Beyond Tablebases:

Utilising Neural Networks for Efficient Chess Endgame Resolution

Mustafa Abdalla

September 22, 2024

Mustafa Abdalla: *Beyond Tablebases: Utilising Neural Networks for Efficient Chess Endgame Resolution*

STUDENT ID
534048

ASSESSORS
First Assessor: Prof. Dr. Jürgen Dix
First Assessor: Prof. Dr. Robert Bredereck

DATE OF SUBMISSION
September 22, 2024

EIDESSTATTLICHE VERSICHERUNG

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Alle Stellen der Arbeit, die wörtlich oder sinngemäß aus anderen Quellen übernommen wurden, wurden als solche kenntlich gemacht. Die Arbeit wurde in gleicher oder ähnlicher Form noch keiner anderen Prüfungsstelle vorgelegt.

Clausthal-Zellerfeld, September 22, 2024

Mustafa Abdalla

PUBLICATION AGREEMENT

Ich erkläre mich mit der öffentlichen Bereitstellung meiner Bachlelor's Thesis in der Instituts- und/oder Universitätsbibliothek einverstanden.

Clausthal-Zellerfeld, September 22, 2024

Mustafa Abdalla

Contents

Abstract

Chess endgames, once solved by brute-force search through tablebases, present a unique opportunity for the application of neural networks. This thesis explores the potential of neural networks to surpass traditional tablebases in efficiency and capability for chess endgame resolution.

Moving beyond the limitations of tablebases, this work investigates the development of a neural network architecture specifically designed for chess endgame evaluation and optimal move selection. The thesis will delve into what a tablebase is and how it works, developing a neural network that provides the same functionality of a tablebase more effectively and comparing both approaches analytically in terms of accuracy, speed, and scalability.

With that, the aim would be to explore the boundaries of chess endgame evaluations and present the prospects of neural networks to replace traditional methodologies of chess programming.

1 Introduction

For years, the main driver for better chess programs was the development of more efficient algorithms such as MinMax, Alpha-Beta Pruning and Monte-Carlo Tree Search amongst many others [17].

The sole purpose of each of them was to cut down on the time and resources used in order to traverse the massive move trees present during most positions of a chess game. The reason for that was due to the large branching factor of the game of chess, a position can have many millions of possible moves only at the third move of a game. Furthermore, the number of possible chess games is estimated to be around 10^{120} [27]. With just these two facts alone, one could quickly notice that a brute force approach of traversing and evaluating each and every possible move in a given position would be futile and rather counterproductive, and that a better approach would focus on only a subset of moves (so called Candidate Moves) that could lead to a position unfolding in one's favour.

To cut down on the computing time needed to come up with the best move in a given position, resources such as Opening Books, Endgame Tablebases and several heuristics could be used in order to substantially decrease the size of the search tree.

In this paper the main focus will be on the Endgame Tablebases. A game of chess is considered to be in the Endgame phase when there's only a few handful of pieces left on the board. The fewer the pieces that are left, the more plausible it would be to calculate each possible move in a given position and evaluate it. This is exactly what a Tablebase is, it is a large compilation of positions containing up to a certain amount of pieces on the board, and each position is then assigned a value that helps determine the evaluation of the position. At first glance, this may seem useful, as it would drastically cut down on the computations needed to evaluate a position algorithmic-ally, but it comes with a drawback just as significant in needing large amounts of storage space, and outside knowledge.

Alongside making the tablebase evaluations more accessible, using neural networks would help in answering the age old question of whether chess is a solved game or not. Solved in this context would mean, whether there's a known outcome that could be consistently reached given optimal play from both sides. In the case of positions with up to 7 pieces this is already determined. Neural networks could take a step beyond the available tablebases and incrementally increase the number of pieces while trying to extract a strategy to solve the types of positions that arise. Whether this approach could be extrapolated to at some point involve all pieces in the start position, this would definitely answer the question regarding the determinism of chess. This is of course far fetched with the current state of the art, but the idea could bare fruit as advancements in the field of AI take place over the coming years.

Due to that, the aim of this paper is to explore the use of Neural Networks as an alternative to tablebases. Both approaches will be discussed in detail, and their significant benchmarks would be compared. The different benefits and drawbacks would be considered and conclusions as to whether this would be a sensible approach to dealing with the issues of tablebases would be outlined at the end of this paper.

2 Chess Programming

Before diving into the details of how tablebases are built and used, a short introduction into the general concepts of chess programming is presented in order to provide an idea of how the contents of this paper fall into place within the bigger picture. The following sections only explain the concept in a rough manner, without diving too deep into the nuances of it all. For a thorough explanation please refer to the relevant sections in the works of Klein [17] & Shannon [27].

2.1 Constituents of a chess program

A chess program is in essence made up of 2 parts:

- **Evaluate**: An evaluation function that provides a score of a given position. Several aspects of a position can contribute to the output of this function but the most rudimentary form compares the number of pieces to decide which side has the "better" position.
- **Search**: A searching function that in its naive form generates a tree of all possible moves up to a certain depth.

2.2 Evaluate

Accurately scoring a chess position relies on many factors of varying significance each playing a role towards shifting the score from one side to another. Before estimating the evaluation, it is necessary to formalise how it is to be interpreted.

An Evaluation E of a position is a real number whose sign determines which side, if any, has an advantage. A positive evaluation favours white, a negative one favours black. The magnitude of the evaluation increases with the advantage that a side has.

As mentioned in the previous section, the most basic scoring function compares the number of pieces in the position, and gives the advantage to the side with more pieces. Using the number of pieces by itself gives a false metric, since a single queen is better than 8 pawns. For this reason, pieces are given relative values based on their effectiveness (see Table 1), which can be used to "weigh" the sum of the pieces.

Hence evaluation E of a chess position is given by:

$$E = \sum_{i \in \{p,n,b,r,q\}} v_i \cdot P_{w,i} - \sum_{i \in \{p,n,b,r,q\}} v_i \cdot P_{b,i}$$

Where $P_{w,i}$ and $P_{b,i}$ denote the number of white and black pieces of type i resepectively and v_i is the value of a piece of type i

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

Table 1: Value of each chess piece

The number of pieces on the board is insufficient to accurately evaluate the position. Several other heuristics are used to more accurately gauge this.

2.2.1 Heuristics

When considering what move to play Grandmasters only consider a subset of the possible moves, commonly referred to as *candidate moves*. These moves are what are deemed as promising for improving the position to ones favour.

When determining whether a move can be categorised as a candidate move, several heuristics are used to guide the process. These heuristics are not concrete rules that apply in all cases, but are rather general rules of thumb. One of the more common heuristics used is "Develop the pieces towards the centre". This essentially means that a piece placed centrally is more valuable and effective than a piece

placed on the edge of the board. This is due to the number of controlled squares being highest for any piece, when it is central on the board. Figure 1 visualises this concept. Figure 2 shows how it can be formalised for use within the algorithm.

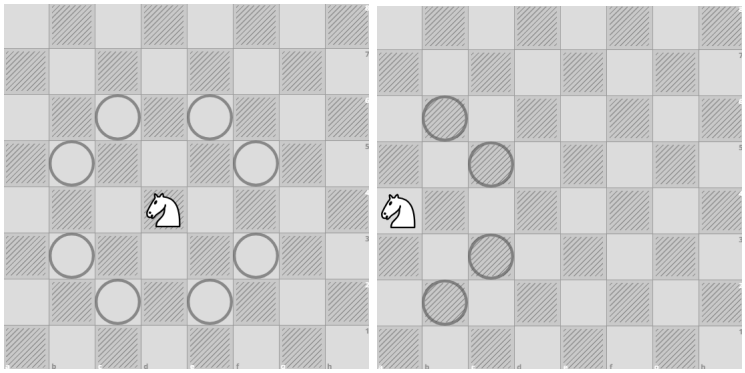

Figure 1: Spaces controlled by a central Knight vs Spaces controlled by a Knight on the rim

-50	-40	-30	-30	-30	-30	-40	-50
-40	-20	0	0	0	0	-20	-40
-30	0	10	15	15	10	0	-30
-30	5	15	20	20	15	5	-30
-30	0	15	20	20	15	0	-30
-30	5	10	15	15	10	5	-30
-40	-20	0	5	5	0	-20	-40
-50	-40	-30	-30	-30	-30	-40	-50

Figure 2: Piece Square Values for the Knight [9]

2.3 Search

In order to find the best move in a given position, one must first know what moves are possible to begin with. So naturally one would generate a tree of all possible moves that could branch out of a position within a certain depth. A simple start would be to look at all available moves, three moves deep (Usually a move refers to two plies i.e. a move by white and a move by white. In this context a move refers to a single ply for the sake of simplicity). Figure 3 shows how the tree would look.

