UNIVERSITY OF GRONINGEN

MASTER'S THESIS

Learning to Play Chess with Minimal Lookahead and Deep Value Neural Networks

Author:
Matthia SABATELLI
s2847485

Supervisors:
Dr. M.A. (Marco) WIERING[1]
Dr. Valeriu CODREANU[2]

[1]Institute of Artificial Intelligence and Cognitive Engineering, University of Groningen
[2]SURFsara BV, Science Park 140, Amsterdam

October 30, 2017

university of groningen

faculty of mathematics and natural sciences

“*Reductio ad absurdum is one of a mathematician’s finest weapons. It is a far finer gambit than any chess gambit: a chess player may offer the sacrifice of a pawn or even a piece, but a mathematician offers the game.*”

Godfrey H. Hardy

University of Groningen

Abstract

Faculty of Mathematics and Natural Sciences

Master of Science

Learning to Play Chess with Minimal Lookahead and Deep Value Neural Networks

by Matthia SABATELLI
s2847485

The game of chess has always been a very important testbed for the Artificial Intelligence community. Even though the goal of training a program to play as good as the strongest human players is not considered as a hard challenge anymore, so far no work has been done in creating a system that does not have to rely on expensive lookahead algorithms to play the game at a high level. In this work we show how carefully trained Value Neural Networks are able to play high level chess without looking ahead more than one move.
To achieve this, we have investigated the capabilities that Artificial Neural Networks (ANNs) have when it comes to pattern recognition, an ability that distinguishes chess Grandmasters from the more amateur players. We firstly propose a novel training approach specifically designed for pursuing the previously mentioned goal. Secondly, we investigate the performances of both Multilayer Perceptrons (MLPs) and Convolutional Neural Networks (CNNs) as optimal neural architecture in chess. After having assessed the superiority of the first architecture, we propose a novel input representation of the chess board that allows CNNs to outperform MLPs for the first time as chess evaluation functions. We finally investigate the performances of our best ANNs on a state of the art test, specifically designed to evaluate the strength of chess playing programs. Our results show how it is possible to play high quality chess only with Value Neural Networks, without having to rely on techniques involving lookahead.

Acknowledgements

This thesis would have never seen the light if it wouldn't have been for the following very special people.
Firstly, I would like to thank my main supervisor Marco. You have provided me with so many great insights during these years in Groningen that I will always be grateful to you for having shared so much knowledge with me. Besides having been a great supervisor you have been an even greater friend who helped me in my stay at the AI department from day 1, when we had lunch together in the canteen.
I also owe my deepest gratitude to my second supervisor Vali. Reading your emails in which you always showed so much enthusiasm about the development of the project helped me a lot in pushing the boundaries of my research always one step further. I would also like to thank you for having provided me with some extra computer power even when you weren't supposed to and for having fixed some of my issues on the cluster when you were technically on holiday in the US.
I am also very grateful to my dear friend Francesco. Thanks to him, I will always have a new fun story to tell at parties about our stay in the Netherlands and about what it meant to study AI. I truly hope that now that you are moving to Japan also the eastern world will enjoy your Bon Jovi karaoke skills.
I'm also indebted to my close Dutch friend Marco Gunnink. Thank you for all the patience and time you invested together with me in debugging my code each time I was struggling with it. Also remember, next time the 2 of us will have a great idea together, let's make sure we'll keep it for ourselves.
I would also like to thank Matteo, Samuele and Francesco V. for having been the only people being actually brave enough to visit me here in Groningen.
Furthermore also thanks to Irene, Zacharias, Yaroslav, Roberts and Kat for all the nice memories.
Finally, my warmest thankfulness goes to my grandparents and mother. I would like to thank you for having dealt with all my struggles here in the Netherlands, that I'm sure, didn't make your life easier. Thank you for your constant support!

Contents

List of Figures

List of Tables

Ricordo a perfezione
ogni colazione [. . .]
solo ora che son grande
guardando la tua foto,
caro padre io capisco
di averti amato troppo poco . . .

Chapter 1

Introduction

Using Artificial Intelligence to teach programs to play games has grabbed the attention of many researchers over the past decades. The chances of finding a game that has not been approached from a Machine Learning perspective yet are in fact very low. Over the years, particular attention has been given to boardgames. Othello, Backgammon, Checkers, Chess and most recently Go are all proofs of how a combination of proper Machine Learning techniques and sufficient computer power, make it possible for computer programs to outperform even the best human players.
In this thesis, the game of chess has been researched. Developed around the 6th century A.D. in China, Persia and India, chess has been part of human history for a very long time now (Murray, 1913). Despite its age, it continues to grab the attention of millions of players around the world. A recent survey has estimated that at the moment there are $\approx$ 600 million regular chess players over the world [1] and more than 170.000 rated players. These statistics show that chess is one of the most played and popular boardgames of all time.
Besides being so popular, chess has been very interesting from an Artificial Intelligence perspective as well. It can, in fact, be considered as a challenging testbed that keeps being used by the AI community to test the most recent Machine Learning developments. Driven by these two reasons, this work investigates the use of Deep Learning algorithms (LeCun, Bengio, and Hinton, 2015) that make it possible for a computer program to play as a highly ranked player.
Besides this, we explore if it is possible to teach a program to play chess without letting it use any lookahead algorithms. This means that the program should be able to maximize the chances of winning a chess game, without having to evaluate a lot of future board states. On the contrary, given any board position, the program should be able to find the next optimal move by only exploring the board states of the immediate possible moves.
More information about this main research question will be presented in section 1.2, but before that, the following main topics will be approached in this chapter: in section 1.1 we investigate the general link between Machine Learning and board games. We explore what it means to teach a computer program to master a board game and present the most popular and successful algorithms that have been used in this domain. Specific attention is given to the game of chess in section 1.1.1, where we present how strong the link between Artificial Intelligence and the game of chess is.

1.1 Machine Learning and Board Games

Regardless of what the considered game is, the main thread that links all the research that has been done in this domain is very simple: teaching computers to play as highly ranked human players, without providing them with expert handcrafted knowledge. This is achieved

[1] http://www.fide.com/component/content/article/1-fide-news/6376-agon-releases-new-chess-player-statistics-from-yougov.html

by finding what is defined as an *Evaluation Function*: a mathematical function that is able to assign a particular value to any board position. Once the system is able to evaluate different board positions very precisely, and do that on a large set of them, it is usually able to master the considered board game (Finnsson and Björnsson, 2008).

From a Machine Learning perspective the goal of finding this evaluation function is usually accomplished by making use of either the *Supervised Learning* or the *Reinforcement Learning* approach. In the first approach, a computer program manages to learn how to play the game by learning from labeled data. This labeled data can either consist of moves played by highly ranked players (Clark and Storkey, 2015), which the system needs to be able to reproduce, or, as will be presented in this thesis, a set of evaluations that tell how good or bad board positions can be. In the case of *Reinforcement Learning*, the system manages to master the game through experience (Wiering and Van Otterlo, 2012). Usually, this is done by learning from the final outcomes of the games that the system plays against itself or expert players (Van Der Ree and Wiering, 2013). According to how well it performs, the program adjusts its way of playing and gets stronger and stronger over time.

The most famous example of a computer program performing as well as the best human players is based on the famous TD(λ) learning algorithm, proposed by (Sutton, 1988) and made famous by (Tesauro, 1994). TD(λ) is able to make predictions in initially unknown environments, about the discounted sum of future rewards, the return, and a certain behavior policy (Ghory, 2004). In terms of game playing programs, this means that the algorithm is able to infer, given a certain position on the board and a certain move, how likely it is to win that particular game. Through TD learning it is possible to learn good estimates of the expected return very quickly (Van Seijen and Sutton, 2014). This allowed Tesauro's program, TD-Gammon, to teach itself how to play the game of backgammon at human expert level by only learning from the final outcome of the games. Thanks to the detailed analysis proposed in (Sutton and Barto, 1998), the TD(λ) algorithm has later been successfully applied to *Othello* (Lucas and Runarsson, 2006), *Draughts* (Patist and Wiering, 2004) and *Chess*, firstly by (Thrun, 1995), and later by (Baxter, Tridgell, and Weaver, 2000), (Lai, 2015) and (David, Netanyahu, and Wolf, 2016).

It is also possible to learn from the final outcome of the games by combining *Reinforcement Learning* and *Evolutionary Computing*. In (Chellapilla and Fogel, 1999) the authors show how, by making use of a combination of genetic algorithms together with an Artificial Neural Network (ANN), the program managed to get a rating $> 99.61\%$ of all players registered on a reputable checkers server. The genetic algorithm has been used in order to fine-tune the set of hyperparameters of the ANN, which was trained on the feedback offered by the final outcomes of each game played (i.e., win, lose, or draw). The program managed to get a final rating of *Class A*, which corresponds to a level of game playing of a player with the Master title. This approach has been improved by (Fogel and Chellapilla, 2002), where the program managed to beat *Chinook*, a world-champion checkers program with a rating of 2814.

It is worth mentioning that all the research presented so far has only made use of Multilayer Perceptrons (MLPs) as ANN architecture. In (Schaul and Schmidhuber, 2009), a scalable neural network architecture suitable for training different programs on different games with different board sizes is presented. Numerous elements of this work already suggested the potential of using Convolutional Neural Networks that have been so successfully applied in the game of *Go* (Silver et al., 2016).

The idea of teaching a program to obtain particular knowledge about a board game, while at the same time not making any use of too many handcrafted features, has guided the research proposed in this thesis as well. However, we have pushed the boundaries of our research even one step further. We want to achieve this without having to rely on any lookahead algorithms. As proposed by (Charness, 1991), there exists a trade off between how well a computer can master the game of chess and how much it has to rely on lookahead algorithms. In this thesis

we aim to find where this trade off starts. We do this by trying to train the system similar to how human players would approach the game, taking inspiration from the work proposed in (Herik, Donkers, and Spronck, 2005).
This is a research question that has hardly been tackled in the history of computer programs playing board games. In fact, even though (Thrun, 1995) and later (Baxter, Tridgell, and Weaver, 2000) have both obtained impressive results on the game of chess, they only managed to achieve them thanks to the adaption of the TD(λ) algorithm to the MinMax algorithm deeply analyzed by (Moriarty and Miikkulainen, 1994). And even the most recent accomplishment of (Silver et al., 2016) makes use of a lot of lookahead in order to master the game of Go, by adapting *Reinforcement Learning* to the Monte Carlo Tree Search algorithm (Banerjee and Stone, 2007).

1.1.1 Artificial Intelligence & Chess

The first example of an automatic machine able to play against humans can already be found in the late 18th century. Called *The Turk*, and created by the baron Wolfgang von Kempelen, this theoretically fully autonomous machine toured over all Europe in order to play against the most famous personalities of its time. Napoleon Bonaparte and Benjamin Franklin are only two of many famous characters that were defeated by it (Levitt, 2000). Presented as a completely self operating machine, *The Turk* actually turned out to be a fraud. Inside the machine, a skilled human chess player was in fact able to govern the complicated mechanics of the automaton and make it look like as it was autonomously playing. Even though *The Turk* is very far from being a concrete example of an Artificial Intelligence playing chess, its story is part of a lot of people's collective imagination. It can in fact be considered as the first human attempt in creating a machine able to play chess, which gives this story a very romantic vibe.
The most famous example of a computer outperforming human players is certainly *Deep Blue*. Created by IBM in the middle of the 90's, it became extremely famous in 1997 when it managed to beat the then chess world champion Garry Kasparov. In a match of 6 games played in New York, IBM's supercomputer defeated the Russian Grand Master (GM) with a score of $3.5-2.5$, being the very first example of a machine outperforming the best human player in chess (Campbell, Hoane, and Hsu, 2002). The impact of the outcome of this match was huge. On the one side, it turned out to be a major shock for the chess community, that for the first time experienced the concrete possibility of being outperformed by a machine in such a complex game. On the other hand, Deep Blue's victory represented a major breakthrough in the history of AI. This kind of turning point, strongly related to boardgames, is probably only outperformed by DeepMind's AlphaGo program (Silver et al., 2016).
Despite the very successful result obtained by *Deep Blue*, IBM's supercomputer is far from being similar to how human players approach the game of chess. It made, in fact, use of a very large amount of handcrafted features, together with a lot of computer power that made it possible to compute $\approx$ 200 million positions per second. The Indian GM Viswanathan Anand, world champion between 2007 and 2012, mentioned that he doesn't compute more than 14 possible board positions ahead, before every move. This makes it obvious that the way computers have been approaching chess is very different from how experienced humans do.
Attempts driven by this idea, make use of ANNs and *Reinforcement Learning* to create programs that perform similarly to humans. The most famous example is the already mentioned program *KnightCap*. A chess engine that, thanks to the combination of ANNs and the previously mentioned TD-Learning algorithm, managed to win the Australasian National Computer Chess Championship, twice (Baxter, Tridgell, and Weaver, 2000). Even though it is

far less famous than *Deep Blue*, we consider *KnightCap* as the first successful example of a program, performing as a highly ranked player that does not use a lot of hard-coded chess knowledge.
As shown by these examples, the link between machines and chess is very strong. The main idea of this thesis is to create a program that is able to play as a highly ranked player without providing it with too much preprogrammed chess knowledge. At the same time, no computer power will be invested in exploring large sets of future board states before committing to a chess move.

1.2 Research Questions

The main research question this thesis aims to answer is:

- Is it possible to train a computer program to play chess at an advanced level, without having to rely on lookahead algorithms during the evaluation process?

Which can be rephrased as follows: is it possible to teach a computer program the skill of understanding if a position is *Winning* or *Losing*, by only relying on the information that is present in the position itself?
In order to find an answer to this research question, multiple minor equally interesting research questions have to be considered. They can be summarized by the following points:

1. How should the system be trained to learn the previously mentioned skill?
2. How should the chess board be represented to the ANNs? In fact, as different board representations can be used as input for the ANNs, which one is able to help the ANN maximize its performance?
3. Is it possible to use Convolutional Neural Networks in chess? Literature seems to be very skeptical about it, and almost no related research can be found about it (Oshri and Khandwala, 2016). A related interesting discussion on *Quora* can be found online: "Can convolutional neural networks be trained to play chess really well?" [2].
4. Assuming it is actually possible to make use of ANNs to teach a program to play without relying on any lookahead algorithms, how much time will the training process take?

[2]https://www.quora.com/Can-convolutional-neural-networks-be-trained-to-play-chess-really

Chapter 2

Methods

In this chapter we present the main methods that have been used during our research. The chapter is divided into five different Sections. In the first one, 2.1, we explain what it means to evaluate a chess position and we introduce how we have taught this particular skill to the system. In section 2.2 we start explaining how we have created the datasets that we have used for all the Machine Learning purposes. This process is explained further in section 2.3 in which we explore how we have decided to represent chess positions as inputs for the Artificial Neural Networks. More details about the development of the *Datasets*, such as the creation of the labels, are explained in section 2.4 and 2.5. We conclude this chapter with section 2.6 where we present into detail the 4 different *Datasets* that have been used for the experiments presented in chapters 6 and 7.

2.1 Board Evaluation

Gaining a precise understanding of a board position is a key element in chess. Despite what most people think, highly rated chess players do not differ from the lower rated ones in their ability to calculate a lot of moves ahead. On the contrary, what makes chess grandmasters so strong is their ability to understand which kind of board situation they are facing very quickly. According to these evaluations, they decide which chess lines to calculate and how many positions ahead they need to check, before committing to an actual move.
It is possible to identify a trade-off between the amount of future board states that need to be explored, and the precision of an evaluation of a current board state. In fact, if the evaluation of a particular chess position is very precise, there is no need to explore a large set of future board states. A very easy case is presented in Figure 2.1 where it is *Black's* turn to make a move.

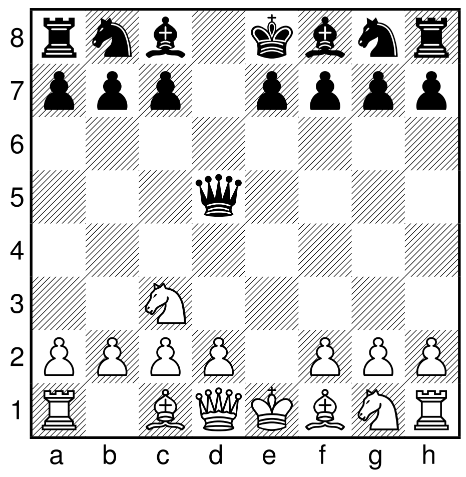

FIGURE 2.1: Example position that does almost not require any lookahead in order to get precisely evaluated.

As it is possible to see from Figure 2.1 *White* is threatening *Blacks's Queen* with the *Knight* in c3. Even an amateur player knows that the *Queen* is the most valuable piece on the board and that it is the piece type that after the *King*, deserves the most attention. This makes the evaluation process of the position just presented very easy, *Black* needs to move its *Queen* on a square in which it will not be threatened by *White's* pieces anymore. Besides being very easy to evaluate, the evaluation of the position itself is very precise as well, in fact *Black* does not have to invest time in calculating long and complicated chess lines in order to understand that, if it does not move its *Queen* to a safe square, the game will be lost very soon.
Chess grandmasters are very good at evaluating a way larger set of chess positions that are usually more complicated than the one just presented, but most of the time they only rely on lookahead in order to check if their initial evaluations that are based on intuition, are actually correct. By doing so they are sure to minimize the chances of making a *Losing* move.
The main aim of this work is to teach a system to evaluate chess positions very precisely without having to rely on expensive explorations of future board states that make use of lookahead algorithms. To do so, we model this particular way of training as a classification task and as a regression one. In both cases different Artificial Neural Network (ANN) architectures need to be able to evaluate board positions that have been scored by *Stockfish*, one of the most powerful and well known chess engines (Romstad et al., 2011). The chess positions that we use come from a broad database of games played between 1996 and 2016 by players with an Elo rating > 2000 and chess engines with a rating > 3000. Out of these games we have extracted more than 3,000,000 positions that are used for 4 different experiments. Considering the classification task, the experiments mainly differ according to the amount of labels that are used, namely 3, 15 and 20. On the other hand the regression experiments do not make use of any categorical labels but investigate the capabilities that ANNs have as mathematical function approximators, by attempting to approximate *Stockfish's* evaluation. The creation of the *Datasets* will now be described.

2.2 Games Collection

In order to perform the classification and regression experiments a large quantity of games to learn from is required. However, besides having a lot of potential positions, a second very important aspect has to be taken into account: the positions need to be played by highly ranked players. The reason of this decision is twofold: the first one is related to the fact that the system will not likely have to deal with completely random board states while it plays real games, while the second one deals with training time constraints. Including non-informative positions in the dataset will only increase the amount of computer power and time required to learn the evaluation skill. The latter reason is particularly important since one of the main aims of this work is to train a system with as much chess knowledge as possible in a reasonable amount of time.
While on one hand gathering chess positions is fairly easy, finding appropriate labels turned out to be way more complicated. In fact no such datasets exist. As a consequence, we have created a database of games collecting positions played both by humans and from chess engines. For the first case we have made use of the Fics Games Database [1]. A collection of games played by highly ranked players between 1996 and 2016, with > 2000 ELO points. In order to have a greater variety of positions, games played with different time control settings have been used. However, to ensure a higher quality of the board positions, $\approx 75\%$ of the games were played with the standard Bronstein/Fischer setting. In addition to these positions, a set of games played by the top chess engines with an ELO ranking of ≈ 3000 have been

[1]http://www.ficsgames.org/download.html

added to the dataset as well.
The final dataset consists of $\approx 85\%$ human played positions, while for the other $\approx 15\%$ it consists of chess engine played games, for a total amount of over 3 million different board positions.
Both games collections were presented in the Portable Game Notation (PGN) format, where each move is represented in the chess algebraic notation. This makes it possible to keep track whether a piece is present on the board or not. An example of a short, but valuable game marked in PGN format, can be seen in Figure 2.2

1 e4 e6 2 d4 d5 3 ♘c3 ♝b4 4 ♝d3 ♝×c3+ 5 b×c3 h6 6 ♝a3 ♘d7 7 ♛e2 d×e4 8 ♝×e4 ♘gf6 9 ♝d3 b6 10 ♛×e6+ f×e6 11 ♝g6# Z1-0

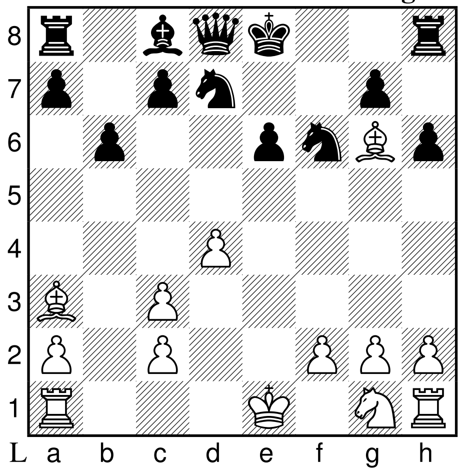

FIGURE 2.2: A miniature game played by the Russian champion Alexandre Alekhine in 1931. The Russian Grandmaster playing White managed to checkmate its opponent with a brilliant Queen Sacrifice after only 11 moves.

The games have been parsed and different board representations suitable for the ANNs have been created. This process is of high importance since it is in fact not possible to feed any machine learning algorithm by simply using the list of moves that have been made in one game. At the same time, it is very important to represent the chess positions in such a way that the information loss is minimized or even null.
The following section explains in detail how we have approached this task and which kind of input representations have been used in our research.

2.3 Board Representations

Literature suggests two main possible ways to represent board states without making use of too many handcrafted features. The first one is known as the *Bitmap Input* and represents all the 64 squares of the board through the use of 12 binary features (Thompson, 1996). Each of these features represents one particular chess piece and which side is moving it. A piece is marked with -1 if it belongs to *Black*, 0 when it is not present on that square and 1 when it belongs to *White*. The representation is a binary sequence of bits of length 768 that is able to represent the full chess position. There are in fact 12 different piece types and 64 total squares which results in 768 inputs. Figure 2.3 visualizes this technique. We successfully made use of this technique in all the experiments that we have performed, moreover, we took inspiration from it, in order to create a new input representation that we have called *Algebraic Input*. In this case we not only differentiate between the presence or absence of a piece, but also its value. Pawns are represented as 1, Bishops and Knights as 3, Rooks as 5, Queens as 9 and the Kings as 10. These values are negated for the *Black* pieces.

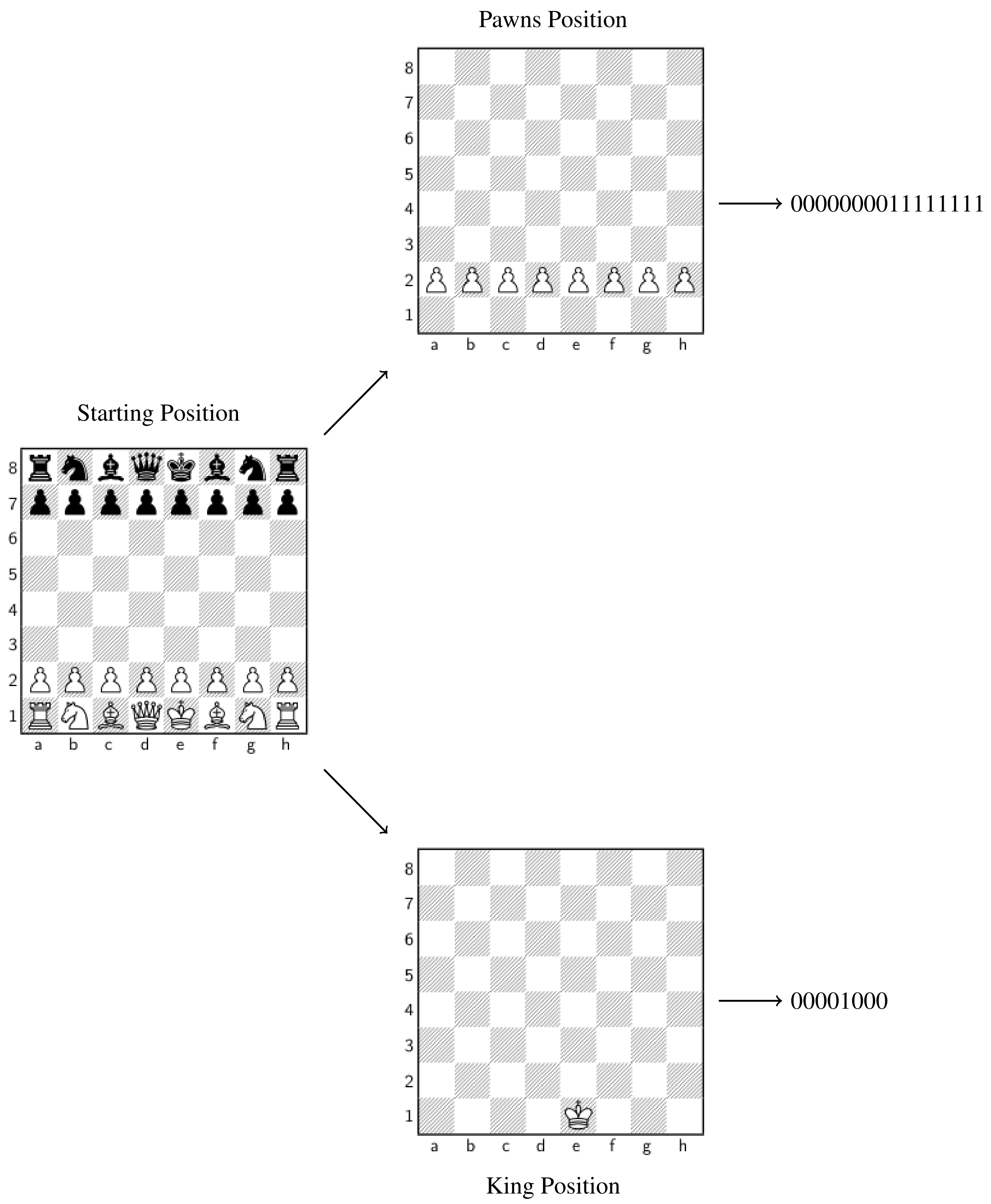

FIGURE 2.3: Bitmap Representation for the pawns and the king.

