Chapter 1

Introduction

Chess has been paradigmatic of the discrete deterministic tradition in AI. With the rise of the probabilistic and machine learning approaches, the board evaluation process has been modulated with probabilistic models (Baxter et al., 1999b,a), but the underlying deterministic search-driven playing mechanism has not changed in the last seven decades since the birth of AI. The basic paradigm for chess-playing machines has been to essentially generate large search trees by expanding a large set of possible moves upto a certain ply-depth heuristically pruning certain nodes, evaluate the leafs and select the best move by minimax algorithm. With the exponential advances in chip speeds and capacities, this brute force way of computers playing chess eventually led Deep Blue to beat Garry Kasparov who was then the reigning world champion (Campbell et al., 2002).

Meanwhile the study of machine learning has become popular amongst the community focusing on other interesting problems related to three major learning paradigms– supervised learning, unsupervised learning and reinforcement learning. However in case of chess, machine learning has seldom been used except addressing several disjoint issues, for instance learning evaluation weights of various handcrafted features like piece values(Beal and Smith, 1997), piece-square values(Beal and Smith, 1999) and mobility, while others use machine learning techniques to learn opening book moves specifically (Hyatt, 1999). Another class of systems use co-evolution, where the system learns by playing itself (Vázquez-Fernández et al., 2012) and optimizing the parameters (Bošković and Brest, 2011) often using evolutionary methods.

Recently, deep learning has been rapidly expanding into new domains as a field of machine learning. The primary advantage of deep learning systems is the ability to learn hierarchical feature representations that make the models generalize better to unseen data (explained in greater detail in 1.4.7, example in figure 1.1). While deep learning methods inspire most of the state of the art in image and language categorization tasks (Krizhevsky et al., 2012; Tompson et al., 2014; Taigman et al., 2014; Ciresan et al., 2012), there is a lack applications to these discrete search situations. Recently, an important achievement of deep learning architectures has been in achieving human-like control for the task of playing video games through a deep reinforcement learning architecture (Mnih et al., 2013). The prime contribution of the work was the ability to learn human-like control directly from high-dimensional sensory input of the video games. Figure 1.1 shows the last fully connected layer of a CNN trained to play Breakout trained using a reinforcement learning architecture. The separation between the state embeddings which are mapped to different actions is visible.

However, the deep reinforcement learning architecture in the work of Mnih et al. (2013) performs poorly in deciding the optimal moves for games like Pacman etc. where the actions are best decided using a search or planning based method. Recently, an extension of this work by Guo et al. (2014) improved the results by a bit on some games that required extensive planning. Using a Monte-Carlo tree search based player to simulate trajectories, they trained a CNN based regression model on the generated state-action-value instances.

This integration of the CNN based RL agent and a planning agent having expert

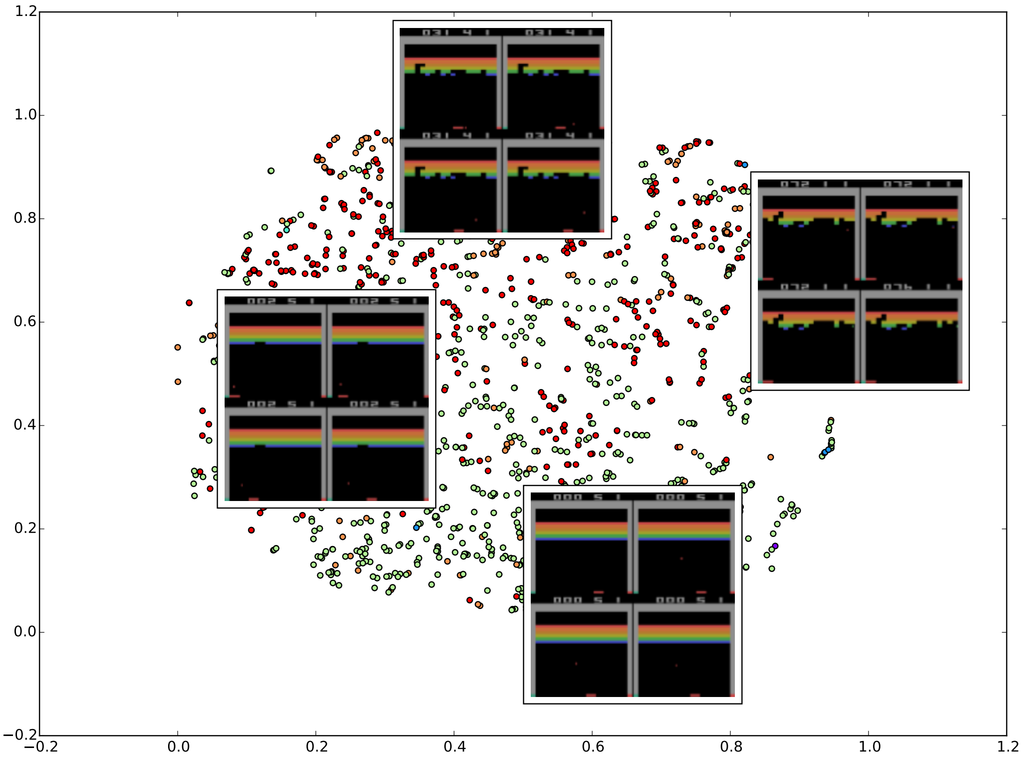


Figure 1.1 The plot shows the TSNE embedding of the last fully connected layer in a network trained for the game of Breakout. The two primary colors–green and brown, respectively, show the two primary actions learned while playing the game of Breakout i.e. left and right respectively and the separation is almost apparent.

control motivates our case of modeling the complex dynamics of chess.

1.1 Learning chess from scratch

In this work, we undertake the task of learning to play the game of chess with minimal prior knowledge. The capability to learn chess, for our task, can be accomplished if a machine can:

- Learn to play legal moves
- Rank the possible moves without any explicit guidance on relative importance of material or position.
- Evaluate board positions

To accomplish this task using minimal knowledge prior, we use Convolutional neural networks to learn the game directly from board positions, moves and outcomes. We will finally evaluate our system on its ability to avoid illegal moves, predict good moves at certain board positions and play against a traditional search based chess player.

We believe that although such a system may not be able to play a championship level game, but it can prove to be an example of a chess player that has learned from scratch only observing chess games and having no clue of the rules.

1.2 Aspects of expert human chess playing

In this section, we look at what motivates us to solve the problem of chess as a pattern recognition task as opposed to the conventional approaches of accomplishing it as a predominantly search problem. We discuss various studies which suggest that most of the chess computer AIs that compete with the best human chess players don't probably play it the actual way humans do. Specifically, we motivate the task of playing chess in a more human way using a pattern recognition system in section 1.2.3.

Adriaan de Groot, a Dutch chess master and a psychologist, after a deep statistical and interpretative analysis of chess players' transcripts of verbal utterances, their eye gaze movement and interviewing a number of beginner and master level chess players concluded that all players usually examine 40-50 positions before playing a move. The difference however is that the master level players develop pattern recognition skills from experience which helps them examine only a few lines of play in much greater depth while ignoring the poor moves, where the beginners spend a lot of their time (De Groot et al., 1996). Another evidence of why this is how humans play chess is that humans, especially at the master level, are capable of recognizing familiar arrangements of board or specific sets of pieces from experiences, rather than random arrangements of the same pieces (Chase and Simon, 1973). Also unique to humans is the capability to learn from experience. This learning can both be ability to recognize patterns from the past games or it could be the ability to learn the weaknesses of the opponent or learn from his/her own strategic mistakes made.

1.2.1 How to become an expert?

Ericsson et al. (2007), based on scientific research that looked at extraordinary and exceptional performance in a number of fields, observes that “experts are always made, never born”. The author studied expertise in wide variety of domains ranging from chess to cooking, before concluding that the amount and quality of practice were the key factors in the level of performance achieved. The author also claims that a minimum of ten years of intense training is required before even the most gifted talents can win international competitions (Ericsson et al., 2006). An aspect of this intense training and practice is that it is deliberate, meaning it consists of considerable and sustained efforts to do something which you can’t already do well. The same goes for chess. Grandmasters, while they are very young, have played a huge number of games and also analyze the games they lost to eliminate all weaknesses in their gameplay.

Before Garry Kasparov, the then reigning world champion, was defeated by Deep Blue (Campbell et al., 2002) in May of 1997 in a chess match under tournament conditions, the two had a match in February 1996, when Kasparov came from 0-1 down to win the match 4-2. The reason Kasparov could make a comeback was that he learned from Deep Blue’s weaknesses and took advantage of them, while the deep blue computer responded similarly the next time too. This goes on to tell us the difference between a chess computer with no learning capability and a human grandmaster who learns about the weaknesses of the opponents. However, eventually Deep Blue was destined to beat Kasparov next year, but that was not an achieved because of a learning component, but more efficient and deeper search along with human intervention for tweaking the computer (CNN, 1997).

1.2.2 How are Grandmasters different?

We associate the ability to play a good game of chess with someone's mental ability– such is the intellectual aura about chess. But Bilalić et al. (2007) suggests that better chess players do not necessarily perform exceptionally well in IQ tests. Further, neither are the chess grandmasters known to evaluate moves more rapidly then others (De Groot et al., 1996). The aspects that mark chess grandmasters from others, appear to be similar to markers of expertise across a wide range of domains:

- The mental representation is in terms of larger chunks, so that positions, and possible actions, are encoded more efficiently (Chase and Simon, 1973; Gobet and Clarkson, 2004). In fact, the work by Mnih et al. (2013) shows that the representation learned by deep reinforcement learning architectures (shown in 1.1) is arranged similarly i.e. states with similar actions are arranged in what may be called chunks.

- Although a grandmaster may evaluate approximately the same number of moves as other players, the moves explored by the grandmaster are the stronger moves. Thus, presented with a board that may arise in a real chess game, the chess expert "sees" only a few good moves. This is also a type of pattern matching, but the complexity is mind-boggling. Perhaps this is possible only because the input space is no longer the set of distinct boards, but the space of all configurations of chunks, which is much smaller.

1.2.3 Chess Reasoning as Pattern Recognition

As we discussed above, the master level human chess players are known to recognize specific arrangements of the board or a specific set of pieces on the board and hence ignore certain poor positions to utilize their time to explore more important lines of play in a greater depth. We believe that an expert pattern recognition system, such as a human, would be able to recognize these patterns learned through experience and hence make a more informed decision to play a certain move. In other words, we think that a pattern recognition system that has seen enough examples of board positions and the corresponding expert moves should be able to predict favorable moves. But at the same time, we expect it to remain a necessity to explore those moves upto a greater depth.

1.3 Related work

In the upcoming text, we will look at some of the related works in the fields of learning games using machine learning rather than logic and search. Some of the very recent works in combining deep learning with reinforcement learning has resulted in human level control of arcade games (discussed above). Many of these works have inspired us to take up Chess playing as machine learning and pattern recognition task.

1.3.1 Convolutional Neural Networks for playing Go

Following the work of Sutskever and Nair (2008) which achieved modest success due to relatively small sized architecture, Maddison et al. (2014) used deep convolutional neural networks to play Go. The network could predict the correct move 55% of the time and beat the traditional search program GnuGo in 97% of the games without using any search and match the performance of a state-of-the-art Monte-Carlo tree search that simulates a million positions per move. They represent the current board position as a 3 channel image, along with adding channels for number of liberties before and after move, legality and the rank of the player. Much of the formulation of our model is motivated by this representation.

1.3.2 Go versus Chess

Using Convolutional networks to predict moves in Go proves to be a great step in the direction of building state of the art Go systems. This motivates us to use the same technique in predicting moves in the game of Chess. However, we must realize that the two games are fundamentally different in many aspects that make Go an easier game to play using a Convolutional Network. The game of Go has smoother arrangements of positions that are almost continuous within and between games.

Every move in Go adds a single piece to the board. Numerically speaking, making a move in Go using a random draw has a chance of $\frac{1}{361}$ on a 19×19 board, while making a move drawing randomly the from and to positions in chess is correct with a chance of a mere $\frac{1}{4096}$. Also, a single move on a chess board makes a change of at least 2 pixels (more than 2 for a piece capture) which is significant for an 8×8 board, while in Go only 1 pixel is added to the board every move making the transition much smoother than chess. Since, much of the knowledge of chess is characterized by strong domain knowledge in a logical form, such as “if bishop on the central diagonal”, “if the king has liberty more than 1” etc., makes it less intuitive if as compared to the case of Go, Convolutional Neural networks can actually model the rules, leave aside the optimality of the moves.

1.3.3 Deep learning for Chess

This work also inspired a modest attempt of using convolutional neural networks to play chess, which achieved minimal success because of a much small dataset and weak representation (Oshri and Khandwala, 2015). We also acknowledge this work to have inspired us to make use of Convolutional neural networks to play chess. Other works that use deep learning to play chess do not learn the piece arrangements and their relative importances to score the table or evaluate it, rather they utilize a set of handcrafted features like king’s safety, king’s liberty, number of rooks on seventh rank etc. (Mannen, 2003; Thrun, 1995)

1.3.4 Learning from Mistakes

An important part of playing chess is to remember the each position in which the program made a mistake, so that the mistake is not repeated next time the position is encountered. The game playing system by Epstein (2001) named HOYLE looks at the last position where the losing player could have made an alternate move by exhaustively searching the game tree. If the search succeeds, the alternate move is recorded and the state is marked as “significant”. If the search fails to find an alternate path to winning position, the state is marked as “dangerous”. In 1.4.8.1, we will look at how machine learning architectures like convolutional neural networks can provide more generalized representations, so that such “significant” and “dangerous” states as well as other similar states are represented and searched for efficiently.

1.4 Background

In this section, we will first look at what it means to play the game of chess perfectly, before moving on to studying how computers are programmed to play chess efficiently and exceptionally well. Further we look at some background about reinforcement learning, deep learning and learning representations and how convolutional neural networks, a specific case of deep learning architectures, has shaped the field of artificial intelligence especially in the area of pattern recognition.and

1.4.1 Ideal Leaf-evaluation function

Solving chess refers to finding an optimal strategy for playing chess. Chess has a finite number of states, estimated to be around 10^{43} by Shannon (1950). Zermelo (1913) proved that a hypothetically determinable optimal strategy does exist for chess.

In a weaker sense, each position can be assigned as a win for white, a win for Black or a forced draw if both the players are using the optimal strategy. We can formulate this as a function f for each board position using the procedure below.

1. Assign all the positions with no further play possible as:

$$f(position) = \begin{cases} 1 \text{ , if White has won} \\ 0 \text{ , if it is a draw} \\ -1 \text{ , if Black has won} \end{cases}$$

2. Use the recursive rule up the game tree:

$$f(p) = \max_{p \to p'}(-f(p'))$$

   where p' is a position reachable in one move from position p.

   Here the negative sign means that a win for the opponent is a loss for the player in consideration and vice-versa.

The above recursive rule is the same as the minimax algorithm for the case when the game tree for chess is fully grown. The function f gives us the "perfect" evaluation function to play a chess game. While playing the game, we just need to choose the move which takes us to the board position p for which $f(p) = 1$.

But this evaluation function, f, cannot be computed with the computing resources available as of now. Hence, we need to resort to approximations of $f(p)$.

1.4.2 Playing Chess using Computers

The first study on computers playing the game of chess was done by Shannon (1950). He predicted two main search strategies that would evolve with computers being programmed to play chess–

1. "Type A" programs would use a "brute force" examination of all the board positions possible through valid moves upto a certain depth of play.

2. "Type B" programs would employ a "quiescence search" looking at only a few good moves for each position.

He also predicted the "Type A" programs to be impractical because of the massive search space. For even a computer evaluating 10^6 moves per second, a lookahead of 3 moves for both sides, when an average of 30 moves are available for each board position, the program would take around 13 minutes.

Depth (ply)	Number of Positions	Time to Search
2	900	0.0009 s
3	2.7×10^3	0.027 s
4	8.1×10^4	0.81 s
5	2.43×10^7	24.3 s
6	7.29×10^9	12.15 mins
7	2.187×10^{10}	6.075 hours

Table 1.1 Time taken to explore 30 moves per position at 10^6 moves per second.

However, with the exponential increase in the computation power since then, the most successful methods for playing chess are programmed to use "Type A" programs. One reason for ignoring "Type B" is that relying on the board configuration to decide on the moves to explore is a much harder problem than using a faster, yet weaker, evaluation metric for a larger depth. It is widely believed that a faster evaluation function with an efficient search along with heuristic pruning would build a better chess playing computer than the one that uses a better but slower evaluation function that involves pattern recognition techniques similar to our brain.

