# Learning to Play Chess from Scratch

Applying Deep Learning Models to Play Chess
with Function Approximation based Reinforcement Learning

Iulian Vlad Serban

MSc Machine Learning
University College London
London, United Kingdom
September 2014

Supervised by

David Barber
and
Peter Dayan

*This report is submitted as part requirement for the MSc Degree in Machine Learning at University College London. It is substantially the result of my own work except where explicitly indicated in the text. The report will be distributed to the internal and external examiners, but thereafter may not be copied or distributed except with permission from the author.*

*"If one could devise a successful chess machine, one would seem to have penetrated to the core of human intellectual endeavor."* (Newell et al., 1988)

# Abstract

This project investigates the possibility of developing a computer player for the game of chess in the direction of using minimal expert knowledge. In particular, we focus on extracting features for the heuristic evaluation function automatically.

We carry out two sets of experiments. The first uses a collection of games played by master-level chess players to train logistic regression, multilayer perceptron network, convolutional neural network and mixture of experts models with maximum likelihood to predict the winning player from a given board position. A binary representation of the chess board is used as the primary feature set, and it is contrasted to feature sets derived from two master-level chess engines. Based on these experiments we find that the multilayer perceptron networks and convolutional neural networks outperform the other models on the same feature sets, and further that they reach accuracies comparable to the logistic regression model with hand-crafted features derived from the chess engines. Contrary to previous research, this suggests that the hierarchical structure imposed by the multilayer perceptron networks and convolutional neural networks are appropriate for the task.

In the second set of experiments, the logistic regression and the more promising multilayer perceptron network are implemented into a chess engine. A modification of a state-of-the-art self-play training algorithm is proposed, and used to learn a heuristic evaluation function based on each model. In particular, each model is initialized based on the parameters found in the previous experiments on the game collection. The resulting programs are then played against each other to determine their playing strength. We find that the multilayer perceptron networks outperform the logistic regression models by a huge margin, when both are trained with the self-play procedure. This finding strongly suggests that the multilayer perceptron networks are able to learn useful non-linear features from a simple binary representation of the chess board, and that the modified self-play algorithm is an effective training procedure for hierarchical models when used in conjunction with proper initialization. Work on training larger and different model architectures, as well as evaluating these against human opposition, is expected to be a highly fruitful path for further research.

# Acknowledgements

I would like to thank my supervisors David Barber and Peter Dayan for embarking on this open-ended project with me, for their many helpful discussions and for their never-failing feedback. The work in this thesis has flourished from their invaluable guidance and broad knowledge base.

I would also like to thank Joel Veness for many helpful discussions and for the source-code to the *Meep* chess program, without which this project would not have been possible. In particular, I would like to thank David Silver for helpful discussions, for helping me to narrow down the scope of the project as well as ideas for concrete model implementations. I also thank Yoshua Bengio, Matko Bosnjak, Ketil Biering Tvermosegaard, Johannes Heinrich, Wolfgang Maass, Tim Rocktäschel, Marzieh Saeidi, Srinivas Turaga and Daan Wierstra for taking an interest in the project, reviewing parts of the thesis and general discussions.

I thank Tord Romstad, Marco Costalba and Joona Kiiski for the open-source chess program Stockfish. I thank the creators of the FICS Database (Free Internet Chess Server) who are known to the public only by the names of Seberg and Ludens, for the collection of chess games. I also acknowledge the use of the UCL Legion High Performance Computing Facility (Legion@UCL), and associated support services, in the completion of this work.

Finally, I thank my beloved Ansona for her patience, love and unconditional support. Compared to you, the quest for new scientific knowledge and artificial intelligence is but a distant echo in my heart.

# Contents

# List of Figures

# List of Tables

# List of Algorithms

# Chapter 1

# Introduction

## 1.1 Motivation

Since the 1950s the development of game-playing computer programs has been a major research area of artificial intelligence (Fürnkranz, 2001; Galway et al., 2008). Games are a convenient and well-studied platform for experiments in machine learning and, more generally, artificial intelligence (Samuel, 2000). They have a set of clearly defined rules and goals, which leaving the opponent aside makes it possible to quantify the performance of game-playing programs in a consistent way, e.g. by evaluating their playing strength. Many games are also complex enough to be intractable from a computational perspective, which implies that some sort of generalization has to take place for them to play at a high level. Since many people are also familiar with games such as chess, backgammon and checkers [1], it is easy to bring research achievements out beyond academia and in many cases often possible to interpret the inner workings of these programs.

It is arguable how far research has advanced in this domain. Although many programs are able to beat even the most talented human players, their models and algorithms can rarely be transferred to other games. Many programs are built directly on top of expert knowledge, and are highly specialized to only one particular game (Fürnkranz, 2001; Schaeffer, 1999). This makes it hard to apply the same methods to tackling new problems (Schaeffer, 2000).

[1] The game checkers is also known as Draughts.

Therefore, it is important to eliminate the need for human knowledge, and as Schaeffer argues, this implies that the *"... computer should be able to discover and refie all the knowledge it needs."* (Schaeffer, 1999). This brings us to the heart of our goal. That is, to engineer a chess player which relies less on expert knowledge than previous approaches, and which is easily transferable to other problem domains. We hope that such a solution will not only bear fruit in tackling other problems, but also contribute in the direction of general artificial intelligence and perhaps inspire new research to understand the human intellect (Newell et al., 1988).

Of all the games in the world, why choose chess? Chess is a deterministic [2] turn-taking two-player constant-sum game [3] of perfect information [4]. This makes it easy to analyse, yet the game exposes an intricate complexity formed by compounding a small set of rules (Russell and Norvig, 2009, pp. 161–162; Newell et al., 1988; Shannon, 1950). Its deterministic and perfect information properties frees us from having to consider probability distributions over states and outcomes, and to model the opponent player (Billings et al., 1998) [5]. For non-deterministic games or games without perfect information, it can be much more rewarding to exploit the weaknesses of the opponent rather than to assume a constant opponent and model the game itself perfectly. Chess is also a game which cannot be solved by sheer brute-force search due to its combinatorial complexity. This implies that some form of heuristic procedure, which balances search depth and selectivity together with an evaluation of the current position, e.g. the probability that one player will win, must be used. Therefore we cannot hope for a perfect solution. As Shannon argued half a century ago *" ... in chess there is no known simple and exact evaluating function ..., and probably never will be because of the arbitrary and complicated nature of the rules of the game ..."* (Shannon, 1950). At the same time, a solution which involves both search and heuristic evaluation would be fruitful to many other games, where similar approaches are now widely used (Russell and Norvig, 2009; Silver, 2014a, Lecture

[2] Chess is deterministic because, leaving aside the opponent, there is no element of chance. This is in contrast to, for example, Backgammon where dice rolls determine the available moves at each turn.

[3] A constant-sum game is a game where the sum of losses and rewards equals a constant for all outcomes. Chess can easily be defined as a constant-sum game, which is what we will do in section 6.1.

[4] A perfect information game, can be defined as a game where the game state is always completely known be all players. In chess, the position of all the pieces are known to both players during the entire game, which makes it a game of perfect information. In contrast, in the game Poker the cards of the opponents are not known, which makes it a game of imperfect information.

[5] As Billings et al. argue, in chess it is not crucial to model the opponent if playing strength is the only benchmark one cares about (Billings et al., 1998). Opponent modelling is more important in other domains, such as Poker games.

10). Taken together, this makes chess a simple domain to work in, yet sufficiently complex to experiment with recent large-scale machine learning models.

## 1.2 Research Scope and Methodology

The work carried out in this project is motivated by two recent developments. First, the work of Veness and others in (Veness et al., 2009) indicate that, given a suitable set of features, i.e. a function which computes a real-valued vector for any chess board position which reflects the characteristics of the game, it is indeed possible to train a chess player based only on self-play procedures, i.e. learning by playing the program against itself repeatedly. In fact, recent work in other game domains suggest that this may be possible without using any hand-crafted features (Mnih et al., 2013; Tesauro et al., 2014, Performance Results). Second, the increase in computational power and memory seen in recent years has made it possible to train larger machine learning models than ever before (Bengio, 2013, p. 6; Hof, 2014). The previous lack of computational resources is believed by some to have been a serious limitation in earlier work (Silver, 2014b; Silver, 2014a, Lecture 10), and is indeed a notable constraint in (Schraudolph et al., 2001; Thrun, 1995; Tesauro, 1995; Campbell et al., 2002; Utgoff, 2001, Mannen, 2003;Levinson and Weber, 2001; van Rijswijck, 2001). (Tesauro et al., 2014, Performance Results) provides further (unpublished) empirical proof in this direction for Backgammon.

Before embarking in the direction of (Veness et al., 2009), we also considered if it was possible to apply methods which have recently been successful in other game domains. In particular, we considered methods based on *Monte Carlo tree search*, which is a general set of procedures that use random sampling to determine their next move. For any move in a given position, the algorithms typically simulate random moves of each player until the end of the game. This is done many times over, after which they choose to play the move with the highest number of winning games. See (Browne et al., 2012) for an overview. These procedures have recently yielded success in several domains (Browne et al., 2012; Silver, 2014a, Lecture 10). In particular, they have been able to improve the performance of algorithms in Computer Go by a considerable magnitude (Lee et al., 2010; Gelly and Silver, 2008). However, after reviewing previous attempts to apply Monte Carlo tree search methods to chess, we found that this was unlikely to be a fruitful path (Ramanujan, 2012; Ramanujan et al., 2010; Arenz, 2012; Barber,

2014; Dayan, 2014). See appendix A for a brief discussion on the issues.

Thus, we have chosen to follow up on the work of (Veness et al., 2009) by attempting to automatically learn features for the heuristic evaluation function, e.g. characteristics that reflect the *value* or *advantage* one player has over another in a given board position, in conjunction with their self-play algorithm. Much work remains to be done in this area, despite the notable efforts of various researchers over the last decades (Fürnkranz, 2007), (Fürnkranz, 2001), (Richards, 1951). For example, as we shall elaborate in the next chapter, the heuristic evaluation function for the famous *Deep Blue* chess engine was hand-crafted over several years. In fact, in 1959 when Samuel constructed his famous checker playing program, he already stated that learning features from scratch was one of the most interesting directions for further machine learning research (Samuel, 1959).

Our project proceeds as follows. Since we are tackling a highly complex problem, we shall start with simple and easily interpretable experiments. These are supervised learning experiments where we learn to predict the winning player based on a collection of chess games. This is highly related to the task of ranking board positions, which can then be used to infer a preferred move, but remains untangled from any search procedure. Based on these results, we will evaluate our proposed models w.r.t. their representational power in capturing important features for ranking board positions. We then propose a modified version of the self-play algorithm in (Veness et al., 2009), implement the most promising models from the earlier experiment into a chess engine and train the models with the self-play procedure. The resulting models are then played against each other to determine their relative playing strength.

## 1.3 Contribution

Our contribution can be broken down into two parts. First, we show through our supervised learning experiments that hierarchical models, such as multilayer perceptron networks and convolutional neural networks, trained with simple well-established supervised training procedures, can extract useful non-linear features from chess game collections based only on a binary representation of the chess pieces. These models outperform their non-hierarchical counterparts on the same feature set, and reach performance comparable to non-hierarchical models based on hand-crafted features w.r.t. accuracy in predicting the game outcome. This suggests that

hierarchical models may be able to replace expert hand-crafted features in heuristic evaluation functions. Furthermore, our results indicate that the hierarchical structure of multilayer perceptron networks and convolutional neural networks are preferable to mixture of expert models. In addition to this, we also observed a significant amount of overfitting for even relatively small models, which may help to explain some previous failures on training multilayer perceptron networks on collections of chess games.

Second, we propose a modified self-play procedure based on (Veness et al., 2009), which we demonstrate with another set of experiments is capable of effectively training multilayer perceptron networks from a proper initialization point. Our results show that the multilayer perceptron networks beat their non-hierarchical counterparts by a substantial margin on the same feature set, which reaffirms that multilayer perceptron networks are able to extract useful non-linear features for the heuristic evaluation function and the potential for hierarchical models based on a simple binary representation to substitute hand-crafted features.

## 1.4 Thesis Structure

This chapter motivates the project and summarizes the research. Chapter two reviews previous work on programming a computer to play chess from the early days of Deep Blue until today, and introduces the important definitions and concepts for chess. Chapter three introduces the machine learning models used to learn the features for the heuristic evaluation function. Chapter four introduces temporal difference and the self-play learning procedure suggested by (Veness et al., 2009), and extends it to hierarchical probabilistic models. Chapter five discusses previous work on automatically learning the heuristic evaluation function in various games. Chapter six describes our experiments on the supervised learning problem using a chess game collection. Chapter seven describes our experiments on learning to play chess based on the modified self-play algorithm. Chapter eight concludes the project and gives directions for further research. The reader is encouraged to look further into the appendix for results, which also support the discussions and claims made.

## 1.5 Chess Terminology

We assume that the reader is familiar with the chess, and to an extent with Backgammon, Go, Othello [6] and checkers. To help the reader we list a set of common terms:

- *Chess board position*: Also referred to as a board position or simply a position. When evident from the context, we take this to be the same as the training instance or training example.
- *Ply*: A move by a single player. For example, 2-plies refers to one move by the player followed by one move by the opponent. Plies are normally used in the context of intervals (e.g. *Black can checkmate White in 4-plies*), while a move number usually refers to the absolute number of moves in the game (e.g. *White captured the rook at move 5*).
- *Rank*: A row of the chess board, numbered $1, \ldots, 8$.
- *File* A column of the chess board, numbered $a, \ldots, h$.
- *Back rank*: The row on which the player's king starts. For White this is rank one and for Black rank eight.

[6] Othello is also known as Reversi.

# Chapter 2

# Chess Engines and Heuristic Search

This chapter introduces important concepts related to chess, which will be refereed to throughout the thesis.

## 2.1 Deep Blue and Early Work in Chess

*Deep Blue* was a chess program developed over several years by a team of researchers at IBM in the mid 1990s. Its development culminated in 1997 when, in a match of six games, it managed to beat world-champion Garry Kasparov, who was described as *"possibly the strongest chess player who has ever lived"* by (Campbell et al., 2002; Hsu, 2002). The challenge to beat a world-champion chess player, which had stood for over 50-years, had at last been reached [1]. With it, many researchers in artificial intelligence anticipated not only many new tools and inventions, but also hoped to have finally reached the core of human intellectual endeavour (Newell et al., 1988; Hsu, 2002; Shannon, 1950). However, as was soon found out, the success of chess was limited. Many other games could not benefit significantly from the *"silver bullet"* of chess, namely the *deep heuristic search* (Schaeffer, 2000).

In a post-match paper Campbell, Hoane and Hsu point out a number of factors which lead to the success of Deep Blue (Campbell et al., 2002):

[1] Although we note that commentators have criticized the matches stating that Kasparov made serious mistakes, which he would not have made against a human opponent.

- a hardware chip chess search engine implemented with massive parallelization,
- a strong emphasis on search extensions,
- extensive use of a grandmaster game collection, and
- a complex evaluation function.

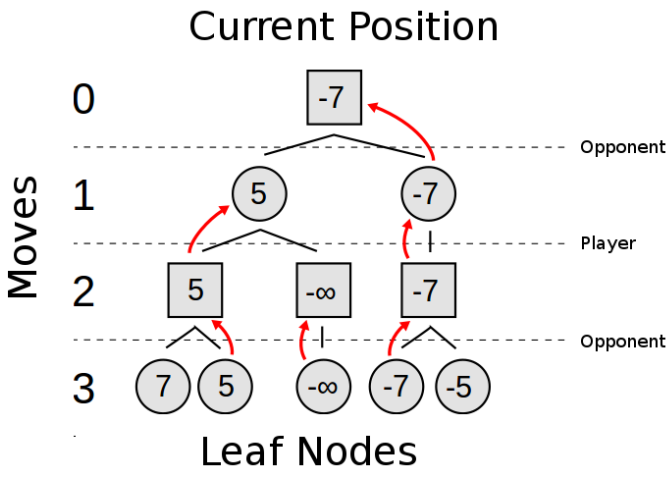
Clearly each of these factors are tailored uniquely for chess and require enormous amounts of work to implement. Although the factor of specially designed hardware chips and massive parallelization have been annihilated by the development of faster computer processing power, the remaining three factors are still very much present in modern chess engines (Russell and Norvig, 2009, p. 186). For example, the chess engine *RYBKA*, winner of the 2008 and 2009 World Computer Chess Championships, was believed to have won in considerable part due to its sophisticated evaluation function, which had been tuned by several grandmasters.

## 2.2 Minimax Search

The minimax algorithm is the workhorse for chess and many other games (Russell and Norvig, 2009, p. 165). The algorithm is most straightforward to understand in the context of non-stochastic perfect-information constant-sum two-player turn-based games, such as chess and Go. We can define the minimax algorithm as computing values of the nodes of a tree, where each node corresponds to a game position. Let the root of the tree be the current position in the game. Any move from the current position corresponds to a branch from the root node to a new node. Any move from this node corresponds to another branch, which leads to yet another node and so on. Suppose that a pay-off is given at the end of the game, e.g. $-1$, 0 and 1 for losing, drawing and winning, respectively. We then define the value at each node in the tree to be the expected pay-off from this game position onward under the assumption that both the player and opponent are playing optimally.

The minimax algorithm starts at the current game position and goes through all possible moves by both players until the end of the game. The endgame positions correspond to the leaves of the tree we defined above. The algorithm sets the values of these leaf nodes to be the reward of the player at the end of the game produced by these moves. It then works its way

backwards from the leaf nodes. The value at each node a level above the leaf nodes is taken to be the maximum value of all leaves connected to it if it's the player's turn to move, and the minimum value of all leaves connected to it if it's the opponent's turn to move. This process is illustrated in figure 2.1 and formalized in algorithm 1.




0

Figure 2.1: Minimax algorithm applied to a fictitious game. The leaf nodes are set to $7, 5, -\infty, -7, -5$ and we assume that it's the opponent's turn to move first. As in the last turn of the game it is also the opponent's turn to move, the nodes in the level above the leaf nodes are set to the minimum of the leaf nodes that they are connected to. This yields $5, -\infty$ and $-7$. At the level above these, it is the player's turn to move and we therefore set each node to the maximum of any node connected to it in the level below, which yields 5 and $-7$. At move zero it is the opponent's turn to move, and the root node is therefore set to the minimum of the two nodes below. Image adapted from `http://en.wikipedia.org/wiki/Minimax`

To make use of the algorithm, the player chooses the move to reach the node with the highest value from the root node. In effect, this strategy chooses the player's optimal move, given that the opponent will choose his or her optimal move afterwards, given that the player will choose his or her optimal move afterwards and so on. From a mathematical viewpoint, the algorithm guarantees the player following it a minimum reward at the end of the game. For example, suppose we define the value to be 1 for winning, 0 for drawing and $-1$ for losing, if the algorithm reaches a node with value equal to 1 at the current position, then it is guaranteed to be able to win the game.

**Algorithm 1** Minimax algorithm for two-player game

Function takes game state (position) as input and returns the minimax value of the current state.

```
1:  function EVALUATE(s)
2:      if GameHasEnded(s) then                  # Has the game ended?
3:          return PlayerScore(s)                # If so, return score for player
4:      else
5:          A ← GetAllMovesByPlayer(s)           # Get states reachable by player
6:          OptimalValue ← −∞
7:          for a ∈ A do
8:              B ← GetAllMovesByOpponent(a) # Get states reachable by opponent
9:              OptimalValue ← max(OptimalValue, min_{b∈B} Evaluate(b))
10:         end for
11:         return OptimalValue
12:     end if
13: end function
```

## 2.3 Heuristic Search and The Heuristic Evaluation Function

The minimax algorithm above works only for small games, such as tic-tac-toe, where it is possible to go through every possible move sequence and calculate the resulting value, but for larger games the algorithm is computationally intractable. Shannon estimated that in chess from a typical board position there are close to 30 legal moves available to the player, and that a typical game lasts 40 moves by each player until one of the players resign. This would yield an enormous $10^{120}$ game variations (Shannon, 1950). In principle though, it would suffice to only find the minimax values at each unique (legal) board position, a number which has been estimated to be $10^{46}$ (Chinchalkar, 1996). The computations could be carried out recursively starting from endgame positions and then looking up the positions preceding these. However, even if we were able to go through a million board positions per second and record their values, we would need more than $3 \times 10^{32}$ years to compute all the minimax values!

Having reasoned along these lines, Shannon was the first to propose that chess programs cut off their minimax search and apply a heuristic evaluation function instead (Shannon, 1950; Russell and Norvig, 2009). He proposed that board positions in chess should be given an

approximate value, w.r.t. the probability of one player winning, as a linear combination of features of the board position. He suggested counting the number of pieces on the board, where queen, rook, bishop, knight and pawn were weighted 9, 5, 3, 3 and 1 respectively. The player's own pieces would be added to the score and the opponent's would be subtracted from it. He further suggested giving positive (negative) values to positions where the player's (opponent's) rock is on an open file, to favour the side with greater mobility, and negative (positive) values to positions where the player's (opponent's) king is exposed, since an exposed king can easily be threatened. This was called the heuristic evaluation function.

The minimax algorithm would now be applied to a truncated tree of the game. The tree would contain only the next $n$ moves by each player, i.e. a search depth of $n$ ply, and at the leaf nodes it would assign the value according to the heuristic evaluation function. The values for the remaining nodes in the tree could be calculated as described earlier, and the move played would be the one corresponding to the node with the highest value connected to the root node.

Of course, many other types of features could be used to construct the heuristic evaluation function. See, for example, (Campbell et al., 2002). To give an example of a heuristic evaluation function outside chess, Samuel used the following in his famous checkers program (Samuel, 2000):

1. piece advantage,
2. denial of occupancy,
3. mobility,
4. a hybrid term which combined control of the center and piece advancement.

Shannon noted that this method should only be applied to relatively quiescent or 'quiet' positions, i.e. positions where the statistics of the game do not change suddenly (such as when swapping pieces). He gave a crude definition of this. A position is non-quiescent if any piece is attacked by a piece of lower value, if any piece is attacked by more pieces than pieces defending it or if any check exists from a square controlled by the opponent (e.g. if a check is coming from an opponent piece which is defended by another piece of the opponent). He used this definition to propose a simple type of what we now call *quiescence search*. In quiescence search,

a position is evaluated w.r.t. the heuristic evaluation function if it is quiescent by the definition above (or a similar definition) after which this line of the tree is truncated. If the position is not quiescent the tree is expanded with nodes corresponding to all possible moves starting from that position. Each new position is again classified as being either quiescent or not, and the previous step is applied repeated.

A simple variant of such a *quiescent search algorithm* is given in figure 2 for clarification. This should be applied to the leaf nodes of the minimax search tree to ensure a minimum search depth.

**Algorithm 2** Quiescent search algorithm for two-player game

Function takes game state (position), current search depth and max search depth as input. Function returns the heuristic value of the current position.

```
1:  function EVALUATE(s, depth, maxdepth)
2:      if GameHasEnded(s) then                  # Has the game ended?
3:          return PlayerScore(s)                # Then return score for player
4:      else if IsStateQuiescent(s)
          or depth = maxdepth then               # Otherwise, is state quiescent?
5:          return HeuristicEvaluation(s)        # Then return score for player
6:      else                                     # Otherwise, launch a new search from node
7:          A ← GetAllMovesByPlayer(s)           # Get states reachable by player in one move
8:          OptimalValue ← −∞
9:          for a ∈ A do
10:             B ← GetAllMovesByOpponent(a) # Get states reachable by opponent
11:             OptimalValue
12:                 ← max(OptimalValue, min_{b∈B} Evaluate(b, depth + 1, maxdepth))
13:         end for
14:         return OptimalValue
15:     end if
16: end function
```

Indeed, the minimax algorithm, heuristic evaluation function and quiescence search form the backbone of most modern chess engines today as well as in many other games (Russell and Norvig, 2009; Fürnkranz, 2001). The very same methods were also applied in the famous chess engine Deep Blue (Campbell et al., 2002), albeit with some changes. Psychological studies have also suggested that the distinguishing factor between strong and weak chess players lies in a mental process analogous to the heuristic evaluation function (Chase and Simon, 1973).

## 2.4 The Elo Rating System

Before we continue, it is important to formalize the definition of playing strength for both human and machine players. For many years chess players have been rated primarily by the *Elo rating system* developed by Arpad Elo (Elo, 1978). We follow the introduction given in (Coulom, 2014). The rating system is based on the expected outcome of a game (or pay-off) $E$ and the difference in ratings (scores) between the two players $D$:

$$E = \frac{1}{1 + 10^{D/400}}. \tag{2.1}$$

The assumption is that the game outcome is drawn from a distribution with a mean $E$, which depends on a single variable for each player indicating the players strength. The stronger a player the higher the variable should be. The definition given by eq. (2.1) can be reversed to estimate the rating difference between two players, and further to yield an estimated score and variance for each player based on a set of game outcomes. Henceforth, we will refer to the score (rating or playing strength) for a player as ELO score.

However in recent years, researchers have pointed out a number of faults in the traditional system. In particular, regarding the variance (or uncertainty) of the scores produced by the Elo rating system. Instead researchers have proposed to more formal probabilistic approach (Hunter, 2004), which is also taken in the open-source program BayesElo we use (Coulom, 2014). Let $P(\text{White wins}), P(\text{Black wins}), P(\text{Draw})$ be the probabilities for White wins, Black wins and drawing respectively. Then assume:

$$P(\text{White wins}) = \frac{1}{1 + 10^{(s_{\text{Black}} - s_{\text{White}} + \kappa)/400}}, \tag{2.2}$$

$$P(\text{Black wins}) = \frac{1}{1 + 10^{(s_{\text{White}} - s_{\text{Black}} + \kappa)/400}}, \tag{2.3}$$

$$P(\text{Draw}) = 1 - P(\text{White wins}) - P(\text{Black wins}), \tag{2.4}$$

where $s_{\text{White}}$ and $s_{\text{Black}}$ are the ELO scores for White and Black players respectively, and $\kappa$ is

a predefined constant [2]. For a set of ELO scores and game outcomes, Bayes rule yields:

$$P(\text{ELO scores}|\text{Game outcomes}) = \frac{P(\text{Game outcomes}|\text{ELO scores})P(\text{ELO scores})}{P(\text{Game outcomes})}, \quad (2.5)$$

where $P(\text{ELO scores})$ is the prior distribution over ELO scores. We assume this prior is uniform, which yields:

$$P(\text{ELO scores}|\text{Game outcomes}) \propto P(\text{Game outcomes}|\text{ELO scores}). \quad (2.6)$$

We can now infer a probability distribution over ELO scores given a set of game outcomes. BayesElo uses an algorithm called minorization-maximization (MM) to estimate precisely this distribution (Coulom, 2014; Hunter, 2004).

[2]The constant used in BayesElo is found by maximum likelihood on $29,610$ games of Leo Dijksman's WBEC (Coulom, 2014).

# Chapter 3

# Machine Learning Models

This chapter introduces the four machine learning models we will use throughout experiments. It also discusses the interpretation of the models to aid in our later understanding of their behaviour, and the issues related to training them.

It is of crucial importance to select the right model to learn the heuristic evaluation function. As we observed earlier, an optimal heuristic evaluation function computes the same ranking as the ranking given by listing the game positions in order of increasing winning probability for the player. Therefore we will attempt to approximate the heuristic evaluation function by a probabilistic model, which estimates the probability of winning, drawing and losing from a given position. We may then assign the value of a board position to the expected pay-off based on this distribution, where we take 1, 0, and $-1$ to be the pay-off of winning, drawing and losing the game.

We can motivate this as follows. Let us assume that our program plays White, and that the opponent is a *perfect player*, i.e. a player who finds the best move by exhaustively going through all possible board positions with the minimax algorithm. Let $\mathbf{x}$ be a board position and let $P(\text{White wins}|\mathbf{x})$, $P(\text{Draw}|\mathbf{x})$ and $P(\text{Black wins}|\mathbf{x})$ be the probabilities for the player to respectively win, draw or lose the game. The board position $\mathbf{x}$ could consist of any set of features extracted from the board position, but for now let us imagine that this a binary vector representing with an entry for each square and piece, which takes value one if the piece is on

that square and zero otherwise. Now, the expected pay-off from any board position $\mathbf{x}$ equals:

$$\mathrm{E}\left[\text{Pay-off}|\mathbf{x}\right] = P(\text{White wins}|\mathbf{x}) - P(\text{Black wins}|\mathbf{x}) \tag{3.1}$$

In fact, this is the true value of each node used in the minimax algorithm in section 2.3. If we knew a function to compute this value, we would be able to play perfect, i.e. select the same moves as if we had exhaustively gone through all possible board positions with the minimax algorithm. Thus, we aim to learn the probability distribution over all three outcomes of the game.

This assumes that the opponent is a perfect player, which is unlikely to exist for a game as complex as chess. Moreover, the best we can hope for is an approximation of the probability distribution. Nevertheless, this has proven to be a fruitful assumption in previous research, as we discussed in chapter 2, and it also allows us to use the same program against any opponent. A similar approach was also taken in (Ban, 2012).

One may argue that the expected pay-off could be modelled as a regression problem instead of a classification problem. A regression model would take into account the ordinal structure of the outcomes to a larger extent, e.g. if White wins is the most probable outcome then draw must also be more probable than Black wins. This, of course, would limit us to only measure the performance w.r.t. expected pay-off, which is arguably a less orthodox method than the actual game outcome since the characteristics of drawn games may be very different from other games. In addition to this, in many chess endgames the fifty-move rule allows the weaker player to keep repeating his positions and either end the game in a draw or force the stronger player to sacrifice material. If sacrificing material allows the stronger play to win a sacrifice should be preferred, but if it allows the weaker player to gain the advantage a draw should be preferred. If the probability of a draw is known explicitly, it can be used to distinguish between these two choices in the truncated search tree, and therefore effectively handle the fifty-move rule in endgames [1] (Veness, 2014b).

Apart from the game outcome, the next question is whether we need to define other random

[1]Note that while the model could take as input the number of moves related to the fifty-move rule, i.e. the number of moves since the last piece capture or pawn movement, this is not strictly necessary. If the opponent forces the player to sacrifice material, a heuristic evaluation function applied to the board position after sacrificing the material, even without knowledge of the fifty-move rule, should be able to evaluate whether sacrifice leads to a win or a draw.

(stochastic) variables. Since chess is a deterministic perfect information game, we argue that it is not necessary to define any other random variables. The only stochastic element in the game is the opponent and this is already modelled through the game outcome itself. We could argue for additional random variables to model playing style etc., but this form of explicit opponent modelling is not necessary in chess to achieve master-level play (Billings et al., 1998). For the same reason we could introduce random variables for the pieces on each board square, if we view the moves of each player as coming from a random distribution, but for simplicity we choose not to do. This has not been necessary in master-level chess engines such as Deep Blue and Stockfish (Romstad et al., 2014). This is perhaps in contrast to many other machine learning domains, such as computer vision and speech, where observations are affected by many sources of noise, and where taking this into account can improve the models substantially. For this reason, we will not consider models such as restricted Boltzmann machines, deep belief networks and auto encoders (Bengio, 2009, pp. 40–45), although we note that they would be interesting directions for further research.

## 3.1 Logistic Regression

### 3.1.1 Definition

We choose to start with one of the simplest classification models. The multinomial logistic regression model (also known as the multinomial logit model or the linear softmax model) is a well-established statistical model, which estimates the probability of a discrete set of classes given a set of observations (Bishop, 2006, p. 215; Hastie et al., 2009, p. 119). For classes $1, \ldots, K$ the model is defined by

$$P(y = k|\mathbf{x}) = \frac{\exp\left(\beta_{k_0} + \boldsymbol{\beta}_k^T \mathbf{x}\right)}{1 + \sum_{k'=1}^{K-1} \exp\left(\beta_{k'_0} + \boldsymbol{\beta}_{k'}^T \mathbf{x}\right)}, \tag{3.2}$$

where the observations are $y \in \{1, \ldots, K\}, \mathbf{x} \in \mathbb{R}^M$ and the parameters are $\beta_{k_0} \in \mathbb{R}$, $\boldsymbol{\beta}_k \in \mathbb{R}^M$ for $k = 1, \ldots, K$. The parameters for class $K$ are redundant due to the normalization, so we assume $\beta_{K_0} = 0$ and $\boldsymbol{\beta}_K = \mathbf{0}$. We denote the product in the exponent of numerator $\beta_{k_0} + \boldsymbol{\beta}_k^T \mathbf{x}$ as the score for w.r.t. class $k$. To predict an outcome, we simply take the class with the highest

probability:

$$\hat{k} = \arg\max_k P(y = k|\mathbf{x}) = \arg\max_k \beta_{k_0} + \boldsymbol{\beta}_k^T \mathbf{x} \tag{3.3}$$

If several $k$ yield the same value we assign the predicted outcome (arbitrarily) to be the class with the highest index.

To handle discrete observations, indicator functions may be used [2]. For example, if $x_1 \in \{1, \ldots, R\}$ we may apply the logistic regression to the transformed $x$:

$$g(x) = \begin{pmatrix} 1_{x=1} \\ \vdots \\ 1_{x=R} \\ x_2 \\ \vdots \\ x_M \end{pmatrix}, \tag{3.4}$$

$$P(y = k|\mathbf{x}) = \frac{\exp\left(\beta_{k_0} + \boldsymbol{\beta}_k^T g(x)\right)}{1 + \sum_{k'=1}^{K-1} \exp\left(\beta_{k'_0} + \boldsymbol{\beta}_{k'}^T g(x)\right)}. \tag{3.5}$$

Although to simplify our experiments, we shall not use this construction.

Henceforth we will call the multinomial logistic regression for logistic regression or, even shorter, LogReg [3].

### 3.1.2 Interpretation

In the two-class scenario, i.e. $K = 2$, with continuous observations the model may be interpreted as classifying points according to which side they fall on an affine hyperplane. This is clear

[2] An indicator function is a function which returns either one or zero, for example $1_{x>0}(x)$. In the literature these are often called *dummy* functions or *dirac delta* functions.

[3] Logistic regression is usually reserved for the two-class scenario, i.e. $K = 2$, but we will take the number of classes to be evident from the context.

from eq. (3.3), which yields:

$$\hat{k} = \underset{k \in \{1,2\}}{\arg\max}\, \beta_{k_0} + \boldsymbol{\beta}_k^T \mathbf{x} = \begin{cases} 1 & \beta_{1_0} + \boldsymbol{\beta}_1^T \mathbf{x} > 0 \\ 2 & \text{otherwise} \end{cases} \tag{3.6}$$

This is illustrated in figure 3.1.

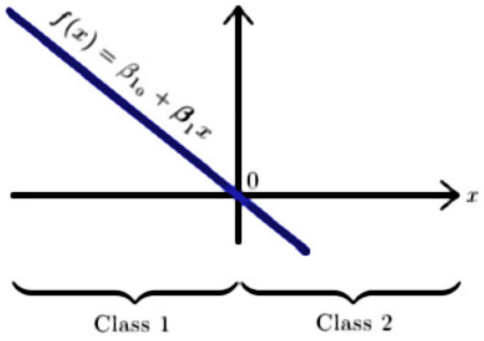



Figure 3.1: Two-class LogReg on a one-dimensional continuous observation $x$, where $f(x) = \beta_{1_0} + \beta_1 x$ corresponds to the score w.r.t. the first class. Note, by assumption the score function for class two is always zero.

A similar interpretation may be applied to the three-class scenario, i.e. $K = 3$, where the classification boundary is now determined by two affine hyperplanes:

$$\hat{k} = \underset{k \in \{1,2,3\}}{\arg\max}\, \beta_{k_0} + \boldsymbol{\beta}_k^T \mathbf{x} = \begin{cases} 1 & \beta_{1_0} + \boldsymbol{\beta}_1^T \mathbf{x} > \max(\beta_{2_0} + \boldsymbol{\beta}_2^T \mathbf{x}, 0) \\ 2 & \beta_{2_0} + \boldsymbol{\beta}_2^T \mathbf{x} > \max(\beta_{1_0} + \boldsymbol{\beta}_1^T \mathbf{x}, 0) \\ 3 & \text{otherwise} \end{cases} \tag{3.7}$$

In particular, for a single continuous observation, this implies that the LogReg model is able to use two thresholds $t_1$ and $t_2$ to classify observations into three classes as shown in figure 3.2:

$$\hat{k} = \underset{k \in \{1,2\}}{\arg\max}\, \beta_{k_0} + \boldsymbol{\beta}_k^T \mathbf{x} = \begin{cases} 1 & x < t_1 \\ 2 & t_1 \leq x \leq t_2 \\ 3 & t_2 < x \end{cases} \tag{3.8}$$

This is important in some of our experiments, where the model will be taking a single scalar variable as input, such as, the QSearch Feature in chapter 6. Here, the LogReg model will be able to use two thresholds to differentiate between three classes, similar to having two thresholds applied to a scalar function of the input, e.g. two thresholds applied to a linear regression on the input. The same interpretation can be extended to a higher input dimensionality, where each class outcome is assigned to a linear affine half-space of the input.

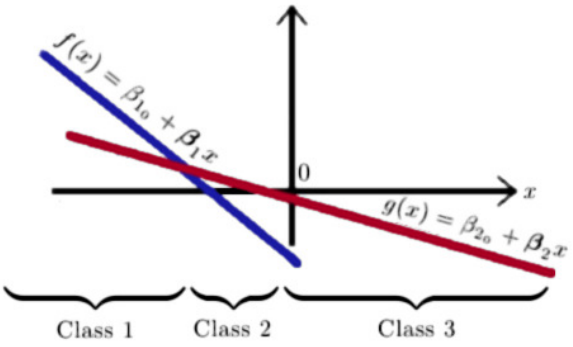



Figure 3.2: Three-class LogReg on a one-dimensional continuous observation $x$, where $f(x) = \beta_{1_0} + \boldsymbol{\beta}_1 x$ and $g(x) = \beta_{2_0} + \boldsymbol{\beta}_2 x$ corresponds to the score w.r.t. the first and second class respectively. Note, by assumption the score function for class three is always zero.

