



Experimenting with Time and Depth Control in Chess Engines

by

Guðmundur Páll Kjartansson

Project Report of 30 ECTS credits submitted to the Department of
Computer Science at Reykjavík University in partial fulfillment of the
requirements for the degree of

Master of Science (M.Sc.) in Computer Science

December 2019

Examining Committee:

Yngvi Björnsson, Supervisor
Professor, Reykjavík University, Iceland

Stephan Schiffel, Examiner
Assistant Professor, Reykjavík University, Iceland

Hrafn Loftsson, Examiner
Associate Professor, Reykjavík University, Iceland

Copyright
Guðmundur Páll Kjartansson
December 2019

Experimenting with Time and Depth Control in Chess Engines

Guðmundur Páll Kjartansson

December 2019

Abstract

Chess programming has come a long way since 1996 when Deep Blue defeated world champion Garry Kasparov. Deep Blue was the result of many years of labor of knowledge-engineering, where chess-specific features were hand-crafted and carefully hand-tuned. Recently, AlphaZero received worldwide attention for mastering the games of chess, Shogi, and Go through machine-learning and self-play using no game-specific knowledge features (except the rules of the games).

In this project, we examine ways to improve chess playing programs using machine learning methods. In our first set of experiments, we test whether a neural network can be trained to determine volatility of chess positions, i.e., whether they are stable or not, but such information may be used to improve time management. In our second set of experiments, we use regression methods to automatically determine the value of the parameters used in a chess engine for depth reduction in search. We use the world-class program Stockfish for our experiments, and although our experiments did ultimately not lead to improved play, some hold promise, for example, in classifying chess positions as volatile or not.

Tilraunir með tíma- og dýptarstjórnun í skákvélum

Guðmundur Páll Kjartansson

desember 2019

Útdráttur

Skákforritun er langt komin síðan árið 1996 þegar Deep Blue sigraði heimsmeistarann Garry Kasparov. Deep Blue var afrakstur margra ára þekkingarverkfræðilegrar vinnu þar sem sértækir eiginleikar skákar voru handsníðaðir og vandlega handstilltir. Nýlega fékk AlphaZero athygli um allan heim fyrir að hafa náð tökum á skák, Shogi og Go í gegnum vélanám og sjálfspilun án þess að byggja á sértækum eiginleikum umræddra leikja (öðrum en reglum leikjanna).

Hér skoðum við leiðir til að bæta skákvélar með því að nota vélnámsaðferðir. Í fyrstu tilraunum okkar prófum við hvort hægt sé að þjálfa tauganet til að ákvarða óstöðugleika skákstaða, en slíkar upplýsingar má nota til að bæta tímastjórnun. Í seinni tilraunum okkar notum við aðhvarfsaðferðir til að ákvarða sjálfkrafa gildi breytanna sem notaðar eru í skákvél til að draga úr leitardýpt. Við notum heimsklassa skákvélina Stockfish við tilraunir okkar. Þó að tilraunir okkar hafi á endanum ekki leitt til umbóta í tefldum skákum þá benda sumar þeirra til annarskonar umbóta, til dæmis í því að flokka skákstöður sem óstöðugar eða ekki.

Experimenting with Time and Depth Control in Chess Engines

Guðmundur Páll Kjartansson

Project Report of 30 ECTS credits submitted to the Department of
Computer Science
at Reykjavík University in partial fulfillment of
the requirements for the degree of
Master of Science (M.Sc.) in Computer Science

December 2019

Student:

.....
Guðmundur Páll Kjartansson

Examining Committee:

.....
Yngvi Björnsson

.....
Stephan Schiffel

.....
Hrafn Loftsson

The undersigned hereby grants permission to the Reykjavík University Library to reproduce single copies of this Project Report entitled **Experimenting with Time and Depth Control in Chess Engines** and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves all other publication and other rights in association with the copyright in the Project Report, and except as herein before provided, neither the Project Report nor any substantial portion thereof may be printed or otherwise reproduced in any material form whatsoever without the author's prior written permission.

.....
date

.....
Guðmundur Páll Kjartansson
Master of Science

I dedicate this project to my parents.

Acknowledgements

This work was funded by 2014 RANNIS grant “Hermi- og brjóstvitstrjáleit í alhliða leikjaspilun og öðrum flóknum ákvörðunarvandamálum.”

Contents

Acknowledgements	xi
Contents	xii
List of Figures	xiv
List of Tables	xvi
List of Abbreviations	xvii
1 Introduction	1
1 Introduction	1
2 Background	3
1 Chess Engines	3
1.1 Chess State	3
1.2 Evaluation Function	4
1.3 Game Tree	5
1.4 Tree Search	5
1.5 Search Window	5
1.6 Quiescence Search	6
1.7 Search Depth Reduction	7
2 FishTest	7
3 Deep Neural Network	8
I Time Management	11
3 Methods	13
1 Deep Neural Clock	13
2 Neural Network Architectures	14
3 Training and Testing the Networks	16
4 Evaluating the Networks	17
4 Results	19
1 Experimental Setup	19
2 The Dense Network Architecture	19
3 The Convolutional Network Architecture	21
4 Comparing the Architectures	22
5 Distinguishing Stable vs. Nonstable Positions	23

6	Testing in Gameplay	24
7	Summary	25
II Depth Reduction		27
5	Methods	29
1	Late-Move-Reductions in Stockfish	29
2	Gathering and Labelling the Data	31
3	Regression and Normalizing	32
4	Matches against the original Stockfish	33
6	Results	35
1	Gathering and Labelling the Data	35
2	Regression and Normalizing	35
3	Match Results	38
4	Other Experiments	38
4.1	Hand-Picked Parameters	38
4.2	Grid Search	39
4.3	Non-linear Modifications	39
5	Summary	40
III Conclusion and Future Work		41
7	Conclusions and Future Work	43
Bibliography		45
A	Code	47
1	Logging code	47
1.1	Opening positions	47
1.2	Logger	48
2	Labelling code	50
3	Stockfish modifications	53
3.1	First version	53
3.2	Second version	54

List of Figures

2.1	An example of a chess position.	3
2.2	Search tree of Tic-Tac-Toe, as an example. Note that there should be 9 edges from the first depth (depth 0) to the next, and then 8 edges and so on, but some edges were removed because of symmetry (I.e. if X plays in the center, O can reply by either playing in a corner or an edge, and all those positions are rotationally symmetric). Image source: [7]	6
2.3	Here the triangle represents a search tree, where the height corresponds to a given search depth and width corresponds to the ordered move number. The black line shows how the search depth is reduced by the formula used by Stockfish. Notice that the first 2-3 moves are always searched at full depth. This is because the moves are ordered in such a way that the first few moves in the list are the ones that seem most promising and thus are searched at full depth.	8
2.4	One neuron in a neural network	9
2.5	A 3×3 filter is propagated through the input which has shape 12×12	10
3.1	A simple full-forward architecture, where each layer is fully connected to the next. The board encoding is bundled together with castling and other parameters in the same input layer.	15
3.2	In the above architecture, the chess board goes through a series of convolutional layers, while other parameters go through dense layers. The result is then combined and goes through another series of dense layers.	16
3.3	For preprocessing, we create a separate 0-1 matrix for each type of piece for each color, where 1 or 0 indicates the presense or absense of that piece in a given position on the board.	17
4.1	Accuracy on the training data vs validation accuracy from tuning the number of neurons, while keeping layers fixed at 2 and $\beta = 18.0$. In the top graphs we use dropout layers set at 0.5 for regularization and in the bottom graphs we have no such regularization.	20
4.2	Accuracy on the training data vs validation accuracy from tuning the number of layers, while keeping neurons fixed at 50 and $\beta = 18.0$	20
4.3	Accuracy on the training data vs validation accuracy from tuning the beta parameter, while keeping neurons fixed at 50 and layers fixed at 2.	21
4.4	Accuracy on the training data vs validation accuracy from tuning the number of neurons, while keeping layers fixed at 2 and $\beta = 18.0$	21
4.5	Accuracy on the training data vs validation accuracy from tuning the number of layers, while keeping neurons fixed at 100 and $\beta = 18.0$	22

4.6	Accuracy on the training data vs validation accuracy from tuning the β parameter, while keeping neurons fixed at 100 and layers fixed at 2.	22
4.7	On the left side we have a Cumulative Gains curve, which shows how well our model does at signifying correctly that Stockfish should stop searching. The blue line shows an ideal model which always outputs correctly and the baseline shows how well a random model would do on average. Our model is on the orange line and clearly has a significant advantage over a random output. On the right side we have the ROC curve, which plots the false positive rate against the true positive rate. Our area under the curve is 0.71, while an ideal area would be 1.0.	24
5.1	Visual map of depth reduction using equation 5.1.	31
6.1	The difference in number of nodes searched between Stockfish using the modified equation and the original Stockfish is shown on the y-axis. The scale factor used on the equation found through regression is on the x-axis. The smaller the scaling factor, the lesser the reduction parameter will be and thus Stockfish will search through a greater amount of nodes.	37
6.2	Visual map of depth reduction using the new equation.	38
6.3	Visual map of depth reduction using equation 6.3	40

List of Tables

4.1	Table data for the ROC curve for stability, sorted by FPR	24
5.1	The upper limit of allowable reduction in search depth is $11 - 3 = 8$, because the search outcome does not change from depth 3 to depth 11.	32
6.1	Regression attempts using different terms in the equation. Note that some of the R^2 scores are negative, implying that the formula found through regression is a worse fit through the data than a horizontal line.	36

List of Abbreviations

FEN	Forsyth–Edwards Notation
PGN	Portable Game Notation
UCI	Universal Chess Interface
CP	Centipawn
LMR	Late Move-Reduction
ROC	Receiver operating characteristic
FPR	False Positive Rate
TPR	True Positive Rate

Chapter 1

Introduction

1 Introduction

Chess programming has come a long way since February 10th 1996. On that fateful day, the chess engine Deep Blue defeated then world chess champion Garry Kasparov. Since then, chess engines have continued to improve well beyond human capability. Those early engines relied heavily on hand-optimized and fine-tuned features for evaluation and search optimization. This is still the case for most chess engines today, including the contemporary open source engine Stockfish [1], which has dominated most computer chess tournaments in recent years.

In the last few years, neural network assisted chess engines have been making an appearance. This is a significant shift in methodology, relying on self-optimization through deep learning, rather than on hand-optimized features and heuristics. Early attempts include the engines Deep Chess developed at Tel Aviv University [2] and Giraffe developed by Matthew Lai [3], which both managed to achieve grandmaster-level performance, but not coming close to the capabilities of the best hand-optimized engines.

AlphaZero, a computer program developed by the artificial intelligence company DeepMind, recently received worldwide attention for mastering the games of chess, Shogi, and Go [4] through self-play alone. The program achieved a super-human level of play within 24 hours of learning, as was demonstrated by the program defeating the top computer programs for each game, which were already better than the strongest humans. The AlphaZero program did effectively learn, using deep neural networks, how to evaluate game positions for those games, as well as to explore the different variations of play effectively and intelligently.

In chess, within four hours of self-play (using massive computing resources), it was able to defeat the previously strongest chess engine, Stockfish [1], somewhat convincingly in a match. However, there was some criticism on the experimental setup of that match, for example, as seen from the following quote by the main author of Stockfish:

The match results by themselves are not particularly meaningful because of the rather strange choice of time controls and Stockfish parameter settings: The games were played at a fixed time of 1 minute/move, which means that Stockfish has no use of its time management heuristics (lot of effort has been put into making Stockfish identify critical points in the game and decide when to spend some extra time on a move; at a fixed time per move, the strength will suffer significantly). [5]

An important aspect of chess, both for humans and computers, is to be able to manage time well, that is, in which positions should one invest time to analyze deeply and in which not. This was, however, not learned by AlphaZero. There has so far been somewhat limited attention placed on this aspect of computer chess.

In the first part of the project, we do some preliminary work on automatically learning time-management in chess using neural networks, using Stockfish as our testbed. In the second part of the project, we slightly change course, and automatically learn search-reduction parameters used by the search process in Stockfish.

Both the abovementioned learning tasks were done in supervised learning settings, that is, we first collected and labeled data that was used as the ground truth. On both tasks, our methods showed a clear ability to learn patterns from the dataset, although ultimately they failed to improve upon Stockfish play. This is maybe not too unsurprising given that Stockfish play is already carefully fine-tuned.

The project is structured as follows. In the following chapter, we provide the necessary background. This is followed by part one, which explores time-management, consisting of two chapters on methods and results, respectively. Part two, on search-depth reductions, which follows, is built up in the same way. Finally, we conclude and discuss future work.

Chapter 2

Background

In this chapter we introduce the terminology used throughout the project and discuss necessary background work.

1 Chess Engines

We used the open-source chess engine Stockfish in our work, but it is one of the strongest chess engines in the world [1]. We use version 8, which was the latest stable release when this work started (as well as the version that AlphaZero played). There are two main components to all such engines: the search, or thinking ahead, component and the position-evaluation component, which is used to statically evaluate the merits of different chess positions. Stockfish uses a traditional depth-first alpha-beta-based game-tree search for thinking ahead, but enhanced with many advanced techniques for being able to expand the search tree more selectively. For evaluating chess positions it uses a highly tuned hand-crafted evaluation function. We now introduce some common chess program terminology.

