

Neural Networks for Chess State Evaluation

Joe Sheehan
Princeton University
Princeton, NJ
jps8@princeton.edu

Abstract

Attempts to mimic an existing chess evaluation function with multilayer fully-linked neural networks. Most chess engines consist of a search function and a state evaluation function that work in tandem to select moves. In recent years, search has seen much innovation, while evaluation has remained stagnant. I propose the novel approach of training neural networks to reproduce current state-of-the-art evaluation functions, with the hope that this could serve as the basis for future neural network drive state evaluation that outperforms traditional approaches.

1. Introduction

Chess is a board game which traces its origins to the early 13th century, but today, more chess is played on-line than on an actual board. The first chess-playing machine was developed in 1912, and could only play king-rook endgames, but mechanizing the process of playing chess has fascinated human imagination for centuries prior to that. Chess is governed by logical rules there is no element of luck involved so it seems a natural problem for which computers would be well suited to solve. In 1769, Wolfgang von Kempelen built an "Automaton Chess-Player," which was not actually an automaton but a machine with a human hidden inside of it. A similar hoax occurred in 1868. Even very early computers were turned towards the problem of playing chess. In 1948, Norman Wiener's book *Cybernetics* explained how chess could be played with a depth-limited depth-first-search and an evaluation function. Current state-of-the-art chess engines take a similar approach. By 1957, the first programs that could play a full game of chess were developed both in the United States and the USSR, and in 1962, the first program to play *credibly*, Kotok-McCarthy, was published at MIT. [1]

1.1. Deep Blue

However, it took a surprising amount of time to go from a credible to a winning chess-playing computer. Deep Blue,

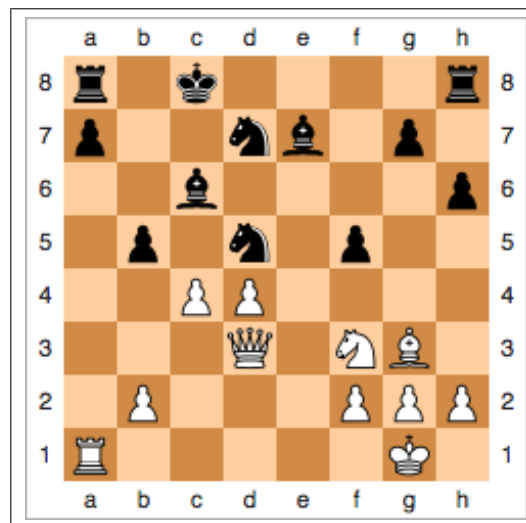


Figure 1. Endgame of Deep Blue and Kasparov Game 6. Kasparov let the pressure get to him and made a few mistakes. He accused IBM of cheating after the match, and asked for a rematch. IBM declined. [2]

a chess engine and computer developed by IBM finally beat the reigning world champion, Garry Kasparov, on February 10, 1997. However, Kasparov went on to win the six game match, and only lost a six-game match the next year in May of 1997 after being heavily upgraded.

Deep Blue's novel strengths lay in its hardware and brute force computing power. In June of 1997, it was the 259th most powerful computer in the world, and could evaluate 200 million moves a second. On the other hand, Deep Blue's evaluation function was not very strong, nor was its search function smart about pruning game nodes, relative to contemporary chess engines. Deep Blue's evaluation function was initially written in a generalized form with many parameters to be determined, and then optimal values for parameters were assigned by the system as it analyzed thousands of games. For example, in the generalized form, Deep Blue didn't know the relative importance of a safe king position compared to a space advantage in the center. Despite

the unsophisticated code (for 2016), Deep Blue was able to beat Kasparov and convince the world that computers were better at chess than humans. [2]

Since 1997, great improvements have been made in the world of chess programming. It is easy to download a chess engine onto one's laptop that could easily beat Deep Blue and the best human players. Like Deep Blue, state-of-the-art chess engines draw upon sophisticated search algorithms and finely tuned state-evaluation functions. The primary differentiator between good and bad chess engines is their search function, but evaluation functions play a large role as well. Unlike Deep Blue, most evaluation functions are not programmed in a general form. They are hard-coded and parameter values are set based on deep and detailed knowledge of chess and chess programming. Speed is prioritized over intricacy, as it is more important for a chess engine to be able to search 20 moves deep or more than to have the most precise evaluation at each of those moves.

1.2. Stockfish

I wondered, as others have before me, if neural networks could learn all of the detailed chess-specific minutiae encoded in an evaluation function by studying thousands of games. I set out to train a neural network to mimic the outputs of a specific evaluation function that of Stockfish, the best open-source chess engine as of early 2016.