### 3.1.3 Training

The parameters of LogReg are usually found using maximum likelihood (Hastie et al., 2009, p. 119). Let our observations and corresponding classes be $\{\mathbf{x}_n, y_n\}_{n=1}^N$ and assume that each training instance is sampled i.i.d. (independent identically distributed). Assume further that we append a constant to each data point $\mathbf{x}_n$ such that we may drop the intercepts $b_{k_0}$. Let $\boldsymbol{\theta} = \{\boldsymbol{\beta}_k\}_k$. Then the log-likelihood is given by:

$$l(\boldsymbol{\theta}) = \log\left(\prod_{n=1}^{N} P(y_n|\mathbf{x}_n)\right) = \sum_{n=1}^{N} \log\left(\frac{\exp\left(\boldsymbol{\beta}_{y_n}^T \mathbf{x}_n\right)}{1 + \sum_{k'=1}^{K-1} \exp\left(\boldsymbol{\beta}_{k'}^T \mathbf{x}_n\right)}\right) \tag{3.9}$$

$$= \sum_{k'=1}^{K-1} \boldsymbol{\beta}_{y_n}^T \mathbf{x}_n - \log\left(1 + \sum_{k'=1}^{K-1} \exp\left(\boldsymbol{\beta}_{k'}^T \mathbf{x}_n\right)\right) \tag{3.10}$$

The interpretation of maximum likelihood for our purposes is straightforward: it maximizes the probability of drawing the same game outcomes, given the board positions, as observed in the dataset. In machine learning, where it is widely used, this is also known as the (negative) *cross-entropy error* (Hastie et al., 2009, p. 32). Generally speaking, it is related to minimizing the entropy (number of bits) required to transfer the class outcome between a *sending party* and *receiving party* given that the receiving party knows the model.

It is straightforward to compute the first and second derivatives of this expression and use these to find the parameters. For small to moderate sized datasets the Newton–Raphson algorithm is often used. For larger models this becomes computationally intractable, since the method requires second-order derivatives, and instead a simple stochastic gradient descent is often applied. In fact, on some datasets a stochastic gradient descent with a fixed step-size appears to perform better than with a decreasing step-size [4]. One can easily show that the log-likelihood is concave and therefore any local maximum must be a global maximum (Hastie et al., 2009, p. 121).

### 3.1.4 Discussion

It has been argued that the LogReg model introduced above is both overconfident and prone to overfitting in high dimensions (Bishop, 2006, p. 216; Prince, 2012, p. 137). In particular, if the dataset is linearly separable the solution becomes degenerate because the log-likelihood in eq. (3.10) can be increased to an arbitrarily number by setting the parameters to a multiplum of a vector perpendicular to the hyperplane which separates the classes.

One solution for handling this is called L1-regularization, which changes the objective function optimized (Hastie et al., 2009, pp. 125–127). The log-likelihood is subtracted by the sum of absolute parameter values, excluding entries corresponding to the intercept terms, multiplied by a positive constant:

$$s(\boldsymbol{\theta}) = l(\boldsymbol{\theta}) - \lambda \sum_{i;\ \text{where}\ \ \theta_i\ \text{is not an intercept}} |\theta_i| \tag{3.11}$$

where $\lambda > 0$. This effectively corresponds to having a Laplace distribution prior over the model

[4] See `http://deeplearning.net/tutorial/logreg.html`.

parameters. In other words, a priori we expect most of our parameters to be close to zero and only a few to be significantly different from zero. This approach is often taken when the inputs have a high dimensionality (Herbster and Pontil, 2013). It can easily be extended to all the models we will work with.

Bayesian logistic regression is also often advocated, where an isotropic normal distribution prior is assumed on the parameters as:

$$P(\boldsymbol{\theta}) = \prod_{k=1}^{K-1} \prod_{m=1}^{M} \text{Norm}_{\beta_{km}}(0, \sigma^2), \tag{3.12}$$

with the solution found by using the MAP (maximum a posteriori) estimate, which simply maximizes the posterior probability:

$$\log P\left(\boldsymbol{\theta} | \{\mathbf{x}_n, y_n\}_{n=1}^{N}\right) = l(\boldsymbol{\theta}) + \log P(\boldsymbol{\theta}) \tag{3.13}$$

The parameter $\sigma^2$ can be found with cross-validation.

This is an interesting solution and indeed it is not clear whether or not such a model could improve the performance of classifying chess board positions. One one hand, we shall work with both a relatively large dataset and an unlimited dataset respectively in our supervised learning and self-play experiments. This implies that the model should not overfit. Furthermore, since we know our dataset is not linearly separable, the model cannot become degenerate [5]. On the other hand, certain features in chess may occur extremely rarely. Suppose that once in a thousand board positions, White has moved both rooks at Black's back rank. Everything else being equal, this is disastrous position for Black, which should be weighted different from either rook being at Black's back rank. The MAP approach should be able to better accommodate this rare event, by attributing a higher class uncertainty for board positions involving White's rooks. Nevertheless, since we shall be using the MAP approach in our other models, we shall also refrain from adopting it now.

[5] Our dataset is not linearly separable because we include the starting positions. Since we have games where White wins, Black wins and draws, and since each class contains at least one training instance with the starting board position, our dataset cannot be linearly separable.

## 3.2 Multilayer Perceptron Network

### 3.2.1 Definition

The multilayer perceptron network (also known as multilayer perceptron, backpropagation neural network or simply neural network), abbreviated MLP, is a classical model in machine learning and artificial intelligence research (Bishop, 2006, p. 227). In the case of classification, the model can be seen as a simple non-linear extension of the LogReg model introduced in the previous section, where the inputs are first mapped through a non-linear function before the LogReg model takes them as input.

As before, suppose we wish to classify an observation $\mathbf{x} \in \mathbb{R}^M$ as one of $K$ classes. Let $z_k$ be the score value corresponding to class $k$ for a given point $\mathbf{x}$. We define the MLP with one hidden layer containing $L$ hidden units as

$$z_k = \sum_{l=1}^{L} \alpha_{lk} g \left( \sum_{m=1}^{M} \beta_{lm} x_m + \beta_{l_0} \right) + \alpha_{k_0}, \tag{3.14}$$

for parameters $\alpha_{k_0} \in \mathbb{R}, \boldsymbol{\alpha}_k \in \mathbb{R}^L$ for $k = 1, \ldots, K-1$, $\beta_{l_0} \in \mathbb{R}, \boldsymbol{\beta}_l \in \mathbb{R}^M$ for $l = 1, \ldots, L$ and $g$ is a non-linear real-valued function called the *activation function*. To convert the function into a probabilistic model we let

$$P(y = k|\mathbf{x}) = \frac{\exp(z_k)}{1 + \sum_{k'} \exp(z_{k'})}, \tag{3.15}$$

which effectively corresponds to a LogReg model on the $L$ transformed inputs $\left[ g \left( \sum_{m=1}^{M} \beta_{1m} x_m + \beta_{1_0} \right), \ldots \right]$ [6]. This is also known as the softmax function. The transformed inputs are called *hidden units* and are said to belong to the *hidden layer*. With the same terminology the inputs are referred to as *input units*.

The model can easily be extended with additional hidden layers by adding non-linear transformations on the inputs. The MLP with two hidden layers, containing respectively $L$ and $R$

[6] The multilayer perceptron does not need to be defined in terms of the log-likelihood as we have done, but can be defined for more general error functions to minimize. As this is irrelevant in our context, we shall refrain from the more general (and notationally more cumbersome) definition. See (Bishop, 2006, p. 227) for a more general definition, which includes regression models.

hidden units each, is given by

$$z_k = \sum_{l=1}^{L} \alpha_{lk} g \left( \sum_{r=1}^{R} \beta_{lr} h \left( \sum_{m=1}^{M} \gamma_m x_{rm} + \gamma_{m_0} \right) + \beta_{l_0} \right) + \alpha_{k_0}, \tag{3.16}$$

for parameters $\alpha_{k_0} \in \mathbb{R}$, $\boldsymbol{\alpha}_k \in \mathbb{R}^L$ for $k = 1, \ldots, K-1$, $\beta_{l_0} \in \mathbb{R}, \boldsymbol{\beta}_l \in \mathbb{R}^R$ for $l = 1, \ldots, L$ and $\gamma_{r_0} \in \mathbb{R}, \boldsymbol{\gamma}_r \in \mathbb{R}^M$ for $r = 1, \ldots, R$, where $g$ and $h$ are non-linear real-valued functions. Stacking additional hidden layers lead to a so-called *deep representation*, which is the core structure of machine learning field named deep learning (Bengio, 2009). The MLP is illustrated in figure 3.3.

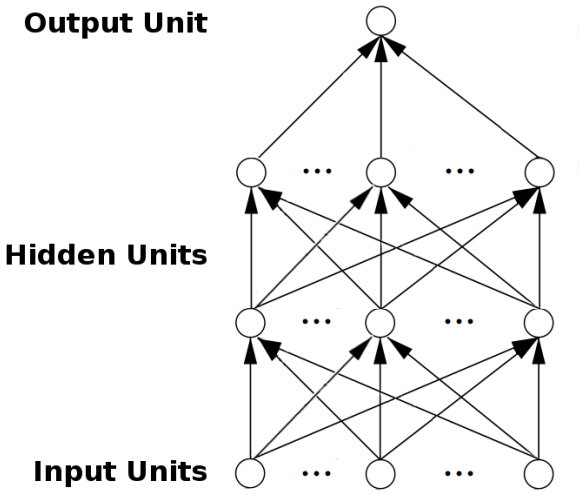



Figure 3.3: Two-layered multilayer perceptron network (MLP) for the two-class case. The second class is assumed to have score function equal to zero, as in the LogReg model. The top node (output units) corresponds to the probability of class one, the vertically centered nodes (hidden units) correspond to transformed values of the inputs and the bottom nodes (input units) correspond to input observations. As opposed to the notation for graphical models, the edges here denote parameters instead of dependencies.

As the activation function the hyperbolic tangent function $f(x) = tanh(x)$ or standard logistic function $f(x) = 1/(1 + e^{-x})$ are commonly, where the former is preferred since it is

anti-symmetric around $x = 0$, however a radial basis function such as

$$f(x) = \sum_i \psi_i \exp\left(-\frac{1}{2\sigma^2}||x - v_i||^2\right), \tag{3.17}$$

where centroids are $v_i$ and the weights are $\psi_i$ for $i = 1, \ldots, Q$ can also be used (LeCun et al., 2012, p. 15). Many other functions are also possible.

Often various combinations of these activation functions are used across and within layers. However, empirical evidence has shown that the hyperbolic tangent work better in the training phase due to the gradient calculations required (LeCun et al., 2012, p. 10; Bengio, 2012, p. 14). For reasons to be explained later and to keep things simple, we will stick with the hyperbolic tangent function.

### 3.2.2 Interpretation

One interpretation of the model is that hidden units at various hidden layers correspond to features the MLP has learned. In the simple LogReg model, each input is assigned a scalar value which is then used to determine the probability of each class given the input. This leads to an affine linear classifier in the two-class case, which completely ignores interactions between input variables. This makes it impossible to solve, for example, the XOR problem shown in figure 3.4.

The MLP is easily able to solve the XOR problem. To see this, it is best to first cast the problem in terms of logical operators. We observe that we can write a perfect classifier for the XOR problem as a function of AND, OR and NOT operators respectively, where we take 1 to be the value ON and $-1$ to be OFF:

$$y = (x_1 \wedge x_2) \vee (\neg x_1 \wedge \neg x_2) \tag{3.18}$$

We can translate this into the hyperbolic tangent functions with:

$$\hat{y} = \tanh(x_1 + x_2 - 0.5) + \tanh(-x_1 - x_2 - 0.5) + 0.5 = \begin{cases} 0.4 & \text{if } x_1 = x_2 \\ -0.4 & \text{if } x_1 \neq x_2 \end{cases} \tag{3.19}$$

| $x_1$ | $x_2$ | y |
|---|---|---|
| -1 | -1 | 2 |
| -1 | 1 | 1 |
| 1 | 1 | 2 |
| 1 | -1 | 1 |

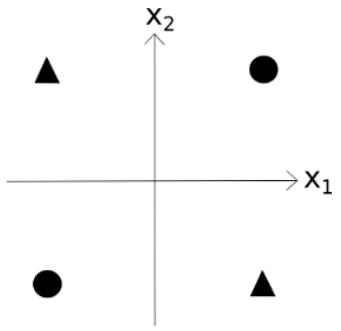



Figure 3.4: The XOR problem cannot be solved by the LogReg model, because there is no affine hyperplane which separates one class from the other (e.g. a straight line between the crosses and circles). Instead the features must first be transformed through a non-linear function before being taken as input by the LogReg.

The first hyperbolic tangent term acts as an AND operator on $x_1$ and $x_2$, the second as an AND operator on $\neg x_1$ and $\neg x_2$, and the sum between the two as an OR operator. With this input the logistic regression can easily separate the two classes using a threshold at zero. Although, note that actual MLP model works in real-valued numbers not in binary logic.

### 3.2.3 Training

#### Training with stochastic gradient descent

The typical approach to learning the model parameters of the MLP is to apply some form of iterative optimization procedure, such as gradient descent, conjugate gradients or quasi-Newton methods on the log-likelihood [7] (Bishop, 2006, p. 240). Methods which require second-order derivatives, i.e. the Hessian matrix, are found to converge much faster w.r.t. number of iterations, but are computationally more expensive per iteration. For large datasets stochastic gradient descent over mini-batches, i.e. small sets of training instances, is often preferred (Bengio, 2012, p. 14; LeCun et al., 2012, p. 5; Bottou, 2012). We defer the in-depth treatment of stochastic gradient descent to section 3.4. For now the reader may assume that stochastic gradient descent is a procedure which iteratively estimates the derivative of each parameter in

[7] Although we call our procedure for gradient descent, it is actually finding a maximum and hence should in principle be called gradient ascent. The term gradient ascent is not as widely used in the literature, and hence we stick with gradient descent.

the model w.r.t. the log-likelihood for a single training instance, adjusts the parameter in the direction of the gradient and then repeats the procedure.

In principle, we could apply gradient descent on any other objective function to train the model. For example, the mean squared error function, which averages the squared norm between the models predicted output and the $K$-vector with a one in the entry corresponding to the class and zeros everywhere else. Nevertheless, we have already motivated the cross-entropy objective function from a probabilistic point of view and, indeed, for classification problems it is often strongly advocated, because it tends to give a better performance than other objective functions (Bengio, 2009, p. 16; Golik et al., 2013; Zhou and Austin, 1998).

### Preprocessing

(LeCun et al., 2012, p. 9) propose that the inputs should be normalized such that their average is close to zero, their covariances are close to one and that they are decorrelated. Decorrelation is important because it will reduce the complexity of the features the model needs to learn. They, and others, suggest applying principal component analysis (PCA) to transform the inputs to become orthogonal (Bengio, 2012, p. 15). See (Bishop, 2006, p. 561) for an introduction to PCA. Nevertheless, it is debatable whether or not PCA should be applied to discrete variables. On one hand, the solution PCA gives for the first $K$ components can be viewed as finding a linear projection to a subspace which preserves the most information (entropy) about the data under the assumption that it follows a Gaussian distribution (Sahani, 2013, Lecture 2). This effectively compresses the representation, which might improve our models by letting them work with effectively fewer inputs. On the other hand, PCA finds an orthogonal projection (Sahani, 2013, Lecture 2). Since our representations are going to be highly redundant, see section 6.2, this may cause PCA to confound many of the input variables. For example, in one representation there is a bit for each possible square a certain piece type can be on. Because there cannot be more than one piece on each square, knowing that the bit is ON implies that all other bits related to the same square (but for other pieces) will be OFF. It is therefore likely that PCA will confound the pieces on the same square by assuming that it is always the most frequent piece which is situated at that square, i.e. the pawn, in the eigenvectors with highest absolute eigenvalues. Hence our models may be unable to distinguish between different pieces on the same square, e.g. a queen might be mistaken for a pawn. To distinguish between pieces,

e.g. to create a feature related only to pawn pieces, the model would have to partially invert the PCA transformation and then apply its non-linear function on to these.

### Shuffling training instances

Shuffling the examples is an important part of the training procedure (LeCun et al., 2012, p. 7). For MLPs and convolutional neural networks, it is also important that all mini-batches retain a balanced number of training instances from each class regardless of the actual class distribution (Turaga, 2014). One way to to achieve this balance is to sample a class with probability one divided by the number of classes, and then sample at uniformly random a training instance with that class from the training set. To retain the true distribution over classes, after training for a while, the classes should then be sampled with probability proportional to their frequency in the dataset. An argument supporting this procedure, given by (LeCun et al., 2012, p. 7), is that the model will learn faster from unexpected training instances. If we assume that different classes are represented by distinct features, then by continuously feeding training instances with different classes the model will effectively be learning from a variety of distinct features.

LeCun et al. go further and propose that, after sampling a class (index), a training instance with the chosen class (index) should be included in the mini-batch with probability proportional to the training error obtained in the mini-batch it was last included in (LeCun et al., 2012, p. 8). This should encourage the model to constantly learn new features. However, as LeCun et al. also note, this may be disastrous in the case of outliers. These would be included in many mini-batches, but would throw off the learning procedure rather than improve it. Since it is possible to make serious blunders in chess (for both human and computer players), many training instances might contain board positions where one player clearly has the advantage but later loses the game (due to a blunder by a human player, or say by incorrect pruning by a chess engine). These should all be considered outliers. Furthermore, many training instances are going to be volatile positions (such as when trading queens) and repeatedly sampling these in the mini-batches may seriously harm the training, at least in the early stages.

To reduce computational time, we choose not to employ a sampling scheme. In both our supervised learning and self-play experiments, the training dataset is almost evenly divided between White wins and Black wins instances, as we will discuss later. To maintain distinct

training instances in each mini-batch we choose to simply shuffle the data randomly at the beginning at the first epoch [8]. In most cases, this corresponds to sampling White wins and Black wins with equal probability. For the reasons discussed above, we choose not to weight the training instances by their training errors.

### Initialization

Initializing the parameters appropriately is necessary to break symmetry and hence to learn a distinct set of features. A method that seems to work well across many problems, where the data has been scaled as suggested by LeCun et al., is to sample each parameter from a uniform distribution Uniform$(-r, r)$ with $r = 6/(\text{fan-in} + \text{fan-out})$, where fan-in is the number of inputs to the unit (number of units in the previous later) and fan-out is the number of outputs (number of units in the next layer) (Bengio, 2012, pp. 14–15). Biases (intercepts) are simply initialized to zero, as well as all parameters in the top layer of the model (LogReg model parameters). We shall apply this simple initialization procedure.

Any variable taking value one (respectively, zero) will have a positive value (respectively, negative value) when the mean has been subtracted. Since the initialization is symmetric around zero, we may interpret it as initializing the model with NOT operators applied to half of the binary variables.

These are then weighted linearly to form various logical operators (though not limited to AND and OR operators). In principle, one could replace the uniform random distribution with a discrete random distribution to force sparser features at initialization, i.e. functions which only operate on a small set of variables. One could also sample the biases from a non-zero mean uniform random distribution to encourage more AND operators (a uniform distribution with negative mean) or OR operators (a uniform distribution with positive mean). However, it is not clear how these features will change when trained with backpropagation. In any case, it is beyond the scope of this project to experiment with additional initialization schemes.

[8] In some experiments, we shuffled the training instances at every epoch, but we did not see any difference in results when we shuffled only in the first epoch and hence we adopted this faster approach.

### Backpropagation

The backpropagation procedure (also known as error backpropagation or simply backprop) is commonly used to find derivatives for MLP and similar models. In brief, the backpropagation procedure uses the chain rule for partial derivatives to recursively calculate an intermediate value at each unit in the network. These intermediate values can then be used to find the derivative w.r.t. parameters associated with each particular unit [9]. The procedure can easily be extended to calculate second-order derivatives. See (Bishop, 2006, pp. 241–245) for a detailed description of the procedure.

To avoid the trouble of manually taking derivatives of highly composed functions while keeping computational complexity low and calculations numerically stable, we have chosen to use the library *Theano* in our project (Bergstra et al., 2010). Theano is a compiler for mathematical expressions in Python, which is able to handle many computations (such as calculating derivatives) efficiently by representing them in an abstract symbolic language, which at runtime is converted into a symbolic graph and compiled into C++ code with numerous optimizations. See appendix C for a more detailed description.

Nevertheless, in general, training MLPs through backpropagation is considered to be a notoriously difficult task (Bengio, 2009, pp. 31–35). This has been shown by the mixed and negative results of many published and unpublished experiments. The training procedure consists of numerous hyperparameters, which need to be tweaked and tuned appropriately over many experiments to give reasonable results (Bengio, 2012). Furthermore, if the training procedure fails for any reason, it is difficult to pinpoint the problem because of the inter-dependencies between all the parameters across the model. For these reasons, some people have referred to the training procedure as *black magic*. In addition to this, one particular problem with stochastic gradient descent is that the training procedure often gets stuck in *apparent local minima* or *plateaus*, in particular, when starting from a random parameter initialization. As the models become deeper, i.e. the number of layers is increased, the problem is amplified by what researchers have termed the *vanishing gradient*. Many hidden units take responsibility for the same error between the observed (true) output and the model (predicted) output, which

[9] The intermediate value is calculated recursively from the top (output) nodes in a layer-by-layer procedure to the bottom layer. The intermediate value can be seen as a message sent by nodes at higher levels, which is why backpropagation is also sometimes referred to as a *message-passing scheme*.

then causes the gradients w.r.t. the parameters of each hidden unto to become very small. There are ways to get around this issue by initializing the networks properly and applying other optimization procedures. See for example (Bengio, 2009, 2012).

## 3.3 Convolutional Neural Network

### 3.3.1 Definition

The convolutional neural network, abbreviated ConvNet, was first suggested by LeCun et. al to recognize digits from images (LeCun et al., 1989) [10]. Inspired by the human vision system, they adapted an MLP with a 2D representation for units in each layer and constrained the incoming edges (parameters) for each unit to be non-zero only w.r.t. units located in a small grid (called receptive field) in the previous layer. For example, with $3 \times 3$ receptive fields, a hidden unit located at $(2, 2)$ in the first hidden layer might weight the $3 \times 3 = 9$ units in the previous layer at $(1, 1), (1, 2), \ldots, (3, 2), (3, 3)$ linearly and then transform the weighted value by the activation function. These layers are termed convolutional layers because the linear weighting of a receptive field can be represented by a convolutional operation.

They further added several sets of hidden units at each layer (called feature maps) and constrained all hidden units in the same set (same feature map) to have the exact same parameters w.r.t. each incoming edge. For example, with $3 \times 3$ receptive fields, two hidden units in the first hidden layer, which are in the same feature map located at $(2, 2)$ and $(3, 3)$ respectively will apply the same weights to the units in the previous layer across all feature maps at $(1, 1), (1, 2), \ldots, (3, 2), (3, 3)$ and $(2, 2), (2, 3), \ldots, (4, 3), (4, 4)$ respectively. The convolutional layers can easily be combined with the layers in an MLP (now called fully-connected layers), by concatenating all rows in the feature maps to form a vector, which is taken as input by the MLP. The computational flow is visualized in figure 3.5.

[10] As Dayan has pointed out, a very similar model was proposed by Fukushima earlier (Fukushima, 1988).

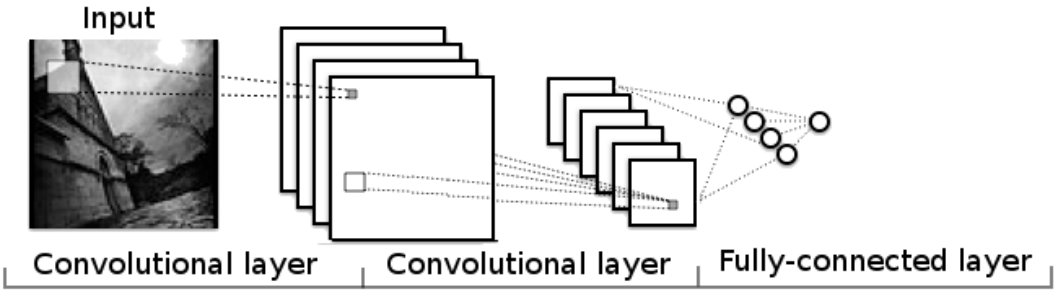



Figure 3.5: Convolutional neural network (ConvNet) for object recognition in images. Each unit linearly weights the values of a small receptive field in the input image and applies the activation function. The process is repeated at the next hidden layer, while the third and LogReg hidden layers correspond to an MLP taking as inputs the second hidden layer (the matrix flattened to a vector). The same structure of two convolutional layers followed by a fully-connected layer was used in (LeCun et al., 1989). Image adapted from `http://deeplearning.net/tutorial/lenet.html`.

Formally, the parameters of the model can be described as a set of matrices, one for each layer, where the columns corresponds to the feature map index and the rows to the parameters w.r.t. the incoming edges. Let $z_{ij}^k$ be the unit value in feature map $k$ at location $(i, j)$ with receptive field size $R \times R$ for $R \in 1, 3, \ldots$. Let $x_{ij}^k$ be the unit value in the layer below in feature map $k$ at location $(i, j)$. Then the value of each hidden unit is given by the recursive operation:

$$z_{ij}^k = f\left(\sum_{k'} \sum_{i'=1,\ldots,R} \sum_{j'=1,\ldots,R} \beta_{i'j'}^{k'} x_{(i'-(R-1)/2)(j'-(R-1)/2)}^{k'} + \beta_0^k\right), \tag{3.20}$$

for parameters $\boldsymbol{\alpha}^k \in \mathbb{R}^{R \times R}, \alpha_0^k \in \mathbb{R}$ where $f$ is the activation function. It is important to observe that the receptive fields do not have to be square-shaped, but could take any other shape such as, rectangles or a discretized circles. They may also differ between layers and feature maps.

In addition to this, researchers have proposed *max-pooling layers*. These are layers where a maximum operator is applied to the values of the hidden units in the layer below instead of an activation function over linearly weighted inputs. Formally, with the previous definitions, this is recursively computed by:

$$z_{ij}^k = \max_{i'=1,\ldots,R, j'=1,\ldots,R} x_{(i'-(R-1)/2)(j'-(R-1)/2)}^{k'}. \tag{3.21}$$

The max-pooling layers makes the inputs at lower levels indistinguishable, and therefore restrict the model to only capturing features which are invariant to offsets in location.

### 3.3.2 Training

As in the MLP model, training is performed using stochastic gradient descent. However, the modification of equating the parameters in each feature map, known as *weight sharing* in the literature, now reduces the number of parameters by a magnitude. This means that there is more data per parameter in the ConvNet compared to the fully-connected MLP. In addition to this, the topological structure induced by the receptive fields ensures that the gradients do not diffuse to the same extent as the MLP, which makes the networks very suitable for stochastic gradient descent (Bengio, 2009, pp. 43–45).

### 3.3.3 Interpretation and discussion

We may apply the same interpretation as we did for the MLP. Each feature map in each layer corresponds to features the ConvNet has learned. These features are restricted to depend on local information only, but required to generalize over the entire input space. This enables the model to better take into account invariant aspects of the inputs, such as translations and symmetries [11]. For this reason, ConvNets networks have been found to work well in Go (Schraudolph et al., 2001). This is believed by some to be important in chess based on psychological studies of human players (Mańdziuk, 2012). Mandziuk argues in favour of models which take into account the geometrical properties of the game when he states: *"Humans define relations among pieces and empty squares based on spacial location of these objects and their geometrical interrelations. It seems, therefore, reasonable to expect that human-like game playing system would possess the ability of representing pertinent game concepts in a similar way."* (Mańdziuk, 2012). Yet the design also prohibits the model from learning features which require integrating knowledge across inputs beyond the receptive fields, a property which might be crucial in chess, where pieces can sometimes traverse the entire board in a single move, e.g.

[11] Although strictly speaking our structure only imposes translational invariance, as we will discuss later in our self-play experiments, we also impose invariance w.r.t. color symmetry by also using the flipped board and piece colors as training instances.

the rock in an end game with only a few pieces left. In principle, the model can still capture this by, for example, propagating the inputs directly forward using an identity function in convolutional layers until a fully-connected layer. That, of course, would require many more training examples than the same model without convolutional layers.

We choose not to use the max-pooling layers as we do not believe that the extreme locational invariance they impose is justified in chess. For example, changing the location of a single piece in chess may alter the game significantly, e.g. imagine placing White's queen in reach of a Black pawn when it's Black to move.

## 3.4 Stochastic Gradient Descent

For large datasets stochastic gradient descent over mini-batches is the most common optimization technique used for MLPs and ConvNets (Bengio, 2012, p. 14; LeCun et al., 2012, p. 5; Bottou, 2012). This is emphasized by (Bottou, 2012, p. 4) with *"Use stochastic gradient descent when training time is the bottleneck."*. In particular, the usage of mini-batches has been advocated when the data is highly redundant, as we might hope is the case in our experiments (LeCun et al., 2012; Tieleman and Hinton, 2012, Lecture 6). We will now refer to stochastic gradient descent over mini-batches as simply stochastic gradient descent.

The stochastic gradient descent procedure iteratively estimates the derivative of each parameter in the model w.r.t. the log-likelihood for a set of training instances, referred to as a mini-batch, adjusts each parameter in the direction of the gradient according to a fixed step-size, known as the learning rate, and repeats the procedure over again on a new disjoint set of training instances (a new mini-batch) [12]. The procedure is run over the entire training set multiple times, where each run is called an epoch.

To ensure convergence the learning rate is typically decreased during training. Let $\gamma_t$ be the learning rate at epoch $t$. Based on theoretical analysis of convergence and empirical evidence this is often taken to be:

$$\gamma_t = \frac{\gamma_0}{1 + \gamma_0 \lambda t}, \tag{3.22}$$

[12] In principle the gradient w.r.t. any objective (loss) function could be used, but to keep things simple we will stick to our probabilistic framework of maximizing the log-likelihood.

where $\gamma_0$ is the initial learning rate, i.e. the learning rate at epoch zero and $\lambda$ is a decaying constant. See (Bottou, 2012, p. 10) for a brief review. Many other schemes exist, see for example (Bengio, 2012, p. 9), but we will stick to this for simplicity.

Instead of moving the parameters in the direction of the gradient (which could be very noisy due to a small batch size) a momentum factor is often used (Bengio, 2012, p. 10; LeCun et al., 2012, p. 14). That is, a moving average of the gradient is used to adjust each parameter. Let $\mathbf{g}$ be the vector of derivatives of the log-likelihood w.r.t. the model parameters, i.e. the Jacobian, for some mini-batch of training instances. Then the moving average of the gradient is given by $\bar{\mathbf{g}}$:

$$\bar{\mathbf{g}} \leftarrow (1 - \beta)\bar{\mathbf{g}} + \beta\mathbf{g}, \tag{3.23}$$

where $\beta \in (0, 1)$ is a constant. Let the model parameters be $\boldsymbol{\theta}$. These are then updated according to:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \gamma_t\bar{\mathbf{g}} \tag{3.24}$$

Bengio argues that this method *"... removes some of the noise and oscillations that gradient descent has, in particular in the directions of high curvature of the [objective] function"*. For example, instead of jumping back and forth between the same two values for a parameter when receiving opposing training instances in consecutive mini-batches, the parameter will settle in the middle between the two allowing other parameters (which depended on it) to also stabilize.

In our algorithms we choose to adopt both the decaying learning rate and the momentum factor to improve the training procedure. We choose to use a separate validation set to find the optimal hyperparameters, such as the learning rate and momentum. The final training procedure with shuffled training data is given in algorithm 3 [13].

Indeed, recent results demonstrate that even without momentum deep MLPs can be trained

[13]The training procedure used in some of our experiments differed on two points. To facilitate debugging the validation error was calculated as a running average with each mini-batch update (on the same parameters as the mini-batch) and averaged at the end of the epoch. This allowed us to plot the validation error during each epoch. This difference should not change any results reported since all models chosen by the validation set were trained until the final or almost final. In some experiments we also choose to only shuffle the data (step 5) in the first epoch to reduce computational time.

**Algorithm 3** Stochastic gradient descent algorithm

**for** $\texttt{t} = 1, \ldots, \texttt{MaxEpochs}$ **do**
  $\gamma_t \leftarrow \dfrac{\gamma_0}{1 + \gamma_0 \lambda t}$ # Update learning rate
  TrainingData $\leftarrow$ Shuffle(TrainingData) # Shuffle training data
  **for** $\texttt{Batch} = 1, \ldots, \texttt{BatchCount}$ **do**
    $\mathbf{x}_1, \ldots, \mathbf{x}_N \leftarrow$ MiniBatch(TrainingData, Batch) # Retrieve disjoint mini-batch
    $\mathbf{g} \leftarrow \dfrac{\delta}{\delta \boldsymbol{\theta}} \dfrac{1}{N} \sum_{n=1}^{N} l(\boldsymbol{\theta}; \mathbf{x}_n)$ # Compute gradient w.r.t. log-likelihood
    $\bar{\mathbf{g}} \leftarrow (1 - \beta)\bar{\mathbf{g}} + \beta \mathbf{g}$ # Update moving average of gradient
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \gamma_t \bar{\mathbf{g}}$ # Update parameters w.r.t. found direction
  **end for**
  ValidationError $\leftarrow$ TestModel($\boldsymbol{\theta}$, ValidationData) # Test model on validation set
  **if** ValidationError $<$ BestValidationError **then** # Determine best model so far
    BestValidationError $\leftarrow$ ValidationError # Save new best error
    $\boldsymbol{\theta}_{\text{Best}} \leftarrow \boldsymbol{\theta}$ # Save new best model
  **end if**
**end for**

to reach state-of-the-art accuracy on classifying handwritten digits in the MNIST dataset (Claudiu Ciresan et al., 2010). However, this may require an enormous dataset.

More advanced methods can also be used to find the gradient update direction $\bar{\mathbf{g}}$. In particular, we will also use the method known as *Nesterov's accelerated gradient* (Nesterov, 1983). As derived in (Sutskever et al., 2013), the method is equivalent to replacing steps (6)–(8) in algorithm 3 with:

$$\bar{\mathbf{g}} \leftarrow \mu \bar{\mathbf{g}} + \gamma_t \frac{\delta}{\delta \boldsymbol{\theta}} \frac{1}{N} \sum_{n=1}^{N} l(\boldsymbol{\theta} + \mu \bar{\mathbf{g}}; \mathbf{x}_n), \tag{3.25}$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \bar{\mathbf{g}}, \tag{3.26}$$

where $\mu$ is an additional constant, which is allowed to vary at each epoch.

## 3.5 Mixture of Logistic Regressors and Mixture of Experts

### 3.5.1 Definition

We argued earlier that MLP and ConvNet models are powerful extensions of the LogReg model, but at the same time observed that training them was extremely difficult. The mixture of logistic regressors (also known as mixture of logistic models) is another extension of the logistic regression model, but one for which there exists a more efficient training procedure (Bishop, 2006, pp. 670–672). In the mixture of logistic regressors model, the probability distribution is defined as a convex combination of logistic regression models. For $I$ mixture components and classes $1, \ldots, K$, the model is defined by

$$P(y = k|\mathbf{x}) = \sum_{i=1}^{I} \pi_i p_i(y = k|\mathbf{x}), \tag{3.27}$$

where $\pi_i$ is the prior probability of each logistic regression, called a mixture component, such that $\pi_i \geq 0$ for $i = 1, \ldots, I$ and $\sum_i \pi_i = 1$, and $p_i(y = k|\mathbf{x})$ is the probability for class $k$ given by the logistic regression model:

$$p_i(y|\mathbf{x}) = \frac{\exp\left(\beta_{i,k_0} + \boldsymbol{\beta}_{i,k}^T \mathbf{x}\right)}{1 + \sum_{k'=1}^{K-1} \exp\left(\beta_{i,k'_0} + \boldsymbol{\beta}_{i,k'}^T \mathbf{x}\right)} \quad k = 1, \ldots, K, \tag{3.28}$$

where the mixture component parameters are $\beta_{i,k_0} \in \mathbb{R}$, $\boldsymbol{\beta}_{i,k} \in \mathbb{R}^M$ for $k = 1, \ldots, K$ and $i = 1, \ldots, I$.

With a single mixture component, the model is equivalent to the LogReg model, but by adding more mixture components it becomes possible to classify datasets which are not linearly separable. For example, the XOR problem can be solved with four mixture components, where each mixture component has prior probability $1/4$ and a linear decision boundary orthogonal to the lines between the center of the coordinate system and the four points $(1, 1), (1, -1), (-1, 1), (-1, -1)$. This is illustrated in figure 3.6.

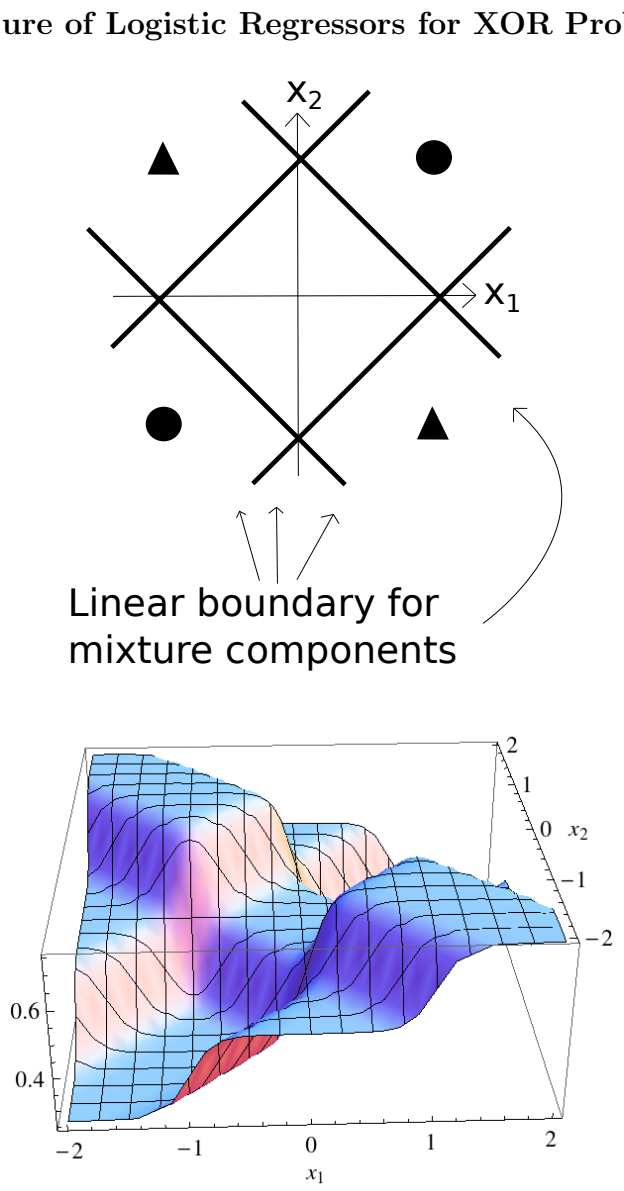



Figure 3.6: A mixture of logistic regressors with four components can solve the XOR problem. One simple solution is to define four mixture components, such that each has prior probability 1/4 and a linear decision boundary orthogonal to one of the four corners $(1,1),(1,-1),(-1,1),(-1,-1)$. Top) Illustration of linear decision boundaries for mixture components. Bottom) 3D plot of $P(y=1(\text{triangle})|x_1,x_2)$, where all mixture components have prior probabilities 1/4, and score functions $f_1(x_1,x_2)=10(x_1-x_2-1)$, $f_2(x_1,x_2)=10(-x_1+x_2-1)$, $f_3(x_1,x_2)=10(-x_1-x_2+1)$ and $f_4(x_1,x_2)=10(x_1+x_2+1)$ respectively.