As mentioned before, chess has quite a large branching factor, meaning at any point of time during a game, the number of possible positions increases exponentially. Table 2 quantifies this fact for the first few moves of a game. Due to that, it would not only be counterproductive to generate all possible moves from a position, but outright impossible. To solve this, analysing how humans play chess (more specifically how Grandmasters play) provide a useful perspective to improving the search function.

Number of plies (half-moves)	Number of possible positions
1	20
2	400
3	8902
4	197281
5	4865609

Table 2: Possible positions with each move

2.3.1 MinMax

MinMax is an approach that can be applied to any 2 player game, and not just chess. There's a *Minimising* player and a *Maximising* player, given an evaluation function, a specific position, and whether we're minimising or maximising, we can then follow the procedure described below [17]

- We create a move tree, where from a root position (in this example we assume the root position is after black made a move) we consider all possible replies. We do the same for each of those replies.
- After a certain predetermined depth this process is halted (or earlier due to lack of legal moves, checkmates or draws). This is done in consideration to the limited time and storage resource, and how fast the tree can grow.
- From there each leaf node of the tree is evaluated. Positions where white won would have an evaluation of ∞, ones where black wins would have -∞ and ones that are drawn would have an evaluation of 0.
- These results are then propagated back to the root, so given a position where all child nodes are evaluated we do the following
  - take the **maximum** of all child node evaluations if it's white to play
  - take the **minimum** of all child node evaluations if it's black to play

The following Figure shows the outcome of following the aforementioned steps, keeping in mind that the moves considered here are mostly piece captures, and alongside that the evaluation function used just sums up the value of the pieces.

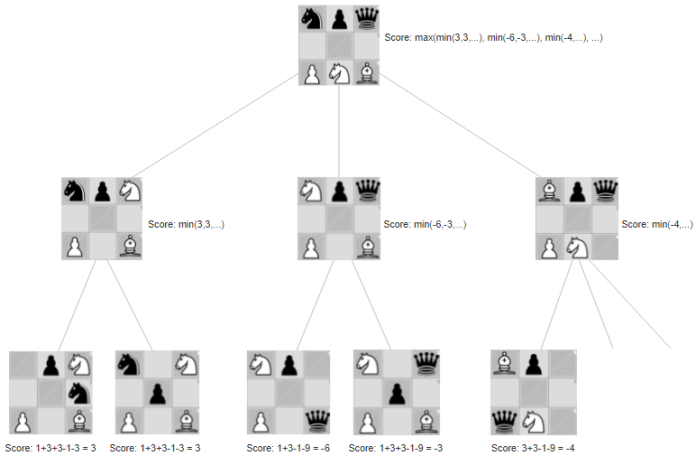

Figure 3: Simplified example of MinMax

2.3.2 Alpha-Beta Pruning

The use of MinMax on its own is effective but comes with the significant drawback of requiring a drastic amount of space and computer power to be able to evaluate each and every possible move, and as mentioned before this would mean that the algorithm would need to evaluate quite a large number of positions depending on the branching factor of a certain position.

To combat this issue Alpha-Beta Pruning was introduced. As the name suggests it *prunes* or cuts off branches that we no longer need to consider.

Consider that we are deep in our move tree, and white is to move. Assuming we have 2 Nodes, **Node 1** & **Node 2**, and each one of them has 3 child nodes. Assume we then evaluate each child node of Node 1 and back propagate them so that now Node 1 has an evaluation of **+2**.

When evaluating the first child of Node 2 we come across an evaluation of 6, so now we know that Node 2 would have an evaluation of *at least* 6. At this point we can say that Node 1 has an evaluation of +2 and Node 2 has an evaluation of $\geq +5$. But we know that in the turn that precedes Node 1 and Node 2, it was black's turn to move, and black would try to minimise the score and thus choose Node 1. With that information we can now confidently say that looking further into the children of Node 2 is no longer necessary and prune those branches off.

Figure 4 [17] shows the application Alpha-Beta Pruning alongside the MinMax algorithm, its application on Figure 3

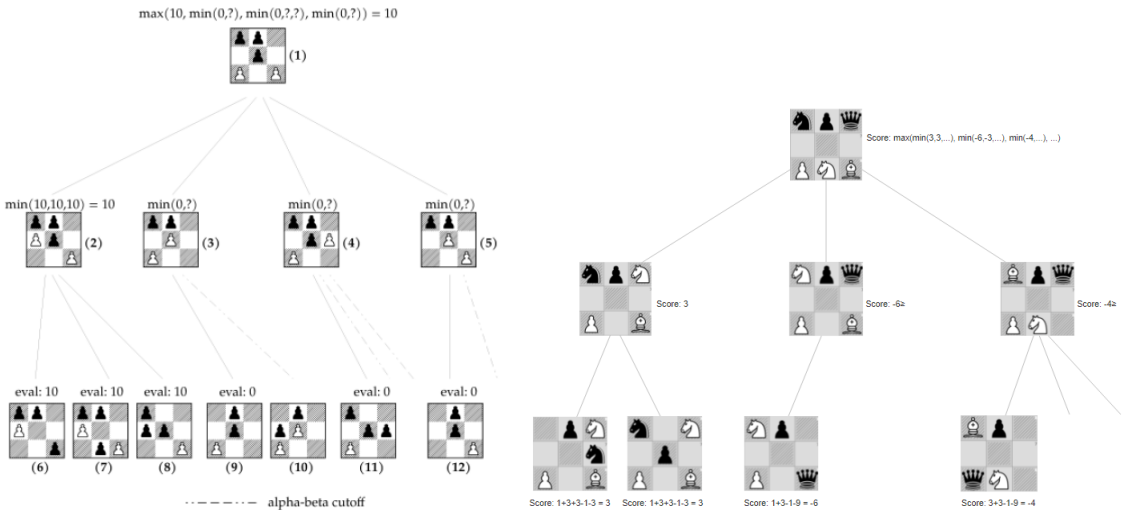

Figure 4: Simplified examples of MinMax & Alpha-Beta Pruning

2.3.3 Move ordering

In Section 2.2.1 it was mentioned that candidate moves are the ones that have priority when looking for the best move in a position. This means that generating moves that satisfy or follow certain heuristics and evaluating them first would lead to a quicker termination of the search function.

This can be done using some form of a priority queue. An example of this is made clear with Table 3

Heuristic	Priority in move list
Captures	10
Checks	5
Remaining moves	0

Table 3: Priority of heuristics used to order the moves for searching

The aforementioned techniques provide a solid base for most chess engines. They can be used for most types of positions that can arise within a game of chess. In the following sections, these techniques alongside several other ideas as discussed explicitily within the context of chess endgames.

3 Tablebases

3.1 Overview

As of writing this paper the largest available and widespread Tablebase is the Syzygy 7 man Tablebase [11], it contains evaluations for all possible positions that could arise from having up to 7 pieces on the board (7 pieces in total including the kings). To download such a Tablebase, one would need to allocate somewhere around 16TBs of storage space, and if this is not possible, one could also generate these tablebases locally. For that one would *only* need a machine with 1TB of RAM and it would take many decades of computing time.

Although they provide a very accurate evaluation for every one of those positions, the monstrosity of these tablebases makes their use obsolete for the average person. This is of significance, because anyone could download Stockfish [31], one of the strongest chess engines of today, and have it immediately available on their machine for use at anytime. Although tablebases aid chess engines like stockfish in directing a position towards a favourable endgame, the high storage demands of tablebases raises the question, whether it would be possible to specifically create an engine that specialises in endgames, and alongside that remove all need for tablebases.

3.2 The make up of a Tablebase

Tablebases are large databases containing all possible (and legal) chess positions that could arise from having a certain amount of pieces on the board. For each of those positions, retrograde analysis [8] is used to determine whether the outcome is a win, draw or loss.

Retrograde analysis works by taking a terminated position where one side is mated, and going backwards to discover all the possible positions that eventually lead up to the root position. This process is repeated for all mate positions for a given number of pieces, and subsequently a subdatabase of all possible positions that terminate in a mate are generated. Any other position that is still legal and valid but not found in the subdatabase is then marked as a draw.