Stockfish is a very powerful chess engine that is also open-source and available for free online. It is currently the second best ranked chess engine in the world, after Komodo. It has an ELO of 3318, and the highest human ELO is 2882, achieved by Magnus Carlsen. Its evaluation function is extraordinarily complicated. [3]

The evaluation function proper consists of 879 lines of C++ code, and calls upon many more helper functions not included in that line count. It assigns specific numerical scores to the value of different types of outposts (knights or bishops supported by pawns), rooks on open files, pieces on each individual square, and various other bonuses titled "Unstoppable," "BishopPawns," and "ThreatByPawnPush." Each of these bonuses corresponds to different early game and endgame numerical values. Suffice it to say Stockfish's evaluation function is complex and nuanced. It returns float values ranging from roughly -1000 to 1000. [4]

The goal of my work has been to design and train a neural network that takes chess boards as inputs and returns an approximation of this evaluation score. That is, I set out to train a neural network to distinguish between good (winning) or bad (losing) chess boards with some precision. My primary motivation was curiosity. Running a trained neural networks takes more time than running the complicated evaluation function and the time cost of the function will almost certainly make it a suboptimal choice for use in a chess engine. However, future work could uncover neural

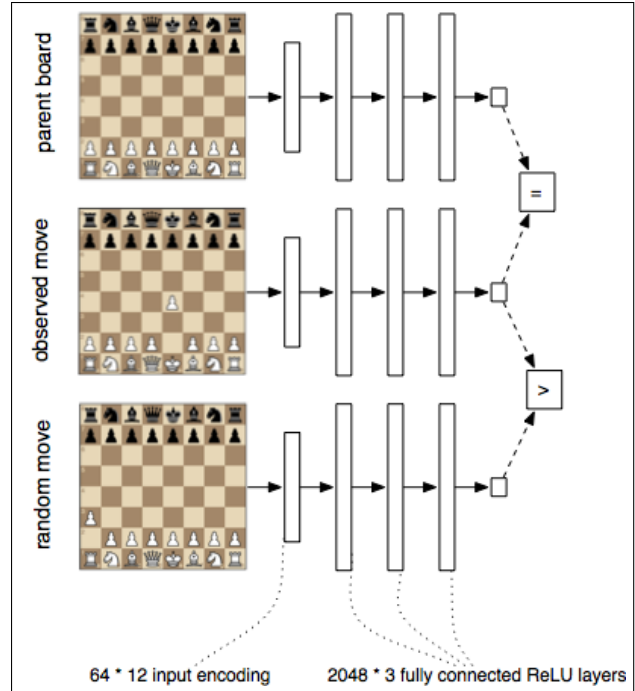


Figure 2. Bernhardsson's neural network. [5]

networks that do better than traditional evaluation functions, which would make them useful in the chess engine arms race.

2. Related Work

Many similar projects to this have been tried. Neural networks have been incorporated into chess engines in various forms, and have been used to develop game playing engines for other games. Despite the wealth of similar work, this project takes a novel approach to creating an evaluation function that could lay groundwork for stronger future evaluation functions. Because chess engines are programmed from people ranging from academics to hobbyists, some of this work has been done outside of academic settings.

2.1. DeepPink

For example, the previous project most similar to my own was carried out by Erik Bernhardsson, the engineer in charge of Spotify's recommendation engine, completed a similar project. Deep learning for chess, used a neural network to derive an evaluation function for a chess engine. However, instead of attempting to replicate a known evaluation function, like I did, Bernhardsson arrived at a wholly novel evaluation function by comparing boards that humans actually played with boards generated through random moves. He also trained his model with raw chess data it initially knew nothing about the game of chess. His network, written in Theano, was fed (p, q, r) triplets, where p

was the parent game node, q was the observed child node (with the move the human player made), and r was an alternate child node. Then, the network learned evaluation by assuming that a given human move did not change the value of the board that much, but a random move was bad for the value of a board. Bernhardtsson’s objective was

$$\begin{aligned} & \text{sum}_{(p,q,r)} \log S(f(q) - f(r)) + \\ & \kappa \log(f(p) + f(q)) \kappa \log(-f(q) - f(p)) \end{aligned} \quad (1)$$

An illustration of this relationship can be found in Figure 2. Bernhardtsson used 3 inner product layers 2048 units wide, a network architecture that I tried, and let his model train for 3 days. In the end, Bernhardtsson was partly successful. He created Deep Pink, a chess engine incorporating his evaluation function. Deep Pink beat another (unsophisticated) chess engine, Sunfish, some of the time, but it usually only won when it was given more time than Sunfish to compute moves. [5, 6]

2.2. AlphaGo

In the news recently, Google’s AlphaGo project used a neural network to build a go evaluation function. Go is another strategy game, and one of the last board games where humans were still able to beat computers. Go is particularly difficult to program an evaluation function for, because the strength of a position is determined primarily by nuanced placement of pieces, not the number or variety of pieces on the board. In Chess, on the other hand, a player with a queen is likely to be in a better position than a player lacking a queen, so evaluation functions can get further on kludgy comparing of piece counts.

The AlphaGo team trained their engine in 3 stages, the first of which is most similar to this project. They built a network of alternating convolutional layers and rectifier nonlinearities, with a final softmax layer, and fed it historical go games in order to generate a probability distribution of legal next moves given a game state. Their network had 13 layers and was trained from 30 million game positions. It predicted expert moves with 55.7% accuracy using only raw board position and move history as inputs. [7, 8]

The following two stages of the training pipeline used reinforcement learning rather than supervised learning. The second of these stages focused on position evaluation (the first refined the policy network for move prediction). It estimated a value function to predict the outcome of a game from position s of games played where both players use policy p .

$$v^p(s) = \mathbb{E}[z_t | s_t = s, a_t \dots T p] \quad (2)$$

While this project attempts to take raw game data and return an evaluation function, AlphaGo worked backwards from move selection to assign an evaluation function. At a typical game node, a go player is presented with more possible

moves than a chess player. AlphaGo consults its evaluation function for the game states after making all of the possible moves and uses that to narrow down its search. While many chess engines do something similar, the evaluation function is also referenced to weight the relative values of search branches, while AlphaGo leans more heavily on its policy network.