1.1 Chess State

A given chess state is a position on the board that can be reached after a sequence of legal moves. Each square on the board is either empty or contains one of 6 different

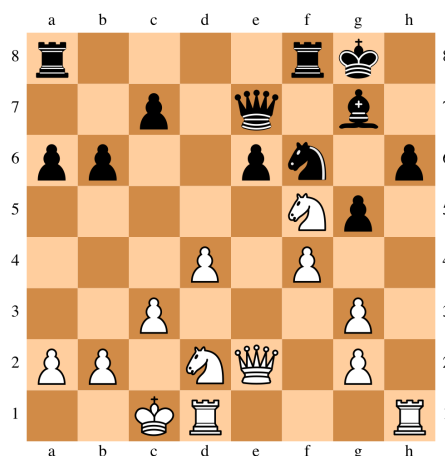


Figure 2.1: An example of a chess position.

types of pieces belonging to one of two players, black or white. The state contains the aforementioned board, along with the following additional information:

1. Which player, black or white, should make the next move
2. Castling rights for each player, signifying whether or not it is still legal for them to castle on the king and/or queen side.
3. An en-passant move, if such a move is currently available (implying that in the last state, a player moved a pawn two squares forward)
4. The number of moves made since a pawn was moved by either player (if neither player has moved a pawn for 50 moves, the game concludes in a draw)
5. How often the current board position has occurred before (threefold-repetition concludes the game in a draw)

The chess state is often encoded in string format using the Forsyth–Edwards Notation **FEN**. For example, the position in Figure 2.1 would be written as:

`r4rk1/2p1q1b1/pp2pn1p/5Np1/3P1P2/2P3P1/PP1NQ1P1/2KR3R b - -`

Here, rows are separated by the `/` character. Letters represent pieces, where lower and uppercase represent black and white pieces, respectively. Numbers represent a sequence of empty squares. As an example, `'r4rk1'` means: a black rook, followed by 4 empty squares, followed by a black rook and a black knight, and finally an empty square (all in one row).

For listing the game history, as opposed to a single chess position, the Portable Game Notation **PGN** is the de-facto standard. It shows the sequence of moves from the starting position leading to a given chess state. This notation is preferred when recording entire games, while **FEN** is preferred for recording a chess state regardless of the sequence of moves leading up to it. The **PGN** for Figure 2.1 is given below (often there is also meta-data about players, tournament and result in the header, but we omit it here):

```
1.d4 Nf6 2.Nf3 e6 3.Bg5 h6 4.Bh4 d6 5.c3 g5 6.Bg3 Nh5 7.e4 Bg7 8.Bd3 Nd7
9.Nbd2 b6 10.Qe2 Bb7 11.Nc4 Qe7 12.Nfd2 Nxe3 13.hxe3 a6 14.f4 d5 15.exd5 Bxd5
16.Ne3 Bb7 17.Be4 Bxe4 18.Nxe4 f5 19.Nd2 Nf6 20.0-0-0 0-0 21.Nxf5
```

1.2 Evaluation Function

The evaluation function is defined as $V : \mathbb{S} \rightarrow \mathbb{R}$, where \mathbb{S} is the state space and \mathbb{R} is the set of real numbers. The evaluation function will assign a number to each state, determining how desirable the state would be for the current player if it would be reached. Ideally, the state assigned the highest value is the one associated with the highest probability of winning. The value of a chess state is usually measured in a unit called the **centipawn**, or one-hundredth of the value of a pawn. Note that the value does not only depend on material advantage (such as having captured more pawns than the opponent), but also on the positioning of the pieces.

1.3 Game Tree

The game tree is an organized collection of all accessible chess states. The root node is the current chess state and the other nodes are states reachable from the current state by some sequence of legal moves. Each edge therefore represents a legal move from a given chess state. An example game tree for the game of Tic-Tac-Toe is shown in Figure 2.2.

1.4 Tree Search

A tree search is a search through the game tree, looking for the best available move for the current player. The best known example is the alpha-beta tree search [6], which is a depth-first algorithm which keeps track of the upper and lower bounds on the evaluation function, cutting away branches of the tree which will provably never be reached if the opponent plays optimally. The lower bound is referred to as alpha, or α , and the upper bound is referred to as beta, or β . Chess engines, including Stockfish, almost universally use a method called **iterative deepening**, where they search the current game state progressively deeper in each iteration, e.g., first to depth 1, then depth 2, etc., until the allotted time is up. This has two benefits, first for easier time management, but secondly, and more surprisingly, it turns out it is more effective to search the game tree iteratively from 1 to a given depth d , than directly to depth d . The reason behind this is that information gathered during previous iterations can be used to improve the move-ordering in subsequent iteration, resulting in additional α/β cutoffs and consequently smaller search trees. Algorithm 1 shows a basic iterative-deepening routine using time management.

Algorithm 1: Iterative deepening

```

Input : chessPosition
1 depth = 1;
2 while depth < MAX_DEPTH do
3   bestMove = alphaBetaSearchDepthLimited(chessPosition, depth) ;
4   if not enough time left then
5     | return bestMove;
6   end
7   depth = depth + 1 ;
8 end
9 return bestMove

```

1.5 Search Window

The lower bound (known as alpha, or α) and the upper bounds (known as beta, or β) in alpha-beta search is often referred to as its search window. A search with a certain window size can either fall within a given window or fail low or high. A position with value V fails low if $V \leq \alpha$, signifying that the given position will never be reached because it is too uninteresting, we can reach a stronger position by choosing a different move further up in the tree. The opposite case when $V \geq \beta$ is called failing high and means we have reached a position which is better than we can hope for, since

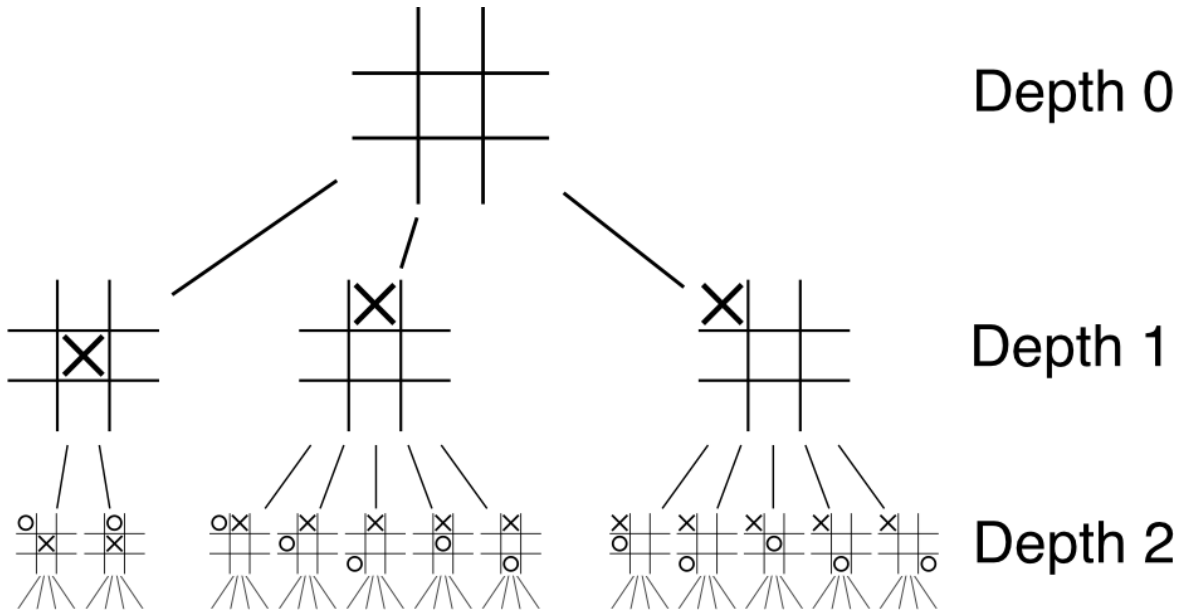


Figure 2.2: Search tree of Tic-Tac-Toe, as an example. Note that there should be 9 edges from the first depth (depth 0) to the next, and then 8 edges and so on, but some edges were removed because of symmetry (I.e. if X plays in the center, O can reply by either playing in a corner or an edge, and all those positions are rotationally symmetric). Image source: [7]

our opponent could have prevented it by playing a different move (provided that he is playing optimally).

If a node falls within a given search window, it is referred to as a Principal Variation node (or PV-node). If a node contains a beta-cutoff (which occurs when failing-high), it is referred to as a Cut-Off node or a Fail-High node. Otherwise it is known as a Fail-Low node. Advanced variants of alpha-beta search, such as Negascout [8] and Principal Variation Search (PVS) [9], first employ a **null window search**, where $\alpha = \beta - 1$, on all nodes other than the principal variation, and only if that search fails high we search the node again with a full alpha-beta window. In simpler terms, we do a **null window search** with the assumption that the move we are considering is not stronger than the current principal variation (in other words we assume that the score is $\leq \alpha$), if we get a score $> \alpha$ then we potentially found a move which is actually stronger than the principal variation and we want to verify that by doing a search with a full window. Given a good move ordering, this typically results in smaller search trees. Stockfish uses this approach.

1.6 Quiescence Search

After reaching a terminal node in the regular tree search, the merit of the corresponding chess position is assessed by calling the evaluation function. However, it is sometimes the case that the terminal position is in a volatile state that cannot be evaluated accurately statically. Say, for instance that a piece was captured in the last move and the opponent can capture immediately, but then it might be sensible to continue the tree-search for a little longer until one has resolved all sensible recaptures. The same can be said for positions where the king is in check. If we do not play out those

moves, we could potentially have the wrong associated score for the current position (i.e. a player might seem like he has a material advantage when that really is not the case). This procedure of doing a limited search expanding only promising captures and checks until the position looks stable is called **quiescence search**.

1.7 Search Depth Reduction

Given a chess position, generating all the legal moves is computationally easy. However, searching a move to a large depth is computationally heavy. Thus it is advisable to order the moves in some way so that the most promising looking ones are evaluated first. What Stockfish does is to give each move a score based on the positioning of the pieces and whether the move is a capture or evasion. The moves are sorted based on this score and the most promising ones are searched through first.

In order to save computation time, we want to reduce the search depth for less interesting moves in a given position. Given a seemingly uninteresting move and a depth reduction value $r < d$ (where d is the full search depth), we can employ a null-window search at depth $d - r$ with bounds $-(\alpha + 1)$ and $-\alpha$. Since $-\alpha$ is an upper bound for our opponents score, we are assuming this reduced depth search to fail low. If this assumption fails and the reduced search fails high, we should reconsider the depth reduction and do a full-depth search. This procedure is often referred to as **Late Move-Reduction (LMR)** and is the implementation of search depth pruning used by Stockfish. It is worth noting that unlike alpha-beta pruning, this is speculative pruning that does not guarantee that the best move is preserved. However, the underlying premise is that the time saved by searching unpromising branches more shallowly, is better used instead to reach an overall deeper iteration-depth. Stockfish currently uses the following formula for calculating the depth reduction term r :

$$r = \left\lfloor \frac{1}{1.95} \log(d) \log(m) \right\rfloor \quad (2.1)$$

In the above formula, d means search depth and $m \in [0, 1\dots]$ means move number. r is how much the search depth should be pruned. The relation between the inputs and depth reduction can be seen in Figure 2.3.

It is apparent that Equation 2.1 gives equal weight to the depth and move number parameters. We will investigate whether this seems like the correct decision to make. We also want to investigate the correlation between reduction depth and other state variables, such as the α number from the search tree and the difference in material value. This experiment and its results are presented in part II of the project.

It is worth mentioning that Stockfish uses several other speculative pruning methods in addition to late-move reductions, but they fall outside the scope of our work.

2 FishTest

Stockfish has an automated way of testing whether modifications to its source code improve the playing strength of the program, and should thus be merged into the main branch of the open source program. This testing framework is called FishTest. It uses volunteer driven distributed computing to run a tournament between the most current version of Stockfish and a version with the proposed modifications. Submitters are advised to make their changes as small and compact as possible and to test each

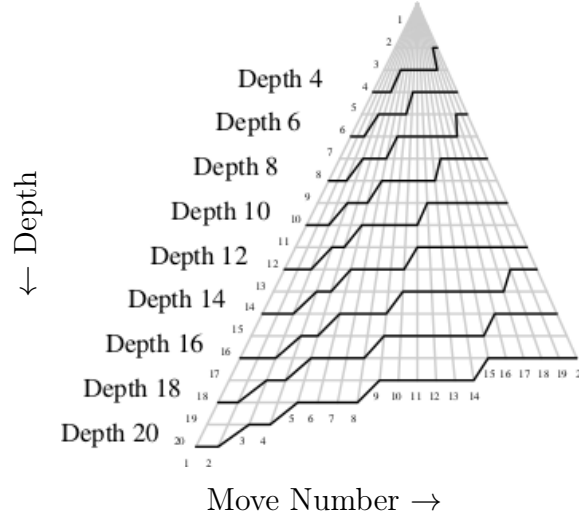


Figure 2.3: Here the triangle represents a search tree, where the height corresponds to a given search depth and width corresponds to the ordered move number. The black line shows how the search depth is reduced by the formula used by Stockfish. Notice that the first 2-3 moves are always searched at full depth. This is because the moves are ordered in such a way that the first few moves in the list are the ones that seem most promising and thus are searched at full depth.

idea separately before combining them. FishTest uses a Sequential Probability Ratio Test (SPRT) as a stop condition for evaluating the strength of a modification. The SPRT will have the null-hypothesis that the engines are equal in strength and the alternative hypothesis that one of the engines is stronger. The test terminates when the probability that either hypothesis is correct reaches a certain threshold. The test has two stages, the fast and the slow stage. Each proposed modification to Stockfish must first pass through the fast stage and then through the slow stage.

3 Deep Neural Network

A deep neural network is a machine-learning method for approximating some function $f^* : X \rightarrow Y$. In the context of classification, the function f^* might map each $x \in X$ to some category $y \in Y$. In the context of tree search, f^* might be the evaluation function, and then X would be the set of possible states and Y the set of real numbers. The neural network defines a new mapping $f(x; \theta)$, where θ is a collection of parameters tuned by the network in order to find the best approximation of f^* . From the universal approximation theorem, we know that all bounded continuous functions can be approximated by an artificial neural network with enough neurons [10].