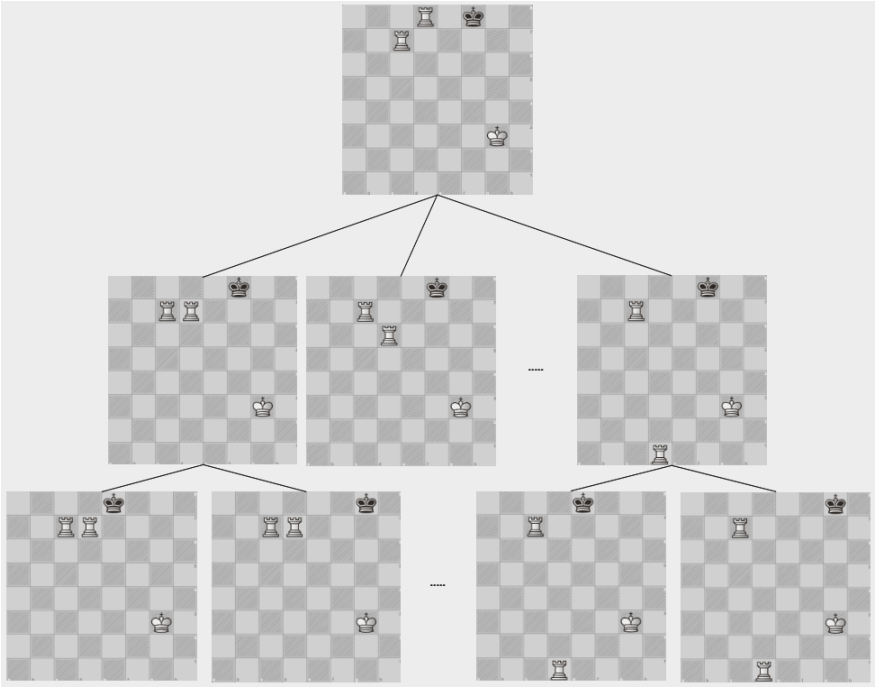

Figure 5: Simple visualisation of Retrograde Analysis

3.2.1 Metrics

Whether a move leads to a checkmate or not is ultimately the main measure of how good a move is, but there are several other aspects one could consider on the way to a move that leads to checkmate. For instance, when no mate can be found, a move that leads to resetting the 50 Move Counter [6] could point in the direction of eventually finding a mating move.

Few of the metrics that are commonly stored within tablebases are explained below:

- **WDL**: The **W**in **D**raw **L**oss metric returns an integer number describing the evaluation of a given position. The possible values that can be returned are:
  - **2**: The given position is won.
  - **1**: The given position is drawn due to the win being prevented by the 50 move rule. This is also known as a cursed win.
  - **0**: The given position is drawn.
  - **-1**: The given position is drawn due to the loss being prevented by the 50 move rule. This is also known as blessed loss.
  - **-2**: The given position is lost.
- **DTM**: The **D**epth **T**o **M**ate metric returns an integer number indicating how many moves the winning side has to make in order to reach mate.
- **DTZ**: The **D**epth **T**o **Z**eroing metric works similarly to DTM but with the objective of reaching a capture or a pawn move, why these are important is made clear with the next metric.
- **DTZ50**: The **D**epth **T**o **Z**eroing in the context of the **50** move rule metric is an extension of the DTZ metric with keeping in mind that the capture or the pawn move must occur within 50 moves so that a draw is avoided.

Depending on the simplicity of the program being developed and the performance sought after, one need not use all the metrics available, and it would suffice to only use one. This is of importance due to the large accumulative size of the files for each metric. In the case of the 6-man Syzygy Tablebases used in the development of the programs accompanying this paper [24], the WDL files alone measure to around 70GBs and they are sufficient to providing a solid base of probing the tablebase to aid in finding the best move in a position. This would save around 80GBs of additional storage space that would've been needed for the DTZ files.

3.3 Probing

Each position is assigned to an index that points to where the metrics such as WDL, DTZ, and others are stored for that specific position. In order to retrieve this information one would need to *probe* the tablebase files.

Probing consists of applying the chosen indexing algorithm of a tablebase on the given position. Once the unique index is calculated, it is used to locate where the metrics of that position are stored in order to extract them [5].

Since the objective of this paper focuses on alternatives for tablebases the implementation of this part, was left out and the publicly available library Python-Chess [13] was used in dealing with the Syzygy 6-man WDL Tablebases.. This library was chosen due to its developer being involved with most of the Syzygy Tablebases related aspects of the chess website Lichess and created the web API for using Syzygy Tablebases [30][14].

4 Neural Networks in Chess

Over the last decade computer engines went from consisting of intelligent efficient search and evaluation algorithms, to being focused on creating larger, and intricate neural network architectures that could reach newer heights when it comes to accuracy and general strength of play. Examples of this include the previously mentioned Stockfish [31] and AlphaZero [28]. The prior being an amalgamation of using algorithms like MinMax & Alpha-Beta Pruning (see Section 2) to guide the neural network in finding the most optimal move, while the latter being made up of a neural network that teaches itself the game of chess without prior outside knowledge, using techniques like Monte Carlo Tree Search [17]. Both approaches present the different paradigms of creating a strong chess engine, which in turn raises the question "How can one make use of these neural networks to improve how Endgames are played by engines?"

4.1 Overview

The main objective of this paper, as highlighted by its title, is to explore how neural networks can be used as a replacement to endgame tablebases in the hope of finding benefits when it comes to the speed of probing, but more importantly to save on the storage space required by the tablebases. For the sake of simplicity and lack of resources, this paper will look into one metric provided by tablebases, and using a neural network in place of the tablebase for probing.

More concretely, the objective is to design a neural network that given a chess position can predict the correct WDL value. Once this is proven feasible, the other metrics can be predicted by the same process.

4.2 Pattern Recognition

Similar to the process of examining how a Grandmaster would think about the problem of finding the best move in Section 2, using this approach here when given a position and determining the WDL, more abstractly determining whether it is good or bad, can help pinpoint what features make up a position.

In the experiments conducted by de Groot [3] players of varying strength ranging from masters to complete novices were presented with various positions that could arise in a standard game of chess and were given the task of reproducing the position on an empty board after viewing it for a short period of time (3-10 seconds). On average, it was found that master level players were able to more accurately reproduce the position with only a couple of pieces being misplaced, if at all, while novice players showcased having a larger margin of error when recreating the given positions.

On first glance, one could hypothesise that since a master player has spent a long period of time in contact with chess positions that their memory for remembering such positions would be better compared to that of a novice player. Nevertheless, this would be disproven when considering the second set of experiments conducted.

In the second set of experiments both the master and novice player where once again presented with the same task, but this time instead of the positions being those that could come up in a standard game, the players were presented positions where the pieces were placed completely randomly. This time around, the accuracy of the master players significantly dropped in contrast to the first experiment, and was barely better than that of the novice players.

Figure 6: Example of a position that could arise in a game

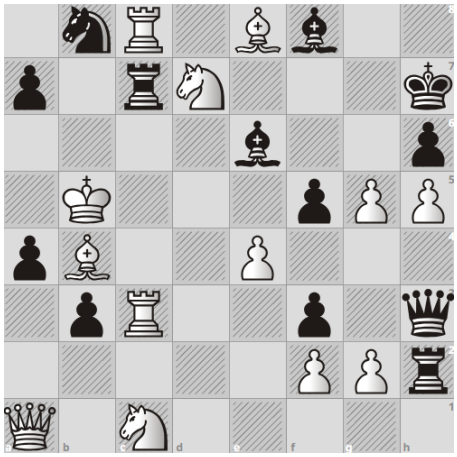

Figure 7: Example of a position where the pieces are randomly placed

Figures 6 & 7 show visually what a position that could arise from a game, and a random position

look like. In Fig.6 a master player would look at symmetrical pawn structure of white, at the e4 pawn being attacked by the Knight, and the pins (meaning a piece can or should not move when being attacked due to it covering a piece of greater value) placed on the d7 Rook and f6 Knight by the b5 and g5 Bishops respectively. While in Fig.7 the pieces are not *harmonious* with each other and are scattered haphazardly, making it harder to extract the relations and patterns between them. Figure 8 demonstrates this.

