NANYANG TECHNOLOGICAL UNIVERSITY

DEEP LEARNING AND COMPUTER CHESS

Low Yu Benedict

School of Computer Science and Engineering

2021

NANYANG TECHNOLOGICAL UNIVERSITY

SCSE20-0672

DEEP LEARNING AND COMPUTER CHESS

Submitted in Partial Fulfillment of the Requirements for the Degree of Bachelor of Computer Science of the Nanyang Technological University

by

Low Yu Benedict

School of Computer Science and Engineering

2021

Abstract

This report presents a chess evaluation function trained using neural networks, without a priori knowledge of chess. The neural network undergoes two phases. In the first phase, it is trained using unsupervised learning to perform feature extraction. Subsequently in the second phase it undergoes supervised learning to compare two chess positions and select the more favourable one. The entire network is trained using only positions of a chess game and the game's outcome, and no other information on chess. Although the neural network utilizes a relatively shallow network architecture by modern standards, it is capable of achieving very high accuracies and has shown great promise in its ability to identify key features that results in a favourable game. This project closely follows the implementation and concepts of DeepChess [1].

Acknowledgements

I would like to thank Dr He Ying for providing extensive resources, and his patience and guidance through the course of the project.

Table of Contents

List of Figures and Illustrations

1. Introduction

Chess engines trained through neural networks such as Leela Chess Zero and AlphaZero have changed how players understand chess - conventionally unorthodox moves that were once considered anti-positional are now widely adopted in today's competitive chess landscape. The superiority of neural networks over rule-based engines was demonstrated in 2019 where a 1,000 game match between the rule-based engine Stockfish 8 and the neural network trained engine AlphaZero resulted in AlphaZero winning 155, drawing 839 and only losing 6 games to Stockfish 8.

For this project, we aim to create a neural network based chess evaluation function by emulating the methods used in DeepChess [1]. We create a static chess evaluation neural network that extracts features through unsupervised learning, and uses these features to determine the better position when presented with any pair of chess positions. The ability of our chess evaluation function to juxtapose between two positions can eventually be used in game trees to determine optimal paths to take.

Moving forward, the chess evaluation neural network created in this project will be interchangeably termed as the project or DeepChess.

1.1. Background

The game of chess can be translated into a game tree such that at every position other than checkmate, there are some number of legal moves that result in a new position.

For simple games like Tic-Tac-Toe with a branching factor of 4, an engine could easily parse through the entire game tree to find the best positions to play that can result in a win. However, as chess is an incredibly complex game with a high branching factor of about 35, and an average game can be expected to have a game tree of 10^{120} possible branches [2], it is computationally infeasible by modern standards to construct and search the entire game tree for the best positions during play. As such, chess engines typically employ different strategies to search only parts of a game tree before coming up with the best play.

In general, regardless of the tree search strategy employed, chess engines use evaluation functions to simulate the expected utility of a node without searching ahead for its child nodes down the tree.

1.2 Early Chess Evaluation Methods

While human players often rely on heuristics (chess principles, identifying chess patterns etc.) when playing, the study of chess theory inevitably led to the search for a method capable of evaluating chess positions to gain a better understanding of chess.

Before the popularity of computer chess, chess positions were often evaluated based on the fundamental concept that different chess pieces yield different levels of potential, and the positions of each chess piece determines the usefulness of it. For example, a queen has significantly higher mobility (i.e. controls more squares on a chess board) than a knight and thus is valued more. Similarly, a knight placed on the edge of the board is usually considered worse than a knight in the center of the board as the former controls at most half of the squares compared to the latter.

With this fundamental concept, piece-square tables were traditionally used in the early 90s. These tables were encoded such that each piece had a corresponding value depending on which of the 64 possible grids it is placed.

Ultimately, the premise for a good chess evaluation function hinges on accurately capturing and factoring for as much relevant information in any given chess position. The relevant information used to evaluate any chess position is called *features*. In the case of piece-square tables, each piece (king, queen, knight, bishop, rook, pawn) has a corresponding table and each player has their own table (i.e. black vs white). As such, piece-square tables can be said to have a total of 768 features, and functions as a linear evaluation function.

1.3. Conventional Chess Engines

The 10th of February 1996 marked an important moment in computer chess history when Deep Blue defeated then-reigning world champion Garry Kasparov. Deep Blue can be considered to be an engine based on the then popular idea of leveraging the powerful computational capabilities of computers with expert domain knowledge. In total, Deep Blue used 6,400 features [3] for its evaluation function.

However, as Garry Kasparov later remarked in his book *Deep Thinking*, the approach Deep Blue took can be considered brute force [4] as it focused more on heuristics rather than understanding the game of chess to play it.

Over the next two decades, incredibly strong chess engines such as *Stockfish, Komodo* and *Houdini* continued to take a similar approach as Deep Blue - by evaluating positions using hand-crafted features (such as king safety, doubled pawns, piece centrality etc.) determined by human grandmasters and carefully tuning these weights [5].

1.4. Neural Network Based Chess Engines

The dominance of rule-based game engines was shattered when in 2016 neural network based engine *AlphaGo Zero* defeated 18-time world champion Lee Sedol in the game of Go. *AlphaGo Zero* showed huge promise in the direction of replacing handcrafted features with features learned via deep learning.