A deep neural network consists of one or more input layers, and two or more layers of neurons. The layers are connected in a sequence, though they might split or merge. Each neuron in a layer takes the outputs from the neurons in the previous layer, usually along with an extra bias neuron which always outputs 1, as an input vector $\mathbf{x} = [x_0, x_1, \dots, x_n, 1]$ and computes their weighted sum $\mathbf{w} \cdot \mathbf{x}$, where $\mathbf{w} = [w_0, w_1, \dots, w_n, w_b]$ is a vector of weights which is then fed through the activation function h of the neuron to produce an output:

$$y = h(\mathbf{w} \cdot \mathbf{x})$$

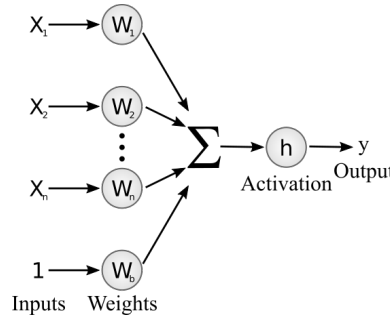


Figure 2.4: One neuron in a neural network

See Figure 2.4 for a clearer demonstration.

Backpropagation is almost universally used for updating the weights in neural networks. It works by first calculating the gradient of the error function for the output of the neural network and then working backwards by recursively using the chain rule of calculus to find the gradient for each neuron in the previous layers with respect to the parameters involved in the activation function of each of those neurons. The result is a cumulative gradient for the whole neural network. A backpropagation algorithm will take one step in the negative direction of the gradient, in hope of reaching a local minimum. The global minimum of this gradient is the optimal configuration of the network given the training data processed. In most cases, there will be multiple local minima and generally there is no way of knowing if one has reached the global minimum or not. For solving this, multiple optimization methods have been introduced that build on top of the basic idea of backpropagation. Those will be discussed later in this section.

There are a few types of layers that we will discuss. The simplest is the dense layer, where each neuron connects to all the neurons in the previous layer. We will use dense layers with $h = \max(0, x)$ as an activation function. These types of neurons are known as Rectified Linear Units (**ReLU**s). A variant is the Leaky-**ReLU** with activation $h = \max(a, x)$, where a is some small negative number. Leaky-**ReLU**s are sometimes preferred since the gradient is still affected by the presense of negative values, reducing the risk of introducing dead neurons which will never be updated in the backpropagation process.

Another type is a convolutional layer. This kind of a layer uses filters of a fixed size and dimension, e.g. $N \times N$ in the case of 2D or $N \times N \times N$ in the case of 3D. Each filter has a number of weights equal to its size and the neurons are connected in such a way that works like a filter sweeping through the layers in the input neurons, as seen in Figure 2.5. Convolutional layers work very well for dimensionally constructed data, such as finding features in pictures or in game states. Convolutional layers are inspired by research on the behaviour of the visual cortex in the brain.

Overfitting can be problematic in training neural networks. Overfitting is when the network adapts well to the training data, but does poorly on the testing and validation data, implying that the network memorized the training data rather than learning some generic features of it. Regularization methods are used to lessen the danger of overfitting. They also tend to speed up the training process, that is, become more accurate earlier.

Dropout layers are a common choice for regularizing dense layers in neural networks. The dropout layer randomly turns off neurons in the previous layer during training with

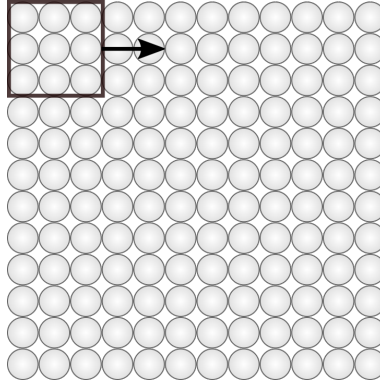


Figure 2.5: A 3×3 filter is propagated through the input which has shape 12×12 .

some probability p , often set to 0.5 for hidden layers and 0.8 for input layers. After training, the outputs of the neurons are multiplied by the probability p , resulting in a geometric mean over an exponentially larger set of networks [11]. The idea of dropout layers is inspired by the concept of ensembles of machine learning models, where the weighted average of multiple machine learning models is used as the output of a combined ensemble of models and the learned weights are adapted based on the performance of each of these models.

Convolutional layers often use either max or average pooling or batch normalization. Max and average pooling work by splitting the input into fixed rectangular chunks and either choosing the maximum or average element from those chunks as output, reducing the number of output values. Batch normalization uses mini-batches from the training data to normalize the mean and variance of the output x of the previous layer to a new output \hat{x} . The batch normalization then produces a rescaled output $\hat{y} = \gamma\hat{x} + \beta$. The parameters γ and β are learned during training. This normalization method often results in a much faster training process than without it [12].

Numerous methods exist for optimizing the objective function in neural networks, but we will mainly focus on two of the most popular ones, which are Stochastic-Gradient-Descent (**SGD**) and Adam. **SGD** is a method similar to gradient descent, except that the training data is sampled in random batches and the gradient is updated after each batch, instead of using the training data in the same order as it appears in the data set. There exist several optimization methods which build on top of **SGD**, one of which is using momentum inspired by physics which increases the learning rate when the gradient is approaching a minimum and decreases the learning rate when it is approaching a maximum (similarly to a ball rolling up or down a hill). Another method is RMSProp, which divides the learning rate by the average of the gradients used previously to update it. Finally, the optimizer Adam is often regarded as the gold standard of neural network optimizers. It is similar to RMSprop, except it uses both the average of the gradient and the average of the second moment of the gradient to scale the learning rate. The outcome is a nice tradeoff between the momentum and RMSprop methods.

Part I

Time Management

Chapter 3

Methods

Time management, the decision of how much time to spend deliberating on each move, is an important concept for both humans and computers when playing chess. Typically a fixed time is provided for the entire game, sometimes with minor time increments on each move, and the players then need to allot that time wisely throughout the game. Often opening moves are played rapidly, as well as obvious move replies (like recaptures) whereas complex middle- and endgame positions might require extra long time to evaluate and analyze adequately.

In this chapter, we experiment with automatically learning time-management for a computer chess-playing agent. In real-life settings, this may be a quite intricate problem (for humans) where many things must be taken into consideration, e.g. one's familiarity with the type of opening or middle game, as well as knowing something about the playing style of the opponent (e.g., some tend to be good in tactical positions where one needs to be extra careful, that is, think longer, before embarking in a tactical fight). Here, however, for research purposes, we use a much-simplified setup. The research question we pose is: *Can we after a fixed iteration-depth, based on the current board-state and limited search history alone, train a neural network-based classifier to determine if one needs to continue to explore the position more deeply?*

1 Deep Neural Clock

The time-management in chess programs is typically done in the iterative-deepening search procedure. One typical strategy is to estimate the number of remaining moves, say 40, and then divide the remaining time by that number. Then, the iterative deepening search proceeds until that time is up. However, before deciding to stop the search and return the best move, one looks at how stable the evaluation of the position has been in previous iterations, and if it has been fluctuating a lot — indicating that this may be difficult position to evaluate accurately — one might decide to invest in searching it slightly deeper. We use a somewhat simplified model of a time-management where we use a fixed maximum iteration-depth instead of elapsed time. The reasons for this choice were both to have a more manageable research environment and to ensure the reproducibility of our empirical results.

Our time-management iterative-deepening procedure is shown in Algorithm 2. It takes three arguments: the current chess position, the maximum iteration depth to search to, and a threshold. The threshold is a predefined parameter we use to interpret the output of our neural network classifier. The output of our neural network (function

$NN_PREDICT$ in line 8) is a real value in the range 0.0 to 1.0, the higher the value the more stable the neural network judges the position to be. If the value exceeds the threshold we judge the position to be stable. Stable positions are searched to a one less iteration depth than we would do otherwise as we do not expect the value (or best move) of such positions to change much from one iteration to next (lines 5–11). The inputs into our neural network classifier ($NN_PREDICT$) are the current chess position, and two iteration-based variables: δ_value telling how much the evaluation of the chess position changed between the last two iterations, and pv_move , a binary variable telling whether the best move (first move on the principal variation) changed between the last two iterations. The routine *search* (line 4) is the chess-program’s main search routine, returning the best move and the score of the current position based on playing that move.

Algorithm 2: Iterative Deepening with max iteration management

Input : chessPosition, max_iteration_depth, threshold
Output: bestMove

```

1 prev_iteration_value = 0
2 depth = 1
3 while depth <= max_iteration_depth do
4   (move,value) = search(chessPosition, depth)
5   if depth == max_iteration_depth - 1 then
6     delta_value = value - prev_iteration_value
7     pv_changed = move != prev_move
8     stable = NN_PREDICT(chessPosition, delta_value, pv_move)
9     if stable > threshold then
10      return move
11   end
12 end
13 prev_iteration_value = value prev_move = move
14 depth = depth + 1
15 end
16 return move

```

2 Neural Network Architectures

Using the Keras library [13], we implemented two different feed-forward (non-cyclic) neural network architectures for predicting how stable a chess position is. They both take as an input: the given chess position (the board position and castling rights), whether the best move changed between the two previous iterations, and how much the evaluation of the position (in centi-pawns, CP) changed between the same two iterations. The board position is represented differently in the two architectures. The output of both networks is a real number in the range 0.0 to 1.0 measuring the perceived stability of the position – the higher the value the more stable the neural network judges the position to be.

The first network architecture is a traditional feed-forward neural network with several dense layers, that is, each layer is fully connected to the next. The board is

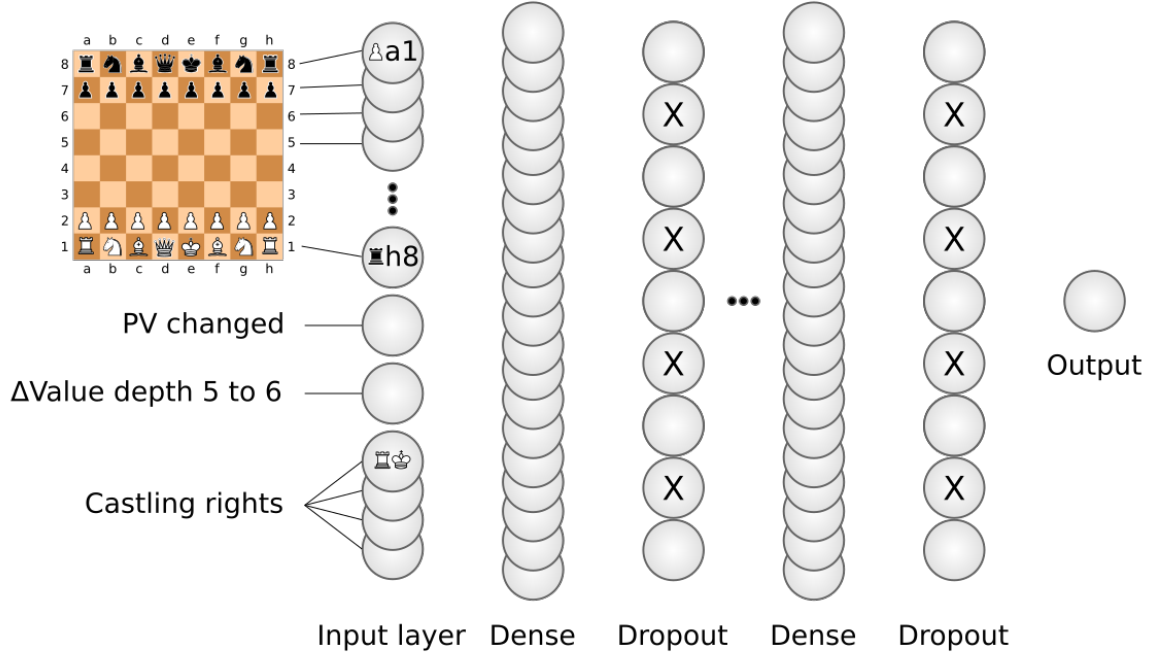


Figure 3.1: A simple full-forward architecture, where each layer is fully connected to the next. The board encoding is bundled together with castling and other parameters in the same input layer.

represented using an one-hot encoding, which results in $12 \times 64 = 768$ binary variables where each variable tells whether there is a given piece type (12 types, 6 of each color) on each of the 64 squares on the chess board. The other input parameters, *castling rights*, *pv changed*, and *delta value*, are encoded as 4 binary values (one for each possible castling), a binary variable, and a real variable, respectively. This architecture is depicted in Figure 3.1.

The second network architecture is a convolutional neural network. The board is encoded as a $12 \times 8 \times 8$ tensor, where each 8×8 matrix encodes the presence on the board for each of the 6 chess pieces of each color. A value of 1 indicates the presence of a piece of the respective type and color and 0 represents the absence of such a piece. The remaining parameters are encoded the same way as in the previous architecture. We then feed the board into a series of convolutional layers, whereas the other input parameters are fed into dense layers. The different sub-networks are then concatenated, before being fed through final dense layers. This architecture is depicted in Figure 3.2

For determining the various architectural parameter for the networks, such as the number of layers and neurons, we did a hyper-parameter search as follows:

- Use dropout layers for regularization or not. For dense layers we used a dropout layer of 50%, as this is the recommended amount by the original paper on dropout layers by Hinton [11]. For the convolutional layers we use batch normalization, as was used in the AlphaZero architecture [14].
- Number of repeated layers. We tried 1-3 repetitions of dense and convolutional layers.