However, this model is very crude since the same prior probabilities will be applied at every vector $\mathbf{x}$. We amend this by extending the model further, such that the prior probabilities for each mixture component depends on $\mathbf{x}$. This is known as the mixture of experts model, abbreviated ME (Jacobs et al., 1991). In the original formulation presented by Jacobs et al. the model was defined in terms of Gaussian distributions, but in our framework we shall reformulate it in terms of logistic functions. That is, each prior probability is going to be a logistic regression itself taking as input the vector $\mathbf{x}$. For $I$ mixture components and classes $1, \ldots, K$, we define the ME model for class $k$ to be:

$$P(y|\mathbf{x}) = \sum_{i=1}^{I} P(i|\mathbf{x}) p_i(y|\mathbf{x}), \tag{3.29}$$

where $p_i$ is defined as in the mixture of logistic regressors, and where

$$P(i|\mathbf{x}) = \frac{\exp\left(\gamma_{i_0} + \boldsymbol{\gamma}_i^T \mathbf{x}\right)}{1 + \sum_{i'=1}^{I-1} \exp\left(\gamma_{i'_0} + \boldsymbol{\gamma}_{i'}^T \mathbf{x}\right)} \quad i = 1, \ldots, I, \tag{3.30}$$

such that the parameters for the prior probabilities are $\gamma_{i_0} \in \mathbb{R}$, $\boldsymbol{\gamma}_i \in \mathbb{R}^M$ for $i = 1, \ldots, I$. Note that we have taken $k$ in $P(y = k|\mathbf{x})$ to be implicit and hence dropped it from our equations.

Clearly, this model is much more powerful than the mixture of logistic regressors. Of course, we can still solve the XOR problem by setting all parameters corresponding to the prior probabilities to zero, except for the intercept, and then apply the previous solution.

In the terminology of Jacobs et al., one imagines that each mixture component, now called an expert, *specializes* in predicting the class for its own subspace of inputs. Another way to view the model, is that the experts are *competing* with each other to produce the best prediction while the prior probabilities act as a *gating network*, which decides how to weight each expert in light of the data.

### 3.5.2 Training

In the seminal paper by (Jacobs et al., 1991), the model parameters were found using maximum likelihood by applying gradient descent on the log-likelihood of the entire dataset. Let our observations and corresponding classes be $\{\mathbf{x}_n, y_n\}_{n=1}^N$ and assume that each training instance

is sampled i.i.d. Then, the log-likelihood for the ME model is given by:

$$l(\boldsymbol{\theta}) = \log\left(\prod_{n=1}^{N} P(y_n|\mathbf{x}_n, \boldsymbol{\theta})\right) = \sum_{n=1}^{N} \log P(y_n|\mathbf{x}_n, \boldsymbol{\theta}) \tag{3.31}$$

$$= \sum_{n=1}^{N} \log \sum_{i=1}^{I} P(i|\mathbf{x}_n, \boldsymbol{\theta}) p_i(y_n|\mathbf{x}_n, \boldsymbol{\theta}) = \sum_{n=1}^{N} \log \sum_{i=1}^{I} P(i|\mathbf{x}_n, \boldsymbol{\theta}) P(y_n|i, \mathbf{x}_n, \boldsymbol{\theta}), \tag{3.32}$$

where $\boldsymbol{\theta}$ is the set of parameters defined earlier. It is straightforward to find the derivative w.r.t. any parameter.

Later, the expectation–maximization (EM) algorithm was proposed to train the ME model (Jordan and Jacobs, 1994). Contrary to gradient descent, which treats the log-likelihood as a black-box function, the EM algorithm exploits the structure of the model to iteratively improve the log-likelihood at every step. This appears to be a better training procedure in practice, since each step of the algorithm solves a convex optimization problem (Bishop, 2006, pp. 670–672). A theoretical justification for the superiority of EM on ME models is also given in (Jordan and Xu, 1995).

Before we describe the EM algorithm, we first observe that the model corresponds to a joint distribution over the output class $k$ and *expert* i, where the random variable over mixture components have been marginalized out. We call the un-marginalized distribution for the joint distribution (sometimes called the full data likelihood). It is given by:

$$P(y, i|\mathbf{x}, \boldsymbol{\theta}) = P(i|\mathbf{x}) P(y|\mathbf{x}, i, \boldsymbol{\theta}) \tag{3.33}$$

$$= \frac{\exp\left(\gamma_{i_0} + \boldsymbol{\gamma}_i^T \mathbf{x}\right)}{1 + \sum_{i'=1}^{I-1} \exp\left(\gamma_{i'_0} + \boldsymbol{\gamma}_{i'}^T \mathbf{x}\right)} \frac{\exp\left(\beta_{i,k_0} + \boldsymbol{\beta}_{i,k}^T \mathbf{x}\right)}{1 + \sum_{k'=1}^{K-1} \exp\left(\beta_{i,k'_0} + \boldsymbol{\beta}_{i,k'}^T \mathbf{x}\right)}. \tag{3.34}$$

To motivate the EM algorithm we follow the general introduction given in (Sahani, 2013, Lecture 3). We can bound the log-likelihood by introducing a new distribution $q_n(i)$ over the

mixture components and use Jensen's inequality:

$$l(\boldsymbol{\theta}) = \sum_{n=1}^{N} \log P(y_n|\mathbf{x}_n, \boldsymbol{\theta}) \tag{3.35}$$

$$= \sum_{n=1}^{N} \log \left( \sum_{i=1}^{I} P(i|\mathbf{x}_n, \boldsymbol{\theta}) P(y_n|i, \mathbf{x}_n, \boldsymbol{\theta}) \right) \tag{3.36}$$

$$= \sum_{n=1}^{N} \log \left( \sum_{i=1}^{I} q_n(i) \frac{P(i|\mathbf{x}_n, \boldsymbol{\theta}) P(y_n|i, \mathbf{x}_n, \boldsymbol{\theta})}{q_n(i)} \right) \tag{3.37}$$

$$\geq \sum_{n=1}^{N} \sum_{i=1}^{I} q_n(i) \log \left( \frac{P(i|\mathbf{x}_n, \boldsymbol{\theta}) P(y_n|i, \mathbf{x}_n)}{q_n(i)} \right) \stackrel{\text{def}}{=} F(q, \boldsymbol{\theta}), \tag{3.38}$$

where $F$ is a lower bound on $l(\boldsymbol{\theta})$ called the free energy. The EM algorithm now iteratively estimates the distribution over mixture components (E-Step) and the model parameters (M-Step) to minimize the free energy. See algorithm 4.

**Algorithm 4** EM algorithm

1: **for** `Iteration` $= 1, \ldots,$ `MaxIterations` **do**
2: $\quad q \leftarrow \arg\max_{q'} F(q', \boldsymbol{\theta})$ # E-Step: Update distribution $q$ for fixed $\boldsymbol{\theta}$
3: $\quad \boldsymbol{\theta} \leftarrow \arg\max_{\boldsymbol{\theta'}} F(q, \boldsymbol{\theta'})$ # M-Step: Update parameters $\boldsymbol{\theta}$ for fixed $q$
4: **end for**

We can further split the probabilities to show that:

$$
\begin{aligned}
F(q, \boldsymbol{\theta}) &= \sum_{n=1}^{N} \sum_{i=1}^{I} q_n(i) \log \left( \frac{P(i|\mathbf{x}_n, \boldsymbol{\theta}) P(y_n|i, \mathbf{x}_n, \boldsymbol{\theta})}{q_n(i)} \right) \\
&= \sum_{n=1}^{N} \sum_{i=1}^{I} q_n(i) \log \left( P(i|\mathbf{x}_n, \boldsymbol{\theta}) P(y_n|i, \mathbf{x}_n) \right) - \sum_{n=1}^{N} \sum_{i=1}^{I} q_n(i) \log q_n(i) \\
&= \sum_{n=1}^{N} \sum_{i=1}^{I} q_n(i) \log \left( P(i|\mathbf{x}_n, \boldsymbol{\theta}) P(y_n|i, \mathbf{x}_n, \boldsymbol{\theta}) \right) + \mathbf{H}[q], \qquad (3.39) \\
F(q, \boldsymbol{\theta}) &= \sum_{n=1}^{N} \sum_{i=1}^{I} q_n(i) \log \left( \frac{P(i|\mathbf{x}_n, \boldsymbol{\theta}) P(y_n|i, \mathbf{x}_n, \boldsymbol{\theta})}{q_n(i)} \right) \\
&= \sum_{n=1}^{N} \sum_{i=1}^{I} q_n(i) \log \left( \frac{P(i|y_n, \mathbf{x}_n, \boldsymbol{\theta}) P(y_n|\mathbf{x}_n, \boldsymbol{\theta})}{q_n(i)} \right) \\
&= l(\boldsymbol{\theta}) - \sum_{n=1}^{N} \mathbf{KL} \left[ q_n(i) || P(i|y_n, \mathbf{x}_n, \boldsymbol{\theta}) \right] \qquad (3.40)
\end{aligned}
$$

where $\mathbf{H}[q] = -\sum_{n=1}^{N} \sum_{i=1}^{I} q_n(i) \log q_n(i)$ is the entropy of $q$ over all training instances, and $\mathbf{KL}$ is the Kullback–Leibler (KL) divergence. From eq. (3.40), we see that we can maximize $F$ w.r.t. $q_n$ by minimizing the Kullback–Leibler divergence $\mathbf{KL}\left[q_n(i)||P(i|y_n, \mathbf{x}_n, \boldsymbol{\theta})\right]$. The KL divergence always takes non-negative values and is zero if and only if the two distributions are equal (Bishop, 2006, pp. 55–58; Sahani, 2013, Lecture 3):

$$
q_n(i) = P(i|y_n, \mathbf{x}_n, \boldsymbol{\theta}) \qquad (3.41)
$$

This is exactly what the E-Step will do. Thus, after each E-Step $F(q, \boldsymbol{\theta}) = l(\boldsymbol{\theta})$. Next, we observe that there is no closed-form solution for the M-Step. So instead we apply a gradient descent update based on the derivative of $F(q, \boldsymbol{\theta})$ w.r.t. all model parameters. If the step-size is small enough, then $F$ is strictly increased since the function is smooth. To keep things simple, we choose not to apply any form of decaying learning rate or momentum. The procedure is given in algorithm 5.

Let $m$ be the training iteration, if the step-size is sufficiently small, then we observe that

**Algorithm 5** EM-based stochastic gradient descent for ME algorithm

**for** $\texttt{t} = 1, \ldots, \texttt{MaxEpochs}$ **do**
  TrainingData $\leftarrow$ Shuffle(TrainingData) # Shuffle training data
  **for** $\texttt{Batch} = 1, \ldots, \texttt{BatchCount}$ **do**
    $\mathbf{x}_1, \ldots, \mathbf{x}_N \leftarrow$ MiniBatch(TrainingData, Batch) # Retrieve disjoint mini-batch
    $q \leftarrow \arg\max_{q'} F(q', \boldsymbol{\theta} | \mathbf{x}_1, \ldots, \mathbf{x}_N)$ # E-Step on mini-batch
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \gamma \nabla_{\boldsymbol{\theta}} F(q, \boldsymbol{\theta'} | \mathbf{x}_1, \ldots, \mathbf{x}_N)$ # M-Step with gradient descent on mini-batch
  **end for**
  ValidationError $\leftarrow$ TestModel($\boldsymbol{\theta}$, ValidationData) # Test model on validation set
  **if** ValidationError $<$ BestValidationError **then** # Determine best model so far
    BestValidationError $\leftarrow$ ValidationError # Save new best error
    $\boldsymbol{\theta}_{\text{Best}} \leftarrow \boldsymbol{\theta}$ # Save new best model
  **end if**
**end for**

the log-likelihood is strictly increased after each iteration of an E-Step followed by an M-Step:

$$F(q^{(m)}, \boldsymbol{\theta}^{(m)}) \leq F(q^{(m+1)}, \boldsymbol{\theta}^{(m)}) = l(\boldsymbol{\theta}^{(m)}) \leq F(q^{(m+1)}, \boldsymbol{\theta}^{(m+1)}) \leq F(q^{(m+2)}, \boldsymbol{\theta}^{(m+1)}) \ldots \quad (3.42)$$

In fact, because the free energy is increased at every iteration and bounded by the log-likelihood, which itself is also bounded above, the procedure is guaranteed to converge to a fixed point.

Obviously, the proof assumes that we apply the procedure to the entire dataset at every step. This will not be computationally tractable in our experiments. Instead, we shall apply the procedure to mini-batches of training instances. If these are made large enough, then we would still expect the algorithm to converge to a fixed point.

We can introduce L1-regularization in the procedure, by subtracting the absolute parameter value,s excluding intercepts, multiplied by a positive constant from the free energy. One can show that this is equivalent to subtracting the same term from the log-likelihood and carrying out the above derivations again without explicitly treating the parameters as random variables.

### 3.5.3 Interpretation and discussion

Unlike neural networks, the ME operate in a divide-and-conquer fashion. This makes them suitable for approximating highly non-smooth functions as demonstrated by (Wiering, 1995, pp.

28–44). Wiering further showed that a more advanced version of the ME model, known as the hierarchical mixtures of experts model (Jordan and Jacobs, 1994), was effective in learning to play endgames in backgammon due to the smoothness of the representation he designed (Wiering, 1995).

# Chapter 4

# Self-Play Learning

## 4.1 Temporal Difference Learning

Before we move on to the main algorithm for self-play learning, it is instructive to briefly introduce TD (temporal difference) learning. We will follow the motivation given by (Baxter et al., 2000), and refer the reader to (Sutton and Barto, 1998) for an excellent treatment of more broad area of reinforcement learning [1]. Suppose we are learning to predict the pay-off of a game by observing a series of games played by fixed players, and that we only have game outcomes to learn from. Suppose we have sequences of board states $\mathbf{x}_1^n, \ldots, \mathbf{x}_{T_n}^n$, where $T_n$ is the number of board positions in game $n$, for games $n = 1, \ldots, N$. Let $\tilde{P}$ be the model for the heuristic evaluation function, which is parametrized by $\boldsymbol{\theta}$. Define the expected pay-off of $\mathbf{x}_t^n$ under the model as:

$$\mathrm{E}_{\tilde{P}}\left[\text{Pay-off}|\mathbf{x}_t^n, \boldsymbol{\theta}\right] \tag{4.1}$$

Assume that it always returns the true value at endgame positions. We then define the *temporal difference* associated with the move $\mathbf{x}_t^n \to \mathbf{x}_{t+1}^n$ as:

$$d_t := \mathrm{E}_{\tilde{P}}\left[\text{Pay-off}|\mathbf{x}_{t+1}^n, \boldsymbol{\theta}\right] - \mathrm{E}_{\tilde{P}}\left[\text{Pay-off}|\mathbf{x}_t^n, \boldsymbol{\theta}\right] \tag{4.2}$$

[1] The reader should note that TD learning was introduced much earlier. See for example (Sutton, 1988).

The temporal difference measures the discrepancy between the estimated pay-off before and after the move $\mathbf{x}_t^n \rightarrow \mathbf{x}_{t+1}^n$. Since the players are fixed and since the pay-off at endgame positions are always known, minimizing the absolute temporal difference will make the model more accurate on predicting the outcome at earlier game positions. In particular, under perfect play by both players, the temporal difference w.r.t. the true model $p$ will always be zero since the actions are deterministic and the pay-off at end of the game known:

$$\mathrm{E}_P\left[\text{Pay-off}|\mathbf{x}_{t+1}^n, \boldsymbol{\theta}\right] - \mathrm{E}_P\left[\text{Pay-off}|\mathbf{x}_t^n, \boldsymbol{\theta}\right] = 0 \tag{4.3}$$

Effectively, the model is updating its parameters to assign a correct pay-off at one board position towards the pay-off at a future board position. This is known as bootstrapping. If the model is a lookup table over all possible board positions, e.g. its input features are a set of indicator functions where each function takes value one in one unique board position and otherwise zero, then the model will converge to the true expected pay-off value as the number of games observed go to infinity for any fixed players. If the model is not a lookup table, and hence is not able to assign a unique expected pay-off value for each board position, then there is no guarantee that this procedure will converge. This is termed function approximation, and is the category that all of our models fall into. See also (Sutton and Barto, 1998, Chapter 6). The procedure is illustrated in 4.1.

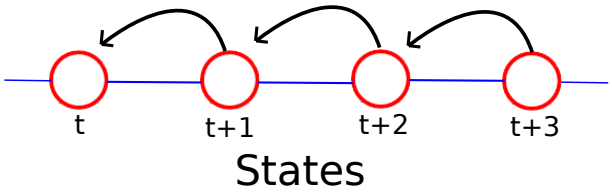



Figure 4.1: Learning in TD(0). The arrows indicate that parameters for states are updated towards the parameters of future states. This is known as bootstrapping.

Let $J(s, s')$ be the objective function we want to maximize. Then we find our parameters $\boldsymbol{\theta}$:

$$\hat{\boldsymbol{\theta}} = \arg\max_{\boldsymbol{\theta}} \sum_{n=1}^{N} \sum_{t=1}^{T_n} J\left(\mathrm{E}_{\tilde{P}}\left[\text{Pay-off}|\mathbf{x}_{t+1}^n, \boldsymbol{\theta}\right], \mathrm{E}_{\tilde{P}}\left[\text{Pay-off}|\mathbf{x}_t^n, \boldsymbol{\theta}\right]\right). \tag{4.4}$$

Typically, the objective function is chosen to be the (negative) squared error

$$J(s, s') = -(s - s')^2, \tag{4.5}$$

and minimized with stochastic gradient descent. This is the most standard form of the TD(0) algorithm.

We can extend the procedure by letting it bootstrap not only between the parameters in two consecutive board positions, but between any two board positions in the same game. Let $\lambda \in (0, 1)$. Then define for $k \in \mathbb{N}$:

$$R_t^{n,k} = \begin{cases} \mathrm{E}_{\tilde{P}}\left[\text{Pay-off}|\mathbf{x}_{T_n}^n, \boldsymbol{\theta}\right] & \text{if } \ t + k \geq T_n, \\ \mathrm{E}_{\tilde{P}}\left[\text{Pay-off}|\mathbf{x}_{t+k}^n \boldsymbol{\theta}\right] & \text{otherwise.} \end{cases} \tag{4.6}$$

This is simply the estimated pay-off w.r.t. the board position at $\mathbf{x}_{t+k}^n$ if the game is longer than $t + k$ moves, and the actual game outcome if the game ends at or before $t + k$ moves. Now we find our parameters $\boldsymbol{\theta}$ by:

$$\hat{\boldsymbol{\theta}} = \arg\max_{\boldsymbol{\theta}} \sum_{n=1}^{N} \sum_{t=1}^{T_n} J_{\boldsymbol{\theta}}(R_t^{\lambda,n}, \mathrm{E}_{\tilde{P}}\left[\text{Pay-off}|\mathbf{x}_t^n, \boldsymbol{\theta}\right]), \tag{4.7}$$

where $R_t^{\lambda,n}$, similar to a geometric series, is defined by:

$$R_t^{\lambda,n} = (1 - \lambda) \sum_{k=1}^{\infty} \lambda^{k-1} R_t^{n,k}. \tag{4.8}$$

Thus, the $\lambda$ value specifies the degree of bootstrapping backwards in time. If $\lambda = 0$ then $R_t^{\lambda,n} = \mathrm{E}_{\tilde{P}}\left[\text{Pay-off}|\mathbf{x}, \boldsymbol{\theta}\right]$, and the procedure is equivalent to the previous procedure. As $\lambda$ increases, parameters of positions are updated toward the vales at positions further in the future.

## 4.2 TreeStrap and Its Extensions

The above procedure works well in some games, such as Backgammon, but it has failed to yield success in chess as we will discuss in chapter 5. Part of the reason seems to be that it does not take into account the search tree generated by the minimax algorithm. Training a heuristic evaluation function without making use of the search tree, and then evaluating it at the leaf nodes of the search tree obviously causes a discrepancy between the training instances and the instances it is applied to.

To handle this, Baxter et al. first proposed to replace the actual board positions played with those in the leaf nodes of the minimax search tree (Baxter et al., 2000). This approach, termed TD-Leaf, appeared to work much better. Their program KnightCap was given a set of hand-crafted, but unweighted, features and was able to achieve master-level play with a linear heuristic evaluation function from a sensible initialization point when trained against human opponents.

Later, Veness et al. proposed a new procedure, which they termed TreeStrap (Veness et al., 2009). They proposed that the heuristic evaluation function, for any board position in the minimax search tree, be updated towards its minimax value. This is illustrated in figure 4.2. Their program *Meep* was given a set of hand-crafted, but unweighted, features and used TreeStrap to train a linear heuristic evaluation function from an arbitrary initialization point. After being either trained with self-play or against a strong chess engine, named Shredder, it reached master-level play on the online chess server named Internet Chess Club. Furthermore, in a tournament against other variants of the procedure, including TD-Leaf, the TreeStrap procedure ranked first. Taken together, this signals that TreeStrap is a highly robust training procedure. Veness et al. compared TreeStrap to TD-Leaf training from self-play, and showed that TreeStrap outperformed TD-Leaf substantially when trained on the same number of games.

To the authors knowledge, this is currently the state-of-the-art method for learning a heuristic evaluation function when training with self-play only. For this reason, we choose to use TreeStrap to learn from self-play in order to keep with our goal of using less expert knowledge.

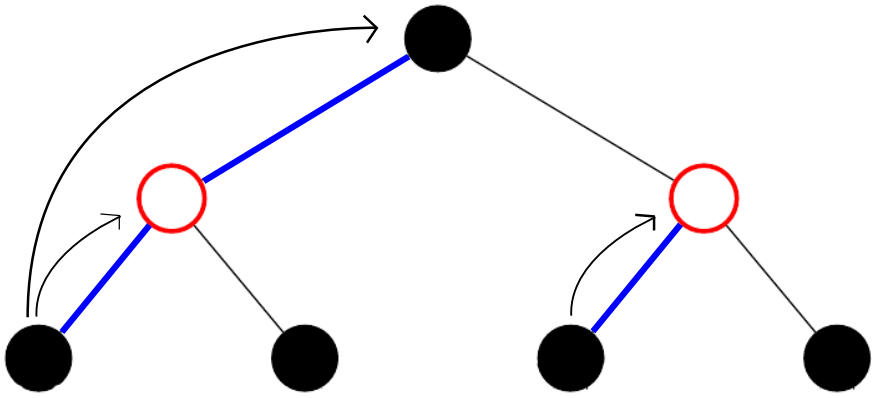


Figure 4.2: Bootstrapping in TreeStrap. The arrows indicate, which state values are updated towards other state values.

The reason TreeStrap performed better was argued to be because it made use of much more information in the search tree (Veness et al., 2009). TreeStrap performs many more updates for each actual game position, and is able to take advantage of the game rules to a higher degree. In particular, all checkmate positions would be scored correctly and propagated back to earlier nodes in the tree. Furthermore, contrary to TD-Leaf which only considers positions in the actually played, TreeStrap considers all positions reachable in the minimax search tree. Since ultimately the heuristic evaluation function will be evaluated at the leaf nodes of the minimax search tree (or quiescence search tree), training on all positions in the minimax search tree will obviously be closer to the distribution it will be evaluated on. Lastly, since the updates in TreeStrap only depend on the current board position, it is less sensitive to the opponent than TD-Leaf.

The original TreeStrap algorithm was designed to update the parameters of a regression model by squared error w.r.t. expected pay-off, but our models are defined in a probabilistic framework. We therefore propose to generalize TreeStrap to non-linear models and arbitrary objective functions. We further propose three new objective functions, which can be made compatible with our probabilistic classification models.

To simplify our problem, we shall assume that draw is not a possible outcome. This effectively makes all our classification models binary. Under this assumption, one can show that ranking board positions according to expected pay-off in a three-class LogReg model is equiv-

alent to ranking board positions in a two-class LogReg model with the same parameters for classes White wins and Black wins. The derivation is straightforward, but as we shall use it to initialize our models it is instructive to demonstrate it. For simplicity, assume the model is playing White. Without loss of generality, drop the assumption that the score function for the last class is set to zero and drop the intercept parameter. Let $\mathbf{x_1}$ and $\mathbf{x_2}$ be the feature sets associated with two distinct board positions, such that:

$$\mathrm{E}_{\tilde{P}_1}\left[\text{Pay-off}|\mathbf{x}_1, \boldsymbol{\theta}_1\right] \leq \mathrm{E}_{\tilde{P}_1}\left[\text{Pay-off}|\mathbf{x}_2, \boldsymbol{\theta}_1\right] \tag{4.9}$$

where $\tilde{P}_1$ is the multinomial logistic regression model with three classes and parameters $\boldsymbol{\theta}_1$ and and $\tilde{P}_2$ is a logistic regression model with two classes and parameters $\boldsymbol{\theta}_2$. Then we use our assumption $\tilde{P}_1(\text{Draw}|\mathbf{x_1}, \boldsymbol{\theta}_1) = \tilde{P}_1(\text{Draw}|\mathbf{x_2}, \boldsymbol{\theta}_1) = 0$ to derive:

$$\begin{aligned}
&\mathrm{E}_{\tilde{P}_1}\left[\text{Pay-off}|\mathbf{x}_1, \boldsymbol{\theta}_1\right] \leq \mathrm{E}_{\tilde{P}_1}\left[\text{Pay-off}|\mathbf{x}_2, \boldsymbol{\theta}_1\right] \\
&\Leftrightarrow \\
&\tilde{P}_1(\text{White wins}|\mathbf{x_1}, \boldsymbol{\theta}_1) - \tilde{P}_1(\text{Black wins}|\mathbf{x_1}, \boldsymbol{\theta}_1) \\
&\leq \tilde{P}_1(\text{White wins}|\mathbf{x_2}, \boldsymbol{\theta}_1) - \tilde{P}_1(\text{Black wins}|\mathbf{x_2}, \boldsymbol{\theta}_1) \\
&\Leftrightarrow \\
&\tilde{P}_1(\text{White wins}|\mathbf{x_1}, \boldsymbol{\theta}_1) - \left(1 - \tilde{P}_1(\text{White wins}|\mathbf{x_1}, \boldsymbol{\theta}_1)\right) \\
&\leq \tilde{P}_1(\text{White wins}|\mathbf{x_2}, \boldsymbol{\theta}_1) - \left(1 - \tilde{P}_1(\text{White wins}|\mathbf{x_2}, \boldsymbol{\theta}_1)\right) \\
&\Leftrightarrow \\
&2\tilde{P}_1(\text{White wins}|\mathbf{x_1}, \boldsymbol{\theta}_1) - 1 \leq 2\tilde{P}_1(\text{White wins}|\mathbf{x_2}, \boldsymbol{\theta}_1) - 1 \\
&\Leftrightarrow \\
&\tilde{P}_1(\text{White wins}|\mathbf{x_1}, \boldsymbol{\theta}_1) \leq \tilde{P}_1(\text{White wins}|\mathbf{x_2}, \boldsymbol{\theta}_1) \\
&\Leftrightarrow \\
&\tilde{P}_2(\text{White wins}|\mathbf{x_1}, \boldsymbol{\theta}_2) \leq \tilde{P}_2(\text{White wins}|\mathbf{x_2}, \boldsymbol{\theta}_2) \\
&\Leftrightarrow \\
&\mathrm{E}_{\tilde{p}_2}\left[\text{Pay-off}|\mathbf{x}_1, \boldsymbol{\theta}_2\right] \leq \mathrm{E}_{\tilde{p}_2}\left[\text{Pay-off}|\mathbf{x}_2, \boldsymbol{\theta}_2\right]
\end{aligned}$$

To simplify notation, let $H_{\boldsymbol{\theta}}(\mathbf{s})$ be the heuristic evaluation function at board position $\mathbf{s}$, i.e. $H_{\boldsymbol{\theta}}(\mathbf{s}) = \tilde{P}(\text{White wins}|\boldsymbol{\theta}, \mathbf{s})$, w.r.t. $\boldsymbol{\theta}$. The high-level TreeStrap procedure is then given in algorithm 6. The algorithm assumes that search tree of board positions, e.g. to a certain depth, is already available. In the algorithm, $J$ is the real-valued objective function and can be any of those proposed in table 4.2, Minimax$(\mathbf{s}, H_{\boldsymbol{\theta}}, D)$ is a function which returns the minimax values of the search tree w.r.t. a (truncated) minimax search (and possibly quiescence search) from board position $\mathbf{s}$ w.r.t. $H_{\boldsymbol{\theta}}(\mathbf{s})$ to depth $D$, $\eta$ is a step-size (called the learning rate), $A$ is the set of possible moves and $\mathbf{s} \circ a$ is the board position when taking move $a$ from $\mathbf{s}$.

**Algorithm 6** Non-linear TreeStrap minimax algorithm

Initialize $t \leftarrow 1, \mathbf{s_1} \leftarrow$ start state
**while** state $\mathbf{s_t}$ is not terminal **do** # Continue until game ends
    $V \leftarrow \text{Minimax}(\mathbf{s_t}, H_{\boldsymbol{\theta}}, D)$ # Compute minimax value estimates
    $\Delta\boldsymbol{\theta} \leftarrow 0$ # Reset gradients
    **for** $\mathbf{s} \in$ search tree **do** # Accumulate gradients over tree nodes
        $\Delta\boldsymbol{\theta} \leftarrow \Delta\boldsymbol{\theta} + \nabla_{\boldsymbol{\theta}} J(V(\mathbf{s}), H_{\boldsymbol{\theta}}(\mathbf{s}))$
    **end for**
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \eta\Delta\boldsymbol{\theta}$ # Update as in stochastic gradient descent
    Select $a_t = \arg\max_{a \in A} V(\mathbf{s_t} \circ a)$ # Select best move
    Execute move $a_t$, receive $\mathbf{s_{t+1}}$ # Execute move and update state features
    $t \leftarrow t + 1$
**end while**

| **Objective Function** | **Definition** |
|---|---|
| Squared Error | $J(V(\mathbf{s}), H_{\boldsymbol{\theta}}(\mathbf{s})) = -\left(V(\mathbf{s}) - H_{\boldsymbol{\theta}}(\mathbf{s})\right)^2$ |
| Hard Cross-Entropy | $J(V(\mathbf{s}), H_{\boldsymbol{\theta}}(\mathbf{s})) = 1_{V(\mathbf{s}) \geq 0.5} \log H_{\boldsymbol{\theta}}(\mathbf{s}) + 1_{V(\mathbf{s}) < 0.5} \log\left(1 - H_{\boldsymbol{\theta}}(\mathbf{s})\right)$ |
| Soft Cross Entropy | $J(V(\mathbf{s}), H_{\boldsymbol{\theta}}(\mathbf{s})) = V(\mathbf{s}) \log H_{\boldsymbol{\theta}}(\mathbf{s}) + (1 - V(\mathbf{s})) \log\left(1 - H_{\boldsymbol{\theta}}(\mathbf{s})\right)$ |
| Reversed KL Divergence | $J(V(\mathbf{s}), H_{\boldsymbol{\theta}}(\mathbf{s})) = H_{\boldsymbol{\theta}}(\mathbf{s}) \log V(\mathbf{s}) + (1 - H_{\boldsymbol{\theta}}(\mathbf{s})) \log\left(1 - V(\mathbf{s})\right)$ |

Table 4.1: Proposed objective functions for non-linear TreeStrap procedure.

The squared error objective function worked well in (Veness et al., 2009), where it was used to weight the Meep chess engine features linearly. The hard cross-entropy is the most direct reformulation of our classification model, which simply takes probabilities above 0.5 to indicate White wins and below to indicate Black wins. However, it throws away a lot of information available regarding the uncertainty of the predicted and bootstrapped minimax values. The soft cross-entropy [2], which is equivalent to minimizing the KL divergence $\mathbf{KL}[V|H_{\boldsymbol{\theta}}]$, takes into account the uncertainty of both the predicted and bootstrapped minimax values. This makes it a better candidate than the hard cross-entropy, and can possibly perform better than the squared error. In addition to this, it has a natural interpretation as minimizing the amount of information (measured in *nats*) required to specify the outcome of the game (Bishop, 2006, p. 55). That is, it is effectively minimizing the amount of information gained by carrying out deeper searches. Minimizing the reversed KL divergence has been proposed in other machine learning problems (Bishop, 2006, pp. 467–469). This also takes into account the uncertainty in the predicted and bootstrapped minimax values, but tends to approximate one of the distribution modes more heavily than all others. Since our problem is binary there can only be one mode to approximate, which should force the model to always make $V(\mathbf{s})$ values very close to zero or very close to one agree with $H_{\boldsymbol{\theta}}(\mathbf{s})$. We might hope that this leads to faster convergence of the algorithm.

Given a logistic regression model, it would seem quite arbitrary to apply a squared error objective function to the winning probability. Instead, here we advocate to use the score value, i.e. the models output before being transformed by the logistic function. If the model contains parameters for both White wins and Black wins classes, i.e. neither of the score functions were fixed to zero [3], one can take the score function White wins minus the score function for Black wins as the expected pay-off. Clearly, this will ensure that the expected pay-off (w.r.t. White) is positive when White has a higher probability of winning than losing, and negative when White has a lower probability of winning than losing.

We further note that there is some empirical evidence showing that adding the logistic function to "squash" the output of the minimax value can improve training. For example, in

[2] We call it soft cross-entropy, because it is based on the bootstrapped values from the minimax search tree contrary to cross-entropy in our supervised experiments, which were binary and based on actual game outcomes.

[3] This will happen in our experiments later, where we convert out three-class logistic regression models to two-class logistic regression by removing the parameters related to the draw outcome.

(Hoki and Kaneko, 2014) a linearly transformed logistic function on top of a linear combination of features seems to perform very well and, as the authors suggest, perhaps better than a linear combination of features optimized without the logistic transformation, e.g. w.r.t. a squared error objective function. For this reason, the logistic regression model trained with soft cross-entropy may perform better than a linear regression model trained with squared error, given the same set of features.

The procedure given in algorithm 6 uses all board positions in the minimax tree to update the heuristic evaluation function in a stochastic gradient decent fashion. However, as we argued in section 3.4, for MLPs and ConvNets it is important to train the model based on mini-batches of a certain size, and in such a way that each mini-batch contains an equal number of instances with each game outcome and preferably positions with different characteristics. We handle this first by modifying the algorithm to accumulate a set of board positions and corresponding minimax values over many moves, and then train on random mini-batches sampled from these. After the accumulated set of board positions grows to a certain size, new training instances will overwrite old training instances [4]. Secondly, for every board position in the minimax search tree we create a new training instance by flipping the board position and inverting the piece colors, and use it for training as well. This will ensure that, even if the model is learning from a game where one player has decidedly lost already, there will always be an equivalent amount of board positions with high estimated pay-off for both White and for Black. Furthermore, if the training procedure is computationally costly, which it is bound to be for larger hierarchical models, it is perhaps better to train only on a subset of the minimax search tree. This should preferably be board positions that have low uncertainty regarding the minimax values assigned to them, which is most often the case for the early nodes in the minimax search tree as they integrate information over many. We therefore choose to train our heuristic evaluation function on a predefined number of the first board positions ordered by increasing depth in the minimax search tree.

The new procedure is given in algorithm 7, where FlipBoardPosition($\mathbf{s}$) flips the board and inverts the piece colors, $N$ is the number of training samples to keep in the accumulated training set $\{X, Y\}$, RandomSample$(X, Y, M)$ returns $M$ uniformly randomly selected training instances (without replacement) from the training set $\{X, Y\}$ and DeleteEarliestElements$(X, Y, M)$ deletes

[4] This is quite efficient from an implementation perspective.

the first $M$ entries in $\{X, Y\}$.

**Algorithm 7** Non-linear TreeStrap minimax over mini-batches algorithm

1: Initialize $t \leftarrow 1, \mathbf{s_1} \leftarrow$ start state
2: Initialize training set $\{X, Y\} \leftarrow \{\emptyset, \emptyset\}$
3: **while** state $\mathbf{s_t}$ is not terminal **do** # Continue until game ends
4: $V \leftarrow \text{Minimax}(\mathbf{s_t}, H_{\boldsymbol{\theta}}, D)$ # Compute minimax value estimates
5: **for** $\mathbf{s} \in$ search tree **do** # Accumulate gradients over tree nodes
6: $X \leftarrow X \cup \{\mathbf{s}, \text{FlipBoardPosition}(\mathbf{s})\}$
7: $Y \leftarrow Y \cup \{V(\mathbf{s}), -V(\mathbf{s})\}$
8: **end for**
9: **if** $|X| \geq N$ **then** # Are there sufficient training instances?
10: # Sample mini-batch and delete earliest entries in training set
11: $X_{\text{mini-batch}}, Y_{\text{mini-batch}} \leftarrow \text{RandomSample}(X, Y, M)$
12: $X, Y \leftarrow \text{DeleteEarliestElements}(X, Y, M)$
13: $\Delta\boldsymbol{\theta} \leftarrow 0$ # Reset gradients
14: **for** $\mathbf{s}, v \in \{X_{\text{mini-batch}}, Y_{\text{mini-batch}}\}$ **do** # Accumulate gradients over mini-batch
15: $\Delta\boldsymbol{\theta} \leftarrow \Delta\boldsymbol{\theta} + \nabla_{\boldsymbol{\theta}} J(v, H_{\boldsymbol{\theta}}(\mathbf{s}))$
16: **end for**
17: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \eta\Delta\boldsymbol{\theta}$ # Update as in stochastic gradient descent
18: **end if**
19: Select $a_t = \arg\max_{a \in A} V(\mathbf{s_t} \circ a)$ # Select best move
20: Execute move $a_t$, receive $\mathbf{s_{t+1}}$ # Execute move and update state features
21: $t \leftarrow t + 1$
22: **end while**

We obtain Nesterov's accelerated gradient variant of algorithm 7, by replacing steps (13)–(17) with:

$$\Delta\boldsymbol{\theta} \leftarrow \mu\Delta\boldsymbol{\theta} + \sum_{\mathbf{s}, v \in \{X_{\text{mini-batch}}, Y_{\text{mini-batch}}\}} \nabla_{\boldsymbol{\theta}} J(v, H_{\boldsymbol{\theta}+\mu\Delta\boldsymbol{\theta}}(\mathbf{s})), \tag{4.10}$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}, \tag{4.11}$$

where $\mu \in (0, 1)$ is again a constant, which is allowed to vary with each mini-batch.