1.4.3 Conventional Chess Playing Computers

The fundamental implementation details of chess-playing computer system include:

- **Board Representation** – how a board position is represented in a data structure. The performance of move generation and piece evaluation depends on the data structure used to represent the board position. The most common representation uses a list of piece positions for each position. Some of the other methods are– mailbox, 0x88, bitboards and huffman codes. We will discuss the representation used for our training and playing tasks in Chapter **??**.

- **Search Techniques** – how to identify and select the moves for further evaluation. Some of the most widely used search techniques are– Minmax, Negamax, Negascout, Iterative deepening depth-first search etc. Much of the focus on state of the art chess playing systems resides on making search more efficient to evaluate more board positions per unit time. One of these algorithms which we make use of is Negamax algorithm and has been explained in section 2.6.3.

- **Leaf Evaluation** – how to evaluate the position of the board if no further evaluation needs to be done. The mapping from a board to an integer value is called the evaluation function. An evaluation function typically considers material value along with other factors affecting the strength of play for each player. The most common values for materials is 1 point for pawn, 3 for bishop, 3 for knight, 5 for rook, 9 for queen and 200 for king. The high value for king ensures that checkmate outweighs everything. The sum of the values for all material on the board with negative weights to the opponent is the evaluation of the table. In addition to pieces, most evaluation functions take into consideration more factors like pawn structure, pair of bishops, protection of the king etc.

1.4.4 Reinforcement Learning

The fundamental principle behind all Reinforcement learning methods is that we use the current policy, run it on the environment, observe the feedback and make the good outcomes more likely, the bad outcomes less likely. Reinforcement learning differs from standard supervised learning in the way that an RL algorithm is not presented with optimal actions to input states (Sutton and Barto, 1998). The basic reinforcement learning model contains the following components:

1. a set of environment states $\mathcal{S}$

2. a set of actions $\mathcal{A}$
3. transition rules between states
4. rules that determine the *scalar intermediate reward* of a transition
5. rules that describe what the agent observes

However it is a very hard problem to do a policy search using a reinforcement learning architecture when we have a combination of complex dynamics and complex policy. The high dimensionality of such policies doesn't allow efficient policy search. However, there have been successful attempts to convert these complex dynamics and complex policy domain problems to only a complex policy problem by decomposing the policy search into two phases– optimal control and supervised learning (Levine et al., 2015). The end to end training for the supervised learning part is done using a function approximator, convolutional neural networks in our case, which we will discuss in section 1.4.8. Other attempts to combine conventional reinforcement learning techniques with deep learning to learn optimal control have yielded near human performance on tasks like playing Atari video games (Mnih et al., 2013). The task that representation learning architectures like CNNs solve is the problem of perception i.e. the feature representation of the states need not be composed of hand-crafted features, but can be learned while training.

1.4.5 Deep Learning

The area of Neural Networks has been inspired by the aim of modeling biological neural systems. Although it has diverged from this aim since then, the basic computational unit in an artificial neural network still remains a neuron, which takes the signal from the axons from other neurons as inputs, with dendrites carrying the signal to the nucleus where it gets summed up and the neuron is activated if the sum is above some threshold. Such a neuron can be represented mathematically as: $f(\sum\limits_i w_i x_i + b)$, where f is the activation function, w_i is the weight given to the input from one of its dendrites. In other words, a neuron computes the dot product of its weights and the inputs, adds the bias and applies an activation function. A mathematical model of a neuron is shown in the figure below.

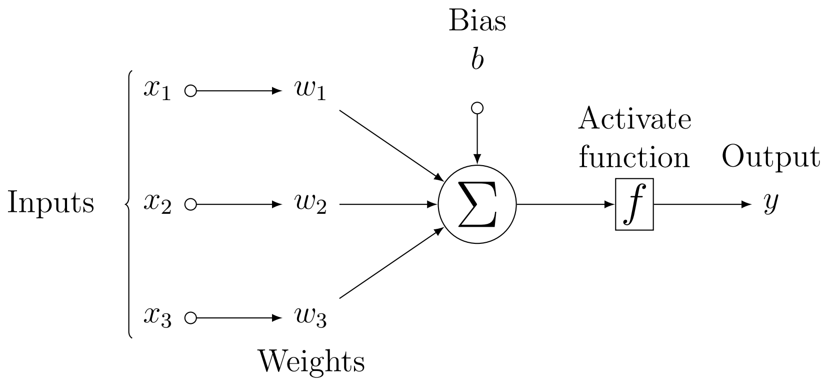


Figure 1.2 An artificial neuron as a mathematical model

Neural networks are the collections of neurons that are connected in an acyclic graph. This means that outputs of some set of neurons becomes the input of another set of neurons. The most common arrangement is a layered neural network with an input and an output layer, along with none or more hidden layers. The input layer has number of neurons equal to the input dimension and the number of output layer neurons is the dimension of the output. The hidden layers can contain different numbers of neurons. A two-layer neural network is shown in the figure below. The two-layer here refers to the number of layers besides the input layer.

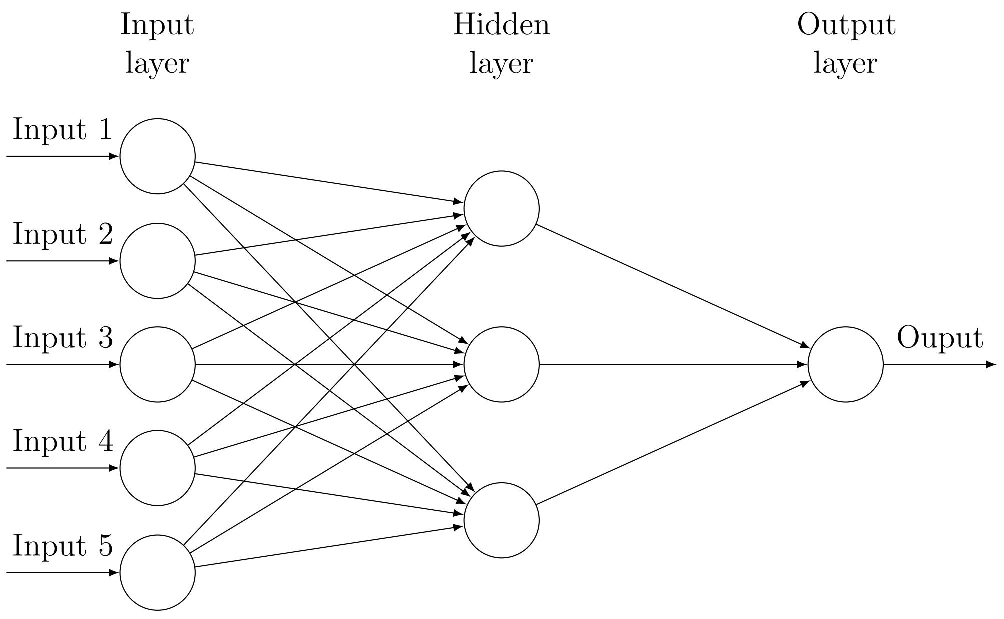


Figure 1.3 A two layer Artificial Neural network

In practice, we model a real valued function using a multi-layer neural network. This real valued function, for example, is the one which outputs the class value in case of a classification task or a continuous function in case of a regression task. In other words, a multi-layer artificial neural network defines a family of functions parameterized by the weights of the network. By learning the weights of such a network we usually means learning a function belonging to this family of functions that best represents the training data.

1.4.6 Universal Approximation Properties of Multilayer Perceptrons

We discussed above that a family of functions is represented by a single artificial neural network with fixed architecture. It has been proved that Neural networks with at least one hidden layer are universal approximators (Hornik et al., 1989). This means that given any continuous function $f(x)$ and some $\epsilon > 0$, a Neural network with one hidden layer containing a sufficient number of hidden layer neurons and a suitable choice of non-linearity, say represented by $g(x)$, exists such that $\forall x, |f(x) - g(x)| < \epsilon$. In other words, we can approximate any given real valued continuous function with a two layer Neural network upto a certain accuracy.

The fact that two layer Neural networks are universal approximators is a pretty useless property in the case of machine learning. Neither does it tell the number of hidden units required to represent a given function upto the desired precision, nor does it promise that it represents a generalized function that fits the unseen data. The generalized function is expected to be smooth, while the overly precise two-layer network may overfit for the input data and not learn a promising representation.

1.4.7 Representation Learning

We learnt that despite being universal approximators, it may not reasonable to approximate a function for a task at hand using two-layer Neural networks because of insufficient generalization ability. Meanwhile, a lot of experimental evidence from the recent past shows that these functions can be learnt to a greater generalization using Neural networks of larger depths (Bengio et al., 2015). Most of these works involve using a specific class of Neural networks architectures, namely Convolutional Neural Networks, which we will discuss in section 1.4.8. This leads us to believe that depth indeed is a useful consideration to make while choosing a Neural network architecture to fit data, and it provides the learner's family of functions multiple levels of representation and hence making the function smoother yielding better generalization.

1.4.8 Convolutional Neural Networks

Convolutional neural networks, sometimes referred to as Convolutional Networks or ConvNets, is a particular architecture of deep neural networks inspired by the seminal work by Hubel and Wiesel (1963) on feedforward processing in early visual cortex. The architecture uses hierarchical layers of tiled convolutional filters to mimic the effects of receptive fields. These filters exploit the local spatial correlations present in the images. In practice, these hierarchical layers are alternated with subsampling layers like max-pool and a non-linearity map, and further connected to fully connected layers just like in other deep learning architectures and the full network is trained using back propagation. In short, Convolutional neural networks are nothing but neural networks which use convolution instead of full matrix multiplications in atleast one of the layers (Bengio et al., 2015).

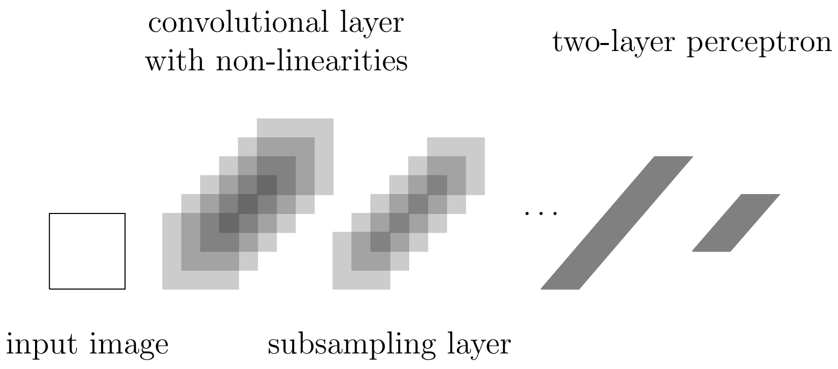


Figure 1.4 A typical Convolutional Neural Network architecture

Convolutional Neural Networks have brought about a revolution in computer vision and is now the most successful approach for almost all recognition and detection tasks in computer vision (Krizhevsky et al., 2012; Tompson et al., 2014; Taigman et al., 2014) and some even approach human performance on some tasks (Ciresan et al., 2012).

1.4.8.1 Utilizing CNN's representation power

In 1.3.4, we looked at a heuristic based game playing system called HOYLE (Epstein, 2001) that recorded a set of "significant" and "dangerous" states by exhaustively searching the tree, starting from states to look for alternate paths to win, whenever a game is lost. It is easy to understand that this system is not scalable to learning from a large set of games and neither can the applicability be explained with a small number of recorded states. However, the representation power of the convolutional neural networks can help in the representation of these "dangerous" and "significant" states. For instance, the first fully connected layer represents a n-dimensional space in which the similar states are embedded closer. This can significantly contribute to the HOYLE's strategy by providing a better generalization and efficient representation to the recorded states.

Organization of the thesis

The thesis is organized in the classic style of first introducing the problem and discussing related work (chapter 1), then chapter 2 where we describe our dataset representation and implementation details while systematically moving towards our aim. Finally we present the results and analysis (chapter 3). In the chapter 2, we specifically mention the convolutional neural network architecture we use, the loss functions and the how we employ them to choose a move while playing a game of chess. In the results chapter, we describe the strengths and weaknesses of our system by considering specific case studies. Finally, we conclude with a discussion about the feasibility of such systems and how they can be a part of the future of computer chess.

Chapter 2

Implementation

The major contribution of our work is to provide a method in which convolutional neural networks can be trained to learn and play the game of chess. In this chapter, we will describe our dataset– acquisition and its representation in sections 2.1 and 2.2 respectively. Then we discuss the details of the architectures we use (section 2.3), before moving on the algorithms to decide which move to play at the current board position (section 2.5).

2.1 Training and Testing Dataset

We acquired two datasets from FICS Games Database[1] website. The first one is a set of games played between chess players with average ELO rating greater than 2000 in the year 2014. The second one consists of the set of games played between computer chess engines played between 2009 and 2014.

The set of games in each of the two datasets was fragmented into moves, where each move was represented by a board representation (described in the next section) and a label (the position of the piece moved and the position to which the piece was moved). We flip the board across the horizontal after flipping the color of each piece to convert the move made by the players with black pieces to the corresponding move by a player with white pieces. We discard the moves played by players with

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

ELO rating less than 2000 since we had enough data to not consider them as an "expert".

We shuffle the data because of two reasons:

- Our models and training procedures do not utilize the order in which the game was played. Each move is learnt with respect to the position of the pieces on the board.

- We use batched stochastic gradient descent to learn the parameters of our models. Learning from consecutive samples can introduce high variance in the updates. This happens due to the strong correlations between consecutive samples and resultantly makes the learning procedure inefficient (Mnih et al., 2013)

  The data-set was split as follows.

Dataset Name	Number of Games	Total number of boards
CvC 2009-14	$370,480$	$74,162,875$
FICS 2014 (avg ELO>2000)	$522,356$	$32,598,059$
FICS 2014 (all ELO, subset)	$\sim 1M$	$\sim 125M$

Table 2.1 Dataset sizes

2.2 Data Representation

2.2.1 Training data for Piece and Move predictors

Since we divide our task of evaluating a move into first evaluating the position of the piece to be played and then evaluating the position to which the piece will be played, we represent the data according to the tasks. Effectively, we have 7 datasets to train on– First where the labels are the positions of the piece to be played for the input board, and rest of the 6 (for each piece type) where the inputs are again the current board positions, but the labels are the final positions if any one of that

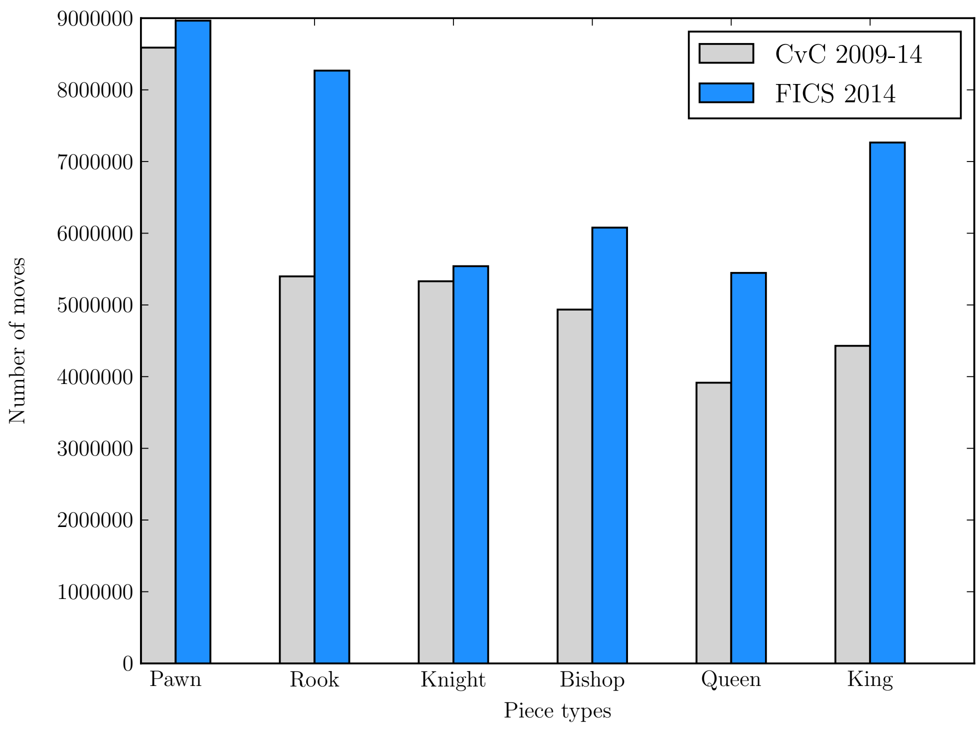


Figure 2.1 Dataset statistics

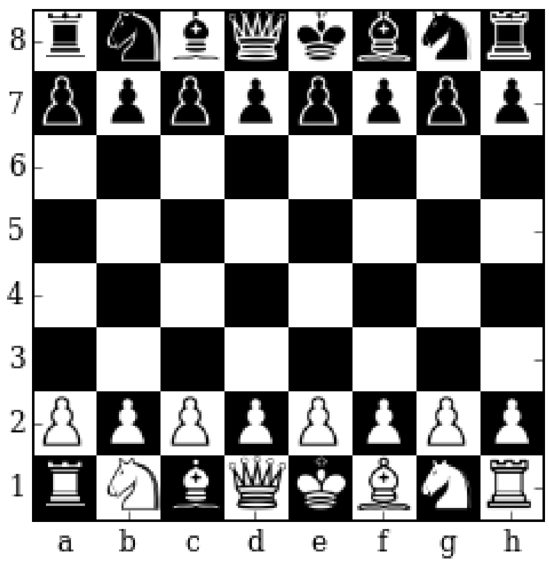
particular type of piece is chosen. We name the datasets as– piece dataset, pawn dataset, rook dataset and so on.