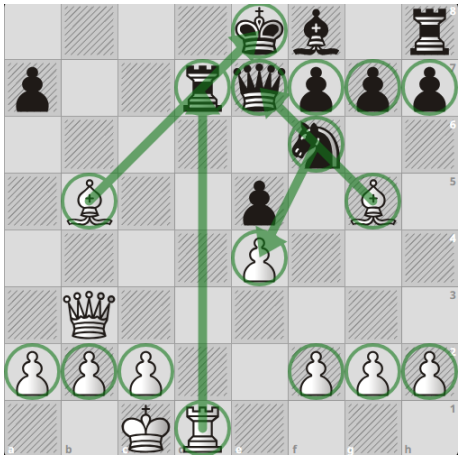

Figure 8: The patterns a master player would see when observing the position in Fig.6

The process just described is exactly what needs to be done by the neural networks, in order to extract the essential features like pawn structures, relative positions of the pieces to each other, and many more so that it can make an accurate estimation about whether a position is good or not. In simpler words, we need a neural network good at recognising patterns.

4.3 Convolutional Neural Networks

Convolutional Neural Network are a special type of neural network architecture used mainly in the field of image/object recognition. When dealing with a grey-scale image like the ones in the MNIST Classification Dataset [12] one could use the traditional Multi-Layer-Perceptron [17] where each pixel would have a value ranging from 0 (black) to 255 (white), and the 2x2 grid of pixel would then be converted to a singular 1-dimensional array that is treated as the input of the perceptron, and fully connected to the following hidden layers with the respective weights, biases and activation functions, eventually leading to the output layer containing 10 neurons/nodes each corresponding to a digit 0-9 with the value outputted at each node being the confidence/probability of the given image being the corresponding number represented by the node.

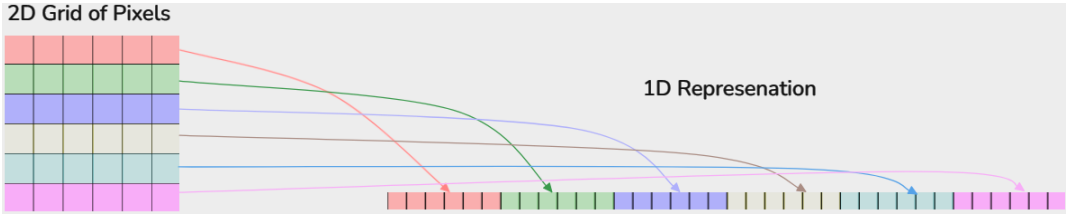

Figure 9: Modelling a 2D Grid of Pixels (commonly known as an image) in one dimension

With sufficient training data, such a model could be very accurate and reliable in several use cases,

but one major problem arises. Scaling. The MNIST Handwritten Digits Dataset contains images of dimension 28x28, in other words there are 784 individual pixels that must each correspond to a single neuron. Furthermore, most images processed in the context of image/object recognition are much larger than the measly 28x28 dimensions of the MNIST Dataset, so in the worst case, one would need to adjust the size of the input layer for each type of image to be processed. This alone increases the complexity of the architecture when using greyscale images, let alone coloured ones.

This is where CNNs shine, their architecture allows for variability in the size of the input images without significantly increasing the size or complexity of the model. The process involves the input going through multiple convolutional layers that contain kernels/filters that pass over the image as a whole and create an output matrix known as a feature map. The size of each convolutional layer depends on the size of the convolutional layer preceeding it and its resepctive kernels. The size of the kernels determines what is known as the receptive field. This tells us how much information from the previous layers is propagated through to the current layer [20].

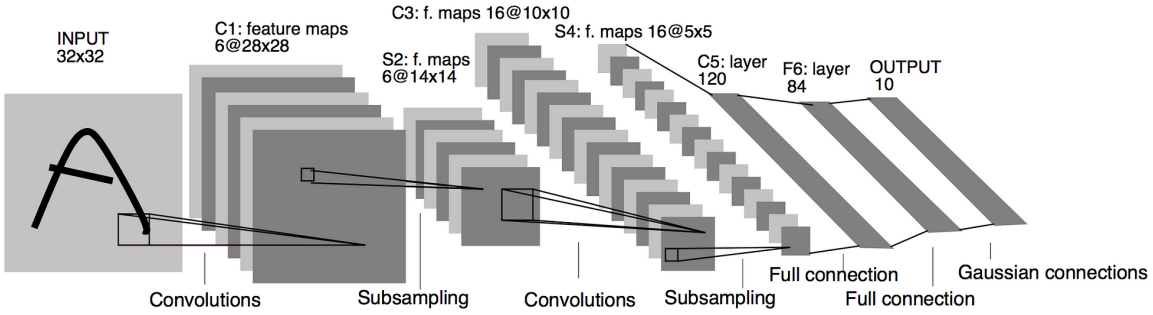

Figure 10: Example architecture of LeNet5 [19] where the kernels and convolutional layers are seen

4.3.1 Convolution & Kernels

Convolution [15] is the fundamental operation that gives CNNs their name and is key to their ability to efficiently process images, or more generally spatial data. In the context of CNNs, convolution is a mathematical operator that involves the input data (such as an image) and the kernel. The convolution operation slides the kernel over the input data, performing element-wise multiplication at each position and summing the results to produce a single output value. This process is repeated across the entire input, creating a feature map that highlights particular features or patterns in the data. For a 2D input image I and a kernel K, the convolution operation can be expressed as follows:

$$(I * K)(i, j) = \sum_m \sum_n I(i + m, j + n) \cdot K(m, n)$$

Where (i, j) represents the position in the output feature map, and (m, n) are the coordinates within the kernel. Let's consider a simple example to illustrate this process: Suppose we have a 5x5 input image and a 3x3 kernel:
Input Image:

$$\begin{bmatrix} 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 \end{bmatrix}$$

Kernel:

$$\begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix}$$

To compute the first element of the output feature map, we would perform the following calculation:

$$(1 \cdot 1) + (1 \cdot 0) + (1 \cdot 1) + (0 \cdot 0) + (1 \cdot 1) + (1 \cdot 0) + (0 \cdot 1) + (0 \cdot 0) + (0 \cdot 1) = 4$$

The kernel then slides over by one position (the stride), and the process is repeated. The final output feature map for this example would be:

$$\begin{bmatrix} 4 & 3 & 4 \\ 2 & 4 & 3 \\ 2 & 3 & 4 \end{bmatrix}$$

This example demonstrates how convolution can detect patterns or features. In this case, the kernel is sensitive to diagonal patterns in the input image. In practice, CNNs use multiple kernels to detect various features, and the convolution operation is typically followed by a non-linear activation function. The kernels' values are learned during the training process, allowing the network to automatically discover important features for the task at hand.

The kernels used within the first several layers are basic like the one used in this example, they can be used for edge detection (whether horizontal or vertical), corner detection, and many other things. Deeper within the architecture of CNNs the kernels begin extracting features that are more abstract and harder to make sense out of, for us as humans.

4.3.2 Pooling layers & Padding

From the example above, one could notice that the convolution of the 5x5 input matrix using a 3x3 kernel, resulted in a feature map output matrix of dimension 3x3. In other words, the essence of the original input matrix (*i.e. image*) was extracted into a smaller matrix. Depending on the situation this extraction into smaller matrices containing the bare minimum when it comes to essential features can be taken further by introducing the concept of pooling [20].

Briefly, pooling combines features that are similar, in other words in close proximity to one another, and spits out a number that is representative of them. There are 2 types of pooling layers that are commonly used across most CNNs.

- Average Pooling Layers
- Max Pooling Layers

Similar to the use of kernels, in Average Pooling Layers a filter is passed over the elements of the convolved input and the average of these elements is then outputted as a single element of the output matrix. Max Pooling Layers work analogously but take the maximum element instead of an average.

Suppose we have a 4x4 input matrix (which could be the output of a previous convolutional layer):

$$\begin{bmatrix} 1 & 3 & 1 & 1 \\ 2 & 8 & 2 & 0 \\ 0 & 4 & 3 & 2 \\ 6 & 1 & 0 & 3 \end{bmatrix}$$

We'll apply both max pooling and average pooling with a 2x2 filter and a stride of 2.
Max Pooling: For max pooling, we take the maximum value in each 2x2 region:

$$\begin{bmatrix} \max(1,3,2,8) & \max(1,1,2,0) \\ \max(0,4,6,1) & \max(3,2,0,3) \end{bmatrix} = \begin{bmatrix} 8 & 2 \\ 6 & 3 \end{bmatrix}$$

Average Pooling: For average pooling, we take the average value in each 2x2 region:

$$\begin{bmatrix} \frac{1+3+2+8}{4} & \frac{1+1+2+0}{4} \\ \frac{0+4+6+1}{4} & \frac{3+2+0+3}{4} \end{bmatrix} = \begin{bmatrix} 3.5 & 1 \\ 2.75 & 2 \end{bmatrix}$$

Both pooling operations reduce the spatial dimensions of the input. Max pooling tends to highlight the strongest features, while average pooling provides a smoother representation of the features. Padding is another important concept in CNNs. It involves adding extra rows and columns of zeros (or other values) around the input matrix. This is often done to preserve the spatial dimensions after convolution or to ensure that the filter can be applied to border pixels. For example, if we add a padding of 0 to our original 4x4 input, we get:

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 3 & 1 & 1 & 0 \\ 0 & 2 & 8 & 2 & 0 & 0 \\ 0 & 0 & 4 & 3 & 2 & 0 \\ 0 & 6 & 1 & 0 & 3 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

This padding allows us to apply a 3x3 convolution while maintaining the original 4x4 output dimensions.