Another potential way to represent chess positions is the *Coordinate Representation*. This particular representation has been proposed by (Lai, 2015) and aims to encode every position as a list of pieces and their relative coordinates. The authors make use of a slot system that reserves a particular amount of slots according to how many pieces are present on the board. In the starting position the first two slots are reserved for the *King*, the next two for

the *Queens* and so on until every piece type is covered. In addition to that, extra information about each piece is encoded as well: e.g. whether it is defended or not, or how many squares it can cover in every direction. According to the authors, the main advantage of this approach is the capability of labeling positions that are very close to each other in the feature space in a more consistent way.
We did not directly test the latter representation, but we have taken inspiration from the idea of adding some extra informative features as inputs to the ANNs. We call this representation the *Features Input* and we explain it in more detail in Chapter 6. All three input representations, namely the *Bitmap*, the *Algebraic* and the *Features*, have been used as inputs both for Multilayer Perceptrons (MLPs) and for Convolutional Neural Networks (CNNs), two types of ANN architectures that will be explained in depth in Chapter 3 and Chapter 4.
Once these board representations have been created, we still need to assign every chess position a label that makes it possible to perform the classification and regression experiments. In order to do so we have used *Stockfish*, one of the most powerful chess engines that has the main benefit of being open source and compatible with `Python`, thanks to the use of the `Python Chess` library [2]. The way *Stockfish* evaluates chess positions will now be described.

2.4 Stockfish

Released under the GPL license, *Stockfish* is one of the strongest open source chess engines in the world. In fact, according to the *International Computer Chess Ranking List* (CCRL) it is ranked first in the list of computer engines playing chess [3]. *Stockfish* evaluates chess positions based on a combination of five different features and a lookahead algorithm.
The most important features are:

1. Material Balance: this is probably the most intuitive and easy to understand feature of the list. In fact, most of the time, equal positions present the exact same amount of pieces on the board. On the other hand, if one player has more pieces than the opponent he/she very likely has an advantage that makes it possible to win the game.

2. Pawn Structure: despite what most naive players think, pawns are very powerful pieces on the board. Their importance increases over time until the endgame is reached, where, together with the King, they can decide the final outcome of a game. Stockfish gives lower evaluations if the pawns are doubled, undefended, don't control the central squares of the board, or have low chances of getting promoted.

3. Piece Placement: the position of the pieces related to how many squares they are controlling is a very important concept, especially in the middle-game. Cases that increase the winning chances are for example the Bishops controlling large diagonals over the board, Knights covering the most central squares and the Rooks attacking the ranks close to the opposite King.

4. Passed Pawns: pawns have the ability to get promoted to higher valued pieces if they reach the opposite side of the board and as a consequence can lead to winning positions. Pawns that do not have any opposing pawns able to prevent them from advancing to the eighth rank improve Stockfish's evaluation score because they have higher chances to promote. These chances become even higher if the opponent's King is very distant.

[2] `https://pypi.python.org/pypi/python-chess`
[3] `http://www.computerchess.org.uk/ccrl/404/`

5. King Safety: since the main objective of chess is to checkmate the opponent's King it is very important that this particular piece is as safe as possible. Stockfish gives priority to castling and to all the pieces that block the opponent from attacking the King directly.

Figure 2.4 represents 4 different chess positions in which a different *Stockfish* feature has a high impact on the evaluation of the engine. We discard the first feature related to the material balance since it is very intuitive and easy to understand.

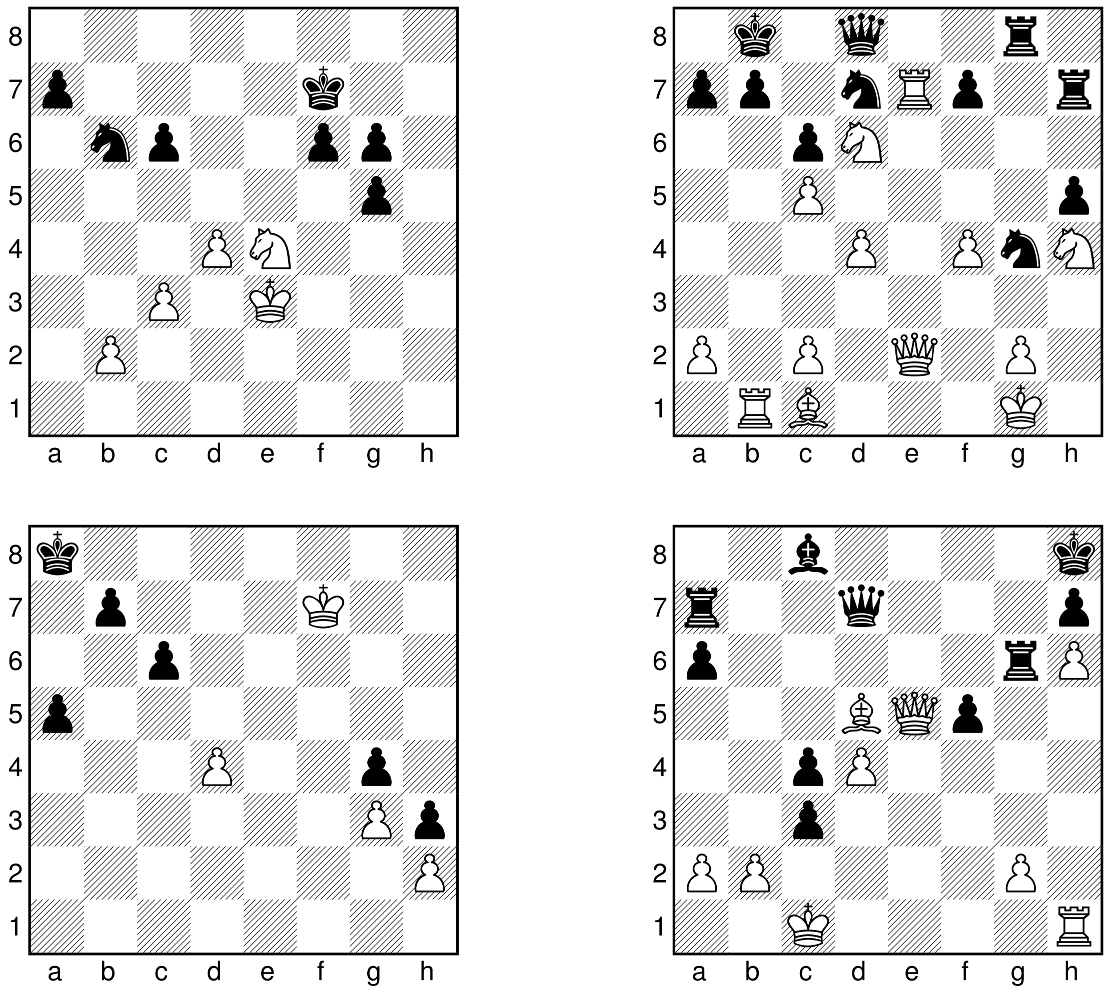

FIGURE 2.4: From left-up to bottom-right the set of Stockfish's last 4 most important features. The first position represents a bad pawn structure for the Black player who has both an isolated pawn and a doubled pawn. The second position highlights how well White's pieces are placed on the board and how they are attacking the area close to Black's King. In the third position we show an example of a passed pawn in a5 which will soon promote to Queen. The final position represents a strong attack from the White player which is checking with its Queen Black's very unsafe King.

The 5 features just presented are the most important ones. However, chess can become incredibly complex and a more precise evaluation can only be reached through the use of lookahead. It is in fact possible to have a highly unsafe king and at the same time threaten mate thanks to a particular high mobility of the pieces. In order to evaluate these particular conditions very precisely, *Stockfish* uses the lookahead algorithm known as $\alpha - \beta$ pruning. Based on the simple MinMax rule it is able to explore $\approx$ 30 nodes deep in one minute, in the tree of possible moves and discard the ones that, based on a counter move, lead to *Losing* positions. We now explain in depth how this particular algorithm works.

2.5 Move Search

Despite of what naive people think chess is still an unsolved game. A game is defined as solved, if given any legal board situation it is possible to predict if the game will end up with a win, draw or loss by assuming that both players will play the optimal sequence of moves. Right now, no matter how much computer power is used it is still impossible to predict this output. It is true that, for example, *White* has a slight advantage in the game after having done the first move, but if this is enough to win the whole game is still unknown. On the other side, an example of a solved board game is the English version of *Checkers*, in 2007 it has in fact been proved that if both players play optimally, all games will end up in a draw (Schaeffer et al., 2007). It is also worth mentioning that chess played on a $n \times n$ board is even a `EXPTIME-hard` problem which puts serious constraints in the *chess-programming* domain (Papadimitriou, 2003).
Keeping this in mind it turns out that it is not very interesting to explore algorithms that allow to search deeper and deeper during the move search process, since no matter the depth of this search, it will still be impossible to reach optimal play. It is far more challenging to understand which kinds of board states deserve a deep exploration and which ones are not worth analyzing. The challenge can be expanded even further by trying to train a system in such a way that is possesses the ability of the most powerful lookahead algorithms, while at the same time not making any direct use of them.
The lookahead procedure can be formalized as follows: we denote with S all the possible board positions in the game and with $t = 1, 2, \dots$ the turns in which a move has been made on the board. At each turn t there is a corresponding board position $x_t \in S$, from where the player can choose a move $m \in M_t$ that leads to a new board state x_{t+1}. The main idea is to find the sequence of moves that maximizes the chances of winning the game. In this work we aim to train an Artificial Neural Network that is able to include this whole procedure in its evaluation function without concretely searching the tree of possible moves.

2.5.1 MinMax Search & Alpha-Beta Pruning

Chess is defined as a zero-sum game, which means that the loss of one player is the other player's gain (Eisert, Wilkens, and Lewenstein, 1999). Both players choose their moves, $m \in M_t$, with the aim of maximizing their own chances of winning. By doing so they minimize at the same time the winning chances of their opponent. MinMax is an algorithm for choosing the set of $m \in M_t$ that leads to the desired ending situation of a game, which in chess corresponds to a mating position. This is achieved by generating S until all terminal states are reached. Once this has been done, an evaluation function is used to determine the value of every board state, i.e. a winning board state would have a value of 1. The same utility function is then applied recursively to the board states x_{t-1} until the top of the tree is reached. Once a value $\forall x_t \in S$ has been assigned it is possible to choose the sequence of moves that according to the evaluation function leads to the win. It is theoretically possible to use MinMax in chess, however due to the previously mentioned computational complexity issues this is not feasible. MinMax is in fact a depth-first search algorithm that has a complexity of $\vartheta(b^d)$, with b being the branching factor and d corresponding to the depth of the search.
In order to deal with this issue the $\alpha - \beta$ pruning algorithm can be used. Proposed by (Knuth and Moore, 1975), this technique is able to compute the same decisions as MinMax but without looking at every node in the tree search. This is achieved by discarding the $x_t \in S$ that are not relevant for the final decision of picking $m \in M_t$. According to (Russell and Norvig, 1995) the general principle is the following: let us consider a random node n_t, at a random turn t in the tree of possible moves; if the chess player already has the possibility of reaching

a better $m \in M_t$ (with $M_t \in n_t$) while being at a n_{t-1} or even further up, it is possible to mark n as not worth exploring. As a consequence n and its descendants can be pruned from the tree of moves. The Pseudocode of this algorithm is presented hereafter:

Algorithm 1 $\alpha - \beta$ Pruning

```
1: function MAXVALUE($x_t$, $X_{t-1}$, $\alpha$,$\beta$)
2:     if PruningTest $x_t$ then
3:         return Eval $x_t$
4:     end if
5:     for $x_t \in S_{t+1}$ do $\alpha \leftarrow$ MAX($\alpha$, MinValue, $x_t$, $X_{t-1}$, $\alpha$, $\beta$)
6:         if $\alpha \geq \beta$ then
7:             return $\beta$
8:         end if
9:     end for
10:    return $\alpha$
11: end function
12: function MINVALUE($x_t$, $X_{t-1}$, $\alpha$,$\beta$)
13:     if PruningTest $x_t$ then
14:         return Eval $x_t$
15:     end if
16:     for $x_t \in S_{t+1}$ do $\beta \leftarrow$ MIN($\beta$, MaxValue, $x_t$, $X_{t-1}$, $\alpha$, $\beta$)
17:         if $\beta \leq \alpha$ then
18:             return $\alpha$
19:         end if
20:     end for
21:     return $\beta$
22: end function
```

The $\alpha - \beta$ pruning algorithm provides a significant improvement from a computational complexity perspective when compared to MinMax, in the optimal case its complexity is in fact of $\vartheta(b^{d/2})$, which allows the algorithm to look ahead twice as far as MinMax for the same cost. However, despite this significant improvement, $\alpha - \beta$ tree search is not enough to solve the game of chess. But it is still very suitable for *chess-programming*. The trick consists in marking $\forall x_{t=d} \in S$ as terminal states and apply the MinMax rule on those particular board states to limit the depth of the search. The higher the value of d, the more computationally demanding the tree search will be.

2.6 Datasets

Now that we have explained how *Stockfish* evaluates chess positions it is possible to explain what the output of this process is. *Stockfish* outputs its evaluations with a value called the fractional centipawn (*cp*). Centipawns correspond to 1/100th of a pawn and are the most commonly used method when it comes to board evaluations. As already introduced previously, with the explanation of the *Algebraic Input*, it is possible to represent chess pieces with different integers according to their different values. When *Stockfish's* evaluation output is a value of $+1$ for the moving side, it means that the moving side is one pawn up or that it will win a pawn in one of the coming plys.
Stockfish is able to explore ≈ 30 nodes deep in the tree of possible moves and discard the ones that, based on a counter move, lead to *Losing* positions. However, it is worth mentioning that the exploration of ≈ 30 nodes can be computationally expensive and, especially for

complex positions, lead to an evaluation process of over one minute long. As a consequence, in order to create the *Datasets* in a reasonable amount of time, we have set the amount of explored nodes to depth 8. We then use the different *cp* values to create 4 different datasets. The first 3 have been used for the classification experiments, while the fourth one is used for the regression experiment.
The datasets will now be described.

- *Dataset 1*: This dataset is created for a very basic classification task that aims to classify only 3 different labels. Every board position has been labeled as *Winning*, *Losing* or *Draw* according to the *cp Stockfish* assigns to it. A label of *Winning* has been assigned if $cp > 1.5$, *Losing* if it was < -1.5 and *Draw* if the *cp* evaluation was between these 2 values. We have decided to set this *Winning/Losing* threshold value to 1.5 based on chess theory. In fact, a *cp* evaluation > 1.5 is already enough to win a game (with particular exceptions), and is an advantage that most grandmasters are able to convert into a win.

- *Dataset 2 and Dataset 3*: These datasets consist of many more labels when compared to the previous one. *Dataset 2* consists of 15 different labels that have been created as follows: each time the *cp* evaluation increases with 1 starting from 1.5, a new winning label has been assigned. The same has been done if the *cp* decreases with 1 when starting from -1.5. In total we obtain 7 different labels corresponding to *Winning* positions, 7 labels for *Losing* ones and a final *Draw* label as already present in the previous dataset. Considering *Dataset 3*, we have expanded the amount of labels relative to the *Draw* class. In this case each time the *cp* evaluation increases with 0.5 starting from -1.5 a new *Draw* label is created. We keep the *Winning* and *Losing* labels the same as in *Dataset 2* for a total of 20 labels.

- *Dataset 4*: For this dataset no categorical labels are used. In fact to every board position the target value is the *cp* value given by *Stockfish*. However we have normalized all these values to be in $[0,1]$. Since ANNs, and in particular MLPs, are well known as universal approximators of any mathematical function we have used this dataset to train both an MLP and a CNN in such a way that they are able to reproduce *Stockfish's* evaluations as accurately as possible.

Chapter 3

Multilayer Perceptrons

In this chapter we first cover the perceptron, the first and most simple type of Artificial Neural Network (ANN). When stacked together, perceptrons give rise to Multilayer Perceptrons (MLPs), an ANN architecture that has successfully been used in this work. Due to their ability to generalize data so well, and their capabilities as mathematical function approximators, MLPs have been used successfully in a broad range of machine learning tasks that go from game playing (Tesauro, 1990), to automatic phoneme generation (Park, Kim, and Chung, 1999), and even protein analysis (Rost and Sander, 1994).
In this work, MLPs have been used in order to find a very good chess evaluation function based on the *Datasets* of chess games that have been presented in Chapter 2.
The structure of this Chapter is as follows: in section 3.1 we introduce the concept of Artificial Neural Network by focusing on the architecture of the perceptron. In section 3.2 we explore the importance of non-linear functions as activation functions for the ANNs and present the ones that are most commonly used in literature. We end the chapter with section 4.3 where we explain what it means to train an ANN, how this procedure works and which problems can be encountered during this process.

3.1 Artificial Neural Networks

Despite having become popular only after the 80's, the concept of Artificial Neural Networks (ANN) is older than expected. The main pioneer in this field can be considered Frank Rosenblatt, an American psychologist who, inspired by work (McCulloch and Pitts, 1943) on the exploration of the computational capabilities of single neurons, developed the perceptron, the simplest form of ANN (Rosenblatt, 1958). Perceptrons are a very simple binary classification algorithm that map different inputs to a particular output value. Every input is associated a weight, and the weighted sum of all the inputs is calculated as

$$s = \sum_{i=1}^{d} w_i \cdot x_i.$$

The result s is then passed through an activation function and according to its result a classification can be made:

$$f(s) = \begin{cases} 1 & \text{if } s \geq 0 \\ 0 & \text{otherwise.} \end{cases}$$

Since it is very unlikely that the perceptron classifies the inputs correctly from the start, an error function is used in order to change the weights of the ANN. The error function checks in fact how different the predicted output of the ANN is when compared to the desired one. We define with n the total number of input samples of the ANN, while with y their target output. $f(s)$ again corresponds to the output of the artificial neuron. The error function is now defined as

$$E = \frac{1}{2}\sum_{i=1}^{n}\big(y_i - f(s_i)\big)^2. \tag{3.1}$$

By multiplying the derivative of E with respect to the weights and the learning rate η, new weights can be assigned to the artificial neurons and better predictions can be made in the future:

$$\Delta w_i = \eta\big(y_i - f(s_i)\big)\mathbf{x}. \tag{3.2}$$

Figure 3.1 shows the structure of a perceptron with 3 input units and 1 output.

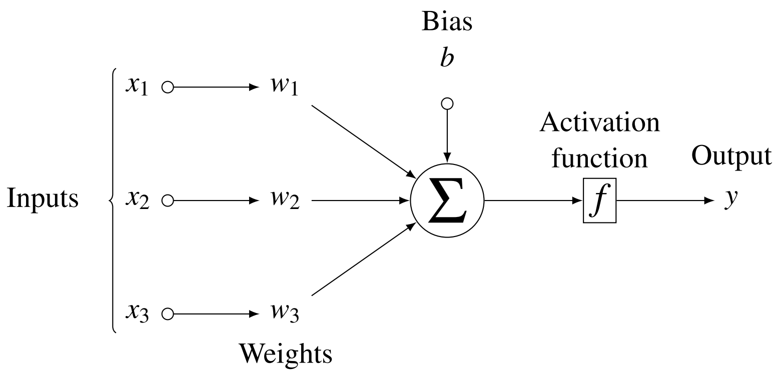

FIGURE 3.1: Example of a simple perceptron, with 3 input units and 1 output unit.

Single layer perceptrons can only be used for a limited amount of problems though, and as a consequence they are not suitable for more complex classification or regression tasks that require non linear separation boundaries. However, it is possible to create a sequence of individual perceptrons in order to create the so called multilayer perceptrons (Baum, 1988). The extra processing elements that are added between the input units and the output ones are defined as hidden layers, the main difference between a multilayer perceptron and its simplified version is the relation between the input and the output. In this case, we can define this relation as a nested composition of nonlinearities in the form

$$y = f(\textstyle\sum f(\sum(\bullet))). \tag{3.3}$$

The amount of function compositions is given by the number of network layers. The next section shows into detail the concept of nonlinearity.

3.2 Non Linearity Functions

As already mentioned, most classification tasks cannot be learned through the use of a single linear combination of the features. However, this is only part of the reasons why MLPs require a nonlinear activation function. It would in fact be possible to argue that, since MLPs consist out of different layers, each layer could be activated separately by a linear function and solve part of the classification problem. However this is not true, the summation of the different layers would only give another linear function. This can be easily seen with the following proof that considers an MLP with 2 layers. $f^n(x)$ denotes the activation function at layer n and in_k and out_k the relative inputs and outputs:

$$out_k^{(2)} = f^{(2)}\left(\sum_j out_j^{(1)} \cdot w_{jk}^{(2)}\right).$$

This is equal to:

$$f^{(2)} = \left(\sum_j f^{(1)}\left(\sum_i in_i w_{ij}^{(1)}\right) \cdot w_{jk}^{(2)}\right).$$

If we then assume that the activations of the hidden layers are linear, like $f^{(1)}(x) = x$, we obtain:

$$out_k^2 = f^{(2)}\left(\sum_i in_i \cdot \left(\sum_j w_{ij}^{(1)} w_{jk}^{(2)}\right)\right).$$

This output is equivalent to a single layer perceptron with the following weights:

$$w_{ik} = \sum_j w_{ij}^1 w_{jk}^{(2)}$$

that is in fact not able to solve a non linear separable problem.
As a consequence, non linearly separable tasks can only be solved through the use of non-linear activation functions. The most important type of non-linear functions are the standard logistic ones, that, applied to an input vector X, are defined as follows:

$$z_h = \frac{1}{1 + exp\left[-\left(\sum_{j=1}^{d} w_{hj}x_j + w_{h0}\right)\right]}. \tag{3.4}$$

There is a broad range of activation functions to choose from when it comes to ANNs. Hereafter a list of the ones that have been used in Chapters 4 and 5, together with their graphical representation is presented.

- *Rectified Linear Unit* known as ReLU and defined as:

  $$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geqslant 0 \end{cases}$$

  with a range of $[0, \infty)$ It is probably the most commonly used rectified activation function. The reason of this is twofold: it is the function that is most similar to the biological processes that occur in the brain, and it makes it possible to train Deep Neural Networks much faster (Wan et al., 2013).

- *Tanh*: Defined as:

  $$tanh(x) = 2\sigma(2x) - 1$$

  where $\sigma(x)$ is the function presented in equation 3.4. Differently from the ReLU, the range of $tanh(x)$ is in $[-1, 1]$.

- *Exponential Linear Unit* known as *Elu* and defined as:

  $$f(\alpha, x) = \begin{cases} \alpha x & \text{for } x < 0 \\ x & \text{for } x \geqslant 0 \end{cases}$$

is a variant of the ReLU $f(x)$ that has been specifically designed for facing the Vanishing Gradient problem. It has in fact a range $(-\infty, \infty)$ and the capability of pushing the mean unit activations closer to zero. As a consequence, the gradients get closer to the unit natural gradient and learning becomes faster.

Figure 3.2 shows a visual representation of the just presented activation functions.

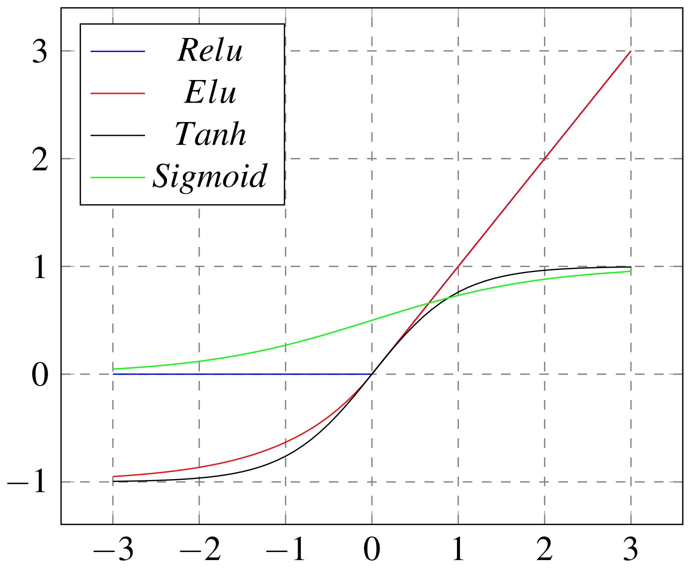

FIGURE 3.2: Graphical representation of the most common activation functions in the [-3, 3] range.