Deep learning enables game engines to develop an understanding of the game beyond the constraints of human imaginations and understanding of chess. Where Deep Blue had a total of 6,400 features, our model was trained to extract up to 1,403,402 relevant features (although it is not correct to say that the model definitely uses all 1,403,402 parameters extensively) in total.

2. Implementation

DeepChess is a siamese network constructed with two identical Deep Belief Networks (DBN) connected to an Artificial Neural Network (ANN). The DBN is constructed based on stacked autoencoders. As the end-to-end implementation of DeepChess was not made clear in its paper, this section covers how we implemented DeepChess based on our judgment.

Full implementation source code can be found at this repository https://github.com/Bot-Benedict/DeepChess. Comments on the technical approach as well as code snippets used in the implementation are included in the Appendix.

2.1. Dataset

As mentioned, our project aims to create an evaluation function from scratch with no a priori knowledge of chess. As such, we solely base our dataset on historical chess games recorded on the Computer Chess Ratings List, CCRL (www.computerchess.org.uk/ccrl). CCRL is a repository containing games played between computer chess engines.

At the point of our development, CCRL contained a total of 1,288,701 games with 438,774 games where White won, 321,010 games where Black won, and 528,917 games where both drawed. It was previously deemed that the inclusion of games that ended in a draw were not beneficial to the training of DeepChess [1]. As such, we use a total of 759,784 games as our dataset.

We extracted ten positions from each game that were neither capture moves nor positions in the first turns of the game. Capture moves were omitted as they can be misleading since these moves tend to be answered with a trade of pieces, and as such often only provides a transient advantage. The first five moves were not used because the games recorded in CCRL permits the use of opening books and these oftentimes can imply that the first few moves are relatively consistent and safe. These heavily studied chess openings as such do not usually yield much impactful information.

With these constraints in place, we extracted a total of 7,597,840 positions of which 4,387,740 were positions where White won and 3,210,100 were positions where Black won. The table below summarizes the data extracted as well as draws comparison to the dataset used by the original DeepChess implementation.

Table 1: Training Datasets for DeepChess and Current Project

	Original DeepChess	Current Project
White Wins Positions	2,216,950	4,387,740
Black Wins Positions	1,643,870	3,210,100
Total Positions	2,216,950	7,597,840

Finally, each position was converted to bitboard representation of size 773. A bitboard is a 773 bit string that encodes each chess piece's position and other miscellaneous information.

As there are two sides (White and Black), six piece types (pawn, knight, bishop, rook, queen and king) as well as an 8-by-8 board (total 64 grids) we get $2 \times 6 \times 64 = 768$ bits. An additional 5 bits is used to encode the playing side (1 bit, 1 for White and 0 for Black) and castling rights (4 bits, both kingside and queenside) of both players.

2.2. Autoencoders

Autoencoders (AE) are the foundational building blocks for the DBN in our model. The general purpose of autoencoders is to learn in an unsupervised manner to reconstruct its input [6].

There are three components to any autoencoder, the encoder, the code and the decoder. The encoder functions by compressing the input into smaller dimensions until eventually it reaches the code which represents the fully compressed information. Information from the code is then passed on to the decoder which then reconstructs the compressed information to the same shape as the input. Ultimately, our goal is for an output that is identical to the input. Typically, the encoder and decoder are mirror images of each other.

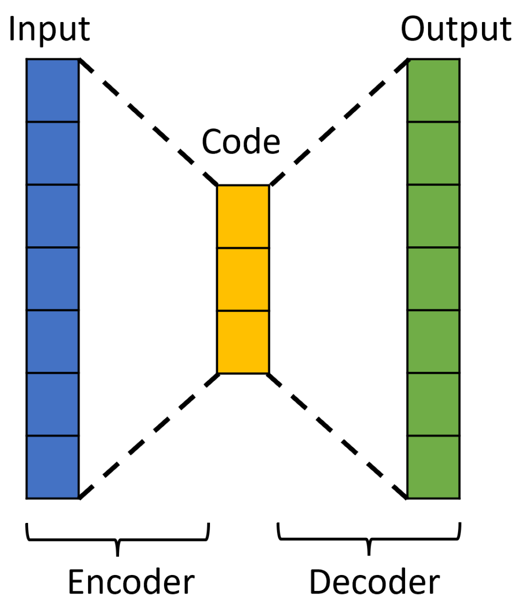

Figure 1: Autoencoder Architecture

For our DBN, we used a total of four different autoencoders with the following parameters.

Table 2: Stacked Autoencoder Hyperparameters

	AE-1	AE-2	AE-3	AE-3
Layer 1	773	600	400	200

(Input/Encoder)				
Layer 2 (Code)	600	400	200	100
Layer 3 (Output/Decoder)	773	600	400	200

2.3. DBN Architecture

As mentioned earlier, our DBN consists of four autoencoders stacked. The DBN is trained in a fashion similar to greedy layer wise training - that is, the first autoencoder (i.e. AE-1, 773-600-773) is trained on its own before freezing the weights and training the next autoencoder (AE-2, 600-400-600) and so forth.