2.3. Other Projects

In the academic world, Amir Bans Automatic Learning of Evaluation, with Applications to Computer Chess, outlines a theoretical framework for learning chess evaluation functions. It details an algorithm that can be used to grade an evaluation function’s correctness (and formally defines the correctness of an evaluation function). Then, Ban suggests a learning algorithm to improve evaluation functions, whereby a given function is repeatedly altered and then graded, retaining changes that increase correctness and rejecting those that do not. Ban’s work here is mostly theoretical. As he admits, “This learning algorithm is not entirely possible to automate, as step 2 [where the evaluation function is altered] depends on the availability of successful improvements to the program, an availability that ultimately depends on the ingenuity and inventiveness of a programmer.” However, Ban does use the learning algorithm to apply improvements to his evaluation function to improve his own chess engine, Deep Junior. [9]

Sebastian Thrun’s Learning to Play the Game of Chess, takes a similar approach, arriving at a neural network generated evaluation function weighting boards by if they resulted in a win or a loss, and tracking backwards. A result of this learning scheme is that NeuroChess, the name given to the resulting chess engine, plays very poorly in the early game. Thrun’s networks are not very large - and are trained based on boards preprocessed to draw out important patterns. Nonetheless, an engine with Thrun’s evaluation function beats GNUChess (another chess engine) 13% of the time. [10]

There have been a few other similar attempts, but none have used the particular learning scheme attempted here. Papers by Sutskever, Nair, Madison, Huang, and Silver investigated the use of neural networks to learn go, demonstrating techniques and insights that have been incorporated into AlphaGo. [11, 12] Despite all of this work, the current state-of-the-art chess engines, including Stockfish, the best open-source engine (which I will incorporate into my research), do not incorporate deep learning techniques at all. But deep learning could be the way forward to faster, better chess engines of the future. It will likely take a significant amount of further study to achieve a comparable chess engine that relies heavily on deep learning, but this project is a step forward.

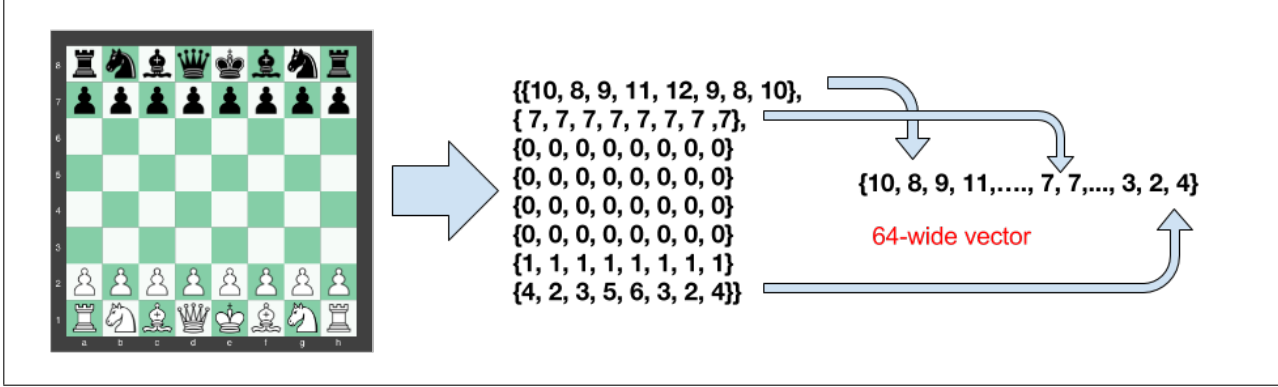


Figure 3. Illustrating how chess boards were turned into an array representation for training. Piece values are represented in Python as integers from 1 to 6, and adding 6 to the black values distinguishes the different colors.

3. Dataset

Like Bernhardsson, I downloaded datasets of played chess games from the Free Internet Chess Server (FICS) Games Database. I downloaded games from two months, April and May in 2015, played with standard time controls, between two humans, with average ratings of above 2000, which indicates quite a high level of human chess play. More games were readily available but the limiting factor in dataset size proved to be the time it took to process the data. I ended up processing sets of 10,000 and 100,000 games which were then used to train the network. These games were played by a wide range of human players. That is, they were not all played by one or two prolific players. The games include examples of all types of endgames. Checkmates, resignations, draws due to repetition, and draws due to mutual agreement are all included. [13]

4. Implementation

4.1. Technology Stack

Data processing code is written in Python. Games are read in PGN format with the Python Chess module and moves are communicated to Stockfish through the Universal Chess Interface (UCI) protocol.

Networks are trained and written using Marvin, a minimalist GPU-only N-dimensional ConvNet framework developed by the Princeton Vision Group.

Weights and results are parsed using Python.