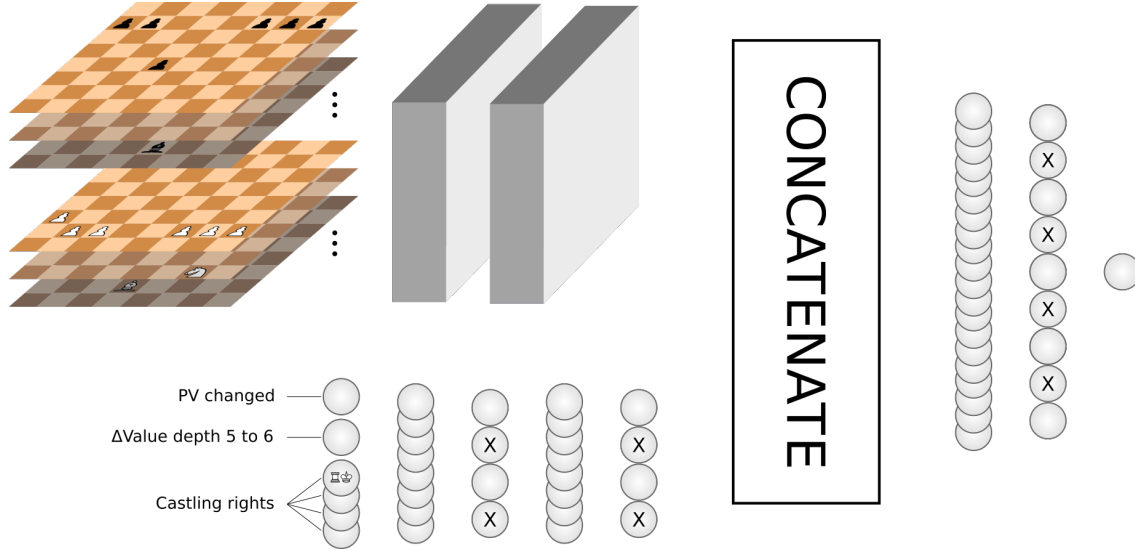


Figure 3.2: In the above architecture, the chess board goes through a series of convolutional layers, while other parameters go through dense layers. The result is then combined and goes through another series of dense layers.

- Number of neurons. We tried 50-300 neurons in dense layers and 50-100 in convolutional layers, with an increase of 50 each time.

Another parameter that was empirically determined was the *threshold* parameter (see Algorithm 2) used for interpreting the output of the network. The output is in the range 0.0 to 1.0, with higher values indicating more stable positions. However, when using the networks in the chess program we must take a binary decision as whether the position is stable enough for us to prematurely stop searching. The most simple approach would be to interpret values exceeding the threshold 0.5 as stable and the others as unstable, however, this is not necessarily the best approach. Instead we use a ROC curve for deciding the most appropriate threshold. A ROC curve is a way of computing the true positive rate and false positive rate of a classifier as a function of the threshold. We report on the result of this and the above-mentioned hyper-parameter search in the result chapter that follows.

3 Training and Testing the Networks

We generated the data for training and testing the networks as follows. We got online chess games from tournaments in a PGN format. These were two files *standard.pgn* (7,667 games) [15] and *carlsen.pgn* (1,213 games) [16]. Each position in each game was then converted to a FEN format using the python-chess library [17], resulting in total 654,555 positions.

We used the python-chess library [17] for writing a Python script to find the evaluation and best move for each of the positions for iteration depths 5, 6, and 7. This was done by calling the Stockfish chess program through the Universal Chess Interface (UCI). This information is used to label each chess position as stable or not. Formally, we define stability as:

$$stability(S, \beta, d) = \begin{cases} 1 & |V_d(S) - V_{d-1}(S)| < \beta \\ 0 & otherwise \end{cases} \quad (3.1)$$

where $V_d(\cdot)$ is the evaluation of position (state) S based on an iteration depth d . We label the target value in the data according to $\text{stability}(S, \beta, 7)$, meaning that if the CP value between depths 6 and 7 changes by an amount less than the parameter β , we consider the position to be stable but otherwise not. The difference between the CP value of depths 5 and 6 is taken as an input into the neural network.

The Python script also records all the input parameters in the right format, that is whether the best move changed between iteration 5 and 6, how much the state evaluation changed between the two iteration, and encodes the board in the right format for each network. Figure 3.3 gives details about how the chess board is encoded as a $(12 \times 8 \times 8)$ tensor, where each 8×8 matrix represents the board for each of the 6 chess pieces of each color. Also, we encode all positions from white's perspective, that is, if black is to move in a given position, the position is mapped as if the board is rotated and the color of the pieces (and castling rights) switched. We verified that the preprocessing works as intended by changing FEN positions into tensors and then back into FENs.

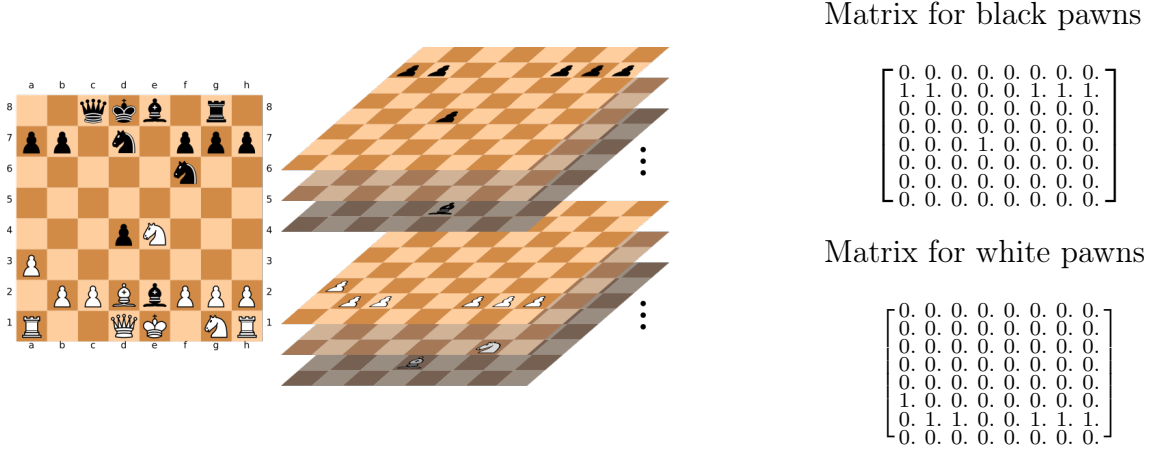


Figure 3.3: For preprocessing, we create a separate 0-1 matrix for each type of piece for each color, where 1 or 0 indicates the presense or absense of that piece in a given position on the board.

4 Evaluating the Networks

We use three methods for evaluating the effectiveness of our neural networks. First, the *prediction accuracy* on our test data (the data was split into a training and a testing set containing 80% and 20% of the records, respectively). Second, we use Cumulative Gains and ROC curves to verify how good the networks are at discriminating between stable and unstable positions. Finally, we match different versions of Stockfish against each other, where the baseline version always searches to a fixed iteration depth d and the other selectively stop at either depth $d - 1$ or d based on the predicted state stability. In the following chapter we report on those and other experiments.

Chapter 4

Results

1 Experimental Setup

The network architectures were implemented in Keras [13] using Tensorflow [18] as a backend. We used Tensorboard, a logging and visualization tool in Tensorflow, to retrieve csv-files with the data presented in the graphs below. The hyper-parameter search for determining the most suitable architectural parameters was done on the node-cluster Garpur [19]. All competitions between different versions of Stockfish were done on a single computer with a 2,5 GHz Intel Core i7 processor and 16GB RAM.

2 The Dense Network Architecture

We first experimented with using different number of nodes in each layer, and whether to use dropout layers for regularization or not. We fixed the dropout rate to 0.5 (50%) as recommended in [11]. During those experiments the remaining architectural parameters, number of layers and β are kept fixed to 2 and 18.0, respectively. The result is depicted in Figure 4.1. By contrasting the left and right graphs we see that with the number of training epochs (steps) the accuracy on the training data continues to increase whereas it saturates quickly on the validation data, independent from whether we use dropout (top graphs) or not (bottom graphs). This is a clear sign of overfitting and it becomes increasingly profound as the number of neurons in each layer increases. The dropout regularization seem to somewhat alleviate the overfitting on the training data, as expected, although it does unfortunately not translate into noticeable better accuracy on the validation data. There seems to be no advantage in using more than 50 neurons in each layer, and little to no advantage in using dropout layers. Consequently, in further experiments with the dense architecture, for the sake of simplicity we fix each layer to 50 neurons and do not use dropout layers.

In the next set of hyper-parameter experiments we change the number of layers to be either 2, 3, or 4. During those experiments the remaining architectural parameters, number of nodes in each layer, use of dropout, and β are kept fixed to 50, False, and 18.0, respectively. The result is depicted in Figure 4.2. The performance of the neural network is just about the same whether we use 2, 3 or 4 layers. We will go with the simplest architecture, as before, and choose to use 2 layers in our final architecture.

Finally, we tune the β parameter which represents the stability boundary as described in Equation 3.1. The value of β is given in centipawns, that is, a value of a

Accuracy when tuning neurons for simple architecture

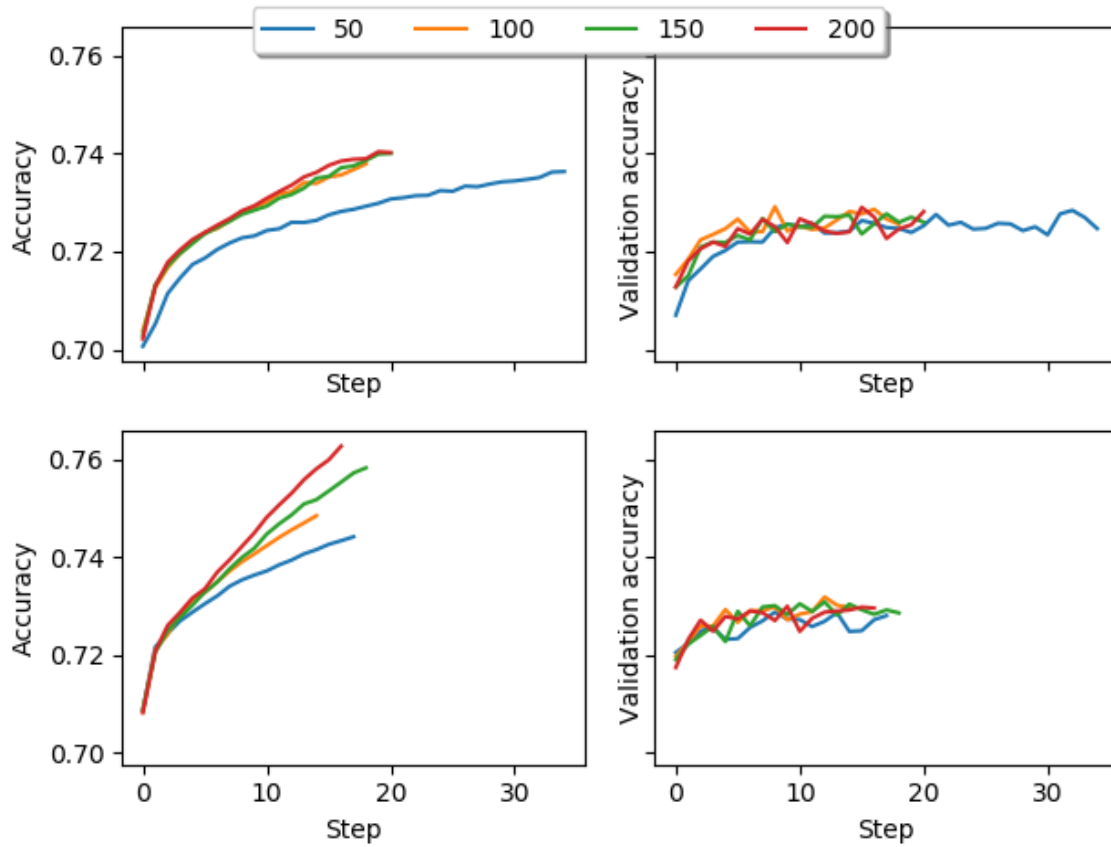


Figure 4.1: Accuracy on the training data vs validation accuracy from tuning the number of neurons, while keeping layers fixed at 2 and $\beta = 18.0$. In the top graphs we use dropout layers set at 0.5 for regularization and in the bottom graphs we have no such regularization.