As the actual implementation of the DBN (which was previously termed *Pos2Vec*) was vague in the DeepChess paper, we decided to go with the approach of

1. training an autoencoder,
2. freezing the weights of existing layers,
3. restructuring the existing DBN architecture to include a new encoder, coder and decoder (i.e. the next AE) and then
4. training the next autoencoder until all autoencoders have been trained.

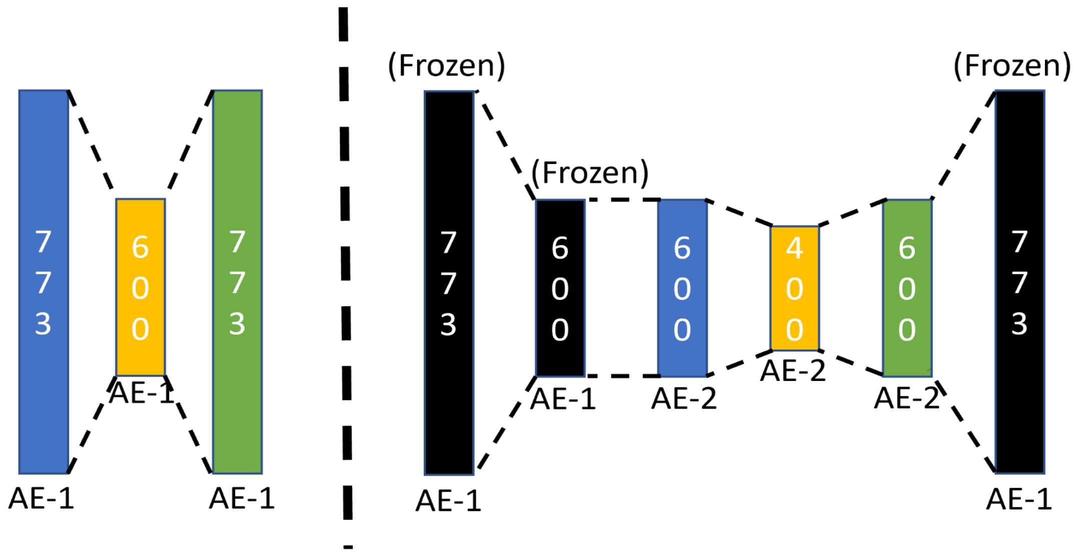

Figure 2: AE1 Implementation

Figure 3: AE1 and AE2 Implementation

As shown in the figures above, we insert the next autoencoder, between the code and decoder of the previous autoencoder. This restructuring was done for each AE-X trained.

The restructuring was necessary as the stacked autoencoders had to be fed an input of 773 (i.e. the size of a bitboard) and an output of the same size in order to compute the difference using the loss function.

For the DBN, a total of 2,000,000 positions were sampled randomly whereby 1,000,000 positions contributed to White winning and 1,000,000 positions from Black winning. The DBN uses rectified linear units (ReLU) as its activation function and the Adam optimizer with a learning rate of 0.005 that is multiplied by 0.98 at the end of each epoch. We experimented with different loss functions and eventually settled on binary cross entropy as it best fit the goal of our DBN - to decode, and in the process classify, each bit of a bitboard to be 0 or 1. The entire stacked autoencoder was trained for 200 epochs.

After training the stacked autoencoders, we extracted the code layer from each autoencoder and stacked them. This model thus constitutes our DBN.

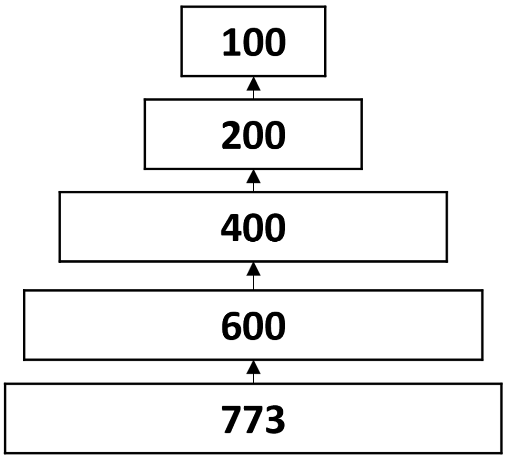

Figure 4: DBN Architecture

2.4. DeepChess Architecture

As mentioned, DeepChess is a siamese network. As such, we stacked two disjoint copies of the DBN side by side and then added four fully connected layers of sizes 400, 200, 100 and 2 on top of it. The DBNs thus serve as high level feature extractors while the four layers below are used to evaluate the more favourable position of any given pair of positions.