2.2.2 Basic representation

Following 2 representations were used to represent a chess board:

1. **6-channel**: Each board position is represented as 6 channels of 8×8 images with values from the set $\{-1, 0, 1\}$. Each channel represents the piece types, namely pawn, rook, knight, bishop, queen, king respectively. The friendly player's pieces are represented by a 1 in the respective position in the respective channel, while the opponent's pieces are represented by a -1 in the respective position in the respective channel. Rest of the elements in the 8×8 grid are 0's.

   Board representation dimensions: $8 \times 8 \times 6$



0	0	0	0	0	0	0	0
-1	-1	-1	-1	-1	-1	-1	-1
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1
0	0	0	0	0	0	0	0

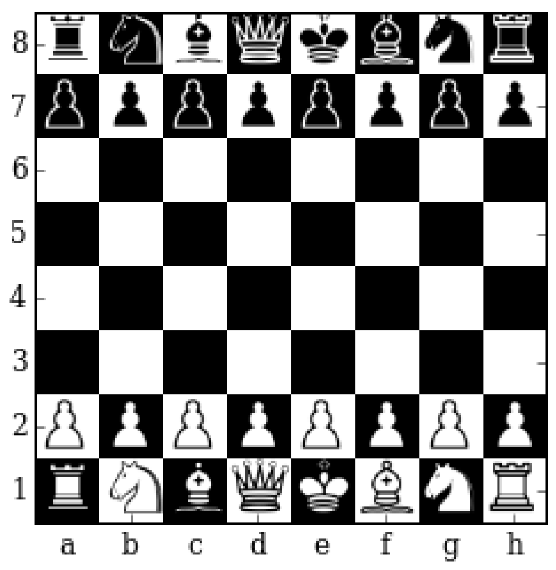
. . . . . .

0	0	0	0	-1	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0

Figure 2.2 6-channel representation of the initial chess board position
Leftmost matrix represents the pawn channel. It is followed by the channels for Rook, Knight, Bishop and Queen and the rightmost matrix represents the king channel.

2. **12-channel**: Each board position is represented as 12 channels of 8×8 images with values from the set $\{0, 1\}$. Every alternate channel belongs to a single player and similar to the 6-channel representation each of these channels represents the piece types, namely pawn, rook, knight, bishop, queen, king respectively. The friendly player's pieces are represented by a 1 in the respective position in the respective channel, while the opponent's pieces are represented by a 1 in the respective position in the respective channel. Rest of the elements in the 8×8 grid are 0's.

   Board Representation dimensions: $8 \times 8 \times 12$ (Width, Height, Depth)



$$\boxed{\begin{matrix} 0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&0\\ 1&1&1&1&1&1&1&1\\ 0&0&0&0&0&0&0&0 \end{matrix}} \quad \cdots\cdots \quad \boxed{\begin{matrix} 0&0&0&0&1&0&0&0\\ 0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&0 \end{matrix}}$$

Figure 2.3 12-channel representation of the initial chess board position Leftmost matrix represents the White-pawn channel. It is followed by a channel for Black-pawns, then two channels each for Rook, Knight, Bishop and Queen and the rightmost matrix represents the Black-king channel.

2.2.3 Additional Bias channels

The dataset provides us with additional information about the players as well as the game. We try to utilize that additional information by augmenting additional channels to the board representations.

1. **ELO Layer**: We append an 8×8 channel with the ELO rating (normalized) of the player playing the current move. So, if the ELO rating of the current player is X, we will augment the 8×8 matrix consisting of all values equal to $\frac{X - MINELO}{MAXELO - MINELO}$, where the values of MINELO and MAXELO are the minimum and maximum ELO ratings of the players in the complete dataset. This adds an additional $8 \times 8 \times 1$ image to the input.

2. **Piece Layer**: Since the task for the move predictor is just to predict the final position of a model-specified piece, we provide the move predictor learning procedure(s) the actual piece in consideration for the move. In particular, we augment an additional channel with all 0s except a 1 in the place of the piece being moved.

   This information makes the learning procedure much more efficient since it no longer has to learn, from the data, which piece is moved and is already learnt by the piece selector. Adding this channel gives us a more consistent probability metric to decide the move (described in 2.6.2).

   This adds an additional $8 \times 8 \times 1$ image to the input.

   Although the inclusion of this layer seems intuitive and beneficial, the downside to a piece specific channel is that it makes the move prediction procedure much slower. This happens because the final position probability distribution has to be predicted for all the pieces of the same type separately. If this layer is not included, we can cache the probability distributions by the move prediction network for each piece types the first time they are queried. Besides this disadvantage, the piece layer increases the prediction accuracy of the Move networks.

3. **Outcome Layer**: The outcome layer is another dynamic bias layer which is appended to the input image layers which indicates the final outcome of the game corresponding to the player facing the current board position. Specifically, we append an image of all ones for a player who finally won the game by resignation or check-mate, an image of all zeros for a player who finally lost or resigned the game, or an image of all values as 0.5 for both the players if the game was a draw. This adds an additional $8 \times 8 \times 1$ image to the input.

   The significance of this layer is to give an additional information to the model which can help evaluating the quality of the move better.

2.3 Piece and Move Predictor

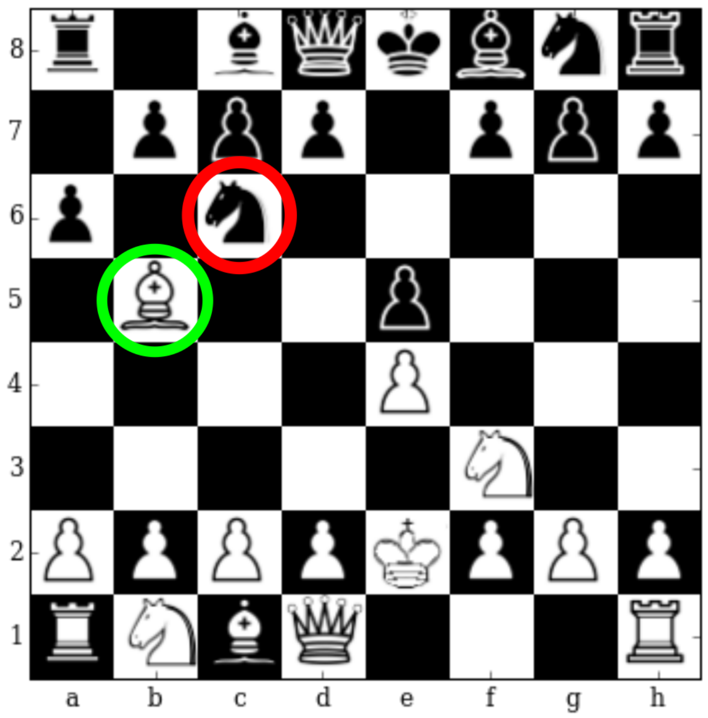


Figure 2.4 Model Overview
The green circle is the position to be predicted by the PIECE selector network, while the red circle is the position to be predicted by the BISHOP move selector network.

A move in chess is characterized by legal movement of a friendly piece. The selected piece is moved onto one of the positions depending on factors like its ability to jump over another piece or not, whether the final position already has a friendly piece, whether the move doesn't cause a check etc. We divide this task into two simple steps–choose a piece, choose its final position. Given a board, we predict the next move by simply making the predictions in this order. While learning we consider the moves from the dataset as gold standard and minimize our loss functions in order to make the network learn certain generalizations, rules and even strategies. In figure 2.4, the two different colored circles represent these two tasks. A formulation of this process would be:

$$P(move|board) = P(from|board) \times P(to|from, board)$$

To learn the probability function, we make use of Convolutional Neural Networks.

Each of the different types of pieces has its own independently trained network, namely– PAWN, ROOK, KNIGHT, BISHOP, QUEEN and KING, while the piece selector network is named PIECE.

The network used for both piece selector and move selectors are shown below:

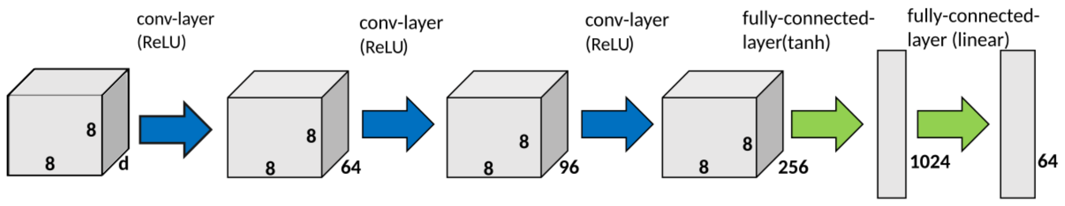


Figure 2.5 Network Architecture: ChessNet-1
This network was used for piece and move predictions.

The convolutional neural network shown in figure 2.5 consists of input layer of size $8 \times 8 \times 6$ (or $8 \times 8 \times 12$ depending on the choice of representation, described in section 1), an output layer of size 64 which predicts the position of the piece to be moved. The other specifications of the network are:

- **Convolutional Layer**–
    1. filter size $=3 \times 3$ each of depth 6
    2. number of filters=64
    3. padding=1
- Non-linearity– **ReLU** $max(0, x)$
- **Convolutional Layer**–
    1. filter size $=3 \times 3$ each of depth 64
    2. number of filters=96
    3. padding=1

- Non-linearity– **ReLU** $max(0, x)$

- **Convolutional Layer**–

  1. filter size $=3 \times 3$ each of depth 96
  2. number of filters=256
  3. padding=1

- Non-linearity– **ReLU** $max(0, x)$

- **Fully connected layer**– dimensions=1024

- Softmax (Loss) Layer – **Softmax**, $p_k = \frac{e^{o_k}}{\sum_i e^{o_i}}$, o_k is the activation at the last fully connected layer in the network.

While training, the softmax layer is replaced by a softmax loss layer, output for the softmax loss layer is defined as follows:

$$L = -\sum_j y_j log p_j$$

where L is the loss, y_j is 0 or 1 depending on the actual label. The derivative of the loss function can be derived as follows:

$$\begin{aligned}
\frac{\partial L}{\partial o_i} &= -\sum_k y_k \frac{\partial log p_k}{\partial o_i} \\
&= -\sum_k y_k \frac{1}{p_k} \frac{\partial p_k}{\partial o_i} \\
&= -y_i(1 - p_i) - \sum_{k \neq i} y_k \frac{1}{p_k}(-p_k p_i) \\
&= p_i \left(\sum_k y_k \right) - y_i \\
&= p_i - y_i
\end{aligned}$$

2.4 Learning the evaluation function

In addition to a piece and move predictor, we also learn an evaluation function. Since we want to be consistent with not injecting any domain knowledge for the game of chess to our models, we formulate the task of learning the evaluation function as regression problem with the evaluations of the boards inspired by TD-Learning (Barto et al., 1989; Sutton, 1988). Particularly, without using any other knowledge of the game, we simply assign discounted reward values to the board as its evaluation. Starting with the final leaf nodes (after which no more gameplay is possible or one of the player has resigned), we assign board values to be 1 for the winning player's board view, -1 for the losing player's board view[2] and 0 for both if it was a draw. This assignment is the same as that described as an optimal evaluation function for chess given the fully expanded game tree described in 1.4.1. But for the preceding board positions, we assign a discounted reward value as the valuation using a discounting factor represented by γ, where $0 \leq \gamma \leq 1$.

$$V(b_{t_{final}}) = \begin{cases} 1 \text{ , if White has won} \\ 0 \text{ , if it is a draw} \\ -1 \text{ , if Black has won} \end{cases}$$

According to the recursive rule the discounted reward for a board at time t into the game is:

$$V(t) = \gamma V(t+1), \forall t < t_{final}$$

[2]Since we are learning from white's perspective, the player's board view refers to how the board would have looked to the player if he was playing as a white i.e. the board flipped across the horizontal as well flipped by color.

Use the following rule moving up into the game:

$$V(b_{t_{final}-i}) = \begin{cases} \gamma^i \text{ , if white won eventually} \\ -\gamma^i \text{ , if black won eventually} \\ 0 \text{ , if the game was a draw} \end{cases}$$

$$V(b'_{t_{final}-i}) = \begin{cases} -\gamma^i \text{ , if white won eventually} \\ \gamma^i \text{ , if black won eventually} \\ 0 \text{ , if the game was a draw} \end{cases}$$

where $b_{t_{final}-i}$ is the board (as it appears to the white player) i steps away from the finish, while $b'_{t_{final}-i}$ is the rotated board with flipped colors i steps away from the finish.

Since the learning is done offline (in our case using game databases), we simulate the whole game before assigning values to each board appearing in the board. Now, each board position seen in the data has some evaluation that is based on the final outcome of the game. It is intuitive to say that a player who is seeing a higher valued board is more likely to win. And higher the value of the current board, the closer I am to winning the game and vice versa.

We model the task of learning such an evaluation function as regression problem on the network shown in figure 2.6.

The input layer size is $6\times8\times8$, while the output layer size is 1 i.e. the evaluation function's value for the input board. The loss function optimized to train the network is the *mean squared error* which is the basic linear regression loss function.

$$L = \frac{1}{2}(f(x) - y)^2$$

where x is the value at the last fully connected layer in the network and f is the

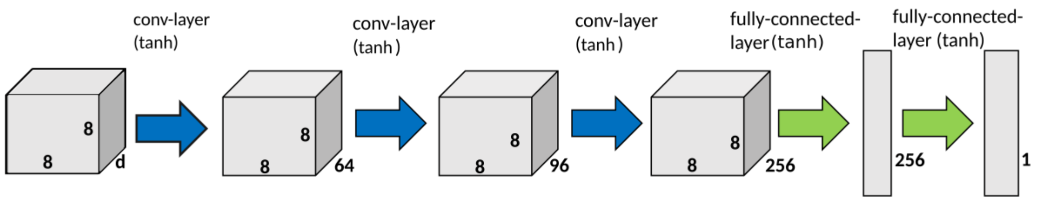


Figure 2.6 Network Architecture: ChessNet-2
This network was used for regression to learn the evaluation function. The difference from the network in 2.5, other than the dimensions of output, is the type of non-linearity used. In this network we use *tanh*, rather than ReLU since we need values to range from -1 to 1. We also tried other non-linear activation functions like leaky ReLU or a sigmoid function with no significant difference in results.

activation function (tanh in our case). Simply the gradient looks like:

$$\frac{\partial L}{\partial x} = (f(x) - y)\frac{\partial f(x)}{\partial x}$$

The backward pass for rest of the layers works in the same way as the previous network.

2.5 Playing

In this section we will look at the algorithms we use during the gameplay to decide which move to play depending on the outputs of our models. The first two algorithms do not make use of search and just make predictions based on the current table, while the rest mix the predictions or evaluations made by our model with a search based mechanism to predict the next move.

2.6 Choosing a move

The output by each of the networks for a given input board (with or without the augmented channels) is a probability distribution over the whole board. The probability distribution output by the trained PIECE network, call it P_{piece}, is a 64-dimensional vector where each value denotes the probability of moving the piece at the corresponding position on the board. Similarly, the probability distribution output by the i^{th} MOVE network, call it $P_{move,i}$, is a 64-dimensional vector where each value denotes the probability of moving a piece of type i to that particular position. We use two or more of these probabilities to choose the complete move (initial and the final position) using one of the methods described below.