4.3.3 The Classifier

Following the convolutional and pooling layers, CNNs typically incorporate one or more fully connected (FC) layers, also known as dense layers. These layers serve as the "classifier" part of the network and play a crucial role in interpreting the high-level features extracted by the preceding layers. In a fully connected layer, each neuron is connected to every neuron in the previous layer, hence the name "fully connected". This structure allows the network to combine global information across the entire feature space, as opposed to the local focus of convolutional layers. The primary functions of these FC layers include:

- **Feature integration:** FC layers combine local features detected by convolutional layers to form a global understanding of the input.
- **Non-linear mapping:** When combined with non-linear activation functions, FC layers can model complex, non-linear relationships between features and target outputs.
- **Dimensionality reduction:** FC layers often progressively reduce the dimensionality of the data, distilling the most relevant information for the final prediction.
- **Task-specific output:** The final FC layer is designed to produce outputs suitable for the specific task at hand, such as classification probabilities or regression values.

For instance, in a classification task, the output of the final FC layer typically has neurons corresponding to each possible class. A softmax activation function [2] is often applied to this layer to produce a probability distribution over the classes. Mathematically, the operation in a fully connected layer can be expressed as:

$$y = f(Wx + b)$$

Where x is the input vector, W is the weight matrix, b is the bias vector, f is the activation function, and y is the output vector [17].

It's worth noting that while FC layers provide powerful classification capabilities, they also introduce a large number of parameters to the network. For example, if a convolutional layer outputs a feature map of size 7x7x512, and the first FC layer has 4096 neurons, this single connection would require 7 * 7 * 512 * 4096 = 102,760,448 parameters. This can lead to increased computational requirements and potential overfitting, especially with limited training data [1]. To mitigate these issues, techniques such as dropout (randomly setting a fraction of inputs to zero during training) are often employed in FC layers. Additionally, some modern architectures have experimented with replacing FC layers with global average pooling, which can drastically reduce the number of parameters while

still maintaining good performance. In the context of transfer learning, the FC layers are often the only parts retrained when adapting a pre-trained CNN to a new task. This allows the network to leverage the general features learned from a large dataset while fine-tuning the final classification for the specific task at hand.

Understanding the role and behavior of fully connected layers is crucial for effectively designing and training CNNs, as they form the bridge between the automated feature extraction of convolutional layers and the final task-specific predictions of the network.

4.4 Images within Chess

For the reader, the previous section may seem to have went off on a tangent away from chess and the topic of this paper, but the connection shall be made clear in the paragraphs to follow.

As mentioned in 4.2, the essence of chess mastery lies within recognising patterns, and the previous section outlined how CNNs can be used exactly for that task. It is now only a matter of modelling the problem in a clever way to be able to make use of these CNNs.

4.4.1 Bitboards

Consider a black and white image. It can be viewed as grid, modelled as a 2-dimensional array of pixels where each pixel has either a value of 1 (white) or 0 (black). Similarly, a chess position can be modelled in a similar manner. Assume, we have a board containing only white pawns. The board is a grid of squares/fields, modelled as a 2-dimensional array where each square has either a value of 1 (contains a white pawn) or 0 (does not contain a white pawn). This is what bitboards are [4]. Figure 11 provides a visual example

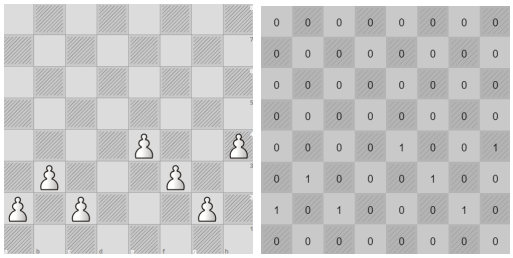

Figure 11: Bitboard representation of the white pawns on the board

To represent a colour image, each pixel component (Red, Green, or Blue) is stored on a separate "channel", this allows for dealing with each channel as a single layer in the case of grey-scale images, but instead of 0 being black and 255 being white, 255 would be the colour of the respective channel.

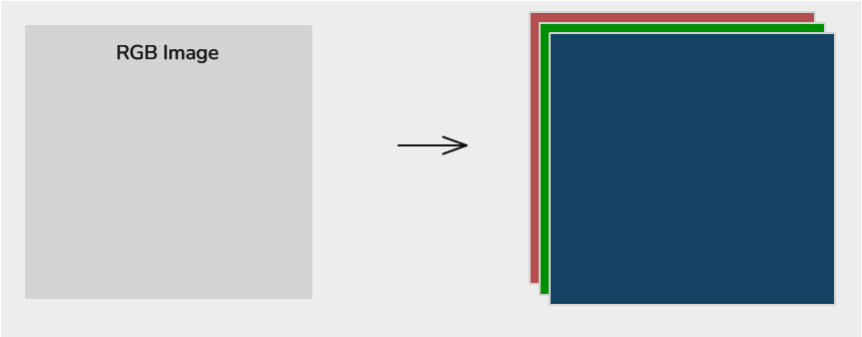

Figure 12: Decomposition of an RGB Image to Three Channels

To represent an entire chess position one would need *at least* 16 bitboards, one for each unique piece (white King, black King, white Queen, black Queen, and so on). Figure 13 shows a simplified version of this.

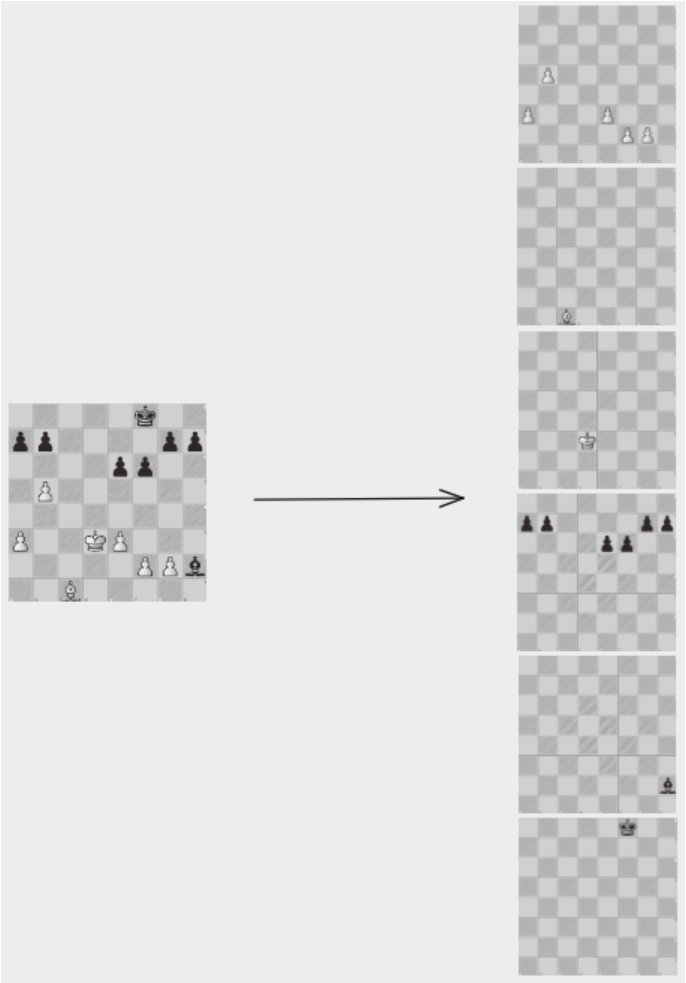

Figure 13: Decomposition of a position into various bitboards (here as actual pieces for simplicity)

With that we get a decomposition of what could be a complex chess position into its bare constituents separated over their respective channels, and from that the CNN would be able to come up with its own abstract filter/kernels to make sense and create relations out of the pieces and their positions in order to be able to come up with a concrete value of whether the given position is winning, drawn, or losing for the side to move.