4.2. Data Preparation

Processing chess game data for use in training a neural network proved to be a major component of this project. The first component of processing involves reading games from PGN format into Python's Chess internal representation, and then translating each board state in from the games into an array representation to be fed into Marvin. I heavily

```

[Event "FICS rated standard game"]
[Site "FICS freechess.org"]
[FICSGamesDBGameNo "390007994"]
[White "zhengqjf"]
[Black "kaater"]
[WhiteElo "2120"]
[BlackElo "1901"]
[WhiteRD "na"]
[BlackRD "na"]
[TimeControl "480+23"]
[Date "2016.01.01"]
[Time "02:04:00"]
[WhiteClock "0:08:00.000"]
[BlackClock "0:08:00.000"]
[ECO "B74"]
[PlyCount "123"]
[Result "1-0"]

1. e4 c5 2. Nf3 d6 3. d4 cxd4 4. Nxd4 Nf6 5. Nc3 g6 6.
Be2 Bg7 7. O-O O-O 8. Be3 Nc6 9. Nb3 Ne5 10. h3 Be6 11.
Nd4 Qd7 12. Nxe6 Qxe6 13. Nd5 Nxe4 14. Nc7 Qd7 15. Nxa8
Rxa8 16. c4 Rc8 17. Rc1 b6 18. b3 Rc7 19. f4 Nc6 20. Bf3
Nc5 21. Qd2 a5 22. Rfe1 Nb4 23. Red1 Nbd3 24. Rb1 Qd8 25.
a3 Qb8 26. Be2 Nb2 27. Rxb2 Bxb2 28. Qxb2 Ne4 29. Bf3 Nf6
30. Qd4 b5 31. cxb5 Qxb5 32. Qb6 Qxb6 33. Bxb6 Rc3 34.
Rb1 a4 35. bxa4 Rxa3 36. a5 e6 37. Bb7 Nd5 38. Bxd5 exd5
39. Rd1 Kf8 40. Rxd5 Ke7 41. Bc7 Kd7 42. Bxd6 Kc6 43.
Bxa3 Kxd5 44. Kf2 Kc6 45. Ke3 Kb5 46. Bb4 f5 47. Kd4 h6
48. h4 Ka6 49. Ke5 g5 50. fxe5 hxe5 51. hxe5 f4 52. Kxf4
Kb5 53. g6 Kxb4 54. a6 Kc3 55. a7 Kd2 56. a8=Q Ke2 57.
Qa3 Kd2 58. g7 Kc2 59. g8=Q Kd2 60. g4 Ke2 61. Qga2+ Kf1
62. Qc1# {Black checkmated} 1-0

ucinewgame
setoption name UCI_AnalyseMode value true
position startpos moves e2e4 c7c5
...

```

Figure 4. PGN to UCI translation. This UCI command creates the board after the first full move (both white and black have played). To process this entire game, each half move is sent to the engine as its own command. There are 123 such half-moves in this particular game.

modified Bernhardsson's data processing code for my purposes. [14] A depiction of this transformation can be seen in Figure 3.

As each board is being turned into an array, each successive move is translated from PGN format into the cor-

Figure 5. Right. Examples of boards where piece differential is not a good evaluation function. Quick recapture shows white up a pawn, but white has hung a pawn that black will capture in the next move, so Stockfish’s evaluation shows the opponents as equally matched. The last three boards are samples from the famous Alekhine Rheshvinsky 1937 game where Alekhine, playing white sacrificed his rook and queen to achieve Mate. Stockfish catches on to this scheme in the third board, but piece differential continues to swing in black’s favor even as black loses the game. These boards also serve as good examples of the power of Stockfish’s evaluation function. Stockfish finds Alekhine’s sacrifice plan before it is obvious. Alekhine is famous for his daring sacrifices but Stockfish is more than his equal.

Piece	Value
Pawn	1
Knight	3
Bishop	3
Rook	5
Queen	9
King	∞

Table 1. General piece values. Skilled chess players rely on more nuanced understandings than this. A piece’s positioning has a huge impact on its value. Bishops are commonly thought to be slightly more valuable than knights, but this also depends on a player’s preference or the demands of a particular game. In a given game, a white-square or black-square bishop can be much more powerful than its counterpart, depending on pawn structure.


responding UCI command and sent to an instance of the Stockfish engine, which calculates and returns an evaluation score for that game state. PGN to UCI translation can be seen in Figure 4.

After all the boards have been processed for a particular set of games, the board array representation are converted into data tensors and the Stockfish evaluation values are converted into label tensors. One quarter of the data is set aside for testing tensors.

4.3. Classification

Because I wanted to train a neural network on a classification problem rather than a regression function, I binned the evaluation labels into 10 buckets. I did this by taking the entire set of evaluation values and dividing it into 10 segments each containing roughly the same number of boards. So the range of values included in the edge buckets, 0 and 9, would be greater than the range of values in the central buckets, 4 and 5, because the edge buckets contain all the outlier boards. The network would then sort boards into one of these 10 buckets, rather than arriving at a precise evaluation number.


Quick recapture



Piece Differential: 1

Stockfish: 0.01


Calm before the storm



Piece Differential: 1

Stockfish: 1.08

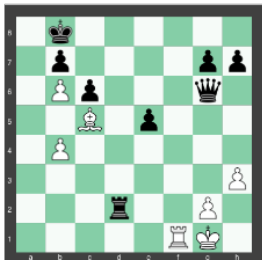
Likely mate



Piece Differential: -1

Stockfish: 34.81 and rising

Sacrificed for victory



Piece Differential: -7

Stockfish: White mate in 3

4.4. Piece Differential

Though some networks were trained on raw board data, I also processed game data with an extra heuristic designed

