

# Teaching Computers to Play Chess Through Deep Reinforcement Learning

Dorian Van den Heede

Supervisors: Prof. dr. ir. Francis wyffels, Prof. dr. ir. Joni Dambre

Counsellors: Ir. Jeroen Burms, Prof. dr. ir. Francis wyffels, Prof. dr. ir. Joni Dambre

Master's dissertation submitted in order to obtain the academic degree of  
Master of Science in Computer Science Engineering

Department of Electronics and Information Systems  
Chair: Prof. dr. ir. Rik Van de Walle  
Faculty of Engineering and Architecture  
Academic year 2016-2017





# Teaching Computers to Play Chess Through Deep Reinforcement Learning

Dorian Van den Heede

Supervisors: Prof. dr. ir. Francis wyffels, Prof. dr. ir. Joni Dambre

Counsellors: Ir. Jeroen Burms, Prof. dr. ir. Francis wyffels, Prof. dr. ir. Joni Dambre

Master's dissertation submitted in order to obtain the academic degree of  
Master of Science in Computer Science Engineering

Department of Electronics and Information Systems  
Chair: Prof. dr. ir. Rik Van de Walle  
Faculty of Engineering and Architecture  
Academic year 2016-2017





## Foreword

*After a long and sporadically lonesome road, an incredible journey has come to an end in the form of this master dissertation. Exploiting experience from exploration is what makes one fail less and less.*

*I had the amazing opportunity to pursue my thesis in a subject I am extremely passionate about, for which I want to thank my supervisor Prof. Francis wyffels.*

*Last year and especially the last three months, I needed a lot of advice and help. Luckily, Jeroen Burms has always been a great counselor and helped me whenever I had a question or idea. Thank you.*

*I want to express my gratitude to 'grand papa et grand maman'. During the fortnight of my isolation, they spoiled me more than my facebook wall spoiled the Game of Thrones story.*

*Furthermore I want to show appreciation to my fellow students with whom I spent countless hours of hard labor. I am thanking Jonas for lifting me up, Loic for always remaining casual, Timon for always being equally happy and Vincent for infecting invincibility.*

*Closer to me, there are my parents whom I love very much. I am sincerely sorry if I do not show this enough.*

*Dorian Van den Heede, august 2017*

## Permission of use on loan

“The author(s) gives (give) permission to make this master dissertation available for consultation and to copy parts of this master dissertation for personal use. In the case of any other use, the copyright terms have to be respected, in particular with regard to the obligation to state expressly the source when quoting results from this master dissertation.”

Dorian Van den Heede, august 2017

# Teaching Computers to Play Chess with Deep Reinforcement Learning

Dorian VAN DEN HEEDE

Master's dissertation submitted in order to obtain the academic degree of  
MASTER OF SCIENCE IN COMPUTER SCIENCE ENGINEERING

Academic year 2016-2017

Supervisors: Prof. dr. ir. F. WYFFELS and Prof. dr. ir. J. DAMBRE

Counsellor: J. BURMS

Faculty of Engineering and Architecture

Ghent University

Department of Electronics and Information Systems

Chair: Prof. dr. ir. R. VAN DE WALLE

## Abstract

Chess computers have reached a superhuman level at playing chess by combining tree search with the incorporation of heuristics obtained with expert knowledge into the value function when evaluating board positions. Although these approaches add a big amount of tactical knowledge, they do not tackle the problem of effectively solving chess, a deterministic game. We try to set the first stone by solving relatively simple chess endgames from a human point of view with deep reinforcement learning (DRL). The proposed DRL algorithm consists of 2 main components: (i) self play (ii) translating the collected data in phase (i) to a supervised learning framework with deep learning. In light of this, we present a novel TD learning algorithm, TD-Stem( $\lambda$ ), and compare it with the state of the art, TD-Leaf( $\lambda$ ).

## Keywords

Chess, deep reinforcement learning, TD-learning

# Teaching Computers to Play Chess with Deep Reinforcement Learning

Dorian Van den Heede

Supervisor(s): Francis wyffels, Jeroen Burms, Joni Dambre

**Abstract**—Chess computers have reached a superhuman level at playing chess by combining tree search with the incorporation of heuristics obtained with expert knowledge into the value function when evaluating board positions. Although these approaches add a big amount of tactical knowledge, they do not tackle the problem of effectively solving chess, a deterministic game. We try to set the first stone by solving relatively simple chess endgames from a human point of view with deep reinforcement learning (DRL). The proposed DRL algorithm consists of 2 main components: (i) self play (ii) translating the collected data in phase (i) to a supervised learning (SL) framework with deep learning. In light of this, we present a novel temporal difference (TD) learning algorithm, TD-Stem( $\lambda$ ), and compare it with the state of the art, TD-Leaf( $\lambda$ ).