2.6.1 Top-move method

Simply choose the piece at position with the maximum probability in the PIECE network's output. Determine its piece type and input the same board to the type's MOVE network. Choose the final position with the maximum probability.

Algorithm 1 Top-move method

```
1: initial_pos = argmax($P_{piece}(board)$)
2: piece_type = getType(initial_pos)
3: final_pos = $argmax(P_{move,piece\_type}(board))$
4: return chess_coords(initial_pos)+chess_coords( final_pos)
```

2.6.2 Top-prob method

Rather than just determining the distribution of the moves originating from the most probable piece, we can also generate the full cumulative probability distribution for every possible from-to pair. This method is based on the principle that:

$$P(move|board) = P(piece|board) \times P(final\_position|piece, board)$$

Algorithm 2 Top-prob method

```
 1: piece_dist = $P_{piece}(board)$
 2: cumulative_dist = $zeros$(64,64)
 3: for $0 \leq i < 64$ do
 4:     if $board[i/8, i\%8] \neq 0$ then
 5:         piece_type=$getType$(i)
 6:         move_distr = $P_{move,piece\_type}(board)$ * piece_distr[i]
 7:         cumulative_distr[i] = move_distr
 8:     end if
 9: end for
10: initial_pos, final_pos = $argmax$(cumulative_distr)
11: return $chess\_coords$(initial_pos)+$chess\_coords$ ( final_pos)
```

2.6.3 TopProb-Negamax Interleaved

In this method we interleave our method of evaluating moves with negamax, which is a kind of minmax search algorithm. We will describe both Negamax and our pruning method interleaved with Negamax. The basic idea is that we will reduce the search space for the Negamax algorithm and also provide it with an evaluation metric for the moves or the boards.

The Negamax algorithm is derived from Minimax algorithm. However it differs in the way that it utilizes the same subroutine for the Min player and the Max player at each step, passing on the negated score following the rule:

$$max(a, b) = -min(-a, -b)$$

The functions *makeMove* and *unmakeMove*, as the names suggest, make the move on the board before calling Negamax again for the child nodes, and then un-make the move to search the subtrees of the siblings.

However, the function calls of our interest is the *generateMoves* and *evaluate*. *generateMoves* function uses chess logic to generate legal moves, which are then explored using recursion, while *evaluate* uses the player's evaluation function to return the value for a leaf node.

Algorithm 3 The basic Negamax algorithm for Chess

```
 1: procedure NEGAMAX(()depth)
 2:     if depth==0 then
 3:         return evaluate()
 4:     end if
 5:     max = $-\infty$
 6:     generateMoves(...)
 7:     while m = getNextMove() do
 8:         makeMove(m)
 9:         score = -Negamax(depth -1)
10:         unmakeMove(m)
11:         if score > max then
12:             max = score
13:         end if
14:     end while
15:     return max
16: end procedure
```

For our case, we can modify the *generateMoves* procedure to actually just generate a reduced number of moves available at every board position, hence pruning the search tree. The reduced number of moves are generated in a way similar to algorithm 2, where we generate a cumulative probability distribution of size 64×64. The difference being that rather than making a move, we just explore the subtree of the generated move. In the same way as before, we can make use of transposition tables to store the values of boards already evaluation while expansion of the tree.

Our idea of making a narrower and deeper search of the game tree instead of a full width and shallow search is motivated by De Groot et al., 1996 as discussed in chapter 1.4.

For *evaluate* function call, we simply ask our network to evaluate one or a batch of boards. The function is better described earlier in the section 2.4.

2.7 Training: Implementation details and Hyper parameters

We performed all our training and testing for the piece and move prediction models on a deep learning library named Caffe (Jia et al., 2014) using its Python API. However, we couldn't use Caffe for the regression training and related experiments because of convergence issues. The implementation for the regression training to learn the evaluation function was done using a Theano-based deep learning library named Keras (Chollet, 2015). The complete source code of our implementation is available on `https://github.com/ashudeep/convchess`.

All the experiments were run on a machine with NVIDIA GeForce GTX 760 GPU with Cuda v6.5.

Move prediction models: We experimented with different hyperparameters for the learning procedure and obtained our best models at: base learning rate of $\alpha = 0.1$ and a batch size of 1000. We used AdaGrad to compute the updates. The learning rate was kept constant for the first $50,000$ iterations before reducing it to 0.01. The move prediction networks in Caffe were trained uptil $300,000$ iterations which took about 2-3 days.

Evaluation Function learning: We used a learning rate of 0.001 and RMSprop with a learning rate decay of 0.9 to minimize the mean squared error. We use a batch size of 1024. We varied the values of γ (described in section 2.4) between 0.7 and 0.99 to train separate models. The results and analysis for each of them are described in this chapter.

Chapter 3

Results and Analysis

The two primary components of a chess playing system are: **evaluation function** and **search**. In chapter 1.4 we already looked at these two aspects of the conventional chess playing systems and how the exponential increase in computational power has led to better search-based methods, due to which we have computer chess systems that can beat the best humans. Our aim here is not to build such a system, but to provide a proof of concept that Convolutional Neural Networks, one of the most powerful pattern recognition architecture, can observe patterns from chess games and eventually learn to play chess. As described in chapter 2, we use CNN to learn an evaluation function and a move predictor directly from chess games with minimum injection of chess knowledge to the predicted probabilities. In this chapter, we empirically evaluate these CNN based models on their ability to play chess.

3.1 Training

Figure 3.1 and 3.2 show the training loss and test loss during training of the PIECE prediction models. The plots for the move prediction models of different pieces are similar and hence omitted for brevity. The loss function plotted below is defined by:

$$L = -\sum_{j} y_j log p_j$$

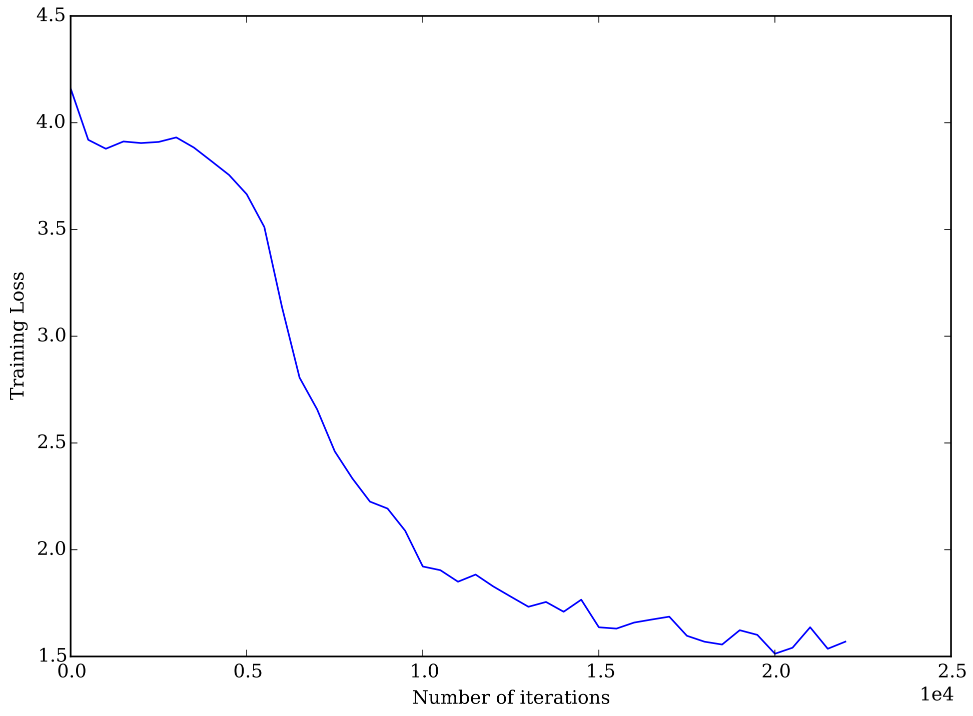


Figure 3.1 Variation of the softmax-loss while training of PIECE model

where L is the loss, y_j is 0 or 1 depending on the actual label. This is the function we minimize while training and has been discussed in section 2.3

3.2 Performance of Piece and Move networks

As discussed in Chapter **??**, we divided the dataset into training and testing sets. We train the network, using the optimum hyper-parameters described in section 2.7. The training and testing accuracies and softmax losses have been reported in the table below. We also report the top-k prediction accuracies for $k = 3, 5, 10$ also.

Model Name	Accuracy at k			
	k=1	k=3	k=5	k=10
Piece	56.0	87.9	95.8	99.5
Pawn	94.8	100.0	100.0	100.0
Rook	58.0	85.2	93.5	99.6
Knight	77.3	96.6	99.3	100.0
Queen	54.2	81.3	90.4	98.2
King	71.7	95.6	99.6	100.0

Table 3.1 Test set performance without masking

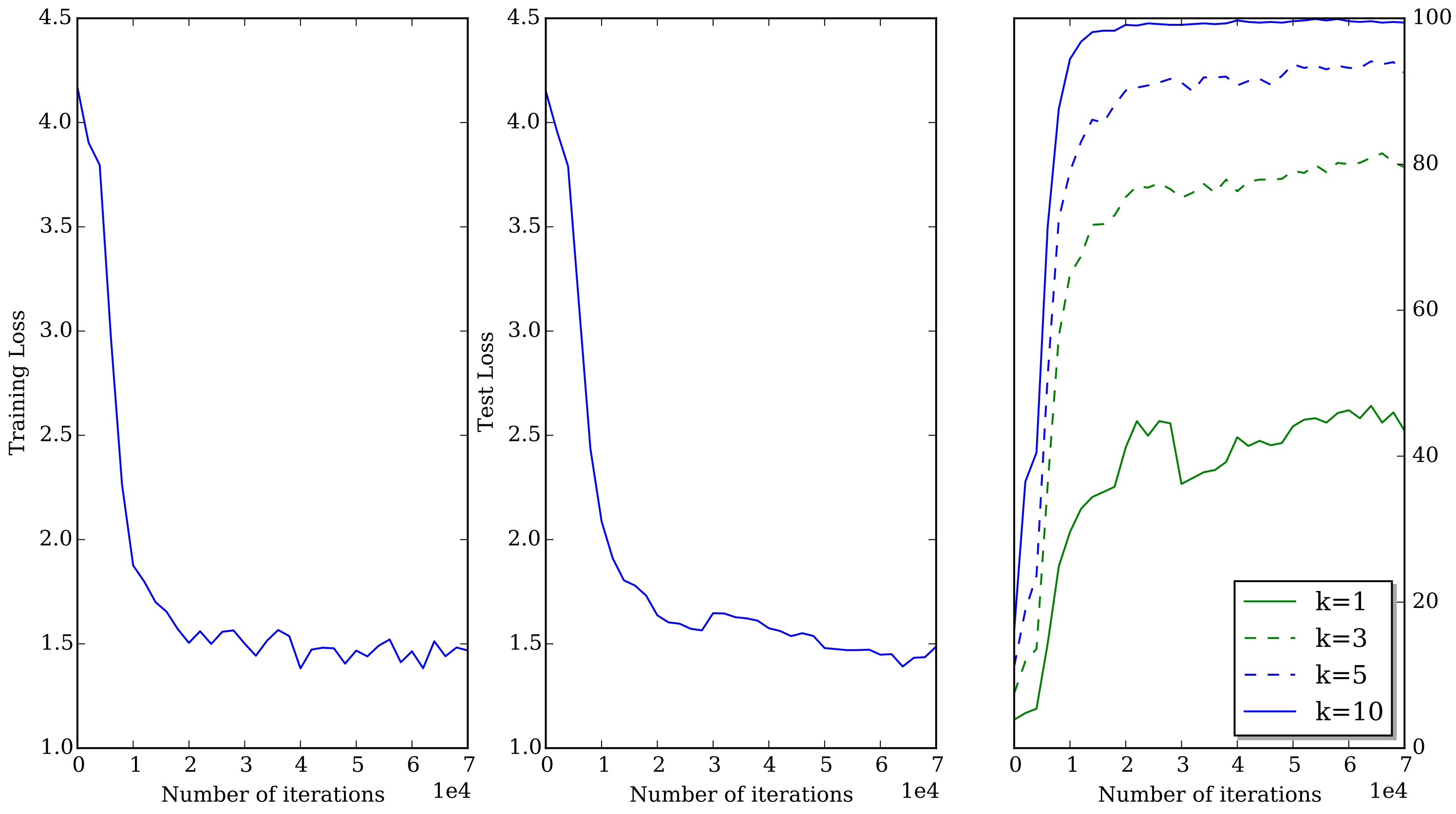


Figure 3.2 (a) Softmax-loss on training dataset (b) Softmax-loss on testing dataset; (c) Accuracies at k=1,3,5,10 on testing dataset.

3.3 Performance after masking

We hypothesize that the model will eventually learn the rules of chess through observation only. In this section, we will try to demonstrate how good our model performs in learning the rules of the game of chess.

Since the model has no prior information of the rules of the chess game, we can expect the model to make some mistakes by predicting an illegal move. For a legal gameplay, we use the method of **masking** the probabilities. Masking is done at two levels due to the inherent design of our method. First, we zero the probabilities of the piece positions which do not contain a friendly piece. Secondly, we zero out the probabilities of the final positions where the chosen piece type cannot reach using a single valid move.

After masking the probabilities for each of the board positions, we compare our results with the unmasked predictions to demonstrate how well our model has learnt the rules of chess.

3.3.1 Rule learning and move legality

We discussed how injection of chess knowledge into predicted probabilities can ensure that we obtain legal moves while playing. However, we also hypothesized that the model eventually learns the rules of the game. We will demonstrate this further by comparing the probability distributions obtained before and after masking. We use two measures to calculate the similarity between two distributions– Chebyshev similarity and squared euclidean distance. The distance metrics are computed using the following expression:

$$D_{sqeuc}(p,q) = ||p-q||^2$$

$$D_{chebyshev}(p,q) = \max_i |p_i - q_i|$$

Both the measures are symmetric. Chebyshev distance is intuitively the maximum zeroed out probability value from the unmasked distribution, while the euclidean distance is a common measure to estimate the distance between the two probability vectors in the n-dimensional space (n=64 in our case). We didn't use KL-divergence, which is the most common measure to compute dissimilarity between probability distributions, since we have zero probabilities.

We run our experiments on a relatively small dataset of 314,740 boards and report the average squared euclidean distance, average Chebyshev distance between masked and unmasked probability distributions and the average number of illegal predictions in the table below.

	$\|\|p-q\|\|^2$	$\max_i \|p_i - q_i\|$	Illegal move	Avg Rank of actual move	
Model			%age	unmasked	masked
PIECE	3.09×10^{-8}	4.28×10^{-5}	0.0	2.06342060113	2.06341424668
PAWN	1.72×10^{-4}	5.82×10^{-4}	0.0045	1.08001395504	1.07996844947
ROOK	3.74×10^{-3}	1.37×10^{-2}	0.72	2.31513493742	2.28972185557
KNIGHT	1.74×10^{-5}	4.8×10^{-4}	0.0	1.44417866616	1.44410761652
BISHOP	3.94×10^{-3}	1.15×10^{-2}	0.47	1.83962121376	1.82638963959
QUEEN	5.49×10^{-3}	1.89×10^{-2}	1.23	2.52614094756	2.47512457486
KING	3.35×10^{-3}	3.82×10^{-3}	0.33	1.59097552371	1.5873805164

Table 3.2 Rule Learning
Distances between predicted probabilities and average percentage of illegal predictions

Model	Percentage Accuracy	
	before masking	after masking
Piece	56.11	56.11
Pawn	53.60	53.60
Rook	50.98	51.26
Knight	72.77	72.77
Bishop	59.89	60.13
Queen	47.99	48.42
King	64.49	64.76

Table 3.3 Rule Learning: Accuracies before and after masking.
In the table above, the accuracies of each of the models is compared before and after masking for the set of $314,740$ board positions.

In the tables 3.2 and 3.3, we noticed that the models predict a legal move almost every single time when we are not masking the probability distributions for legal moves. This is a very strong result since one of the hypothesis of our work was whether the dynamics of chess are caused by the interactions of the pieces and the patterns they exist in. The convolutional neural network based models, which have no prior knowledge of the game of chess, can predict a legal move every single time after a few tens of million games where only one in every 10^{36} board configurations has the exposed to the network. Conclusively, we can say that the networks which are trained on move optimality also optimize move legality, probably because of their extreme representation power.