3.3 Training Procedure

As introduced at the beginning of this Chapter, training an ANN means adjusting its weights in such a way that the error function 3.1 is minimal. This adjustment is done via the use of the backpropagation algorithm. First introduced in the 70's, this particular algorithm only became popular and extensively used after the publication of (Rumelhart, Hinton, and Williams, 1986). This work happened to be a breakthrough in the scientific community since it reported the first proof of how backpropagation was able to outperform the back then standard perceptron-convergence procedure in several complicated tasks.
The core point of backpropagation relies in the calculation of the partial derivatives (or gradients) with respect to the error function from equation 3.1, defined as:

$$\frac{\partial E}{\partial w} \tag{3.5}$$

for the weights of the network, and

$$\frac{\partial E}{\partial b} \tag{3.6}$$

for its bias.

By combining these two expressions it is possible to compute how the output of equation 3.1 changes in relation to the weights and biases of the network for every training example. In fact, it is possible to know the activations of individual neurons for each layer of the network and how much they affect the output of the error function 3.1 by applying:

$$\partial w = \left(\frac{\partial E}{\partial w_1} \frac{\partial E}{\partial w_2} \cdots \frac{\partial E}{\partial w_n} \right). \tag{3.7}$$

The final step is the actual update of the weights. The weights are changed proportionally to the negative of the derivative of the error as

$$\Delta w_i = -\eta \left(\frac{\partial E}{\partial w_i} \right).$$

By being able to iteratively change the weights of the network, at some point it is possible to obtain ΔE close to 0 and satisfy the main idea of minimizing a cost function.

3.3.1 Stochastic Gradient Descent

Computing the previously mentioned derivatives can be very expensive in the case of deep MLPs that need to be trained on a large amount of data. As a consequence, some simplified variations have been proposed in the years. The most famous of them is called Stochastic Gradient Descent (SGD). The original gradient descent method is defined as a batch algorithm, which means that the adjustment of the weights is computed only after having analyzed all the samples of the dataset. It is mathematically proven that it is always able to find a solution to the optimization problem that is considered, however its main drawback is the time required to get to the desired solution.
SGD provides an alternative to Batch Gradient Descent by avoiding the computation of the exact value of all the gradients. Instead, the estimation of Δw is computed on the basis of randomly picked samples of the input, also known as batches, that give this algorithm the property of stochasticity (Bottou, 2010). However, the price that has to be paid for this approximation is a trade-off between the amount of time the algorithm requires to converge, and how well the cost function is minimized. The latter property is in fact directly proportionate to how much the real value of Δw is approximated.
The convergence of the algorithm can be improved in several ways, it is in fact possible that the algorithm thinks it has minimized the error function 3.1 but it is actually blocked in what is called a *Local Minimum*. It is also possible that the algorithm is actually adjusting the weights of the ANN towards the correct direction, but the speed of this process is extremely slow. We now briefly describe how the performance of SGD can be improved while training deep ANN architectures.

- Nesterov Momentum: Introduced by (Polyak, 1964), Momentum is able to accelerate the gradient descent while the minimization process is persistent during training. For t_{t+1} we define the Momentum as follows:

  $$\Delta w_{t+1} = \mu_t \cdot \Delta w_t - \eta \frac{\partial E}{\partial w_t}$$

  where $\mu \in$[0,1] is the momentum rate and t an update vector that keeps track of the gradient descent acceleration. The higher the value of μ is, the quicker the algorithm will converge, however the risk of this, especially if combined with a high η, is that the algorithm might become very unstable.

- Adagrad: Analyzed in detail by (Neyshabur, Salakhutdinov, and Srebro (2015)) has the ability of adapting the learning rate with respect to all the different parameters

(θ) of the ANN. This results in larger updates for the least updated parameters, while similarly smaller updates are performed on the most updated ones. This adaptive way of changing the learning rate is done at each time step t as follows:

$$\theta_{t+1} = \theta_t \frac{\eta}{\delta^2 + \varepsilon} \odot \partial_{t,w}.$$

η corresponds again to the learning rate, δ is the collection of all the individual gradients of the parameters until that particular time step. ε is a constant that is added in order to avoid particular cases that result in a division by 0. Despite having been successfully used by (Dean et al., 2012), in (Duchi, Jordan, and McMahan, 2013) the authors show how the constant accumulation of the gradients in the denominator of the algorithm, eventually leads to very large numbers. As a consequence the update of the learning rate can result in very small values which influences the training procedure negatively.

- Adadelta: proposed by (Zeiler, 2012), is largely inspired by the Adagrad algorithm and both algorithms work very similarly. However, Adadelta has been specifically developed to obstruct the exponentially large growth of δ. In this case instead of keeping track of all the previous gradients, Adadelta only focuses on the ones that match a particular time window, and replaces δ with its moving average value $\mu(\delta^2)_t$ which is multiplied by γ, a constant usually set to 0.9.
  Adadelta is very similar to the RMSProp optimizer proposed by (Tieleman and Hinton, 2012). In fact both techniques do not keep track of the whole sum of the squared gradients but only make use of the most recent ones.

- Adam: introduced by (Kingma and Ba, 2014) the Adam optimizer can be seen as a combination of Nesterov Momentum and Adadelta. In fact, Adam also makes use of adaptive learning rates for each parameter of the ANN, this is again done by storing an exponentially decaying average of the past squared gradients as done by Adadelta, but in addition to that a second similar parameter is added to the algorithm. The first one, defined as $m_t = \beta_1 m_{t-1}(1-\beta_1)\partial_{t,w}$ is an estimate of the first moment (mean), while the second one, defined as $v_t = \beta_2 v_{t-1}(1-\beta_2)\partial^2_{t,w}$ corresponds to the second moment (uncentered variance) (Radford, Metz, and Chintala, 2015). m_t and v_t are two vectors that are biased towards 0, hence they are corrected by computing $\hat{m} = \frac{m_t}{1-\beta_1^t}$ and $\hat{v} = \frac{v_t}{1-\beta_2^t}$ with β_1 set to 0.9 and β_2 to 0.99.
  The final update consists in computing:

$$\theta_{t+1} = \theta_t \frac{\eta}{\sqrt{\hat{v}_t} + \varepsilon} \hat{m}_t.$$

3.3.2 Vanishing Gradient Problem

Even though training MLPs with the previously mentioned methods is quite successful, there is still an important issue that can be encountered: the vanishing gradient problem. First introduced by (Hochreiter, 1998), it is a particular difficulty of training the bottom layers of MLPs with gradient based methods. As already explained, the main idea of backpropagation is to understand the impact that the changes on the network's weights have on its relative outputs. This works really well if the impact is quite big, however it is possible that a big change in the weights leads to a relatively small change in the predictions made by the MLP. The consequence of this inversely proportional phenomenon is that the MLP will not be able to properly adjust the weights of particular features, and as a consequence never learn. Moreover, if this already happens in the first layers, the output, which depends on them, will be

built on these inaccuracies and make the ANN “corrupted”.
Hochreiter identified that the vanishing gradient problem is strongly related to the choice of the nonlinear activation functions explained in section 3.2. Some of them are more susceptible to this issue than others and a good example of this phenomenon can be seen if we consider the sigmoid function:

$$f(x) = \frac{1}{1+e^{-x}}$$

and its relative derivative:

$$f^{'}(x) = \frac{1}{1+e^{-x}}\left[1 - \frac{1}{1+e^{-x}}\right] \tag{3.8}$$

If we compute the maximum value of equation 3.8 we obtain 0.25, which is quite low and corresponds to the start of the vanishing gradient problem. In fact we know that the outputs of the units of the MLP all contain the derivative of the sigmoid function, and that they are all multiplied by each other when computing the error of the ANN deeper down the architecture. If the range of the derivative is already very small as shown by the first order derivative of the sigmoid, this will lead to exponentially lower values that will make the bottom layers of the ANN very hard to train.
In (Hochreiter, 1998) the author shows how the Vanishing Gradient problem is not only related to MLPs but also to Recurrent Neural Networks (RNNs), a particular type of ANNs with cyclic connections, these connections enable the ANN to maintain an internal state and be very suitable for processing sequences of different lengths. The Vanishing Gradient issue is solved in (Hochreiter and Schmidhuber, 1997) where the authors introduce Long Short Term Memory nodes that are able to preserve long lasting dependencies in the ANN. Since the introduction of these cells the Vanishing Gradient problem has been largely solved (Sutskever, Vinyals, and Le, 2014) and has led to successful applications of both very deep MLPs (He et al., 2016) and RNNs (Shkarupa, Mencis, and Sabatelli, 2016).

Chapter 4

Convolutional Neural Networks

This chapter covers Convolutional Neural Networks (CNNs), a type of Artificial Neural Network (ANN) that takes inspiration from *receptive fields*, a particular type of cells that are present in the visual cortex of the brain (Lawrence et al., 1997).
Compared to Multilayer Perceptrons (MLPs), CNNs started to become popular later, only after work (Krizhevsky, Sutskever, and Hinton, 2012) was published. However, since then, this kind of ANN architecture started to outperform all other computer vision algorithms in various image recognition tasks, establishing themselves as the current *State of the Art* technique for such problems (Simonyan and Zisserman, 2014).
Similarly to what has been introduced in chapter 3, we have investigated the capabilities of CNNs to recognize different kinds of board positions and hence, perform as evaluation function of our chess system. The reason that has driven the exploration of their capabilities is twofold: on the one hand, as has already been introduced in chapter 1, CNNs have barely been used in chess, and literature seems to be very skeptical about their potential in this particular context. On the other hand, since CNNs are well known for their performances on images together with their pattern recognition abilities, the choice of exploring this ANN was straightforward.
The structure of this chapter is as follows: in section 4.1 we explore the mathematical operation of “convolving” and how this is related to the representation of the chess board that has been used as input for the ANN. In section 4.2 we explore the geometrical operations that are related to the process of convolution and show how they are of particular importance when it comes to chess. We end the chapter with section 4.3 in which we explore how CNNs are trained and how this process partially differs from the training procedure of an MLP.

4.1 Convolutions & Kernels

The input of a CNN is usually a three dimensional array of pixels in which the first dimension represents the width of the picture, the second one its height and the third one its depth. This latter dimension corresponds to the amount of channels the picture has, i.e. binarized pictures have 1 channel since the pixels of the images can only be either black or white. On the other hand colored pictures, that are usually represented with the RGB color space, have 3, in which each channel represents one primary color.
As has been explained in chapter 2, in the specific case of chess positions, we explore the performances of in total 3 possible board representations. The first one, which is defined as *Bitmap Input* represents whether or not a particular piece is present on each square of the chess board, the second one, defined as *Algebraic Input* assigns a numerical value to each piece according to its strength, while the final third representation, defined as *Feature Input* adds particular chess features to the input and is explained in detail in chapter 7.
We have seen that the first two representations can be expressed with a feature vector of length 768 which is ideal for MLPs, however the dimensionality of this input changes when it comes to CNNs. In fact, this feature vector is reshaped into a $8 \times 8 \times 12$ tensor, where the

first 2 dimensions correspond to the size of the chess board and the latter channel represents the amount of feature layers of the input. In this case there is a total amount of 12 different layers that correspond to the different piece types that are present on the board [1]. We show a visual representation of the *Bitmap Input* as $8 \times 8 \times 12$ tensor in Figure 4.1, while Figure 4.3 shows an example of a chess position that is represented according to the *Algebraic Input*, in which each pixel of the image has a different grayscaled value according to its strength.

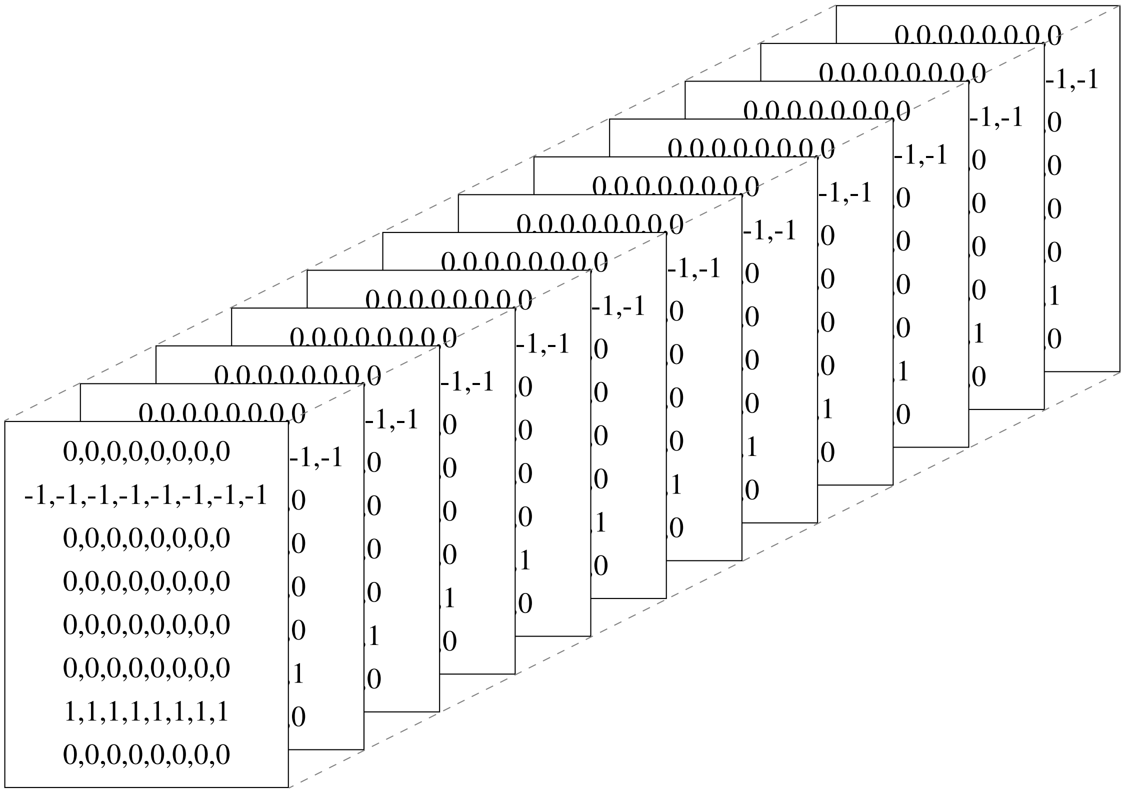

FIGURE 4.1: The representation of an initial chess board as $8 \times 8 \times 12$ Tensor.

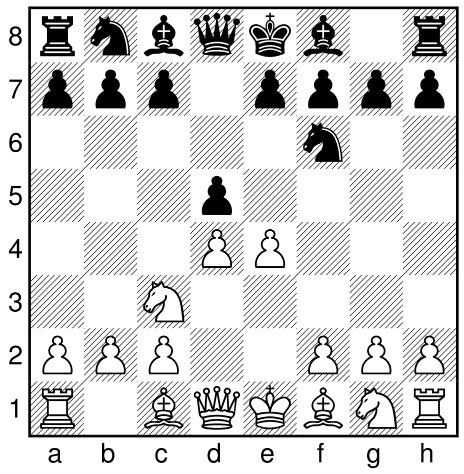

FIGURE 4.2: Original Position.

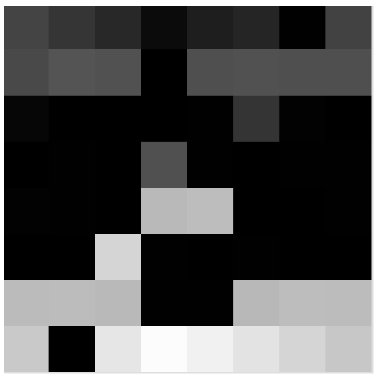

FIGURE 4.3: Image representing the *Algebraic Input*.

As their name already suggests, CNNs make use of the algebraic operation known as convolution. To do so they make use of kernels, small matrices that by sliding over the

[1]As will be explained in Chapter 7, the amount of channels is increased with the use of the *Feature Input*

picture have the ability to detect features in it. An example of this operation can be expressed as follows: let's assume we have a 2×2 kernel K and a $2D$ matrix M:

$$K = \left[\begin{array}{c|c} 2 & 2 \\ \hline 2 & 2 \end{array}\right]$$

$$M = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 3 & 4 & 5 & 7 \\ 4 & 6 & 8 & 1 \end{bmatrix}$$

The convolution operation consists in overlapping K on every pixel of M starting from 1 as shown hereafter:

$$\left[\begin{array}{c|c} 2 & 2 \\ \hline 2 & 2 \end{array}\right] \left[\begin{array}{cc} 1 & 2 \\ 3 & 4 \end{array}\right]$$

At the end of this overlapping procedure we simply obtain a single number corresponding to the following calculation $2 \cdot 1 + 2 \cdot 3 + 2 \cdot 2 + 2 \cdot 4 = 20$. The same procedure is applied recursively on every cell of the matrix, and basically on every pixel of the image, until the bottom right is reached. The output of this convolution operation is a new 3×2 matrix N:

$$\left[\begin{array}{c|c|c} 20 & 28 & 38 \\ \hline 34 & 46 & 42 \end{array}\right]$$

This example shows how convolution works in a 2 dimensional space, however as discussed by (Keys, 1981) for tensors of the third order or higher, this process is defined similarly. The only difference is that, in the case of third order tensors, a dimension is added and as a consequence the convolution starts in the $(0,0,0)$ location.
CNNs make use of a large amount of kernels that are mostly square shaped and of different sizes. The size of the kernel is directly related to the dimensions of the input image. In fact, in the case of chess positions of size 8×8 we have mostly made use of 5×5, 3×3 and 1×1 filters.
Showing the effect of these filters on such small images is very complicated and turns into very "confusing" pictures, however, in order to get an idea of which effects these filters have on images we show in Figure 4.4 an example of how 2 different types of kernels modify a RGB picture representing Bobby Fisher, the most famous American chess player. For the convolution operation the *Elu* activation function has been used.

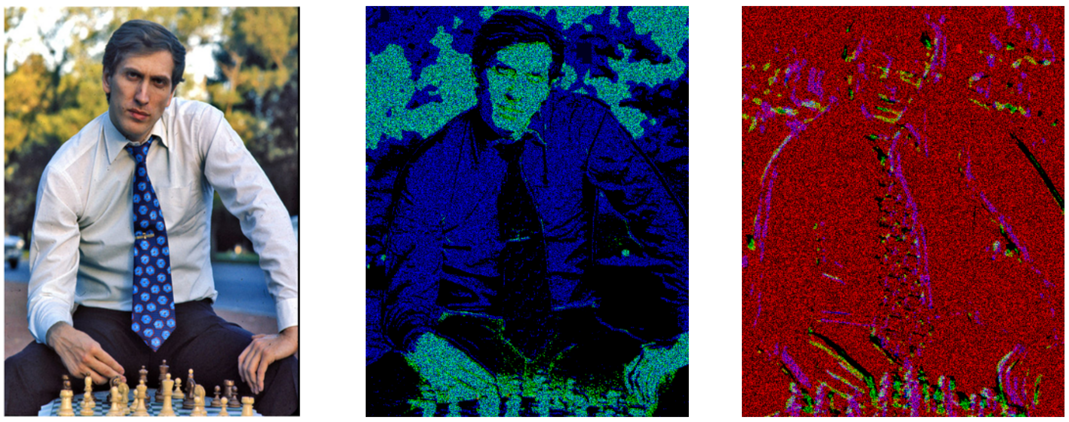

FIGURE 4.4: The effects of 5×5 and 3×3 kernels on a picture representing Bobby Fisher.

Now that the general process of convolution has been explained, we present in the next Section the geometrical implications of this operation and how this is related to the size and content of chess positions.

4.2 Geometrical Properties

As can be easily seen by the previous example, the output matrix N has a different size when compared to the original M one. This is related to the size of the kernel that is used, in fact if this size is $> 1 \times 1$ the spatial extent of the output matrix will always be smaller than the original one. However, in chess we are restricted by the need to have the convolved board representation of the exact same size as the original one. In fact, in order to classify if a position is *Winning* or not we need to see the entire board with all the pieces present on it. It is in fact not possible to give a precise evaluation of a chess board state with only partially looking at one particular area of the board.

4.2.1 Padding

A technique that is able to preserve the original geometrical properties of the picture is *Padding*. As already introduced, let's assume we have the input of the CNN represented as follows: $X^l \times W^l \times D^l$ with l being the $l-th$ layer of depth of the input. We know that the size of the kernel is $H \times W \times D^l \times D$ which as explained in (Wu, 2016) results in a convolution with the following size:

$$\left(H^l - H + 1\right) \times \left(W^l - W + 1\right) \times D.$$

The idea of padding is very simple. We fill the geometrical missing information of the picture as follows: we insert $\lfloor \frac{H-1}{2} \rfloor$ rows above the first one and $\lfloor \frac{H}{2} \rfloor$ under the last one. Furthermore, we also add $\lfloor \frac{W-1}{2} \rfloor$ columns next the first one on the left and $\lfloor \frac{W}{2} \rfloor$ next to the last one on the right. [2]
By doing so it is possible to obtain a picture which has the following size $H^l \times W^l \times D$ that actually corresponds to the original size of the input. This extra geometrical information is usually filled with $0s$ even though it is technically possible to use any number (Simonyan and Zisserman, 2014).

4.2.2 Pooling

The most powerful CNN architectures such as (Krizhevsky, Sutskever, and Hinton, 2012) make all use of a technique called *Pooling*. *Pooling* is a sampling technique that has the ability to reduce the dimensionality of the feature maps, while at the same time being able to preserve the most important information of the input. As presented by (Boureau, Ponce, and LeCun, 2010) CNNs make use of this technique in order to obtain more compact representations of images, hence, this makes CNNs more robust to noise and clutter.
There are 2 main different types of *Pooling* namely *Max Pooling* (Nagi et al., 2011) and *Average Pooling* (Hinton et al., 2012). Even though literature highlights the benefits of this technique especially when it comes to very large pictures, since it allows the CNNs to go extremely deep and explore relevant features in the images, we now show why both types of this technique cannot be used in chess.
The reason of this rejection is again related to the importance of the geometrical properties that the images representing chess positions need to preserve. In order to show how chess information gets lost when using *Pooling* let us consider the following matrix O on which we

[2] The $\lfloor\rfloor$ operator corresponds to the mathematical floor function.

apply a 2×2 filter with a stride size of 2. We represent the cells of the matrices on which this filtering is applied with different colors.

$$O = \begin{bmatrix} 1 & 1 & 2 & 4 \\ 5 & 6 & 7 & 8 \\ 3 & 2 & 1 & 0 \\ 1 & 2 & 3 & 4 \end{bmatrix}$$

What *Max Pooling* does is simply mapping a subregion out of the one on which the filters are applied to its maximum value. Similarly, the output of *Average Pooling* maps the subregion to the average of the elements present in the filtered window. The output of both *Pooling* types is represented by matrices Max_A and Av_A.

$$Max_A = \begin{bmatrix} 6 & 8 \\ 3 & 4 \end{bmatrix}$$

$$Av_A = \begin{bmatrix} 3.25 & 5.25 \\ 2 & 2 \end{bmatrix}$$

Even though comparisons made by (LeCun and Bengio, 1995) have shown that taking *Max Pooling* definitely outperforms *Average Pooling* and has become a standard added layer to most CNN architectures, it is not possible to make use of it when it comes to chess. In fact its effect can be devastating as, Figure 4.5 shows a position in which *Pooling* should be avoided together with the 8×8 *Bitmap Input* representation.

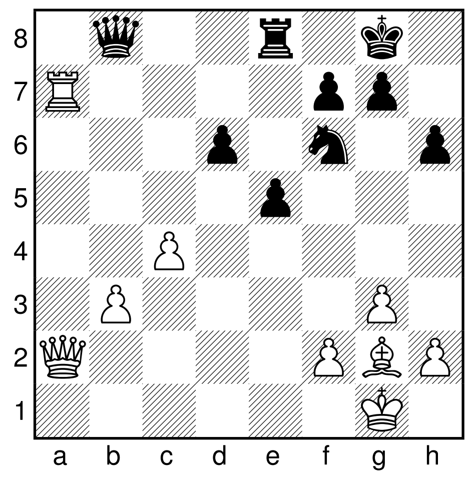

FIGURE 4.5: Example Position that shows the disadvantages of *Pooling*.

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

FIGURE 4.6: *Bitmap* representation for the CNN.

If we now assume that we use a 3×3 filter on the top left part of the image as marked by the red box, and perform an *Average Pooling* operation on the area

$$\begin{array}{cccc} & \Downarrow & & \\ \end{array}$$
$$\left(\begin{array}{|ccc|c} \hline 0 & -1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 \\ \hline 0 & 0 & 0 & 0 \end{array}\right)$$

we would obtain a value of 0 for the region of interest, which could be possibly interpreted by the CNN as a part of the chess board without any relevant information. However,

this would not match with what the chess position actually represents. In fact, that area of the board is in total control of *White* thanks to the presence of the *Queen* in `a2` and the *Bishop* in `g2`.
Pooling would have had even worst effects if the position would have been represented with the *Algebraic Input* combined with a 2×2 filter:

$$\begin{pmatrix} 0 & \overset{\Downarrow}{-9} & 0 & 0 \\ 5 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

In this case we would obtain a value of -1 which could even be interpreted as if *Black* is in control of the region of interest. Again, due to the presence of the *Queen* and the *Bishop* this is not true.
Until now we have seen how to represent chess positions as inputs for CNNs and how these input representations can be very sensitive to the geometrical operations that most standard CNN architectures use nowadays. Hence we know that the design process of the ANN architectures needs to be done very carefully when making use of CNNs in chess. The optimal ANN architectures that have been used in this research are presented in chapter 6 and 7. We end this chapter with the next Section in which we show how the training procedure of a CNN works and how it differs from the one that is used by MLPs.