4.5 Architecture

The CNN architecture chosen is designed to process 12-channel 8x8 input "images". The network consists of three convolutional layers followed by two fully connected layers. The individual layers of the model are visualised in Figure 14 and are explained below [1]:

[1] It may seem that some kernels are larger than their respective layers, but this is due to the omission of padding in the visualisation

- Input Layer:
  - Dimensions: 12 x 8 x 8
  - The 12 channels each represent the position of a unique type of chess piece on the 8x8 chessboard.
- Convolutional Layers:
  - Conv1: $12 \rightarrow 64$ channels, 3x3 kernel, padding=1
  - Conv2: $64 \rightarrow 128$ channels, 3x3 kernel, padding=1
  - Conv3: $128 \rightarrow 256$ channels, 3x3 kernel, padding=1

    In most modern CNN architectures the number of channels tends to incrementally increase, in order to allow for more complex patterns to be captured and learned so to speak. Additionally, padding is added in order to maintain the spatial dimensions of the "images" and not lose out on the minute details from layer to layer.
- Pooling Layers:
  - MaxPool2d with 2x2 kernel and stride 2 after each convolutional layer

    Pooling reduces spatial dimensions, providing some translation invariance and reducing computational load. The 2x2 pooling halves the spatial dimensions after each convolution, resulting in 4x4, then 2x2, and finally 1x1 feature maps.
- Flatten Layer
  - Converts the 256 x 1 x 1 Tensor to a 256 dimension vector.
- Fully Connected Layers:
  - FC1: $256 \rightarrow 512$ neurons
  - FC2: $512 \rightarrow 5$ neurons (output layer)

    The first FC layer expands the representation to allow for more complex combinations of the learned features. The final layer reduces to 5 outputs, corresponding to the 5 possible values of the WDL metric. This part of the model is where the classification occurs.
- Activation Functions:
  - ReLU after each convolutional and the first fully connected layer

    ReLU introduces non-linearity without suffering from vanishing gradient problems [18], allowing for effective training of deep networks.

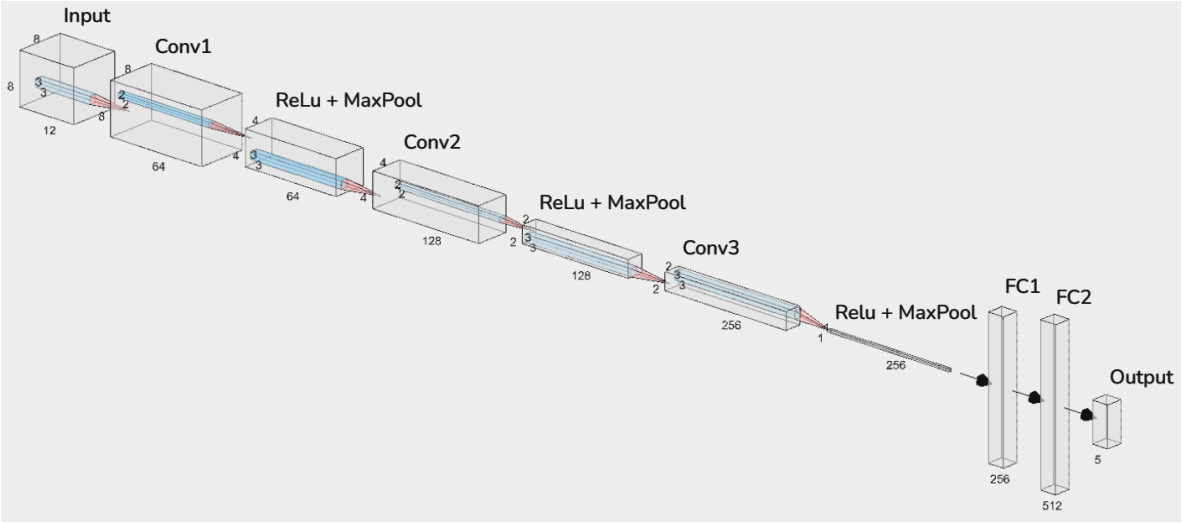

Figure 14: Endgame CNN Model Architecture

This architecture progressively reduces spatial dimensions while increasing the number of features, a common pattern in CNNs. The choice of layer sizes balances the need for feature extraction (through increasing channel counts) with the constraints of the input size. Starting with 64 channels and doubling to 128, then doubling again to 256, allows for a rich feature hierarchy without excessive computational demands given the small 8x8 input size. The use of powers of 2 for the layer sizes is also another common approach that makes use of the efficient computation of GPUs which usually have operations optimised on power of 2 sized arrays, while also keeping the process of memory access due to the used addressing schemes.

The modern CNN architectures chosen as inspiration for the Endgame Model are mainly AlexNet [18] and ResNet [16], due to them establishing significant breakthroughs in the context of image classification [26]

4.6 Training

With the model architecture now in place, the next step is to train it to achieve the desired outcome. The training step can be simplified into the following:

1. Create a dataset consisting of the features that are relevant to the prediction, and the expected output value for these respective features.
2. Split the dataset into training and testing data
3. Allow the model to iterate through the training data, and adjust its internal randomly initialised weights accordingly
4. Test the chosen weights using the testing data, and compare it to the actual expected output
5. Repeat from the second step until a satisfactory accuracy is reached

4.6.1 Dataset

To create the dataset needed for training one must first thing about how the functionality of the model would be used. In our case, we want to replace tablebases. Tablebases are provided with a position and are probed in order to output the specific metric we're after (in this case WDL). This would mean

our dataset should consist of a representation of a chess position as the *feature* and the WDL as the *target*. In other words all we need are two list containing the features, and their respective targets.

There are several options to acquiring chess endgame positions. Several collections of millions upon millions of games are publicly available online, some of them containing only games of the highest rated players [22][21]. These are valid ways of acquiring endgame position, the drawback to them would be that one would need to parse all the games so that only 6-man positions are considered.

The method chosen in this paper was to write a script capable of generating all FENs of positions containing 6 pieces or fewer [7], with that the need for precomputing the games is removed, since the endgame position is directly created. Due to the nature of neural networks and how they *learn* the data in order to find the ideal weights/kernel sizes for each layer, the larger that the dataset used is the higher the chances of improving the model's accuracy are, therefore the dataset used contains 1,000,000 endgame positions alongside their correct WDL evaluation.

4.6.2 Training Loop

The training loop is the core process where the model learns from the data [32]. It involves repeatedly exposing the model to the training data, computing the loss, and adjusting the model's parameters to minimize this loss. It works as follows:

1. Batch Processing: The training data is divided into batches. This allows for more efficient computation and helps in generalising the model by introducing some noise in the gradient computations.

2. Forward Pass: For each batch, the model performs a forward pass. The input data (12x8x8 Bitboard representation of FEN) is fed through the network, resulting in predictions for the desired outcome (WDL).

3. Loss Computation: The model's predictions are compared to the actual WDL values using a loss function. In this case, we use the cross-entropy loss function [23], which is particularly suitable for multi-class classification problems like our WDL prediction.

4. Backpropagation: The gradients of the loss with respect to the model's parameters are computed using backpropagation. This process calculates how much each parameter contributed to the error.

5. Parameter Update: The model's parameters are updated using an optimization algorithm (e.g., Stochastic Gradient Descent or Adam [15]) to minimize the loss.

6. Iteration: Steps 2-5 are repeated for each batch in the training data, constituting one epoch. Multiple epochs are typically performed to refine the model's performance.

Cross-Entropy Loss Function The cross-entropy loss function [33] is used in this model due to its effectiveness in multi-class classification tasks. It measures the dissimilarity between the predicted probability distribution and the actual distribution of the WDL outcomes. For a single training example, the cross-entropy loss is defined as:

$$L = -\sum_{i=1}^{C} y_i \log(\hat{y}_i)$$

Where:

- C is the number of classes (in our case, 3 for Win, Draw, Loss)
- y_i is the true probability of class i (either 0 or 1 in our one-hot encoded targets)

- $\hat{y}_i$ is the predicted probability of class i

The loss is minimized when the predicted probabilities closely match the true probabilities. In our case, since each position has only one correct WDL value, the true probability will be 1 for the correct class and 0 for the others. The use of cross-entropy loss encourages the model to be confident in its correct predictions while heavily penalizing confident incorrect predictions. This aligns well with our goal of accurately classifying chess endgame positions into Win, Draw, or Loss categories. During training, the model aims to minimize this loss function across all training examples, effectively learning to accurately predict the WDL values for given chess positions.