## 4.3 Discussion

There are pros and cons of learning from self-play. Self-play requires only the model itself to generate training examples. Ghory argues that this implies the model will not become biased towards the strategies of any fixed experts, be they human or computers (Ghory, 2004, p. 23). Based on a literature review, he further argues that learning by playing works most effectively when playing against an opponent of the same level, which is exactly what happens during self-play. A similar observation is noted by Furnkranz in his literature review (Fürnkranz, 2001). However, the disadvantages of self-play is that it takes a long time to generate training data (when compared to a game collection), and that learning in the initial phases will be slow.

Nevertheless, other self-play training procedures have been proposed with success. Samuel initially tried to apply the same self-play procedure as Veness et al., but failed (Samuel, 2000). Instead he divided the program into two players called *Alpha* and *Beta*. Player *Alpha* would update its parameters while playing, and the parameters of *Beta* would stay fixed. After each game the outcome was recorded. If *Alpha* consistently won a fixed number of matches (in Samuel's paper, only a single match), then the parameters of *Alpha* would be copied to *Beta*. If, on the other hand, *Beta* consistently won a fixed number of matches (in Samuel's paper, three matches), it was assumed that *Alpha* had updated its parameters in the wrong direction, and the parameters of *Beta* would be copied back to *Alpha*. Samuel argued that this was necessary in order to find the optimal point in the high dimensional parameter space, since the program could easily get stuck in local maxima. Lee and Mahajan have criticized this procedure since as one player may win either due to: 1) a superior evaluation function, 2) luck and 3) errors made by the opponent (Lee and Mahajan, 1988). In particular, for weak players, the observation that the algorithm with one set of parameters wins over another with a different parameters is a highly unreliable estimate. It would, however, still be interesting to apply this procedure, if the standard self-play procedure does not work well.

An interesting idea Veness has proposed, based on unpublished positive results from a third party, is to use the board positions from the quiescence search, instead of the standard minimax search. This might work better for two reasons. Firstly, the quiescence search positions are easier to assign a numeric value, because they are 'quiet'.

Secondly, these are exactly the positions the heuristic evaluation function has to evaluate during the quiescent search applied, which is applied after the minimax search (Veness, 2014b).

# Chapter 5

# Previous Work on Learning The Heuristic Evaluation Function

The most studied problem in game playing is the automatic adjustment of the heuristic evaluation function (Fürnkranz, 2001, 2007). Most often a set of features has hand-crafted, as in Shannon's proposal, and the problem is to weight the features correctly through some linear or non-linear function. This is the subject of this chapter, where we go through previous related work on learning the heuristic evaluation function.

## 5.1 Simple and Complex Features

At this point, it is important to establish what we mean when we say that we want to *learn a heuristic evaluation function*. There exists a continuous spectrum of degrees to which the heuristic evaluation function can be been defined by a human, ranging from an exhaustive set of hand-crafted features (which need only to be weighted linearly) to a set of primitive variables describing the game position (which need to be compounded non-linearly to form an accurate evaluation function) [1]. In particular, it is important to distinguish between extracting a set of features non-linearly from a simple representation, and weighting a set of features linearly.

[1] We leave aside the case of a heuristic evaluation function, which has been completely specified by the user without any parameters to adjust, as there is nothing to learn. For the same reason, we also leave aside the case of a heuristic evaluation function with no representation of the game position, i.e. an empty set of features.

The above definition of complexity is quite vague, so to illustrate the point we will elaborate with an example. Similar to mathematics, the rules of any game can be described in terms of operations on symbolic entities, which we will call primitive variables. For example, each piece type in chess is a primitive variable and each square is a primitive variable. The status of a square, i.e. whether it is empty, contains a white pawn, a white bishop and so on, is an operation on two primitive variables, namely the piece type and the square, which returns either true or false. Observe that all elementary functional predicates are also operations which can be applied to these. We may then define the *complexity* of a single feature to be the degree of primitive variables and operations required to compute it, e.g. the minimum number of primitive variables it needed to carry out the computation, and the complexity of a feature set to be the sum of the complexities of each single feature. For example, by our previous definition, counting the number of White pieces on the board requires summing over all 64 squares and hence has a complexity of 64, assuming that there exists an operation to determine whether a square has a White piece on it or not. While counting the number of squares the White queen can move to requires finding the queen and then checking if any of the neighbouring squares are free and hence has a maximum complexity of $64 + 4 \times 7 = 92$ (for horizontal, vertical and two diagonal directions), assuming that there exists an operation to determine if a square contains the White queen and one to determine if a square is free. These definitions are, of course, arbitrary, but by inspecting the source code of the chess engines Meep and Stockfish we find that their features are computed very much along these lines (Romstad et al., 2014; Veness et al., 2009). As such, a set of hand-crafted features can be said to have a high complexity, if it compounds many primitive entities and operations as is often the case, and a set of simple variables, such as the positions of pieces, can be said to have a low complexity, if they can be derived from the primitive entries in very few operations. Despite the arbitrary choice of primitive variables and operations, such a working definition of complexity is still very useful. Being able to automatically extract complex features from simple features will allow humans with no expert knowledge to easily engineer features for new games once they are presented with the primitive entities and operations of the game, which we might assume happens as soon as they understand the rules of the game.

As we have already stated, we are primarily concerned with learning from low complexity feature sets. Nevertheless, previous work in the literature related to learning from both low

and high complexity feature sets are of interest to us, because they are essentially solving the same problem at different levels of difficulty.

## 5.2 Learning Approaches

Furnkranz divides the solutions into three types: supervised learning, comparison training and reinforcement learning. In supervised training a set of recorded game positions and their corresponding estimated values are used for training. The values of game positions can either be estimated by the outcome of the game or by expert players according to how strong one player is compared to the other player. In comparison training a set of positions and their relative preferences are used for training. Given a collection of games played by master-level players, the board positions are typically ranked such that positions corresponding to the moves taken by the players are ranked above any other moves. Strictly speaking, both methods fall into the category of supervised learning in machine learning terminology, but it is important to make a distinction between them. Lastly, in reinforcement learning the algorithm receives a stream of observed positions and their actual pay-off values. For example, in chess the pay-off is usually defined only at the end position, which means that all other positions have pay-off value of zero.

In our first experiments we are primarily concerned with supervised learning, but since the literature on learning heuristic evaluation functions for chess is far from enormous we shall also review reinforcement learning.

It has been argued that the main problem of supervised learning is to assign appropriate values to the game positions (Fürnkranz, 2001, 2007). In addition, Furnkranz points out that the supervised learning approach imposes the strong and unjustified assumption that moves played by the masters are actually the best moves. This is not always the case, and certainly there exists moves which are just as good in many cases. However, when the goal is only to use the heuristic evaulation function to play games, Tesauro argues that the primary purpose of the training instances in supervised learning is simply to guide the model parameters into good directions, since only the ranking not the exact values are used in play. In order to do this, it is only necessary to have approximate values (Tesauro and Sejnowski, 1989).

### 5.2.1 Learning in Othello

Lee and Mahajan applied a supervised learning method to learn a Bayesian discriminant function in the game Othello (Lee and Mahajan, 1988). Their Bayesian discriminant function was formally defined as:

$$g_c(\mathbf{x}) = \log P(\mathbf{x}|c) + \log P(c), \tag{5.1}$$

$$P(c|\mathbf{x}) = \frac{e^{g_c(\mathbf{x})}}{\sum_{c'} e^{g_{c'}(\mathbf{x})}}, \tag{5.2}$$

where $P(\mathbf{x}|c)$ is a multivariate normal distribution with mean $\mu_c$ and covariance $\Sigma_c$, i.e. the features are assumed to be normally distributed given the outcome of the game, and $P(c)$ is the prior probability of outcome $c \in \{1, \ldots, K\}$. The method differs from logistic regression (LogReg) since it may find a non-linear decision boundary, due to the second-order term in the normal distribution. Reasons for choosing this function included its non-linear decision boundary, concavity of the log-likelihood and its ability to deal with redundant features [2] (Lee and Mahajan, 1988, p. 20). Another interpretation of the model is as a LogReg model over the features $x_1, \ldots,$ second-order terms $x_1x_2, x_1x_3, \ldots$ and a constant term. Lee and Mahajan used a strong program named BILL to simulate a series of games. The first 20 plies were played out randomly and the game was then played to the end by letting the program play against itself. The features for each game position were identical to the features used by BILL and, together with the game outcome, were then used to train the classifier. When the classifier was implemented into a modified version of BILL it outperformed the original BILL program.

### 5.2.2 Learning in backgammon

Tesauro and Sejnowski trained MLPs (multilayer perceptron networks) to be the heuristic evaluation functions for a backgammon program based on several thousand expert-rated training positions (Tesauro and Sejnowski, 1989). They used a variety of sources to obtain these, and in particular included both a good and a bad move for each position. The bad moves were

[2] It is not entirely clear to the author how redundant features are handled by the model, but suppose that two features $x_1$ and $x_2$ are positively correlated. Then their covariance will be high, and thus the probability mass in the orthogonal direction, i.e. $-x_1x_2$, will be very small. If the correlation is high enough, then this will effectively reduce the dimensionality of the covariance matrix and mean vector.

included to help guide the backpropgatation algorithm into good directions. Their supervised learning setup differs from ours though, because their MLP took as input both the current position and the future (one-ply ahead) position where our models shall only take into account one position. Their argument for doing this was that the value of moving to a new position is relative to the former position, in particular if the values of positions are to be assigned by human experts as in their case. On the other hand, in our experiments we will take the game outcome alone as an indication of the position value. We will not consider the previous position in giving the current position its value, as this does not appear to be necessary in heuristic evaluation functions used in many master-level chess engines (Campbell et al., 2002; Russell and Norvig, 2009, pp. 171–173). Furthermore, it is uncertain whether such an approach will work once it is applied to the leaf-nodes of a tree generated by a heuristic search procedure. Their setup also differs from ours, as they designed some of their own positions to *teach* the MLP certain concepts of the game. They argue that designing positions is essential for the model to learn certain preferences, because some situations come up highly infrequently. We believe that this argument is debatable, and that it does not necessarily hold for chess. In any case, our goal is to minimize the amount of expert human knowledge, hence we will refrain from designing any such training examples.

Tesauro and Sejnowski also carried out similar experiments with ConvNet (convolutional neural networks), but found that these did not perform as well as MLPs. This does not necessarily apply to chess, but for the sake of completeness we shall carry out experiments with both fully-connected and locally-connected layers in our models.

As input features, Tesauro and Sejnowski experimented with several representations. In one they used a unary encoding scheme for the Backgammon board. Each of the twenty-four points (triangles) on the board was assigned five variables for each player, each of these take value one if there was respectively one, two, three, four or five or more men (checkers) on it and zero otherwise. In particular, the second variable would always take the value one if there was more than two men on, and the fifth variable would take the value of the number of men in addition to four if there were more than four men on it. This yielded ten different variables for each point. Since their representation included both the current board position and the future board position, each point would have a total of twenty variables which yielded $20 \times 24 = 480$ features.

They noted that this representation did not work well and experimented with a second representation, which instead of having the same encoding for the current position and the future position, assigned the variables in the next position according to the relative difference in number of men on each point. This representation worked better, but was still insufficient. Therefore they chose to add an additional nine simple features, which aggregated information across the entire board. Their reasoning was as follows: *"With only "raw" board information, the order of the desired computation ... is undoubtedly quite high, since the computation of relevant features requires information to be integrated from over most of the board area"* (Tesauro and Sejnowski, 1989).

Their MLPs contained three and four hidden layers and were trained with stochastic gradient descent (backpropagation) over a single training instance at a time with constant learning rate and momentum. Unlabelled board positions were included at random with scores assigned at random. This was done to ensure that the model was exposed to a sufficiently wide variety of positions. Perhaps this could be justified by the stochasticity of the dice rolls in Backgammon: in one board position taking a particular move may benefit the player because of a certain dice outcome in a later turn, but if there the dice eyes had shown different numbers another move should have been preferred even if no such training example exists in the dataset.

Tesauro and Sejnowski observed that more hidden units improved performance, though also resulted in overfitting when trained too long, and that additional hidden layers also improved performance. Their best model was able to win against the intermediate-level Backgammon program Gammontool nearly 60% of the time. Based on a qualitative analysis, they were able to conclude that the MLPs had learned important intuitive concepts of Backgammon.

In another line of work, Tesauro proposed to use comparison training for learning the heuristic evaluation function from a game collection (Tesauro, 1989). The game collection consisted of a set of board positions and preferred moves for each. He used a somewhat simpler feature representation compared to the one described above, but provided two future (one-ply ahead) board positions as input to the MLPs (as opposed to the current and future board position). He further applied a simple symmetry constraint on the model parameters, to ensure that they produced a consistent ranking and that swapping the two board positions would produce the same result. For each pair of board positions, he trained the models with a binary target to choose the board position selected by the expert, e.g. a zero if the first position is preferred

and a one if the second is preferred. This outperformed the supervised learning approach taken in (Tesauro and Sejnowski, 1989) with many fewer hidden units. Tesauro continued work on Backgammon and deviced the famous Backgammon program TD-Gammon based on TD($\lambda$) learning. See (Tesauro, 1995) for further details.

### 5.2.3 Learning in Connect-4

Mandziuk implemented a ConvNet (without max-pooling) to learn a heuristic evaluation function for the game Connect-4 (Mańdziuk, 2012). As input he used a local feature vector consisting of four consecutive squares horizontally, vertically and diagonally. As output he defined 7 units at the top layer. The move chosen by the heuristic function was defined to be the node corresponding to the unit with the highest value. The model was trained on game positions and their corresponding optimal move. The resulting model was able to pick the optimal move (without any heuristic search) in 58.62% percent of the positions presented to it.

### 5.2.4 Learning in chess

A variety of supervised learning models have been applied to learn chess concepts from databases. Early work in this area consisted primarily in weighting expert hand-crafted features to evaluate endgame positions, as researchers had observed that it was impossible to generalize from simpler features: *"If the [board] positions are only represented by obvious attributes like the location of the pieces and the side to move, the programs are not capable of making useful generalizations."* (Fürnkranz, 1997). See (Fürnkranz, 1997) for an overview on the early work. As the literature review unfolds, it should be clear that almost all approaches based on primitive features have failed to produce a master-level player. This is our motivation from a historical perspective, and justifies why we need to adopt the new approach given in section 4.2 to learn non-linear features.

Perhaps one of the earliest applications of learning an evaluation function for chess from scratch with MLPs was in the experiments undertaken by Gherrity (Gherrity, 1993). Gherrity developed a general game playing program named SAL, which used MLPs trained with TD(0) learning and gradient descent (backpropagation). As features for chess he used a bitboard representation to encode the positions of the pieces. That is, for each player and each piece

type there is a set of features corresponding to a grid over the board where each square takes value one if the piece is on that square and otherwise zero (Gherrity, 1993, p. 69). Gherrity also used a number of other features for his MLPs. In particular, Gherrity used a binary representation to encode pieces under attack, i.e. a grid for each piece type where each square takes value one if the piece on it is under attack and otherwise zero, and a representation of potential movements, i.e. a grid for each piece type where each square takes value one if the piece can move to that square and otherwise zero. His program was trained against the expert program GNU Chess, which was customized to have an ELO rating at 1,500-1,600 [3], on 4,200 games, but played very poorly in comparison. It only managed to beat GNU Chess a few times due to a bug.

Thrun developed a similar program two years later (Thrun, 1995). He used a set of hand-crafted features, and based on these trained two separate MLPs. The first MLP learned the changes in board position: how each feature value increased or decreased between the current board position and the board position following two consecutive moves, i.e. it was learning to predict the changes in feature values. The second MLP used a TD(0) algorithm to learn the value, e.g. probability of winning, from a given position. The TD-target for each board position was taken as the true value and backpropagation was used to adjust the parameters of the MLP. In addition to this, the parameters were further adjusted to agree with the gradient of the first MLP using the TangentProp technique (Simard et al., 1992). Thrun used a collection of 120,000 games played by grandmasters. Late positions in the collection were chosen at random and simulated to the end by letting the program play against itself, and the MLP was trained on these games. Thrun found this simulation approach to be of major importance in increasing performance, which he argued was because it allowed the program to learn the most relevant features with less training instances. In addition to this, the MLP was only trained on quiescent positions. Despite having a huge game collection available, only between 100 and 2,400 games were used for training in total.

Thrun played his program against GNU Chess, where he fixed the search depth to two and four ply respectively, and found that it was able to beat GNU Chess in some of the games. When both programs were fixed to a search depth at two ply, his program only managed to beat

[3]This corresponds to the strength of an average tournament chess player according to (Gherrity, 1993, p. 93).

GNU Chess in 13.17% of the games. Thrun argues that its low performance can be explained by two factors. Firstly, the training time was limited due to computational resources. Secondly, with each step of TD learning information is lost. Thrun suggested that in a domain as complex as chess, it may be necessary to have tremendous amount of examples available to learn certain situations in a supervised way such as, for example, a knight fork [4] (Thrun, 1995).

Similar to Thrun, in his Masters thesis, Mannen tried to train MLPs through TD($\lambda$) learning (Mannen, 2003) to become the heuristic evaluation function. In the experiments he trained the models on several feature sets, including 1) complex hand-crafted features, 2) binary representations of kings and pawns, and 3) binary representations of all piece types (Mannen, 2003, pp. 55–56). The hand-crafted features were included as input in all the MLPs. The models were trained on 50,000 games, played by expert players, in a single epoch. The resulting models were then played against each other and against a rather weak open-source program tscp 1.81 in a tournament setting. The results indicated that the binary representations of both kings and pawns and of all piece types improved the playing strength. The same experiment is discussed in (Wiering et al., 2005).

Similar in spirit to our work, Levinson et al. attempted to learn a heuristic evaluation function without the use of any expert knowledge (Levinson and Weber, 2001). They argued that a *naive* representation, such as the Bitboard Feature set we propose later, would be unable to generalize across positions and require far too much training to be feasible in practice. Therefore, for each of the 64 board squares, they constructed a vector with entries corresponding to the $3 \times 3$ neighbourhood and 8 possible squares a knight can move to. They transformed this vector to products of all possible 2-tuples and 3-tuples between variables, and used it as input to train a modification of a multilayer regression neural network [5]. Each unique 2-tuple or 3-tuple combination of pieces was assigned its own parameter. Symmetry was enforced by averaging model parameters over ranks, files, diagonals and knight movement symmetries during training. Levinson et al. learned a set of initial parameters based on training the model with TD($\lambda$) learning on both a collection of games and based on self-play. The parameters found corresponding to the piece values matched the traditional values assigned by chess mas-

[4] A knight fork is a situation where the knight of one player attacks two of the opponent pieces of higher value simultaneously.

[5] A multilayer regression neural network can be defined as an MLP where the top-layer is replaced by a linearly weighted sum of the hidden units in the previous layer instead of a logistic function

ters, which suggests that their local representation had managed to capture some of the game properties. They played the model based on 2-tuples against the model based on 3-tuples, both with a 2-ply search, and found that the 3-tuple model performed slightly better. This further suggests that higher-order features are relevant to improving play, but as they remark also require longer training time. They played the 2-tuple model against humans on the Internet Chess Server, but their model appeared to only reach a weak beginner level.

Compared to both Gherrity and Thrun we are training on a magnitude of 100 times more training instances, and compared to Mannen and Levinson on a magnitude of 10 times more training instances. Contrary to previous work, we are repeating our experiments over multiple runs to verify that our models are able to consistently reproduce the obtained results. The reader should however observe that our training procedures are markedly different.

The comparison training approach has also been suggested and explored to some extent for chess in (Tesauro, 2001). Tesauro adapted the original comparison training approach to work in conjunction with an alpha-beta search procedure and used it learn a subset of the parameters for the chess engine Deep Blue. Qualitative study showed that this improved its playing strength in some games.

## 5.3 Discussion

The approach taken by Tesauro and others has been criticized, because their models are not learning a hierarchical feature representation (Utgoff, 2001). Utgoff speculated that the reason why researchers in the game playing field still hand-craft their own features was due in large part because their models had been too shallow to learn the advanced concepts, which are necessary to play the at a high level. The idea to learn hierarchical feature representations is also advocated by Mandziuk, who further related this to the nature of human game playing (Mańdziuk, 2012). While there are many other ways of exploring feature learning, it is these arguments that encourage us to focus on hierarchical models and hence deep learning.

Utgoff has also pointed out that previous models took only the outcome or preference as a training signal (target), while the knowledge of the game rules was taken to be implicit. He illustrated that this is different from human play, because if a human player made an illegal move the opponent would correct it. This leads to the proposition of also training the models to

recognize legal from non-legal moves. The same idea of learning from multiple problems posed by the same game is proposed by Mandziuk (Mańdziuk, 2012, p. 3). For example, a model could be trained to classify board positions according to whether they contain a check or not. Evidently, after training such models would have constructed many features which would also be useful in learning the heuristic evaluation function. This is an interesting research direction, but unfortunately outside our scope.

# Chapter 6

# Supervised Experiments on Game Collection

## 6.1 Setup

In our first set of experiments we are concerned with the supervised learning of the heuristic evaluation function from recorded games. We take each board position to be a training instance with the game outcome as its class, i.e. White wins, draw, Black wins. Thus, the problem our models are solving is, given a board position, predict the outcome of the game. We prefer this approach for the game collection over comparison and reinforcement learning discussed earlier, because of its simplicity, and because we will later continue work in the direction of (Veness et al., 2009).

Researchers have argued that it is difficult to infer a heuristic evaluation function from a recorded set of games, because each board position may not necessarily reflect the outcome of the game under optimal play. For example, it might be that both players make mistakes during the game, and therefore that the some board positions favour one player while others favour the other player in the same game. This is a part of the credit assignment problem: to assign an approximate expected pay-off value to a board position based only on the pay-off at the end of the game (Gherrity, 1993, p. 17). Veness, a former chess programmer, estimates from personal observation that even chess grandmasters make several blunders during each game,

often in the number of three blunders per game around depths of 4-7 ply (Veness, 2014b). However, many of the blunders are quite subtle and often not disastrous when playing against another human opponent. Furnkranz also considers this problem, which he terms *noise*. He observes that *"[noise] appear[s] quite frequently in chess databases. In particular with respect to the evaluation of a certain position or move, where there is usually contradicting evidence from different commentators or from the various outcomes of games that continued from the given position."* (Fürnkranz, 1997).

Researchers have suggested that the problem can be simplified by performing supervised learning only on the quiescent positions in the game collection (Thrun, 1995; Wiering et al., 2005, p. 3; Veness, 2014b). This would make the problem easier, because the model would not need to evaluate volatile positions [1]. However, to both limit the amount of expert knowledge used and simplify our problem, we choose not to apply this method. Instead, we hope that our models will separate quiescent positions from non-quiescent automatically by, for example, assigning higher uncertainty to non-quiescent positions.

Chess programmers have proposed to use only a single board position from each game in the collection to ensure that the training samples are close to independent (Ban, 2012). The position should be chosen at random and should not include any of the first or last positions of the game, as these are believed to be either irrelevant or too trivial to classify. However, we do not consider co-dependence to be a major problem because we are either shuffling our data samples or using huge mini-batches ($\sim$ 20,000 positions) in training all our models. As we will discuss later, the purpose of the supervised learning experiments is to verify the representational power of our models. This necessarily includes opening positions, e.g. discriminating between weak and strong opening lines, and trivial end positions, e.g. identifying the winning player based on simple material counts.

## 6.2 Feature Representations

In our experiments we will work with several feature sets. All feature sets, will contain the binary feature *side-to-move*, which equals one when it is White to play and otherwise zero. It is

[1] For example, board positions before and after a queen trade.

arguable whether such a feature benefits the classification in a linear model (Veness, 2014b) [2], but the side to move may certainly be of importance in our hierarchical models. The multilayer perceptron networks (MLPs) and convolutional neural networks (ConvNets) migt be able to deduce features depending on the player to move. To keep our models consistent, we choose to include the side-to-move feature in all feature sets.

### 6.2.1 Stockfish chess engine

One of our benchmarks will be the heuristic evaluation function of the Stockfish 5.0 engine. This is a strong master-level chess engine, which is believed to have one of the best heuristic evaluation functions available (Romstad et al., 2014; Veness, 2014b). For each board position, we will use the output of its heuristic evaluation function (a scalar) together with the side-to-move feature. We call this feature set for **Stockfish's Static Features** [3].

As our second benchmark, we will use the heuristic evaluation function applied to quiescent positions found by a quiescent search after a heuristic search, as discussed in section 2.3. We set the maximum search depth to 14 ply [4]. This should put the Stockfish engine above 2,000 ELO [5]. The heuristic evaluation function together with the side-to-move feature will be the feature set called **Stockfish's QSearch Features**.

### 6.2.2 Meep chess engine

Since we intend to extend the chess engine Meep, developed by in (Veness et al., 2009), we shall take its heuristic evaluation function as a second benchmark. The engine has 3,624 features in

[2]It seems likely that there is some advantage to having the move turn, but this could in principle be negligible. However, we do observe that White wins more often than Black in our game collection, which implies that at least the evaluation at the first move (and probably the entire opening line) does depend on the side to move.

[3]In fact, the side-to-move feature is already included in the heuristic evaluation function used by Stockfish. Nevertheless, to keep the feature representation identical to others, we shall include it explicitly in the logistic regression model.

[4]This is a hard limit, which restricts the standard minimax search.

[5]See `http://people.csail.mit.edu/heinz/dt/node49.html`. From inspecting Stockfish source code the depth of the quiescent search depends on the preceding heuristic search, but is otherwise deterministic. Veness states that the quiescent searches is typically in the order of 5-6 ply (Veness, 2014b). Additionally, ELO ratings of Stockfish played against other chess engines can be found at `http://www.computerchess.org.uk/ccrl/404/rating_list_all.html`

total (1,812 for each player), which are weighted linearly to produce the heuristic evaluation value. These were derived from a subset of the features used in an earlier tournament chess engine, named Bodo, which had a master-level rating at an ELO 2,500-2,600 on the Internet Chess Club server (Veness, 2014a).

We shall use all features in Meep. The majority of features are binary and also sparse, i.e. only a small number of them are non-zero for any given board position. Veness et. al note that they are *"simplistic compared to a typical tournament engine."* (Veness et al., 2009). We have inspected the source-code for Meep and found discrete features including material counts for each piece, isolated and doubled pawns [6], the mobility of each piece [7] and king safety. The Meep features with the side-to-move feature will be called **Meep Features**.

### 6.2.3 Bitboard representation

Our first step in the direction of using less expert knowledge will be to create the simplest imaginable feature set, which is almost unique for every board position [8], as first suggested to the author by Silver (Silver, 2014b). This representation was also suggested by (Beal and Smith, 2000) and used by (Gherrity, 1993). For each player and each piece type, we construct a binary vector corresponding to an $8 \times 8$ grid, where each entry takes value one if the piece resides on the corresponding square and zero otherwise. We concatenate these to form a vector of $12 \times 8 \times 8 = 768$ features. In principle, pawns can never be situated on the first and eight rank. We shall keep this in mind, but for simplicity ignore it in our further discussions. See figure 6.1. We will call these features with the side-to-move feature for **Bitboard Features**.

[6] An isolated pawn is an unprotected pawn. Doubled pawns are two pawns positioned on the same file.

[7] The mobility of each piece is often defined as the number of squares it is able to move to.

[8] The representation is unique with the exception of castling, rule for draw after three repetitive moves and rule for draw after fifty moves without any captured pieces.

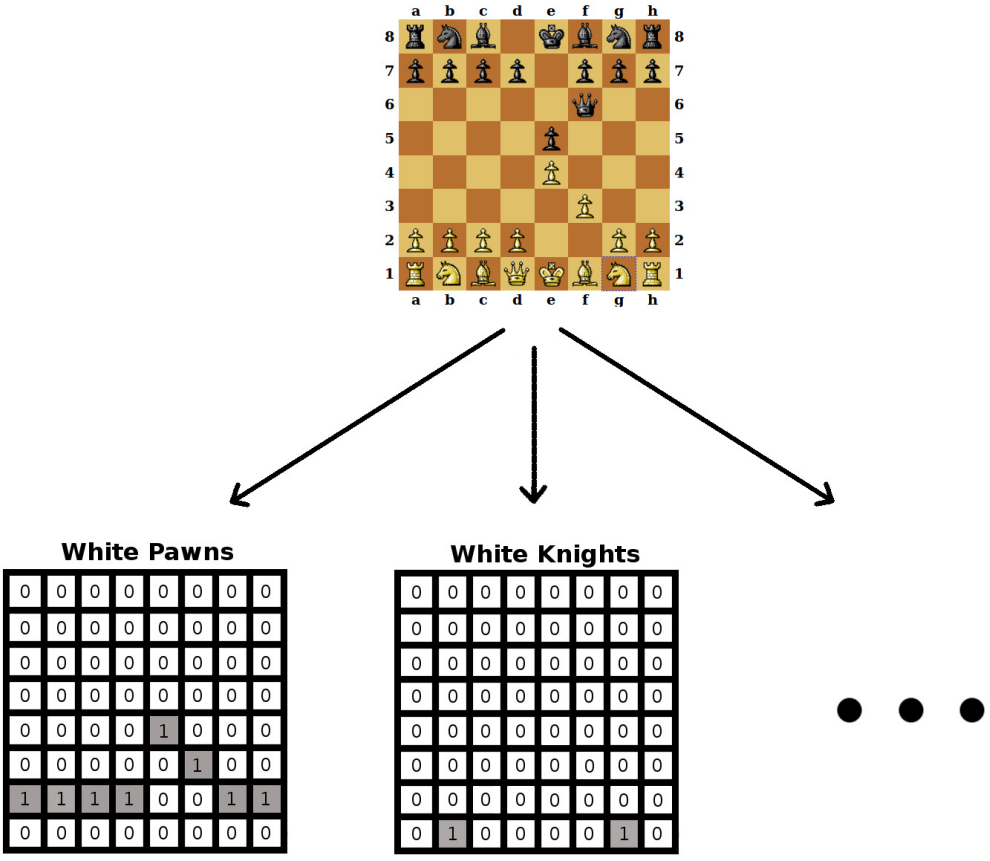



Figure 6.1: For each player and each piece type, we construct a binary vector corresponding to an $8 \times 8$ grid, where each entry takes value one if the piece resides on the corresponding square and zero otherwise. We concatenate these to a vector of $12 \times 8 \times 8 = 768$ features.

There are six piece types for each player, which yields $12^{64} \approx 10^{69}$ combinations. Thus, the representation is extremely sparse and the it arguable how well the model will be able to generalize from it. Observe further that, since the representation is binary, it is already normalized w.r.t. a maximum absolute value of one.

A similar representation was used by Krafft for training a deep belief network to learn a distribution over game positions in $9 \times 9$ Go (Krafft, 2010). Krafft obtained partial success in his attempt, but noted that the model was not able to generalize positions occurring in one area of the board to other areas of the board. Although the chess board contains 64 squares compared to 81 squares in $9 \times 9$ Go, we should expect to obtain a somewhat similar result with this

representation, if we do not enforce translation invariance, as for example in ConvNets (Silver, 2014b).

In retrospect, the author also discovered the work by Mayer where three different feature representations were evaluated for Go (Mayer, 2007). An MLP was trained for each representation with TD. The feature representation, which Mayer named *Koten*, where each square takes value one if a man (piece) is on it and zero otherwise, quite similar to our $8 \times 8$ binary grid for each piece type, performed the worst. Therefore it is clear that there is plenty of room to improve the feature set, while still using minimal expert knowledge.

## 6.3 Evaluation

In general, it is doubtful whether the accuracy of our models is correlated with how well they will perform when used as a heuristic evaluation function. As Dayan has argued, not all classifications are equally important (Dayan, 2014). This is because the heuristic evaluation function is used in conjunction with a search procedure, where it is only applied in certain phases of the game to quiescent positions. For example, mistakes at the beginning of the game may be disastrous and therefore require a highly accurate ranking. Meanwhile, end game positions can be evaluated correctly by either search or lookup tables, which means that the heuristic evaluation function is never required to evaluate these. Yet, in our experiments we apply the models to all positions in the game collection with equal weight. In addition to this, the performance of the heuristic evaluation function also depends on the particular search depth employed. Furnkranz observes this: *"the importance of being able to recognize tactical patterns such as fork threats may decrease with the program's search depth or depend on the efficiency of the program's quiescence search."* (Fürnkranz, 2001, p. 19). Nevertheless, chess programmers have observed that the performance of a heuristic evaluation function on a game collection tends to be strongly correlated with the programs playing strength, at least when the features are hand-crafted by experts (Ban, 2012) [9].

We therefore only interpret the accuracy of our models as a measure of how effectively they

[9]In particular, the geometric mean over the likelihoods of each board position has been suggested as a quantity highly correlated with the performance of the game in (Ban, 2012). Although, we should note that the model proposed by Ban is rather heuristic from a probabilistic point of view. In particular, it includes a *drawishness factor*, which may have affected his results significantly.

are capturing important features of chess. A model with a high accuracy must, at least, be able to extract or weight features which are good indicators for determining the winning player given that both players are strong players. Hence, our assumption is that a model able to capture important features for determining the winning player also has the representational power to capture important features for ranking board positions in conjunction with a search procedure, if the model is trained for this particular purpose. This implies that we can use our supervised learning experiments to validate the representational power of our models, while their actual performance can be estimated only by implementing them into a heuristic evaluation function and observing the playing strength of the program.

We will evaluate our models against the feature sets constructed based on the Stockfish and Meep chess engines. These features should weight important aspects of each board position, since both engines have achieved master-level play. However, the features were constructed to rank positions and not to approximate the expected pay-off. Thus, applying any increasing function to transform the heuristic evaluation function will not change the ranking of the board positions, yet such a transformation may change our models trained on these features tremendously. For example, take a LogReg model with a single scalar feature as input. Suppose all scalar values above some constant are multiplied by a large positive constant, and all values below remain the same. This would not change the board rankings for a fixed set of parameters, but after training the linear decision boundary, found by the LogReg model w.r.t. log-likelihood, would be moved towards a boundary optimal for these training instances with large points only. Furthermore, the Stockfish and Meep feature sets were constructed for evaluating quiescent positions, while in our experiments they are used to evaluate all board positions. Therefore, the accuracy of our models based on features derived from Stockfish and Meep should be interpreted as a conservative estimate of their representational power. This is in particular true for the Stockfish feature sets, since these consist of only two scalar features which the training procedure can optimize with respect to,

To conclude that our new models have a sufficient representational power to reach master-level play strength, we should therefore aim to achieve an accuracy on par or higher than the models based on Stockfish and Meep.

## 6.4 Data Collection

We use the FICS Database [10] (Seberg and Ludens, 2014a). The collection contains games played online predominantly in 2013 and 2014.

In our experiments, we choose to only use games where both players have an ELO rating above 2,000. We expect strong players to make fewer blunders, which will make the classification problem less stochastic. If players play to the best of their strength without serious blunders, then the board position at any point in the game should always favour the stronger player. It would be less likely that the weaker player got the upper hand at any point. If, on the other hand, the game contained many blunders by both players, there would be many positions during the game where the losing player would have a favourable position and this would lead to a more stochastic classification problem. The same problem was also recognized by Lee and Mahajan (Lee and Mahajan, 1988, p. 13) in their experiments. Ban, the author of the strong chess engine Deep Junior, also advocates to adjust the parameters of the heuristic evaluation function based on games played between strong players of preferably the same level (Ban, 2012).

The games were of varying length, as shown in the table below (Seberg and Ludens, 2014b):

| Game type | Description | Count |
|---|---|---|
| Standard games | Expected duration of game is more than 15 min. per player | 9,388 |
| Blitz games | Expected duration of game is 3 and 15 min. per player | 19,107 |
| Lightning games | Expected duration of game is less than 3 min. per player | 15,063 |

## 6.5 Preprocessing

We subtracted the mean of all the input features [11]. This was found to perform better than not subtracting the mean in initial experiments. We did not scale any features to have variance one [12], but noted instead that the features used in our hierarchical models, the Bitboard Features, had values which were already in the interval $[-1, 1]$.

[10] Free Internet Chess Server; An online server and software which allows people to play chess against each other for free.

[11] The mean was not subtracted in the experiments involving the logistic regression models as the intercept should be able handle any offsets in the data.

[12] For the LogReg model taking Stockfish quiscience feature as input we multiplied all values by $10^{-3}$ to avoid arithmetic overflow.

As discussed in subsection 3.2.3 it is often preferred to decorrelate the inputs. We applied the PCA transformation to a small subset of the training data and retrained MLP on bitboard inputs with various hyperparameters, but found that these did not improve performance. For this reason we discarded the idea of decorrelating the inputs and so did not consider other methods, such as probabilistic principal components analysis (PPCA) or factor analysis (FA).

## 6.6 Rollout Simulated Games

Our first set of initial experiments used only standard games, i.e. games where each player is given 15 minutes or more in total to play. We expected these games to have even fewer blunders than the other games, which would lead to an even less stochastic classification problem. Based on these initial experiments, we found that the dataset was too small. We therefore chose to extend the dataset to include games with other time controls, which we hoped would make the dataset sufficiently large for effective learning. We note, however, that this also has the potential of making the classification problem more difficult as a position could be judged differently depending on the available thinking time. For example, take a position where one player is forked. Under the lightning time-control the defending player might be taken to have lost the game, but under a longer time-control setting the player might find a solution to balance out the material loss incurred by the fork and the game should therefore be classified as a draw.