4.3 Training a CNN

Even though both MLPs and CNNs make use of the backpropagation algorithm to get trained, there is one main difference between the two architectures. This difference is specifically related to the weights of the ANN. As explained in chapter 3, when it comes to MLPs each artificial neuron of the network has an independent weight vector associated to it. Training the network means adjusting the weights in such a way that an error function gets minimized. In CNNs the principle of minimizing an error function remains the same, however the artificial neurons of the network share the weights between each other. This is done in order to make the training procedure computationally less expensive, since a smaller amount of parameters needs to be tuned in order to make the ANN converge.
The role of the weights is of particular importance in CNNs, since the kernels that are used in order to perform the convolution operation consist of the weights themselves. Once the convolution operations have been performed, there usually is a final fully connected layer at the end of the network that flattens the convolved matrices into a vector which is used for the final prediction of the network.
In the case of training a CNN on an image with only 1 channel we can express the convolution operation with the following formula:

$$x^l_{i,j} = \sum_m \sum_n w^l_{m,n} o^{l-1}_{i+m,j+n} + b^l_{i,j}$$

$$o^l_{i,j} = f(x^l_{i,j}).$$

Here x corresponds to the input of the ANN with dimensions H and W. For both dimensions we have iterators defined as i and j. $x^l_{i,j}$ is the convolved input at layer l of the network while o defines its output. $w^l_{m.n}$ is the weight vector between layer $l-1$ and layer l while $f(\cdot)$ is the non linear activation function that is used between the layers of the ANN. b^l is the bias

that is added at layer l.
Once this process is applied on the whole input and for all the kernels, the output of the CNN is measured by an error function just as is done for the Multilayer Perceptrons. According to the error the weights of the network are changed towards the direction of its gradient. This process differs from the one used for standard MLPs since a double procedure is required. First of all the gradients have to be computed, while only secondly the weights can be updated (LeCun, Bengio, and Hinton, 2015).

Chapter 5

Artificial Neural Networks as Mathematical Function Approximators

This chapter explores the capabilities that Artificial Neural Networks (ANNs) have as universal approximators of any existing mathematical function. To do so, particular emphasis is put on the mathematical concepts that allow proving this very powerful capability. We firstly present a clear definition of the *Approximation Problem*, and see how it is directly related to the Perceptrons that have been presented in chapter 3. Secondly, we present the main theorems that make it possible to state that standard multilayer perceptrons, with a single hidden layer, a finite number of hidden neurons, and an arbitrary non-linear activation function, are universal approximators.
We explore this particular mathematical ability in detail, since we have trained different ANN architectures to approximate *Stockfish's* evaluation function as precisely as possible. The output of this function is then used by our chess playing program.

5.1 The Approximation Problem

Before investigating the *Approximation Problem* in depth, it is important to state what *Approximation Theory* is. According to (Poggio and Girosi, 1989), this specific branch of mathematics is concerned with how continuous and univariate functions can be expressed by more simple functions, producing similar results. The function that has to be approximated is usually defined as $f(x)$, and its approximation as $F(W,X)$. $X = [x_1, x_2, ..., x_n]$ and $W = [w_1, w_2, ..., w_n]$ are in this case real vectors that are part of a particular set defined as P. The main goal of the approximation problem is to find the best set of parameters, W, such that it makes F the closest possible approximation of f. The quality of this approximation is usually measured by a distance function, defined as p, that is able to determine how different $F(W,X)$ is from $f(x)$. We define this distance as:

$$p = p[f(x), F(W,X)].$$

As has been proposed by (Poggio and Girosi, 1990) we can formulate the *Approximation Problem* in mathematical terms as follows:

Assuming that $f(x)$ is an arbitrary squashing function defined on set $\mathbf{X}$, and that $F(W,X)$ is an approximating function of $f(x)$, that depends continuously on $W \in P$ and X, satisfying the approximation problem means finding the set of parameters such that the following inequality holds:

$$p[F(W^*,X), f(x)] \leq p[F(W,X), f(x)] \forall W \in P. \tag{5.1}$$

If a solution to this inequality exists it is said to be a *Best Approximation* (Poggio and Girosi, 1990).

5.1.1 Squashing Functions

The main part of the *Approximation Problem* relies in the importance of $f(x)$ being squashing. The importance of non-linear activation functions has been explained in chapter 3, however their properties will now be explained further.
According to (Hornik, Stinchcombe, and White, 1989), a function $\Phi : \Re \to [0,1]$ can be considered as squashing if it is not decreasing, and if its limits satisfy the following 2 conditions:

$$\lim_{\lambda \to \infty} \Phi(\lambda) = 1$$

and

$$\lim_{\lambda \to -\infty} \Phi(\lambda) = 0.$$

The most famous squashing function is the sigmoid one that has already been presented in chapter 3. In Figure 5.1 we show it once again since it nicely shows how it satisfies the 2 previously mentioned conditions.

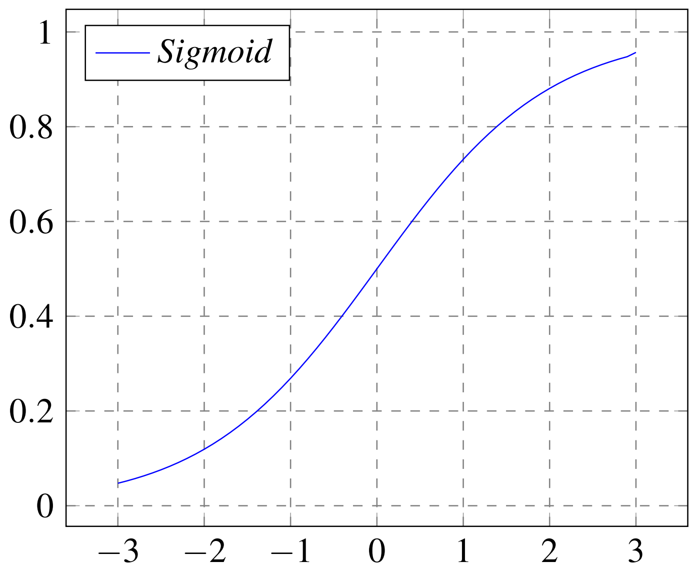

FIGURE 5.1: Sigmoid function

These 2 conditions are extremely important since they allowed Hornik, firstly in (Hornik, Stinchcombe, and White, 1989) and later in (Hornik, 1991), to propose the 3 main theorems that prove how single hidden layer perceptrons are able to approximate any mathematical function.
The theorems will now be presented: we define a single hidden layer perceptron, $\Sigma\Pi$, as used in (Hornik, Stinchcombe, and White, 1989).

5.1.2 Hornik's Theorems

Theorem 1 *For every squashing function Φ, every r and every finite measure μ on (R^r, B^r), both $\Sigma\Pi^r(\Phi)$ and $\Sigma^r(\Phi)$ are uniformly dense in C^r and p_μ dense in M^r.*

The theorem states that any single hidden layer perceptron is able to approximate any measurable function, regardless of the dimension of the input space r, the input space environment μ, and activation function Φ, as long as it is squashing (Hornik, Stinchcombe, and White, 1989).
This theorem however, only states that it is possible to approximate any function, but does not mention anything about how well the function is actually approximated. As presented by the distance measure p, we want $F(W,X)$ to not only be an approximation of $f(x)$ able to satisfy the inequality 5.2, but we also want p to be as low as possible. Hornik's second theorem takes this into account stating that:

Theorem 2 *For every function g in M^r there is a compact subset K of R^r and an $f \in \Sigma^r(\Phi)$ such that for any $\varepsilon > 0$ we have $\mu(K) < 1-\varepsilon$, and for every $X \in K$ we have $|f(x) - g(x)| < \varepsilon$, regardless Φ, r or μ.*

This theorem goes one step further, when compared to the first one, since it states that there actually is a $\Sigma\Pi$ that is able to approximate any measurable function to any desired degree of accuracy, on a compact set K. In this case it is possible to have p as low as possible.
However, even though this is very promising from a formal perspective, to make $\Sigma\Pi$ actually capable to approximate a function to any degree of accuracy, some conditions need to be satisfied. The first one is related to the amount of hidden units $\Sigma\Pi$ has, in fact the more hidden units, the higher the chances are to minimize p. Theoretically there are no constraints on this approximation process, but a very important role is played by the data on which the ANN needs to be trained. If the relationship between the inputs and the relative target values is non deterministic, it is impossible to satisfy the approximation problem.
In other words, more related to the work in this thesis: if to any chess position we do not associate a proper evaluation corresponding to the way *Stockfish* evaluates board states, its evaluation function will never be properly approximated.
Also, the depth of the ANN matters: the higher the amount of hidden units is, the quicker it will converge. However, Hornik also proved that it is possible to approximate any function with exactly 1 single hidden layer as shown by the third theorem:

Theorem 3 *Let $x_1, x_2, ..., x_n$ be a set of distinct points in R^r and let $g : R^r \to R$ be an arbitrary function. If Ψ achieves 0 and 1, there is a function $f \in \Sigma^r(\Psi)$ with n hidden units such that $f(x_i) = g(x_i) \forall i$.*

This can be considered as Hornik's main theorem, since it gathers together all the main points of the 2 previously mentioned theorems. Hence, we report Hornik's actual proof proposed in (Hornik, Stinchcombe, and White, 1989) hereafter:

Proof: Let $x_1, x_2, ..., x_n \subset R^1$. We can relabel it in such a way that we obtain $x_1 < x_2 < ...x_n$.
Now we pick $M > 0$ such that $\Psi(-M) = 1 - \Psi(M) = 0$.
We now define A_1 as the constant affine functions $A_1 = M$ and set $\beta_1 = g(x_1)$.
We set $f^i(x) = \beta_1 \cdot \Psi(A_1(x))$.
We define by induction A_k by $A_k(x_{k-1}) = -M$ and $A_k(x_k) = M$.
We define $\beta_k = g(x_k) - g(x_{k-1})$.
We set $f^k(x) = \sum_{n=1}^{k} \beta_j \Psi(A_j(x))$ for $i \leq k f^k(x_i) = g_k(x_i)$.
The desired function $f(x)$ is f^n.

5.2 Cases of Approximation Problems

It is possible to combine the information that has been presented in the current chapter together with the one presented in chapter 3, to distinguish 3 main examples of approximating functions, $F(w,x) : \Re^n \rightarrow \Re$, that are strictly related to MLPs. The examples here were first presented in (Poggio and Girosi, 1990).

- Linear Case: mathematically defined as

$$F(W,X) = W \cdot X$$

  with W and X being n dimensional vectors, this case corresponds to the basic perceptron presented in chapter 3 that is not provided with any hidden layers.

- Linear in a Suitable Basis of functions: where the basis is defined as $\Phi_{i=1}^{m}$ and the case itself as

$$F(W,X) = \sum_{i=1}^{m} W_i \Phi_i(X).$$

  In this case Φ corresponds to the product and power operations on the input X, and the type of ANN refers to a 1 hidden layer perceptron.

- Nested Sigmoid Scheme: where we define the case as

$$F(W,X) = \sigma\Big(\sum_n w_n \sigma\Big(\sum_i v_i \sigma\big(\ldots\sigma\big(\sum_j u_j X_j\big)\ldots\big)\Big)\Big)$$

  and corresponds to the one presented in chapter 3, by equation 3.1. In this case σ corresponds to a non linear activation function, and W the vector of weights $w_n, v_i, u_j, \ldots$ of the ANN. This case corresponds to a multilayer perceptron that performs a sigmoidal transformation on the sum of the input units and the bias.

Training an ANN to approximate an existing mathematical function is considered a regression task, since the aim of the neural architecture is to fit the underlying original function. To do so, when a *Supervised Learning* approach is chosen, the choice of the loss function is particularly important. The two main common choices are either the Mean Absolute Error function defined as:

$$MAE = \frac{1}{n}\sum_{i=1}^{n} |y_i - x_i|$$

or the Mean Squared Error one, defined as:

$$MSE = \frac{1}{n}\sum_{i}^{n} (y_i - x_i)^2.$$

In both cases the output of the two functions corresponds to p, presented in equation 5.2, y_i is the output of the ANN and r_i the desired target value given by the function which needs to be approximated.
It is also very important to mention that it is possible to find an approximation function through the use of *Reinforcement Learning* techniques. This is the case presented in (Baxter, Tridgell, and Weaver, 2000), where the ANN learned a very good chess evaluation function by combining the TD-learning with the MinMax searching algorithm. Here the authors define

$$J^*(\cdot)$$

as the evaluation function required to play optimal chess which, according to (Block et al., 2008), has to be approximated by a second unknown evaluation function defined as

$$\widetilde{J}(\cdot, w). \tag{5.2}$$

Training the ANN means optimizing equation 5.2 by adjusting its weights according to the following formula:

$$w := w + \alpha \sum_{t=1}^{N-1} \nabla \widetilde{J}(X_t, w) \left[\sum_{j-t}^{N-1} \lambda^{j-t} d_t \right]$$

where α corresponds to the learning rate of the ANN, $\nabla \widetilde{J}(\cdot, w)$ the vector containing the partial derivatives of the weights, and λ is the eligibility trace parameter, a constant controlling the contribution of the temporal difference error (d_t) from a particular position x_t until the end of the game. In (Baxter, Tridgell, and Weaver, 2000) λ is multiplied with a discount factor variable and had a value of 0.7 for their experiments.

Chapter 6

Results

This chapter presents the results that we have obtained on the classification and regression experiments. The main goal of this set of experiments was to find a very good chess evaluation function that allows our chess program to play high level chess without relying on deep lookahead algorithms.
To accomplish this task we have created 4 different *Datasets* from scratch, we use the first 3 of them for the classification experiments, presented in section 6.1, and the last one for a regression experiment that is presented in section 6.2. Considering the classification task, the experiments mainly differ according to the amount of labels that are used, namely 3, 15 and 20. On the other hand the regression experiment does not make use of any categorical labels but investigates the capabilities that Artificial Neural Networks (ANNs) have as mathematical function approximators, by attempting to approximate *Stockfish's* evaluation.
We investigate the performances of both Multilayer Perceptrons (MLPs) and Convolutional Neural Networks (CNNs) and explore the impact of two different board representations. The first one representing if a piece is present on the board or not, and the second one in which we assign a numerical value to the piece according to its strength. We refer to the first representation as *Bitmap Input* while to the second one as *Algebraic Input*, as mentioned in chapter 2.
For all the experiments we have split the dataset into 3 different parts: we use 80% of it as *Training Set* while 10% is used as *Testing Set* and 10% as *Validation Set* [1]. We use `Tensorflow` and `Python 2.7` for all programming purposes in combination with the parallel computing platform `cuda 8.0.44` and the deep learning library `cuDNN 5.1`. This allows us to have efficient GPU support for speeding up the computations. All experiments have been run on `Cartesius`, the Dutch national supercomputer.
Furthermore, considering the classification experiments all ANNs are trained with the *Categorical cross entropy* loss function. In the case of a binary classification task the function is defined as

$$L(X,Y) = -\frac{1}{n}\sum_{i}^{n} y_i \log \sigma(x_i) + (1-y_i)\log(1-\sigma(x_i)).$$

Where X is the set of inputs, Y the relative labels, σ a non linear activation function as presented in chapters 3 and 5 and n the total amount of input samples. On the other hand, the ANNs that are used for the regression experiment have been trained using the *Mean Squared Error* loss function defined as

$$MSE = \frac{1}{n}\sum_{i}^{n}(y_i - r_i)^2$$

where y_i is the output of the ANN, r_i the desired target value and again with n corresponding to the amount of training inputs.

[1] All the experiments use the same divisions in the dataset.

No matter which kind of input is used (*Bitmap* or *Algebraic*), in order to keep the comparisons fair we did not change the architectures of the ANNs.
The results together with the best performing ANN architectures and the best set of hyperparameters will now be presented.

6.1 Classification Experiments

We now present the ANN architectures that have provided the best results on the 4 different *Datasets*. The set of hyperparameters are the result of a lot of preliminary experiments that were needed in order to fine tune the ANNs.

6.1.1 Artificial Neural Network Architectures

Dataset 1

We have used a three hidden layer deep MLP with 1048, 500 and 50 hidden units for layers 1, 2, and 3 respectively. In order to prevent overfitting a *Dropout* regularization value of 20% on every layer has been used. Each hidden layer is connected with a non-linear activation function: the 3 main hidden layers make use of the *Rectified Linear Unit* (*ReLU*) activation function, while the final output layer consists of a *Softmax* output. The *Adam* algorithm has been used for the stochastic optimization problem and has been initialized with the following parameters:

- $\eta = 0.001$;
- $\beta_1 = 0.90$;
- $\beta_2 = 0.99$;
- $\varepsilon = 1e - 0.8$

as proposed by (Kingma and Ba, 2014). The network has been trained with *Minibatches* of 128 samples. The CNN consists of two *2D* convolution layers followed by a final fully connected layer consisting of 500 hidden units. During the first convolution layer 20 5×5 filters are applied to the image, while the second convolution layer enhances the image even more by applying 50 3×3 filters. Increasing the amount of filters and the overall depth of the network did not provide any significant improvements to the performance of the CNN. On the contrary it only drastically increased the training time. The *Exponential Linear Unit* (*Elu*) activation function has been used on all the convolution layers, while the final output consists of a *softmax* layer. The CNN has been trained with the *SGD* optimizer initialized with $\eta = 0.01$ and $\varepsilon = 1e - 0.8$. We do not use *Nesterov momentum*, nor particular time based learning schedules, however, a *Dropout* value of 30% together with *Batch Normalization* has been used in order to prevent the network from overfitting. This ANN has also been trained with *Minibatches* of 128 samples.
It is important to mention that the CNNs have been specifically designed to preserve as much geometrical information as possible related to the inputs. When considering a particular chess position, the location of every single piece matters, as a consequence no pooling techniques of any type have been used. In addition to that, the *"border modes"* related to the outputs of the convolutions has been set to *"same"*. This way the output of the convolutions is of the exact same size as the original chess positions. Hence, we are sure to preserve all the necessary geometrical properties of the input without influencing it with any kind of dimensionality reduction.

Dataset 2 and Dataset 3

On these datasets we have only changed the structure of the MLP while the CNN architecture remained the same. The MLP that has been used consists of 3 hidden layers of 2048 hidden units for the first 2 layers, and of 1050 hidden units for the third one. The *Adam* optimizer and the amount of *Minibatches* have not been changed. However, in this case all the hidden layers of the network were connected through the *Elu* activation function.

6.1.2 Accuracies

- *Dataset 1:* Starting from the experiments that have been performed on *Dataset 1*, presented in Figure 6.1, it is possible to see that the MLP that has been trained with the *Bitmap Input* outperforms all other 3 ANN architectures. This better performance can be seen both on an accuracy level and in terms of convergence time. However, the performance of the CNN trained with the same input is quite good as well. In fact the MLP only outperforms the CNN by less than 1% on the final *Testing Set*. We noticed that adding the information about the value of the pieces does not provide any advantage to the ANNs. On the contrary both for the MLP and the CNN this penalizes their overall performances. However, while on the experiments that have been performed on *Dataset 1* this gap in performance is not that significant, with the MLP and the CNN that still perform $> 90\%$ of accuracy, the same cannot be said for the ones that have been ran on *Dataset 2* and *Dataset 3*.

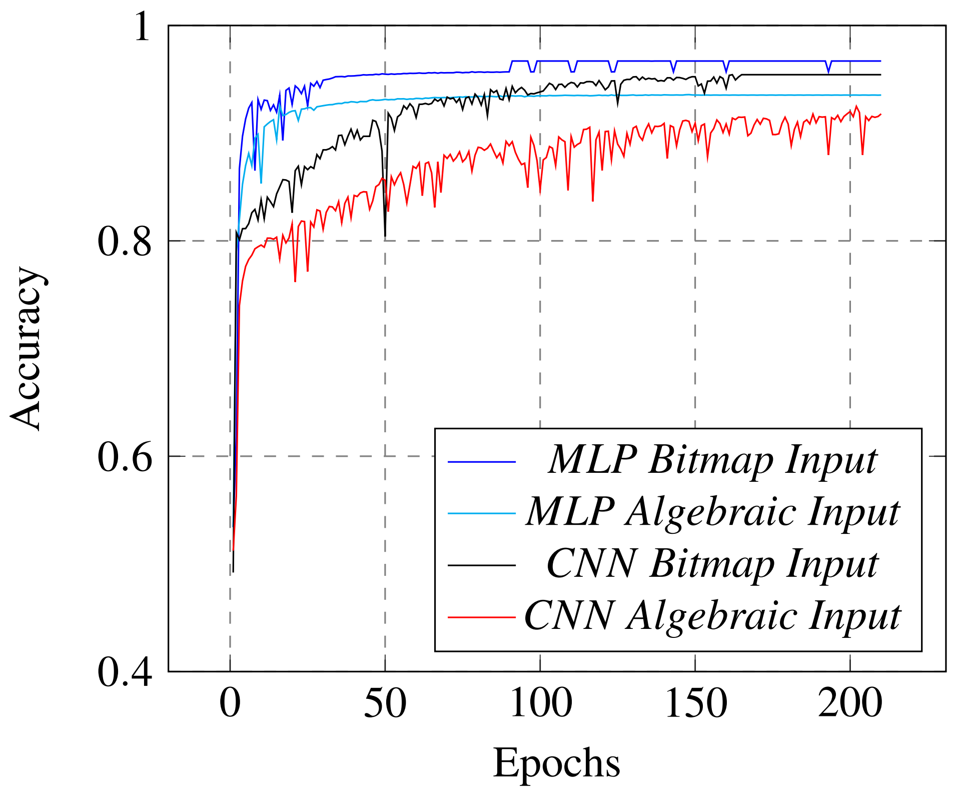

FIGURE 6.1: The *Testing Set* accuracies on *Dataset 1*

- *Dataset 2:* On a classification task consisting of 15 classes, we observe in Figure 6.2 lower accuracies by all the ANNs. But once more, the MLP trained with the *Bitmap Input* is the ANN achieving the highest accuracies. Besides this, we also observe that for the CNNs, and in particular the one trained with the *Algebraic Input*, the increment in the amount of classes to classify starts leading to worse results, which shows the superiority of the MLPs. A superiority that becomes evident on the experiments performed on *Dataset 3*.

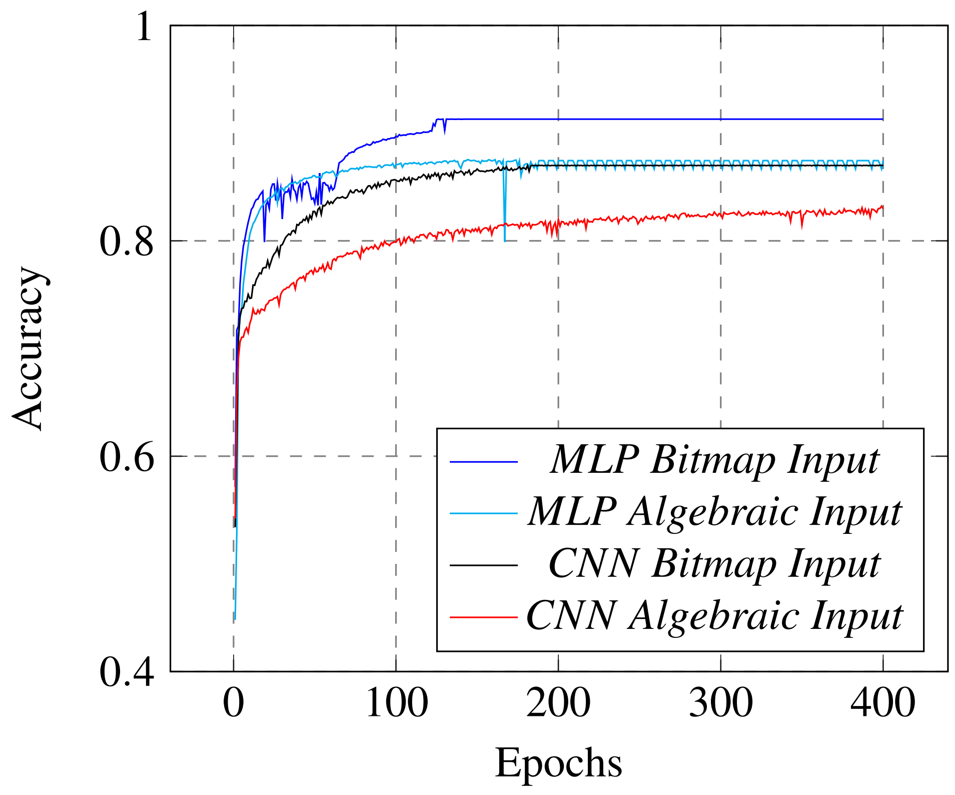

FIGURE 6.2: The *Testing Set* accuracies on *Dataset 2*

- *Dataset 3:* This dataset corresponds to the hardest classification task on which we have tested the ANNs. As already introduced, we have extended the *Draw* class with 6 different subclasses. As Figure 6.3 shows, the accuracies of all ANNs decrease due to the complexity of the classification task itself, but we see again that the best performances have been obtained by the MLPs, and in particular by the one trained with the *Bitmap Input*. In this case, however, we observe that the learning curve is far more unstable when compared to the one of the *Algebraic Input*. This may be solved with more fine tuning of the hyperparameters.