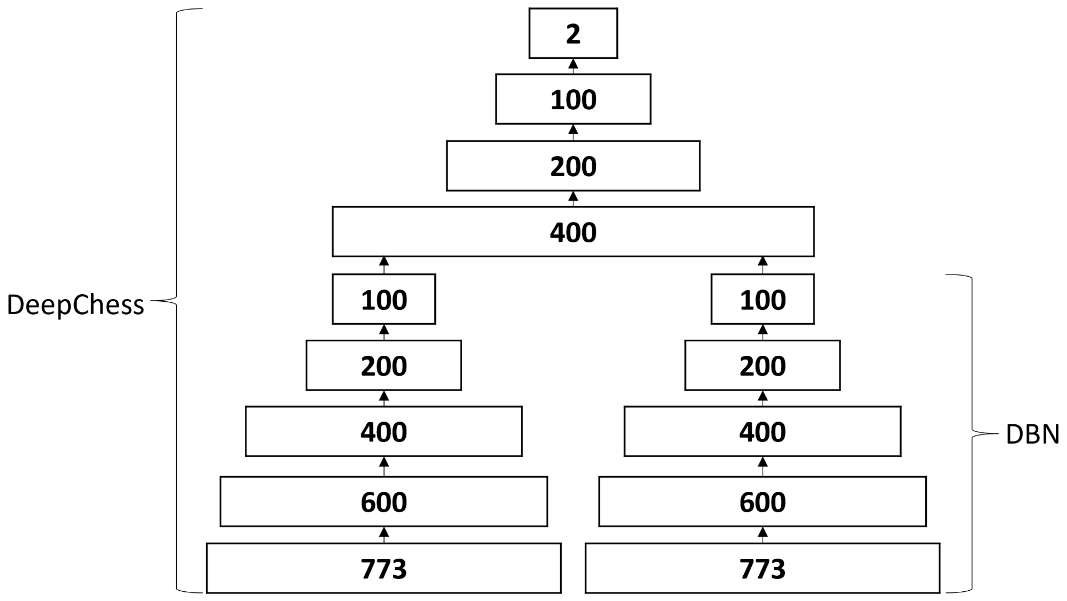

Figure 5: DeepChess Architecture

While there were initial attempts to train the model on the entire 7,597,840 positions - and other varying numbers of subsets of the entire dataset - the sheer amount of memory needed made it infeasible even with the use of generators, compression data types and other strategies.

Ultimately, we decided to reduce our training data down to the original DeepChess's implementation of using 2,116,950 White wins and 1,543,870 White losses (conversely also known as Black wins) as our input instead. This proved workable, albeit precariously so as our RAM tended close to memory exhaustion during each epoch.

The entire DeepChess model was trained over 1,000 epochs with 1,000,000 pairs of training input (1,000,000 White wins and 1,000,000 White losses) and 100,000 pairs of validation input (100,000 White wins and 100,000 White losses) with a training batch size of 50,000 and a validation batch size of 25,000. The DBNs weights were tied, and the weights of the entire DeepChess model were trained.

The pairs of training and validation inputs were updated on every epoch. As such, at the start of each epoch, we randomly sample 1,000,000 pairs of inputs and randomly arrange these inputs into (White Wins, White Loss) and (White Loss, White Win). This method of data augmentation significantly improved the robustness of our dataset as the random ordering and sampling of games amounts to 6.5×10^{12}potential pairs of positions in each epoch, thus making virtually all training samples to be considered new training data.

It was also with this understanding that we felt that reducing the training sample size from over 7 million down to just over 3 million would be reasonable as the data can be sufficiently augmented to prevent overfitting.

We used the ReLU activation function in all layers except for the output layer where we used softmax. The Adam optimizer was also used with a learning rate of 0.01 that is multiplied by 0.99 at the end of each epoch. Finally, as DeepChess can be deemed to be a binary classifier of favourable positions, the binary cross entropy loss was used.

Unsurprisingly, as a result of both training and validation datasets being augmented on each epoch both validation and training accuracies followed similar trajectories as shown in the figure below.