Accuracy when tuning layers for simple architecture

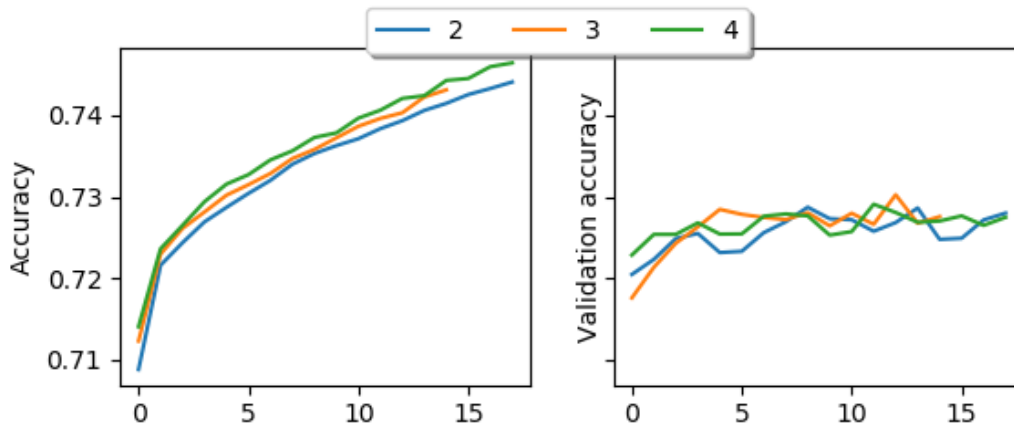


Figure 4.2: Accuracy on the training data vs validation accuracy from tuning the number of layers, while keeping neurons fixed at 50 and $\beta = 18.0$.

pawn is fixed to 100. A value of $\beta = 18$ thus correspond to slightly less than one fifth of a pawn. During those experiments the remaining architectural parameters, number

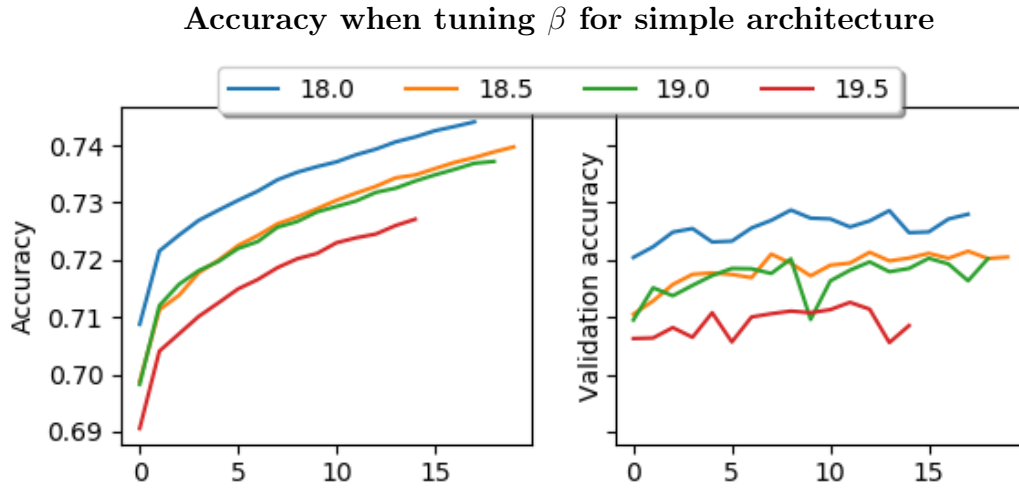


Figure 4.3: Accuracy on the training data vs validation accuracy from tuning the beta parameter, while keeping neurons fixed at 50 and layers fixed at 2.

of nodes in each layer, use of dropout, and number of layers are kept fixed to 50, False, and 2, respectively. The results are shown in Figure 4.3. The value of $\beta = 18.0$ seems to be the best of the values we experimented with. It is worth noting that lower levels of β than 18.0 were also tested, and while they generally gave a slightly better accuracy, they resulted in a significant decrease in the number of positions labelled as stable (see Section 3 on the role of β is determining stable positions). We decided thus to fix $\beta = 18$, as a compromise.

3 The Convolutional Network Architecture

We did a similar hyper-parameter tuning for the convolutional network architecture. As dropout plays a much lesser role for regularization in this architecture than the previous one, we choose not to use dropout and did thus not test for that parameter.

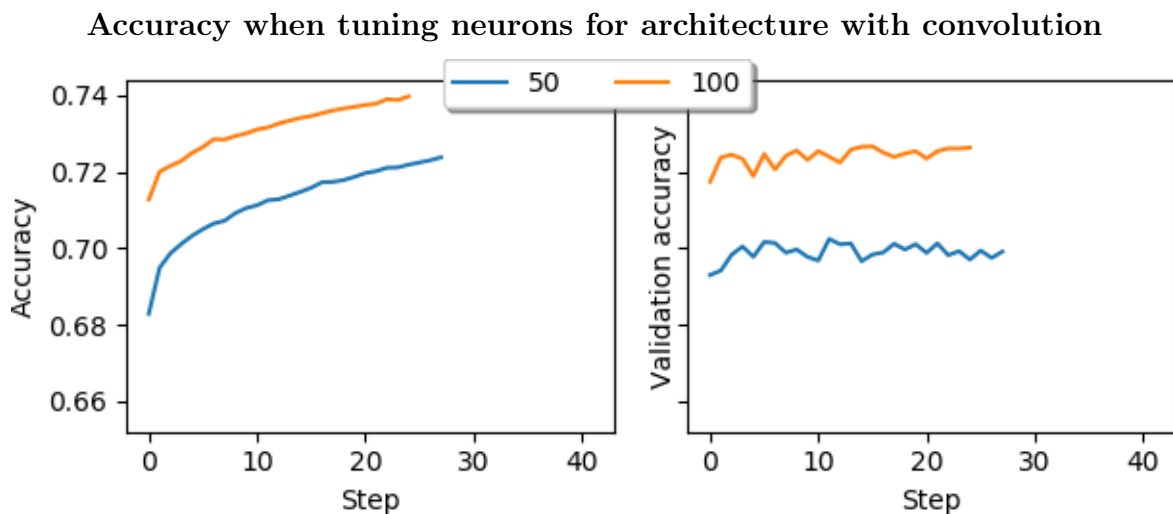


Figure 4.4: Accuracy on the training data vs validation accuracy from tuning the number of neurons, while keeping layers fixed at 2 and $\beta = 18.0$.

Figure 4.4 shows the result of using different number of neurons in each layer (we tested only 50 and 100 neurons per layer). Here there is a clear advantage in using more than 50 neurons per layer.

Accuracy when tuning layers for architecture with convolution

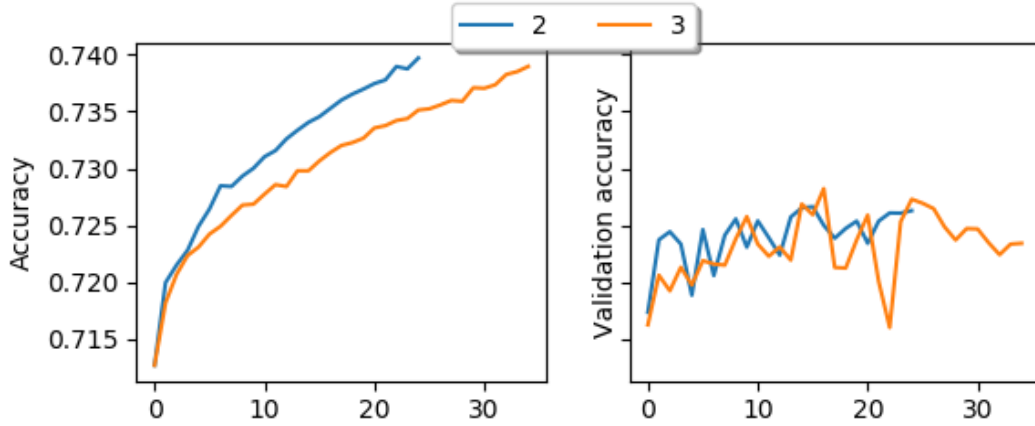


Figure 4.5: Accuracy on the training data vs validation accuracy from tuning the number of layers, while keeping neurons fixed at 100 and $\beta = 18.0$.

Based on the results in Figure 4.5 we chose to use only 2 layers, that is, for both the convolutional and non-convolutional sub-networks before they are concatenated. Having 3 layers seems to just train slower with no obvious increase in accuracy. Similarly, based on the experiments shown in Figure 4.6 we determine the most suitable value of β to be 18, as before.

Accuracy when tuning β for architecture with convolution

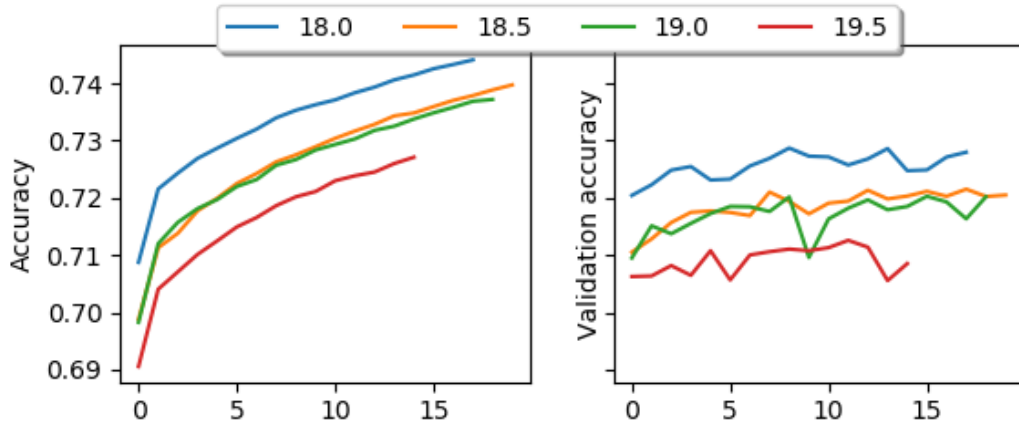


Figure 4.6: Accuracy on the training data vs validation accuracy from tuning the β parameter, while keeping neurons fixed at 100 and layers fixed at 2.

4 Comparing the Architectures

The most appropriate hyper-parameters determined for the two architectures only differed in the number of neurons in each layer, that is, 50 and 100 neurons for the dense and convolutional architectures, respectively. The other hyper-parameters were the

same for both architectures, that is, no dropout, 2 layers, and $\beta = 18$. When contrasting the accuracy achieved by the two architecture there is no difference, they both lie between 72-73% percent validation accuracy. After choosing the best parameters, we evaluate the accuracy on the test dataset, which yielded the same results. As before, we chose the simpler model in such situations. All further experiments were thus done using only the dense architecture.

5 Distinguishing Stable vs. Nonstable Positions

Accuracy is only one possible metric to measure prediction effectiveness. It tells the percentage of positions that the network correctly identifies as stable or not. The accuracy our classifier receives is between 72-73% and although better than an uniform random guess, it would have less accuracy than a classifier that would always predict that a position is unstable (the percentage of unstable positions in our $\beta = 18$ dataset is approximately 77%). However, such a classifier would be totally useless for time management as it would never stop the search prematurely. For our task, it is important that we are able to distinguish between stable and non-stable positions. We thus also looked at other metrics that are possibly more meaningful than accuracy for our task, *precision* and *recall*.

Cumulative Gains and ROC curves allows us to have a holistic look at precision and recall at the same time, as depicted in Figure 4.7. The graph to the left shows the Cumulative Gains curve. The test data is ordered in a decreasing order by prediction of the dense network along the x -axis, whereas the y -axis shows the percentage of stable positions correctly identified so far. As you can see on the Cumulative Gains curve, by looking at only 20% of the data we already identify almost 40% of the stable positions, where a random baseline model would only get 20%. The recall of our model is thus substantially better than random, however, it still has a long way to go compared to a perfect model, which would have identified around 70% of all stable positions by that point. The ROC curve to the right provides an alternative view where the false-positive-rate (FPR), on the x -axis, and the true-positive-rate (TPR), on the y -axis, are contrasted as the discriminating threshold of the classifier is varied. The area under the curve for our classifier is 0.71 for predicting both stable and unstable positions, which again is substantially better than random.

We can also use the ROC curve to determine an appropriate value for the *threshold* parameter used in Algorithm 2. Essentially, we want to choose a value for the parameter such that it has a very low FPR, while still having a relatively high TPR when predicting whether a position is stable. We can achieve a low FPR by choosing a somewhat high threshold, but if too high the networks's prediction as used in our algorithm would only kick in too infrequently and thus any measurable effect.

Table 4.1 shows a zoomed-in view of the lower-left corner of the ROC curve. Based on the table we chose the threshold to be 0.76, which identifies slightly over 6% of the stable positions while only being wrong in 0.5% of the cases. This seems like an acceptable tradeoff between **FPR** and **TPR** because the consequences of unnecessarily searching a stable positions deep are potentially much less severe than not searching an unstable position to a full depth. This is because the former mistake results only in spending too much time on a given position, while the other problem could result in a suboptimal move or even a game-losing blunder.

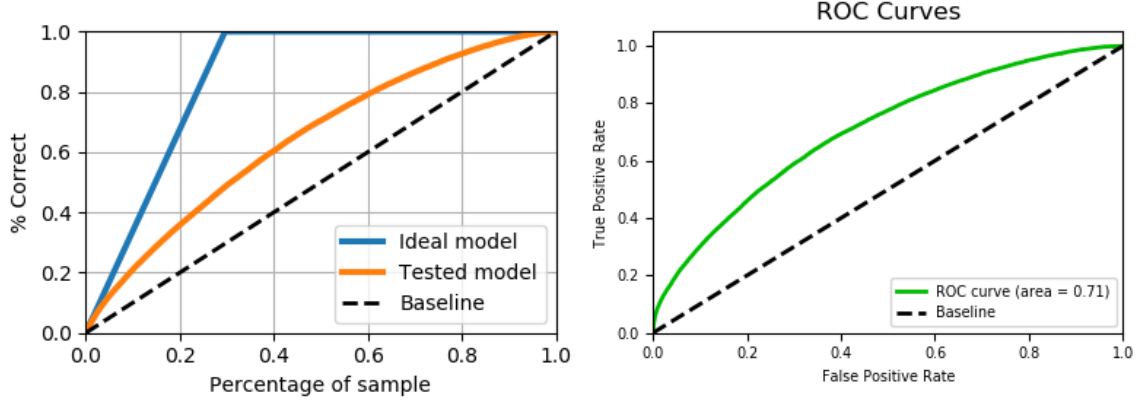


Figure 4.7: On the left side we have a Cumulative Gains curve, which shows how well our model does at signifying correctly that Stockfish should stop searching. The blue line shows an ideal model which always outputs correctly and the baseline shows how well a random model would do on average. Our model is on the orange line and clearly has a significant advantage over a random output. On the right side we have the ROC curve, which plots the false positive rate against the true positive rate. Our area under the curve is 0.71, while an ideal area would be 1.0.

Table 4.1: Table data for the ROC curve for stability, sorted by **FPR**.