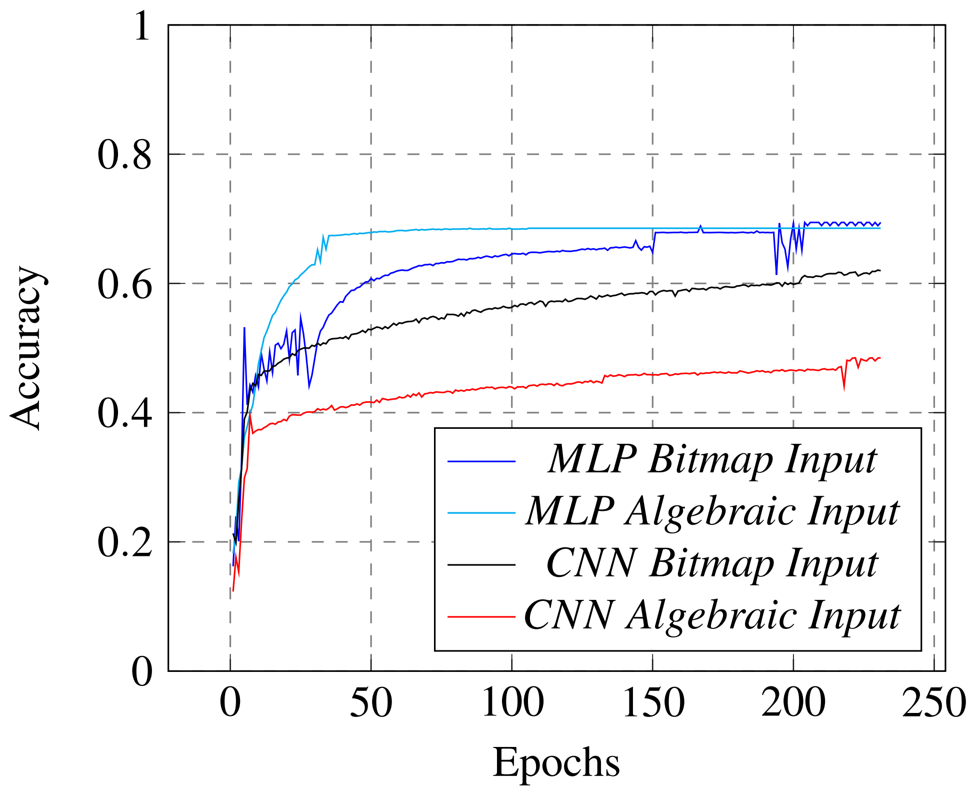

FIGURE 6.3: The *Testing Set* accuracies on *Dataset 3*

We summarize the performances of the ANNs on the first 3 datasets in the following tables. Table 6.1 reports the accuracies obtained by the MLPs while Table 6.2 shows the accuracies of the CNNs.

Dataset	*Bitmap Input*		*Algebraic Input*	
	ValSet	*TestSet*	*ValSet*	*TestSet*
*Dataset*1	98.67%	**96.07%**	96.95%	93.58%
*Dataset*2	93.73%	**93.41%**	87.45%	87.28%
*Dataset*3	69.44%	**68.33%**	69.88%	66.21%

TABLE 6.1: The accuracies of the MLPs on the classification datasets.

Dataset	*Bitmap Input*		*Algebraic Input*	
	ValSet	*TestSet*	*ValSet*	*TestSet*
*Dataset*1	95.40%	95.15%	91.70%	90.33%
*Dataset*2	87.24%	87.10%	83.88%	83.72%
*Dataset*3	62.06%	61.97%	48.48%	46.86%

TABLE 6.2: The accuracies of the CNNs on the classification datasets.

6.2 Regression

With the regression experiment that aimed to train the ANNs to reproduce *Stockfish's* evaluation function, we have obtained the most promising results from all architectures. The set of best performing hyperparameters are the following:

Dataset 4
A three hidden layer deep perceptron with 2048 hidden units per layer has been used. Each layer is activated by the *Elu* activation function and the *SGD* training parameters have been initialized as follows:

- $\eta = 0.001$;
- $\varepsilon = 1e - 0.8$
- *Nesterov Momentum* = 0.7

In addition to that *Batch Normalization* between all the hidden layers and *Minibatches* of 248 samples have been used. Also in this case, except for the final single output unit, the CNN architecture has not been changed when compared to the one used in the classification experiments. We tried to increment the amount of filters and the overall depth of the network by exploring the *Inception* module presented in (Szegedy et al., 2017). However, similar to what happened for the classification experiments this only drastically increased the training time.
In Table 6.3 we report the Mean Squared Error (MSE) that has been obtained on the *Validation* and *Testing Sets*.

ANN	*Bitmap Input*		*Algebraic Input*	
	ValSet	*TestSet*	*ValSet*	*TestSet*
MLP	0.0011	**0.0016**	0.0019	0.0021
CNN	0.0020	0.0022	0.0021	0.0022

TABLE 6.3: The MSE of the ANNs on the regression experiment.

We managed to train all the ANNs to have a Mean Squared Error lower than 0.0025. By taking their square root, it is possible to infer that the evaluations given by the ANNs are on average less than 0.05 *cp* off when compared to the original evaluation function provided by the chess engine. Once again, the best performance has been obtained by the MLP trained on the *Bitmap Input*. The MSE obtained corresponds to 0.0016, meaning that *Stockfish* evaluates chess positions only ≈ 0.04 *cp* differently when compared to our best ANN that does not use any lookahead. It is also important to highlight the performances of the CNNs. While during the classification experiments the superiority of the MLPs was evident, the gap between CNNs and MLPs is not that large, even though the best results have been obtained by the latter architecture. Our results show in fact how both types of ANNs can be powerful function approximators in chess.

6.3 Discussion

The results that have been obtained make it possible to state two major claims. The first one is related to the superiority of MLPs over CNNs as optimal ANN architecture in chess, while the second one shows the importance of not providing the value of the pieces as inputs to the ANNs.
We think that the superiority of MLPs over CNNs, that is highlighted in our classification experiments, is related to the size of the board states. The large success of CNNs is mainly due to their capabilities to reduce the dimensionality of pictures while at the same time enhancing their most relevant features. Chess, however, is only played on a 8×8 board, which seems to be too small to fully make use of the potential of this ANN architecture in a classification task. Generalization is made even harder due to the position of the pieces. Most of the time

they cover the whole board and move according to different rules. On the other hand, this small dimensionality is ideal for MLPs, the size of the input is small enough to fully connect all the features between each other and train the ANN to identify important patterns. In this case it is also very feasible for the ANN to understand how different piece types move and perform on the board.
Considering the importance of not providing the ANNs with the material information of the pieces, we have identified a bizarre behavior. Manual checks show how the extra information provided by the *Algebraic Input* is able to trick the ANNs especially in *Endgame* positions. A particular case is presented in Figure 6.4.

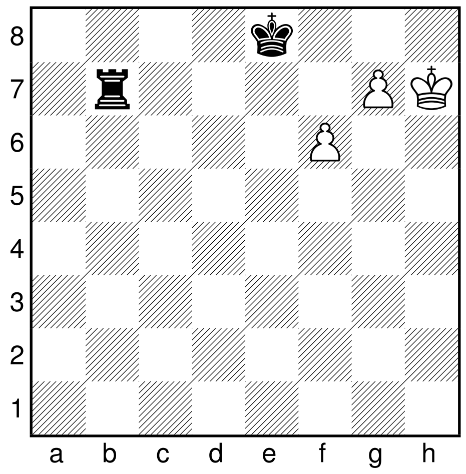

FIGURE 6.4: An example of a position misclassified by the ANNs that have been provided with the *Algebraic Input*.

White is trying to promote its 2 *Pawns* while *Black* is trying to prevent this with the help of the *King* and the *Rook*. This position has been scored as a *Draw* by both the MLP and the CNN trained with the *Algebraic Input*, while correctly classified as *Winning* for White by the MLP and the CNN trained with the *Bitmap Input*. It is interesting however, how the *Algebraic Input* had less impact on the regression experiments. In this case the performances of the ANNs trained with this input still provide very good results. Also the small board state does not seem to influence the function approximation capabilities of CNNs that fit *Stockfish's* evaluation function very well.
To conclude some further remarks have to be made by considering the amount of training time that the ANNs needed to converge. We have checked how much time the MLPs and CNNs needed to successfully complete one training epoch. Considering the MLP, 158 seconds, 334, 415 and 440 seconds were needed to finish one epoch on the four different *Datasets*. On the other hand the CNNs needed 213, 215 225 and 235 seconds on the same type of experiments. It might look that the CNNs seem to be the more efficient ANN architecture to use, since they are on average $\approx$ 115 seconds faster when compared to the MLP, however if we also take into account the amount of epochs that are needed in order to make the ANNs converge we see that the MLP is still the fastest architecture. In fact, as can be seen from Figures 6.1, 6.2 and 6.3 the MLPs always converge faster than the CNNs. In general, when making use of the MLP the ANN always converged in a little bit less than 24 hours for the first 3 *Datasets*. On the other hand, the CNN especially on *Dataset 2* and *Dataset 3* required almost 3 days of training before converging. The same amount of time was required by the MLP during the *Regression* experiment performed on *Dataset 4* where in this case the quicker architecture turned out to be the CNN, which converged after $\approx$ 48 hours of training. The distinction between *Bitmap* and *Algebraic Inputs* did not have any effect on the amount of time required to complete one training epoch. Furthermore, considering that the best results have always

been obtained by the MLPs it is possible to confirm the previously mentioned statement that sees MLPs as optimal ANN architecture to use in chess.
In the next chapter we explore if the gap between MLPs and CNNs can be reduced by making use of deeper CNNs that are trained on the *Features Input*, the final input representation that has been mentioned in chapter 2. Moreover, we also investigate the actual playing performances of the best performing ANNs.

Chapter 7

A Novel Board Input Representation

In this chapter we investigate if it is possible to improve the performances of Convolutional Neural Networks (CNNs) in such a way that they perform as well as Multilayer Perceptrons (MLPs). To do so, we have developed a novel representation of the chess board that is used as input for the CNNs. We refer to this new input representation as *Feature Input* and we use it to perform the same experiments that have been presented in the previous chapter. We designed it by taking inspiration from the results obtained in chapter 6 trying to combine the best ideas that guided the design process of the *Bitmap* and *Algebraic Inputs*.
The structure of this chapter is as follows: In section 7.1 we present the novel board representation on which we have tested the CNNs by explaining in detail which kind of new feature maps have been used as inputs for the ANN. While in section 7.2 we investigate the impact of these new feature maps by presenting and discussing the results.

7.1 Adding Channels to the CNNs

We have shown in chapter 2 how it is possible to represent a complete chess board in an informative way while being at the same time sure to minimize the information loss as much as possible. This board representation, called the *Bitmap Input*, represents all the 64 squares of the board with 12 binary features that represent each piece on the board together with which side is moving it. The chess board is represented as a binary sequence of bits of length 768 that can be reshaped into a tensor of $8 \times 8 \times 12$, with the first 2 dimensions representing the size of the chess board, and the last one the amount of piece types. Considering the results presented in the previous chapter we have also shown how this input representation is ideal for MLPs, since from a computational perspective it is very suitable to fully connect all the features between each other. However, we also know that it is not possible to get this full connectivity between the features when making use of CNNs. Hence, we have developed a novel *Input* representation that adds new channels, next to the already 12 present ones, in order to reduce the gap between MLPs and CNNs, and maybe let the latter ANN architecture even outperform the first one.
The design of these new extra features is largely based on chess theory (Hellsten, 2010), in fact all the features that we have added play a very important role when highly skilled chess players evaluate chess positions. These features will now be described. It is important to mention that in order to tell the ANN which player is moving, we always encode board positions as if it would be *White's* turn, despite having a negative evaluation when it is *Black's* turn.

7.1.1 New Features

- *"Check whenever you can. It may be mate"* is a very famous quote by Hector Rosenfeld, a member of the Manhattan Chess Club and professional puzzle-maker who inspired the first feature that we have added as input to the CNNs. In fact, we know that the final goal of chess is to mate the opponent's *King*, as a consequence, whether a player is threatening its opponent's most valuable piece is information of high importance. At the same time, it is also very important to avoid as much as possible getting checked. In fact, this leads to a forced move, which means that the player who is under attack is forced to make a move that "releases" the *King*. The effects of getting checked can be devastating since they do not allow the player to attack their opponent properly and can determine the final outcome of the game as shown in Figure 7.1.

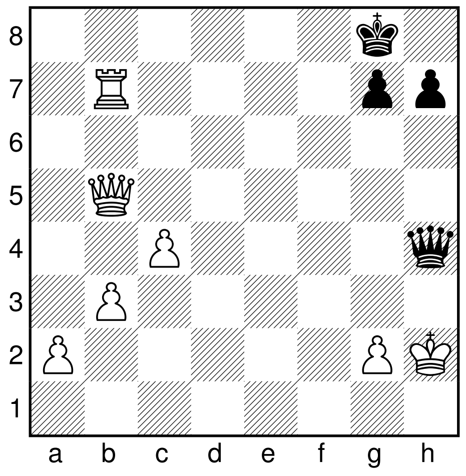

FIGURE 7.1: Example Position of a Perpetual Check

As we can see from the Figure, *White* has two *Pawns* and an entire *Rook* of advantage when compared to *Black*, furthermore it will be also able to mate its opponent with the move `Qe8`, however, *Black* is checking *White's King* and is able to force a draw by repetition. In fact, they are able to perpetually check *White* with the following forced line of moves that starts after `Qh4+`: `Kg1`, `Qe1+`, `Kh2`, `Qh4+` ... Considering the importance of *Checking/Getting Checked* we have added this information as a new feature layer to the input of the CNN. If one player is checking its opponent we add a 8×8 matrix containing 1 to the input, while it is a matrix containing only 0*s* if neither of the two players is checking their opponent.

- *"The Pin is mightier than the sword"* said the top ten American player of the early 1930*s*, Fred Reinfeld. In fact being able to pin a piece corresponds to highly limit the mobility of the opponent's pieces. This is particularly relevant in the case of *"Absolute Pins"* in which a piece is pinned on its own *King* and as a consequence is unable to move. In this case this results with the player who is pinning the piece being able to play with an extra piece on the board as shown by Figure 7.2 where *White* is unable to move its *Knight*.
  This information has been added as input to the CNN as follows: if there is a *White* piece on the board that is pinning a *Black* one we mark all the squares that are controlled by the pinning piece with 1, the value is again flipped if it is *Black* being the player who is pinning. We check if there are pinning pieces on each square of the board, in case there aren't any we simply add a 8×8 matrix of 0*s* as input to the ANN. Figure 7.3 shows how this extra feature layer looks like in the case in which there is a pinned piece on the board.

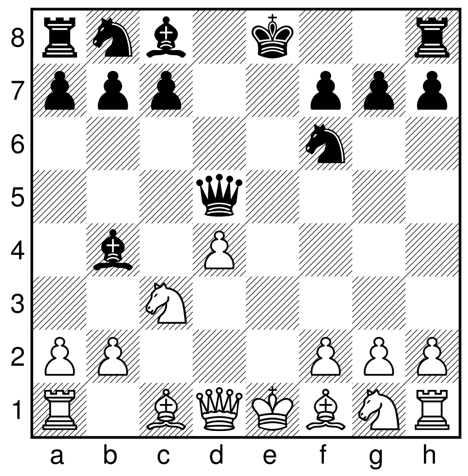

FIGURE 7.2: Example Position with Black Bishop pinning a White Knight.

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

FIGURE 7.3: Relative feature layer for the CNN.

- *"Best by test*: `1.e4`" said the most famous American chess player Bobby Fischer. A component that plays a very important role in chess is the control of the most central squares of the board. Being in control of the center has two main benefits: firstly it allows the controlling player to develop pieces such as the *Bishops* and the *Knights* without too much effort. Secondly if a piece is in the center of the board it simply controls more squares than a piece that is located on the border of the board. As a consequence it is more powerful and effective. An opening that starts with `1.e4` follows these two very simple principles and are the reasons that made Bobby Fisher consider this move as the best possible way to start a game.
  However, the `e4` square is not the only central square of the board, in fact `c4,c5,d4,d5` and `e5` can be considered equally important. In the feature layer that is related to this kind of information we check how many pieces control one of the previously mentioned squares, if there are more *White* pieces than *Black* ones we mark those squares with a 1, on the other hand we mark them with a -1 if they are controlled by *Black*. We show this feature layer in Figure 7.5.

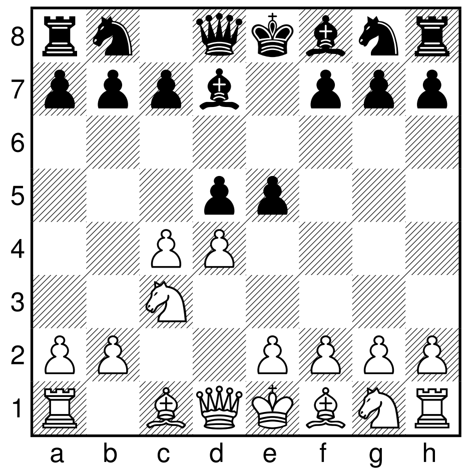

FIGURE 7.4: Example Position with White controlling the central squares.

$$\begin{bmatrix}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & \mathbf{2} & \mathbf{1} & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{bmatrix}$$

FIGURE 7.5: Relative feature layer for the CNN.

- *The vulnerable f2 and f7 squares.* Similarly to what has been done in the previous feature layer, for what can be considered as the strongest set of squares of the chess

board. In this case we reserve particular attention for the two weakest squares of the board, namely: `f2` and `f7`. In fact, these two squares are especially vulnerable since they are the only two squares of the board that are only defended by one single piece. Furthermore, as can be seen in Figure 7.6, the defending piece of these squares is the *King*, which makes this area of the board particularly susceptible to mating attacks.
We have encoded the importance of *Attacking/Defending* `f2` and `f7` similarly to how we have encoded the control of the center in the previous feature layer. However, besides counting how many pieces are attacking the square, we also mark the set of squares from where the attack starts. Furthermore, we assign a value to the attacked square according to the total strength of the pieces that are controlling it. Figure 7.7 visualizes this feature layer.

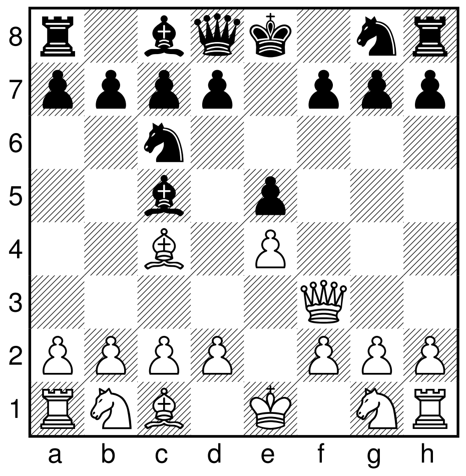

FIGURE 7.6: Example Position with *White* attacking the f7 square.

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 12 & 0 & 0 \\ 0 & 0 & 0 & 0 & \mathbf{3} & \mathbf{9} & 0 & 0 \\ 0 & 0 & 0 & \mathbf{3} & 0 & \mathbf{9} & 0 & 0 \\ 0 & 0 & \mathbf{3} & 0 & 0 & \mathbf{9} & 0 & 0 \\ 0 & \mathbf{3} & 0 & 0 & 0 & \mathbf{9} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

FIGURE 7.7: Relative feature layer for the CNN.

7.2 Results

In this section we explore if the strategy of adding channels to the input of the CNN improved its performances on the four different *Datasets* that we have created. Before presenting the results it is important to mention that the architecture of the ANN has been changed when compared to the one presented in chapter 6. In fact, since new feature layers have been added as input to the network we have decided to increase its depth. We have also carefully retuned the set of hyperparameters which allowed to obtain the results that will now be presented.
In total we make use of 3 convolution layers, in the first one we apply a 7×7 filter, while in the second and third one we apply a 5×5 filter and a 3×3 one. We end the network with a final fully connected layer of 250 units connected to a *Softmax* output. Unlike in the previous architecture, this time we did not make use of any *Batch Normalization* between the convolution layers. We have also reduced the *Dropout* value which has been set to 0.25 on the last convolution layer and the final fully connected layer only. The ANN has been trained with *Minibatches* of 128 samples and with the SGD optimizer set as follows: $\eta = 0.01$ and $\varepsilon = e - 0.8$. We again preserve all the geometrical properties of the input by setting the border modes of the convolutions to *"same"* and by avoiding *Pooling*. The architecture of the network has not been changed according to which *Dataset* has been used for the experiment. Considering the loss functions we make use of the *Categorical Crossentropy* for the classification experiments, performed on *Datasets* 1, 2 and 3, while we use the *Mean Squared Error* for the regression experiment performed on *Dataset 4*. Finally, the activation function that has been used is the *Exponential Linear Unit (Elu)* one.

This set of experiments has been run on `Peregrine`, the high performance computing cluster of the University of Groningen. The tools that we have used for all programming purposes are however the same ones when compared to the previous set of experiments that has been ran on `Cartesius`. In fact `Python 2.7` in combination with `Tensorflow` has been used to program the ANNs. `Peregrine` also provided efficient `GPU` support with the high parallel computing platform `cuda 8.0.44` and the deep learning library `cuDNN 5.1`. The results will now be presented. We show the comparisons between the just presented CNN architecture that has been trained on the *Feature Input*, and the 2 best performing ANN architectures presented in chapter 6: the MLP and the CNN that have both been trained on the *Bitmap Input*.

7.2.1 Dataset 1

The first results that we present are the ones performed on *Dataset 1*. We aimed to train the CNN on a classification task that made use of only 3 classes which represented the 3 possible final outcomes of a game. To sum up, the accuracies that were obtained by the 2 best architectures presented in the previous chapter were both very high. In fact the CNN obtained a *Testing Set* accuracy of 95.15% while the MLP one of 96.07%. Nevertheless, as can be seen in Figure 7.8 we managed to improve these results. The CNN trained on the *Feature Input* obtained a final accuracy on the *Testing Set* of 97.44%, performing better by $\approx 2\%$ when compared to the previous CNN and beating the MLP by $\approx 1\%$, which is the architecture that obtained the best result on all 4 *Datasets*.

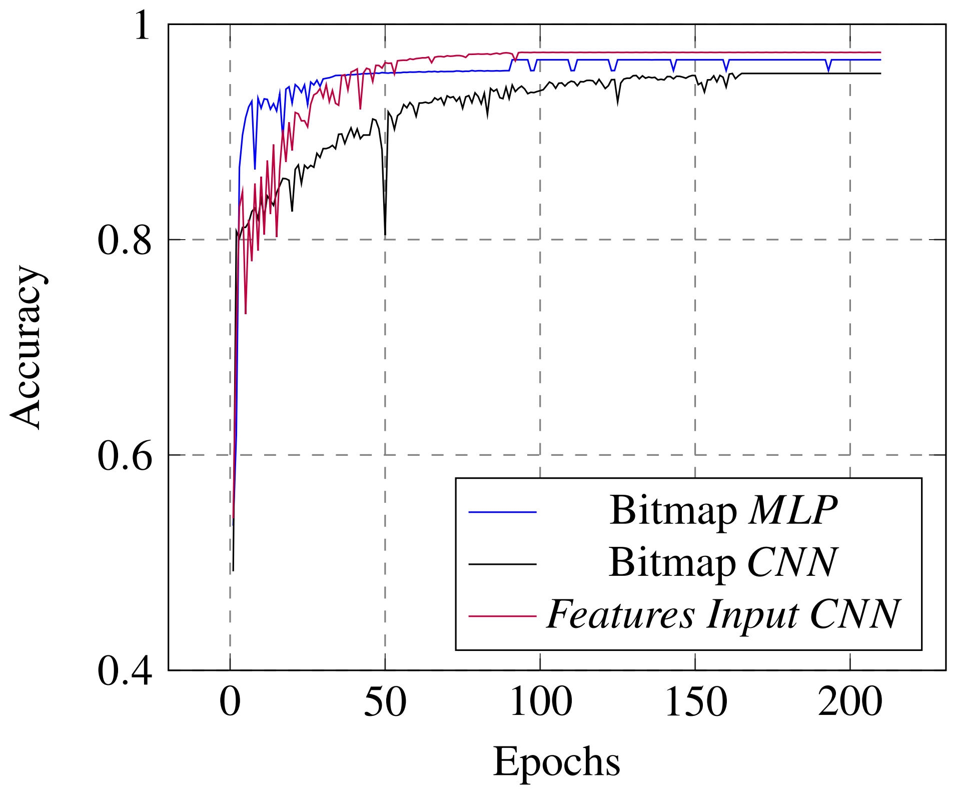

FIGURE 7.8: Comparisons between the CNN trained on the *Features Input* and the 2 best architectures of the previous experiment for *Dataset 1*.

The results obtained on *Dataset 1* introduce the possibility that a CNN trained with the novel proposed input representation, could at least perform as well as an MLP. However, an improvement of $\approx 1\%$ on the *Testing Set*, although positive, makes it still hard to state any

major claims about the superiority of this architecture.
Hence, we have explored the capabilities of this architecture and input representation on *Dataset 2*, in which the complexity of the classification increased.