**Keywords**—Chess, deep reinforcement learning, TD-learning

## I. INTRODUCTION

IN the early 50s, scientists like Shannon and Turing were already musing about how a machine would be able to play chess autonomously [1], [2]. The enormous complexity ( $\sim 10^{120}$  different states) combined with its popularity and rich history made it the holy grail of artificial intelligence (AI) research in the second half of the 20<sup>th</sup> century. Research attained its peak in 1997 when *Deep Blue* succeeded at winning a duel with the human world champion at that time, *Gary Kasparov* [3]. Conventional chess computers now attain a superhuman level. Current state of the art (SOTA) engines use a combination of deep search trees with expert knowledge heuristics to evaluate positions at leaf nodes (*Komodo*, *Stockfish* and *Houdini* at time of writing) [4]. Furthermore, the search is heavily optimized as an attempt to reduce the influence of the exponentially increasing complexity when looking more moves ahead. The height of the search tree is the most crucial factor for the performance of an engine. These observations lead to the following shortcomings: (i) the strategic strength is entirely based on the heuristical evaluation at leaf nodes, which is humanly biased and suboptimal as a consequence (ii) conventional play by engines is very tactical as the reached depth is the most important variable for performance, resulting in a lack of intuitive game play (iii) as current engines mostly rely on human insight, they do not take steps at objectively solving the game, but more at how to beat humans with their advantage in calculation power. These downsides could eventually be solved by looking at the game in a primitive and unbiased viewpoint and using a combination of self play with supervised machine learning in a reinforcement learning (RL) framework. As a first step we have implemented a deep learning architecture where we use a novel TD-learning variant in combination with neural networks being fed by positions represented as bitboards.

This paper is organized as follows. In section II a brief overview of previous research in this domain is given. Section III introduces the used DRL framework after which we focus on how

different modifications of the well known TD( $\lambda$ ) algorithm may unite strategic and tactical information into the value function in section IV. The value function itself is discussed in V. We compare our novel algorithm with the SOTA TD-learning algorithm for chess in section VI and form an appropriate conclusion in section VII.

## II. RELATED WORK

Already in 1959, a TD-learning algorithm was used to create a simplistic checkers program [5]. The first real success story of using TD-learning (arguably for RL in general) and Neural network (NN) for value function approximation in boardgames was *Tesouro's* backgammon program *TD-Gammon* with a simple fully connected feedforward neural network (FNN) architecture learned by incremental TD( $\lambda$ ) updates [6]. One interesting consequence of *TD-Gammon* is its major influence on strategic game play of humans in backgammon today. As chess is a game where tactics seem to have the upper hand over strategic thinking, game tree search is used to calculate the temporal difference updates in TD-Leaf( $\lambda$ ) proposed by *Baxter et al.* The chess software *KnightCap* that came out of it used linear expert knowledge board features for function approximation and had learned to play chess up until master level by playing on a chess server. The same method was used in combination with a two layer NN initialized with biased piece value knowledge in *Giraffe* to reach grandmaster level, effectively proving the potential strength of TD-learning methods with search at a limited depth [8]. *Veness et al.* went even further and stored as much information from game tree search during self play as possible in *Meep* [9].

The potential of using millions of database games between experts to learn to play chess has been exploited in *DeepChess* [10].

In RL research concerning other games, convolutional neural networks (CNNs) have already been proven its worth. A lot of research has been undertaken to Atari video games, which makes sense as CNNs were originally designed for computer vision tasks [11]. The current SOTA is a dual network learning the advantage function with double deep Q-learning [12]. The first successful application of CNN into boardgames is *AlphaGo*, which is the first *Go* program to beat a player with the highest possible rating [13].

## III. DEEP REINFORCEMENT LEARNING FRAMEWORK FOR CHESS

Our system consists out of two main components, namely self play and the incorporation of newly seen data into the system.



These two phases are performed iteratively. In section III-A chess is formulated as an RL framework. The system updates are explained with the help of an SL framework in section III-B.

#### A. Reinforcement Learning

In RL, agents are released in an environment where they have to find out by exploration which actions generate the best cumulative rewards. In this case, the board is the environment and white and black are the agents. When one player succeeds at mating the opponent, a positive reward is provided.