3.3.2 Evaluating performance of full move prediction

Till now we evaluated our networks individually for piece and move prediction tasks. Now, we need evaluate our models in an ensembled fashion to predict the entire move, i.e. from and to position on the board. For this task, we evaluate predictions for each move as the games in our test dataset proceed. The plot in figure 3.3 shows the variation of accuracy of the correct move (piece and destination position combined) lying in top-k predictions (k=1,5,10,20,30) with the index of the move into the game. The plot in figure 3.4 shows the variation of mean accuracy for the correct move to lie in the top-k predictions (k is on the x-axis). The moves were

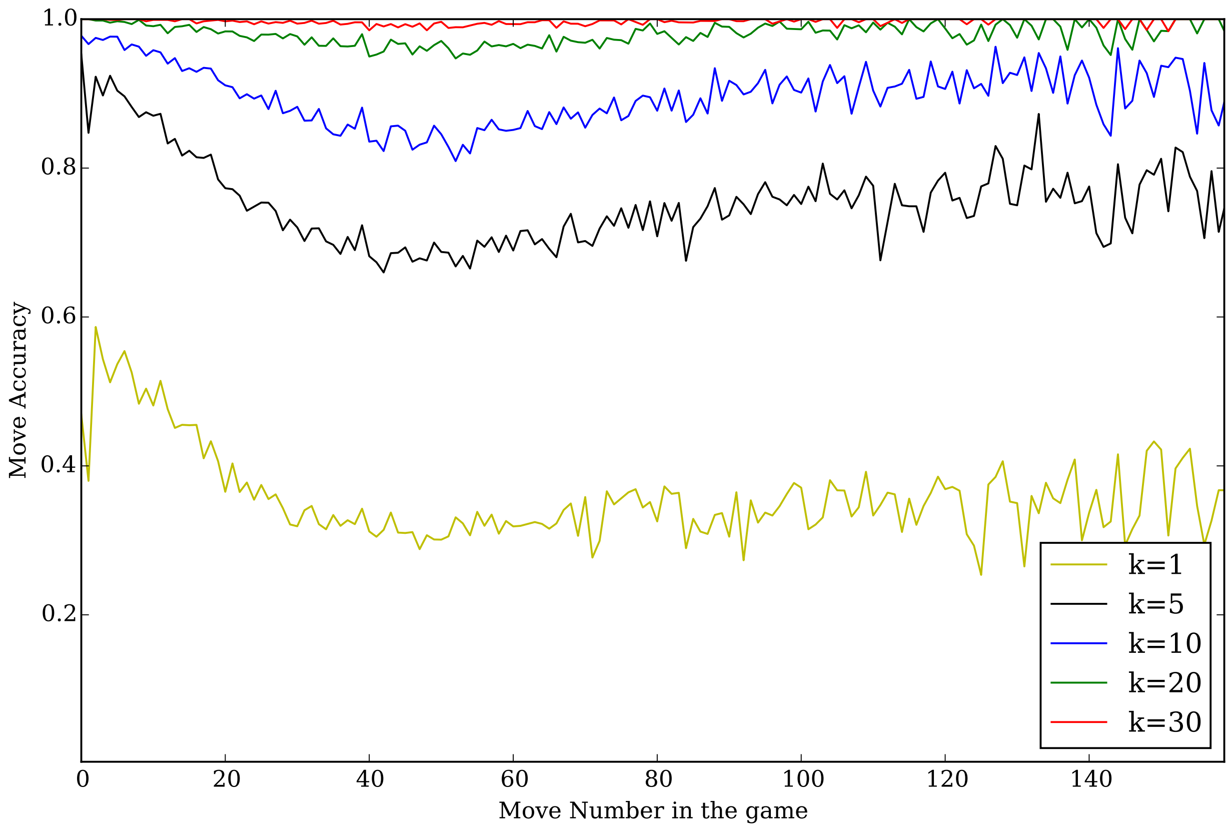


Figure 3.3 Average accuracies at different move numbers in test dataset games for top k predictions (k=1,5,10,30).

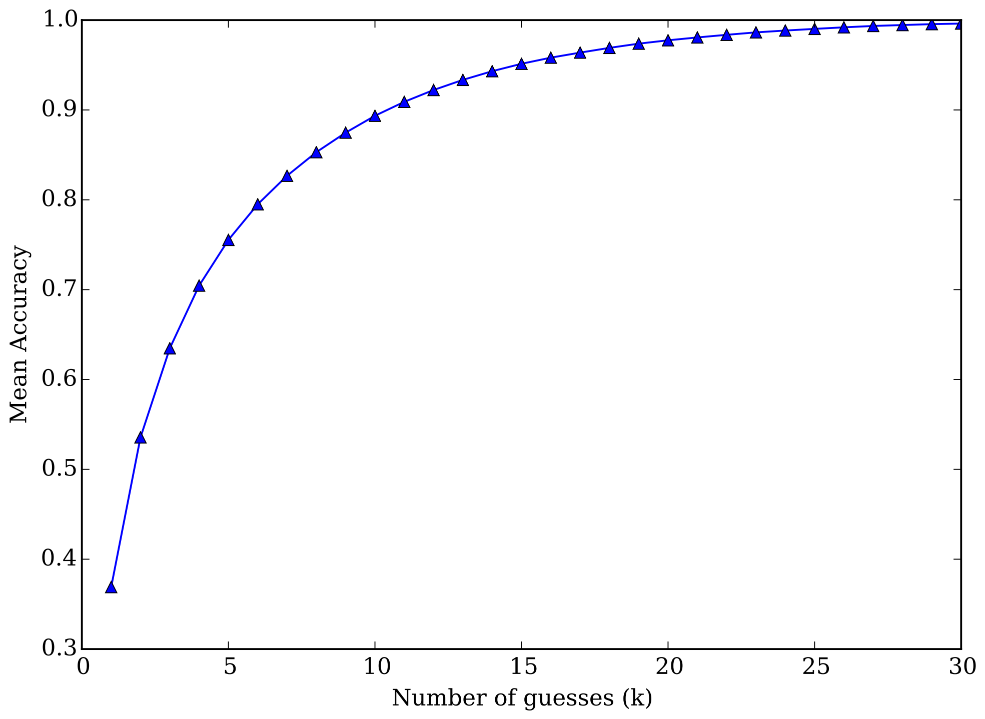


Figure 3.4 Plot of the variation of mean accuracy of the actual move lying in the top-k predictions with k (the number of guesses).

As we increase k from 1 to 30, the error diminishes to less than 1%. This means

1 in every hundred moves doesn't lie in top 30 of our predictions. We suspect that the error is caused, mostly in the middle game when a large number of moves are possible and the actual move played is a tactical one with a long term strategy in mind.

3.4 Sample moves taken by the network

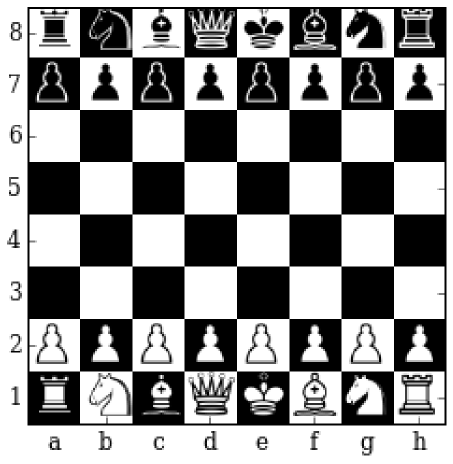
Here are a sample of moves predicted by the network for the board positions. We present two cases where the model was asked to predict the probability distributions for the "from" positions and "to" positions for some of the pieces on the board. In all the figures below, the chess board used as an input to the piece and move predictors. The 64-sized probability distributions are visualized as 8×8 matrices with a color bar on the right of each. The probability values shown in the move selector distribution are the joint probabilities of piece and move selection for that specific move.

3.4.1 Initial board position

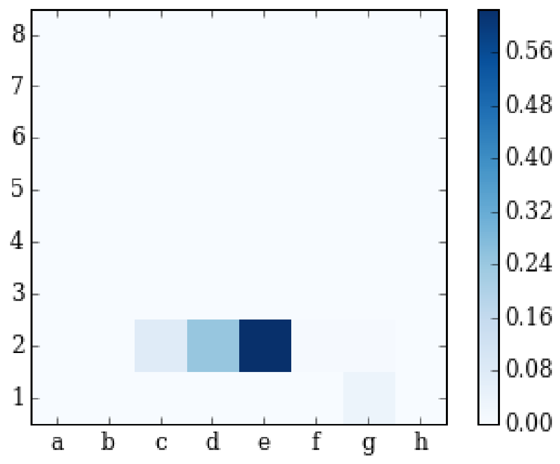
We evaluate our piece and network predictors on the initial board position of a chess game. Figure 3.5a shows the board position. Figure 3.5b shows the distribution of the probabilities for the piece to be moved. Note that only the pieces in the second rank and the knights in the last rank can move which is apparent in the predictions. Table 3.4 shows the comparison of the predicted probabilities and the percentage times the opening move was played in the dataset. Bobby Fischer said: "Best by test: 1.e4", and we shall see this.

Move	Predicted Probability	Number of times played	%age times played
All	1.0000000	4970725	100.00%
e2e4	0.6088475	2221439	44.7%
d2d4	0.2467637	1618291	32.6%
c2c4	0.0725896	286295	5.8%
g1f3	0.0334573	334741	6.7%
e2e3	0.0171652	62744	1.3%
f2f4	0.0059381	67248	1.4%
g2g3	0.0052637	93072	1.9%
b1c3	0.0021147	98306	2%
b2b3	0.0013837	64692	1.3%
c2c3	0.0013727	17449	0.4%
d2d3	0.0011852	43647	0.9%
g2g4	0.0008549	11245	0.2%
h2h3	0.0007792	5814	0.1%
b2b4	0.0007009	13761	0.3%
a2a3	0.0005240	5843	0.1%
g1h3	0.0002781	3875	0.1%
b1a3	0.0002716	1013	0%
h2h4	0.0002087	7216	0.1%
a2a4	0.0002071	4967	0.1%
f2f3	0.0000310	9067	0.2%

Table 3.4 Opening Move: The table shows the the moves predicted by the model alongside the predicted probability. The next two columns show the number of times and the percentage of times a certain opening was played according the the opening database on `http://www.ficsgames.org/openings.html`. The table is sorted on the probability predicted by our pair of networks.



(a) Board position(B)

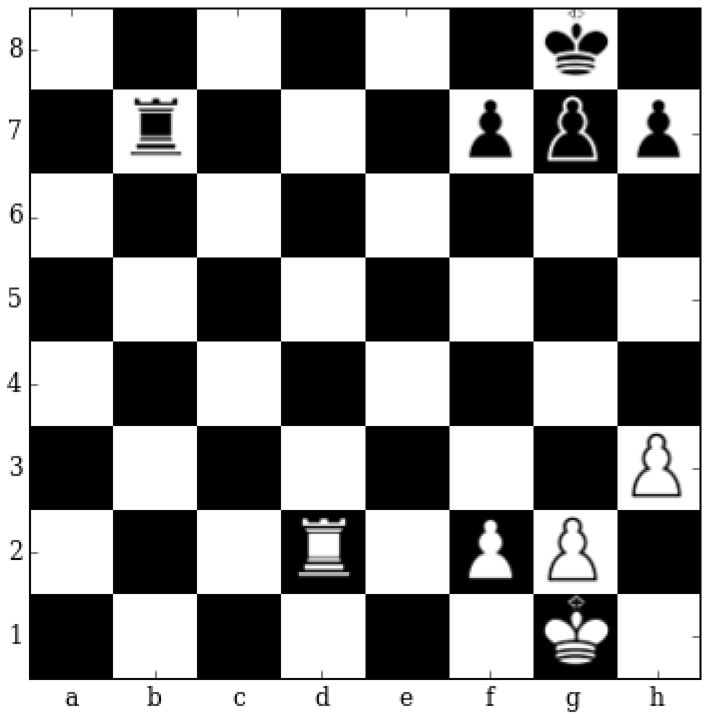


(b) $P_{\text{PIECE}}(B)$

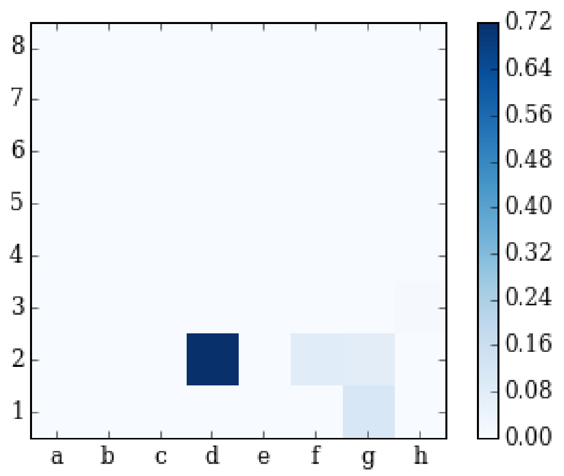
Figure 3.5 (a) The initial board position for a chess game, (b) The predicted probability distribution for the "from" piece by the PIECE network

3.4.2 Checkmating

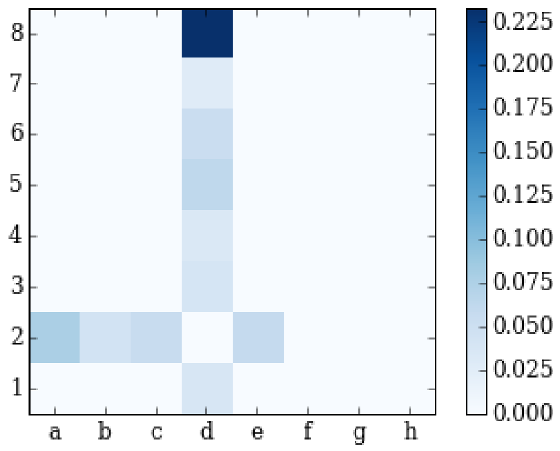
In figure 3.6, there is a checkmate possible in one move. We observe that the most probable move output by our model actually wins it for the white player.



(a) Board position. There is a check and mate possible in one move of the white.



(b) $P_{\text{PIECE}}(B)$

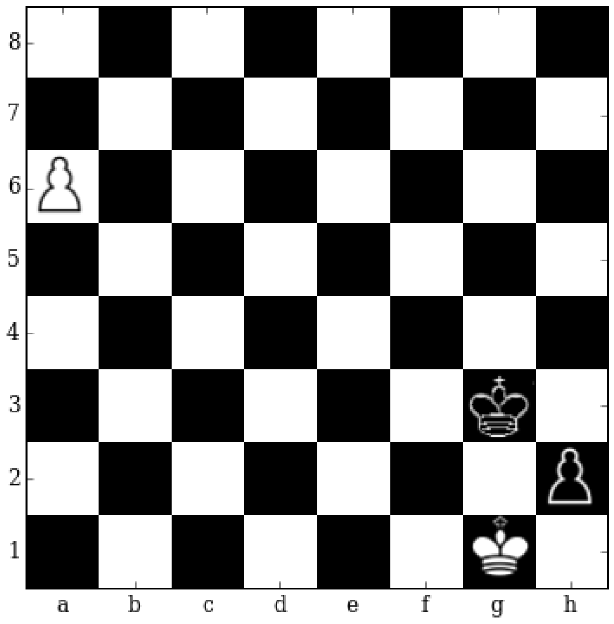


(c) $P_{\text{ROOK}}(B, d2)$

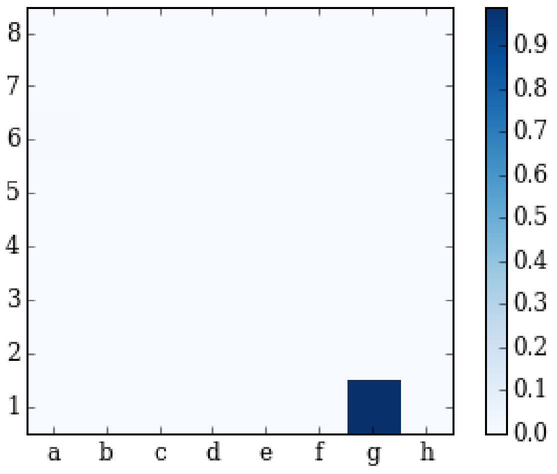
Figure 3.6 Predictions for a checkmate-in-1 position
(a) Black king is trapped in the last row. Moving the white rook at d2 to d8 will win the game for white. (b) The PIECE network predicts moving the rook at d2 with $p = 0.72$. (c) The ROOK network predicts moving the rook to d8 with a total probability of $p = 0.225$.