7.2.2 Dataset 2

In fact, the classification task related to this *Dataset* makes use of 15 different labels. More in detail the extra labels that are added to the *Dataset* are all related to the classification of either *Losing* or *Winning* positions. This classification task has been created in order to explore the capabilities that the ANN has both in classifying how severely a *Losing* position can be, as well as how strongly *Winning* it can be.
Due to the amount of labels we noticed in the previous set of experiments, that this task generally led to worse performances of the ANNs. Furthermore, it started to explicitly show the difference in performance between the MLP and the CNN. In fact, the first architecture obtained a *Testing Set* final accuracy rate of 93.41%, while the latter architecture performed $\approx 6\%$ worse, with an accuracy rate of 87.24%. We expected to see a drop in the performance of the newest CNN of at least 3%, meaning that it would have performed as good as the MLP. However, the results obtained have been much more positive than expected. In fact the ANN managed to obtain a final *Testing Set* accuracy of 96.26%. This result is remarkable for multiple reasons. First of all it shows how the increment in the *Dataset* complexity did not influence the robustness of the ANN architecture. In fact, when compared to the performance obtained on the previous *Dataset*, the ANN only performed $\approx 1\%$ less. This introduces the second reason why this result is so promising. Both previous ANN architectures showed a drop in performance of $\approx 3\%$ in the case of the MLP, and even $\approx 10\%$ in case of the CNN. Last but not least, the results obtained in this experiment are the first ones in which a CNN manages to outperform an MLP so clearly. This means that the additional feature layers that we have added as input to the CNN are extremely representative of the chess positions and help the ANN during the evaluation process. They do not only replace the full connectivity of the features that makes MLPs such a powerful ANN architecture for chess, but even seem to be able to completely outperform this neural architecture. The results that have been obtained on this *Dataset* are presented in the following Figure 7.9.

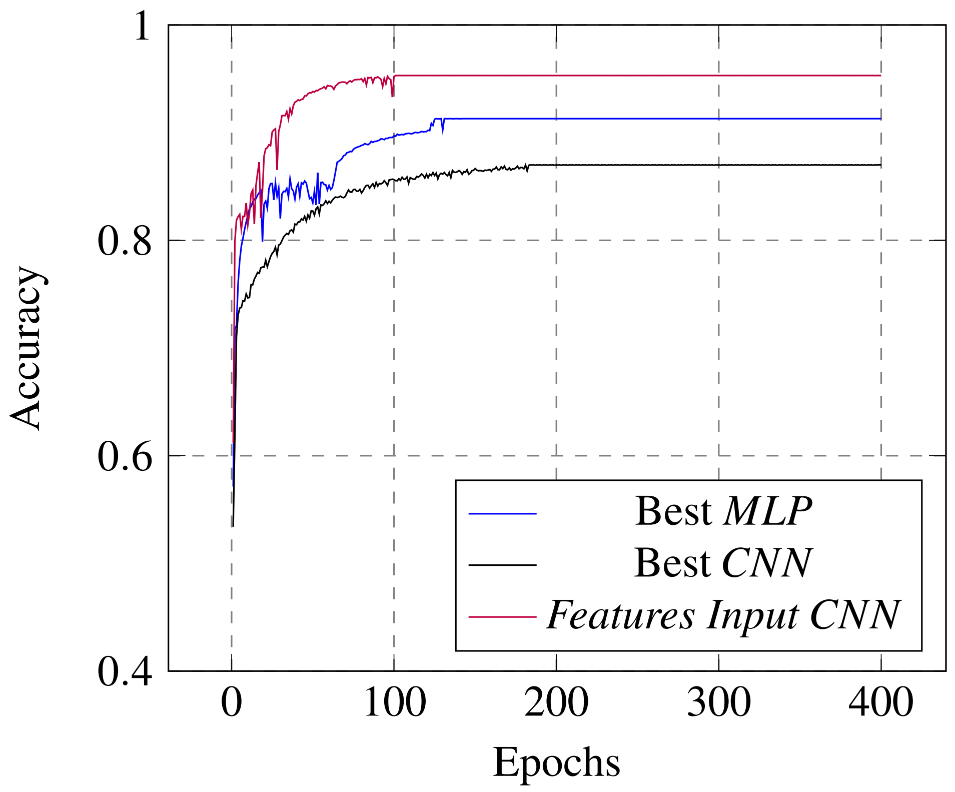

FIGURE 7.9: Comparisons between the CNN trained on the *Features Input* and the 2 best architectures of the previous experiment for *Dataset 2*.

7.2.3 Dataset 3

Finally, we have explored the performances of the CNN trained with the *Feature Input* on the most complicated classification task. In this case the ANN is supposed to classify 6 extra labels when compared to the previous *Dataset*. The difficulty of this experiment is however not only related to the increasing amount of classes to classify but is mainly related to the complexity of them. In fact they all represent different *Draw* positions that are very hard to distinguish even for the strongest human players.
This complexity drastically reduced the performances of both the CNN and the MLP trained on the *Bitmap Input*. In fact, the first architecture obtained a final accuracy of $\approx 62\%$ on the *Testing-Set*, while the latter got $\approx 68\%$. Both ANN architectures were very hard to train due to the hyperparameters engineering process and, especially in the case of the MLP, resulted in a very unstable ANN. Furthermore, this experiment again showed the superiority of the MLP when compared to the CNN, with the first neural architecture significantly outperforming the latter one.
However, as can be seen in Figure 7.10, similarly to what was introduced in the experiments performed on *Dataset 1*, and to what has been made explicit by the ones performed on *Dataset 2*, the CNN trained on the novel *Feature Input* resulted in the strongest architecture.

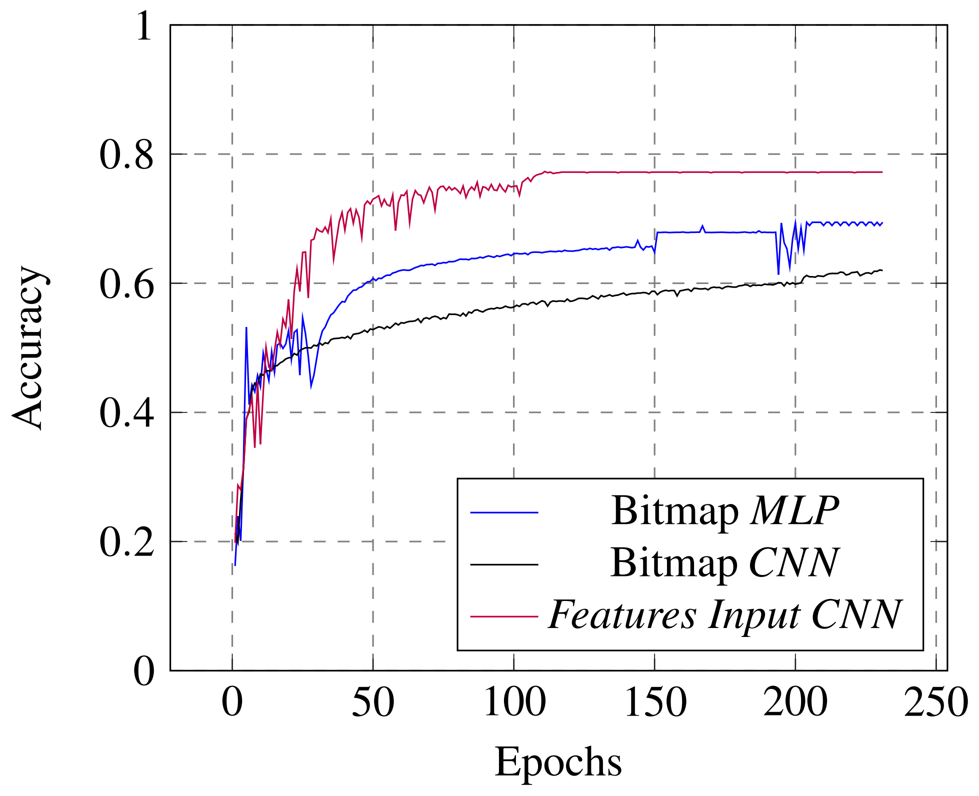

FIGURE 7.10: Comparisons between the CNN trained on the *Features Input* and the 2 best architectures of the previous experiment for *Dataset 3*.

We can see from the just presented figure that the accuracy the has been obtained on the *Testing Set* is again very high, especially when we consider the difficulty of the classification task. The CNN managed to outperform the previous CNN architecture by obtaining a $\approx$ 15% higher *Testing-Set* accuracy, while performing better than the MLP by $>$ 10%. Besides having achieved an accuracy of $\approx$ 78% it is also possible to infer from how nicely the ANN converges, how robust this architecture trained on the novel input representation is.
We sum up the performances that have been obtained on the classification experiments in Table 7.1 in which we clearly see the superiority of the *Feature Input* as better representation of the chess board in combination with a CNN.

ANN	*Dataset1*		*Dataset2*		*Dataset3*	
	ValSet	*TestSet*	*ValSet*	*TestSet*	*ValSet*	*TestSet*
Best MLP	98.67%	96.07%	93.73%	93.41%	69.44%	68.33%
Best CNN	95.40%	95.15%	87.24%	87.10%	62.06%	61.97%
Feat CNN	98.91%	**97.44%**	96.48%	**96.26%**	78.02%	**77.82%**

TABLE 7.1: The accuracies of the best performing ANNs on the 3 different classification datasets. The results show the superiority of the CNN trained on the *Feature Input* in all the experiments we have performed.

7.2.4 Regression

On the regression experiment, that aimed to train the ANNs in such a way that they are able to reproduce *Stockfish's* evaluation function as precise as possible we obtained slightly less remarkable results. The results that have been presented in chapter 6 were already very promising, and improving them turned out to be a very challenging task. In fact, as shown in

the previous chapter, no matter which kind of neural architecture and which input is used the Mean Squared Errors obtained by the ANNs on the *Testing Set* were never higher than 0.0022. Hence their exact Cp evaluations were on average only ≈ 0.05 Cp off when compared to the evaluation provided by the engine. However, the best results were again obtained by the MLP trained on the *Bitmap Input* that obtained a final MSE of 0.0016 which outperformed the CNN, trained on the same type of input by only 0.0005.
However, we still managed to improve both the performances of the CNN and the one of the MLP, even though as shown by Table 7.2 in a less impressive way when compared to the results obtained on the classification experiments.

ANN	Validation Set	Testing Set
$Bitmap_{MLP}$	0.0011	0.0016
$Bitmap_{CNN}$	0.0019	0.0021
$Feature_{CNN}$	0.0013	**0.0015**

TABLE 7.2: Comparison between the Mean Squared Error obtained by the CNN trained on the new feature representation and the MLP and CNN trained on the *Bitmap Input*

As we can see from the table the CNN trained on the novel proposed input representation performs better when compared to the two best performing architectures evaluated in chapter 6. The final MSE obtained by the ANN is of 0.0015, meaning that its evaluations are only 0.038 different from the ones that would be given by *Stockfish*. This is an improvement compared to the potential cp value that would be given by the previous best performing CNN (0.045) and also towards the best MLP (0.04). While it is true that, even the smallest changes in the output of the ANN matter, since they actually improve the quality of their play, we suppose that this small improvement in performance will not affect the game playing quality of the ANN remarkably. This hypothesis will be explored in detail in chapter 8.
Despite being less impressive when compared to the results obtained in the classification experiments, it still is possible to extract valuable insights from these results. The first one is related to the differences between the MSE obtained on the *Validation Set* and on the *Testing Set* by the ANNs. We see from Table 7.2 that the difference between the 2 values is way larger in the case of the MLP, showing that this architecture seems to be more susceptible to overfitting issues in training. In fact, the difference between the MSE values obtained by the CNN is in both cases 0.0002.
Another important insight can be obtained by looking at the training time the ANNs required to converge and finish one epoch. The MLP required 440 seconds to complete one training epoch and converged after ≈ 3 days of training. On the other hand the *CNN* finished one epoch in 150 seconds, which allowed the ANN to converge in a little bit more than 36 hours. Hence, besides performing slightly better when compared to the MLP, the CNN has a remarkable advantage in training more efficiently.
A final remark is required when analyzing the capabilities that ANNs have as mathematical function approximators. Considering the very similar results that have been obtained by all neural architectures, we believe that the MSE performances can hardly be improved on the current *Datasets* containing $\approx 3,000,000$ positions. It might be possible that *Stockfish's* evaluation function could be approximated even better if more games would be provided to the ANNs. On the other hand, it might also be possible that their performances cannot be improved any better or that the same performances can be obtained by making use of fewer positions. We leave these research questions as possible future work.

Chapter 8

Final Playing Performances

In this chapter we investigate the actual chess playing capabilities of the best performing ANNs. We firstly do this by performing the Kaufman test which we present in section 8.1. This test consists of 25 chess positions that aim to evaluate the strength of chess playing programs.
Secondly we make the best performing ANNs play against each other. The set of experiments are presented in section 8.2 and have been performed in order to find out which ANN architecture between the MLP and the CNN is the most suitable one for the game of chess. Furthermore, from the obtained results we are able to infer which board representation between the *Bitmap Input* and the *Feature Input* is best. It is however important to specify that the *Feature Input* has been specifically designed for the use of CNNs and that we use two different architectures for this comparison. In fact, the tensor representing the *Feature Input* results into a too large stacked vector to make it possible to successfully use MLPs without an extensive hyperparameter engineering process. Hence their use has not been investigated in this thesis.

8.1 The Kaufman Test

The first approach that we have used in order to test the chess strength of the ANNs is the Kaufman Test, a state of the art testbed developed by the chess Grandmaster Larry Kaufman which has been specifically designed to evaluate the strength of chess playing programs (Kaufman, 1992). The test consists of a dataset of 25 extremely complicated chess positions and the main goal of the chess playing program is, given a testing position, to play what is considered as the best possible move by the test. According to the test, there always is an optimal, unique, move to be played in any position. Most chess engines, like *Stockfish*, are able to easily find the move prescribed by Larry Kaufman, however it is important to notice that this requires a lot of lookahead exploration. Since most of the positions in the test are heavily tactical, it seems that the best move can only be found by looking ahead very deeply and by evaluating each position for more than one minute.
However, the way our ANNs approach this test is different. Since the main goal of this thesis is to train a computer program to play chess without having to rely on deep lookahead algorithms we tackle the test as follows: given any chess position of the test, defined as S_t, the ANNs only compute all possible board states looking ahead for just one move. This creates a set of new board states that we define as S_{t+1}. The ANNs evaluate each single board position $s \in S_{t+1}$ and play the move that maximizes the chances of winning, which basically corresponds to the board position with the highest evaluation.
We evaluate the quality of the move played by the ANNs in two ways. The first one by simply checking if the move played by the ANN is the one which is prescribed by the Kaufman test. Secondly by computing Δcp. Δcp is a measurement that we introduce in this thesis in order to find out how large the gap between the move that should be played by the test is compared to the one that is actually being played by the ANNs. We compute Δcp as follows:

we firstly evaluate the board state that is obtained by playing the move suggested by the test with *Stockfish*. The position is evaluated very deeply for more than one minute, as a result we assign *Stockfish's cp* evaluation to it. We name this evaluation δ_{Test}. We then do the same on the position obtained by the move of the ANN in order to obtain δ_{NN}. Δcp simply consists of the difference between δ_{Test} and δ_{NN}. The closer this value is to 0, the closer the move played by the ANN is to the one prescribed by the test.
We performed the test on the 2 ANNs that have obtained the best results on the set of experiments described in chapter 6 and in chapter 7. Hence, the neural architectures on which we have performed the test are the MLP that has been trained on the *Bitmap Input* and the CNN trained on the novel feature representation that is described in chapter 7.
In the following table we report for each position of the test [1] which move should be played according to Kaufman, which move has actually been played by the ANNs and the previously introduced Δcp value.

Position	*Best Move*	*MLP Move*	*MLPΔ cp*	*CNN Move*	*CNNΔ cp*
1	Qb3	Nf3	0	Bb5+	0.21
2	e6	Bd7	0.8	h5	0.7
3	Nh6	Nh6	0	Nh6	0
4	b4	Qc2	0.8	Be2	1
5	e5	e6	0.3	h5	1.2
6	Bxc3	Bxc3	0	Bxc3	0
7	Re8	Bc4	0.1	Bb5	0.1
8	d5	Qd2	0.9	b4	1
9	Nd4	Ne7	0.3	Na5	0
10	a4	a3	0.1	Nbd2	3.6
11	d5	h5	1.2	Nb1	1.9
12	Bxf7	Nf3	4.2	h6	2
13	c5	Nxe4	1.7	Qd2	4.8
14	Df6	f5	5.9	Nxe4	0.3
15	exf6	Bd4	4.6	f6	0.2
16	d5	Rb8	0.6	Nd2	0.6
17	d3	c4	0.8	a5	1.4
18	d4	Qe1	1.4	Nxc3	1.1
19	Bxf6	h3	4.2	Ne6	4.6
20	Bxe6	Bd2	1.7	Bf4	1.5
21	Ndb5	Rb1	1.4	Qd2	1.6
22	dxe6	Kxe6	20	Rg1	4.0
23	Bxh7+	Nh4	5.7	g3	4.5
24	Bb5+	b4	1.2	Rb1	0.8
25	Nxc6	bxc6	0.6	Bh3	4.8

TABLE 8.1: Comparison between the best move of the *Kaufman Test* and the ones played by the ANNs. The value of 20 in position 22 for the MLP is symbolic, since the ANN chose a move leading to a forced mate.

Multilayer Perceptron Considering the performance obtained by the MLP, as it is possible to see from Table 8.1 the ANN only plays the move which is prescribed by the test 2 times, namely in positions 3 and 6. Both positions correspond to a board state in which one piece is under attack and the agent has only one possible safe square on the board to move it if it does not want to lose any material.

[1]The test has been downloaded in the PGN format from `http://utzingerk.com/test_kaufman`

At a first look, this result can seem disappointing, since the moves that are being played by the ANN are almost every time different from the ones prescribed by the test. An analysis of the Δcp value shows much more promising results. In fact, the moves that are chosen by the ANN are very rarely blunders and rarely even end up in a *Losing* position. In order to see this clearly, we present Figure 8.1.

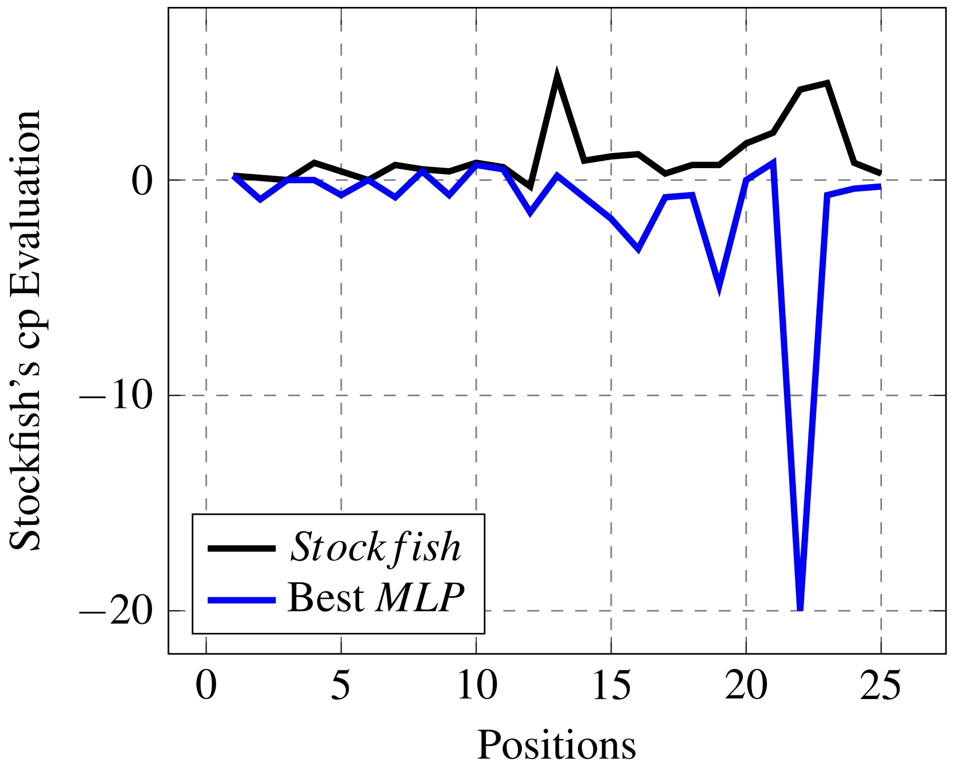

FIGURE 8.1: The performance of the Bitmap MLP on the Kaufman Test

The figure shows on the x axis the total amount of 25 positions prescribed by the test, while on the y axis the Cp value that has been assigned to the position by a deep analysis of *Stockfish*. The goal of the ANN is to commit to a move which has a Cp evaluation as close as possible as the one prescribed by *Stockfish*, hence a perfect scenario for this test, would see the 2 lines perfectly overlap, meaning that the move which would be played by the ANN would exactly correspond to the one prescribed by Kaufman. However, as already introduced this only happened twice in the experiment. On the other hand, the moves played by the MLP have still very similar evaluations to the ones that are suggested by the test. This is particularly true for the first 14 moves where the Δcp value is of by only ≈ 0.5. This means that even though the move that is played by the ANN is not the one recommended by the test, the ANN still chooses a very valuable alternative that does not compromise its winning chances.
The ANN still wastes some winning chances, and even commits blunders, in some positions. An example in which it chooses a move that leads to a drawing position instead of keeping the position winning can be seen in position 13. In this case the situation on the board would have been completely winning, by having a Cp value of ≈ 5, but the move chosen by the ANN has a Cp value of ≈ 0. Still, it is important to notice that the move chosen by the ANN does not lead to a losing position.
Unfortunately this cannot be said for every position in the test. In fact, as can be seen by what happens in positions 16, 19 and 23 the gap between the two plots is very large. This means

that in these cases the move which has been played by the ANN actually ends up in a losing position, meaning that something went wrong during the evaluation process of the position.

Convolutional Neural Network A similar analysis has been performed in order to establish the quality of the evaluations given by the CNN. Again, the amount of times in which the move made by the ANN matches with the one proposed by the test is 2. Both cases correspond to the positions in which also the MLP committed to the move prescribed by the test. If we explore the trend of the evaluations given by the ANN compared to the ones given by *Stockfish* we see a very similar behavior to what has been observed for the previous neural architecture. The analysis is presented in Figure 8.2

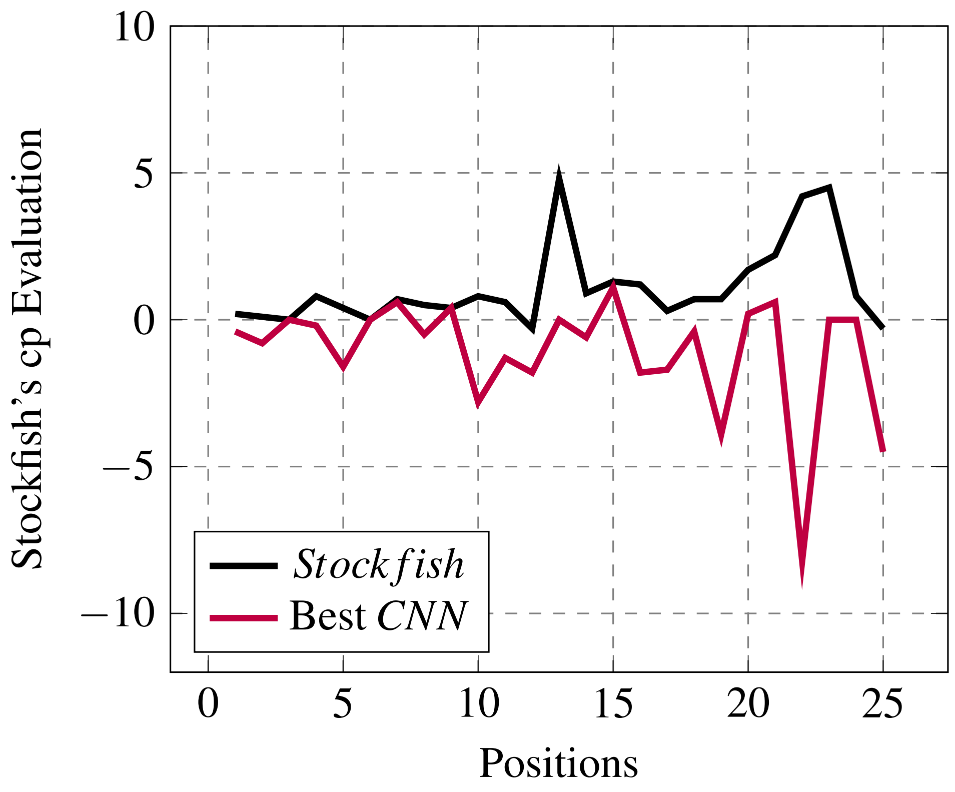

FIGURE 8.2: The performance of the Features CNN on the Kaufman Test

As can be seen in the line plots, the 2 evaluation trends fit again very well in most of the cases. This again means that even though also the CNN does not make the move prescribed by the test, the alternatives it commits to are still very valuable. In this case this is true for the first 10 moves of the test, where an important difference can be seen between the performance of the MLP and the one of the CNN for position 10. In this case the CNN actually commits to a move leading to a *Losing* position, going from a Cp value of ≈ 0.25 to one of ≈ -4. It is already interesting to highlight how the previous neural architecture did not blunder this position. In fact, if we count the amount of times this ANN made a move leading to a losing evaluation we observe an interesting behavior. The amount of times this happened is again 3, but only in position 19 both the MLP and the CNN evaluated wrongly the exact same position. This is actually very interesting since it shows that even though the performances of both ANNs are very similar when we investigate their Mean Squared Errors, their evaluation performances change according to the different chess positions that need to be evaluated. However, the general quality of the moves that is played is the same, very rarely both ANNs commit to a move leading to what can be considered as a *Losing* position. To present this more in detail we propose the following analysis.