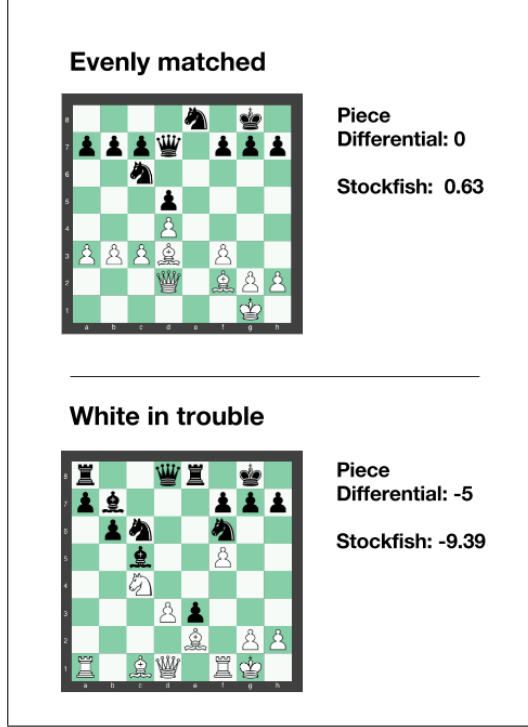


Figure 6. Examples of boards where piece differential is a good evaluation function. Both of these boards are fairly standard. Stockfish’s scores differ from the piece differentials because Stockfish is identifying positional strengths or weaknesses that give it a more nuanced view of the quality of the boards.

to help the neural network learn to differentiate between good and bad chess boards. This heuristic is piece differential. In conventional chess-playing strategy, each piece has a whole-number value. As a very naive evaluation function, the sum of the values of black’s pieces can be subtracted from the sum of the values of white’s pieces. Thus a higher piece differential correlates with a better board for white. However, piece differential is quite a bad evaluation function as there are many situations when it does not accurately capture the salient goodness or badness of a board. Some examples of when piece differential is ineffective can be seen in Figure 5. Examples of when piece differential is effective can be seen in Figure 6.

This piece differential is calculated as the boards are converted to arrays and is stored with the boards as the 65th value in a 65-wide vector. As I will show, this value is helpful for the neural network.

4.5. Networks

Once the data had been processed, I experimented with many different network designs. I ended up testing 3 different designs, all fully linked networks. The designs of the layers can be seen in Figure 7. These were the most ef-

Network	Initial Rate	Step Size	Gamma	Momentum
Net A	0.001	1000	0.1	0.9
Net B	0.01	10000	0.95	0.9
Net C	0.001	1000	0.5	0.97

Table 2. Learning rate schemes for the three networks.

fective of the designs I tested, some of which incorporated convolution layers or rectified linear unit activation layers.

Net A was my most accurate network. Net B was modeled after Bernhardsson’s network. Net C was my most accurate network on raw board data without piece differential included. Net A and Net B were trained on boards with piece differential included as the 65th value in the board vectors. Net C was trained on raw boards with 64-width vectors.

4.5.1 Loss

All three networks end with a final 10-width vector, where each space represents one of the 10 buckets of Stockfish evaluation scores. When the trained networks are input a board, they return this vector which, when normalized, is a probability distribution of what evaluation score the network thinks the board should receive.

This 10-width vector is fed into a softmax layer, which is then fed into a multinomial logistic stable softmax layer, which compares the network’s prediction to a label to determine if the prediction is correct.

The softmax function, also known as the normalized exponential, normalizes a k dimensional vector ν_k to return the vector μ_k :

$$\mu_k = \frac{\exp(\nu_k)}{1 + \sum_j \exp(\nu_j)} \quad (3)$$

This function is incorporated into a multinomial regression. [15]

4.5.2 Weight initialization

I used two different weight initialization methods in the three networks. Gaussian draws from an i.i.d. Gaussian distribution. Xavier transforms that distribution into another i.i.d. distribution but adjusts the variance to correspond to the dimensions of the network. Given a weight matrix W with dimensions $m \times n$ and variance $\text{Var}(W)$, Xavier initialization sets the variance such that $1 = n\text{Var}(W)$. It would be optimal if $1 = m\text{Var}(W)$ as well, but if n and m are different this is obviously impossible, so Xavier initialization sets weights proportionally to the size of the input vector n . [16]