3.4.3 Detecting a check and blocking a promotion

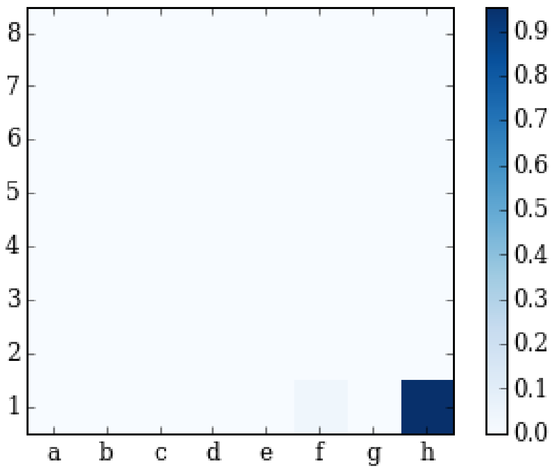
In figure 3.7, the king is under attack by the pawn, which is also seeking a promotion in the next move if the king moves away. The model successfully that the king should prevent the check by making the move **g1h1** which eventually also prevents the pawn promotion.



(a) Board position. The white king is under check.



(b) $P_{\text{PIECE}}(B)$

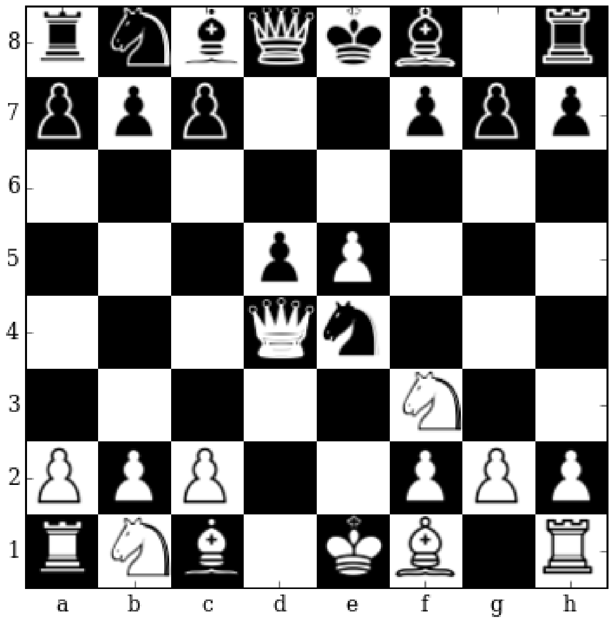


(c) $P_{\text{KING}}(B, g1)$

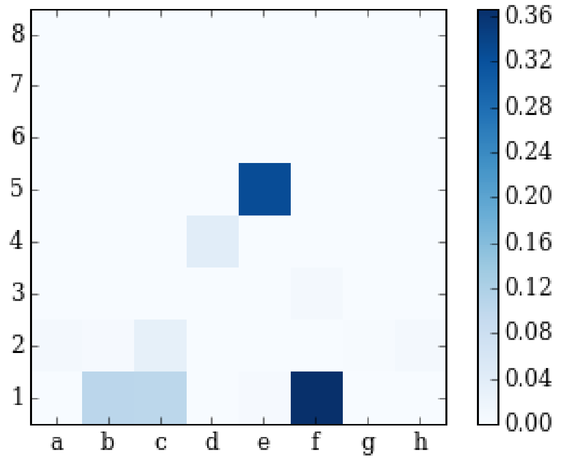
Figure 3.7 Prediction: Detecting a check
(a) White king is under a check. The black pawn at h2 is also looking for promotion in the next move. (b) The PIECE network predicts moving the king at g1 with $p > 0.95$. (c) The KING network predicts moving the white king to h1 with a total probability of $p > 0.9$.

3.4.4 En passant Move

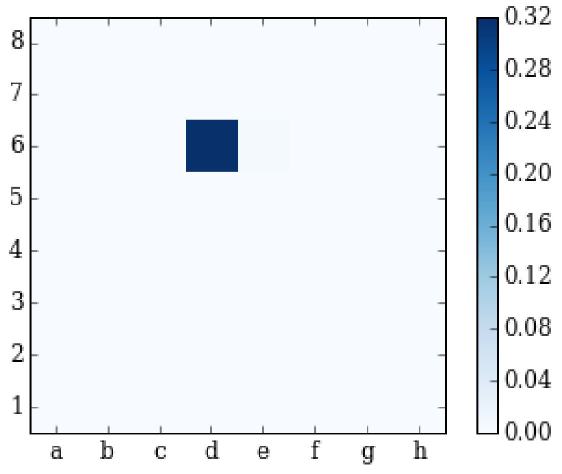
A very striking observation in our experiments was the the prediction of an en-passant move. It is interesting to note that an en-passant move is a very special case of pawn movement and attack, and we might expect it to appear only a very few number of times in our training dataset. However, its effectiveness is properly captured by the ensemble of piece and move selector networks.



(a) Board position. The d-pawn just moved 2 steps. En passant is possible



(b) $P_{\text{PIECE}}(B)$

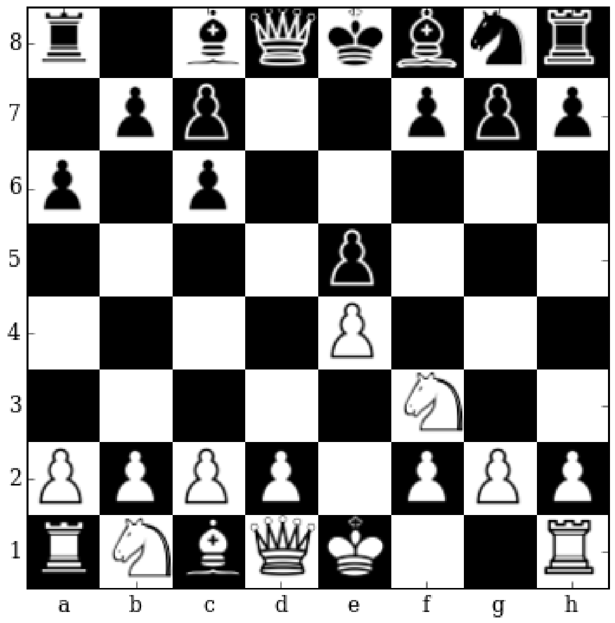


(c) $P_{\text{PAWN}}(B, e5)$

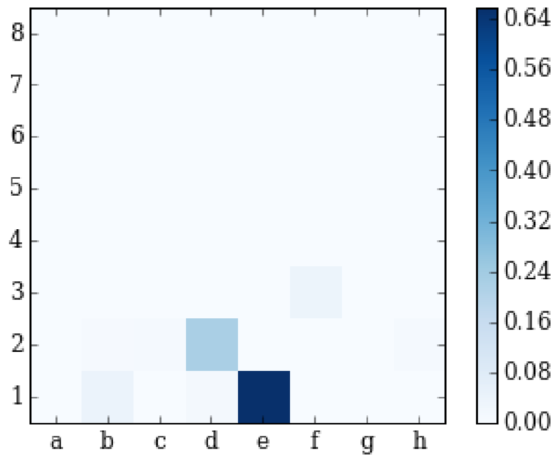
Figure 3.8 Prediction: En passant move
(a) An en-passant move is possible. (b) The maximum probable piece to move is the e5 pawn. (c) Unmasked probability has the en-passant move as the most probable move. During gameplay, if the black didn't move two steps in the last move, we can mask out this probability.

3.4.5 Castling

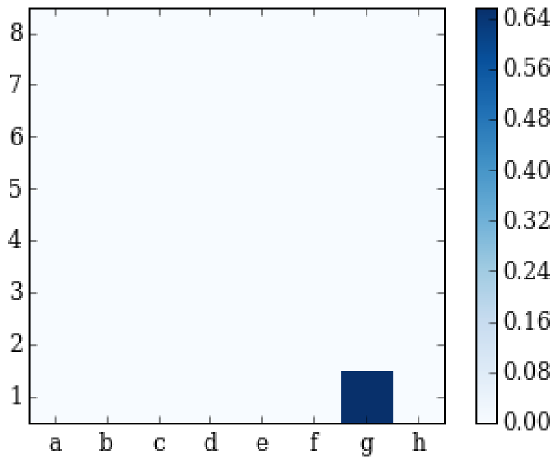
In figure 3.9 below, a castling move is predicted as the most favorable move.



(a) Board position. Castling is one of the favorable moves available.



(b) $P_{\text{PIECE}}(B)$



(c) $P_{\text{KING}}(B, e1)$

Figure 3.9 Prediction: Castling move
(a) Castling is available for the current board position (b) The PIECE network predicts moving the king at e1 with $p > 0.64$. (c) The KING network predicts moving the white king to g1 with a total probability of $p > 0.64$ completing the castling move.

3.4.6 Middle game behavior

Playing a move using the first guess is not ideal in a middlegame situation. Most of the players try to adopt tactics and long term strategies. However, our network trained on just 1 step outputs, may not be aware of such strategies. But, it is worth observing the behavior of our model when it comes to blunders by the opponents and we can take advantage of it.

We present here a game between Magnus Carlsen (white) and Viswanathan Anand (black), where Carlsen made a mistake at the 26th move while he easily had a good control of the game. Anand oversaw the better move Nex5! to play a4?. Anand later realized his mistake, but it was too late and Carlsen had already regained the control over the board.

We make our model predict the moves Vishwanathan Anand should have played after Carlsen's blunder. It is worth noting that the most favorable move as suggested by experts was Nex5 (i.e. g6e5), while Anand played a4 (i.e. a5a4). Our model predicts that the more favorable move be played with a much higher probability (around 6 times) than the actual move. Since it is not the best ranked move in our model's prediction, this also demonstrates the need to use search along with a prediction model to play a better game especially during the middle games.

The commentary is available at:
http://en.chessbase.com/post/sochi-g6-carlsen-won-anand-missed-big-chance
Most of the commentators following the game called Anand's move 26 a missed opportunity. A tweet said– *"Wow 26...Ne5! and black is winning....Anand plays 26...a4? Whats going on :) #CarlsenAnand"*

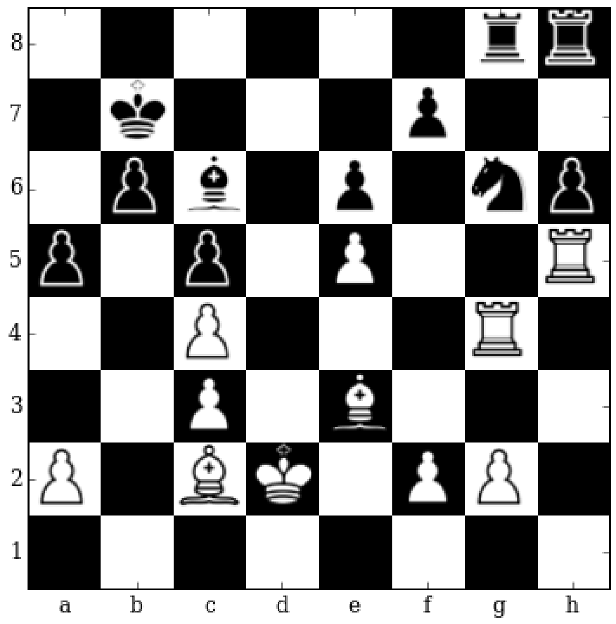


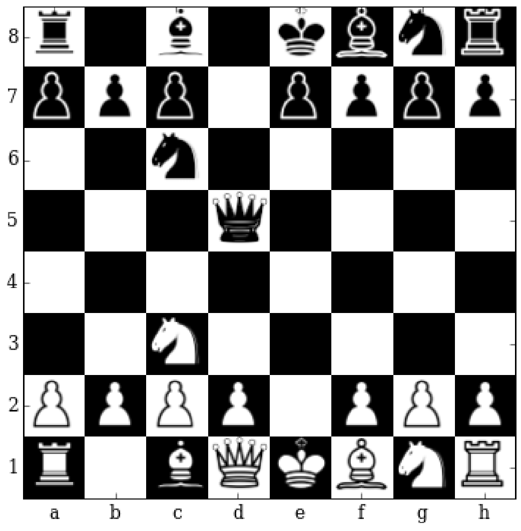
Figure 3.10 Black to move. 26th move. 6th game of the World Championship match, 2014– Magnus Carlsen vs Viswanathan Anand

Move	Predicted Probability	
g8d8	0.252880722284	
g8g7	0.148237019777	
g6e5	0.13111974299	Expected Move
b7c7	0.0660261586308	
h8h7	0.0471959412098	
c6g2	0.0403394699097	
c6e8	0.0358173549175	
g8c8	0.0323895104229	
b7c8	0.0302288047969	
c6d7	0.0250529013574	
a5a4	0.0231475103647	Vishwanathan Anand's actual move
g6e7	0.0229441132396	
b7a6	0.0208601523191	
g8a8	0.0198916308582	
b7b8	0.0190593209118	
c6a4	0.0153007712215	
g6f8	0.0150824002922	
g8f8	0.0115283448249	
g8e8	0.00727259740233	
b7a7	0.00666899653152	

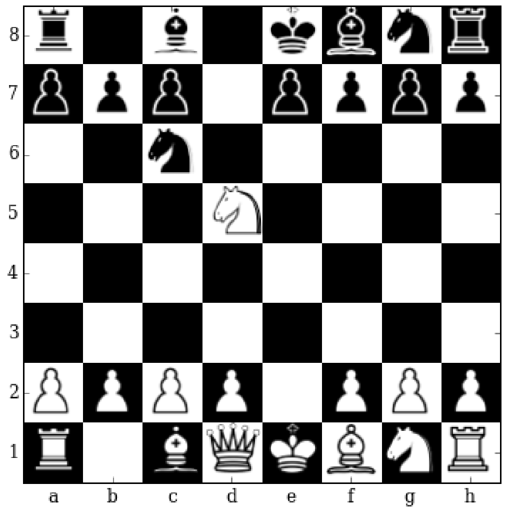
Table 3.5 Possible moves after the mistake by Carlsen in Move 26

3.5 Board Evaluations

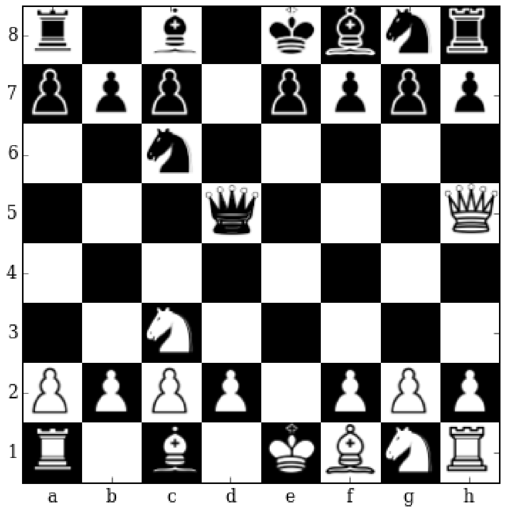
In this section we will look at some of the evaluation values ($V_\gamma(board)$) predicted by the CNN trained to learn an evaluation function (V_γ) as described in chapter 2. For each board position, we present boards with the highest evaluation values that can be reached from the current board position through a legal move. For probity, we also include the moves that might leave or put the king in check, but are otherwise valid. We also include legal castling moves.