General Analysis In order to easily compare the differences between the evaluations made by the 2 ANNs we present Figure 8.3 in which the previously mentioned results are plotted together. We define a threshold interval between -1.5 and 1.5 that corresponds to a *Winning/Losing* interval inspired by what has has been done for the first classification experiment performed on *Dataset 1*. We consider as optimal moves all the moves that lead to an evaluation within this interval, meaning that the moves made by the ANNs do not lead to *Winning* or *Losing* positions but actually keep the situation on the board balanced.

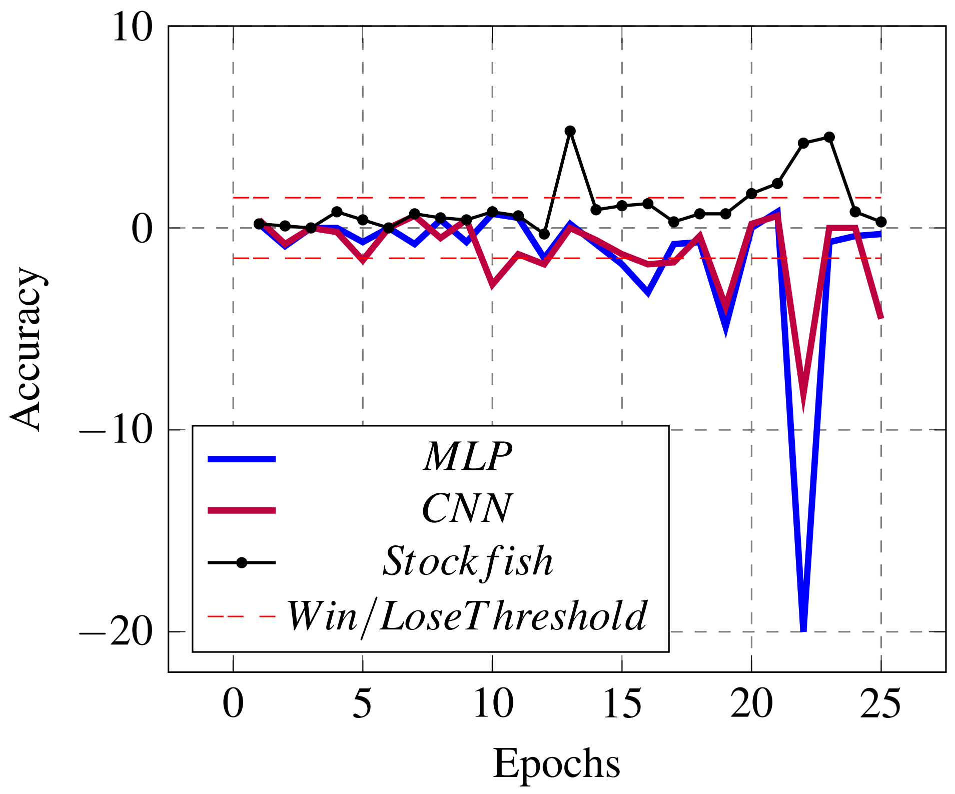

FIGURE 8.3: Analysis of the performances of the MLP and the CNN towards Stockfish on the Kaufman Test

As can be seen by Figure 8.3 the results are very promising. Most of the moves that have been made by both the MLP and the CNN end up in the *Draw* region. It is however important to highlight that this is not always the most correct strategy. In fact, as happened for position 13 an optimal ANN would choose a move that would keep the position *Winning*. However, it is still interesting to notice how the moves that are chosen by the ANNs usually do not end in the *Losing* area.
Furthermore, as has already been introduced, it is very interesting to highlight how even though in machine learning terms of performances the ANNs performed very similarly, their behavior on the test was different. We see from Table 8.1 that very rarely the MLP and the CNN opt for the exact same move, and also in terms of evaluations their moves are always slightly different between each other. Last but not least, it is really noteworthy what happened in positions 10, 15 and 25. In the first and third positions the MLP even makes a move which has almost the same evaluation to the one prescribed by the test, but the CNN makes in both cases a move that ends up in a *Losing* position. Similarly, in position 15 the opposite

phenomenon happens. It is the MLP which makes a move leading to a *Losing* position, while the CNN opts for a move which is only ≈ 0.25 evaluated differently when compared to the one of the test.
As suggested by all these results the ANNs still perform very similarly between each other, even though if we compute the average Δcp we see that the CNN performs slightly better compared to the MLP. In fact its average Δcp value is 1.55, which is 0.01 lower when compared to the one obtained by the MLP (1.56).
The Kaufman test is able to provide important insights about the playing strength of a computer system playing chess, however we believe that it might not be ideal for testing the playing strength of a system that does not make use of any lookahead algorithms in order to play the game of chess. This is mainly related to the fact that most of the positions of the test require the use of a relatively deep search in order to get solved according to the test. Nevertheless, the positions that are present in the test can still be used for evaluating the strength of systems that do not rely on lookahead exploration. In fact they cover a broad range of potential game situations which make this test extremely valuable. The positions cover board states of different game stages that were derived from very different openings and are always unique when it comes to the amount of pieces that are present on the board. Nevertheless, it is not possible to judge the strength of the ANNs solely based on the information if the move they make is the one prescribed by the test. Therefore, we believe that the introduction of the Δcp measurement, in combination with the test itself, can become a new appropriate testing bed for chess playing agents that discard lookahead algorithms to play chess.

8.2 Best MLP vs Best CNN

In this section we explore how well the MLP trained on the *Bitmap Input* and the CNN trained on the *Feature Input* actually play the game of chess. Hence, their performances have not only be evaluated on single particular positions, as with the Kaufman test, but over the entire games. This set of experiments has been programmed in order to find out which neural architecture would be the most suitable one for playing the game of chess without having to rely on lookahead.
Furthermore, we also explore the quality of the moves that are played by the ANNs in order to investigate how often the ANNs miss winning chances during the games. We do this in two ways: firstly by evaluating the moves that have been played with a deep analysis of *Stockfish*, which again assigns a *Cp* evaluation to the moves played by the ANNs. This evaluation serves as a ground truth for establishing the *Goodness/Badness* of the moves played. Secondly we investigate how many times during the game the ANNs enter in what is defined as a theoretically winning endgame, and check if they were actually able to convert this theoretical advantage into a win.

8.2.1 Structure of the Experiment

We have created 6 sets of different starting positions that are based on chess theory. Each set contains on average 3 different kind of openings. In total there are 20 openings from where the ANNs start playing the game. This is done in order to introduce sparsity and stochasticity to the experiment. In fact, if this would not be done the ANNs would keep playing the same game between each other over and over.
The precise starting positions will not be described in this section, however they can be found in the Appendix. It is however important to mention the differences within the 6 different sets in order to properly understand the results that have been obtained.

- Set1: contains the openings that start with the moves `1.e4, e5`
- Set2: contains the openings that start with `1.e4` but have a different reply such as `c5`
- Set3: contains the openings that start with `1.d4, d5`
- Set4: contains the openings that start with `1.d4` but have a different reply such as `Nf6`
- Set5: contains a set of openings in which a pawn is offered as a gambit in the first 5 moves
- Set6: contains openings that start with more uncommon moves such as i.e. `1.c4` or `1.Nf3`

The ANNs play the games according to the official Fide rules that are currently in use for tournaments between human players. Hence, the ANNs are able to claim a draw if the same position occurs on the board for in total three times during the game. Furthermore, a game is automatically drawn (without a claim by one of the players) if a position occurs for the fifth time on consecutive alternating moves. Lastly, the fifty-move draw rule is applied, in which a draw can be claimed once the clock of plys, since the last capture or pawn move, becomes equal or greater than 100. Figure 8.4 shows the percentages reporting the average outcomes of the games per set.

8.2.2 Results

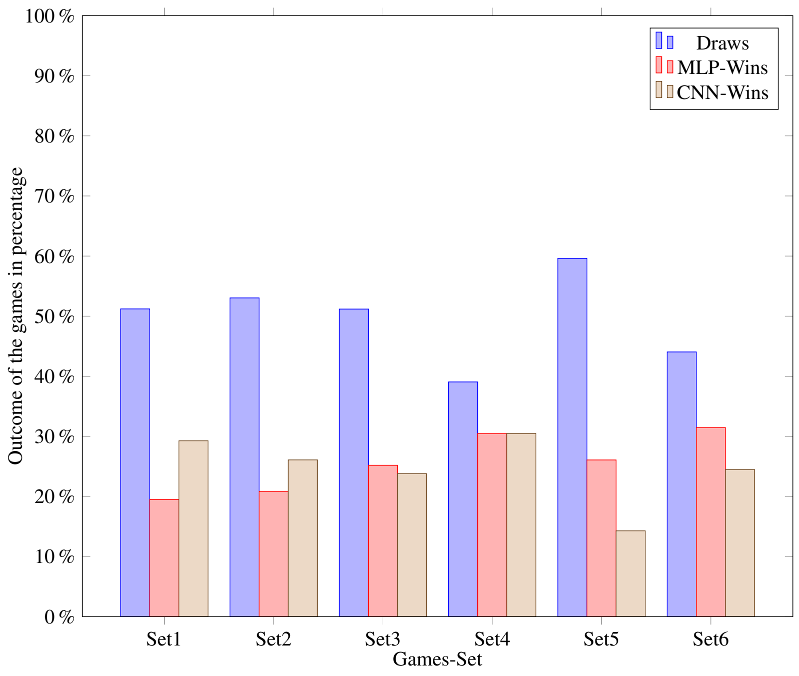

FIGURE 8.4: Bar plots representing the final outcomes of the chess games played between the MLP and the CNN

As shown by Figure 8.4 the main outcome of the games played between the ANNs is a Draw. This is not surprising and can be considered as a predictable result. In fact, reconsidering the performances that were obtained on the regression experiment both neural architectures performed very similarly. The MLP obtained a final Mean Squared Error of 0.0016 on the Testing Set, while the CNN outperformed this architecture by only 0.0001 obtaining an error

of 0.0015. This confirmed the hypothesis in chapter 7 about the limit that ANNs might have as function approximators of *Stockfish*. In fact, the results obtained showed two very similar performances for two extremely different architectures trained with even more different inputs. At the end of this experiment we see on average the MLP triumphing over the CNN by winning 25.6% of the games and losing 24.73% of the games that did not end in a draw. It would be possible to argue that the CNN should have been the architecture able to win most of the games since its MSE was lower than the one of the MLP. However, we believe that the difference between the 2 architectures is so small as to be irrelevant for the final playing performances of the ANNs.
What is however much more interesting to highlight is how the first two sets of games see the CNN winning over the MLP, while the last two ones show the opposite result. Between these wins we observe on Set3 and Set4 a less strong prevalence of one neural architecture over the other. If it is true that on Set3 it is actually the MLP being the "Winning" architecture, on Set4 none of the two ANNs established itself as best ANN.
This is particularly interesting since it again shows how there are some internal differences in how the ANNs evaluate different chess positions despite having performed very similarly from a machine learning perspective. It is in fact noteworthy that the CNN managed to win the set of openings that start with `1.e4` and that the MLP prevailed in the opening sets that have a complete different type of opening. Furthermore, the games that start with `1.d4` have been the games in which neither the CNN nor the MLP managed to significantly outperform each others. `1.d4` is often considered as a much safer opening if compared to the ones present in the first and last two sets, and the results obtained in this experiment seem to agree with this chess theory.

8.2.3 Quality of the Games

In order to understand the quality of the moves that have been played by the ANNs during the previous experiment we have evaluated them with a deep analysis of *Stockfish*. We have analyzed the Cp values that the engine assigned to the moves played by the ANNs and explored the trend that these values had during the games. This allowed us to understand how many winning chances the ANNs missed while playing and how consistent their game playing quality was. This basically aims to answer the question whether the ANNs are able to build up an advantage while the game proceeds, and if so, are they able to convert this advantage into a final win?
To answer this question we have analyzed 18 games played by the ANNs. Out of these games 6 correspond to a win of the MLP playing with Black, 6 correspond to a win of the CNN playing White and 6 are games that ended up in a Draw with the MLP playing White in 4 out of the total 6 games. Our analysis will now be described.

MLP Wins The first neural architecture on which we have performed our analysis on is the MLP winning its set of games versus the CNN while playing with the Black pieces. In this case the more negative the evaluations assigned by *Stockfish* to its moves are, the better it is for the MLP. In fact, since it is playing *Black* the main goal is to obtain as negative evaluations as possible during the game. We present our analysis in Figure 8.5.

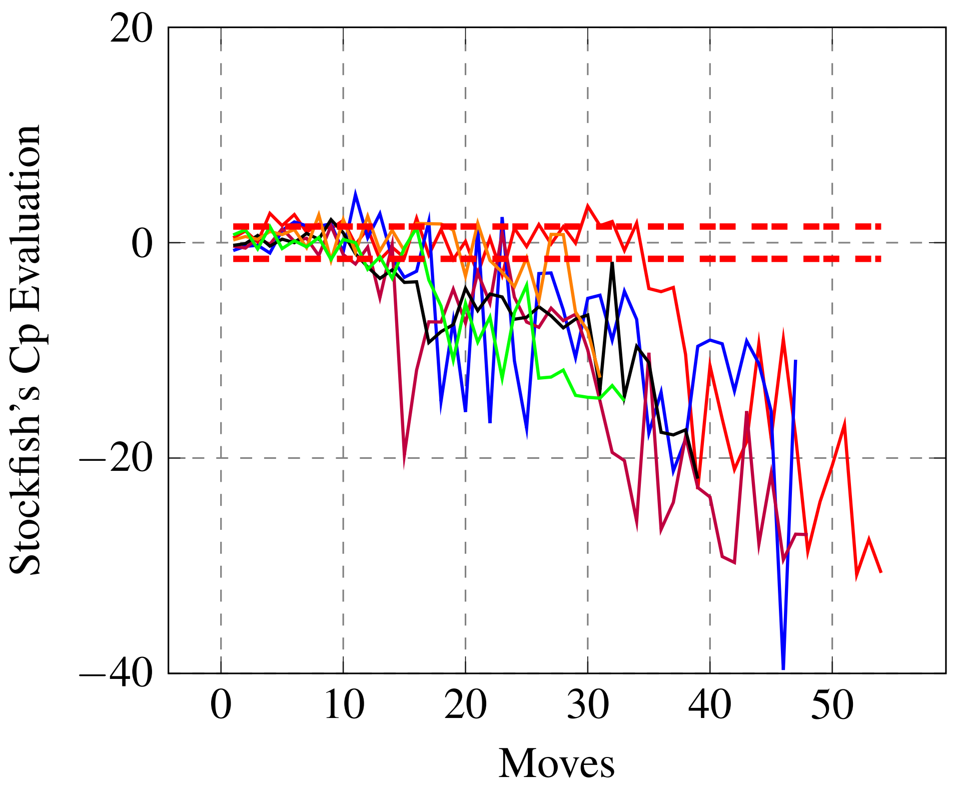

FIGURE 8.5: Quality analysis of the moves leading to a win for the MLP

The figure represents on the x axis the amount of moves that the MLP required in order to mate its opponent, while on the y axis *Stockfish's* evaluation is represented. We again define the region between -1.5 and 1.5 as the Draw region as has been done for the analysis performed on the Kaufman Test. Each game is represented by a different color and corresponds to one of the 6 opening sets previously presented.
As we can see from the figure the results obtained are very promising, we can clearly see how the MLP is able to build up its advantage over time and finally convert it into a win. In fact, the evaluations assigned by *Stockfish* generally become more negative with the progress of the game. It is however possible to observe how the MLP sometimes is not able to keep the advantage that it is able to obtain. This can be seen clearly by the game plotted in blue. Between moves 17 and 24 it happened at least 3 times that the board situation would have had a *Cp* evaluation of ≈ -15, however the moves made by the ANN ended up in the Draw region. What is however worth to point out is how, despite having missed some winning chances, the MLP very rarely exceeded the Draw region while playing. This is in line with what was observed on the Kaufman Test. Even though the ANNs did not commit to the actual optimal move, the alternatives that were chosen were in general very valuable. Similarly, in this case, even though it would have been theoretically possible to win the game more quickly the ANN did not make moves that would have compromised their chances of winning.
Furthermore, it is also very interesting to see how for the first ≈ 15 moves the position on the board was extremely balanced by being between the -1.5 and 1.5 range. This means that the last part of the opening and the beginning of the middle game were actually played by both ANNs without any blunders.

CNN Wins We performed a similar analysis on the games that were won by the CNN playing *White*. In this case however, the value of the evaluations has to be flipped in order to judge the quality of the moves that were played. In fact, the more positive the evaluations are the higher the chances of winning the game. We present the results obtained in Figure 8.6.

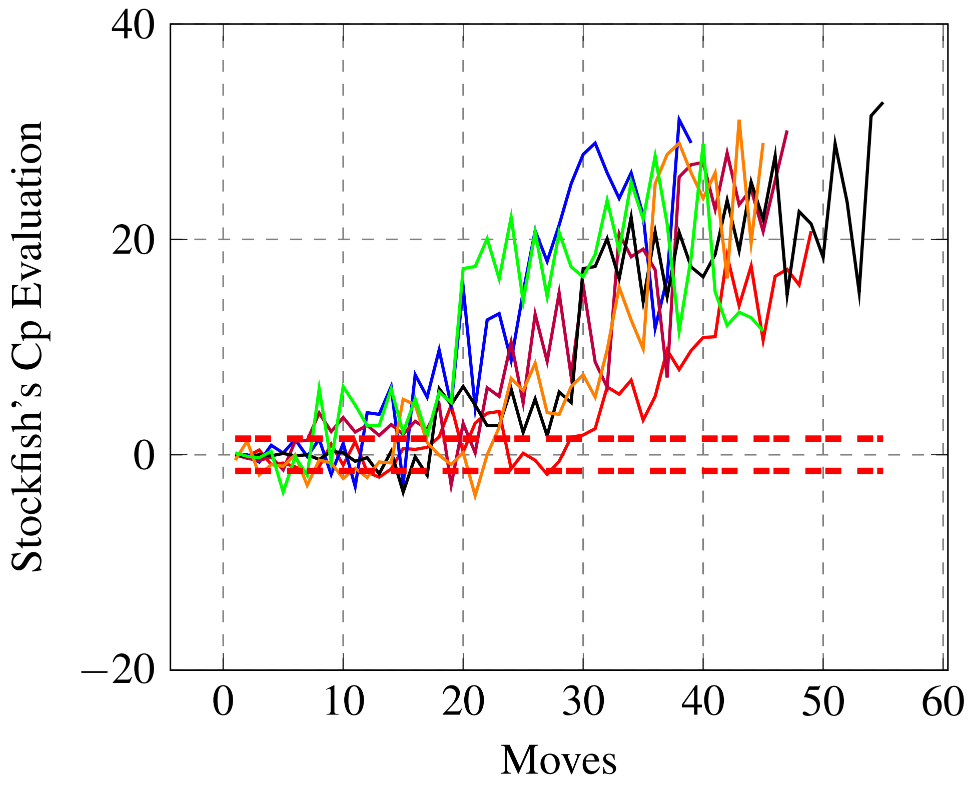

FIGURE 8.6: Quality analysis of the moves leading to a win for the CNN

The results obtained by the CNN are very similar to the ones obtained by the MLP, in fact also in this case it very clearly observable how the ANN starts building up its advantage over time. Similarly to what has been presented in Figure 8.5 we see that the positions remained within the Draw range for ≈ 20 moves and then the ANN managed to slowly enter in definitely winning positions that allowed it to finally win the game. We again see some fluctuations in the Cp trend, meaning that sometimes even though the ANN manages to obtain a winning position on the board it is not able to completely keep it. However, we also clearly see that the moves played by the ANN do not end in a losing position. This nicely shows how the neural architecture mostly commits to moves that in the worst case scenario would guarantee a draw.

Draws The final analysis that we performed consisted in evaluating the games that had as final outcome a Draw. In this case we would expect an optimal scenario in which the evaluations assigned by *Stockfish* to the moves played by the ANN are as close as possible to 0, or at least within the $-1.5, 1.5$ range. This would in fact mean that both ANNs are able to neutralize each other and do not give to their opponent the chance to end up in a winning position. However, in this case the results that we have obtained are a little bit less promising when compared to the previous two ones. We present them in the following figure.

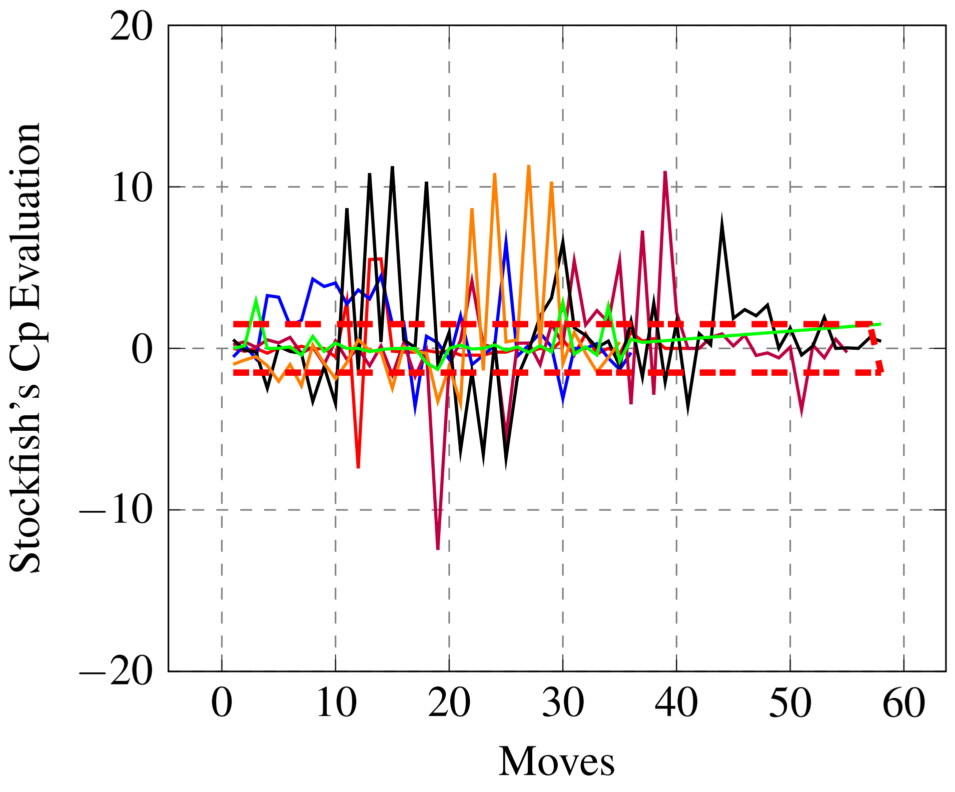

FIGURE 8.7: Quality analysis of the moves leading to a Draw between the ANNs

Figure 8.7 clearly shows how in some of the games one neural architecture would have had the chance to clearly win the game. As can be seen by i.e. the black and orange lines the evaluations assigned by *Stockfish* between moves 10 and 30 were clearly winning for one of the 2 players. More in detail, if we count the amount of times the evaluations reached a value of ≈ 10 we can clearly state that the ANNs missed at least 3 winning chances per game.
It is however very important to highlight that these evaluations have been given by a deep analysis of *Stockfish*. This means that the engine evaluates those positions as completely winning thanks to its deep lookahead algorithm. Both our ANNs do not make use of this technique at all. This could mean that those positions would have been winning only if a particular tactical combination would have been played on the board. Of course, as has also been seen during the Kaufman test, these kinds of chess positions are very complicated to evaluate for the way our ANNs approach the game of chess. These results show how the neural architectures still require some future improvement in order to play as well as the best human players. In fact, a human Grandmaster would rarely miss three winning chances in a row as happened for example by the ANN playing the game marked in black.
Despite this sour note, it is also important to mention that in one random picked game the ANNs managed to play a game in which they completely managed to neutralize each other. This is the game which is marked in green, in this case the ANNs played for almost 60 moves without exceeding the Winning/Losing threshold at all.

Theoretical Winning Endgames The final experiment that we have performed aimed to investigate the performances of the ANNs on theoretical winning endgames. In chess, there is a particular group of positions that occur in the last stages of the game, in which, no matter the strength of the opponent is, the positions will always be winning for one of the two players. Winning them can however be very complicated and is a matter of fine technique that requires extremely precise calculations. Figure 8.8 shows an example of one of the hardest theoretical winning endgames: *White* needs to mate its opponent only with the use of its *King, Bishop* and *Knight*.

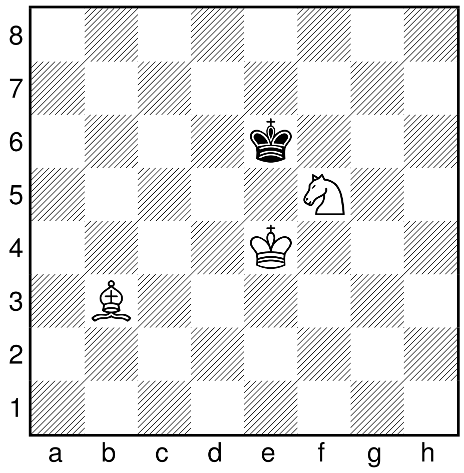

FIGURE 8.8: Example of a theoretical winning endgame.