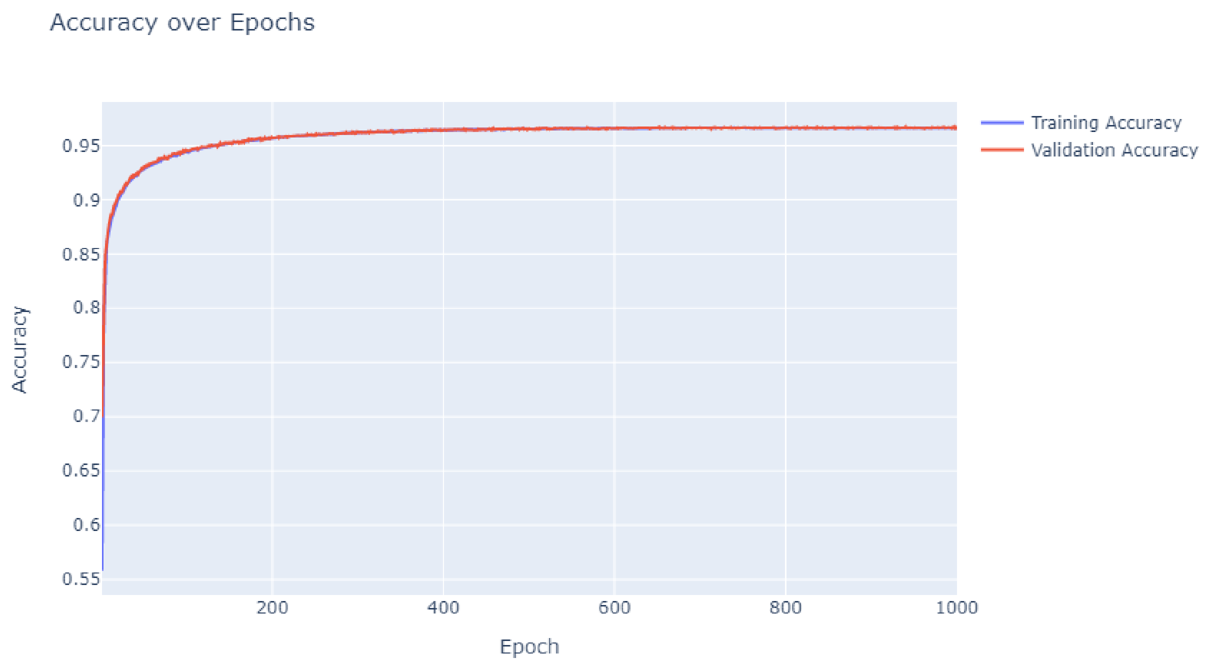

Figure 6: DeepChess Training Accuracy

Furthermore, the loss values continued to decrease over the span of 1,000 epochs, proving the hypothesis that the data augmentation used was capable of preventing overfitting.

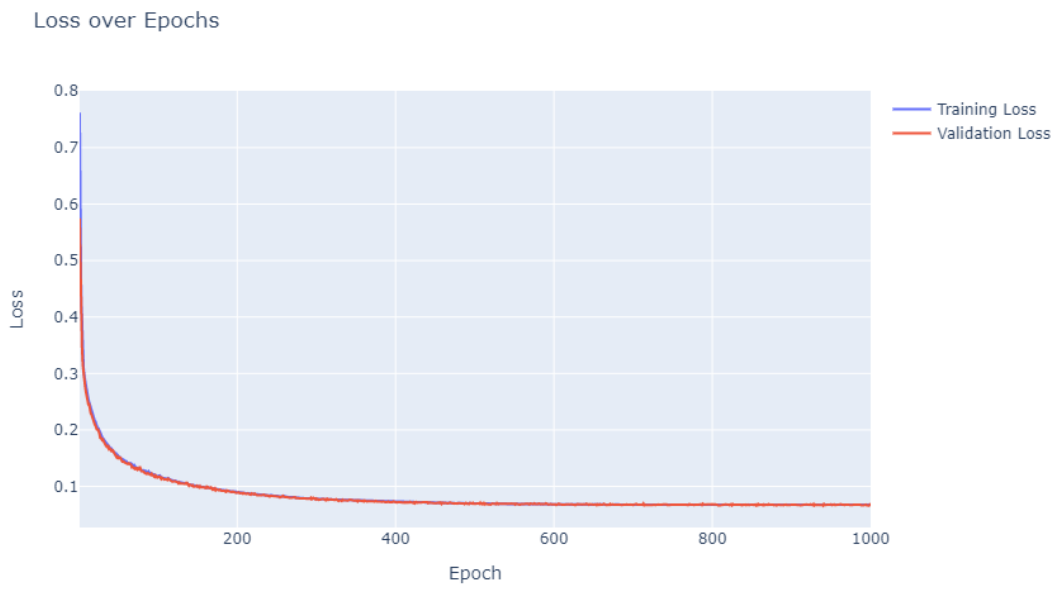

Figure 7: DeepChess Training Loss

The training and validation accuracy both peaked at 96.68% and 96.74% respectively. Considering the model was never explicitly taught any knowledge of chess (piece values, rules of chess etc.) it has proven that it is adept at extracting the most important features and identifying which features contribute to a winning position.

3 Experiments

3.1. Experimental Overview

In total, we performed two experiments to test the effectiveness of DeepChess. The first experiment tests the capability of the static evaluation function, while the second tests the evaluation function's ability to solve tricky positions that would generally require some form of tree search. As this report does not cover the implementation of a tree search, we implemented some form of work around for Experiment 2.

In the first experiment, we evaluate the effectiveness of DeepChess by comparing it against the static evaluation function employed by Stockfish. We used a dataset containing 88,398 positions extracted from non-drawn games, with positions only extracted after the first five turns of a game and positions not involving a capture move. The dataset was then used for evaluating four different scenarios:

- Scenario A: A pair of even positions (x_{left}, x_{right}) for both Black and White as input.
- Scenario B: A pair of positions where White is in an advantageous position (x_{left}) and Black is in an advantageous position (x_{right}) as input.
- Scenario C: A pair of positions where White is in an advantageous position (x_{left}) and Black is in an even position (x_{right}) as input.
- Scenario D: A pair of positions where White is in an even position (x_{left}) and Black is in an advantageous position (x_{right}) as input.

In all four scenarios, x_{left} resulted in White winning and x_{right} resulted in Black winning. There was no need to perform random swaps of x_{left} and x_{right} like we did when training DeepChess as the model is being run in inference mode and will not have its weights updated.