After manually inspecting several games in the dataset, we found that many games were often terminated by one of the players resigning early in the game. This would sometimes happen at a point where the resigning player had made a serious blunder, which we speculate may have triggered them to give up. Sometimes this would happen without any apparent blunders, which we speculate may have been due entirely to external circumstances (not everyone can afford the luxury of playing chess all day long). We therefore supposed that there were many games from which no winner could be assigned.

To resolve this issue we applied rollout simulation. For every game in the collection, which did not end effectively in either checkmate or draw, we simulated further moves from each player with the Stockfish engine (Romstad et al., 2014) playing both sides until either one side won or the game ended in a draw. We configured the strength of the Stockfish engine by setting its maximum search depth to 14 ply, which as noted before should put its playing strength above

2,000 ELO rating [13]. No opening book was used. The outcome of the game was taken to be the result from the simulation, and the additional board positions of the simulated moves were appended to the collection [14].

The rollout simulations neglect the issue of games where one of the players loses due to time-out. Therefore, we assume that at this level of play players rarely forfeit due to time-out. In any case, our supervised experiments are primarily concerned with the evaluation of board positions, regardless of the actual time it took the players to execute their moves [15].

The rollout simulations increase the number of board positions considerably.

**Rollout simulated games overview**

| | Board positions |
|---|---|
| Non-simulated games | 3,405,201 |
| Rollout simulated games | 5,478,860 |

Table 6.1: Training instance statistics for non-simulated and rollout simulated games.

To verify the effects of simulating games, we carried out the following three experiments with LogReg based on Meep Features (Veness et al., 2009). The first experiment was to train the model on simulated games and test it on simulated games, the second was to test the same model on non-simulated games and the third to both train and test the model on non-simulated games. By comparing the performance of these models we hoped to gain insight into how the rollout simulations change the original classification problem. All models were trained with stochastic gradient descent over mini-batches, as described in algorithm 3 where we removed the momentum by setting $\beta = 1$.

[13] The *skill level* was set to maximum, as Stockfish is known to play in its own (perhaps non-human) style if this is decreased.

[14] As a safety hatch, and to reduce computing time, if after simulating 100 moves for each player the game did not finish the rollout simulation would be aborted and the original game with its original outcome (determined by the resigning or timed-out player) would be used as before. This happened only in a very small fraction of games.

[15] One can also argue that simulating time-out games to the end is equivalent to adding a time bonus to the player who forfeited due to time-out, given that the moves both players would take after the time-out are the same as those Stockfish take. A similar view, is that the game was instead played with no time-limit between two virtual players: the non-forfeiting player coupled with Stockfish after $n$ moves and the forfeiting player coupled with Stockfish after $n$ moves.

From preliminary experiments on a small subset of the training data, we found that $10^{-2}$ for the (non-decaying) learning rate 5 for the number of updates per batch and 5 for the number of epochs worked well. Increasing the number of epochs seemed to improve the results very slightly, but due to its additional computational cost versus gain in performance we decided to stay at 5 epochs. We then ran a hyperparameter optimization on the entire training set w.r.t. the (non-decaying) learning rates $\{10^{-1}, 10^{-1.5}, 10^{-2}, 10^{-2.5}, 10^{-3}\}$ for updates per batch and number of epochs fixed at 5 each. This is in agreement with the experimental evidence by Bengio, who suggests that it is necessary to be within a factor of two from the optimal learning rate (Bengio, 2012, p. 5). The tables below show the results in the form of overall mean accuracies [16], mean accuracies as a function of move numbers and contingency tables [17] for each model.

**LogReg trained on Meep Features**

| **Model** | **Mean accuracy** |
|---|---|
| Trained and tested on simulated games | 62.45% |
| Trained on simulated games and tested on non-simulated games | 53.39% |
| Trained and tested on non-simulated games | 53.92% |

Table 6.2: Mean accuracies for LogReg models, with Meep Features, trained and tested on both simulated and non-simulated games.

[16]The mean accuracy is the accuracy averaged over all games and board positions, where each board position counts as a single entry.

[17]A contingency table is also called a confusion matrix in machine learning.

**LogReg with Meep Features on rollout simulated games**

Mean accuracy: 62.45%

| **True outcome** \ **Predicted outcome** | Black wins | Draw | White wins |
|---|---|---|---|
| Black wins | 47,328 | 2,031 | 17,055 |
| Draw | 15,525 | 8,507 | 15,170 |
| White wins | 16,456 | 1,774 | 57,295 |

Table 6.3: LogReg model, with Meep Features, trained and tested on rollout simulated games. The table shows the mean accuracy and contingency table of the model. Model was trained on 683,932 instances (board positions) and tested on 181,141 instances. In comparison, predicting with the most frequent outcome on the entire dataset yields 40.95% accuracy.

It is clear that the model trained and tested on simulated games performed the best with a mean accuracy at 62.45%. Our earlier assumption appears therefore to be correct: by applying simulations and substituting the outcome with the simulated outcome, it has become easier to predict the outcome of each game from a single board position. Furthermore, we also observe that the accuracy of the model trained on simulated games is on par with the model trained on non-simulated games (trained on a dataset of comparable size) when tested on a third dataset of non-simulated games. This indicates that training the model on simulated games will not hurt its accuracy at predicting non-simulated games.

The same conclusion can be drawn from the plot over mean accuracies as a function of move numbers, shown in figure 6.2, where training and testing on the simulated games lead to a more accurate prediction as we would expect. Similar accuracy results w.r.t. move numbers are reported by Lee and Mahajan for Othello (Lee and Mahajan, 1988, p. 20). Finally, the contingency table shows us that the model trained on simulated games is somewhat better at classifying drawn games correctly than the model trained on non-simulated games. This may be because features indicating a draw game are more evident during endgames than during middle game. We therefore hope that the simulated games will help our models to better learn the concept of a draw game.

How do these results compare to what we would expect from the an optimal classifier, i.e.

**LogReg with Meep Features trained on simulated games and tested on non-simulated games**

Mean accuracy: 53.39%

| **True outcome** \ **Predicted outcome** | Black wins | Draw | White wins |
|---|---|---|---|
| Black wins | 41,549 | 2,179 | 26,643 |
| Draw | 9,660 | 2,348 | 10,651 |
| White wins | 25,841 | 1,732 | 43,982 |

Table 6.4: LogReg model, with Meep Features, trained on rollout simulated games and tested on non-simulated games. The table shows the mean accuracy and contingency table of the model. Model was trained on 683,932 instances and tested on 164,585 instances. In comparison, predicting with the most frequent outcome on the entire dataset (rollout simulated training and non-simulated test set) yields 41.28% accuracy.

the *Bayes optimal classifier*? In other words, what is the highest accuracy we should expect possible? As discussed in section 6.1, game collections often contain many blunders made by each player, which implies that we should not expect a very high accuracy for any model in general. In addition to this we should expect the accuracy to increase with move numbers, because there are fewer moves in which each player can make blunders. We did indeed observe this until around move 100, where the accuracy starts to decrease although it continues to be consistently above the accuracies derived from the non-simulated datasets.

## 6.7 Dataset Partitioning

Next, we split our complete rollout simulated dataset into three parts in proportions approximately $6 : 2 : 2$ for training set, validation set and test set. The split was carried out such that games were divided at random, but no single game was contained in more than one set.

**LogReg with Meep Features on non-simulated games**

Mean accuracy: 53.92%

| **True outcome** \ **Predicted outcome** | Black wins | Draw | White wins |
|---|---|---|---|
| Black wins | 46,158 | 462 | 23,751 |
| Draw | 10,418 | 1,929 | 10,312 |
| White wins | 30,490 | 411 | 40,654 |

Table 6.5: LogReg model, with Meep Features, trained and tested on non-simulated games. The table shows the mean accuracy and contingency table of the model. Model was trained on 681,390 instances and tested on 164,585 instances. In comparison, predicting with the most frequent outcome on the entire dataset yields 43.72% accuracy.

**Dataset partitioning**

| | Percentage | Board positions |
|---|---|---|
| Training set | 58.30% | 3,194,376 |
| Validation set | 20.15% | 1,104,013 |
| Test set | 21.55% | 1,180,471 |

Table 6.6: The dataset is partitioned into a training set, a validation set and a test set as shown in the table.

We believe that this partitioning will allow our models to learn sufficiently complex features, while still giving a reliable estimate of the model performance.

For reference, predicting naively with the most frequent outcome on the entire dataset yields 38.69% accuracy.

## 6.8 Benchmarks

To establish a benchmark for our models, we trained and evaluated a LogReg model taking different feature sets as input. We used the same hyperparameter optimization procedure as in our experiments comparing rollout simulated games with non-simulated games. That is, we

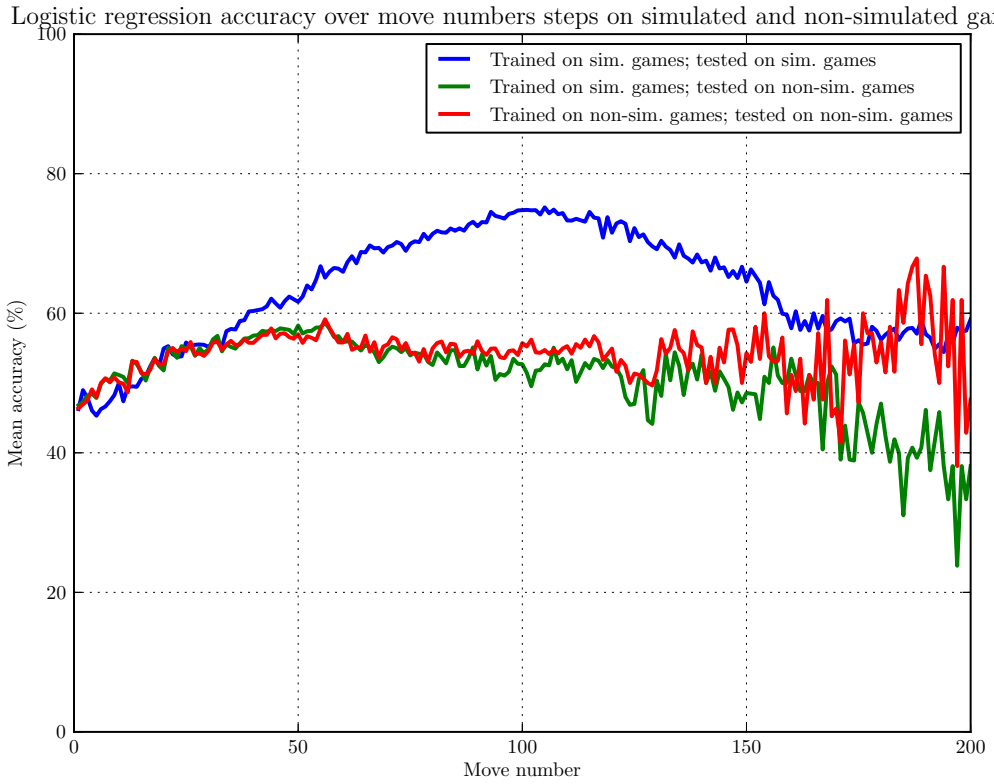



Figure 6.2: Accuracy for LogReg models, with Meep Features, as a function of move numbers, when 1) trained and tested on simulated games, 2) trained on simulated games and tested on non-simulated games, and 3) trained and tested on non-simulated games.

optimized the learning rates over the set $\{10^{-1}, 10^{-1.5}, 10^{-2}, 10^{-2.5}, 10^{-3}\}$ for updates per batch and number of epochs fixed at 5 each [18]. We trained the LogReg taking as input Bitboard Features, Meep Features, Bitboard Features + Meep Features, Stockfish's Static Features and Stockfish's QSearch Features. The tables below show the results in the form of overall mean accuracies and contingency tables for each model.

[18]For the LogReg with Meep Features we only used learning rates in the set $\{10^{-1}, 10^{-1.5}, 10^{-2}\}$, as initial experiments had already shown this was a suitable range. We also normalized the feature vectors to have mean zero.

**Summary over LogReg models**

| **Features** | **Mean accuracy** |
|---|---|
| Bitboard Features | 57.34% |
| Meep Features | 64.48% |
| Bitboard Features + Meep Features | 64.65% |
| Stockfish's Static Features | 58.28% |
| Stockfish's QSearch Features | **67.00%** |

Table 6.7: Mean accuracies for LogReg models taking as input Bitboard Features, Meep Features, Bitboard Features + Meep Features, Stockfish's Static Features and Stockfish's QSearch Features. In comparison, predicting with the most frequent outcome on the entire dataset yields 40.95% accuracy.

**LogReg with Bitboard Features**

Mean accuracy: 57.34%

| **True outcome** \ **Predicted outcome** | Black wins | Draw | White wins |
|---|---|---|---|
| Black wins | 239,039 | 66,970 | 115,503 |
| Draw | 75,816 | 147,249 | 93,782 |
| White wins | 90,004 | 61,528 | 290,580 |

Table 6.8: LogReg model with Bitboard Features. The table shows the mean accuracy and contingency table of the model. Model was trained on 4,298,389 instances and tested on 1,180,471 instances.

**LogReg with Meep Features**

Mean accuracy: 64.48%

| **True outcome** \ **Predicted outcome** | Black wins | Draw | White wins |
|---|---|---|---|
| Black wins | 290,995 | 45,211 | 85,306 |
| Draw | 74,920 | 161,061 | 80,866 |
| White wins | 95,121 | 37,827 | 309,164 |

Table 6.9: LogReg model with Meep Features. The table shows the mean accuracy and contingency table of the model. Model was trained on 4,298,389 instances and tested on 1,180,471 instances.

**LogReg with Meep Features and Bitboard Features**

Mean accuracy: 64.65%

| **True outcome** \ **Predicted outcome** | Black wins | Draw | White wins |
|---|---|---|---|
| Black wins | 291,104 | 45,287 | 85,121 |
| Draw | 75,034 | 162,633 | 79,180 |
| White wins | 93,880 | 38,830 | 309,402 |

Table 6.10: LogReg model with Bitboard Features + Meep Features. The table shows the mean accuracy and contingency table of the model. Model was trained on 4,298,389 instances and tested on 1,180,471 instances.

**LogReg with Stockfish's Static Features**

Mean accuracy: 58.28%

| **True outcome** \ **Predicted outcome** | Black wins | Draw | White wins |
|---|---|---|---|
| Black wins | 304,022 | 2,942 | 114,548 |
| Draw | 137,339 | 16,820 | 162,688 |
| White wins | 72,052 | 2,883 | 367,177 |

Table 6.11: LogReg model with Stockfish's Static Features. The table shows the mean accuracy and contingency table of the model. Model was trained on 4,298,389 instances and tested on 1,180,471 instances.

**LogReg with Stockfish's QSearch Features**

Mean accuracy: 67.00%

| **True outcome** \ **Predicted outcome** | Black wins | Draw | White wins |
|---|---|---|---|
| Black wins | 304,147 | 51,847 | 65,518 |
| Draw | 85,065 | 137,745 | 94,037 |
| White wins | 42,317 | 50,733 | 349,062 |

Table 6.12: LogReg model with Stockfish's QSearch Features. The table shows the mean accuracy and contingency table of the model. Model was trained on 4,298,389 instances and tested on 1,180,471 instances.

As shown in table 6.7, we find that Bitboard Features is outperformed by all other feature sets when only first-order terms are considered. This is surprisingly followed by Stockfish's Static Features, which we know can achieve master-level play when coupled with Stockfish's search procedure. From contingency table 6.12, we note that the model performs extremely rarely ever predicts draw games compared to all others. This may be caused by its heuristic evaluation function which, of course, was designed primarily for scoring winning positions accurately and therefore is very volatile around board positions containing draws. We also

speculate that its low accuracy is due to not being optimized for non-quiescent positions and positions during the beginning of the game, where Stockfish normally uses an opening book. This is also confirmed by the high accuracy of Stockfish's QSearch Features, which includes the same heuristic evaluation function but at future quiescent positions. Next, Meep Features and Bitboard Features + Meep Features both reach an accuracy close to 64.50%. This indicates that the features derived from Meep are far better than Bitboard Features, and that adding Bitboard Features makes little difference. We therefore conclude that Meep Features already contain all relevant information available in Bitboard Features when only considering first-order terms. We speculate that the Meep Features achieve a higher accuracy than Stockfish's Static Features, because the former has 3,625 parameters and the later only 2 parameters which can be optimized. Finally, as expected, Stockfish's QSearch Features reach the highest accuracy at 67.00%. However, again the contingency table 6.12 reveals that is markedly worse at predicting draw games than all other models. It will be our goal to reach a similar accuracy in our next experiments.

An interesting interpretation of LogReg with Bitboard Features is that the model is weighting the benefit, or advantage, of each piece on each square given the other squares are on their average position [19]. The logistic regression coefficients w.r.t. White wins illustrate this in figure 6.3, where several patterns can be observed. For White's knights and bishops, White clearly has an advantage when these pieces are in the center of the board, and conversely a disadvantage when Black has these pieces in the center. This pattern is no surprise, as controlling the board center is a standard chess tactic. In particular, the white knight only has a large advantage when positioned in the very center of the board, while the bishop still has an advantage in the outskirts of the center. This can be explained by the high mobility of the bishop. The white rook appears to be most powerful when it is at the 7th file, which also agrees with common chess knowledge since it is often an advantageous attack position. For White's king and queen, it is advantage when they are both near the back rank of Black, which would indicate that Black is in a defensive or undeveloped position. The pawns are more difficult to interpret, which we speculate is because their role is most dominant in end games, where grandmasters

[19]The reader should observe that this is different from weighting each piece on each square separately, e.g. by counting their number of occurrences with each game outcome. For example, if two binary features tends to be highly correlated the magnitude of their respective parameters will decrease. The interpretation of a parameter should therefore be along the following lines: given a piece on a particular square, what is the value of this piece given the typical positions of other pieces.

are able to foresee many moves ahead. Our results agree very much with the Bitboard Features found in (Beal and Smith, 2000, pp. 12–14), where they are referred to as piece-square tables. In addition to this, Veness has made similar observations in his previous work (Veness, 2014b). The reader should observe that these results are also consistent if L1-regularization is applied, as we demonstrate in appendix section D.1.

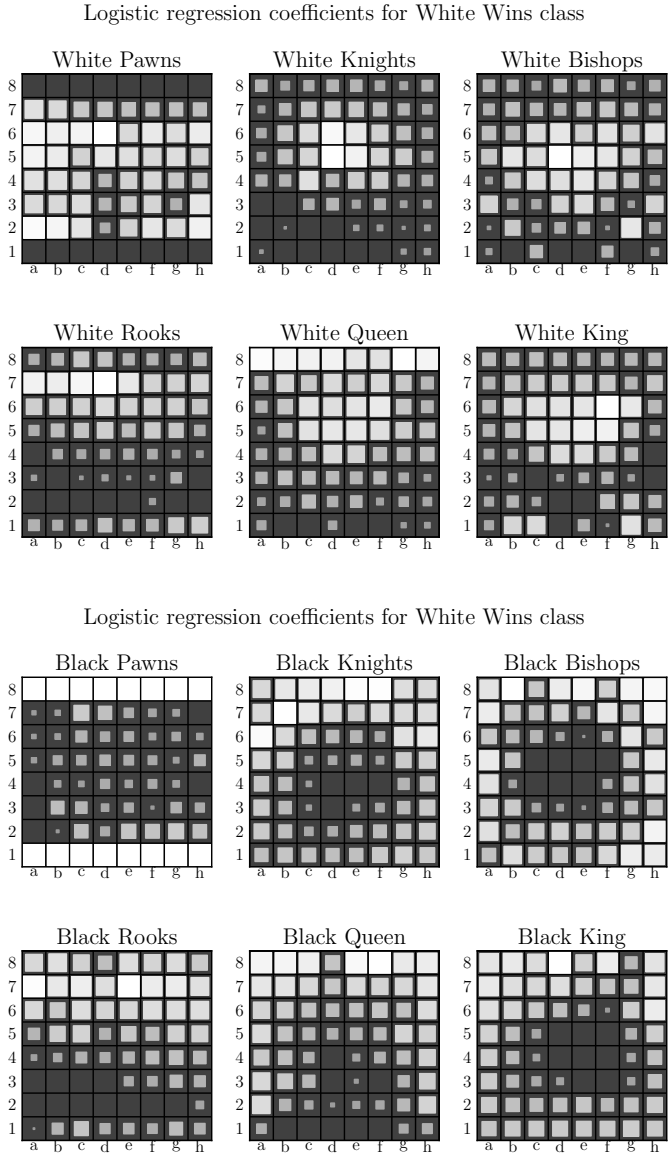



Figure 6.3: LogReg coefficients for White wins class. To make interpretation easier, the coefficients were transformed linearly to be in the interval [0, 1] for each piece type, and hence should only be compared within each piece type. Larger and whiter squares correspond to an increased probability of winning the game for White. The first and last rank for coefficients w.r.t. White pawns take values equal to zero, because pawns can never be located on these squares and because White pawns on all other square have an average value above zero. Similarly, the same coefficients w.r.t. Black pawns take value one, after the linear transformation, because pawns can never be located on these squares and because all other Black pawns have an average value below zero.

## 6.9 Results

### 6.9.1 One-Layer MLPs

For the one-layer MLP models, we ran a number of grid search experiments over the hyperparameters for $2, 4, \ldots, 20$ hidden units:

- mini-batch sizes in $\{25, 50, 75, 100, 200, 500, 1000, \sim 20000\}$,
- number of updates per mini-batch in $\{1, 5, 10\}$,
- beta momentums in $\{0.2, 0.4, \ldots, 0.8, 1.0\}$,
- lambda decay rates in $\{0, 1, 10\}$,
- initial learning rates in $\{10^{-0.5}, 10^{-1}, \ldots, 10^{-2.5}, 10^{-3}\}$.

We found that a beta momentum at 0.8, lambda decay rate at 10 and only one update per mini-batch per epoch performed well across all models. We found that mini-batches of size 25, 50, 75 and 100 performed equally well for all models, but due to speed considerations chose mini-batches of size 100. The optimal parameters for initial learning rates differed for each model. Therefore we ran a final hyperparameter optimization for MLP models with hidden units $2, 4, \ldots, 18, 20$ and initial learning rates in $\{10^{-0.5}, 10^{-1}, \ldots, 10^{-2.5}, 10^{-3}\}$ over the entire training set and tested on the validation set. The initial learning rate $10^{-2}$ had the lowest accuracy averaged across all number of hidden units and was therefore chosen. The models were all retrained with this initial learning rate on the training and validation sets, and tested on the test set. The tables and plots below show performance w.r.t. the test set unless otherwise stated.

As shown in figure 6.4, when we take into account the standard deviations the MLPs are clearly on par with the LogReg model taking Meep Features as input. In particular, The MLP with 14 hidden units performed best on average with a mean accuracy 62.95% and standard deviation 0.36.

In the later training stages, we observed some overfitting in the models. This is shown in figure 6.5 and 6.6, where the accuracy and log-likelihood on the training set consistently

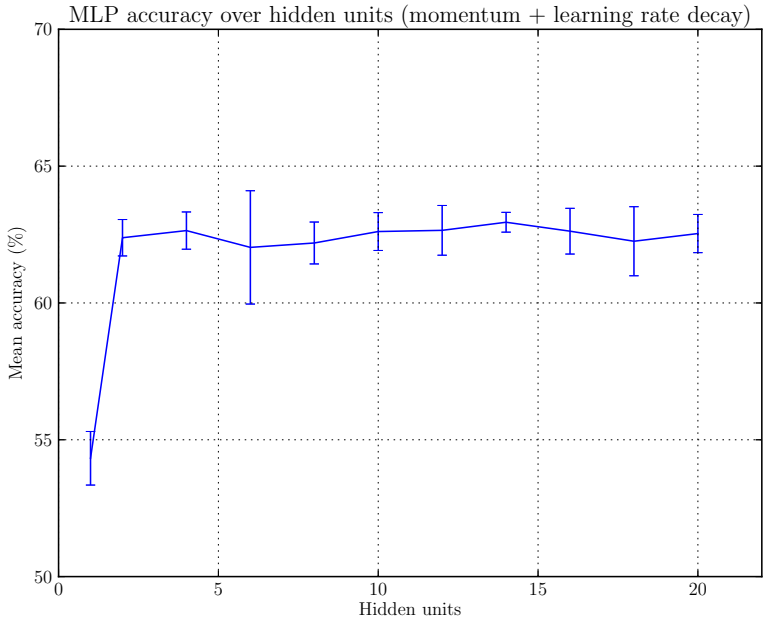



Figure 6.4: Mean accuracies and standard deviations for MLP models trained with momentum and learning rate decay on Bitboard Features over 10 trials. For comparison, the accuracy of the MLP model with a single hidden unit is also shown. Predicting with the most frequent outcome on the entire dataset yields 40.95% accuracy and with LogReg model based on Bitboard Features 57.34% accuracy.

improves with the number of hidden units while staying constant on the validation and test sets. We speculate that this is due to two reasons. Firstly, the bitboard representation is extremely sparse and hence the model would require many examples of a particular situation before they can construct a feature to recognize it. Secondly, the training instances are highly correlated which encourages the MLPs to recognize patterns occurring in the early board positions. For example, pieces which are left immobile during the opening game, e.g. any square they could move to is occupied by other pieces which in turn are rarely moved because they could harm the player, will have a relatively constant representation over the entire game, and hence the model could easily learn to associate the game outcome based on these piece locations instead of the other, more mobile, pieces. In general, however, such features are much less indicative of the actual game outcome since they do not represent many of the moves occurring during the game.

Furthermore, since the accuracies are only substantially outperformed by the LogReg model with Stockfish's QSearch Features, and on par with the accuracy of the LogReg model with Meep Features, it is also possible that the performance can only be improved by constructing highly complex features, for example temporal features which are analogous to searching several moves ahead.

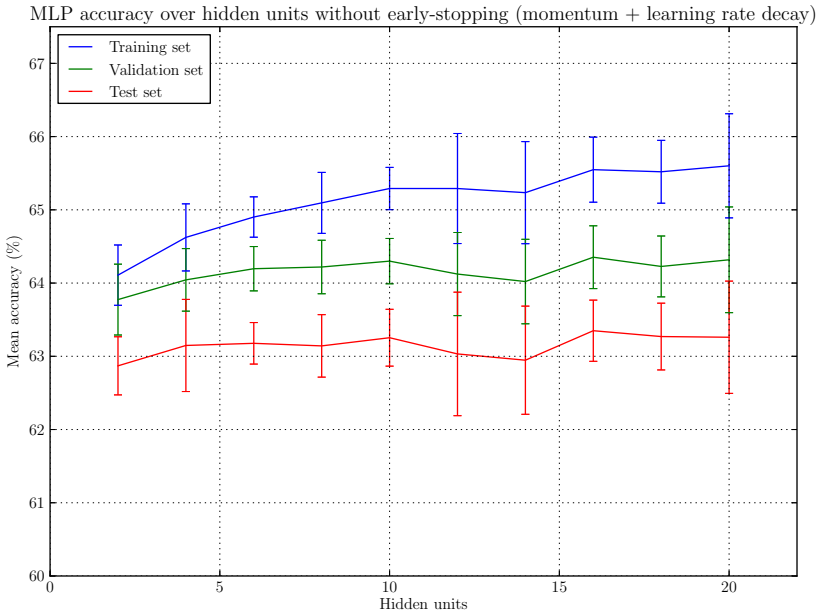



Figure 6.5: Mean accuracies and standard deviations over 10 trials for MLP models, trained with momentum and learning rate decay, on Bitboard Features over 10 trials at 10th epoch, i.e.. the MLP parameters without applying early-stopping.

We further analysed the features learned by the best performing MLP with 20 hidden units. This model obtained an accuracy at 63.40%. Most notably, we observed that hidden units tended to specialize on s a subset of pieces. In particular, some of the hidden units would weight White rooks and White queens and not assign any significant weights to any other pieces. This might result in the same behaviour as simply counting material values, or perhaps detecting the co-existence of two particular pieces. For comparison to the LogReg coefficients, two of the most influential features are shown in figure 6.7 and figure 6.8, which primarily

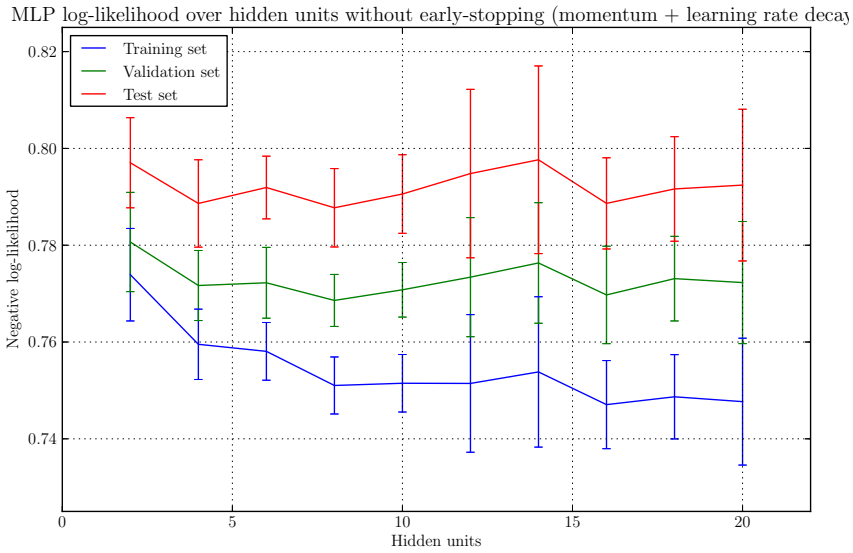



Figure 6.6: Mean (negative) log-likelihoods and standard deviations over 10 trials for MLP models, trained with momentum and learning rate decay, on Bitboard Features over 10 trials at 10th epoch, i.e.. the MLP parameters without applying early-stopping.

recognize the existence of rooks and queens [20]. We further observed that some hidden units appeared to recognize initial starting positions, and some to recognize rooks on the back rank of the opponent player. Contrary to the LogReg model, we did not observe any patterns around the center squares. Although the reader should observe that it is generally very difficult to interpret the features produced by MLPs, since they are all highly co-dependent on each other and affected by the frequency with which each square contains a certain piece.

In addition to this, we observed that the parameter values for White wins and Black wins in the LogReg layer were highly (negatively) correlated with a Pearson's correlation coefficient at $-0.9607$. This implies that the model, without any predefined prior knowledge or invariance structure, has constructed features which are highly symmetric w.r.t. playing colors [21].

[20] The reader should note that these features are w.r.t. absolute values transformed over all piece types, where conversely the LogReg features are not absolute values and linearly transformed across each piece type.

[21] We did not observe any similar correlation between drawing and other classes.

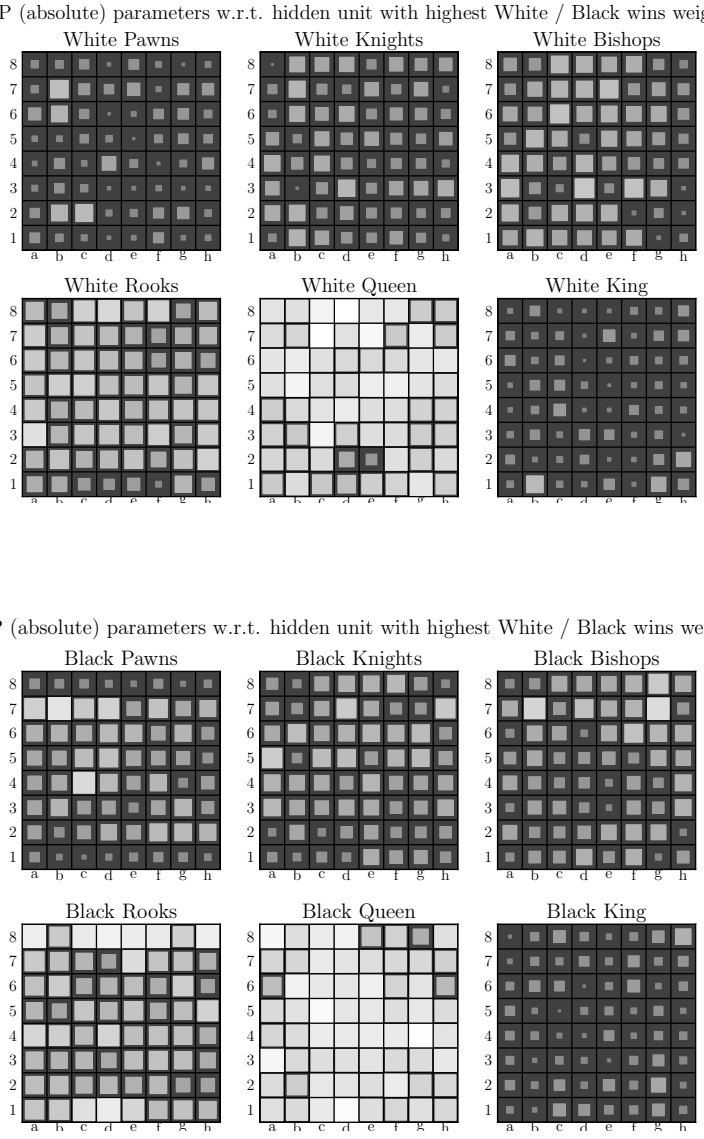



Figure 6.7: Parameters for the hidden unit with the highest absolute parameter in the LogReg layer w.r.t. White wins and Black wins (which happened to be the same hidden unit). The parameters shown are w.r.t. the best performing MLP model with 20 hidden units trained on momentum and learning rate decay. To make interpretation easier, the absolute value of the parameters were divided by the maximum absolute value across all pieces. The values can be compared across piece types, where larger and whiter squares correspond to an increased influence in the hidden unit.

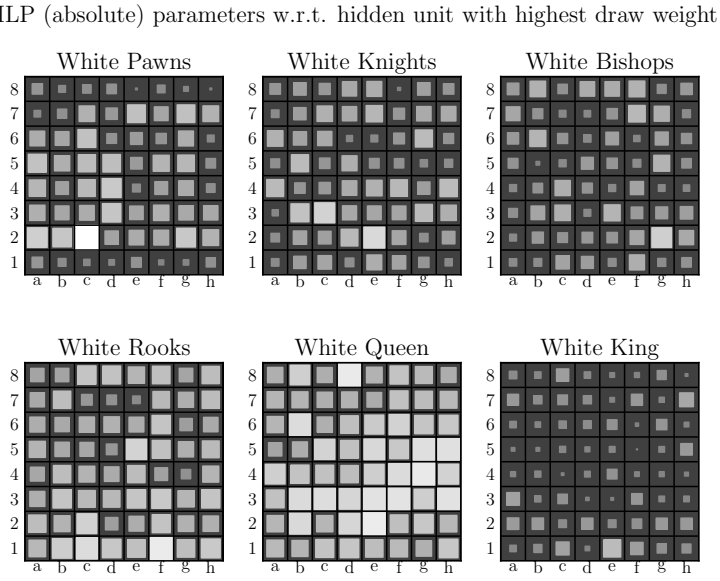



Figure 6.8: The feature related most heavily to draws outcome, i.e. the hidden unit with the highest absolute parameter in the LogReg layer for drawing. The parameters shown are w.r.t. the best performing MLP model with 20 hidden units trained on momentum and learning rate decay. To make interpretation easier, the absolute value of the parameters divided by the maximum absolute value across all pieces is shown. The values can be compared across piece types, where larger and whiter squares correspond to an increased influence in the hidden unit.

To investigate the influence of the training procedure, we carried out an almost identical experiment where the parameter update direction was determined by Nesterov's accelerated gradient, instead of momentum with learning rate decay. We also did a preliminary experiment where we compared Nesterov's accelerated gradient to method *AdaGrad* described in (Duchi et al., 2011), but found that Nesterov's accelerated gradient gave consistently more stable results. From preliminary experiments we set $\mu_t = 1 - 3(t - 5)$ where $t$ is the epoch number and determined that Nesterov's accelerated gradient is more effective with larger mini-batches.

We therefore ran a number of grid search experiments for $2, 4, \ldots, 20$ hidden units over the hyperparameters:

- mini-batch sizes in $\{200, 500, 1000, 2000\}$,
- learning rates in $\{10^{-1}, 10^{-1.5}, \ldots, 10^{-4}, 10^{-4.5}\}$.

We applied only one update per mini-batch per epoch.

Based on the grid search, we found that a mini-batch size of 2000 and a learning rate decay at $10^{-3.5}$ obtained the highest accuracy averaged over all models. The models were then retrained with these parameters on the training and validation sets, and tested on the test set. The results are shown below.

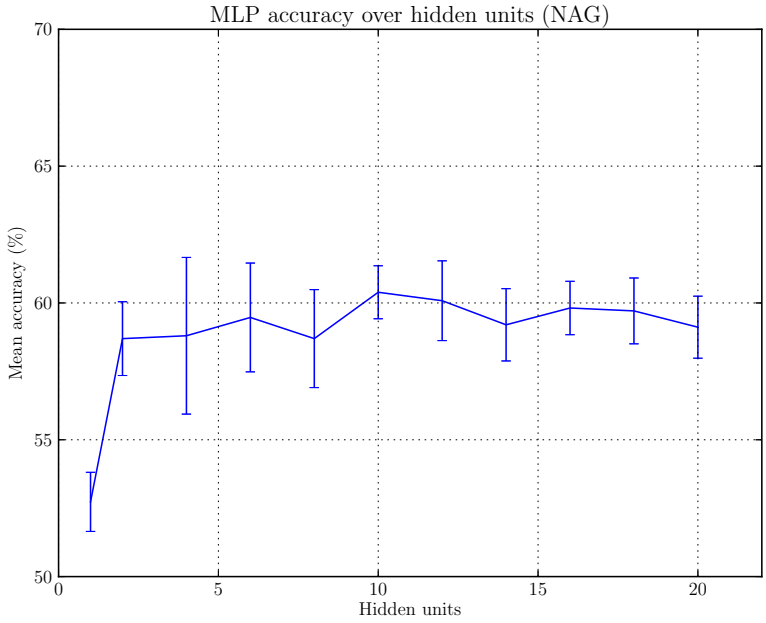



Figure 6.9: Mean accuracies and standard deviations for MLP models trained with Nesterov's accelerated gradient on Bitboard Features over 10 trials. For comparison, the accuracy of the MLP model with a single hidden unit is also shown. Predicting with the most frequent outcome on the entire dataset yields 40.95% accuracy and with LogReg model based on Bitboard Features 57.34% accuracy..