4.7 Results

Two versions of the model were trained, one using 10 Epochs (a.k.a. iterations of the training loop) and the other trained using 50. The relevant metrics throughout the epochs are the training and validation loss (calculated with the previously mentioned Cross Entropy Loss).

The following Plots visualise how these change, alongside a Plot of the accuracy of each model.

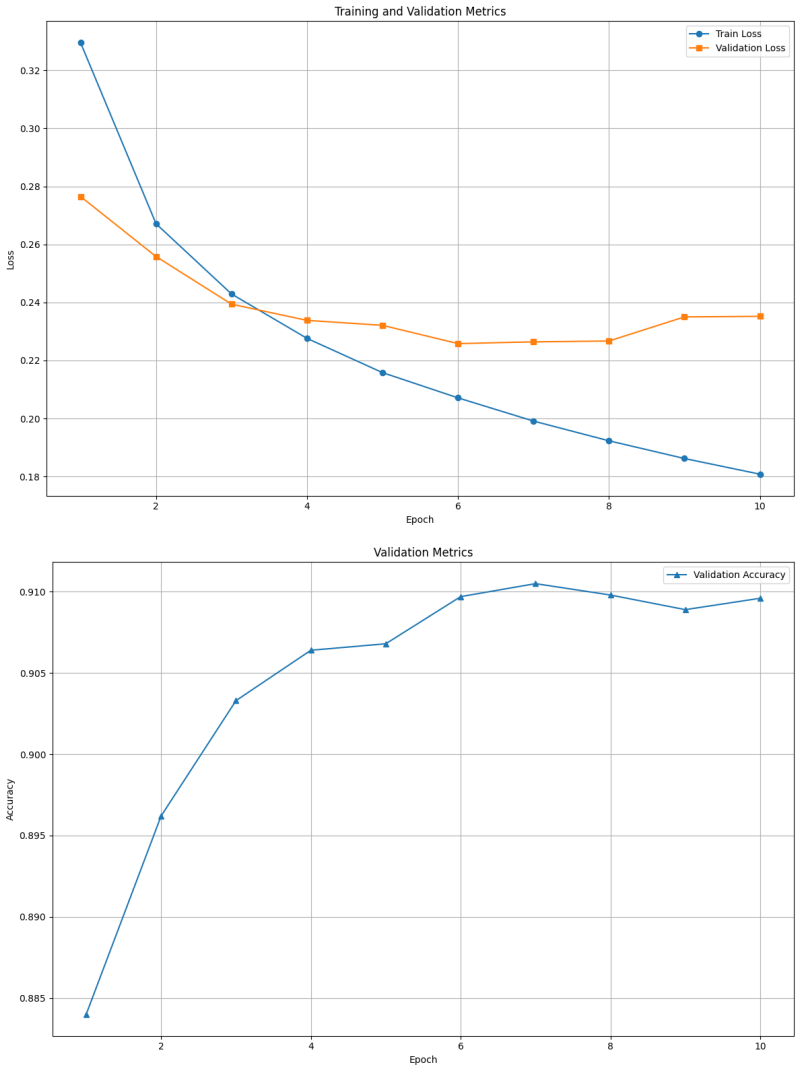

Figure 15: Training and Validation Loss alongside Validation Accuracy for 10 Epochs

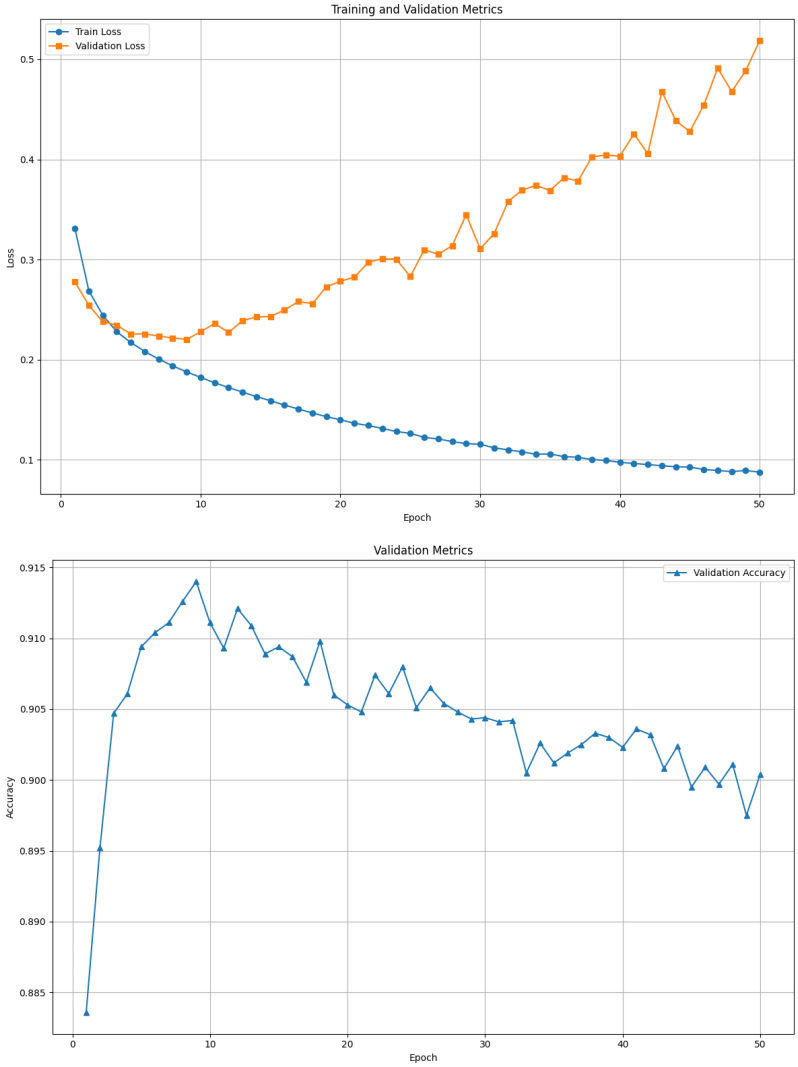

Figure 16: Training and Validation Loss alongside Validation Accuracy 50 Epochs

We notice that after 10 epochs the model begins to have an increasing validation loss, this usually points towards the fact that the model is overfitting. This essentially means that instead of learning patterns, and generalising these findings accross the possible positions it could face, the model starts to memorise the specific inputs it gets. This in turn leads to a decrease in the accuracy of the predicted values.

A possible reason for this happening, would be the "limited" sample space. The amount of patterns that the model could learn isn't that large when there are at most 6 figures on the board. This would lead to the model coming across similar or even identical patterns several times as it goes through the training dataset. This repeated exposure to the same pattern leads to the model simply memorising the output that is expected whenever it faces that said pattern.

When it comes to practical use below is an example of the models being used to probe a position.

Listing 1: Example comparing output of both models to Syzygy 6-man Tablebase

```
Input FEN: 8/K2p4/8/8/6P1/3P4/Q7/2k5 w - - 0 1
. . . . . . . .
K . . p . . . .
. . . . . . . .
. . . . . . . .
. . . . . . P .
. . . P . . . .
Q . . . . . . .
. . k . . . . .
Predicted WDL (50 Epoch Model): 2 (0.172 s)
Predicted WDL (10 Epoch Model): 2 (0.016 s)
Actual WDL: 2 (0.093 s)
```

On average the standard approach of probing the tablebase, and loading the 10 epoch model and predicting the WDL for the input FEN have very close times. The main difference would be to the 50 Epoch model.

Below is a practical comparison of the models with 1000 randomly generated FEN strings.

Listing 2: Practical comparision of accuracy between models

```
No. of tested FENs: 1000
10 Epoch Accuracy: 0.921
50 Epoch Accuracy: 0.910
```

This comparison only takes into consideration the time needed to load the weights of the trained models and using them to predict the WDL of the given position.

The actual training time was left out in this section due to it varying depending on the hardware on which it was trained. The hardware used in this case was the NVIDIA GeForce GTX 1650 GPU which is considered to be on the lower end of the today's GPU, but with the help of CUDA [10] it was possible to vectorise the process of training the model by using PyTorch Tensors [25], allowing the complete utilisation of the GPU. With that in mind, the training of both models took an accumulative time of around 3 hours.