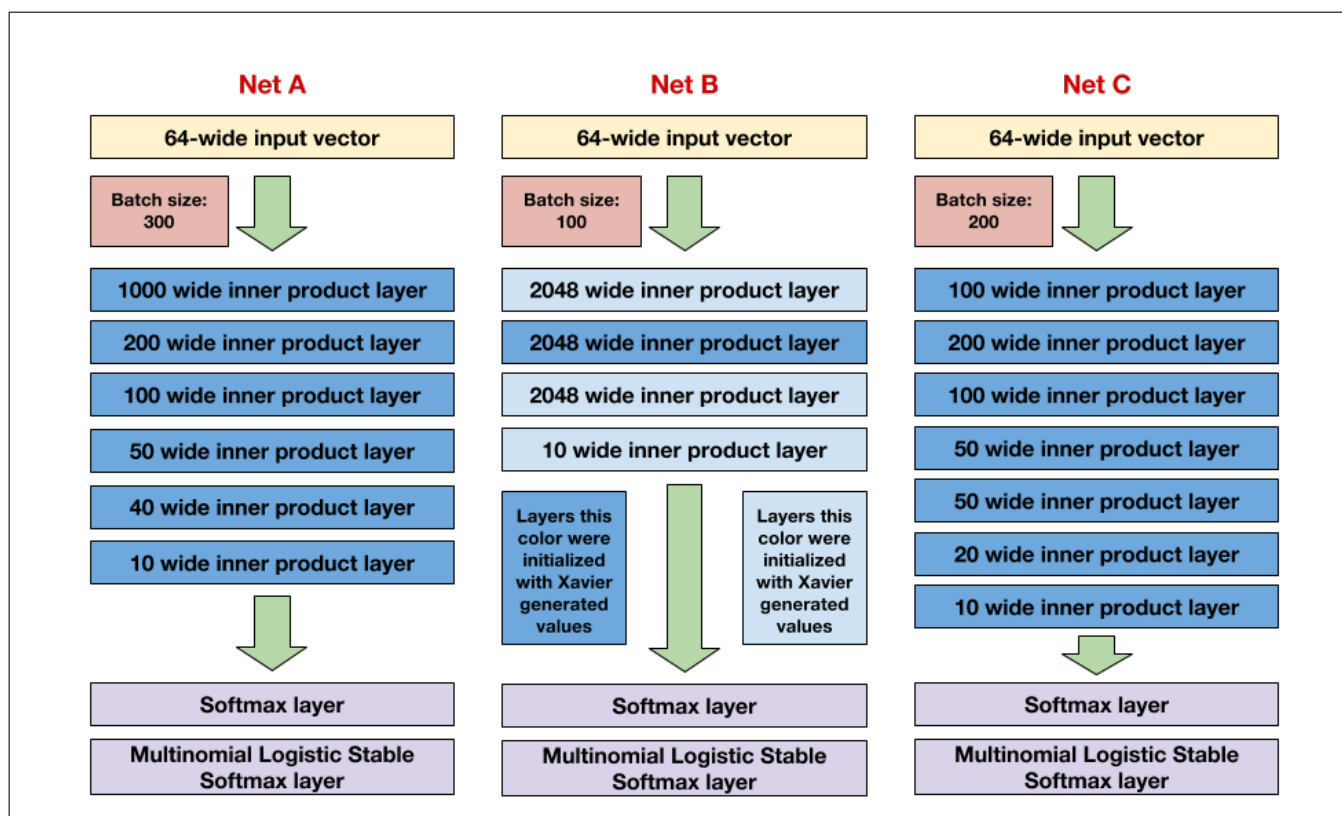


Figure 7. Examples of boards where piece differential is a good evaluation function. Both of these boards are fairly standard. Stockfish’s scores differ from the piece differentials because Stockfish is identifying positional strengths or weaknesses that give it a more nuanced view of the quality of the boards.

Network	Accuracy
Net A	34.03%
Net B	14.13%
Net C	19.60%

Table 3. Accuracy of networks on withheld test datasets. Net A and Net B were trained on data with piece differential precalculated while Net C was not.

4.5.3 Learning Rates

The learning rates of the three networks are represented in Table 2. I arrived at these rates through trial and error. I attempted to find rates that allowed the network to learn as rapidly as possible without diverging. In the end, I had success with rapidly decreasing learning rates. I found that the networks frequently diverged otherwise.

5. Evaluation

All three networks were not very accurate. Their relative accuracies are reflected in Table 3. However, there are successes in Net A and Net C. Building from a baseline

accuracy of 10%, both demonstrate measurable learning, with and without piece differential precalculated. This is especially exciting given the intricacy of Stockfish’s evaluation function and the fact that the untrained networks knew nothing about the rules of chess. Additionally, the strength of Net A over Net C is a testament to the usefulness of piece differential as an evaluation heuristic. Despite all its flaws, including the piece differential improves accuracy by 14.43%, a significant boost.

Moreover, perhaps the 34% accuracy statistic is deceptively low, because Net A often makes near miss predictions where it predicts one greater or one less than Stockfish’s actual evaluation. Sample boards and Net A’s predictions for them are shown in Figure 8. On the other hand, it is worth noting that Net A predicts each of the 10 scores with close to equal probability for each board. The winning score that Net A actually predicts is only slightly more probable than the other scores. This is true both with Net A is right and when it is wrong.

Net B, the network I modeled on Bernhardsson’s Deep Pink, was not very effective. It didn’t demonstrate any real learning and barely improved over the baseline, even when