The results are shown in figure 6.9. The MLP with 10 hidden units performed best on average with a mean accuracy 60.39% and standard deviation 0.97. We observed the same overfitting phenomena as before. However, all models now performed worse than those trained with momentum and learning rate decay. Their mean accuracies were considerably lower and their standard deviations higher.

We speculate that this is because momentum and learning rate decay induces a higher degree of noise into the training procedure, which allows the parameters to jump between local optima more frequently. Combined with the early-stopping procedure, this becomes analogous to training several models and picking the best model based on its accuracy on the validation set. The same behaviour could happen when applying Nesterov's accelerated gradient, but due to its faster convergence rate, probably to a much lower extent.

To alleviate the overfitting problem, we tried to use L1-regularization with Nesterov's accelerated gradient. We used the same hyperparameters as in the previous experiment with L1-regularization parameters in $\{10^{-5}, 10^{-4.5}, \ldots, 10^{-3}, 10^{-2.5}\}$.

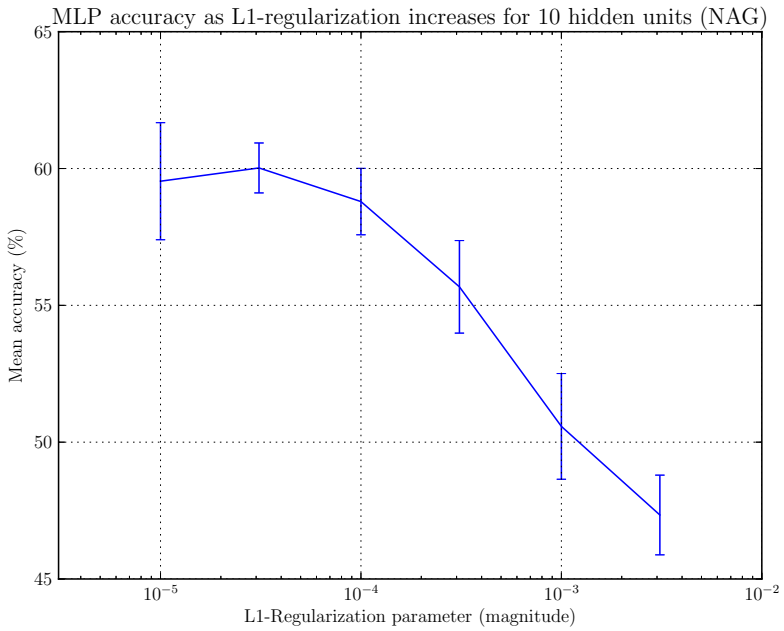



Figure 6.10: Mean accuracies and standard deviations for MLP models with 10 hidden units,

As shown in figure 6.10, L1-regularization improves the accuracy slightly but still fails to reach the accuracy of the MLPs trained with momentum and learning rate decay. This suggests that it is more beneficial to train many models from random initialization points, and pick the best model according to a validation set, than to apply L1-regularization.

We also carried out similar experiments for 20, 30 and 40 hidden units, on a smaller dataset, but these did not show any significant improvement in accuracy when adding L1-regularization.

### 6.9.2 Two-Layer MLPs

In this section we investigate the performance of two-layer MLPs. Despite having observed a significant amount of overfitting in the previous experiments, there are two arguments in favour of experimenting with larger and deeper models. Firstly, researchers have proposed that MLPs with fixed randomly initialized hidden units, where only the output layer is trained, may work very well (Huang et al., 2006). The idea is that, while the hidden units are random, some of them will still represent useful features the top layer can use for classification. Although we shall continue to train all parameters in the model, we hope that a larger and deeper randomly initialized MLP will contain more useful features at the beginning, which can be narrowed down as training progresses. In particular, since our bitboard representation is extremely sparse, it is highly unlikely that any single hidden unit represents a sensible feature at initialization. Secondly, it has been observed that increasing the number of effective parameters by adding hidden units rarely decreases the performance (Lawrence et al., 1996), but that it can actually improve performance under proper initialization (Bengio, 2012, p. 11).

We choose to experiment on two-layer MLPs with various architectures, containing 20, 50, 100 and 200 hidden units in the first layer, and 20, 50 and 100 hidden units in the second layer. We continued to use Nesterov's accelerated gradient with $\mu_t = 1 - 3(t - 5)$, and ran a grid search over the hyperparameters:

- mini-batch sizes in $\{200, 500, 1000, 2000\}$,
- learning rates in $\{10^{-0.5}, 10^{-1}, \ldots, 10^{-3}, 10^{-3.5}\}$.

We applied only one update per mini-batch per epoch.

We found that a mini-batch size of 2000 and learning rate at $10^{-3.5}$ obtained the highest accuracy, when averaged over all the models. We retrained the models and tested them on the test set. The results are given in table 6.13.

Clearly these models perform worse than all previous MLPs. They obtain lower mean accuracy and have higher standard deviations in most cases. In fact, they are not substantially better than the benchmark logistic regression model. This suggests that adding additional hidden units, initialized at random, does not find features which generalize well.

| First layer hidden units | Second layer hidden units | Mean accuracy |
|---|---|---|
| 20 | 20 | $58.52\% \pm 1.26$ |
| 20 | 50 | $58.55\% \pm 0.82$ |
| 20 | 100 | $58.87\% \pm 2.00$ |
| 50 | 20 | $59.75\% \pm 1.44$ |
| 50 | 50 | $58.74\% \pm 1.65$ |
| 50 | 100 | $57.75\% \pm 2.80$ |
| 100 | 20 | $58.96\% \pm 2.36$ |
| 100 | 50 | $57.63\% \pm 1.86$ |
| 100 | 100 | $59.62\% \pm 1.03$ |
| 200 | 20 | $57.80\% \pm 2.14$ |
| 200 | 50 | $58.90\% \pm 1.30$ |
| 200 | 100 | $58.17\% \pm 1.75$ |

Table 6.13: Mean accuracies and standard deviations for two-layer MLP models trained with Nesterov's accelerated gradient on Bitboard Features over 10 trials. In comparison, predicting with the most frequent outcome on the entire dataset yields 40.95% accuracy and with LogReg model based on Bitboard Features 57.34% accuracy.

### 6.9.3 ConvNets

For the ConvNets we choose a 3-layered architecture similar to (LeCun et al., 1989), where the first layer was designed with either a $5 \times 5$ or a $3 \times 3$ receptive field, the second layer a $3 \times 3$ receptive field and the third layer fully-connected. As we argued in section 3.3, many features in chess require integrating information across most of the board. Although, in principle, small receptive fields can still propagate the information untransformed, e.g. by applying the identity function in the first layers and then model the features at the top fully-connected layer, this would require many more feature maps. For this reason we choose to experiment with both $3 \times 3$ and $5 \times 5$ receptive fields in the first layer. For a $5 \times 5$ receptive field in the first layer, the input to the second layer would be a $4 \times 4 \times$ *number of feature maps in first layer*, which makes the second layer $3 \times 3$ receptive field a natural choice as it is the largest (non fully-connected) square receptive field possible. Of course, many other architectures are possible, but due to time constraints those are outside the scope of this project.

A bitboard representation for each piece type is added as a separate channel in the 3D input tensor. To handle the side-to-move boolean feature, we expand it to a 2D tensor (a matrix

with identical values) and add it as a channel in the 3D input tensor. Without regularization, this is effectively the same as adding the side-to-move feature to every feature map in the first layer.

We experimented with various number of feature maps ranging from 5 to 20 feature maps in the first layer, 10 to 40 feature maps in the second layer and 100 to 200 feature maps in the third layer. No max-pooling was used. Under the Bitboard Features, our largest model (with $5 \times 5$ receptive fields in the first layer) with 20, 40 and 200 feature maps in the first, second and third layer respectively has 45,882 parameters in total (6,040, 7,240, 32,200 and 402 parameters corresponding to hidden units in the first, second, third and LogReg layer respectively). Compared to LeCuns 3-layered ConvNet with a total of only 9,760 parameters, which was able to classify digits $0, \ldots, 9$ highly accurately, our model has over a four-fold number of effective parameters (LeCun et al., 1989). On the other hand, our smallest model (with $5 \times 5$ receptive fields in the first layer) with 5 and 10 feature maps in the first and second layer respectively and 100 hidden units in the third layer, has 6,272 effective parameters (1,510, 460, 4,100 and 202 parameters corresponding to hidden units in the first, second, third and LogReg layer respectively), which is about four times the number of effective parameters in the LogReg model taking Bitboard Features as input. We further note, with the exception of different activation functions and model size, the same 3-layered convolutional structure was used in (Mnih et al., 2013).

Similar to the MLPs, we ran a number of grid search experiments to find suitable ranges for our hyperparameters. Our experiments included:

- mini-batch sizes in $\{25, 50, 100, 200, 500, \sim 20000\}$,
- beta momentums in $\{0.2, 0.4, \ldots, 0.8, 1.0\}$,
- lambda decay rates in $\{0.1, 0, 1.0, 10.0\}$,
- initial learning rates in $\{10^{-1.5}, 10^{-2}, \ldots, 10^{-2.5}, 10^{-3}\}$.

The number of updates per mini-batch was fixed to one.

Based on these experiments, we fixed the mini-batch size to 25, momentum to 0.6, learning rate decay to 10 as these performed well across most models. We then trained the networks

and tested them on the validation set for initial learning rates in $\{10^{-1}, 10^{-1.5}, \ldots, 10^{-3}, 10^{-3.5}\}$, and found that the initial learning rate $10^{-3}$ averaged over all the models performed best w.r.t. accuracy. All models were then retrained on the entire training and validation set with this learning rate, and tested on the test set [22]. The results are given in table 6.14.

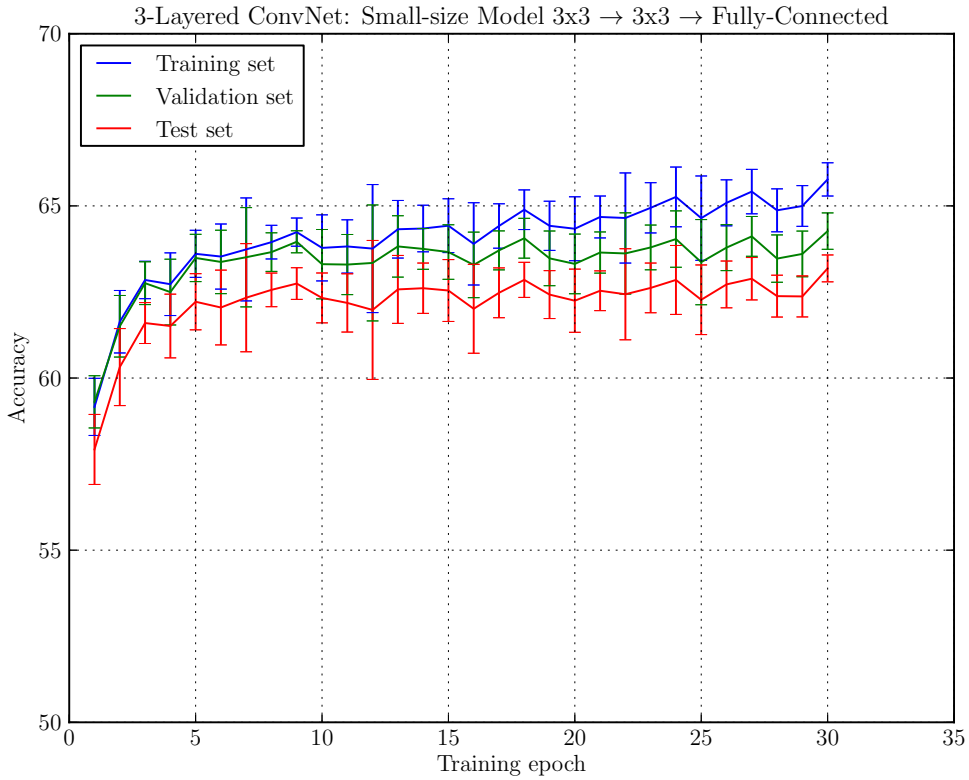



Figure 6.11: Mean accuracies and standard deviations over 10 trials for 3-layered $3 \times 3 \rightarrow 3 \times 3$ ConvNet, with 5, 10 and 100 feature maps in layer one, layer two and layer three respectively, plotted against training epoch.

As table 6.14 clearly shows, all ConvNet models are achieve substantially better mean accuracies than the LogReg model with Bitboard Features. However, as the plots in figures 6.12 and 6.13 also show, a significant amount of overfitting is occurring for the larger models

[22]The ConvNet models were trained on slightly fewer training examples than the MLP models, since the number of training instances had to be clipped to fit training batches of size 25, validation batches of size 500 and test batches of size 5000.

| **ConvNet** | **Configuration** | **Mean accuracy** |
|---|---|---|
| Bitboard Features | • $3 \times 3$ pooling with 5 features in first layer<br>• $3 \times 3$ pooling with 10 features in second layer<br>• 100 hidden units in third layer | $\mathbf{63.25\% \pm 0.50}$ |
| Bitboard Features | • $3 \times 3$ pooling with 5 features in first layer<br>• $3 \times 3$ pooling with 10 features in second layer<br>• 200 hidden units in third layer | $62.08\% \pm 0.83$ |
| Bitboard Features | • $3 \times 3$ pooling with 10 features in first layer<br>• $3 \times 3$ pooling with 20 features in second layer<br>• 100 hidden units in third layer | $62.57\% \pm 0.82$ |
| Bitboard Features | • $3 \times 3$ pooling with 10 features in first layer<br>• $3 \times 3$ pooling with 20 features in second layer<br>• 200 hidden units in third layer | $62.74\% \pm 0.37$ |
| Bitboard Features | • $3 \times 3$ pooling with 20 features in first layer<br>• $3 \times 3$ pooling with 40 features in second layer<br>• 100 hidden units in third layer | $62.45\% \pm 0.69$ |
| Bitboard Features | • $3 \times 3$ pooling with 20 features in first layer<br>• $3 \times 3$ pooling with 40 features in second layer<br>• 200 hidden units in third layer | $62.24\% \pm 1.21$ |
| Bitboard Features | • $5 \times 5$ pooling with 5 features in first layer<br>• $3 \times 3$ pooling with 10 features in second layer<br>• 100 hidden units in third layer | $63.04\% \pm 0.68$ |
| Bitboard Features | • $5 \times 5$ pooling with 5 features in first layer<br>• $3 \times 3$ pooling with 10 features in second layer<br>• 200 hidden units in third layer | $62.80\% \pm 0.92$ |
| Bitboard Features | • $5 \times 5$ pooling with 10 features in first layer<br>• $3 \times 3$ pooling with 20 features in second layer<br>• 100 hidden units in third layer | $62.94\% \pm 0.85$ |
| Bitboard Features | • $5 \times 5$ pooling with 10 features in first layer<br>• $3 \times 3$ pooling with 20 features in second layer<br>• 200 hidden units in third layer | $62.90\% \pm 0.83$ |
| Bitboard Features | • $5 \times 5$ pooling with 20 features in first layer<br>• $3 \times 3$ pooling with 40 features in second layer<br>• 100 hidden units in third layer | $62.87\% \pm 0.46$ |
| Bitboard Features | • $5 \times 5$ pooling with 20 features in first layer<br>• $3 \times 3$ pooling with 40 features in second layer<br>• 200 hidden units in third layer | $62.48\% \pm 0.76$ |

Table 6.14: Mean accuracies for ConvNets, with two pooling layers and a fully-connected layer, taking Bitboard Features as input over 10 trials. The term *features* in the configuration refers to the number of feature maps at each level of the network. In comparison, predicting with the most frequent outcome on the entire dataset yields 40.95% accuracy and with LogReg model based on Bitboard Features 57.34% accuracy.

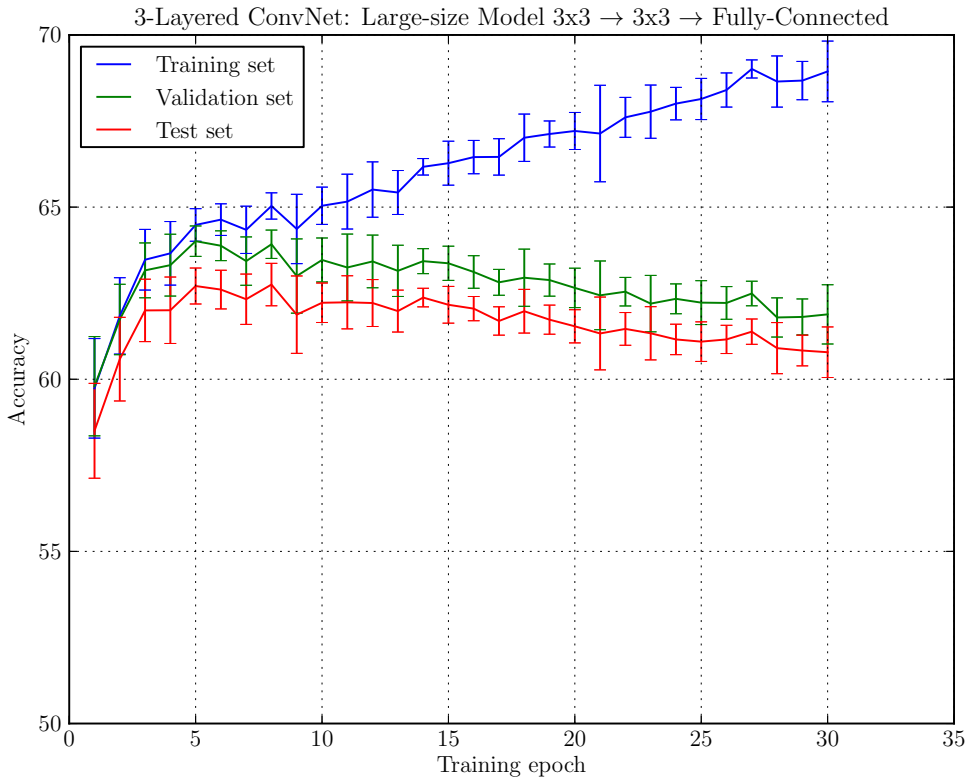



Figure 6.12: Mean accuracies and standard deviations over 10 trials for 3-layered $3 \times 3 \rightarrow 3 \times 3 \rightarrow$ fully-connected ConvNet, with 20, 40 and 200 feature maps in layer one, layer two and layer three respectively, plotted against training epoch.

for both convolutional structures. Furthermore, although this is confounded by the overfitting phenomena, the smallest ConvNets appear to perform slightly better than the corresponding MLPs. For example, both the MLP models with 2 and 4 hidden units are outperformed by the ConvNet model with 5 and 10 feature maps in layer one and layer two respectively, and 100 hidden units in layer three. This appears to suggest that the convolutional restriction helps the model to generalize to unseen positions, but further experiments would clearly be necessary to verify this. Even if the convolutional structure does improve the performance, it is perhaps less useful in chess than in other games such as Go where experiments have shown it to beneficial (Schraudolph et al., 2001).

Since both ConvNet models with $5 \times 5$ and $3 \times 3$ convolutions in the first layer appear to

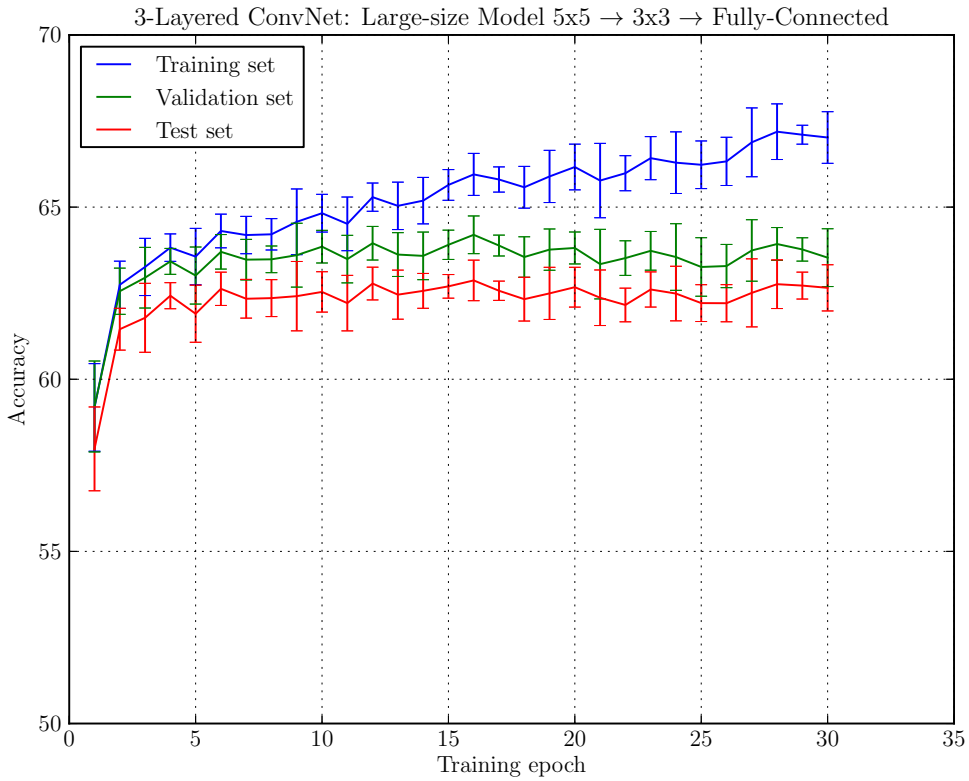



Figure 6.13: Mean accuracies and standard deviations over 10 trials for 3-layered $5 \times 5 \rightarrow 3 \times 3 \rightarrow$ fully-connected ConvNet, with 20, 40 and 200 feature maps in layer one, layer two and layer three respectively, plotted against training epoch.

perform almost equally well, it is not clear which one should be preferred. However, after being selected for by lowest accuracy on the validation set, the largest model with $5 \times 5$ convolution in the first layer clearly performs best w.r.t. accuracy on the training set, as shown table D.2 in the appendix. Given more training data, we might hope that this will be the best model.

We also note that there is no significant difference between models with 100 hidden units and models with 200 hidden units in the last layer. Perhaps this can be explained by the fact that the lower levels do not produce sufficiently many distinct features for the fully-connected layer to employ all of its hidden units.

We further analysed the features learned by the best performing ConvNet with $5 \times 5 \rightarrow 3 \times 3 \rightarrow$ fully-connected convolutional structure, and 10 and 20 feature maps in the first and

second layer, and 100 hidden units in the last layer. The model obtained an accuracy at 64.18%, which is on par with the LogReg model taking Meep Features as input. We observed in the first layer that some feature maps assigned equal weights to the same piece type over all squares, perhaps as a way of counting material, while other feature maps appeared to recognize proximity between pieces of the same color [23], in particular for rooks and queens. A few of the feature maps also appeared to assign significant weights only to one or two pieces along one or two files, ranks or diagonals [24]. Some feature maps also appeared if a pawn is defended by another pawn.

However, contrary, to the MLP features analysed earlier, the correlation between parameter values w.r.t. White wins and Black wins in the LogReg layer were considerably lower, with a Pearson correlation coefficient at only $-0.4132$. Perhaps this can be explained by the large number of hidden units in the third layer, of which many may be useless to the classification and hence are assigned a small random value in the LogReg layer.

### 6.9.4 Mixture of experts

We ran experiments with ME models with 2, 4, 6, 8 and 10 mixture components on the Bitboard Feature set. We also experimented with an ME model with a single mixture component for comparison, because it is equivalent to a LogReg model trained with our EM procedure. Based on preliminary experiments, we found that the model very easily overfitted on the validation set. We therefore chose to use L1-regularization.

We carried out two sets of experiments. The first used the backtracking line-search method based on the *Goldstein conditions* as described in (Nocedal and Wright, 2006, Chapter 3, Algorithm 3.1) in the M-Step on batches of size $20,000$ with early stopping. The algorithm effectively replaced the M-Step in algorithm 5 with a line-search optimization on the free energy. This approach can be motivated by the fact that the M-step is a convex optimization problem, as we discussed in section 3.5, where line-search methods are often (including ours) guaranteed to converge. We tried with various hyperparameters for a fixed mini-batch size of $20,000$ training

[23] This is arguably very simple for the ConvNets to model as they are already translation invariant, and only need to assign significant weights in a small area of the receptive fields for two pieces to recognize proximity.

[24] One peculiar feature appeared to simultaneously recognized if the White queen and White rook were on the same file and if the White queen and White bishop were on the same diagonal.

instances, and found that while the log-likelihood and accuracy was improving on the training set it was consistently decreasing on the validation set w.r.t. number of mixture components. We tried with smaller mini-batch sizes as well, but this quickly became computationally prohibitive due to which we were unable to improve the accuracy w.r.t. the validation set. In all experiments the model with a single expert, equivalent to the LogReg model, was found to perform better than any of the models with more mixture components. The regularization induced by the early-stopping procedure did not manage to improve on this.

We therefore carried out a second experiment, where we used algorithm 5. As before, we ran a number of grid search experiments to find suitable ranges for our hyperparameters. We experimented with different number of mini-batch sizes and found that sizes of 500 worked well across most hyperparameter settings. This performed markedly better than the experiments with line-search based on the validation set, but the models with more than one expert were still performing worse than the model with a single expert w.r.t. accuracy on the validation set. See figure 6.14.

Our final experiment, which due to its negative result we chose to only evaluate on the validation set, included learning rates in $5.0, 1$ and L1-regularization penalties in $\{10^{-3}, 10^{-3.5}, \ldots, 10^{-5.5}, 10^{-6}\}$. We used the the approximation to the L1-regularization term given in (Schmidt et al., 2007, eq. (1)–(4) ). The results are given in figure 6.15.

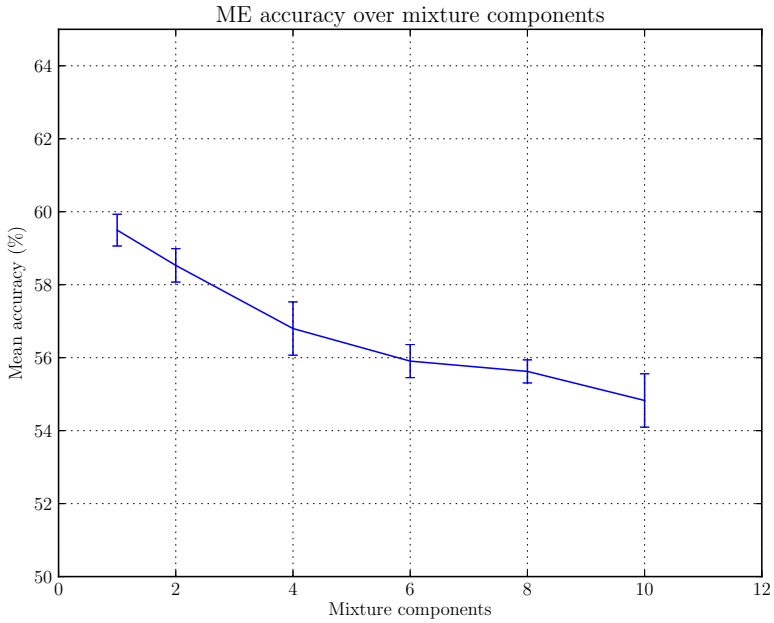



Figure 6.14: Mean accuracies and standard deviations for ME models trained with stochastic gradient descent over mini-batches of size 500 w.r.t. number of mixture components over 3 trials. See algorithm 5. The results reported are based on the validation set for the hyperparameters which achieved the highest average validation accuracy and should therefore be considered slightly biased towards higher accuracies.

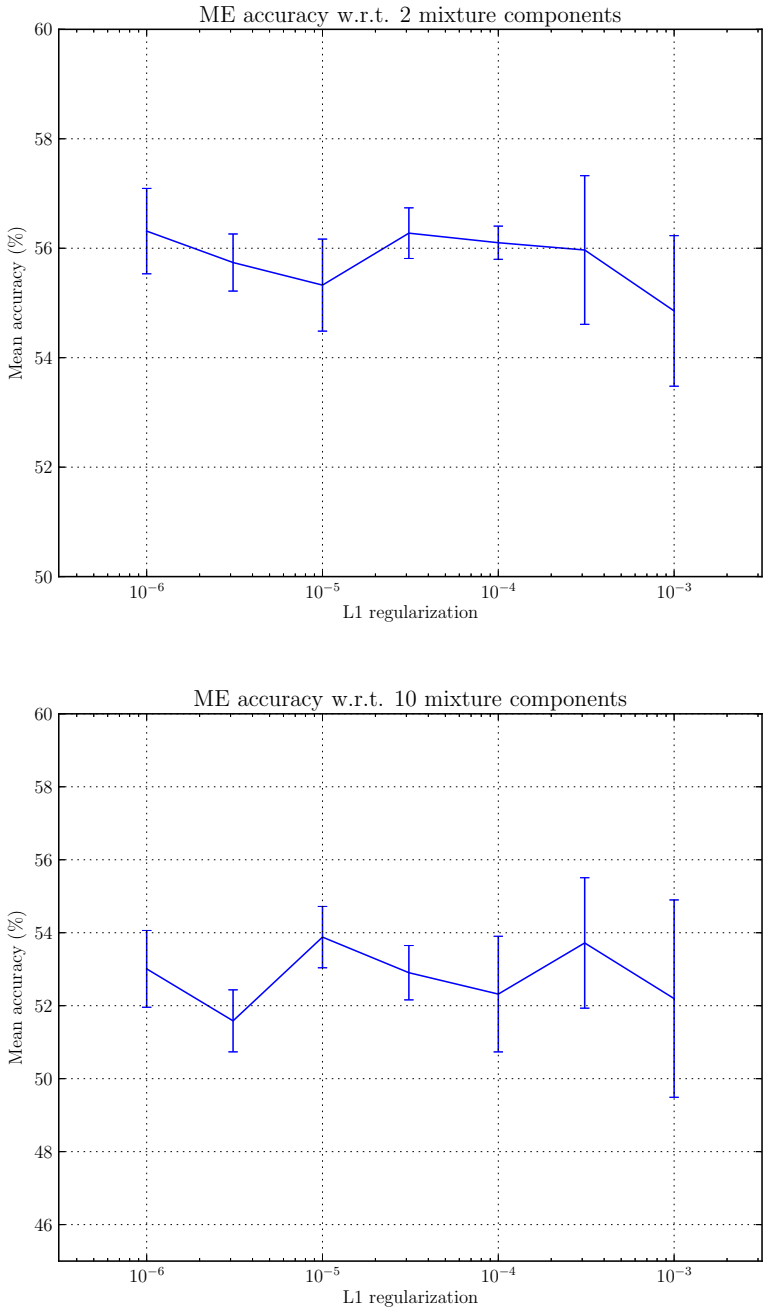



Figure 6.15: Mean accuracies and standard deviations for ME models, with 2 and 10 mixture components respectively, trained with stochastic gradient descent over mini-batches of size 500 w.r.t. L1-regularization parameter. The results reported are based on the validation set for the hyperparameters which achieved the highest average validation accuracy and should therefore be considered slightly biased towards higher accuracies.

We speculate that it is the large number of parameters, which makes the models overfit heavily. The LogReg model has only $2 \times (12 \times 64 + 2) = 2 \times 770 = 1,540$ effective parameters, while the ME model with 2 mixture components has $5 \times 770 = 3,850$ effective parameters. By having twice as many effective parameters compared to the LogReg model, yet not taking into account higher-order terms directly, it is difficult for the model to capture statistical correlations between few pieces. Instead, the model structure promotes overfitting by adjusting each mixture component to fit an entire board position or sets of board positions. The ME model with 10 mixture components has $(9 - 1) + 10 \times 2 \times 770 = 15,409$ effective parameters, i.e. almost a five-fold increase from the model with 2 mixture components, which would only exaggerate this hypothesis.

Although L1-regularization should improve the sparsity of the coefficients, and hence its generalization ability and the accuracy on the validation set, we speculate that it also makes it more likely that a single expert takes responsibility for all observations. This phenomena was observed in several models, although we did not have time to quantify it statistically. This would effectively reduce the more L1-regularized ME models to standard LogReg models, which may explain why L1-regularization did not improve the accuracy substantially.

## 6.10 Discussion

In our supervised learning experiments we have found MLPs and ConvNets to be the most promising models. Although we observed a significant amount of overfitting in all the models, due in part to the extremely sparse bitboard representation and co-dependence of training instances, the one-layer MLPs and three-layer ConvNets outperformed the LogReg model with Bitboard Features, and further managed to reach mean accuracies comparable to the LogReg model with Meep Features. By our hypothesis, this suggests that they do indeed have the representational power necessary to achieve master-level play if they are trained to work in conjunction with a search procedure. We attribute the success to their structural design, which contrary to the ME, promotes features that work on a particular subset of pieces and squares. However, due to the overfitting phenomena, it is not possible to conclude which of the one-layer MLP or three-layer ConvNet models will perform better in the self-play learning experiments. Although, we further observed that two-layer MLPs overfitted significantly more than one-

layer MLPs and ConvNets, which might suggest that the translational invariance property is important for learning a hierarchical feature representation in our domain.

The fact that our models required around 10 epochs over $43,558$ games to reach a peak performance and afterwards started to overfit for the bitboard representation, may help to explain the poor results reported in (Mannen, 2003; Wiering et al., 2005) based on similar representations but trained on a magnitude fewer training instances. This may also explain why the more structured regularization technique TangentProp technique was more successful in (Thrun, 1995).

In general, we cannot conclude whether training MLPs with momentum and learning rate decay or Nesterov's accelerated gradient will work better in the self-play experiments, however momentum and learning rate decay appears to be sufficient for training ConvNets in the self-play experiments.

Although the granularity of most features appeared to be on the level of piece types, we did observe a few well-known chess features. Perhaps most interestingly, we found some evidence that ConvNets were measuring proximity between pieces of the same color as a feature. In comparison, the LogReg model with Bitboard Features, by design, appeared to take into account mobility, attack and defence ability independently for all pieces.

# Chapter 7

# Self-Play Experiments

## 7.1 Setup

In our second set of experiments, we are concerned with learning the heuristic evaluation function from self-play games with TreeStrap and evaluating the performance w.r.t. its actual playing strength.

We shall implement the heuristic evaluation functions into the Meep chess engine (Veness et al., 2009) [1]. Meep contains both a minimax search and an alpha-beta pruning search procedure. To avoid any artefacts caused by the alpha-beta pruning procedure, as for example in (Veness et al., 2009, Figure 2), we use only the minimax search. Tesauro has also argued to avoid the heuristic cut-offs imposed by typical alpha-beta pruning procedures, such as null-move pruning (Tesauro, 2001, p. 5). The minimax search procedure is an optimized version of algorithm 1, which also uses a transposition table with move ordering and extends any leaf node containing a king in check with a one ply search [2] [3] (Russell and Norvig, 2009, p. 170). A quiescence search is further applied to the leaf nodes of the minimax search tree. This is configured to only examine check evasions (Beal, 1990) [4]. Following the same search procedure

[1] Unlike our previous experiments implemented in Python, this implementation is carried out in C++ which will then subsequently call Python functions.

[2] These modifications simply extend the minimax search tree to include additional nodes and hence increase the search depth along particular lines of play.

[3] The search tree was stored entirely in memory, and the transposition table was cleared after each move.

[4] That is, all positions containing a check is considered non-quiescent. In the original paper proposing

as (Veness et al., 2009) makes it easier to compare our results to theirs. To reduce computational time, we fix the minimax search tree to have a maximum of $5,000$, and the quiescence search tree to have a maximum of 500 nodes for each leaf in the minimax tree [5] [6].

One may have discarded self-play learning with TreeStrap and instead naively used the models trained in the supervised experiments as heuristic evaluation functions. However, these models were trained on a very different set of board positions compared to those the chess engine encounters in the search tree. The search tree contains lines of all possible movements, including e.g. disastrous movements such as sacrificing pieces, while our masters game collection contains mainly good lines of play and only rarely disastrous moves. In addition to this, since we are using a quiescence search, the heuristic evaluation function will only be applied to quiescent positions, while our supervised learning models have been trained on all types of positions in the game collection.

To limit the scope of the project, we choose only to experiment with the two objective functions squared error and soft cross-entropy. The first is the most promising based on earlier research in (Veness et al., 2009), and the second is the most promising from our theoretical analysis in section 4.2. Furthermore, to accommodate the scalar minimax values used by the minimax procedure in Meep, we shall use only binary classification models. That is, the top logistic regression layer in every model has only White wins and Black wins classes.

To induce variation into the games played, we used the opening book Encyclopaedia of Chess Openings (ECO) (ECO, 2014) [7]. The ECO contains 2,014 opening variations, comprising a total 20,697 moves. It was produced by notable chess players for the publishing company Chess Informant. In both self-play and tournament games, each game is started from the last board position in an opening variation selected at uniformly random. To reduce variance, the same

TreeStrap, the quiescence search also examined capture evasions (Veness et al., 2009). Due to a recommendation by Veness, we choose not to do this in our experiments (Veness, 2014b).

[5] Due to its recursive nature, the implementation actually searches slightly more nodes.

[6] The reader should observe that our implementation is a magnitude slower than otherwise possible, since the C++ program has to call a Python function for every board evaluation. It can be made a magnitude faster by implementing the models directly into C++. The program can be made even faster by incrementally updating binary representation for one board position based on the representation of the preceding board position. If the hidden units and feature maps are also updated incrementally the speed gain would be even higher, in particular for ConvNet models where only a fraction of feature maps would have to be updated after every move position.

[7] The opening book can be downloaded in pgn format from `http://www.chessvisor.com/v-aperturas.html`. Other formats are also available elsewhere online.

seed for the opening variation and starting player was used in each subexperiment.

## 7.2 Evaluation

The best way of evaluating the playing strength of our algorithm, apart from playing it against human opponents, would be to play it against a strong and well-known chess engine which rarely makes serious blunders (Ghory, 2004, pp. 31–34). Stockfish is such an engine (Romstad et al., 2014). We could in theory do this, but because our models will start out by playing very weakly, it will require thousands or hundreds of thousands of games to get a reliable estimate of its playing strength. This is not computationally feasible [8]. Therefore, we choose to play our models against each other. In particular, we will choose a subset of models as benchmark models, and play all other models against these.