FPR	TPR	Threshold
0.1%	2.11%	0.93
0.2%	3.25%	0.875
0.3%	4.28%	0.829
0.4%	5.18%	0.789
0.5%	6.02%	0.762
0.6%	6.52%	0.738
0.7%	7.09%	0.713
0.8%	7.73%	0.692
0.9%	8.21%	0.674
1.0%	8.69%	0.658
1.1%	9.06%	0.644
1.2%	9.44%	0.633
1.3%	9.72%	0.624
1.4%	10.15%	0.613
1.5%	10.48%	0.605
1.6%	10.86%	0.597
1.7%	11.27%	0.589
1.8%	11.58%	0.582
1.9%	11.92%	0.576

6 Testing in Gameplay

Having chosen the architecture, hyper-parameters, and the discrimination *threshold* for our neural network, we have all the information needed to test Algorithm 2 in practice.

We matched a version of Stockfish using a fixed iteration-depth of 7 against two versions of Stockfish that selectively choose between either depth 6 and 7. The former version uses the prediction of stableness of our feed-forward neural network to determine if to stop searching after iteration-depth 6 (the network has 2 layers of 50 neurons each, trained without dropout layers, and using training/testing data labelled using $\beta = 18$). The latter version, used as a baseline, randomly chooses between iteration-depth 6 or 7 with the same frequency as the former version.

The full-iteration-depth version played a match consisting of over 1000 games against each of the two selective iteration-depth versions. The result is best summarized as both the selective versions did slightly worse against the full-depth player, as expected, however, somewhat dissapointingly, there was not a statistically meaningful difference between the performance of the two selective versions.

7 Summary

We conclude that although our neural-network classifier is able to discriminate between stable and unstable positions substantially better than random (with respect to recall), it is nonetheless not accurate enough for that benefit to transfer into actual game play.

Part II

Depth Reduction

Chapter 5

Methods

The Stockfish chess program uses a traditional mini-max-based game-tree search for planning. Additionally, it employs numerous state-of-the-art algorithmic search enhancements, such as iterative-deepening [20], alpha-beta cutoffs [6], Negascout/PVS [8] [9], transposition table, singular-extensions [21], move-ordering, and more. In particular, it uses a quite aggressive forward-pruning mechanism, which is based primarily on null-move-search [22] and late-move-reductions [1]. Instead of searching all branches in the search tree to full depth, forward-pruning may choose to terminate some branches prematurely, that is, search them to less depth if they are deemed unlikely to yield a better result. In this part of the project, we experiment with tuning the main set of depth-reduction parameters used for the late-move-reduction forward-pruning scheme in Stockfish.

1 Late-Move-Reductions in Stockfish

Algorithm 3 gives a high-level overview of how late-move-reductions are performed in Stockfish. We only show the core functionality of the scheme, with just enough details to clarify the following discussion. We refer interested readers to the code-base for further details.

The algorithm shows a recursive alpha-beta-based search (a variant named Negascout, to be more specific). Most of the code is standard for chess game-playing programs, for example, terminating the recursions with quiescence search (lines 1-4), initializing and looping through available moves (the while loop in lines 7-36), making and unmaking moves (lines, 12 and 25, respectively), updating bounds and best value found so far (lines 26-35). The code specific to late-move-reduction is in lines 13-20. Essentially, if the current position is deemed suitable for a reduced search (e.g., a quiet move which is not the first move in the movelist), a search with a reduced search depth is first tried, and only if it fails-high, is the move searched again, but now to a full depth.

At each internal node, the movelist is ordered such that perceived promising moves are searched before the less promising ones. The later a move is in the movelist, the less likely it is to yield an improvement, and under certain non-threat conditions, it may be searched to less than the original intended depth. The depth-reduction equation used in the standard version of Stockfish is a function of both the remaining search depth and the location of the move in the movelist, and grows logarithmically and symmetrically with its two arguments, as shown below (Equation 2.1 replicated here

Algorithm 3: Overview of search code

```

Input : pos, alpha, beta
1 if depth < 1 then
2   // do quiescence-search and evaluate pos
3   return qsearch(pos, alpha, beta)
4 end
5 best_value = -VALUE_INFINITE
6 move_count = 0;
7 while move = next_move(pos) do
8   move_count += 1 // keep track of number of moves
9   // Apply search extensions if applicable (e.g. in check)
10  new_depth = depth - 1 + extension
11  // Search recursively, either shallow-depth, full-depth, or both.
12  do_move( move )
13  if do_reduced_search(pos, new_depth, alpha, beta, move_count) then
14    Depth r = reduction(depth, move_count)
15    value = -search(pos, -(alpha+1), -alpha, new_depth - r)
16    if value > alpha then
17      // fail-high, do full-depth search.
18      value = -search(pos, -(alpha+1), -alpha, new_depth)
19    end
20  end
21  else
22    // do only full-depth search
23    value = -search(pos, -(alpha+1), -alpha, new_depth)
24  end
25  undo_move( move )
26  // Record best value (move) found so-far.
27  if value > best_value then
28    best_value = value
29    if value > alpha then
30      alpha = value
31      if value >= beta then
32        break
33      end
34    end
35  end
36 end
37 return best_value

```

from Section 2 for convenience):

$$r = \left\lfloor \frac{1}{1.95} \log(d) \log(m) \right\rfloor \quad (5.1)$$

Figure 5.1 depicts the reduction using a heat-map.

Our hypothesis is that a more sophisticated and, in particular, non-symmetrical (w.r.t. depth and move locations) depth-reduction scheme might potentially be more

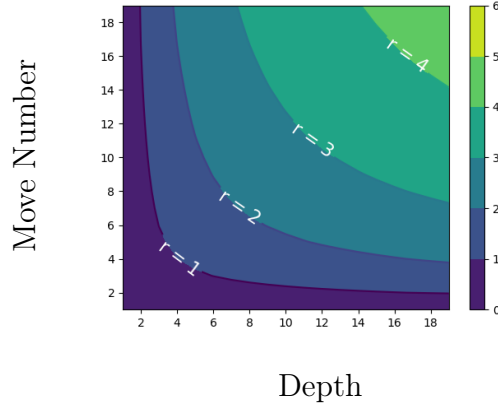


Figure 5.1: Visual map of depth reduction using equation 5.1.

effective, that is, an equation taking the general form:

$$r = \lfloor A \log(d) \log(m) + B \log(d) + C \log(m) + D |\alpha - s| + Em \rfloor \quad (5.2)$$

where A, B, C, D, E are constants, $|\alpha - s|$ is the absolute difference between the α value and the static evaluation at root and m is the material difference between the two players (black and white). The most appropriate constants in the above equation can be determined using a regression model. In the next subsection we describe the process for gathering the training data for the regression.

2 Gathering and Labelling the Data

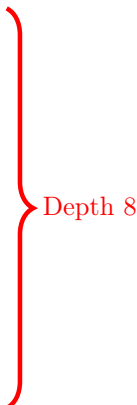
To look for a more effective depth-reduction mechanism than Equation 5.1, we need to investigate the relationship between the game state and allowable reduction in search depth. If we gather enough data about game states and label them with the allowable search reduction, we can use regression methods on that data to parameterize a given candidate search reduction formula. We can then run a match to test how well the new candidate reduction formula performs compared to the original.

We start by running a match between two instances of Stockfish and periodically log out the following variables: FEN notation, search depth, move number, α and material value for white and black. To make sure that the data would include different games, we created a set of 696 opening positions. These positions were made from the position after 6 moves from two different databases of games, one from a set of games played by Magnus Carlsen and one from a set of games from online players. See how these opening positions were created in the appendix at A.1.

After gathering data from different search states we programmatically analyze the search outcome at maximum depth for each unique FEN in the data and calculate the maximum possible depth reduction before the search outcome changes from a fail-low to a fail-high. See table 5.1 for an illustration. We only considered states where the result of the full search agrees with the reduced search depth and we only focused on cases where the search resulted in a fail low. The code used for labelling can be seen in the appendix at A.3.

Table 5.1: The upper limit of allowable reduction in search depth is $11 - 3 = 8$, because the search outcome does not change from depth 3 to depth 11.

depth	value (CP)	≤ 100
1	99.7	True (Fail-Low)
2	109.1	False (Fail-High)
3	99.9	True (Fail-Low)
...
8	99.4	True (Fail-Low)
9	99.3	True (Fail-Low)
10	97.2	True (Fail-Low)
11	96.5	True (Fail-Low)



3 Regression and Normalizing

After labelling we run linear regression to determine the best parameters for Equation 5.2. As some of the parameters seemed to be of a little consequence, we also fitted various other more simplified forms of the equation (discussed later).

The depth-reductions resulting from the regression model are far more aggressive than the original depth-reductions used by Stockfish. This is unsurprising, and simply an artifact of how we gather the training data, where we compute the most appropriate *intermediate reduction* for each depth/move locations. In practice, this implies that sometimes we reduce too much and sometimes too little. However, the consequences of reducing the depth too aggressively are much more severe and can lead to an immediate loss (as opposed to only some unnecessary search overhead). What we want to do is to renormalize the depth-reductions such that the total number of nodes searched under the new scheme is approximately the same as in the old. The search trees explored will though have radically different shape.

To ensure that the newly found formula doesn't reduce the depth too aggressively, we use the built-in benchmarking program in Stockfish which counts the number of nodes searched when evaluating a certain set of positions. The total sum of nodes searched is used internally by Stockfish to decide whether it is worth it at all to consider running your changes inside FishTest. We use this benchmarking tool as a way of normalizing our equation. We find a scaling factor for our formula which gives a similar total number of nodes searched in the benchmark as in the unmodified version of Stockfish.

In order to ensure that we are not comparing apples to oranges, we made sure to always use the same Stockfish executable (just configured to use a different reduction equation at runtime) when we run the benchmarking tool. This ensures that any dif-

ferences seen in the number of nodes searched happens only because of our choice of equation and not due to differences in optimization levels in the compiler or other code changes. We made simple modifications to Stockfish which allow us to modify the reduction equation at runtime, rather than recompiling the program. The modifications can be seen in the appendix at A.4.

4 Matches against the original Stockfish

Finally, we ran matches where we match the original Stockfish chess program against modified counterparts, each using a search-reduction equation from the different regression models. The overall match performance is the ultimate test of how good each regression model is in practice.

Chapter 6

Results

In this chapter we show the results from using the regression-based approach to tuning the depth-reduction function in Stockfish. We used the cute-chess [23] program for managing all the matches.

1 Gathering and Labelling the Data

In order to obtain log records, we ran a match between Stockfish and itself with time controls of 3 min and 2 seconds (increment), logging down statistics in some of the nodes during LMR search. We logged only a small fraction of the nodes searched and mostly at depths in the range 1-10 and move number in the same range (the exact conditions can be seen in the function `do_log` in A.2 in the Appendix). We used only the logs from the search which resulted in a fail-low (around 82% of the logs), since this is the expected outcome in LMR search.

We labelled roughly 100,000 log records from 100 different games before applying linear regression. The maximum depth reduction as described in table 5.1 was on average 3.37, while the full depth which would have been searched through was on average 4.55.

2 Regression and Normalizing

Table 6.1 shows the results of the regression. The score on the right is calculated using the coefficient of determination, R^2 , and the simplest formula with the best fitting score is chosen. Note that even though two formulas have an R^2 score of roughly 0.8, the latter one has two extra terms which do not seem to contribute significantly to the score. As a result, the following equation was chosen:

$$2.3988\log(d) * \log(m) + 2.12698 * \log(d) - 4.64504 * \log(m) \quad (6.1)$$

Other parameters that were tested with regression, such as m and $|\alpha - S|$, were dropped since they apparently do not have noteworthy relations with the depth reduction (as one can see in Table 6.1, those two terms are always given a weight of zero).

As noted in the previous chapter, the depth-reductions resulting from the regression model are far more aggressive than the original depth-reductions used by Stockfish (an artifact of how we gather the training data). We thus renormalize the depth-reduction such that the total number of nodes search under the regression result based scheme

Formula	R^2 Score
$1.0915 * \log(d) * \log(m)$	0.3588
$2.0351 * \log(d)$	0.4157
$1.8865 * \log(m)$	0.0105
$0.000110438 * \alpha - S $	-1.6759
$0.000981318 * md$	-1.7541
$0.3761 * \log(d) * \log(m) + 1.3697\log(d)$	0.4352
$3.1016\log(d) * \log(m) - 3.8611\log(m)$	0.6274
$2.3988\log(d) * \log(m) + 2.1270 * \log(d) - 4.6450 * \log(m)$	0.8005
$1.0867 * \log_d * \log_m + 8.4988e - 06 \alpha - S $	0.3595
$1.0897 * \log_d * \log_m + 7.0620e - 05md$	0.3591
$1.0856 * \log_d * \log_m + 7.9740e - 06 \alpha - S + 5.6869e - 05md$	0.3597
$0.0001 \alpha - S + 0.0008md$	-1.6441
$2.39\log(d)\log(m) + 2.14\log(d) - 4.64\log(m) - 3.78e-6 \alpha - S - 2.5e-5md$	0.8007

Table 6.1: Regression attempts using different terms in the equation. Note that some of the R^2 scores are negative, implying that the formula found through regression is a worse fit through the data than a horizontal line.