(a) Board position (White to move)

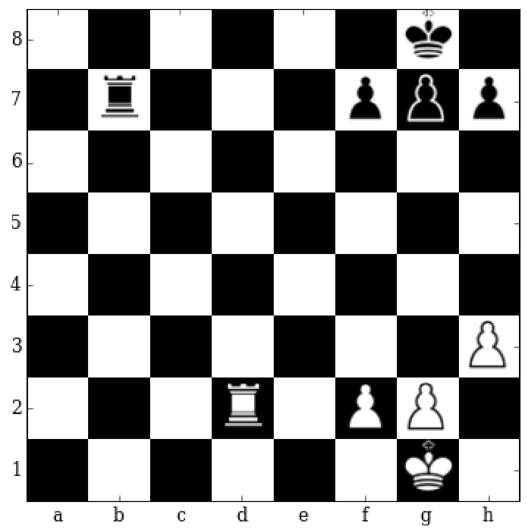


(b) Move: c3d5
$V_{\gamma=0.7} = 0.0545$

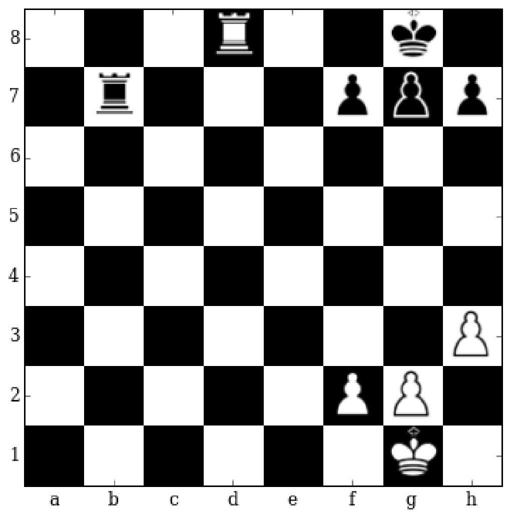


(c) Move: d1h5
$V_{\gamma=0.7} = 0.0144$

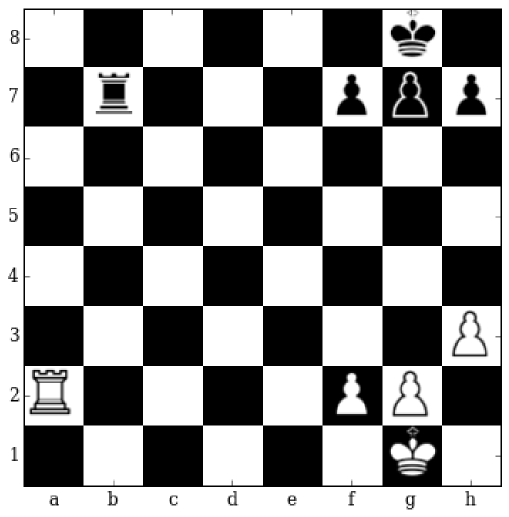
Figure 3.11 Evaluation function: Capturing a Queen with a knight
(a) Black has made a mistake by moving the queen to an open position where it is under attack from a white knight; (b)-(c) The top two evaluated boards reachable from the current position through a legal move.



(a) Board position (White to move)

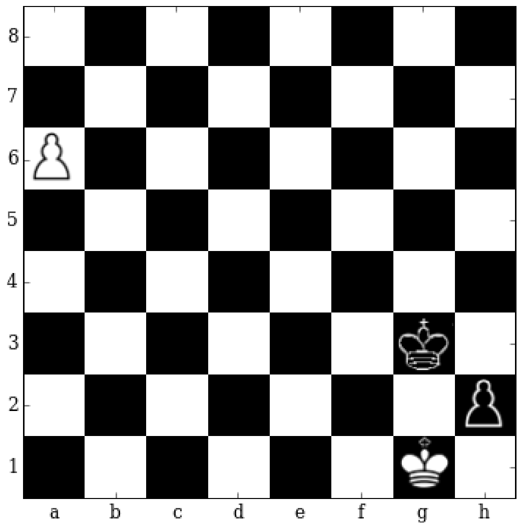


(b) Move: d2a8
$V_{\gamma=0.7} = 0.0543$

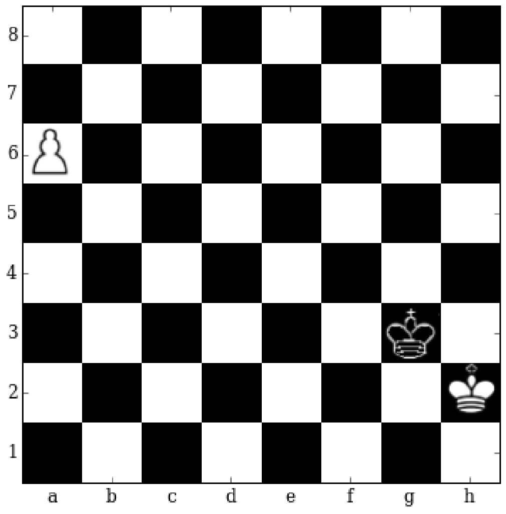


(c) Move: d2a2
$V_{\gamma=0.7} = 0.0177$

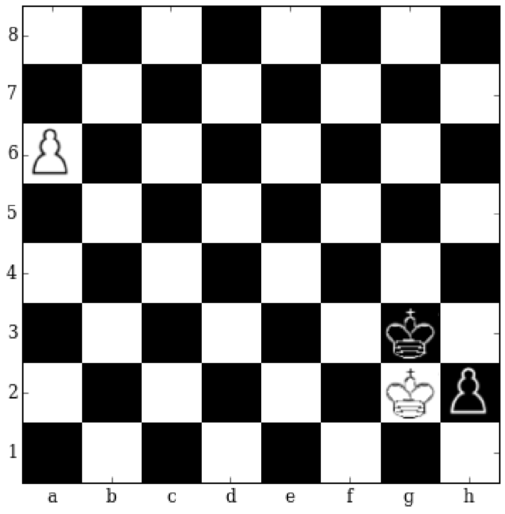
Figure 3.12 Evaluation function: Checkmate in one move
(a) A checkmate is possible in just one move of white; (b) The board with highest evaluation score in one move distance from the current board. It is a check and a mate. (c) The second best move predicted by the evaluation function. The value is less than 0.4 times the highest value.



(a) Board position (White to move)



(b) Move: g1h2
$V_{\gamma=0.7} = 0.0216$



(c) Move: g1g2
$V_{\gamma=0.7} = 0.0084$

Figure 3.13 Evaluation function: Check detection and promotion prevention
(a) White to move. The white king is currently under check by the black pawn at h2. If the king moves away, the pawn will promote to a queen and can wrap up the game easily afterwards; (b) The board with the highest evaluation is the one where the king captures the pawn. (c) The board with the second highest evaluation is the one where the king puts itself into check.

There are a couple of interesting observations that we can make from the few examples above:

1. **Piece Capture** In figures 3.11 and 3.13, the boards with highest value in the next move have one less opponent piece than the preceding board i.e. a white piece captures a black piece. This is not the case in figure 3.12, probably because there was no direct piece capture possible.

2. **Board value is analogous to sum of piece values** In most of the examples above, we can observe a similar trend– the board is evaluated in a way similar to a function that computes the difference in the values of the pieces remaining for both the players. We will probe further into this observation further in 3.5.1 where we see the correlation between this and a common piece evaluation system.

3.5.1 Correlation with the Material heuristic

To investigate the properties of the evaluation function, we computed the correlation of the values given by $V_\gamma(board)$ with the following heuristically derived evaluation function ($V_{\text{MATERIAL}}(board)$):

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

Table 3.6 The most common assignment of piece valuations

Some variations value the king highly, we do not use any such assignment because kings are present for both the players during the whole course of the game and hence won't impact our evaluation as such.

So the function V_{MATERIAL} is as follows:
$V_{\text{MATERIAL}}(board) = 1\times(P-P')+3\times(N-N')+3\times(B-B')+5\times(R-R')+9\times(Q-Q')$
where P, N, B, R, Q represent the number of pawns, knights, bishops, rook, queen respectively for the white player and P', N', B', R', Q' refer to their black counterparts.

The value of Pearson's correlation coefficient[1] we obtain for a set of $665,727$ boards between evaluation done with the model $V_{\gamma=0.7}$ and V_{MATERIAL} is **0.8194**. Figure 3.14 plots the evaluation function values for two functions for different boards.

[1]Pearson's correlation coefficient (ρ) between two random variables is also referred to as the *product moment* i.e. the mean of the product of the adjusted random variables. It can be computed using the formula: $\rho = \dfrac{E[(X-\mu_X)(Y-\mu_Y)]}{\sigma_X\sigma_Y}$

The correlation seems evident from the scatter plot. We also consider some example boards in figure 3.15. The boards with a positive material value for white have a positive evaluation in our function. It is worth noting that in figure 3.15d, white has almost won and it has a board value of 20 and an evaluation function suggests that white is approximately 3 moves away from winning ($(0.7)^3 = 0.34$).

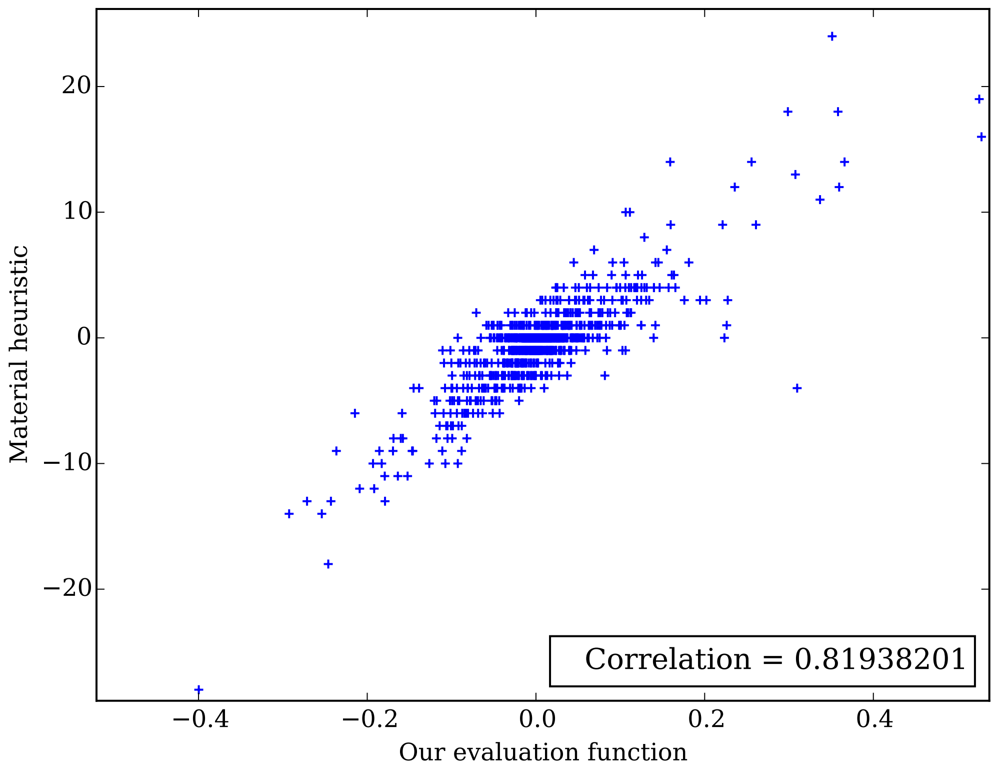
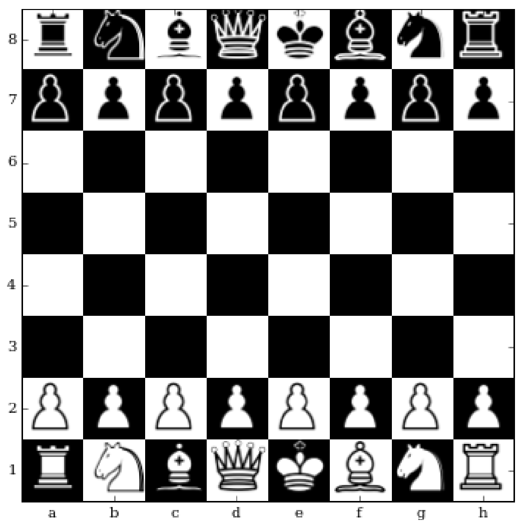
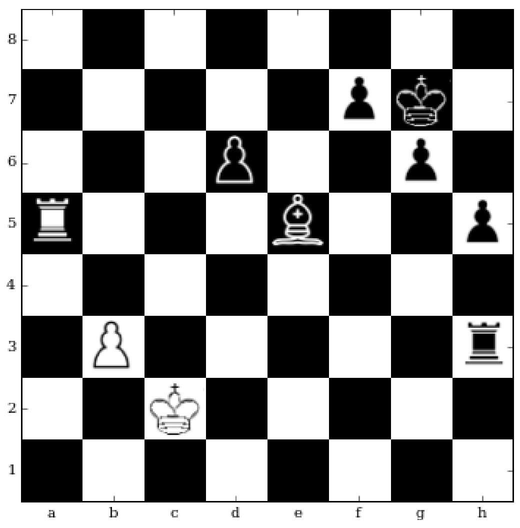


Figure 3.14 Scatter plot showing $V_{\gamma=0.7}$ and V_{MATERIAL} values for different boards in the test dataset

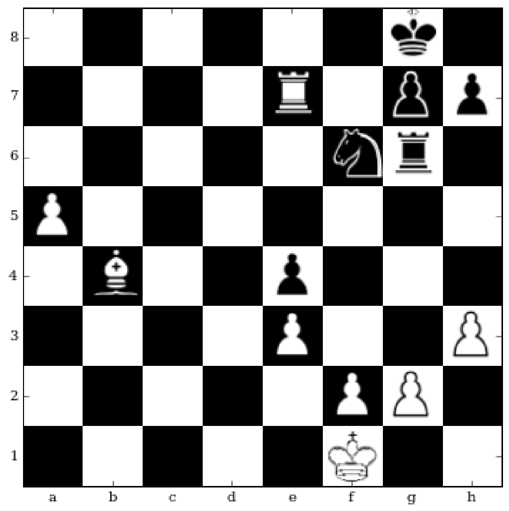
While a correlation coefficient of 0.8194 is pretty impressive, it doesn't actually imply a good evaluation function. Rather it provides a significant evidence of the nature of the evaluation function learned by our CNN based architecture without any prior knowledge of the importance of pieces or even any knowledge of the rules. The difference however is that the CNN based evaluation function is much more complex and makes use of shared weights to score local patterns to evaluate the complete board. This is what motivated to solve a regression problem using a convolutional



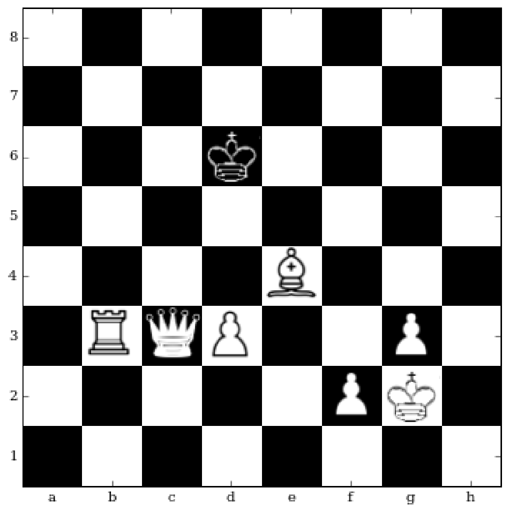
(a) $V_{\gamma=0.7} = 0.0015796$
$V_{\text{MATERIAL}} = 0.0$



(b) $V_{\gamma=0.7} = -0.0748723$
$V_{\text{MATERIAL}} = -6.0$



(c) $V_{\gamma=0.7} = 0.0902274$
$V_{\text{MATERIAL}} = 2.0$



(d) $V_{\gamma=0.7} = 0.302926$
$V_{\text{MATERIAL}} = 20.0$

Figure 3.15 Comparing the evaluation functions $V_{\gamma=0.7}$ and V_{MATERIAL}

neural network.

Although there are significant variations to the basic evaluation scheme we compare here, like the ones that score single pieces or pairs of pieces on the basis of its rank and file and/or the number of liberties, we omit comparing our evaluation function against them since our aim was just to present a idea of the similarity between such a set of hand-crafted heuristically defined evaluation functions.