An exhaustive discussion on the pros and cons of evaluating a game playing program against an opponent program is given in (Ghory, 2004, pp. 31–34). A first issue is that the evaluation may be biased by the strategies of the opponent program. If one program learns to exploit the weaknesses of the other program, it may appear to be playing strongly even if it is playing weakly against other opponents. This is a serious concern, but we hope that because all our models use the same set of features no serious weaknesses can be exploited by one of them easily. If on the other hand, for example, one model contained temporal features, e.g. a bitboard representation over squares it can attack in one move, the model would effectively have a larger search depth w.r.t. attacking lines than the other model and hence could easily exploit the opponent's shallow search.

Since our program is fixed to a certain search depth, it may be hard to separate the playing strength of the models from the search depth. We hope that this is mitigated, to some extent, by the fact that we are training in self-play with the same number of search tree nodes as when

[8] Effectively our models needs to win at least a few games against the opponent, perhaps due solely to serious blunders made by the opponent, to get a reliable estimate of its playing strength. Indeed, in our preliminary experiments we were not able to make our models (trained in the supervised experiment) win against Stockfish even when Stockfish was tuned to its lowest level. However, it should be noted that even at this level Stockfish still carries out a significant alpha-beta pruning and quiescence search corresponding to at least a 5 ply (Veness, 2014b) search, compared to our program which was limited to a minimax search tree and quiescence search tree with approximately 20,000 nodes (2-3 ply search depth) and 500 nodes (1-2 ply search depth) respectively.

playing against opponent programs, and since all models make use of the exact same search procedure.

Another issue, which we are not able to amend unfortunately, is that it may be hard for other researchers to reproduce the results of our experiments. The original codebase for Meep belongs to Veness, and they would therefore have to acquire it from him to reproduce the experiments (Veness, 2014b). However, the reader should observe that a very simple minimax and quiescence search procedure is used. In principle, it should be straightforward to implement it based on (Beal, 1990; Veness et al., 2009).

Fortunately by playing the programs against the benchmark programs we are sure that their strength is not significantly different, and we should therefore be able to measure their relative playing strength accurately in a small number of games. Furthermore, the entire experiment setup can easily be transferred to other game domains as we set out to do in the beginning of the project. Had we instead used an expert program, we would have had to find a separate (expert) program for every new game domain.

## 7.3 Models

We will use the training procedure described in algorithm 7. We will use a training set size of $N = 20,000$ instances from which we sample during the procedure.

As our benchmarks, we experiment with LogReg (logistic regression) models taking Bitboard Features as input, exactly as in the supervised experiments. We train them on the standard version of our modified self-play procedure given in algorithm 7. We shall play all other models against the benchmark LogReg models.

We experiment with one-layer MLP (multilayer perceptron network) models with 20 hidden units taking Bitboard Features as input. The MLP model has 15,442 effective parameters (770 per hidden unit plus 42 for the final two-class logistic regression layer). We believe this to be a sensible size for the self-play experiments, since it did not overfit substantially more than smaller models in the supervised learning experiment, yet it is a magnitude larger than the smallest MLP models on par with the LogReg benchmark model taking Meep Features as input. We choose to train all MLP models with algorithm 7 and Nesterov's accelerated gradient, as we do

not believe overfitting to be a problem in our self-play experiments and hence do not need the additional noise we speculate that training with momentum and learning rate decay induces.

Due to time constraints, we do not experiment with ConvNets.

## 7.4 Model Initialization

As we have already argued in sections 3.2 and 3.3, the objective functions of both MLPs and ConvNets contain many local optima. If we train them from a random initialization point, we risk that they learn features which are only useful for weak play and afterwards never improve. Of course, one could repeat the experiment with many different initialization points and hope some of them will learn features useful for intermediate and master-level play. One could also repeatedly add new hidden units and feature maps as training progresses to allow the models to construct additional features for when it reaches intermediate and master-level play. However, by taking either of these approaches we introduce both a higher computational cost, e.g. by having to repeat the experiment from different random initialization points many times, and additional hyperparameters which need to be tuned, e.g. parameters of the initialization distribution and the controlling how many additional hidden units and feature maps to add as learning progresses.

Instead, we therefore choose to keep our model structure fixed and initialize the parameters based on the models learned in the supervised learning experiments. These parameters have managed to pick up useful features for detecting the winning player in master-level games, and hence might be a good starting point to learn features with self-play which can reach master-level play. We use the approaches discussed in section 4.2 to convert them from three-class to two-class models.

We choose three models with different test accuracies. This will show us if models which are more fit for predicting the game outcome, under the assumption that both players are playing at a high level, are also more fit as heuristic evaluation functions. This might shed some light on our earlier hypothesis: a model, which is able to capture important features for predicting the winning player, also has the representational power to capture important features for ranking board positions in conjunction with a search procedure

## 7.5 Benchmarks

Based on preliminary experiments, we trained various LogReg models with both squared error and soft cross-entropy to find suitable hyperparameters:

- mini-batch sizes in $\{100, 200, 500, 1000, 2000, 5000\}$,
- (fixed) learning rates in $\{10^{-1}, 10^{-1.5}, \ldots, 10^{-5}, 10^{-5.5}\}$.

We used only one update per mini-batch. We set out to train all combination of hyperparameters for 50 games, but as the experiment unfolded we found that some models played for many hundreds of moves per game, while others only played less than a few hundred moves per game. We therefore assumed that the models playing longer games were either converging very slowly or diverging. After a fixed amount of computational time for each model, where about 10% of the models had reached 100 games, we stopped the experiment and discarded all models which had yet to play 50 games in total. Any model discarded should therefore on average have been playing twice as long games as the shortest 10% of the models. Using this procedure 32.5% of the models were discarded.

The remaining models were then used at the point where they were trained on exactly 50 games. Henceforth, we shall denote the models initialized based on the supervised learning experiments, and not trained any further, for SL models, and the models initialized based on the supervised learning experiments and trained for 50 games for RL models. All the trained LogReg RL models were then played against the soft cross-entropy LogReg SL model for 40 matches (games) each. In each match, if after 50 moves the game had not ended Stockfish, configured to a 14-ply search depth, would play both sides until the end of the game. This was done to speed up the procedure. Similar to before, we again discarded about 6% of the models which did not finish playing 40 games in fixed amount of computational time.

The model with the highest ELO score was the soft cross-entropy LogReg with mini-batch size of 2000 and learning rate 0.0031. When fixing the soft cross-entropy SL LogReg to 250, the best RL LogReg model achieved an ELO score at 748 with 95% confidence interval $[596, 1003]$. In comparison, less than 15% of the other model ELO scores were within this confidence interval.

To validate its strength, the model played an additional 200 matches against the soft cross-entropy SL LogReg model, where Stockfish would play both sides until the end after a total

of 200 moves by both players. Figure 7.1 shows that it consistently beats the untrained SL LogReg model after 10 training games, where its performance appears to converge.

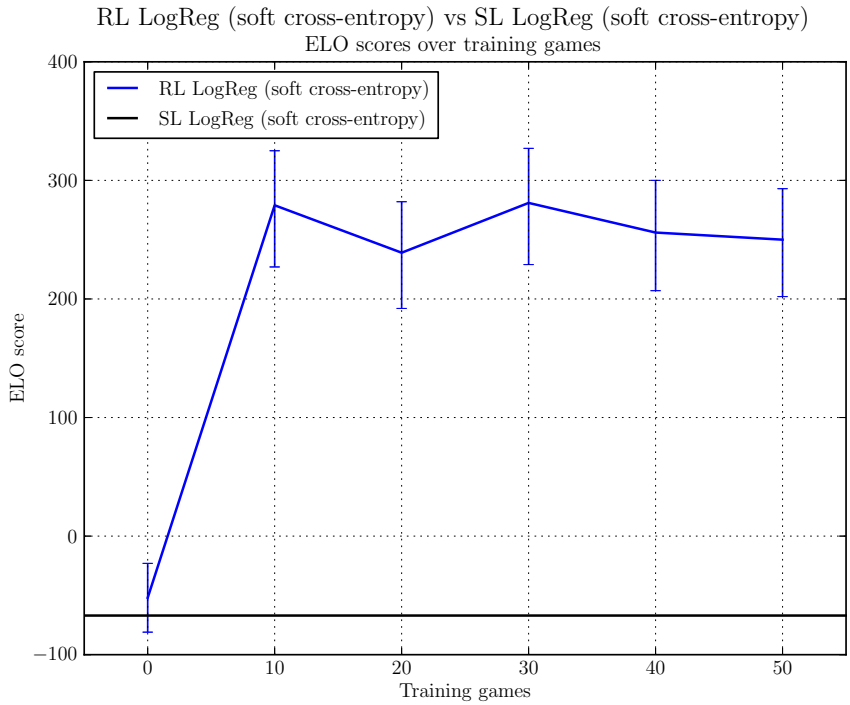



Figure 7.1: ELO scores with 95% confidence intervals for best performing soft cross-entropy RL LogReg w.r.t. training games. The models, selected at intervals of 10 training games, played 200 matches against the soft cross-entropy SL LogReg model. If the game had yet to finish after 200 moves, Stockfish configured to a 14-ply search depth would play both sides until the end. The scores are calculated w.r.t. fixing the score for the RL LogReg, trained over 50 games, to 250.

Over the 50 self-play games, the model parameters were updated in $24,696$ mini-batches which yields a total of $24,696 \times 2000 = 49,392,000$ board positions. This is a magnitude more than our supervised learning experiments. However, it is also magnitudes lower compared to the number of games played in (Veness et al., 2009), yet the playing performance seems to have converged already. Perhaps this may be explained by both the simple bitboard representation, and the initialization based on the supervised learning models.

## 7.6 Results

Based on preliminary experiments, we fixed $\mu_t = 1 - 3(t \times 10^{-4} - 5)$ where $t$ is the mini-batch number and trained MLP models with both squared error and soft cross-entropy to find suitable hyperparameters:

- mini-batch sizes in $\{500, 2000, 5000\}$,
- learning rates in $\{10^{-2}, 10^{-2.5}, \ldots, 10^{-4}, 10^{-4.5}\}$ for squared error,
- learning rates in $\{10^{2}, 10^{1.5}, \ldots, 10^{-1}, 10^{-1.5}\}$ for soft cross-entropy.

In this experiment all models managed to reach playing 50 self-play games after a fixed amount of computational time [9]. We therefore did not have to discard any models. We then played all RL MLPs trained, trained for 50 games, against the soft cross-entropy SL LogReg model in 40 matches (games) and, to reduce variance further, against the best performing soft cross-entropy RL LogReg model in 20 matches [10]. The best RL MLPs w.r.t. each type of objective function and each of the three initialization points were then determined.

The best RL MLPs then played 400 games against the soft cross-entropy SL LogReg model, trained over 50 games. As before, Stockfish would play both sides until the end of the game after 200 moves by both players in total.

[9] Perhaps this can be explained by the more restricted hyperparameter combinations.

[10] Due to node failure on the Legion Cluster, a handful of models did not play the total of 60 games. However, we do not believe to affect our results significantly.

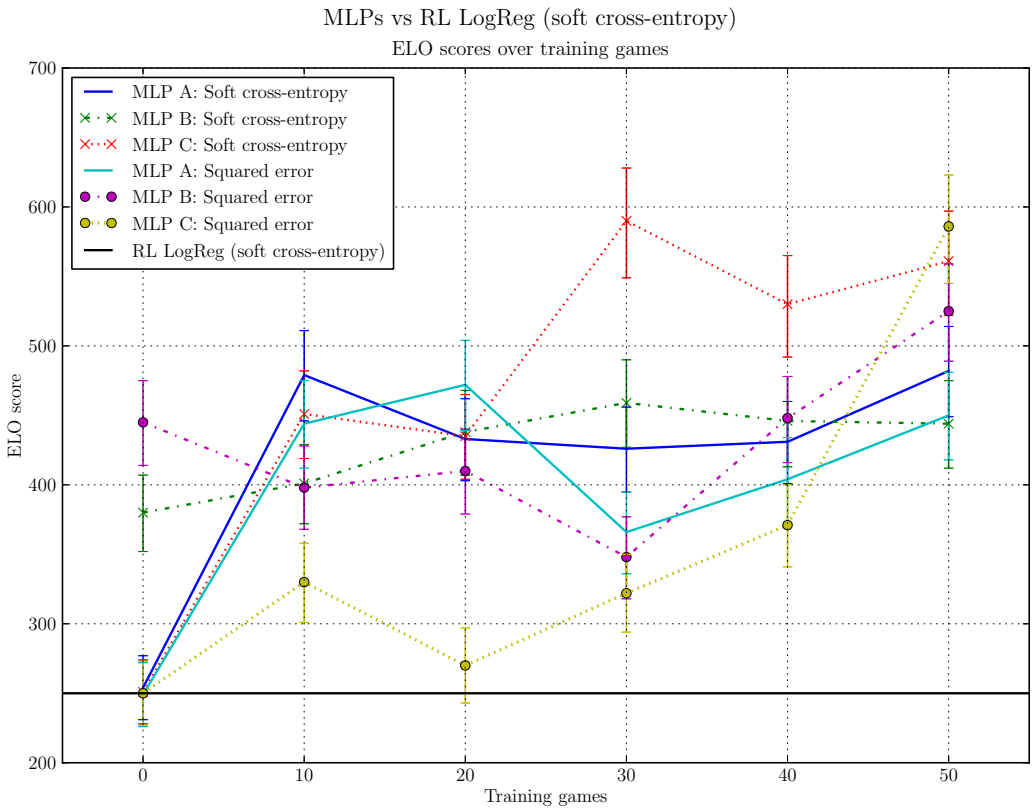



Figure 7.2: ELO scores with 95% confidence intervals for best performing RL MLPs w.r.t. training games. The models, selected at intervals of 10 training games, played 400 matches against the best performing soft cross-entropy RL LogReg model, which was trained for 50 games. The letters A, B and C refer to three different initialization points, where model A achieved the highest accuracy in the supervised learning experiments, B a medium accuracy and C one of the lowest accuracies. If a game had yet to finish after 200 moves, Stockfish configured to a 14-ply search depth would play both sides until the end. The scores are calculated w.r.t. fixing the score for the RL LogReg to 250.

As shown in figure 7.2, all MLPs are improving their playing strength significantly within the first 50 training games. Since the LogReg model appeared to converge, and since the MLPs were never trained against any LogReg model, we deduce that they have learned (non-linear)

features which are substantially better compared to weighting the piece squares linearly [11]. To the best of the authors knowledge, this is a novel finding. Previous research with a similar board representation in (Gherrity, 1993), and the other primitive representation in (Levinson and Weber, 2001). The rapid playing strength increase over only 50 training games also appears, at a first sight, to be favourable in comparison to the procedure suggested in (Thrun, 1995) [12].

The fact that the all MLPs are improving their playing strength significantly, and the majority of them without any significant decline at any point, over the 50 training games, suggests that the proposed non-linear modification of the TreeStrap algorithm, given in algorithm 7, is an effective self-play training procedure for MLP models. Furthermore, since even the poorest initialization points reached a relatively high playing strength, it also possible that the procedure is robust to poor initialization points to some extent.

Over the 50 self-play games, the parameters for the six best performing models were updated based on between $40,360,000$ and $73,217,500$ board positions. We did not observe any pattern between the final scores and the number of board positions the models had been trained on, even though some of the models were trained on twice as many board positions after 50, which might suggest that the variation in the games played are more important than the raw number of board positions. We speculate that inducing additional variation into the games, for example by selecting highly different opening positions, may benefit the training procedure. This would also encourage the models to learn features invariant to opening position, and avoid the potential problem we discussed in section 6.9: that the high correlation between the game outcome and pieces left immobile during the opening game, pieces which consequently remain stationary during most of the game, could lead model to primarily model learn features for recognizing the opening position.

It is not really possible to conclude if the models have converged or not. The models should preferably be trained for an additional few hundred games, and then played against each other in an all-versus-all tournament.

Unfortunately we cannot deduce whether the soft cross-entropy or the squared error objective function is to be preferred, but it does appear that larger mini-batches are more beneficial

[11] Strictly speaking we did select the MLPs based on their performance against SL LogReg and RL LogReg, but even before selecting them we saw that the majority of hyperparameter combinations for RL MLPs outperformed the SL LogReg model.

[12] Although we note that a very different feature set was used.

in conjunction with soft cross-entropy than with squared error.

The author observed that the soft cross-entropy RL MLP models, which were not trained with self-play, appeared subjectively to play very aggressively against the soft cross-entropy SL LogReg model. In particular, they would put their pieces under attack without any pieces defending them. However, the SL LogReg model was not always able to exploit this.

The same observation was made for some soft cross-entropy SL ConvNet models.

## 7.7 Discussion

As expected, we found that the supervised learning models without any self-play training played very weakly, w.r.t. playing strength measured in ELO. In particular, we observed that the self-play trained LogReg models outperformed their counterparts without self-play training. This can be explained by the fact that they are trained under the implicit assumption that a master-level chess player will continue the game from the given board position, and that only reasonable lines of play are considered.

Most importantly, we found that MLPs considerably outperform LogReg models with self-play training when initialized properly and trained for the same number of games. This strongly suggests that MLP models trained with the modified TreeStrap procedure, given in algorithm 7, are able to construct useful non-linear features which are better than their linear counterparts. More generally, the experiments demonstrate the potential of applying hierarchical models to learn the heuristic evaluation function for chess, and suggest that this is indeed promising line for further research in developing a chess program based on minimal expert knowledge.

Nevertheless, further experiments are necessary to evaluate the absolute performance of the models, for example, against human opposition on an internet chess server. Future experiments are also necessary to evaluate the performance of ConvNets with the self-play training procedure, and to investigate the effects of increasing the model size.

# Chapter 8

# Conclusion

We have attempted to develop a computer player for the game of chess, by automatically constructing the heuristic evaluation function, based on a collection of chess games and on self-play learning. In particular, we choose to use a simple binary representation of the chess board.

We first carried out a set of experiments on a collection of games played by master-level chess players, where we trained logistic regression, multilayer perceptron network, convolutional neural network and mixture of experts models with maximum likelihood to predict the winning player from a given board position. We found that the multilayer perceptron networks and convolutional neural networks outperform all other models on the same feature sets, and that they reached accuracies comparable to logistic regression on hand-crafted features. These results show that the hierarchical structure imposed by the multilayer perceptron networks and convolutional neural networks are appropriate for learning to predict the game outcome, and suggest that the same models may be effective as heuristic evaluation functions.

We further carried out a set of self-play experiments, where we implemented logistic regression and multilayer perceptron network models into a chess engine. A modification of a state-of-the-art self-play training algorithm was proposed, and used to learn a heuristic evaluation function for each model. In particular, each model was initialized based on the parameters found in the previous experiments on the game collection. The resulting programs were then played against each other, under a fixed search depth, to determine their playing strength. Based on these we found that the multilayer perceptron networks outperformed the logistic

regression models by a huge margin, when both were trained with the self-play procedure. This evidence strongly suggests that the multilayer perceptron networks are able to learn useful non-linear features, given only on a simple binary representation of the chess board, when trained in conjunction with the proposed modified self-play procedure and initialized properly.

## 8.1 Directions for Further Work

### 8.1.1 Self-Play experiments

The next step obvious step should be to train the MLP (multilayer perceptron network) models for hundreds of games more, while playing the MLP models against each other, to determine if and when the self-play procedure will converge. It is also very interesting to carry out a similar same experiment with ConvNet (convolutional network) models. It may also be necessary to further modify the self-play procedure to escape local solutions, for example by adding hidden units or feature maps as training progresses, and increase the learning rate.

Once convergence has been determined, or at least a sensible set of model parameters found, it would be natural to implement the models efficiently into a state-of-the-art chess engine, such as Stockfish. The program should then be played against human opposition, for example, on an internet chess server to determine its playing strength.

For initializing models used in the self-play experiments, Dayan has proposed to sample from a Laplace approximation (Dayan, 2014). He suggests to assume a uniform distribution over the model parameters found in the supervised learning experiment, and then approximate the posterior over the model parameters with a Laplace approximation. This is the same as a normal distribution with mean equal to the models found in the supervised learning experiments, and covariance matrix equal to the inverse of the Hessian matrix w.r.t. the objective function over the entire supervised learning training set. The parameters for the self-play models can then be drawn from the resulting normal distribution.

### 8.1.2 Model regularization

Since we observed a significant amount of overfitting in the supervised learning experiments, further investigation of regularization methods could potentially result in a considerable increase in performance. This would also be useful in self-play, because without regularization the models could easily overfit on games played while they are still weak and never acquire the features necessary to play at a stronger level.

It would be very interesting to investigate L1-regularization applied to ConvNet models both when learning from the game collection and when learning from self-play. These models are already translation invariant, and therefore adding L1-regularization will force each feature map to concentrate on a few squares and pieces.

It is also of great interest to design more structured regularization functions. The problem with L1-regularization, in particular for MLPs which do not impose any translation invariance, is that it penalizes each parameter independently. This works well in many domains, but for our problem the parameters are clearly highly dependent on each other. One way to address this issue would be to use a logistic function [1] on the sum of the absolute parameter values, excluding the bias, for each hidden unit. If the sum of absolute parameters of a hidden unit grows too large it will be penalized heavily, e.g. a large increase in log-likelihood would be required to justify integrating information over many squares in the same hidden unit, and otherwise the regularization will not affect the training procedure, e.g. the objective function remains unchanged for all hidden units working on a few or zero squares.

One could extend either the simple L1-regularization or any structured L1-regularization to include separate magnitudes for each piece type, e.g. by weighting the parameters for each piece type before the logistic function transformation. This would take into account the different frequencies with which each piece occurs on any square. For example, since pawns occur much more frequently in the vector of the bitboard representation, we should penalize parameters related to these much higher than the parameters related to the king.

[1] In this case, the logistic function is an approximation for the heavyside step function.

### 8.1.3 Performance metrics when learning from a game collection

In our supervised experiments, we have used the accuracy as the primary measure of model performance. As we have already discussed, accuracy is a simple and easy to interpret metric, but does not take into account several important aspects. Many other methods have been proposed to evaluate performance only based on a game collection, such as Pearson's correlation coefficient (van Rijswijck, 2001), geometric mean over the likelihood of board positions (Ban, 2012) and various methods taking into account the future moves taken by the (master) chess players, including the agreement between the model and the chess player's moves (David-Tabibi et al., 2009; Hoki and Kaneko, 2014) and the agreement in ranking different moves (Gomboc et al., 2004). We believe that further supervised learning experiments would benefit very much from also applying these metrics, in particular, an ordinal metric such as Kendall tau proposed in (Gomboc et al., 2004)

### 8.1.4 Alternative training procedures

It would be interesting to use comparison training in future supervised learning experiments, as it makes use of much more information in the games. We would hope not to see the same amount of overfitting in such a setup (Tesauro, 1989).

Because binary representation is discrete, it may be better to use the adapted variant of *rprop* named *rmsprop* for stochastic gradient descent instead of momentum with learning rate decay or Nesterov's accelerated gradient (Tieleman and Hinton, 2012, Lecture 6). This can be applied in both supervised learning and self-play learning experiments. This was also used in (Mnih et al., 2013). The *hessian free* optimization method may also be useful (Martens, 2010).

### 8.1.5 Input representation

Naturally it is possible to experiment with other alternative representations. However, given the positive results for the simple bitboard representation, the author believes it is most interesting to continue work with the bitboard representation and self-play learning experiments until it is found inadequate.

Nevertheless alternative representations may improve performance substantially. From preliminary experiments, we found some evidence that appending material counts for each piece type and color to be useful for MLPs. The square-based representation proposed in (Levinson and Weber, 2001) may also be useful.

### 8.1.6 Model architecture

Apart from MLPs and ConvNets, many other model architectures are possible. Due to the training difficulties involved with MLps, Veness suggests that using auto encoder networks will help improve the accuracy (Bengio, 2009; Veness, 2014b). In particular, he suggests to use a two-layered composition to first learn the relevant filters with one auto encoder, and then take these as input into another set of auto encoders. For example, there could be an auto encoder for each game outcome.

# Appendix A

# Why Monte Carlo Tree Search Methods are Inadequate for Chess

As we remarked in the section 1.2, *Monte Carlo tree search* (MCTS) methods have been successful in several domains (Browne et al., 2012; Lee et al., 2010). In particular, these methods have been able to improve the performance of algorithms in Computer Go by a considerable magnitude over the last few years (Gelly and Silver, 2008; Lee et al., 2010). This is interesting because Go is known to be extremely difficult for computers to play, due to a very high branching factorm i.e. the number of possible moves from any given position, a deep game tree and no known reliable heuristic evaluation function. Researchers have argued that this is the reason for the failure of heuristic search in Go, despite its success in chess (Schaeffer, 2000).

After reviewing the literature, we found that previous attempts to apply MCTS methods to chess had failed. In light of these, we were not able to propose any experiments with a reasonable chance of success (Barber, 2014; Dayan, 2014).

Ramanujan has carried out several experiments which demonstrate that one of the most widely applied MCTS procedures, UCT, is inferior to standard minimax search in chess (Ramanujan, 2012; Ramanujan et al., 2010). He argued that the failure is due to the many *traps* that exist in chess games, but which do exist in, for example, Go. A trap is a move by the current player from where there exists a sequence of $K$ moves, which the opponent can force which will guarantee the opponent to win the game. Since MCTS methods are based on a stochastic sampling schemes, they are unable to identify these as frequently as minimax search.

Therefore, a program playing with a minimax search (or a shrewd player) will be able to exploit the traps and win against the MCTS based methods. To reach this conclusion, Ramanujan carefully tested out this hypothesis on a set of synthetic games and another game, Mancala, which shares some of the trap characteristics of chess. Despite these arguments, Oleg experimented with UCT and several variants of it, including the new UCT-RAVE method proposed in (Gelly and Silver, 2008), applied to chess in his undergraduate thesis under the supervision of Fürnkranz. His experiments reached the same conclusion as Ramanujan: that UCT is inferior to minimax search.

Nevertheless, these experiments do not necessarily imply that it is impossible to make progress in this direction. For example, Ramanujan suggests that a hybrid approach combining MCTS and minimax search could outperform minimax search alone (Ramanujan, 2012, pp. 82–85). Complementary to this, Silver also notes that some of the best chess engines today use MCTS to estimate node values when the leaf nodes of the minimax search tree are deemed to be inaccurate (Silver, 2014b). Some papers propose hybrid approaches, which could be applied to chess and compared to standard UCT and minimax algorithms, such as (Lanctot et al., 2013). We suspect that this would outperform either single approach, but that in itself might not be an interesting result as we should consider hybrid approaches to strictly be a superclass of each approach.

As we motivated in the beginning of the thesis, we are primarily interested in minimizing the need for expert knowledge in constructing the heuristic evaluation function for chess. This is why we finally chose to focus on learning features for chess automatically.

# Appendix B

# A Theoretical Interpretation of Feature Representations

Ghory was perhaps the first to develop a framework for evaluating feature representations of board games from a theoretical viewpoint (Ghory, 2004, pp. 15–20). He defines the two measures *smoothness* and *divergence rate*.

Let $A$ and $B$ be vectors representing two game positions, and let $||A - B||^2$ represent the *distance* between them. For example, if the vectors are real-valued this could be the standard L2 norm. A representation is smooth if $||A-B||^2$ is small implies that $|g(A)-g(B)|$ is small, where $g(X)$ is the expected pay-off from position $X$, equivalent to the game theoretical value returned by an exhaustive minimax search from $X$. The smoothness of a representation is loosely defined as the average degree of smoothness between any two distinct valid game positions. Ghory argues that the ability of an MLP with a sigmoid activation function to learn a heuristic evaluation function is closely correlated to the smoothness of the input representation. When the the representation is very smooth, changing a feature value slightly should only change the expected pay-off slightly. In particular, this implies that a linear (or higher-order polynomial) approximation at any point will be very accurate around this point.

Next, let us as assume that the expected pay-off in a game changes very little w.r.t. each move taken, e.g. it would require many blunders to loose the game from any given position [1].

[1] Obviously, this assumption is invalid for chess. For example, in a single move a player can sacrifice his queen by putting it under attack at an undefended square.

Let $A_1, A_2, \ldots, A_n$ be distinct game positions reachable from position $A$ in one ply. We then define the divergence of $A$ as:

$$d(A) = \frac{\sum ||A_1 - A_2||^2}{2n}. \tag{B.1}$$

The average divergence rate for a representation is then defined as the average $d(A)$ over all valid game positions $A$. Since the expected pay-off value changes only slowly with each move, we would prefer a representation with as small divergence rate as possible. For example, take tic-tac-toe with a trinary representation, i.e. a vector of nine values corresponding to each of the nine fields where each takes value zero if its corresponding square is empty, always has a divergence rate of one. However, since the average divergence rate does not take into account the true expected pay-off at any point, this is a much weaker justification for a representation than the one motivating smooth representations.

# Appendix C

# Theano A Compiler for Symbolic Mathematical Expressions

We use the open-source library Theano in our project (Bergstra et al., 2010) [1]. Theano is a compiler for mathematical expressions in Python, which is able handle many computations efficiently by representing them in a mathematical symbolic language similar to the python library *NumPy*. This allows Theano to perform symbolic differentiation while storing reoccurring computations, such as the intermediate values required for backpropagation described in section 3.2.3, efficiently in memory. At runtime Theano compiles the symbolic expressions to fast C++ code either for CPU processsing or for GPU processing with CUDA [2]. This enables the library to train models an order of magnitude faster than competing libraries, while representing all models in a simple abstract mathematical language.

Note that due to technical difficulties on UCL Legion High Performance Computing Facility, Theano was only used with CPU processing. However, the loss in performance should not be very high. The bitboard representations we used are $8 \times 8$ matrices, which limits the advantage of the GPU speed-up when compared to the larger $256 \times 256$ matrices benchmarked in (Bergstra et al., 2010).

[1] Theano can be downloaded from the website `http://deeplearning.net/software/theano/`.

[2] Compute Unified Device Architecture (CUDA) is a parallel computing platform, which allows developers to directly access graphical processing unit (GPU) instructions. In short, this allows us to take advantage of highly specialized graphic cards hardware to speed up computations. See `http://www.nvidia.com/object/cuda_home_new.html`.

# Appendix D

# Supervised Experiments

## D.1 Regularized LogReg Experiments

Per suggestion from (Dayan, 2014), we also trained a LogReg model with L1-regularization. We applied only a tiny bit of regularization, with the motivation to extract features which suffer less from *clutter* induced by certain positions which occur extremely rarely. For example, suppose the White king was found in the square A8, which would be indeed be rare, in only one or two games in the entire game collection, and furthermore that White won in both games. Then the standard LogReg will immediately attribute a high score to White king on A8, while the regularized one might choose to attribute it a lower score (though still positive score) in favour of attributing a higher score to other bits. The method also naturally removes noise caused by positions which cannot occur, e.g. White pawns on flank one. Unlike the experiments presented in section 6.8, we also normalized the bitboards to have mean zero.

**L1-Regularized LogReg with Bitboard Features**

Mean accuracy: 57.35%

| **True outcome** \ **Predicted outcome** | Black wins | Draw | White wins |
|---|---|---|---|
| Black wins | 239,146 | 66,950 | 115,363 |
| Draw | 75,487 | 147,440 | 93,889 |
| White wins | 89,781 | 61,968 | 290,327 |

Table D.1: L1-Regularized LogReg model with Bitboard Features. The table shows the mean accuracy and contingency table of the model. Model was trained on 4,298,389 instances and tested on 1,180,471 instances. The L1-regularization parameter was set to $10^{-8}$, which yielded the same accuracy as the non-regularized LogReg model, but was still numerically stable.

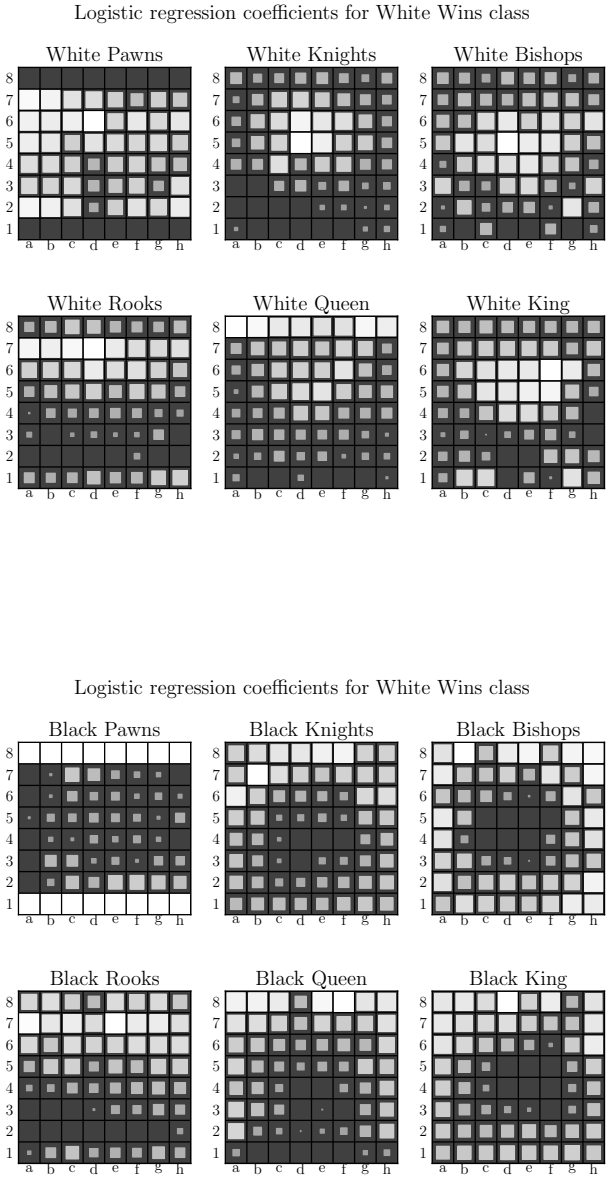



Figure D.1: LogReg coefficients for White wins class. To make interpretation easier, the coefficients were transformed linearly to be in the interval [0, 1]. The values should only be compared within each piece type, where larger and whiter squares correspond to an increased probability of winning the game for White.

## D.2 ConvNet Experiments

The table below shows the mean accuracy on the training set.

The graphs below show the mean accuracy and standard deviations for other ConvNet models against training epoch. This gives a more clear picture of how overfitting emerges as the number of feature maps are increased and the receptive fields are made smaller. Observe also how early-stopping selects a good model by choosing the model with the highest accuracy on the validation set. This is also the reason why we are not seen significant differences in accuracy on the training set between small and large models in table D.2.

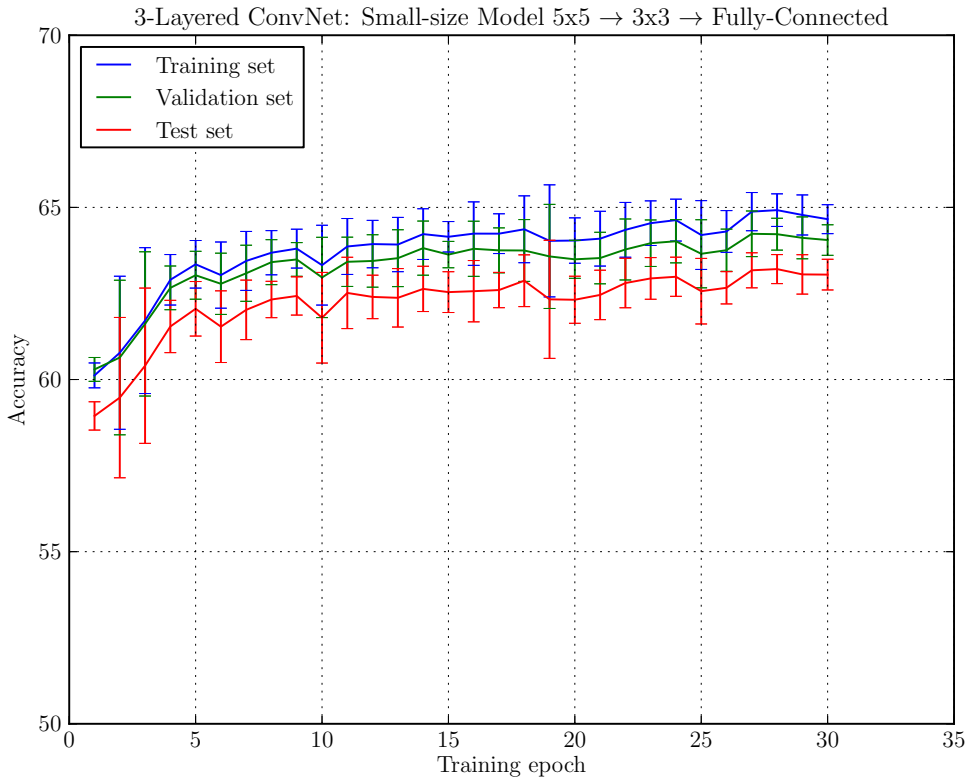



Figure D.2: Mean accuracies and standard deviations over 10 trials for 3-layered $5 \times 5 \rightarrow 3 \times 3$ ConvNet, with 5, 10 and 100 feature maps in layer one, layer two and layer three respectively, plotted against training epoch.

| ConvNet | Configuration | Mean accuracy |
|---|---|---|
| Bitboard Features | • $3 \times 3$ pooling with 5 features in first layer<br>• $3 \times 3$ pooling with 10 features in second layer<br>• 100 hidden units in third layer | $64.36\% \pm 0.92$ |
| Bitboard Features | • $3 \times 3$ pooling with 5 features in first layer<br>• $3 \times 3$ pooling with 10 features in second layer<br>• 200 hidden units in third layer | $65.48\% \pm 0.76$ |
| Bitboard Features | • $3 \times 3$ pooling with 10 features in first layer<br>• $3 \times 3$ pooling with 20 features in second layer<br>• 100 hidden units in third layer | $65.28\% \pm 0.96$ |
| Bitboard Features | • $3 \times 3$ pooling with 10 features in first layer<br>• $3 \times 3$ pooling with 20 features in second layer<br>• 200 hidden units in third layer | $65.52\% \pm 0.62$ |
| Bitboard Features | • $3 \times 3$ pooling with 20 features in first layer<br>• $3 \times 3$ pooling with 40 features in second layer<br>• 100 hidden units in third layer | $64.52\% \pm 1.00$ |
| Bitboard Features | • $3 \times 3$ pooling with 20 features in first layer<br>• $3 \times 3$ pooling with 40 features in second layer<br>• 200 hidden units in third layer | $64.74\% \pm 1.10$ |
| Bitboard Features | • $5 \times 5$ pooling with 5 features in first layer<br>• $3 \times 3$ pooling with 10 features in second layer<br>• 100 hidden units in third layer | $64.32\% \pm 0.99$ |
| Bitboard Features | • $5 \times 5$ pooling with 5 features in first layer<br>• $3 \times 3$ pooling with 10 features in second layer<br>• 200 hidden units in third layer | $64.65\% \pm 0.67$ |
| Bitboard Features | • $5 \times 5$ pooling with 10 features in first layer<br>• $3 \times 3$ pooling with 20 features in second layer<br>• 100 hidden units in third layer | $65.62\% \pm 0.81$ |
| Bitboard Features | • $5 \times 5$ pooling with 10 features in first layer<br>• $3 \times 3$ pooling with 20 features in second layer<br>• 200 hidden units in third layer | $65.69\% \pm 0.76$ |
| Bitboard Features | • $5 \times 5$ pooling with 20 features in first layer<br>• $3 \times 3$ pooling with 40 features in second layer<br>• 100 hidden units in third layer | $65.55\% \pm 0.97$ |
| Bitboard Features | • $5 \times 5$ pooling with 20 features in first layer<br>• $3 \times 3$ pooling with 40 features in second layer<br>• 200 hidden units in third layer | $66.00\% \pm 0.57$ |

Table D.2: Mean accuracies for ConvNet models on training set, with two pooling layers and a fully-connected layer at the end, taking Bitboard Features as input over 10 trials. The term *features* in the configuration refers to the number of feature maps at each level of the network.