Since the amount of these endgames is very large we have created a subset of them as follows: we first define an endgame position as a particular board state in which the sum of the values of all pieces on the board does not exceed a total value of 16. Secondly, we check if this condition is satisfied in the game that is being played by the ANNs that are competing against each other by matching the position to an existing database [2]. If it is, we assign a *Cp* evaluation to the board by again making use of a deep analysis of *Stockfish*. If the evaluation is greater than 6, the position is used for the experiment. The main goal of the experiment was to understand how many times the ANNs managed to actually win the game if they entered in a theoretical winning endgame.
Out of all the games the MLP played 62% of the total endgame positions, while the CNN played the rest 38%. Surprisingly, both ANN architectures managed to always win the endgame, meaning that they always converted a theoretical winning advantage into a win. It is however worth mentioning that the amount of moves the ANNs required to win the position, was not the minimum one. This means that both neural architectures could perform more efficiently. Furthermore, since the ANNs played against each other, it is important to highlight that they might not have always opted for the optimal defense while playing the endgame and hence increased the winning chances of the opponent. On the other hand it is nice to point out that since they played the games according to the previously mentioned official Fide rules, they always managed to actually win the positions, without letting the game finish in a *Draw* due to for example the fifty-move draw rule.

[2]`http://www.k4it.de/index.php?lang=en&topic=egtb`

Chapter 9

Conclusions

To the best of our knowledge, this thesis corresponds to the first attempt in training a computer program to play chess, while not making use of any lookahead algorithms to play the game at a high level.
All the information that has been presented in this work aimed to investigate if this research challenge could actually be accomplished. The results that have been presented show how it is indeed possible to make use of Artificial Neural Networks (ANNs) to train a program that is able to evaluate chess positions as if it would be using lookahead algorithms. However, the system that has been presented is still far from the strongest existing chess engines and best human players. Nevertheless, the level reached by the best performing ANNs is still remarkable since it reached an Elo rating of ≈ 2000 on a reputable chess server [1]. The ANN played in total 30 games according to the following time control: 15 starting minutes are given per player at the start of the game, while an increment of 10 seconds is given to the player each time it makes a move. The ANN played against opponents with an Elo rating between 1741 and 2140 and obtained a final game playing performance corresponding to a strong Candidate Master titled player. The games show how the ANN developed its own opening lines both when playing as *White* and as *Black* and performed best during the endgame stages of the game, when the chances of facing heavy tactical positions on the board are very small. The chess knowledge that it learned, allowed it to easily win all the games that were played against opponents with an Elo rating lower than 2000, which correspond to $\approx 70\%$ of the total games. However, this knowledge turned out to be not enough to competitively play against Master titled players, where only 2 *Draws* were obtained. We report in Figure 9.1 one of the most interesting positions that the ANN faced while playing on the chess server.
The ANN, playing *White*, won this complicated endgame in which, even though *Black* has an advantage of one *Knight*, it managed to promote one of its pawns to *Queen*. The situation on the board presented in the figure is particularly interesting. *Black* is checking *White's King* with the *Bishop*, hence *White* plays `Kxh5`. After this move, *Black* has again the chance of checking *White* with the move `Be2+`. *White* moves its *King* back to `g5`, where it gets checked again by *Black's Bishop* in `d3`. This is a crucial moment in the game, since if *White* would go back to `h5` *Black* would be able to get a *Draw* by repetition. However, the ANN, without having been specifically trained on it, moved its *King* to `f7`, avoiding the *Draw* and winning the endgame after the following sequence of moves: `Nxf6`, `Kxf6`, `Bh7`, `Kf7`. These results clearly give an answer to the main research question of this thesis which was:

Is it possible to train a computer program to play chess at an advanced level, without having to rely on lookahead algorithms during the evaluation process?

Besides having shown that it is possible to play chess without making use of lookahead algorithms, we believe that this thesis contributes to the research field of *chess-programming*

[1] https://chess24.com/en

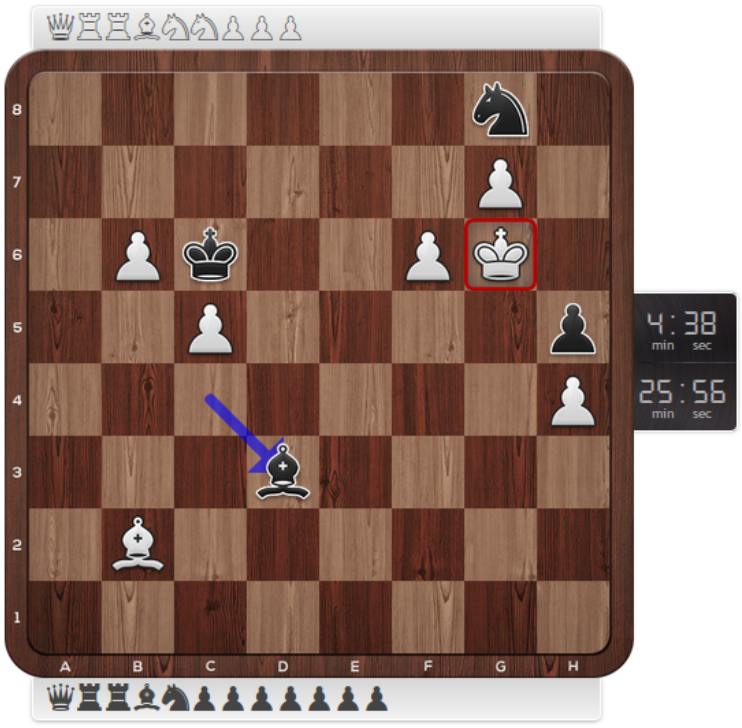

FIGURE 9.1: A remarkable endgame won by the ANN against an ≈ 1900 Elo player.

in several important ways. In fact, we managed to provide insightful answers to the secondary research questions that have been presented at the beginning of this work in chapter 1. The first main contribution is related to the research question:

How should ANNs be trained in order to master the skill of looking ahead without concretely making use of specifically designed algorithms?

We answer this by describing the methods presented in chapter 2 in which we establish a training procedure specifically designed to pursue this goal that we formally extended in chapter 5. We show how to make use of the capabilities that ANNs have as universal approximators of any mathematical function in order to approximate as good as possible the evaluation function of a strong chess engine. Secondly we give an answer to the research question:

How should the chess board be represented to the ANNs? And which input representation is able to help the ANNs maximize their performances?

We show how the already used *Bitmap representation* works really well when used in combination with a Multilayer Perceptron (MLP). However, we also show in chapter 6, how this input representation is not as effective when a Convolutional Neural Network (CNN) is used. Furthermore in the same chapter, we also show how providing the ANNs with information related to the strength of the pieces, in addition to whether they are present on the board or not, is counter productive for both MLPs and CNNs.
The use of the latter neural architecture in this work deserves particular attention. CNNs have so far been used only once in chess (Oshri and Khandwala, 2016), nevertheless the results presented in this work show how this type of ANN can still be a powerful architecture when it comes to chess. Hence we managed to answer the research question:

Is it possible to use Convolutional Neural Networks in chess?

We show that this is possible both in chapter 6 and in chapter 7. To do so, it is extremely important to design the ANN architecture in such a way that the input preserves as much geometrical information as possible during the training process, as has been highlighted in chapter 4. Training this ANN results in lower performances when compared to the ones obtained by the MLP on standard *state of the art* board representations. However, when combined with the novel representation presented in chapter 7 its performances become even better than the ones obtained by MLPs. Furthermore, we also show how the training time required by this ANN is much more efficient when compared to the ones required by the MLP, if appropriate `GPU` support is provided. This is particularly the case for the CNNs trained on the *Feature Input* which converged in ≈ 36 hours for the experiment performed on *Dataset 4*. These results give an answer to the question:

Assuming it is actually possible to make use of ANNs to teach a program to play without relying on any lookahead algorithms, how much time will the training process take?

Furthermore, we extend the use of *state of the art* testbeds that aim to evaluate the strength of chess playing programs to systems that do not make use of lookahead algorithms. We do this by extending the already existing Kaufman Test in chapter 8 by introducing the Δcp value, a unit of measurement specifically designed to evaluate the quality of moves played by systems that do not make use of any lookahead algorithms. Despite these contributions there is still a lot of future work that can be done in order to improve the approach that has been presented in this thesis. Some potential interesting ideas for future research will now be presented.

9.1 Future Work

Even though the ANNs managed to play chess at a high level without making use of lookahead algorithms at all, the chess knowledge they learned from the databases that we described in chapter 2, turned out not to be enough to win against Master titled players. During these games the ANNs lost most of the games already during the middle game when, due to tactical combinations they lost material on the board. As also supported by the results obtained on the Kaufman Test presented in chapter 7, it seems that in order to completely master the game of chess some lookahead is required. Hence we believe that the most promising approach for the future will be to combine the evaluations given by the current ANNs together with quiescence and selective search algorithms. By doing so the ANNs will be able to avoid the horizon effect (Berliner, 1973) and also perform well on tactical positions.
Secondly it could be possible to improve the performance of the ANNs through the use of *Reinforcement Learning* (RL). Mastering a board game by making use of RL from scratch can be very hard. In fact, a lot of exploration is required before finding a good game playing policy. Our system provides a solution to this issue since it has already learned a lot of chess knowledge that allows it to play the game as a Candidate Master. Hence, the training process that would lead to Grandmaster performances like the ones presented by (Lai, 2015) would cost much less time.
Thirdly it would be possible to extend the *Feature Input* by adding more feature maps to the ones that we have proposed in chapter 7. Furthermore, since this input representation has only been used on CNNs, an interesting future idea would be the one of representing the same information more efficiently. In such a way also MLPs could benefit from it.

Last but not least the dataset on which the ANNs has been trained in this thesis consisted of $\approx$ 3,000,000 of positions, it could be possible that increasing its size would improve their performances. It could be interesting to explore if training the ANNs on more games would compensate the need for selective search algorithms or if, on the other hand, the game playing performances obtained in this thesis can not be improved upon.

This thesis ends with the hope of having provided the reader with insightful knowledge about the fields of Machine Learning and chess. In addition to that, we hope that this work can become a starting point for other researchers that will create their own chess playing programs that do not make use of any lookahead algorithms to master the game of chess.

Appendix A

Appendix

The list of the 20 different initial positions that have been used for the experiment presented in Chapter 7. The experiment aimed to test which Artificial Neural Network between the Multilayer Perceptron and the Convolutional Neural Network is stronger.
The positions are presented in the Forsyth–Edwards Notation (FEN), a standard way for describing chess positions which allows to restart a game given the starting board state.

- `r1bqkbnr/pppp1ppp/2n5/1B2p3/4P3/5N2/PPPP1PPP/RNBQK2R w KQkq -`
- `r1bqkbnr/pppp1ppp/2n5/4p3/2B1P3/5N2/PPPP1PPP/RNBQK2R w KQkq -`
- `r1bqkbnr/pppp1ppp/2n5/4p3/3PP3/5N2/PPP2PPP/RNBQKB1R w KQkq -`
- `rnbqkbnr/pppp1p1p/8/6p1/3PPp1P/5N2/PPP3P1/RNBQKB1R w KQkq -`
- `rnbqkbnr/ppp2ppp/8/3pp3/4PP2/8/PPPP2PP/RNBQKBNR w KQkq -`
- `rnbqkb1r/1p2pppp/p2p1n2/8/3NP3/2N5/PPP2PPP/R1BQKB1R w KQkq -`
- `r1bqkbnr/pp1ppppp/8/8/3QP3/8/PPP2PPP/RNB1KB1R w KQkq -`
- `r1bqkbnr/pp1ppp1p/2n3p1/8/3NP3/8/PPP2PPP/RNBQKB1R w KQkq -`
- `rn1qkbnr/pp2pppb/2p4p/7P/3P4/6N1/PPP2PP1/R1BQKBNR w KQkq -`
- `rnbqkbnr/pp3ppp/4p3/2ppP3/3P4/5N2/PPP2PPP/RNBQKB1R w KQkq -`
- `rnbqkb1r/ppp1pp1p/3p1np1/8/3PPP2/5N2/PPP3PP/RNBQKB1R w KQkq -`
- `rnbqkb1r/pp3ppp/2p1pn2/3p4/2PP4/4PN2/PP3PPP/RNBQKB1R w KQkq -`
- `rnbqkbnr/ppp2ppp/8/4P3/2P5/4p3/PP3PPP/RNBQKBNR w KQkq -`
- `rnbqk2r/ppp1ppbp/3p1np1/8/2PPP3/2N2N2/PP3PPP/R1BQKB1R w KQkq -`
- `rn1qk2r/3pppbp/5np1/2pP4/4P3/5NP1/PP3P1P/RNBQ1K1R w KQkq -`
- `rnbqk1nr/ppp2ppp/3b4/8/8/5N2/PPPPP1PP/RNBQKB1R w KQkq -`
- `rn1qkb1r/ppp2ppp/4bn2/8/3P4/2N5/PPP2PPP/R1BQKBNR w KQkq -`
- `rnbq1rk1/pp2ppbp/3p1np1/2p5/2P5/2NP1NP1/PP2PPBP/R1BQ1RK1 w KQkq -`
- `rn1qkbnr/pbpp1ppp/1p2p3/8/8/3P1NP1/PPP1PPBP/RNBQK2R w KQkq -`
- `rnbqkb1r/pppppppp/5n2/6B1/3P4/8/PPP1PPPP/RN1QKBNR w KQkq -`

Bibliography

Banerjee, Bikramjit and Peter Stone (2007). “General Game Learning Using Knowledge Transfer.” In: *IJCAI*, pp. 672–677.

Baum, Eric B (1988). “On the capabilities of multilayer perceptrons”. In: *Journal of complexity* 4.3, pp. 193–215.

Baxter, Jonathan, Andrew Tridgell, and Lex Weaver (2000). “Learning to play chess using temporal differences”. In: *Machine Learning* 40.3, pp. 243–263.

Berliner, Hans J (1973). “Some Necessary Conditions for a Master Chess Program.” In: *IJCAI*, pp. 77–85.

Block, Marco et al. (2008). “Using reinforcement learning in chess engines”. In: *Research in Computing Science* 35, pp. 31–40.

Bottou, Léon (2010). “Large-scale machine learning with stochastic gradient descent”. In: *Proceedings of COMPSTAT’2010*. Springer, pp. 177–186.

Boureau, Y-Lan, Jean Ponce, and Yann LeCun (2010). “A theoretical analysis of feature pooling in visual recognition”. In: *Proceedings of the 27th international conference on machine learning (ICML-10)*, pp. 111–118.

Campbell, Murray, A Joseph Hoane, and Feng-hsiung Hsu (2002). “Deep blue”. In: *Artificial intelligence* 134, pp. 57–83.

Charness, Neil (1991). “Expertise in chess: The balance between knowledge and search”. In: *Toward a general theory of expertise: Prospects and limits*, pp. 39–63.

Chellapilla, Kumar and David B Fogel (1999). “Evolving neural networks to play checkers without relying on expert knowledge”. In: *IEEE Transactions on Neural Networks* 10.6, pp. 1382–1391.

Clark, Christopher and Amos Storkey (2015). “Training deep convolutional neural networks to play go”. In: *International Conference on Machine Learning*, pp. 1766–1774.

David, Omid E, Nathan S Netanyahu, and Lior Wolf (2016). “DeepChess: End-to-End Deep Neural Network for Automatic Learning in Chess”. In: *International Conference on Artificial Neural Networks*. Springer, pp. 88–96.

Dean, Jeffrey et al. (2012). “Large scale distributed deep networks”. In: *Advances in neural information processing systems*, pp. 1223–1231.

Duchi, John, Michael I Jordan, and Brendan McMahan (2013). “Estimation, optimization, and parallelism when data is sparse”. In: *Advances in Neural Information Processing Systems*, pp. 2832–2840.

Eisert, Jens, Martin Wilkens, and Maciej Lewenstein (1999). “Quantum games and quantum strategies”. In: *Physical Review Letters* 83.15, p. 3077.

Finnsson, Hilmar and Yngvi Björnsson (2008). “Simulation-Based Approach to General Game Playing.” In: *AAAI*. Vol. 8, pp. 259–264.

Fogel, David B and Kumar Chellapilla (2002). “Verifying Anaconda’s expert rating by competing against Chinook: experiments in co-evolving a neural checkers player”. In: *Neurocomputing* 42.1, pp. 69–86.

Ghory, Imran (2004). “Reinforcement learning in board games”. In: *Department of Computer Science, University of Bristol, Tech. Rep.*

He, Kaiming et al. (2016). “Deep residual learning for image recognition”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778.

Hellsten, Johan (2010). *Mastering chess strategy*. Everyman Chess.

Herik, H Jaap van den, HHLM Donkers, and Pieter HM Spronck (2005). “Opponent modelling and commercial games”. In: *Proceedings of the IEEE 2005 Symposium on Computational Intelligence and Games (CIG’05)*, pp. 15–25.

Hinton, Geoffrey E et al. (2012). “Improving neural networks by preventing co-adaptation of feature detectors”. In: *arXiv preprint arXiv:1207.0580*.

Hochreiter, Sepp (1998). “The vanishing gradient problem during learning recurrent neural nets and problem solutions”. In: *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems* 6.02, pp. 107–116.

Hochreiter, Sepp and Jürgen Schmidhuber (1997). “Long short-term memory”. In: *Neural computation* 9.8, pp. 1735–1780.

Hornik, Kurt (1991). “Approximation capabilities of multilayer feedforward networks”. In: *Neural networks* 4.2, pp. 251–257.

Hornik, Kurt, Maxwell Stinchcombe, and Halbert White (1989). “Multilayer feedforward networks are universal approximators”. In: *Neural networks* 2.5, pp. 359–366.

Kaufman, Larry (1992). “Rate your own computer”. In: *Computer Chess Reports* 3.1, pp. 17–19.

Keys, Robert (1981). “Cubic convolution interpolation for digital image processing”. In: *IEEE transactions on acoustics, speech, and signal processing* 29.6, pp. 1153–1160.

Kingma, Diederik and Jimmy Ba (2014). “Adam: A method for stochastic optimization”. In: *arXiv preprint arXiv:1412.6980*.

Knuth, Donald E and Ronald W Moore (1975). “An analysis of alpha-beta pruning”. In: *Artificial intelligence* 6.4, pp. 293–326.

Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E Hinton (2012). “Imagenet classification with deep convolutional neural networks”. In: *Advances in neural information processing systems*, pp. 1097–1105.

Lai, Matthew (2015). “Giraffe: Using deep reinforcement learning to play chess”. In: *arXiv preprint arXiv:1509.01549*.

Lawrence, Steve et al. (1997). “Face recognition: A convolutional neural-network approach”. In: *IEEE transactions on neural networks* 8.1, pp. 98–113.

LeCun, Yann, Yoshua Bengio, and Geoffrey Hinton (2015). “Deep learning”. In: *Nature* 521.7553, pp. 436–444.

LeCun, Yann, Yoshua Bengio, et al. (1995). “Convolutional networks for images, speech, and time series”. In: *The handbook of brain theory and neural networks* 3361.10, p. 1995.

Levitt, Gerald M (2000). *The Turk, Chess Automation*. McFarland & Company, Incorporated Publishers.

Lucas, Simon M and Thomas P Runarsson (2006). “Temporal difference learning versus co-evolution for acquiring othello position evaluation”. In: *Computational Intelligence and Games, 2006 IEEE Symposium on*, pp. 52–59.

McCulloch, Warren S and Walter Pitts (1943). “A logical calculus of the ideas immanent in nervous activity”. In: *The bulletin of mathematical biophysics* 5.4, pp. 115–133.

Moriarty, David E and Risto Miikkulainen (1994). “Evolving neural networks to focus minimax search”. In: *AAAI*, pp. 1371–1377.

Murray, Harold James Ruthven (1913). *A history of chess*. Clarendon Press.

Nagi, Jawad et al. (2011). “Max-pooling convolutional neural networks for vision-based hand gesture recognition”. In: *Signal and Image Processing Applications (ICSIPA), 2011 IEEE International Conference on*, pp. 342–347.

Neyshabur, Behnam, Ruslan R Salakhutdinov, and Nati Srebro (2015). “Path-SGD: Path-normalized optimization in deep neural networks”. In: *Advances in Neural Information Processing Systems*, pp. 2422–2430.

Oshri, Barak and Nishith Khandwala (2016). "Predicting moves in chess using convolutional neural networks". In: *Stanford University Course Project Reports-CS231n*.

Papadimitriou, Christos H (2003). *Computational complexity*. John Wiley and Sons Ltd.

Park, Eun-Young, Sang-Hun Kim, and Jae-Ho Chung (1999). "Automatic speech synthesis unit generation with MLP based postprocessor against auto-segmented phoneme errors". In: *Neural Networks, 1999. IJCNN'99. International Joint Conference on*. Vol. 5. IEEE, pp. 2985–2990.

Patist, Jan Peter and MA Wiering (2004). "Learning to play draughts using temporal difference learning with neural networks and databases". In: *Benelearn'04: Proceedings of the Thirteenth Belgian-Dutch Conference on Machine Learning*, pp. 87–94.

Poggio, Tomaso and Federico Girosi (1989). *A theory of networks for approximation and learning*. Tech. rep.

— (1990). "Networks for approximation and learning". In: *Proceedings of the IEEE* 78.9, pp. 1481–1497.

Polyak, Boris T (1964). "Some methods of speeding up the convergence of iteration methods". In: *USSR Computational Mathematics and Mathematical Physics* 4.5, pp. 1–17.

Radford, Alec, Luke Metz, and Soumith Chintala (2015). "Unsupervised representation learning with deep convolutional generative adversarial networks". In: *arXiv preprint arXiv:1511.06434*.

Romstad, Tord et al. (2011). *Stockfish, open source chess engine*.

Rosenblatt, Frank (1958). "The perceptron: A probabilistic model for information storage and organization in the brain." In: *Psychological review* 65.6, p. 386.

Rost, Burkhard and Chris Sander (1994). "Combining evolutionary information and neural networks to predict protein secondary structure". In: *Proteins: Structure, Function, and Bioinformatics* 19.1, pp. 55–72.

Rumelhart, David E, Geoffrey E Hinton, and Ronald J Williams (1986). *Leanrning representations by back-propagating errors*. na.

Russell, Stuart and Peter Norvig (1995). "A modern approach". In: *Artificial Intelligence. Prentice-Hall, Egnlewood Cliffs* 25, p. 27.

Schaeffer, Jonathan et al. (2007). "Checkers is solved". In: *science* 317.5844, pp. 1518–1522.

Schaul, Tom and Jürgen Schmidhuber (2009). "Scalable neural networks for board games". In: *Artificial Neural Networks–ICANN 2009*, pp. 1005–1014.

Shkarupa, Yaroslav, Roberts Mencis, and Matthia Sabatelli (2016). "Offline Handwriting Recognition Using LSTM Recurrent Neural Networks". In: *The 28th Benelux Conference on Artificial Intelligence*.

Silver, David et al. (2016). "Mastering the game of Go with deep neural networks and tree search". In: *Nature* 529.7587, pp. 484–489.

Simonyan, Karen and Andrew Zisserman (2014). "Very deep convolutional networks for large-scale image recognition". In: *arXiv preprint arXiv:1409.1556*.

Sutskever, Ilya, Oriol Vinyals, and Quoc V Le (2014). "Sequence to sequence learning with neural networks". In: *Advances in neural information processing systems*, pp. 3104–3112.

Sutton, Richard S (1988). "Learning to predict by the methods of temporal differences". In: *Machine learning* 3.1, pp. 9–44.

Sutton, Richard S and Andrew G Barto (1998). *Reinforcement learning: An introduction*. Vol. 1. 1. MIT press Cambridge.

Szegedy, Christian et al. (2017). "Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning." In: *AAAI*, pp. 4278–4284.

Tesauro, Gerald (1990). "Neurogammon: A neural-network backgammon program". In: *Neural Networks, 1990., 1990 IJCNN International Joint Conference on*. IEEE, pp. 33–39.

— (1994). "TD-Gammon, a self-teaching backgammon program, achieves master-level play". In: *Neural computation* 6.2, pp. 215–219.

Thompson, Ken (1996). “6-piece endgames”. In: *ICCA Journal* 19.4, pp. 215–226.

Thrun, Sebastian (1995). “Learning to play the game of chess”. In: *Advances in neural information processing systems*, pp. 1069–1076.

Tieleman, Tijmen and Geoffrey Hinton (2012). “Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude”. In: *COURSERA: Neural networks for machine learning* 4.2, pp. 26–31.

Van Der Ree, Michiel and Marco Wiering (2013). “Reinforcement learning in the game of Othello: learning against a fixed opponent and learning from self-play”. In: *Adaptive Dynamic Programming And Reinforcement Learning (ADPRL), 2013 IEEE Symposium on*, pp. 108–115.

Van Seijen, Harm and Rich Sutton (2014). “True online TD (lambda)”. In: *International Conference on Machine Learning*, pp. 692–700.

Wan, Li et al. (2013). “Regularization of neural networks using dropconnect”. In: *Proceedings of the 30th international conference on machine learning (ICML-13)*, pp. 1058–1066.

Wiering, Marco and Martijn Van Otterlo (2012). “Reinforcement learning”. In: *Adaptation, Learning, and Optimization* 12.

Wu, Jianxin (2016). “Introduction to convolutional neural networks”. In: *National Key Lab for Novel Software Technology Nanjing University, China*.

Zeiler, Matthew D (2012). “ADADELTA: an adaptive learning rate method”. In: *arXiv preprint arXiv:1212.5701*.