3.5.2 Game Trajectories

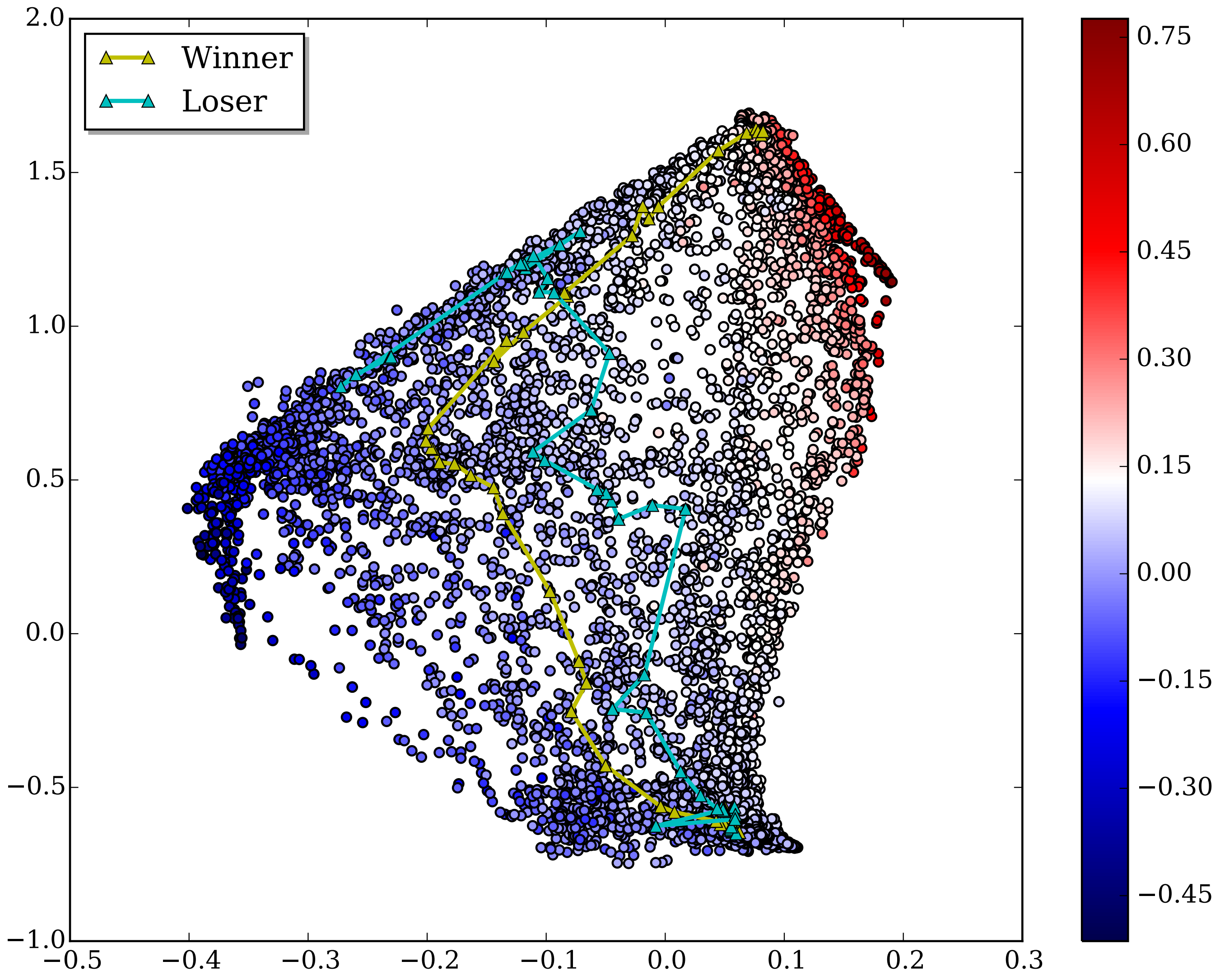


Figure 3.16 t-SNE embedding of the activations at the last fully connected layer of the board evaluation CNN. The red end is the set of boards close to winning, the blue end is the set of boards close to losing a game. We plot a game on the embedding. The winner(yellow) ends on the red side of the embedding while the loser(cyan) ends on the blue side of the embedding

As an analysis to the evaluation function proposed, we tried to observe the evaluations of the boards observed in a few test set games. In figure **??**, the scatter plot shows a set of 10,000 boards from our test dataset embedded onto a two dimensional embedding of the activations caused at the last fully connected layer of the board evaluation CNN using t-SNE (van der Maaten and Hinton, 2008). The game trajectories i.e. the path of the boards as seen on the two dimensional embedding for both the winning and the losing players is shown on the plot. A description of the game could be that the two players had boards with similar evaluation before one of them starts to win i.e. gets to see boards with higher positive evaluation values.

3.6 Gameplay

In this section we describe and discuss how well our models do when playing against a computer. It needs to be emphasized again that the primary motivation of our work is not the build the strongest chess playing system, but to show as a proof of concept that the convolutional neural network architecture can indeed learn how to play chess and also make strong predictions.
We already demonstrated the strengths and weaknesses of both our models on various cases in a game of chess. Since the actual gameplay is tougher than solving individual cases even with high accuracy, we may expect our system to make blunders which makes it lose matches. It is apparent that a heuristic search with a large amount of chess knowledge can help make the system robust to blunders.

Since our entire source code is in *Python*, we wanted to choose a Python based chess playing system which was easy to integrate with our prediction and evaluation system rather than be in an arms race with the best systems like Rybka (**?**) and Stockfish, which use optimizations and tweaks upto the level of memory addresses and assembly code generation. We went for Sunfish (Ahle, 2015), which is a short and lucid python chess program which implements MTD-f for search. We also utilize the API like structure of Sunfish to inject the minimal chess knowledge required for our system and hence play the predicted moves.

Method Used	Games Played	Won	Drawn	Lost	Details
Top-Prob	73	7	20	46	$10 \leq maxn \leq 1000$
TopProb-Negamax	19	3	4	12	$10 \leq maxn \leq 100$
Evaluation function ($V_{\gamma=0.7}$) with Negamax	21	6	4	11	2 <Negamax depth <5
Evaluation function ($V_{\gamma=0.7}$) with Negamax	25	16	2	7	Negamax depth=4

Table 3.7 The table shows the result statistics for evaluation of gameplay against sunfish. We test our models under different conditions. In most of our experiments we limit the number of nodes explored by Sunfish between 10 to 1000 (chosen randomly on a log scale). For other deviations, the details are mentioned in the details column

The fact that our models do not perform well against Sunfish, that too with limited search capability, is not disheartening. Looking at the implementations of some of the best-known chess computers with all the bitmap optimizations and computations, we believe that there is still a long way in a Convolutional neural network becoming the primary backend of such a system. As we already emphasized enough, that the aim of our work has never been to be in an arms race with the best known chess computers around, but to provide a proof of evidence that convolutional neural networks can indeed play a game of chess.

It is worth mentioning that examples like the model learning to predict only the legal moves at any board position is a strong result in itself. Further, the analysis shown earlier in the chapter (sections 3.4, 3.5) shows positive outcomes for even some of the tricky cases. Also, during the gameplay it was observed that a proper pawn structure, castling move and defense for the king were prominent features of most of the games. This strengthens the evidence this work provides in human-like chess players with generalization and pattern recognition capability. We discuss about these strengths, possible reasons for the weaknesses and propose extensions of this work in the next chapter.

Chapter 4

Conclusion and Discussion

We started with an aim to make a machine learn how to play chess giving it no or minimal prior knowledge. In the previous chapter, we saw a mannerly analysis of the results we obtain when convolutional neural networks are used to learn the game of chess. Although showing a promising behavior in various case studies performed, we saw that the gameplay is not very effective against a decent chess computer that uses heuristic search to choose a move. But, we believe that the work significantly bridges the gap between the cognitive aspects of chess playing and how the best chess computers play it. In this chapter, we discuss the contributions of our work and alongside try to ponder upon the shortcomings and discussing possible solutions.

4.1 Positives

We started with a motive to make a machine learn the game of chess with minimal prior knowledge and eventually have the ability to:

- Learn to play legal moves
- Rank the possible moves without any explicit guidance on relative importance of material or position
- Evaluate board positions

Motivated to accomplish this task using minimal knowledge prior, we used convolutional neural network based architectures to learn– the piece and move predictors as well as an evaluator. In the last section, we empirically demonstrated that the models learned the rules of the chess as well were able to make strong predictions on tasks where a novice could easily fail.

4.2 Weaknesses

We have already emphasized that our motivation was not to build a state of the art chess playing system, but to build a chess player from scratch i.e. having minimal prior knowledge. We saw in the last chapter that the gameplay of our system is not a championship level one. But it still could make strong predictions on certain tasks. However in this section, we will try to focus on the weaknesses of our models.

4.2.1 Reasons

The convolutional neural network trained on all the moves doesn't know if the given move at hand is a blunder or not. It can be the case that a common blunder is mistaken to be a favorable move in certain situations or if a board position is so rare that the only available supervision is not the best move to make in that situation.

However this can be explained to be considered while learning the evaluation function using discounted rewards. The problem with the evaluation function seems to be that the learning task is not very specific or well formulated. A visible weakness is the already present ambiguity in the training data for the network for instance the starting board, which looks the same for both the winning and the losing player, gets both a positive and a negative score. Although the value is small or can be made making a proper use of the decay parameter.

4.3 Further Work

4.3.1 Reinforcement Learning based player

In chapter 1, we mentioned that it is a hard problem to do a policy search using a reinforcement learning architecture when we have a combination of complex dynamics and complex policy. Intuitively, why a reinforcement learning agent (RL agent) is a bad idea for chess is that a reward is received only once and after a long time. Hence, a learner who starts from scratch, using a random policy, would never reach the point when it receives a positive reward i.e. a win. There has to be a heuristic evaluation of the policy learned by such a system which allows it to update its policy. It is exactly at this point where the evaluation function learned from a large dataset of already played games can fit. The evaluation function (described in 2.4) computed at a certain ply depth can provide the board values as a feedback to the current policy of the RL agent.

4.3.2 Evolving chess programs using Genetic Algorithms

Another class of systems that use co-evolution, where the system learn by playing itself (Vázquez-Fernández et al., 2012) and adjusting the parameters (Bošković and Brest, 2011). A possible extension of this work could be to utilize such genetic programming architectures to improve the basic piece-move predictor presented in this thesis.

4.4 Conclusion

Through our work, we presented a novel proof of evidence that cognitively inspired architectures rather than the conventional chess playing systems can also be used to build chess systems. Such chess systems have a more human way of playing chess and do not require much knowledge about the rules of chess too. We demonstrated the strengths and weaknesses of our work and discussed some extensions which could lead to better chess playing systems in the future.

Bibliography

(Ahle, 2015) Ahle, T. D. (2015). Sunfish. `https://github.com/thomasahle/sunfish`. (pages v and 57)

(Barto et al., 1989) Barto, A. G., Sutton, R. S., and Watkins, C. J. C. H. (1989). Learning and sequential decision making. In *LEARNING AND COMPUTATIONAL NEUROSCIENCE*, pages 539–602. MIT Press. (page 28)

(Baxter et al., 1999a) Baxter, J., Tridgell, A., and Weaver, L. (1999a). Knightcap: a chess program that learns by combining td (lambda) with game-tree search. *arXiv preprint cs/9901002*. (page 1)

(Baxter et al., 1999b) Baxter, J., Tridgell, A., and Weaver, L. (1999b). Tdleaf (lambda): Combining temporal difference learning with game-tree search. *arXiv preprint cs/9901001*. (page 1)

(Beal and Smith, 1999) Beal, D. and Smith, M. (1999). Learning piece-square values using temporal differences. *ICCA JOURNAL*, 22(4):223–235. (page 1)

(Beal and Smith, 1997) Beal, D. F. and Smith, M. C. (1997). Learning piece values using temporal differences. *Journal of The International Computer Chess Association*, 20(3):147–151. (page 1)

(Bengio et al., 2015) Bengio, Y., Goodfellow, I. J., and Courville, A. (2015). Deep learning. Book in preparation for MIT Press. (pages 16 and 17)

(Bilalić et al., 2007) Bilalić, M., McLeod, P., and Gobet, F. (2007). Does chess need intelligence?a study with young chess players. *Intelligence*, 35(5):457–470. (page 6)

(Bošković and Brest, 2011) Bošković, B. and Brest, J. (2011). Tuning chess evaluation function parameters using differential evolution algorithm. *Informatica*, 35(2). (pages 2 and 61)

(Campbell et al., 2002) Campbell, M., Hoane, A. J., and Hsu, F.-h. (2002). Deep blue. *Artificial intelligence*, 134(1):57–83. (pages 1 and 5)

(Chase and Simon, 1973) Chase, W. G. and Simon, H. A. (1973). Perception in chess. *Cognitive psychology*, 4(1):55–81. (pages 5 and 6)

(Chollet, 2015) Chollet, F. (2015). Keras. `https://github.com/fchollet/keras`. (page 34)

(Ciresan et al., 2012) Ciresan, D., Meier, U., and Schmidhuber, J. (2012). Multi-column deep neural networks for image classification. In *Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on*, pages 3642–3649. IEEE. (pages 2 and 17)

(CNN, 1997) CNN, n. (1997). Deep blue vs kasparov: Rematch. (page 6)

(De Groot et al., 1996) De Groot, A. D., Gobet, F., and Jongman, R. W. (1996). *Perception and memory in chess: Studies in the heuristics of the professional eye.* Van Gorcum & Co. (pages 4, 6, and 33)

(Epstein, 2001) Epstein, S. L. (2001). Learning to play expertly: A tutorial on hoyle. *Machines that learn to play games*, pages 153–178. (pages 9 and 17)

(Ericsson et al., 2006) Ericsson, K. A., Charness, N., Feltovich, P. J., and Hoffman, R. R. (2006). *The Cambridge handbook of expertise and expert performance.* Cambridge University Press. (page 5)

(Ericsson et al., 2007) Ericsson, K. A., Prietula, M. J., and Cokley, E. T. (2007). The making of an expert. *Harvard business review*, 85(7-8). (page 5)

(Gobet and Clarkson, 2004) Gobet, F. and Clarkson, G. (2004). Chunks in expert memory: Evidence for the magical number four... or is it two? *Memory*, 12(6):732–747. (page 6)

(Guo et al., 2014) Guo, X., Singh, S., Lee, H., Lewis, R. L., and Wang, X. (2014). Deep learning for real-time atari game play using offline monte-carlo tree search planning. In *Advances in Neural Information Processing Systems*, pages 3338–3346. (page 2)

(Hornik et al., 1989) Hornik, K., Stinchcombe, M., and White, H. (1989). Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366. (page 15)

(Hubel and Wiesel, 1963) Hubel, D. and Wiesel, T. (1963). Shape and arrangement of columns in cat's striate cortex. *The Journal of physiology*, 165(3):559–568. (page 16)

(Hyatt, 1999) Hyatt, R. M. (1999). Book learning-a methodology to tune an opening book automatically. *ICCA JOURNAL*, 22(1):3–12. (page 1)

(Jia et al., 2014) Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., Guadarrama, S., and Darrell, T. (2014). Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093.* (page 34)

(Krizhevsky et al., 2012) Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105. (pages 2 and 17)

(Levine et al., 2015) Levine, S., Finn, C., Darrell, T., and Abbeel, P. (2015). End-to-end training of deep visuomotor policies. *arXiv preprint arXiv:1504.00702.* (page 13)

(Maddison et al., 2014) Maddison, C. J., Huang, A., Sutskever, I., and Silver, D. (2014). Move evaluation in go using deep convolutional neural networks. *arXiv preprint arXiv:1412.6564*. (page 7)

(Mannen, 2003) Mannen, H. (2003). Learning to play chess using reinforcement learning with database games. *Phil. uu. nl/preprints/ckiscripties*, pages 11–34. (page 9)

(Mnih et al., 2013) Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. (2013). Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*. (pages 2, 6, 13, and 20)

(Oshri and Khandwala, 2015) Oshri, B. and Khandwala, N. (2015). Predicting moves in chess using convolutional neural networks. (page 8)

(Shannon, 1950) Shannon, C. E. (1950). Xxii. programming a computer for playing chess. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 41(314):256–275. (pages 9 and 10)

(Sutskever and Nair, 2008) Sutskever, I. and Nair, V. (2008). Mimicking go experts with convolutional neural networks. In *Artificial Neural Networks-ICANN 2008*, pages 101–110. Springer. (page 7)

(Sutton, 1988) Sutton, R. S. (1988). Learning to predict by the methods of temporal differences. *Machine learning*, 3(1):9–44. (page 28)

(Sutton and Barto, 1998) Sutton, R. S. and Barto, A. G. (1998). *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge. (page 12)

(Taigman et al., 2014) Taigman, Y., Yang, M., Ranzato, M., and Wolf, L. (2014). Deepface: Closing the gap to human-level performance in face verification. In *Computer Vision and Pattern Recognition (CVPR), 2014 IEEE Conference on*, pages 1701–1708. IEEE. (pages 2 and 17)

(Thrun, 1995) Thrun, S. (1995). Learning to play the game of chess. *Advances in neural information processing systems*, 7. (page 9)

(Tompson et al., 2014) Tompson, J., Goroshin, R., Jain, A., LeCun, Y., and Bregler, C. (2014). Efficient object localization using convolutional networks. *arXiv preprint arXiv:1411.4280*. (pages 2 and 17)

(van der Maaten and Hinton, 2008) van der Maaten, L. and Hinton, G. (2008). Visualizing data using t-sne. *The Journal of Machine Learning Research*, 9(2579-2605):85. (page 57)

(Vázquez-Fernández et al., 2012) Vázquez-Fernández, E., Coello, C. A. C., and Troncoso, F. D. S. (2012). An evolutionary algorithm coupled with the hooke-jeeves algorithm for tuning a chess evaluation function. In *Evolutionary Computation (CEC), 2012 IEEE Congress on*, pages 1–8. IEEE. (pages 2 and 61)

(Zermelo, 1913) Zermelo, E. (1913). Über eine anwendung der mengenlehre auf die theorie des schachspiels. In *Proceedings of the fifth international congress of mathematicians*, volume 2, pages 501–504. II, Cambridge UP, Cambridge. (page 9)