrary to some of our initial assumptions. Instead, there was a steep drop off our win rate when increasing the depth from 1 to 2 and again from depth 2 to 3. Increasing the depth further had marginal effects on the win rate. One interesting observation is that despite the decrease in win rate, increased depth did lead to fewer losses for our model via a higher rate of draws. Regardless, this decreased win rate is likely a product of the chess engines being more optimized/efficient and therefore better at taking advantage of the increased depth.

While the DeepChess architecture actually performed better than our SimpleDense network, it was unable to consistently perform better than the chess engines. Some of our limitations include the size of the dataset used, and the lack of computational power at our disposal. More computational power would allow us to train the model with more epochs, possibly giving rise to a better performing model. We can try to improve the architecture by using hashing to store the best move for a certain board state that can be revisited without having to be calculated every time. The model can also be trained using different regularization techniques in addition to the random shuffling.

Conclusions

The purpose of our work and report was to use the familiar concept of chess to explore the potential of current computing architectures on consumer hardware. We certainly achieved an understanding of what it takes to actually create such networks, such as the necessity of deeper training given a complex problem.

Further research should be concentrated on further simplifying the entire process from gathering data to producing a working model. Our path was riddled with failure which ultimately cost us time. If the process was made more simple, more students such as ourselves will be able to efficiently construct neural networks. The libraries we used, such as Keras, already take the blunt of the work off your shoulders, but as we have seen, have some limitations. We

hope that our work will ultimately find use as a case study on the limitations of Keras and the complexity of representing chess efficiently to a neural network. Our work can help those interested in the field save days of training, given the initially intuitive models that we found to be ineffective.

References

[1] Keras. https://keras.io/

[2] TensorFlow. https://www.tensorflow.org/api_docs/python/tf

[3] Niklas Fiekas, python-chess 0.28.3. https://pypi.org/project/python-chess/

[4] Mitchell J, Chess Game Dataset. https://www.kaggle.com/datasnaek/chess

[5] Chess programming Wiki, Bitboards. https://www.chessprogramming.org/Bitboards

[6] PyChess. http://pychess.org/about/

[7] Several Authors, Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm. https://arxiv.org/pdf/1712.01815.pdf

[8] Several Authors, DeepChess: End-to-End Deep Neural Network for Automatic Learning in Chess. https://www.cs.tau.ac.il/~wolf/papers/deepchess.pdf

[9] Several Authors, Autoencoders. http://ufldl.stanford.edu/tutorial/unsupervised/Autoencoders/

[10] Branislav Hollander, Siamese Networks: Algorithm, Applications, and PyTorch Implementation https://becominghuman.ai/siamese-networks-algorithm-applications-and-pytorch-implementation-4ffa3304c18

[11] Yasoob Khalid, Generators. http://book.pythontips.com/en/latest/generators.html

[12] Rosen, CS 161 Reciation Notes - Minimax with Alpha Beta Pruning. http://web.cs.ucla.edu/~rosen/161/notes/alphabeta.html

[13] Matthew Lai, Giraffe: Using Deep Reinforcement Learning to Play Chess. https://arxiv.org/pdf/1509.01549.pdf

[14] Barak Oshri, Nishith Khandwala, Predicting Moves in Chess using Convolutional Neural Networks. https://pdfs.semanticscholar.org/28a9/fff7208256de548c273e96487d750137c31d.pdf

[15] Michele McPhee, K.C. Baker and Corky Siemaszko, Deep Blue, IBM's supercomputer, defeats chess champion Garry Kasparov in 1997. https://www.nydailynews.com/news/world/kasparov-deep-blues-losingchess-champ-rooke-article-1.762264

[16] William Saletan, Chess Bump. https://slate.com/technology/2007/05/the-triumphant-teamwork-of-humans-and-computers.html

[17] Daylen Yang, StockFish. https://stockfishchess.org/

[18] Robert M. Hyatt, Crafty. http://craftychess.com/

[19] Natasha Regan and Matthew Sadler, DeepMind's superhuman AI is rewriting how we play chess. https://www.wired.co.uk/article/deepmind-ai-chess

[20] David Silver, Thomas Hubert, Julian Schrittwieser, and Demis Hassabis. AlphaZero: Shedding new light on chess, shogi, and Go. https://deepmind.com/blog/article/alphazero-shedding-new-light-grand-games-chess-shogi-and-go

[21] Thibault Duplessis, Lichess. https://lichess.org/

[22] ChessAi Algorithms Team, Chess-AI Repository. https://github.com/ucdchessai/chess-ai

Author Contributions

Joel Lee: Project/Report Manager
Vivienne Chiang: Presentation Manager
Muhammed Halbutogullari: Report Manager
Nick Van: Data Processing, Algorithms
Pavit Bath: Data Processing, Testing, Report Contributor
Mohamed Shafi: Research, Algorithms, Testing
Lawrence Lee: Dense Networks, Report Contributor
Ezekiel Morton: Data Processing, Algorithms, Debugging
Zain Munad: Research, Presentation Manager
Alexander Recalde: Algorithms, Testing, Debugging
Nikhila Thota: Data Processing, Testing, Report Contributor