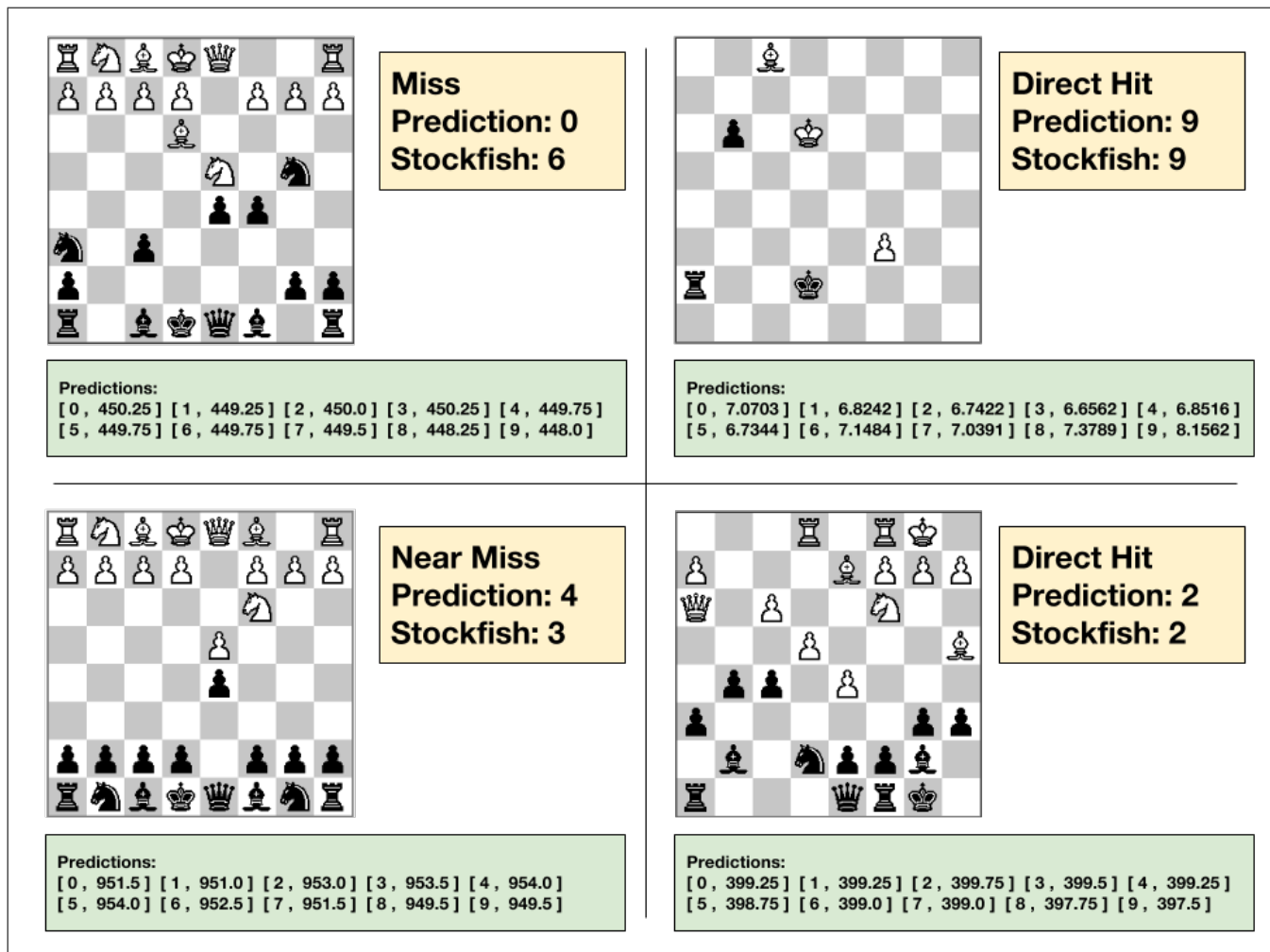


Figure 8. Net A's predictions of evaluation score for four test boards. With a true accuracy rate of 34%, this sample of Net A's predictions is not random, but rather chosen to highlight the network's strengths as well as show its weaknesses. In the two direct hits, the network succeeds at predicting both endgame and midgame scores. In the early game, shown in the near miss board, Net A is able to make close guesses but often is off by one. In the miss, Net A makes a totally incorrect prediction for seemingly no reason. Net A does this randomly throughout all types of boards. The prediction vectors shown below the boards demonstrate that Net A is never very certain about its predictions. The (not normalized) probabilities for each of the 10 classifications are relatively similar.

supplied with precalculated piece differentials. Net A and Net C, the networks that did show learning, made most of their improvements in the first segment of their training, before leveling off at roughly the same accuracy rate. Net B, in contrast, made basically no improvements throughout its training. Network prediction accuracy for Net A and Net B throughout their training process can be seen in Figures 9 and 10.

6. Conclusion, Limitations, and Future Work

I set out to design and train networks to evaluate chess boards. By and large I succeeded at that goal. Net A and Net C were successfully trained networks, and they arrive at

rough approximations of an evaluation function, both with and without precalculated piece differential. This project lays the groundwork for more sophisticated neural network evaluation functions, functions that could one day surpass hard-coded evaluation functions in accuracy and efficiency.

6.1. Limitations

There are two main limitations with Net A and Net B. The first is that they are not incredibly accurate, nor, by grouping boards into 10 buckets, are they very precise. With more work, more accurate and precise networks could be trained. The second limitation is that neural networks are not a very time efficient means of computing an evaluation function. Potential speedups in network processing power,

as well as software optimization could make neural networks competitive in the future. Additionally, if a network becomes significantly more accurate than a traditional evaluation function, it could be worth the extra time cost.

6.2. Future Work

A number of steps forward present themselves. Processing more chess games to create larger training datasets could increase the accuracy of these networks. Different network architectures could be experimented with, as well as different learning schemes. Networks could be incorporated into full-functioned chess engines and evaluated on their ability to win chess games. Once incorporated into full chess engines, reinforcement learning becomes a viable option. This project is a foundation upon which future work can build. Weights from Net A or Net C could be the initialization weights for an evaluation function trained through reinforcement, for example.

Acknowledgements: Thank you to Professor Jianxiong Xiao and Fisher Yu for their guidance and support throughout this project.

Honor Code: I pledge my honor that this project represents my own work in accordance with University regulations.