Having the models predict the WDL in a somewhat comparable time frame to the tablebase probing it is a nice advantage but the main objective here was to significantly decrease the amount of storage space required.

The storage would mostly be needed for the dataset used to train the models, and once that step is done, we'll need to store the weights of the trained models. With the help of the NumPy library, alongside using PyTorch to extract and store the weights in a convenient compatible way lead to them needing no more than 3-5GBs of storage all together.

This is a substantial decrease in the storage needs compared to the nearly 70GBs of storage needed just for the WDL Tablebase. The drawback is that the prediction are no longer perfect, but with a more carefully tailored dataset, or cleverly optimised CNN Architecture the accuracy could be increased even more.

4.8 Shortcomings

Some readers may have noticed what could only seem like a logical fallacy when the dataset and its creation were first mentioned. The tablebase was used to create the dataset for the model trained to replace said tablebase. In other words, we use the tablebase to replace the tablebase.

As foolish as it may sound at first, this has its reason. The original aspect of this paper was to take a zero-knowledge base approach. This means that the neural network would learn to evaluate

positions from randomly guessing WDL values at first, and with the help of a function that rewards the model when it guesses correctly and penalises it heavily when its wrong, to accurately returning the desired output value. In essence, it would be a process of trial and error.

This approach is how Alpha-Zero [28] was capable of training itself towards chess mastery in less than a day.

As it was mentioned in Klein's Chapter of implementing AlphaZero techniques in HexaPawnZero [17], programming a zero-knowledge base neural networks requires colossal amounts of computing power that simply were not readily available in the volume necessary to make this possible, and could theoretically last many hours longer training.

Assuming one could gain access to sufficient computing power, this would be an interesting continuation of this paper, perhaps on tablebases with 3-5 pieces as a start.

Another major point of improvement would be the design of the CNN architecture. The architecture proposed in Section 4.5 was settled upon by a series of trial and error experiments comparing the different outcomes, while also using the pointers mentioned in Stanford's CS231n course as a guide [29]. I was attempting to find a good balance between the resulting models and their accuracies, alongside ensuring that the training time does not take excessively long.

The training time could also be a limiting factor if not dealt with properly. There were several points throughout the process that took several hours to deal with. For instance, downloading the tablebase metrics was no easy task, but even after writing my own script [24] to download them, it took a little over an hour to have all the necessary files available locally. When it comes to training the models, the 10 Epoch model took on average 40 minutes to complete its training, while the 50 Epoch model took around 90 minutes.

The different versions of the concrete implementations, could be found in my code repository titled *"NeuralBases"* [24]. It showcases how the end goal was incrementally refined. Initially, the idea was to create a chess playing neural network and compare it to the use of the tablebase in assisting a minimal implementation of a MinMax Chess engine. Upon closer inspection, this idea was scrapped due to it not being a fair comparison.

The objective of a tablebase is to provide metrics about the given positions. Therefore, the original idea would be limited by how good my own implementation of the chess engine is, and would give an inaccurate representation of the true capabilities of tablebases. At the end, I settled on comparing the ability of neural networks to come up with the same metrics (in this case only the WDL) as those of the tablebase.

5 Conclusions

This study has demonstrated the significant potential of neural networks as a viable alternative to traditional chess endgame tablebases. By employing a Convolutional Neural Network architecture specifically tailored for chess endgame positions, it was shown that it is possible to achieve high-quality predictions while drastically reducing storage requirements. While the neural network approach does not yet match the perfect accuracy of tablebases, the trade-off between a small margin of error and a substantial reduction in storage space is compelling. This balance is particularly significant given the exponential growth in storage requirements for traditional tablebases as the number of pieces increases. The results, though not flawless, serve as a proof of concept that opens up new avenues for research in this area. They invite further exploration into optimal neural network architectures that could potentially replace tablebases entirely. Future work in this direction could involve:

- Developing zero-knowledge models similar to AlphaZero, which learn purely from self-play without human knowledge
- Exploring more advanced and deeper neural network architectures
- Investigating the application of reinforcement learning techniques to improve model performance
- Addressing the challenges of computational power and training time requirements

The rapid advancements in machine learning suggest that a complete replacement of tablebases by neural networks may be achievable in the foreseeable future. Moreover, this approach could potentially extend beyond the current limitations of tablebases. While tablebases are currently limited to 7-piece endgames due to storage constraints, neural networks could theoretically handle more complex positions with additional pieces, opening up new possibilities in chess endgame analysis. In conclusion, while there is still work to be done, this research highlights the potential for neural networks to revolutionize chess endgame analysis. As we continue to push the boundaries of machine learning and artificial intelligence, we may soon see a paradigm shift in how chess engines handle endgame positions, combining the efficiency of neural networks with the precision required for high-level chess play.

References

[1] S. H. Shabbeer Basha, Shiv Ram Dubey, Viswanath Pulabaigari, and Snehasis Mukherjee. Impact of fully connected layers on performance of convolutional neural networks for image classification. *CoRR*, abs/1902.02771, 2019.

[2] John Scott Bridle. Probabilistic interpretation of feedforward classification network outputs, with relationships to statistical pattern recognition. In *NATO Neurocomputing*, 1989.

[3] Charles Brook and Adriaan Groot. Thought and choice in chess. *The American Journal of Psychology*, 79:348, 06 1966.

[4] Bitboards. `https://www.chessprogramming.org/Bitboards`, 2024. Accessed: 2024-08-26.

[5] Endgame tablebases. `https://www.chessprogramming.org/Endgame_Tablebases`, 2024. Accessed: 2024-07-29.

[6] Fifty move rule. `https://www.chessprogramming.org/Fifty-move_Rule`, 2024. Accessed: 2024-07-29.

[7] Forsyth-edwards notation. `https://www.chessprogramming.org/Forsyth-Edwards_Notation`, 2024. Accessed: 2024-08-26.

[8] Retrograde analysis. `https://www.chessprogramming.org/Retrograde_Analysis`, 2024. Accessed: 2024-07-29.

[9] Simplified evaluation function. `https://www.chessprogramming.org/Simplified_Evaluation_Function`, 2024. Accessed: 2024-07-29.

[10] Cuda with pytorch. `https://pytorch.org/docs/stable/cuda.html`, 2024. Accessed: 2024-09-11.

[11] Ronald de Man. Syzygy tablebases. `https://github.com/syzygy1/tb`, 2024. Accessed: 2024-07-29.

[12] Li Deng. The mnist database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine*, 29(6):141–142, 2012.

[13] Niklas Fiekas. Python-chess. `https://github.com/niklasf/python-chess`, 2024. Accessed: 2024-07-29.

[14] Niklas Fiekas. Syzygy api. `https://syzygy-tables.info/`, 2024. Accessed: 2024-07-29.

[15] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

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

[17] Dominik Klein. Neural networks for chess, 2022.

[18] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25, 2012.

[19] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[20] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436–444, 2015.

[21] Lichess elite database. https://database.nikonoel.fr/, 2024. Accessed: 2024-09-11.

[22] Lichess open database. https://database.lichess.org/, 2024. Accessed: 2024-09-11.

[23] Anqi Mao, Mehryar Mohri, and Yutao Zhong. Cross-entropy loss functions: Theoretical analysis and applications, 2023.

[24] Neuralbases. https://github.com/Mufasaxi/NeuralBases, 2024. Accessed: 2024-09-22.

[25] Pytorch tensors. https://pytorch.org/tutorials/beginner/introyt/tensors_deeper_tutorial.html, 2024. Accessed: 2024-09-11.

[26] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. Imagenet large scale visual recognition challenge, 2015.

[27] Claude E. Shannon. Programming a computer playing chess. *Philosophical Magazine*, Ser.7, 41(312), 1959.

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

[29] Stanford. Cs231n: Deep learning for computer vision. https://cs231n.github.io/convolutional-networks/#fc, 2024. Accessed: 2024-09-22.

[30] Lichess team. Lichess. https://lichess.org/, 2024. Accessed: 2024-07-29.

[31] Stockfish Team. Stockfish 16.1. https://stockfishchess.org/blog/2024/stockfish-16-1/, 2024. Accessed: 2024-07-26.

[32] Training loops in pytorch. https://pytorch.org/tutorials/beginner/introyt/trainingyt.html, 2024. Accessed: 2024-09-22.

[33] Zhilu Zhang and Mert R. Sabuncu. Generalized cross entropy loss for training deep neural networks with noisy labels, 2018.