It is worth highlighting that our dataset only used Stockfish's static evaluation function. In other words, there was no tree search being done and the position was only evaluated based on fixed features programmed into Stockfish, with no foresight into the position itself. With this in mind, we thus did not look to compare DeepChess and Stockfish based on whether they predicted similar evaluation outputs (i.e. comparing to see if given two positions that resulted in wins for White, can DeepChess identify the one deemed by Stockfish to be better). This is because DeepChess's and Stockfish's approach to an evaluation function is fundamentally different. Stockfish also uses a linear evaluation function, while DeepChess uses a non-linear evaluation function. More importantly, DeepChess was trained to classify whether a position might eventually lead to a loss or win and thus could pick up features that are indicative of a winning position even when down on material. This behaviour was noted by the creators of DeepChess [1] as well.

In the second experiment, we used the Strategic Test Suite (STS) [7]. The STS comprises 15 themes with 100 positions for each theme amounting to a total of 1,500 positions. Each theme entails a different problem such as center control, activity of king, simplification and knight outposts; and each position has a single best move with a score of 10, and every other legal moves are given scores ranging from 0 to 9. This translates to a maximum attainable score of 15,000.

Ultimately, the STS is designed to evaluate a chess engine's long term understanding of strategic and positional concepts. The STS has also been considered a useful tool for gauging a chess engine's ELO rating.

3.2. Experiment 1 - Threshold Values for Classifying Positions

The threshold values for even and winning (or losing) positions were decided based on how Stockfish's evaluation function works. Stockfish values pawns with a score of 1, minor pieces (i.e. Bishops and Knights) with a score ranging between 4 and 6, and rooks with a value hovering around 6 to 10 (https://github.com/official-stockfish/Stockfish/blob/master/src/types.h#L194-L198). Stockfish also uses centipawns - such that the positional equivalent of a single pawn amounts to 100 centipawns. White often starts with a rating of 20 centipawn (0.02) in favour as it has the first move of any game [8].
It is worth mentioning that Stockfish uses many hand-craft features beyond piece relative values such as King's safety and doubled pawns for its evaluation function. However, using the piece relative values can be considered sufficient as we only need to understand the scale that Stockfish uses for evaluating a position in order to determine a reasonable threshold value for analyzing the positions [8].

With these values in mind, we used the threshold value of 5 (positive for White inclination and negative for Black inclination) to represent winning positions as the value 5 lies between the score of a minor piece and a rook. It is safe to say that playing without material equal to a minor piece, a rook or five pawns can be considered a disadvantageous position. Naturally, on the flip side playing with additional material amounting to 5 points can be considered an advantageous position.

We also used a threshold value of 0.05 (i.e. 5 centipawns) to represent even positions. In other words, positions are determined to be relatively even when its evaluation score falls between the range of -0.05 to +0.05.

3.3. Experiment 1 - Data Augmentation

As the dataset contained only 88,398 positions, we had to perform data augmentation in order to get a good grasp of how DeepChess was performing.

Data augmentation was done by matching each x^i_{left} to every other x^k_{right} thus creating at most $count(x_{left}) \times count(x_{right})$pairs of input. We introduced an augmentation factor ranging from 0 to 1, where 0 implies no augmentation and 1 implies using the maximum number attainable from our data augmentation. The augmentation factor was thus used to regulate the size of datasets to achieve a general size of 100,000 for each experimented scenario.

3.4. Experiment 2 - Strategic Test Suite

Typically, the STS is run alongside a complete chess engine that performs tree searching before deciding on the next best position. However, as this report does not entail the construction of a game tree, we decided to do the following:

1. Perform a tree search of depth 1, for every given position in the STS.
2. Compare positions in the solutions of the STS and allow for DeepChess to select the best position/move.

The goal of this experiment is to gain some form of cursory understanding on the performance of DeepChess when applied in a tree search setting.

3.5. Results

DeepChess performed exceedingly well in the first experiment and only fell short in Scenario A when comparing two very even positions. This was not unexpected as most chess engines rely on game tree searching, beyond static chess evaluation functions to determine the better move when comparing positions.

As a recap, these are the four scenarios we experimented with:

- Scenario A: A pair of even positions (x_{left}, x_{right}) for both Black and White as input.
- Scenario B: A pair of positions where White is in an advantageous position (x_{left}) and Black is in an advantageous position (x_{right}) as input.
- Scenario C: A pair of positions where White is in an advantageous position (x_{left}) and Black is in an even position (x_{right}) as input.
- Scenario D: A pair of positions where White is in an even position (x_{left}) and Black is in an advantageous position (x_{right}) as input.

In all four scenarios, x_{left} resulted in a win for White and x_{right} resulted in a loss for White. The table below summarizes our findings.

Table 3: Experiment 1 Results

	Size of x_{left}	Size of x_{right}	Size of Augmented Data	Accuracy
Scenario A	1,813	2,016	108,780	69.37%
Scenario B	278	156	43,368	95.85%
Scenario C	278	2,016	112,034	95.01%
Scenario D	1813	156	112,406	90.89%

With the results shown, it is clear that DeepChess possesses not only the understanding of each chess piece's relative worth, but also the foresight that positional advantages can supersede a temporary material advantage. This was made apparent in Scenario A where DeepChess accurately identified close to 70% of positions to be more favourable for a winning chance for White, despite both positions having relatively neutral static evaluation scores. The results ultimately show that DeepChess is capable of functioning as an evaluation function with an extremely high accuracy.