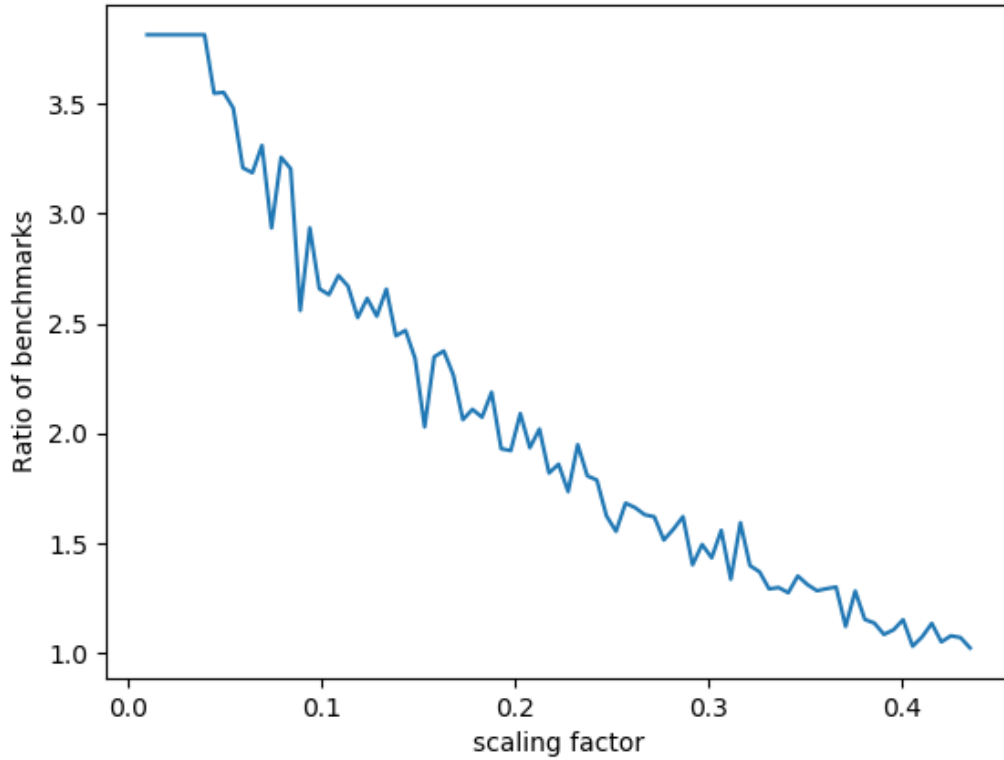


Figure 6.1: The difference in number of nodes searched between Stockfish using the modified equation and the original Stockfish is shown on the y-axis. The scale factor used on the equation found through regression is on the x-axis. The smaller the scaling factor, the lesser the reduction parameter will be and thus Stockfish will search through a greater amount of nodes.

is approximately the same as in the original one. Although the total node count will be approximately the same, the search trees explored will have radically different shape.

One can see the ratio between the number of nodes searched when using Equations 5.1 and 6.1 in Figure 6.1. One sees that the ratio starts approaching 1 as the scaling factor increases. We ended up choose a scaling factor of 0.40596, which results in the following equation (after scaling):

$$r = \lfloor 0.9738 \log(d) * \log(m) + 0.8635 \log(d) - 1.8857 \log(m) \rfloor \quad (6.2)$$

The number of nodes searched in the benchmark for this equation was 3,756,481 while the original gives 3,789,679, which is only 1.0% higher. Using this new equation we also got the same best move as with the original equation only 76% of the time, indicating that the search is radically different although the total search effort is similar. Also, in Figure 6.2, the depth-reduction is depicted as a heat-map, and differ significantly from the original one (see Figure 5.1). For example, the new equation seems to put little emphasis on the move number until one reaches a depth of 5.

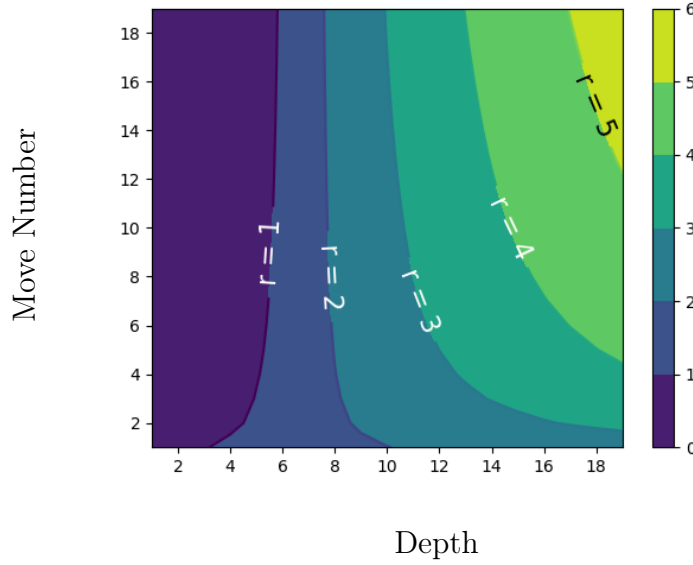


Figure 6.2: Visual map of depth reduction using the new equation.

3 Match Results

In the below tables, A , B and C refer to the constants in Equation 5.2. For match results, a score like 686.5/705.5 means that the sum of wins and draws for the original Stockfish was 705.5 and 686.5 for the modified Stockfish (a win adds 1 to the sum and a draw adds 0.5). TC stands for time control.

The following matches were run on Garpur, except for one experiment which was run on FishTest. The number of wins and losses are shown along with win percentage.

A	B	C	W/L with TC $10s + 0.01s$	W/L with TC $3m + 2s$
0.9738	0.8635	-1.8857	637.5/754.5 (45.8%)	39.5/53.5 (42.5%)

4 Other Experiments

Finally, we did a few other experiments with the reduction equation. Some of the more interesting ones are depicted below.

4.1 Hand-Picked Parameters

We tried a few parameterizations of equation 5.2 by hand based on insight gained from reading into the heatmaps and previous game results. We ran one of the better ones found through that method on FishTest. We first ran a match using a regression formula with the following parameters on the Garpur cluster:

A	B	C	W/L with TC $10s + 0.01s$	W/L with TC $3m + 2s$
0.1	0.5	0	654.0/738.0 (47%)	56.0/58.0 (49%)

Since this formula gave a promising win ratio with a longer time control we decided to try it on FishTest and got these results: [24]

Played on	Wins/Losses with TC $10s + 0.1s$	p-value
25 computers	1339.5 / 1491.5 (47.3 %)	92.64%

Note that FishTest only used a short time control, while our own tests used both a long and short time control. The reason for this is that FishTest finishes early if the p-value reaches a certain threshold and doesn't move on to the next stage.

4.2 Grid Search

We ran a grid based parameter search using the first three terms of equation 5.2 using the above mentioned Stockfish benchmark tool as an evaluation function. We ran matches against the original Stockfish using the best parameters found in the grid search. Below are the match results based on those parameters.

A	B	C	Wins/Losses with TC $10s + 0.01s$	Wins/Losses with TC $3m + 2s$
$0.2\bar{3}$	0.1	$0.4\bar{5}$	489.5/902.5 (35%)	92.5/140.5 (40%)
$0.3\bar{2}$	$0.1\bar{8}$	0.1	680.0/712.0 (48.9%)	110.5/121.5 (47.6%)
$0.3\bar{2}$	$0.2\bar{7}$	$0.1\bar{4}$	686.5/705.5 (49.3%)	106.0/125.0 (45.8%)
$0.2\bar{3}$	$0.3\bar{2}$	$0.1\bar{4}$	671.5/720.5 48.2%	113.5/114.5 (49.8%)
$0.1\bar{4}$	0.5	$0.1\bar{8}$	672.5/719.5 (48.3%)	113.5/118.5 (48.9%)
$0.2\bar{7}$	$0.1\bar{4}$	$0.2\bar{7}$	687.5/704.5 (49.4%)	107.0/119.0 (47.3%)

4.3 Non-linear Modifications

Based on the results from the previous experiments we also ran a match using a non-linear modification of the search reduction formula:

$$r = \left\lfloor \frac{1}{1.95} \log(2d) \log(m/2) \right\rfloor \quad (6.3)$$

The difference between equation 6.3 and 5.1 is that the depth and move number have been scaled differently. We modified Stockfish in a similar way as before to support this new version of the search reduction formula, as can be seen in A.5.

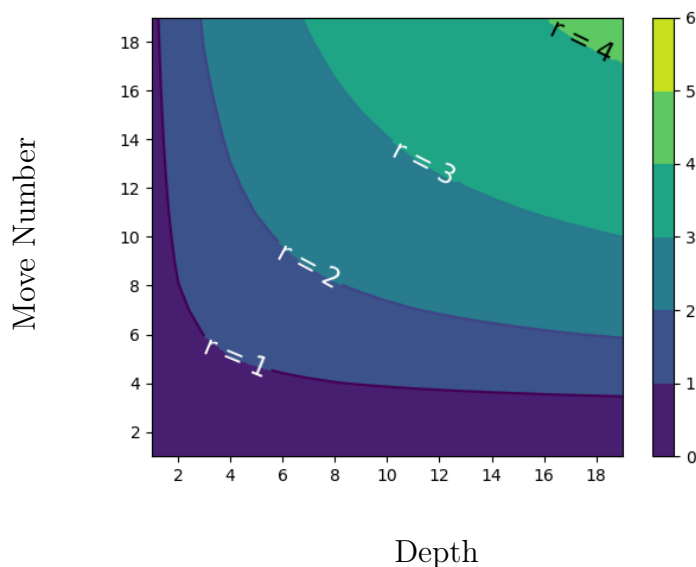


Figure 6.3: Visual map of depth reduction using equation 6.3

Match results using this nonlinear modification can be seen below.. A visual map of this equation can be seen in figure 6.3. The performance of this equation seems quite similar to the original.

Wins/Losses with TC $10s + 0.01s$	Wins/Losses with TC $3m + 2s$
698.5/693.50 (50%)	41.0/41.0 (50%)

5 Summary

To summarize, we made modifications to the depth reduction method in Stockfish. We found alternative formulas to the one used by Stockfish using various methods, most notably linear regression with normalization based on benchmarking. Despite our efforts, we did not manage to improve the performance of Stockfish with these alternative depth reduction formulas. This is maybe not surprising, given that the current search-reduction parameters in Stockfish are based on careful tuning and extensive gameplay evaluation.

Part III

Conclusion and Future Work

Chapter 7

Conclusions and Future Work

The time-management experiments did show some promise in discriminating between stable vs. unstable chess positions, although ultimately it did not help with time management. There are other potential uses in the chess program for such a classification, for example in the quiescence search, although we did not experiment with them.

As for the depth reductions, despite our efforts, we did not manage to improve the search in the Stockfish chess engine. The Stockfish engine is more than a decade old and includes a lot of tuneable parameters that have been modified and optimized, mainly though extensive tournament play. FishTest runs dozens of tournaments on hundreds of computers every single day, but most of the improvements tested in these tournaments will not make it into the engine. Thus it is in itself not surprising that we did not manage to improve the engine.

As for future work, we believe that there is still some potential scope for improving Stockfish through machine learning. For example, in retrospect, we suspect that our data-gathering mechanism might not have resulted in the ideal ground-truth data. Similarly, we believe that tuning β as an hyper-parameter with respect to accuracy was not the best suited approach for data labeling. Instead it might have been better to manually determine β using domain (chess expert) knowledge. For example, given better such data one might for example try more sophisticated regression methods, e.g. non-linear ones. Another possibility would be to avoid altogether having to generate labeled ground-truth data, and instead use reinforcement learning in an unsupervised fashion. However, such an approach would likely require quite extensive computing resources.

Bibliography

- [1] *Stockfish homepage*, <https://stockfishchess.org/>.
- [2] O. E. David, N. S. Netanyahu, and L. Wolf, “Deepchess: End-to-end deep neural network for automatic learning in chess”, in *International Conference on Artificial Neural Networks*, Springer, 2016, pp. 88–96.
- [3] M. Lai, “Giraffe: Using deep reinforcement learning to play chess”, *arXiv preprint arXiv:1509.01549*, 2015.
- [4] D. Silver, et al ..., and D. Hassabis, “Mastering chess and shogi by self-play with a general reinforcement learning algorithm”, *arXiv*, Dec. 2017.
- [5] *AlphaZero chess: Reactions from top gms*, *stockfish author*, <https://www.chess.com/news/view/alphazero-reactions-from-top-gms-stockfish-author>.
- [6] D. Knuth and R. Moore, “An analysis of alpha-beta pruning”, *Artificial Intelligence*, vol. 6, pp. 293–326, 1975.
- [7] *Tic-tac-toe game tree image from wikipedia*, https://en.wikipedia.org/wiki/Game_tree#/media/File:Tic-tac-toe-game-tree.svg, License CC BY-SA 3.0.
- [8] A. Reinefeld, “An improvement of the scout tree search algorithm”, *ICCA Journal*, vol. 6, no. 4, pp. 4–14, 1983.
- [9] T. A. Marsland, “Relative efficiency of alpha-beta implementations”, A. Bundy, Ed., Karlsruhe, FRG: William Kaufmann, Aug. 1983, pp. 763–766, ISBN: 0-86576-064-0.
- [10] B. C. Csáji, “Approximation with artificial neural networks”, *Faculty of Sciences, Eötvös Loránd University, Hungary*, vol. 24, p. 48, 2001.
- [11] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov, “Improving neural networks by preventing co-adaptation of feature detectors”, *arXiv preprint arXiv:1207.0580*, 2012.
- [12] S. Ioffe and C. Szegedy, “Batch normalization: Accelerating deep network training by reducing internal covariate shift”, *arXiv preprint arXiv:1502.03167*, 2015.
- [13] *Keras homepage*, <https://keras.io/>.
- [14] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, *et al.*, “Mastering chess and shogi by self-play with a general reinforcement learning algorithm”, *arXiv preprint arXiv:1712.01815*, 2017.
- [15] *Site where you can download pgn files with games from the fics chess game server*, <https://www.ficsgames.org/download.html>.