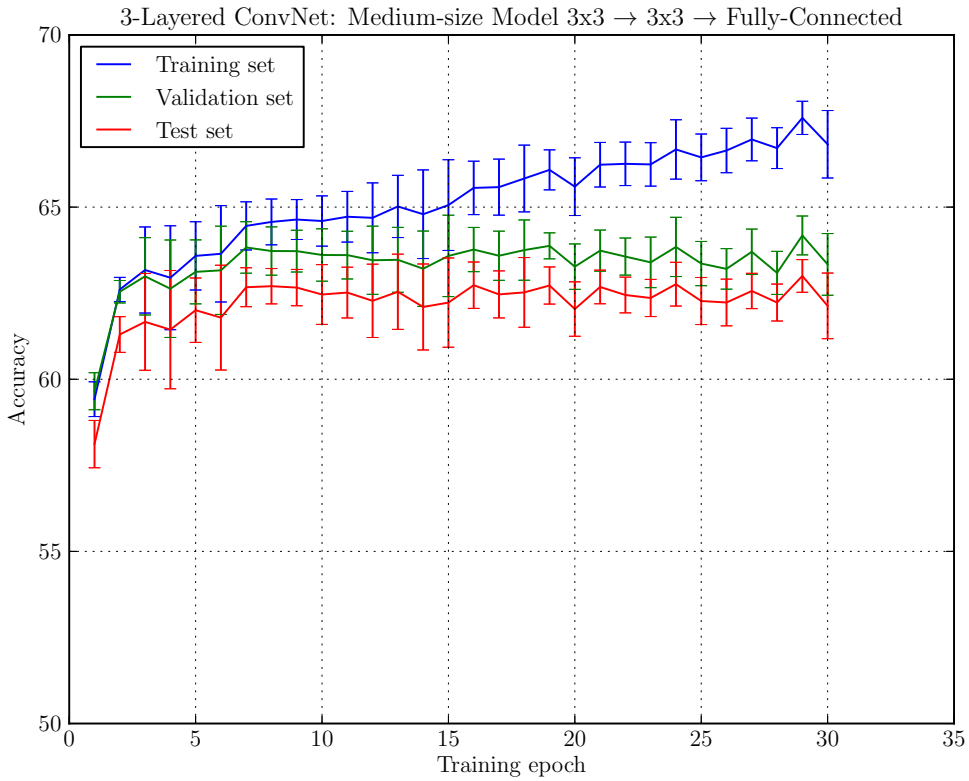



Figure D.3: Mean accuracies and standard deviations over 10 trials for 3-layered $3 \times 3 \rightarrow 3 \times 3$ ConvNet, with 10, 20 and 100 feature maps in layer one, layer two and layer three respectively, plotted against training epoch.

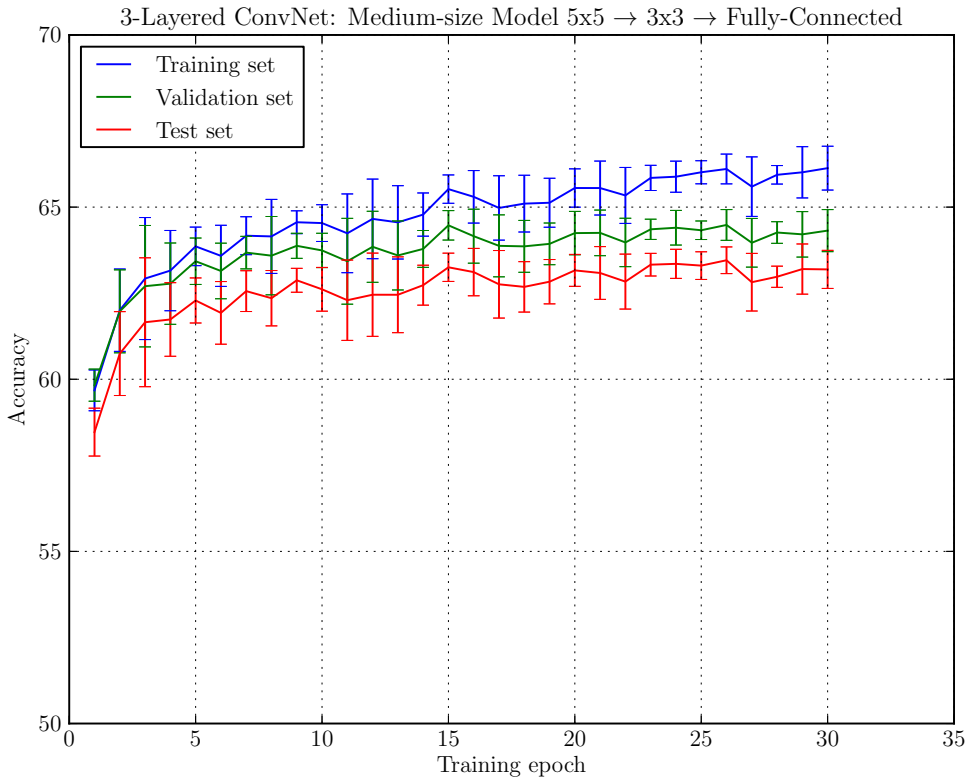



Figure D.4: Mean accuracies and standard deviations over 10 trials for 3-layered $5 \times 5 \rightarrow 3 \times 3$ ConvNet, with 10, 20 and 100 feature maps in layer one, layer two and layer three respectively, plotted against training epoch.

The graphs below show the mean negative log-likelihood and standard deviations for all ConvNet models against training epoch. Overfitting occurs here as well, which confirms that the overfitting is not due to using accuracy as a performance criteria.

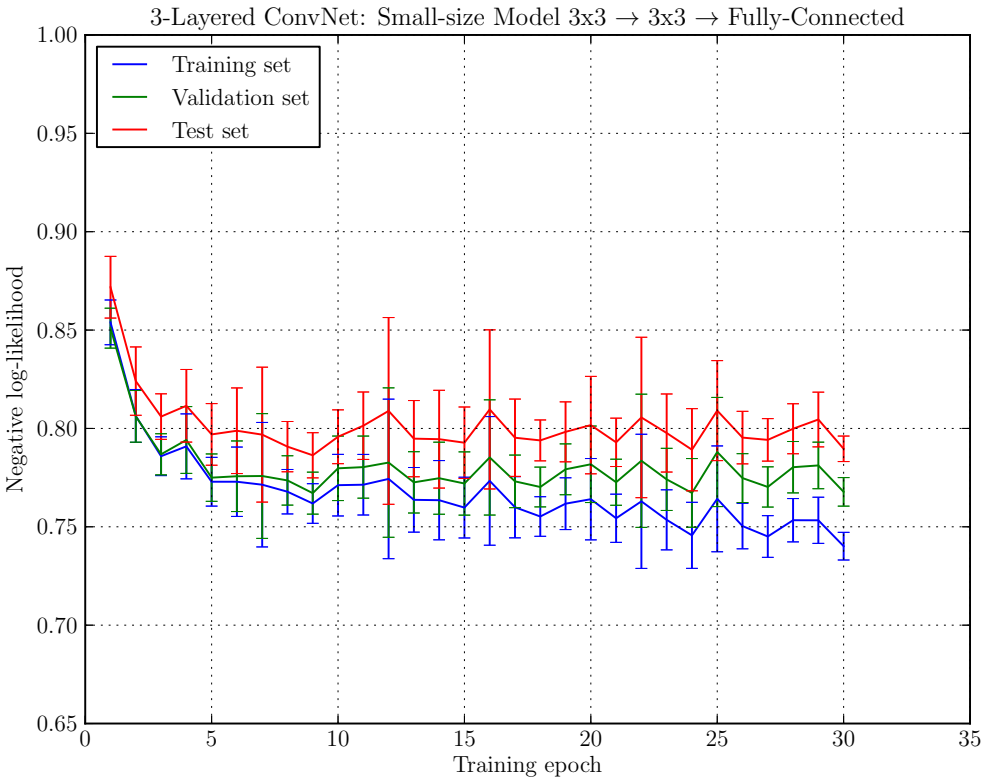



Figure D.5: Negative log-likelihood and standard deviations over 10 trials for 3-layered $3 \times 3 \rightarrow 3 \times 3$ ConvNet, with 5, 10 and 100 feature maps in layer one, layer two and layer three respectively, plotted against training epoch.

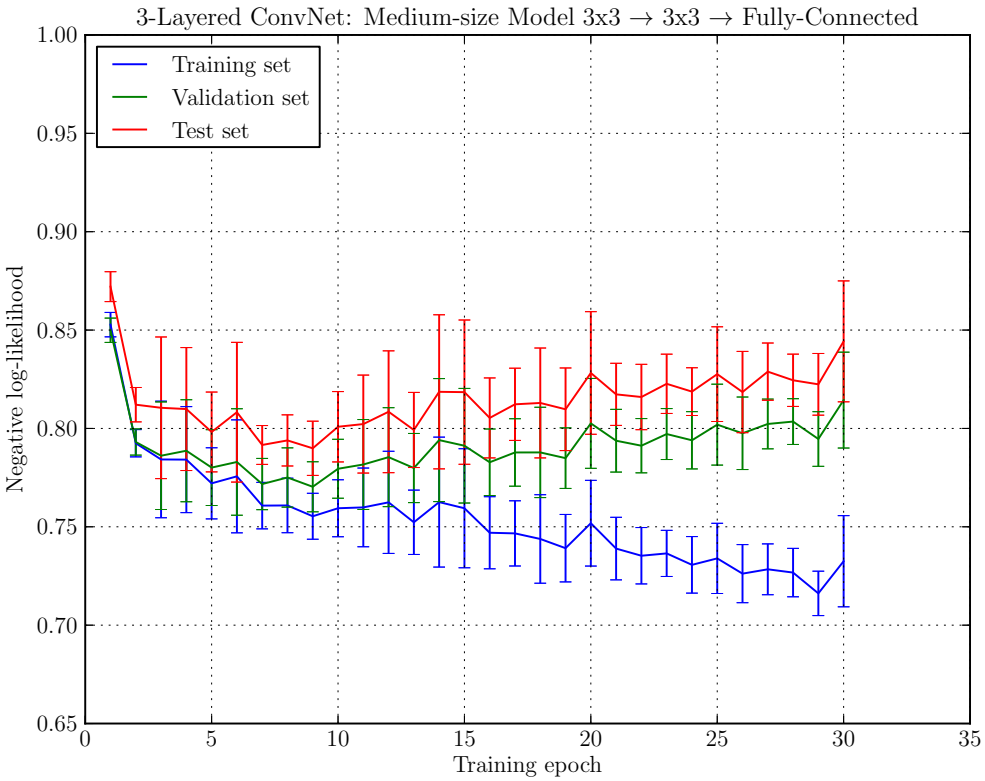



Figure D.6: Negative log-likelihood and standard deviations over 10 trials for 3-layered $3 \times 3 \rightarrow 3 \times 3$ ConvNet, with 10, 20 and 100 feature maps in layer one, layer two and layer three respectively, plotted against training epoch.

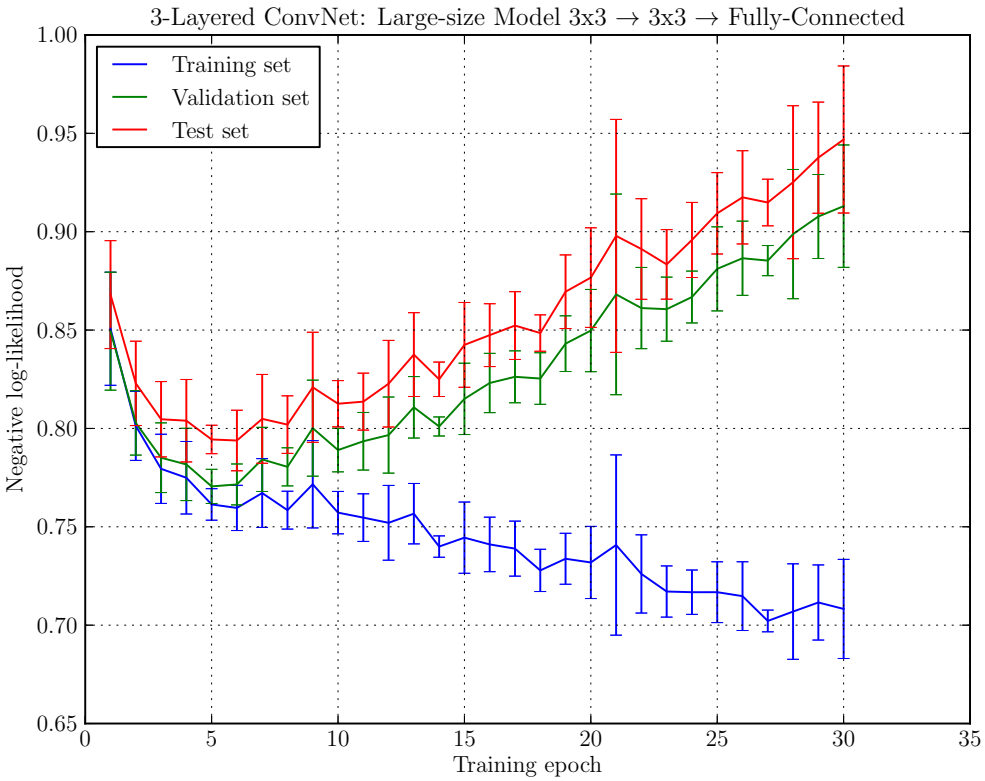



Figure D.7: Negative log-likelihood and standard deviations over 10 trials for 3-layered $3 \times 3 \rightarrow 3 \times 3$ ConvNet, with 20, 40 and 200 feature maps in layer one, layer two and layer three respectively, plotted against training epoch.

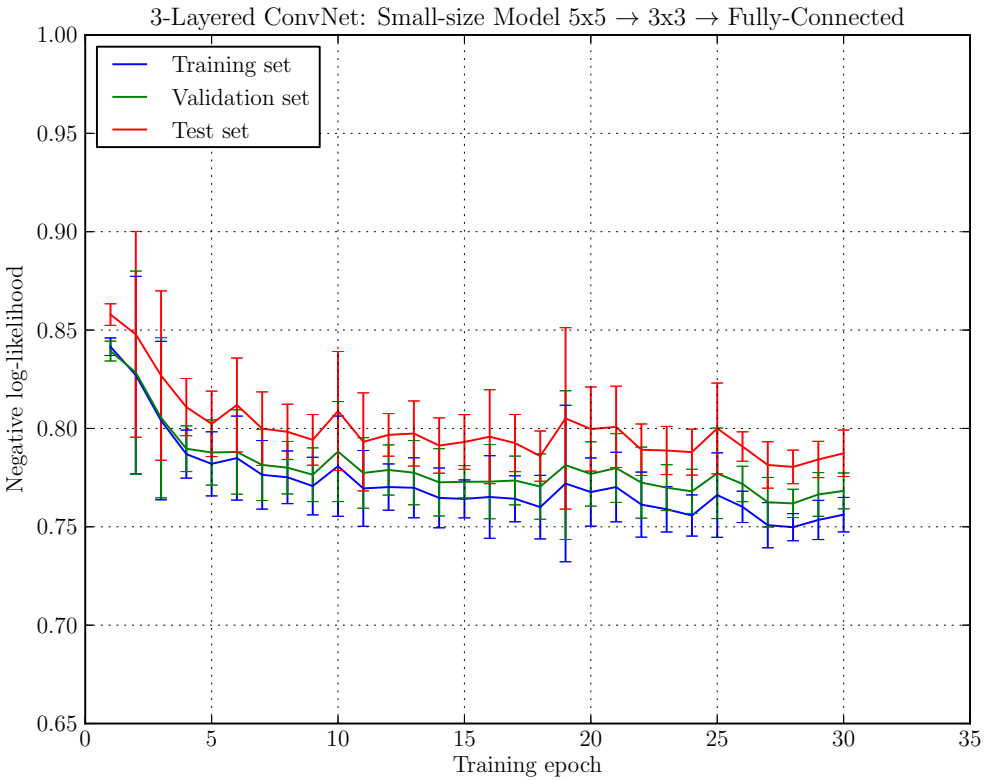



Figure D.8: Negative log-likelihood and standard deviations over 10 trials for 3-layered $5 \times 5 \rightarrow 3 \times 3$ ConvNet, with 5, 10 and 100 feature maps in layer one, layer two and layer three respectively, plotted against training epoch.

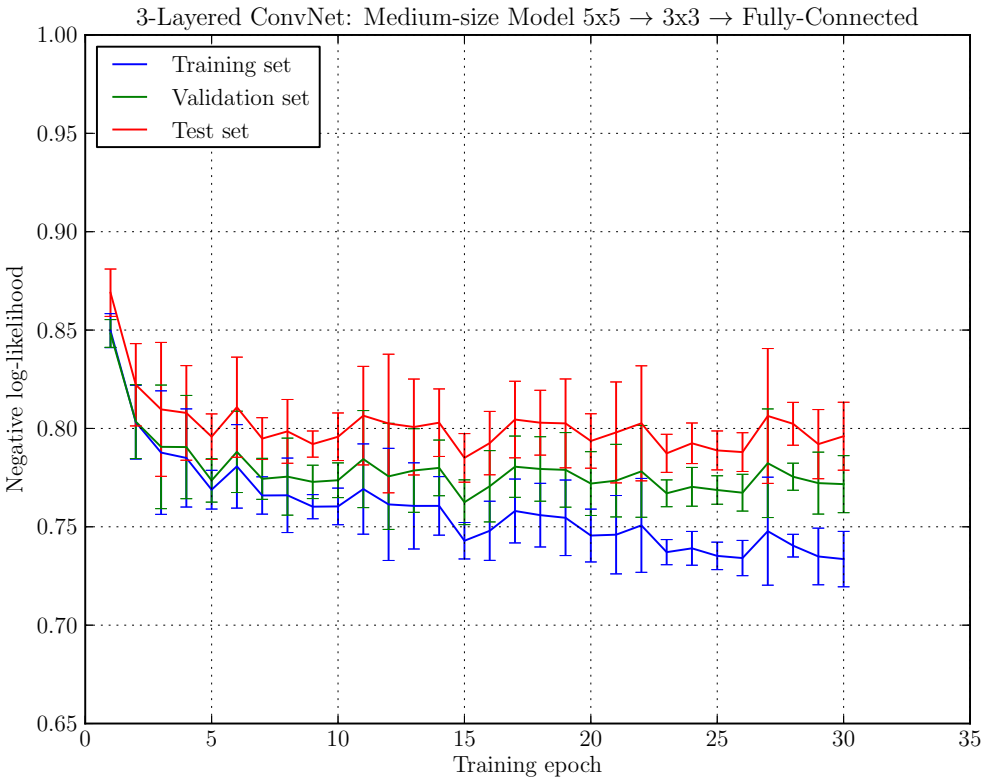



Figure D.9: Negative log-likelihood and standard deviations over 10 trials for 3-layered $5 \times 5 \rightarrow 3 \times 3$ ConvNet, with 10, 20 and 100 feature maps in layer one, layer two and layer three respectively, plotted against training epoch.

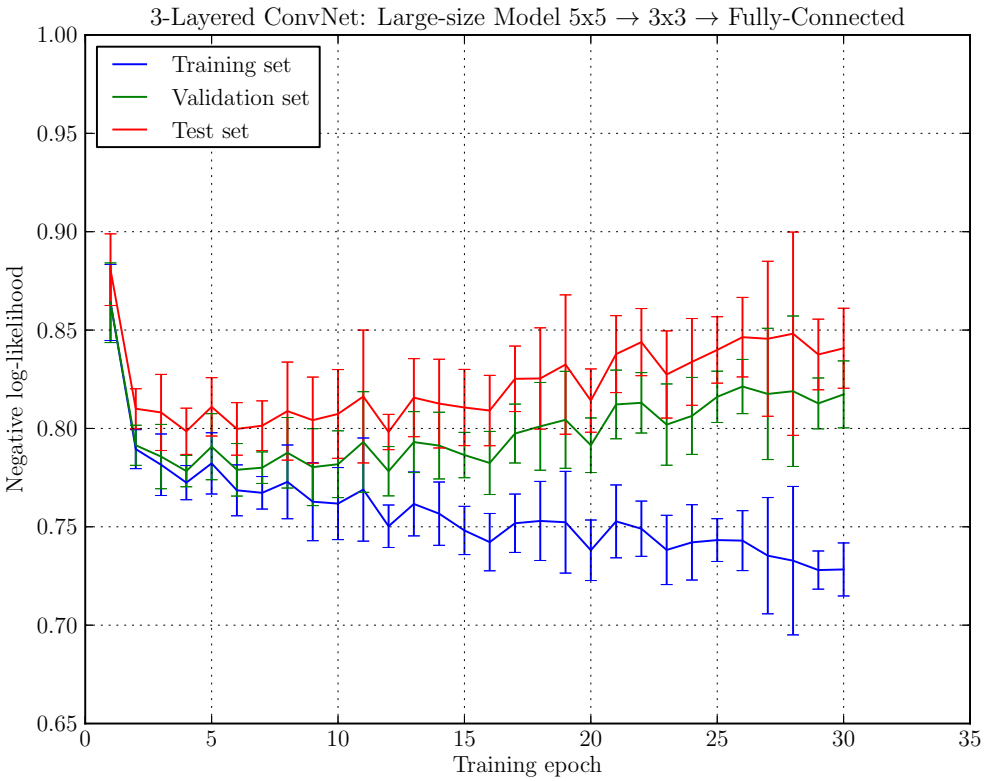



Figure D.10: Negative log-likelihood and standard deviations over 10 trials for 3-layered $5 \times 5 \rightarrow 3 \times 3$ ConvNet, with 20, 40 and 200 feature maps in layer one, layer two and layer three respectively, plotted against training epoch.

# Bibliography


Chess informant, publisher of the encyclopedia of chess openings, 2014. URL `http://www.chessinformant.rs/eco-encyclopedia-of-chess-openings`. Retrieved 30 July 2014.

O. Arenz. Monte carlo chess. Master's thesis, Technische Universität Darmstadt, April 2012. Undergraduate thesis supervised by Johannes Fürnkranz carried out at Fachbereich Informatik.

A. Ban. Automatic Learning of Evaluation, with Applications to Computer Chess. Technical report, Hebrew University of Jerusalem, 2012. Discussion Paper: 613.

D. Barber. Private communication, 2014. Email correspondence and meetings were between February 2nd and September 3rd.

J. Baxter, A. Tridgell, and L. Weaver. Learning to play chess using temporal differences. *Machine Learning*, 40(3):243–263, 2000.

D. F. Beal. A generalised quiescence search algorithm. *Artificial Intelligence*, 43(1):85–98, 1990.

D. F. Beal and M. C. Smith. Temporal difference learning for heuristic search and game playing. *Information Sciences*, 122(1):3–21, 2000.

Y. Bengio. Learning deep architectures for AI. *Foundations and trends® in Machine Learning*, 2(1):1–127, 2009.

Y. Bengio. Practical recommendations for gradient-based training of deep architectures. In *Neural Networks: Tricks of the Trade*, pages 437–478. Springer, 2012.

Y. Bengio. Deep learning of representations: Looking forward. In *Statistical Language and Speech Processing*, pages 1–37. Springer, 2013.

J. Bergstra, O. Breuleux, F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, J. Turian, D. Warde-Farley, and Y. Bengio. Theano: a CPU and GPU math expression compiler. In *Proceedings of the Python for Scientific Computing Conference (SciPy)*, June 2010. Oral Presentation.

D. Billings, D. Papp, J. Schaeffer, and D. Szafron. Opponent modeling in poker. In *AAAI/IAAI*, pages 493–499, 1998.

C. M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006. ISBN 0387310738.

L. Bottou. Stochastic gradient descent tricks. In *Neural Networks: Tricks of the Trade*, pages 421–436. Springer, 2012.

C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton. A survey of monte carlo tree search methods. *Computational Intelligence and AI in Games, IEEE Transactions on*, 4(1):1–43, 2012.

M. Campbell, A. J. Hoane Jr, and F.-h. Hsu. Deep blue. *Artificial intelligence*, 134(1):57–83, 2002.

W. G. Chase and H. A. Simon. Perception in chess. *Cognitive psychology*, 4(1):55–81, 1973.

S. Chinchalkar. An upper bound for the number of reachable positions. *ICCA JOURNAL*, 19 (3):181–183, 1996.

D. Claudiu Ciresan, U. Meier, L. M. Gambardella, and J. Schmidhuber. Deep big simple neural nets excel on handwritten digit recognition. *arXiv preprint arXiv:1003.0358*, 2010.

R. Coulom. Bayeselo: Bayesian elo rating, 2014. URL `http://remi.coulom.free.fr/Bayesian-Elo/`. Retrieved 2 September 2014.

O. David-Tabibi, H. J. Van Den Herik, M. Koppel, and N. S. Netanyahu. Simulating human grandmasters: evolution and coevolution of evaluation functions. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 1483–1490. ACM, 2009.

P. Dayan. Private communication, 2014. Email correspondence and meetings were between February 2nd and September 3rd.

J. Duchi, E. Hazan, and Y. Singer. Adaptive subgradient methods for online learning and stochastic optimization. *The Journal of Machine Learning Research*, 12:2121–2159, 2011.

A. E. Elo. *The rating of chessplayers, past and present*, volume 3. Batsford London, 1978.

K. Fukushima. Neocognitron: A hierarchical neural network capable of visual pattern recognition. *Neural networks*, 1(2):119–130, 1988.

J. Fürnkranz. Machine learning in games: A survey. *Machines that learn to Play Games*, pages 11–59, 2001.

J. Fürnkranz. Recent advances in machine learning and game playing. *ÖGAI Journal*, 26(2): 19–28, 2007.

J. Fürnkranz. Knowledge discovery in chess databases: A research proposal. Technical report, Austrian Research Institute for Artificial Intelligence, 1997.

L. Galway, D. Charles, and M. Black. Machine learning in digital games: a survey. *Artificial Intelligence Review*, 29(2):123–161, 2008.

S. Gelly and D. Silver. Achieving master level play in 9 x 9 computer go. In *AAAI*, volume 8, pages 1537–1540, 2008.

M. Gherrity. *A game-learning machine*. PhD thesis, Citeseer, 1993.

I. Ghory. Reinforcement learning in board games. Technical Report CSTR-04-004, Department of Computer Science, University of Bristol, 2004.

P. Golik, P. Doetsch, and H. Ney. Cross-entropy vs. squared error training: a theoretical and experimental comparison. In *INTERSPEECH*, pages 1756–1760, 2013.

D. Gomboc, T. A. Marsland, and M. Buro. Evaluation function tuning via ordinal correlation. In *Advances in Computer Games*, pages 1–18. Springer, 2004.

T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer, second edition, 2009.

M. Herbster and M. Pontil. Lecture notes for the course Supervised Learning taught by Mark Herbster and Massimiliano Pontil at the Computer Science department at University College London, autumn term, 2013. See `http://www0.cs.ucl.ac.uk/staff/M.Herbster/GI01/`., 2013.

R. D. Hof. Deep Learning, 2014. URL `http://www.technologyreview.com/featuredstory/513696/deep-learning/`. Retrieved 04 July 2014.

K. Hoki and T. Kaneko. Large-scale optimization for evaluation functions with minimax search. *J. Artif. Intell. Res.(JAIR)*, 49:527–568, 2014.

F.-H. Hsu. *Behind Deep Blue: Building the computer that defeated the world chess champion*. Princeton University Press, 2002.

G.-B. Huang, L. Chen, and C.-K. Siew. Universal approximation using incremental constructive feedforward networks with random hidden nodes. *Neural Networks, IEEE Transactions on*, 17(4):879–892, 2006.

D. R. Hunter. Mm algorithms for generalized bradley-terry models. *Annals of Statistics*, pages 384–406, 2004.

R. A. Jacobs, M. I. Jordan, S. J. Nowlan, and G. E. Hinton. Adaptive mixtures of local experts. *Neural computation*, 3(1):79–87, 1991.

M. I. Jordan and R. A. Jacobs. Hierarchical mixtures of experts and the em algorithm. *Neural computation*, 6(2):181–214, 1994.

M. I. Jordan and L. Xu. Convergence results for the em approach to mixtures of experts architectures. *Neural networks*, 8(9):1409–1431, 1995.

P. Krafft. Applying deep belief networks to the game of go. Master's thesis, Computer Science Department, Massachusetts Institute of Technology, 2010. Undergraduate thesis supervised by Andrew Barto and George Konidaris.

M. Lanctot, A. Saffidine, J. Veness, C. Archibald, and M. H. M. Winands. Monte Carlo *-Minimax Search. *ArXiv e-prints*, Apr. 2013.

S. Lawrence, C. L. Giles, and A. C. Tsoi. What size neural network gives optimal generalization? Convergence properties of backpropagation. Technical report, NEC Research Institute and University of Queensland , 1996.

Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4): 541–551, 1989.

Y. A. LeCun, L. Bottou, G. B. Orr, and K.-R. Müller. Efficient backprop. In *Neural networks: Tricks of the trade*, pages 9–48. Springer, 2012.

C.-S. Lee, M. Müller, and O. Teytaud. Special issue on monte carlo techniques and computer go. *Computational Intelligence and AI in Games, IEEE Transactions on*, 2(4):225–228, Dec 2010. ISSN 1943-068X. doi: 10.1109/TCIAIG.2010.2099154.

K.-F. Lee and S. Mahajan. A pattern classification approach to evaluation function learning. *Artificial Intelligence*, 36(1):1–25, 1988.

R. Levinson and R. Weber. Chess neighborhoods, function combination, and reinforcement learning. In *Computers and Games*, pages 133–150. Springer, 2001.

J. Mańdziuk. Human-like intuitive playing in board games. In *Neural Information Processing*, pages 282–289. Springer, 2012.

H. Mannen. Learning to play chess using reinforcement learning with database games. Master's thesis, Utrecht University, 2003. Graduate thesis in Cognitive Artificial Intelligence supervised by Marco Wiering.

J. Martens. Deep learning via hessian-free optimization. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pages 735–742, 2010.

H. A. Mayer. Board representations for neural go players learning by temporal difference. In *Computational Intelligence and Games, 2007. CIG 2007. IEEE Symposium on*, pages 183–188. IEEE, 2007.

V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

Y. Nesterov. A method of solving a convex programming problem with convergence rate o (1/k2). *Soviet Mathematics Doklady*, 27(2):372–376, 1983.

A. Newell, J. Shaw, and H. A. Simon. Chess-playing programs and the problem of complexity. In *Computer Games I*, pages 89–115. Springer, 1988.

J. Nocedal and S. Wright. *Numerical optimization*. Springer series in operations research and financial engineering. Springer, 2. ed. edition, 2006.

S. J. Prince. *Computer vision: models, learning, and inference*. Cambridge University Press, 2012.

R. Ramanujan. *Understanding Sampling-Based Adversarial Search Methods*. PhD thesis, Cornell University, August 2012. Ph.D. thesis carried out at Faculty of the Graduate School.

R. Ramanujan, A. Sabharwal, and B. Selman. On adversarial search spaces and sampling-based planning. In *ICAPS*, volume 10, pages 242–245, 2010.

P. I. Richards. Machines which can learn. *American Scientist*, pages 711–716, 1951.

T. Romstad, M. Costalba, and J. Kiiski. Stockfish Chess Engine, 2014. URL `http://stockfishchess.org`. Retrieved 03 June 2014.

S. J. Russell and P. Norvig. *Artificial intelligence: a modern approach (3rd edition)*. Prentice Hall, 2009.

M. Sahani. Lecture notes for the courses Probabilistic and Unsupervised Learning and Approximate Inference and Learning in Probabilistic Models taught by Maneesh Sahani at Gatsby Computational Neuroscience Unit at University College London, autumn term, 2013. See `http://www.gatsby.ucl.ac.uk/teaching/courses/ml1-2013.html`., 2013.

A. L. Samuel. Some studies in machine learning using the game of checkers. *IBM J. Res. Dev.*, 3(3):210–229, July 1959. ISSN 0018-8646. doi: 10.1147/rd.33.0210. URL `http://dx.doi.org/10.1147/rd.33.0210`.

A. L. Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of research and development*, 44(1.2):206–226, 2000.

J. Schaeffer. The role of games in understanding computational intelligence. In *Artificial Intelligence*. Citeseer, 1999.

J. Schaeffer. The games computers (and people) play. *Advances in computers*, 52:189–266, 2000.

M. Schmidt, G. Fung, and R. Rosales. Fast optimization methods for l1 regularization: A comparative study and two new approaches. In *Machine Learning: ECML 2007*, pages 286–297. Springer, 2007.

N. N. Schraudolph, P. Dayan, and T. J. Sejnowski. Learning to evaluate go positions via temporal difference methods. In *Computational Intelligence in Games*, pages 77–98. Springer, 2001.

Seberg and Ludens. FICS Games Database, 2014a. URL `http://www.ficsgames.org`. Retrieved 03 June 2014.

Seberg and Ludens. FICS description of lightning games, 2014b. URL `http://www.freechess.org/Help/ficsfaq.html#Q005.012`. Retrieved 19 June 2014.

C. E. Shannon. Xxii. programming a computer for playing chess. *Philosophical magazine*, 41 (314):256–275, 1950.

D. Silver. Lecture notes for the course Advanced Topics in Machine Learning: Reinforcement Learning taught by David Silver at the Computer Science department at University College London, spring term, 2014. See `http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching.html`., 2014a.

D. Silver. Private communication, 2014b. Email correspondence and meetings were between February 2nd and September 3rd.

P. Simard, B. Victorri, Y. LeCun, and J. S. Denker. Tangent prop-a formalism for specifying selected invariances in an adaptive network. In *Advances in neural information processing systems*, pages 895–903, 1992.

I. Sutskever, J. Martens, G. Dahl, and G. Hinton. On the importance of initialization and momentum in deep learning. In *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, pages 1139–1147, 2013.

R. S. Sutton. Learning to predict by the methods of temporal differences. *Machine learning*, 3 (1):9–44, 1988.

R. S. Sutton and A. G. Barto. *Introduction to reinforcement learning*. MIT Press, 1998.

G. Tesauro. Advances in neural information processing systems 1. chapter Connectionist Learning of Expert Preferences by Comparison Training, pages 99–106. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1989. ISBN 1-558-60015-9. URL `http://dl.acm.org/citation.cfm?id=89851.89863`.

G. Tesauro. Temporal difference learning and td-gammon. *Communications of the ACM*, 38 (3):58–68, 1995.

G. Tesauro. Comparison training of chess evaluation functions. In *Machines that learn to play games*, pages 117–130. Nova Science Publishers, Inc., 2001.

G. Tesauro and T. J. Sejnowski. A parallel network that learns to play backgammon. *Artificial Intelligence*, 39(3):357–390, 1989.

G. Tesauro, E. M. Izhikevich, and I. Stevenson. Scholarpedia article on Gerald Tesauro's work, 2014. URL `http://www.scholarpedia.org/article/User:Gerald_Tesauro/Proposed/Td-gammon`. Retrieved 07 August 2014.

S. Thrun. Learning to play the game of chess. *Advances in neural information processing systems*, 7, 1995.

T. Tieleman and G. Hinton. Lecture notes for the course Introduction to Neural Networks and Machine Learning taught by Tijmen Tieleman and Geoffrey Hinton at De-

partment of Computer Science at University Of Toronto, spring term, 2012. See http://www.cs.toronto.edu/~tijmen/csc321/., 2012.

S. Turaga. Private communication, 2014. Discussions between June 29th and September 3rd.

P. E. Utgoff. Feature construction for game playing. In J. Fürnkranz and M. Kubat, editors, *Machines that learn to play games*, pages 131–152. Nova Science Publishers, 2001.

J. van Rijswijck. Learning from perfection. In *Computers and Games*, pages 115–132. Springer, 2001.

J. Veness. Bodo Chess Engine, 2014a. URL http://jveness.info/software/default.html. Retrieved 9 July 2014. See also http://www6.chessclub.com/activities/finger.php?handle=bodo.

J. Veness. Private communication, 2014b. Email correspondence and meetings were between February 2nd and September 3rd.

J. Veness, D. Silver, W. Uther, and A. Blair. Bootstrapping from game tree search. In *NIPS*, volume 19, pages 1937–1945, 2009.

M. A. Wiering. Td learning of game evaluation functions with hierarchical neural architectures. Master's thesis, University of Amsterdam, 1995. Graduate thesis at Department of Computer Systems supervised by Ben Krose.

M. A. Wiering, J. P. Patist, and H. Mannen. Learning to play board games using temporal difference methods. Technical Report 2005-048, Institute of Information and Computing Sciences, Utrecht University, 2005.

P. Zhou and J. Austin. Learning criteria for training neural network classifiers. *Neural computing & applications*, 7(4):334–342, 1998.