On the other hand, DeepChess expectedly did not perform well on the STS due to the lack of a tree search implemented. When we applied DeepChess on the STS with a tree search depth of 1, we got a score of 1,160 out of 15,000 - which can be translated to only a 7.73%.

However, given that Experiment 1 showed that DeepChess can be extremely effective as a standalone static evaluation function, we next decided to try running DeepChess against the STS for all positions deemed as solutions for any given position to solve.

For the purpose of this experiment, we omitted any position with only a single solution. We performed this experiment under the assumption that DeepChess in most cases should perform well enough during tree searching to find the several positions provided as solutions in STS, and as such we would like to see if it is able to distinguish between positions such that while these positions are all beneficial, they are beneficial to varying degrees.

In this given experiment setup, we achieved a score of 8,546 out of a maximum score of 13,810 (i.e. 1,381 positions with more than one solution provided by STS). Normalizing this result out of 100 we get 61.88%. Comparing this result to Giraffe, a similar three-layer deep neural network chess engine with a rating of 2,400 [9], as well as other popular chess engines, we got the following results.

Table 4: Experiment 2 Results

Engine	Approx. Rating	Average Nodes Searched	STS Score	Normalized Score / 100
DeepChess (No Tree Search)	-	38	1160	7.73
DeepChess (Selecting Solution Positions)	-	3	8546 (out of 13,810)	61.88
Giraffe (1s)	2400	258570	9641	64.27
Giraffe (0.5s)	2400	119843	9211	61.41
Giraffe (0.1s)	2400	24134	8526	56.84
Stockfish 5	3387	108540	10505	70.00
Senpai 1.0	3096	86711	9414	62.76
Texel 1.04	2995	119455	8494	56.63
Arasan 17.5	2847	79442	7961	53.07
Scorpio 2.7.6	2821	139143	8795	58.63
Crafty 24.0	2801	296918	8541	56.94
GNU Chess 6 / Fruit 2.1	2685	58552	8307	55.38
Sungorus 1.4	2309	145069	7729	51.53

In a nutshell, Experiment 2 was designed to look ahead into the development of DeepChess as a complete end-to-end chess neural network engine despite only having the static evaluation function completed. While Experiment 2 can be slightly speculative in nature, using the findings gleaned from Experiment 1 in conjunction with Experiment 2 paints a picture of a chess engine with good promise of performing amongst the state of the art chess engines.

4. Recommendations

Going forward, we might want to further investigate the dataset used as computer chess engines can often end up playing a long series of redundant moves as resignation is not enabled in the CCRL dataset. This long-winded behaviour was repeatedly exhibited in the recent 2021 chess games of Leela Chess Zero and Stockfish. Positions that are a result of, or creates unuseful moves should not be included in the dataset.

Secondly, the author of Giraffe proposed a particularly intriguing method of representing chess boards using a list of pieces and their coordinates instead of the widely used bitboard format. The author pushes a compelling argument that bitboards fails to capture the state of a chess board accurately as the distance in a bitboard's feature space can be misleading such that three different positions with stark difference in implications (i.e. defensively placed bishops, overextended bishops and centralized bishops) can have similar distance to each other in the feature space. Using coordinates to encode a position can thus serve as a clear method of distinguishing between positions that are inherently serving different purposes [7].

Finally, with an evaluation function established we can next look to creating a tree searching algorithm to complete an end-to-end chess engine. DeepChess in its current iteration takes an average of 30 seconds per 100,000 positions evaluated. A possible development for a tree searching algorithm could be the Monte-Carlo Tree Search with early playout termination (MCTS-PT) [10]. MCTS-EPT has shown great promise with games with high branching factors like Amazons - which has an average branching factor of 374 or 299 (if you're the second player) [11], compared to Chess which has an average branching factor of 35 - and outperformed all mini-max based programs it played against. Considering that mini-max based tree search is one of the most popular methods for chess engines today, we believe that the MCTS-EPT proven superiority over mini-max based engines in Amazons makes it a game tree search algorithm worth exploring for the domain of chess.

5. Conclusion

This report presents the end-to-end construction of a neural network based chess evaluation function that is capable of identifying the best positions with the most promising continuation for a win. DeepChess has also shown the ability to look beyond immediate material advantage and instead understands the prospect of positional advantage in the long term. While computer chess engines have generally been rule-based, DeepChess has shown the promising prospect of using deep learning with minimal domain knowledge to create powerful chess evaluation functions.

References

[1] David, O. E., Netanyahu, N. S. and Wolf, L., “DeepChess: End-to-End Deep Neural Network for Automatic Learning in Chess,” 2016. [Online]. Available: arXiv:1711.09667

[2] C. Shannon, "XXII. Programming a computer for playing chess", The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science, vol. 41, no. 314, pp. 256-275, 1950. Available: https://doi.org/10.1080/14786445008521796.

[3] M. Campbell, A. J. Hoane, and F.-hsiung Hsu, “Deep Blue,” Artificial Intelligence, 09-Aug-2001. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0004370201001291.

[4] D. Hassabis, “Artificial Intelligence: Chess Match of the century,” Nature News, 27-Apr-2017. [Online]. Available: https://www.nature.com/articles/544413a.