- [16] *Site where you can download pgn files with games by various famous chess players*, <https://www.pgnmentor.com/files.html#players>.
- [17] *python-chess library*, <https://github.com/niklasf/python-chess>.
- [18] *Tensorflow homepage*, <https://www.tensorflow.org/>.
- [19] *Garpur homepage*, <https://ihpc.is/garpur/>.
- [20] D. J. Slate and L. R. Atkin, “CHESS 4.5—The Northwestern University chess program”, in *Chess Skill in Man and Machine*, P. W. Frey, Ed., Berlin: Springer-Verlag, 1977, pp. 82–118.
- [21] T. Anantharaman, M. S. Campbell, and F.-h. Hsu, “Singular extensions: Adding selectivity to brute-force searching”, *Artificial Intelligence*, vol. 43, no. 1, pp. 99–109, 1990.
- [22] D. Beal, “Experiments with the null move”, in *Advances in Computer Chess 5*, D. Beal, Ed., 1989, pp. 65–79.
- [23] *Cute-Chess homepage*, <https://github.com/cutechess/cutechess>.
- [24] *Fish Test results*, <https://web.archive.org/web/20191019104856/http://tests.stockfishchess.org/tests/view/5d148c7a0ebc5925cf0b4255>, Ran on: 2019-06-27.

Appendix A

Code

1 Logging code

1.1 Opening positions

The below python script was used to create a set of unique opening positions. Using this increases the chance that the chess games will be unique.

Listing A.1: Create a set of opening positions

```
1 import chess.pgn

3 files = [open('./datasets/Carlsen.pgn','r'), open('./datasets/AepliBase.pgn','r')]

5 openingFens = set()

7 for f in files:
    games = 0
    game = chess.pgn.read_game(f)
    while(game and games < 10000):
11         board = chess.Board()
            m = 0
13         for move in game.mainline_moves():
            board.push(move)
15             m += 1
            if m == 6:
17                 break

19         fen = board.fen().split(' ')
            openingFens.add(' '.join(fen[0:4] + ['0_0']))

21         game = chess.pgn.read_game(f)
            games += 1

23 for fen in openingFens:
    print(f'[Event_ "Dummy_Data"]')
27    print(f'[Site_ "Reykjavik, Iceland"]')
    print(f'[Date_ "??"]')
```

```

29  print(f'[White:_"White,_Mx."']')
    print(f'[Black:_"Black,_Mx."']')
31  print(f'[Result:_*]')
    print(f'[SetUp_1]')
33  print(f'[FEN_"{fen}"]')
    print()

```

1.2 Logger

The below code was used when logging out state variables in StockFish.

Listing A.2: Search logging

```

1  #include <fstream>
2  #include <string>

4  class Record {
    public:
6      Record(const std::string &fen,
              bool pv, bool improving, int d, int m,
8              int nd, int r_org, int r, int ds, int alpha, int value, int ply,
              int gply, int npw, int npb, int np, int seval)
10         : fen_(fen),
            pv_(pv), improving_(improving), depth_(d), move_no_(m),
12         depth_new_(nd), reduction_org_(r), reduction_(r_org), ↵
            ↵depth_searched_(ds), alpha_(alpha), value_(value),
            ply_(ply), game_ply_(gply),
14         np_material_w_(npw), np_material_b_(npb), np_material_(np), ↵
            ↵stat_eval(seval) {

        }

16     const std::string fen_;

18     bool pv_;
20     bool improving_;
    int depth_;
22     int move_no_;

24     int depth_new_;
    int reduction_org_;
26     int reduction_;
    int depth_searched_;
28     int alpha_;
    int value_;

30     int ply_;
32     int game_ply_;

34     int np_material_w_;

```

```

    int np_material_b_;
36  int np_material_;

38  int stat_eval;
    };
40

42  class Logger {
    public:
44
        const char delim = ':';
46
        Logger() {
48        }

50        ~Logger() {
            f_.close();
52        }

54        void open(std::string name) {
            #ifndef DEBUG_YB_LOG
56            sync_cout << "info_string_debug_opening_file:_ " << name << sync_endl ↵
                ↵;
            #endif
58            f_.open(name);
            ready = true;
60        }

62        bool do_log(int d, int m) {
            // return true;
64            return in_range(d) && in_range(m) && (rand() % 2048 == 0);
        }

66        void add_to_log(const Record &rec) {
68            if (!ready) {
                open(Options["YBLogFile"]);
70            }
            f_ << rec.fen_ << delim
72
                << rec.pv_ << delim
74            << rec.improving_ << delim
                << rec.depth_ << delim
76            << rec.move_no_ << delim

78            << rec.depth_new_ << delim
                << rec.reduction_ << delim
80            << rec.reduction_org_ << delim
                << rec.depth_searched_ << delim
82            << rec.alpha_ << delim

```

```

    << rec.value_ << delim
84
    << rec.ply_ << delim
86    << rec.game_ply_ << delim

88    << rec.np_material_w_ << delim
    << rec.np_material_b_ << delim
90 << rec.np_material_ << delim
    << rec.stat_eval << '\n';
92 if (count % 5 == 0) {
    f_ << std::flush;
94 }
    count++;
96 }

98 private:
    bool ready = false;
100
    bool in_range(int n) {
102     return (n <= 10) ||
        (n <= 30 && n % 5 == 0) ||
104         (n <= 60 && n % 10 == 0);
    }
106
    std::ofstream f_;
108 int count = 0;

110 };

```

2 Labelling code

The below code was used to label the data gathered from StockFish tree search.

Listing A.3: Label the data

```

1
2 #!/usr/bin/env python
  # -*- coding: utf-8 -*-
4
  import asyncio
6 import time
  import argparse
8 import itertools
  import logging
10 import sys

12 import chess
  import chess.engine
14 import chess.variant

```

```

16 import multiprocessing as mp

18 async def test_pos(engine, board, depth, scores, moves):
    limit = chess.engine.Limit(depth=depth)
20     with await engine.analysis(board,limit) as analysis:
        async for info in analysis:
22             #print(info.get("score"), info.get("pv"))
            if info.get("score") != None:
24                 try:
                    scores.append(-info.get("score").relative.score())
26                     moves.append(info.get("pv")[0].uci())
                except TypeError:
28                     return False
        return True

30 def depth_reduction(alpha, scores, moves):
32     for d in range(len(scores)-1,0,-1):
        if scores[d] > alpha:
34             return d
    return 1

36 def init_worker(args):
38     global engine

40     loop = asyncio.get_event_loop()
    __, engine = loop.run_until_complete(chess.engine.popen_uci(args.uci, ))
42     engine.configure({
        'Threads': 1
44     })

46 def close_worker(w):
    global engine
48
    loop = asyncio.get_event_loop()
50     loop.run_until_complete(engine.quit())

52 def do_processing(lines):
    global engine
54     # print('do_processing')
    results = []
56
    for line in lines:
58         fields = line.strip().split(':')
        # print(fields[0])
60         board = chess.Board(fields[0])
        depth_full = int(fields[5])
        reduction_org = int(fields[6])
62         depth_searched = int(fields[8])

```

```

64     alpha = int(fields[9])
        value = int(fields[10])
66     # print(board)
        # print(depth_full, depth_searched, alpha, value)
68     scores = []
        moves = []
70     if depth_full > 30: continue
        loop = asyncio.get_event_loop()
72     ok = loop.run_until_complete(test_pos(engine, board, depth_full, scores↵
        ↵, moves))
        if ok:
74         if scores[depth_searched - 1] <= alpha:
            if scores[depth_full - 1] <= alpha:
76                 ok = "1"
            else:
78                 ok = "0"
                r = depth_full - 1
80                 for d in range(depth_full, 0, -1):
                    if scores[d - 1] > alpha:
82                         r = depth_full - d
                            break
84                 r_per = r / depth_full
                    scores = [str(x) for x in scores]
86                 results.append('.'.join(fields+scores+moves+[str(round(r_per,6)↵
                    ↵), str(r),ok])+'\n')

88     return results

90
    chunks = 100
92
    from itertools import islice
94
    def split_every(n, iterable):
96         i = iter(iterable)
            piece = list(islice(i, n))
98         while piece:
            yield piece
100         piece = list(islice(i, n))

102 def main():
        # Parse command line arguments.
104     parser = argparse.ArgumentParser(description=__doc__)

106     engine_group = parser.add_mutually_exclusive_group(required=True)
        engine_group.add_argument("-u", "--uci",
108         help="The_UCI_engine_under_test.")
        parser.add_argument("-d", "--debug", action="store_true",
110         help="Show_debug_logs.")

```

```

    parser.add_argument('-r', '--recs', required=True, help="Name_of_file_↵
        ↵with_the_data_records")
112
    args = parser.parse_args()
114
    # Configure logger.
116    logging.basicConfig(level=logging.DEBUG if args.debug else logging.↵
        ↵WARNING)

118    # Open and configure engine.

120    with open( "new.txt", "w") as wf:
        with open(args.recs) as f:
122        cpus = 4 # mp.cpu_count()

124        with mp.Pool(cpus, init_worker, (args,)) as pool:
            # print('wtf')
126            results = pool.imap(do_processing, split_every(100, f))
            for result in results:
128                if result == None: continue
                wf.write(''.join(result))
130            # print('closing ...')

132            pool.map(close_worker, range(cpus))
            pool.close()
134            pool.join()
            wf.flush()
136

138 if __name__ == "__main__":
    main()

```

3 Stockfish modifications

Below are some of the modifications made to StockFish in order to support different search reduction formulas.

3.1 First version

Listing A.4: Stockfish modifications for formula 5.2

```

1 // main.cpp
  int main(int argc, char* argv[]) {
3     double d_m_factor = 1/1.95, d_factor = 0, m_factor = 0, y_intercept= 0;

5     if(argc >= 6 && strcmp(argv[1], "reduction_params") == 0) {
        d_m_factor = std::stod(argv[2]);
7         d_factor = std::stod(argv[3]);

```

```

    m_factor = std::stod(argv[4]);
9    y_intercept = std::stod(argv[5]);
    argc -= 5;
11    char* arg0 = argv[0];
    argv = std::next(argv, 5);
13    argv[0] = arg0;
}
15    std::cout << engine_info() << std::endl;

17    UCI::init(Options);
    PSQT::init();
19    Bitboards::init();
    Position::init();
21    Bitbases::init();
    Endgames::init();
23    Search::init(d_m_factor, d_factor, m_factor, y_intercept);
    Threads.set(Options["Threads"]);
25    Search::clear(); // After threads are up

27    UCI::loop(argc, argv);

29    Threads.set(0);
    return 0;
31 }

33 // search.cpp
void Search::init(double d_m_factor, double d_factor, double m_factor, ↵
    ↵double y_intercept) {
35    for (int d = 0; d < 64; ++d) {
        Reductions[d][0] = 0;
37        Reductions[d][1] = 0;
    }
39    for (int m = 0; m < 64; ++m) {
        Reductions[0][m] = 0;
41        Reductions[1][m] = 0;
    }

43    //int d_term, m_term;
45    for (int d = 1; d < 64; ++d)
        for (int m = 1; m < 64; ++m) {
47        Reductions[d][m] = std::max(int(1024 * 1024 * (
            std::log(d)*std::log(m)*d_m_factor + std::log(d)*d_factor + std↵
            ↵::log(m)*m_factor + y_intercept)), 0);
49    }
}

```

3.2 Second version

Listing A.5: Stockfish modifications for formula 6.3

```

1 // main.cpp
2 int main(int argc, char* argv[]) {
    double d_m_factor = 1/1.95, d_factor = 0, m_factor = 0, y_intercept= 0;
4    int func_number = 0;
    if(argc >= 5 && strcmp(argv[1], "reduction_func2") == 0) {
6        if(strcmp(argv[2], "default") != 0) {
            d_m_factor = std::stod(argv[2]);
8        }
        d_factor = std::stod(argv[3]);
10       m_factor = std::stod(argv[4]);
        func_number = 1;
12       argc -= 4;
        char* arg0 = argv[0];
14       argv = std::next(argv, 4);
        argv[0] = arg0;
16     }
    if(argc >= 6 && strcmp(argv[1], "reduction_params") == 0) {
18       d_m_factor = std::stod(argv[2]);
        d_factor = std::stod(argv[3]);
20       m_factor = std::stod(argv[4]);
        y_intercept = std::stod(argv[5]);
22       argc -= 5;
        char* arg0 = argv[0];
24       argv = std::next(argv, 5);
        argv[0] = arg0;
26     }
    std::cout << engine_info() << std::endl;
28
    UCI::init(Options);
30    PSQT::init();
    Bitboards::init();
32    Position::init();
    Bitbases::init();
34    Endgames::init();
    Search::init(d_m_factor, d_factor, m_factor, y_intercept, func_number);
36    Threads.set(Options["Threads"]);
    Search::clear(); // After threads are up
38
    UCI::loop(argc, argv);
40
    Threads.set(0);
42    return 0;
    }
44
    // search.cpp
46 void Search::init(double d_m_factor, double d_factor, double m_factor, ↵
    ↵double y_intercept, int func_number) {

```

```

    for (int d = 0; d < 64; ++d) {
48     Reductions[d][0] = 0;
        Reductions[d][1] = 0;
50     }
    for (int m = 0; m < 64; ++m) {
52     Reductions[0][m] = 0;
        Reductions[1][m] = 0;
54     }

56     for (int d = 1; d < 64; ++d)
        for (int m = 1; m < 64; ++m) {
58         switch(func_number) {
            case 1:
60             Reductions[d][m] = std::max(int(1024 * 1024 * (
                std::log(d*d_factor)*std::log(m*m_factor)*d_m_factor)↵
                ↵), 0);
62             if(d == 10 && m == 10) {
                sync_cout << "f1_reductions:_" << Reductions[d][m] ↵
                ↵<< sync_endl;
64             }
            continue;
66         default:
            Reductions[d][m] = std::max(int(1024 * 1024 * (
68             std::log(d)*std::log(m)*d_m_factor + std::log(d)*↵
                ↵d_factor + std::log(m)*m_factor + y_intercept)),↵
                ↵ 0);
            if(d == 10 && m == 10) {
70             sync_cout << "f0_reductions:_" << Reductions[d][m] ↵
                ↵<< sync_endl;
            }
72         continue;

74     }
    }
76 }

```