Joe Sheehan

References

- [1] Wikipedia, "Computer chess." https://en.wikipedia.org/w/index.php?title=Computer_chess&oldid=713713006. [Online]. Accessed 28 Apr. 2016.
- [2] Wikipedia, "Deep Blue versus Kasparov, 1997, Game 6." https://en.wikipedia.org/w/index.php?title=Deep_Blue_versus_Kasparov,_1997,_Game_6&oldid=709083484. [Online], Accessed 21 Apr. 2016.
- [3] Pete, "The 5 Best Computer Chess Engines." <https://www.chess.com/article/view/the-best-computer-chess-engines>. [Online]. Accessed 15 Feb. 2016.
- [4] Stockfish, "Stockfish source code." <https://github.com/official-stockfish/Stockfish>. [Online]. Accessed 20 Mar. 2016.
- [5] E. Bernhardtsson, "Deep learning for chess." <http://erikbern.com/2014/11/29/deep-learning-for-chess/>. [Online]. Accessed 26 Apr. 2016.
- [6] E. Bernhardtsson, "Deep pink." <https://github.com/erikbern/deep-pink/>, Apr. 2016. [Online]. Accessed 24 Apr. 2016.

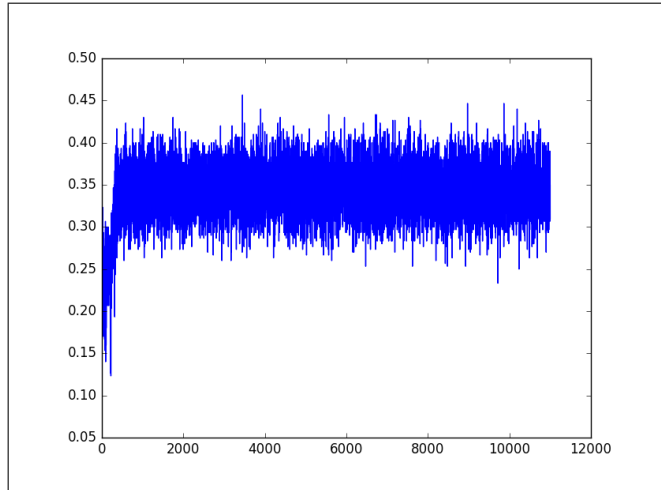


Figure 9. Net C makes strong initial progress and then flattens out.

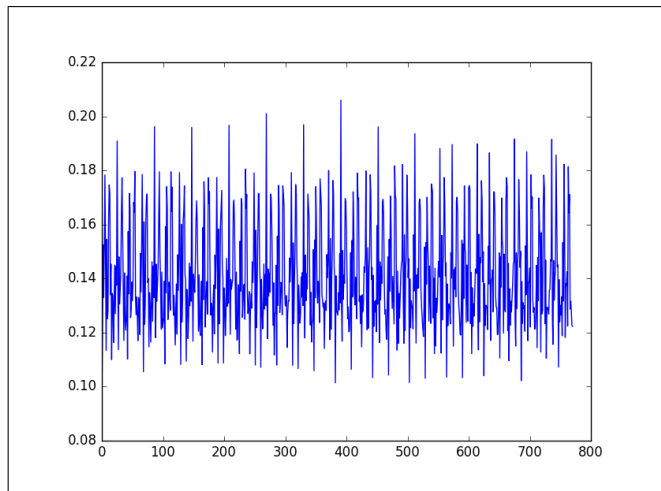


Figure 10. Net B does not make any progress. Net B logged evaluation every 100 iterations which is why X-axis values are smaller than those in Figure 9.

- [7] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, "Mastering the game of Go with deep neural networks and tree search," *Nature*, vol. 529, pp. 484–489, Jan. 2016.
- [8] "AlphaGo: using machine learning to master the ancient game of Go." <https://googleblog.blogspot.com/2016/01/alphago-machine-learning-game-go.html>. [Online]. Accessed 28 Apr. 2016.
- [9] A. Ban, "Automatic Learning of Evaluation, with Applications to Computer Chess | The Hebrew University of Jerusalem - Center for the Study of Rationality," Jun 2012.

- [10] S. Thrun, “Learning to play the game of chess,” *Advances in neural information processing systems*, vol. 7, 1995.
- [11] C. J. Maddison, A. Huang, I. Sutskever, and D. Silver, “Move Evaluation in Go Using Deep Convolutional Neural Networks,” *arXiv:1412.6564 [cs]*, Dec. 2014.
- [12] I. Sutskever and V. Nair, “Mimicking Go Experts with Convolutional Neural Networks,” in *Artificial Neural Networks - ICANN 2008* (V. Krkovic, R. Neruda, and J. Koutník, eds.), no. 5164 in *Lecture Notes in Computer Science*, pp. 101–110, Springer Berlin Heidelberg, Sept. 2008.
- [13] “FICS Games Database - Statistics for 2015.” http://www.ficsgames.org/2015_stats.html. [Online]. Accessed 12 Apr. 2016.
- [14] E. Bernhardsson, “train.py.” <https://github.com/erikbern/deep-pink/blob/master/train.py>, Apr. 2016. [Online]. Accessed 24 Apr. 2016.
- [15] C. M. Bishop, *Pattern Recognition and Machine Learning*. Springer, 2006.
- [16] G. Larsson, “Initialization of deep networks,” *Deepdish*, Nov 2015. [Online]. Accessed 12 Apr. 2016.