[5] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan, and D. Hassabis, “Mastering chess and shogi by self-play with a general reinforcement learning algorithm,” arXiv.org, 05-Dec-2017. [Online]. Available: https://arxiv.org/pdf/1712.01815.pdf

[6] D. Bank, N. Koenigstein, and R. Giryes, “Autoencoders,” arXiv.org, 03-Apr-2021. [Online]. Available: https://arxiv.org/abs/2003.05991.

[7] Dann Corbit and Swaminathan Natarajan. Strategic test suite. https://sites.google.com/site/strategictestsuite/about-1, 2010.

[8] R. McIlroy-Young, S. Sen, J. Kleinberg, and A. Anderson, “Aligning Superhuman AI with Human Behavior: Chess as a Model System,” Aligning Superhuman AI with Human Behavior | Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, 23-Aug-2020. [Online]. Available: https://dl.acm.org/doi/pdf/10.1145/3394486.3403219.

[9] M. Lai, "Giraffe: Using Deep Reinforcement Learning to Play Chess", arXiv.org, 2015. [Online]. Available: https://arxiv.org/abs/1509.01549.

[10] R. Lorentz, “Using evaluation functions in Monte-Carlo Tree Search,” Theoretical Computer Science, 27-Jun-2016. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0304397516302717.

[11] J. Kloetzer, H. Iida, and B. Bouzy, “The Monte-Carlo Approach in Amazons,” *CiteSeerX*. [Online]. Available: https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.79.7640.

Appendix

The implementation of DeepChess can be decomposed into three steps:

1. Data Acquisition and Cleaning
2. Training the DBN
3. Training DeepChess

Perhaps the biggest obstacle for anybody interested in implementing DeepChess is the required hardware capability. My current setup uses the following hardware specifications:

- Intel i7-7700HQ CPU
- GeForce GTX 1060 6GB
- 2400 MHz 16 GB RAM

Finally, the following libraries were used:

- TensorFlow-GPU 2.6.0
- numpy
- pandas
- python-chess

1. Data Acquisition and Cleaning

Data Acquisition in itself is not a challenge as all of it can be scraped from CCRL (http://www.computerchess.org.uk/ccrl/4040/games.html). However, the cleaning can be problematic as one would have to analyze over a million games, convert each from chess notation to bitboard and then extract positions in accordance to what DeepChess was trained using. The python-chess library is particularly helpful at this stage for parsing pgn files and reading games, although this process took several days to analyze and required a total of 13 hours to fully parse through over 800,000 games.

2. Training the DBN

Using the aforementioned hardware setup, training the DBN over 200 epochs took just over two hours. As with any deep learning task, it is highly necessary to use the correct loss function. In an earlier iteration of

DeepChess, the stochastic gradient descent (SGD) optimizer was used alongside mean squared error (MSE) as its loss function. Generally speaking, MSE is used for regression and not classification tasks. As such, the DeepChess model trained performed extremely poorly with an accuracy of only 56% at the end of 1,000 epochs.

3.Training DeepChess

Training DeepChess is likely the hardest of all three tasks as it may require extensive experimentation and optimization due to it being highly resource intensive.

The following optimization methods were employed in order to train DeepChess:

1. Callbacks (TensorFlow): Callbacks are customizable methods that enable one to hook into various stages of the model training and inference lifecycle.
    a. Saving Weights: We regularly saved the weights of a trained model. In the event of memory resource exhaustion terminating the training process, we can always resume it by loading the trained weights and continuing the training process.

```
periodic_save =
tf.keras.callbacks.ModelCheckpoint(filepath='../models/DeepC
hess/DeepChess', verbose=0, save_best_only=True,
monitor='val_acc', mode='max', save_weights_only=False)
```

    b. Clearing Memory: At the end of each epoch, we cleared all memory using keras backend and python’s garbage collector modules. This was crucial as we were facing an old but notoriously known TensorFlow bug of memory leaks resulting in overflow when using data generators.

```
class ClearMemory(tf.keras.callbacks.Callback):
    def on_epoch_end(self, epoch, logs=None):
        gc.collect()
        k.clear_session()
```

2. Using Data Generators: For a project involving such large datasets, loading entire datasets into RAM (or GPU RAM) for training can be infeasible. As such, we used a data generator to incrementally feed new data in at each step during an epoch.

```
class DataGenerator(tf.keras.utils.Sequence):
    def __init__(
        self,
        x_left,
        x_right,
        y,
        batch_size=50000,
        num_classes=None,
    ):
        self.x_left = x_left
        self.x_right = x_right
        self.y = y
        self.batch_size = batch_size
        self.num_classes = num_classes

    def __len__(self):
        # Denotes the number of batches per epoch
        return len(self.y) // self.batch_size

    def __getitem__(self, index):
        return
(self.x_left[index*self.batch_size:(index+1)*self.batch_size
],
self.x_right[index*self.batch_size:(index+1)*self.batch_size
]), self.y[index*self.batch_size:(index+1)*self.batch_size]
```

Glossary

Term	Definition
DBN	Deep Belief Network
ANN	Artificial Neural Network
AE	Autoencoder
CCRL	Computer Chess Ratings List, a repository of computer chess games.