To this cause, we have to model chess as a deterministic Markov decision process (MDP)  $\mathcal{M} = (\mathcal{S}, \mathcal{A}, r, \gamma)$  with [14]

- $\mathcal{S}$ : state space containing all possible board positions
- $\mathcal{A}$ : action space, which holds all possible actions.  $\mathcal{A}_s$  is the set of playable moves in board position  $s \in \mathcal{S}$ .
- $r_A : \mathcal{S} \times \mathcal{A}_S \mapsto \mathbb{R}$  is the immediate reward function for player  $A \in \{W, B\}$  (white or black).
- $\gamma \in \mathbb{R} \cap [0, 1]$ : the discount factor

The chess environment is episodic, meaning it has terminal states (checkmate, stalemate, ...). An episode is represented with its history from whites perspective

$$H = \{(s_0, a_0, r_1), (s_1, a_1, r_2), \dots, (s_{T-1}, a_{T-1}, r_T)\}$$

where we assume  $a_i \in \mathcal{A}_{s_i}, \forall i \in 0, \dots, T-1$ , i.e. only legal moves according to the rules of chess can be made in the environment, and  $r_{i+1} = r_W(s_i, a_i)$ .

The history is used for the calculation of the return of a certain state in an episode, which is defined as the exponentially discounted sum of the observed rewards:  $R_t = \sum_{t'=t}^{T-1} \gamma^{t'-t} r_{t'+1}$ . The return is the quantity agents (players) acting on the environment try to optimize. As chess is a game with the zero-sum property,  $r_W(s_t, a_t) = -r_B(s_t, a_t)$ , the optimization goals for white and black are maximizing and minimizing  $R$  respectively (white is generally called the max-player, black the min-player). Consequently, it is the goal of the agents to arrive in positions which have the most optimized expected return, as this will in general optimize their future rewards. This is represented by the value function:

$$V(s_t) = \mathbb{E}[R_t | s] \quad (1)$$

$$= \mathbb{E} \left[ \sum_{t'=t}^{T-1} \gamma^{t'-t} r_{t'+1} | s \right] \quad (2)$$

Furthermore, the chess environment is simplified as follows:

- $\gamma = 1$
- $r_W(s_t, a_t) = \begin{cases} 1 & \text{if } s_{t+1} \text{ is checkmate} \\ 0 & \text{else} \end{cases}$

This allows the rewards to be transposed to the final outcome  $z_H$  of an episode with history  $H$ :

$$z_H = r_T \quad (3)$$

$$= r_W(s_{T-1}, a_{T-1}) \quad (4)$$

As rewards are only accounted at the terminal state, the expression for the value function is reduced to the expected value of the final outcome  $z$

$$V(s) = \mathbb{E}[z | s] \quad (5)$$

The impossibility to store  $V(s)$  for every state speaks for itself, which is why it is represented with a function approximation method (section V). How to update  $V(s)$  with SL is the topic of section III-B.

The optimal policy given  $V^*(s)$  is simply making the greedy choice (define the state successor function  $\Sigma(s_t, a_t) = s_{t+1}$  yielding the next board after making action  $a_t$  on state  $s_t$ )

$$a_t^* = \arg \max_{a \in \mathcal{A}_{s_t}} V^*(\Sigma(s_t, a)) \quad (6)$$

However, due to the tactical nature of chess it is very hard to obtain a good policy by only looking one ply ahead. That is why here and in chess in general some sort of optimized minimax search is performed. This consists of building a tree up to a depth  $d$  and evaluating the leaf nodes. For the minimax policy it is then assumed that both max- and min-player are playing the best move with respect to the leaf node evaluations at depth  $d$ . The path in the tree following the minimax policy is called the principal variation (PV).

Because of the exploration-exploitation dilemma, an over time decaying  $\epsilon$ -greedy policy is applied during self play [14].

#### B. Supervised Learning

In SL, the task is to infer a function (in our case  $V(s)$ , making this a regression problem) that generalizes labeled training data. Supposing  $V_\theta$  is differentiable with respect to its tunable parameters  $\theta$ , an obvious choice for the learning algorithm in an SL context is stochastic gradient descent (SGD) with the mean squared error (MSE) as loss function:  $L_\theta(s, \hat{z}) = (V_\theta(s) - \hat{z})^2$ . The total loss over all  $N$  encountered states in an iteration of self play is then  $L_\theta = \frac{1}{N} \sum_s L_\theta(s, \hat{z})$ . The parameters are updated in the direction of steepest descent with step size  $\alpha$  by calculation of the gradient of the loss function:

$$\theta \leftarrow \theta - \alpha \nabla L_\theta \quad (7)$$

Every episode individually corresponds to the update

$$\theta \leftarrow \theta - \frac{\alpha}{T} \sum_{t=0}^{T-1} (\hat{z}_t - V_\theta(s_t)) \nabla V_\theta(s_t) \quad (8)$$

Both update rules are equivalent, but equation 8 gives more insight regarding the direction we are optimizing to. Equation 7 is more a black box, ideal for software libraries only needing a set of  $(s, \hat{z})$  tuples to perform (mini-batch) SGD.

We chose  $V(s)$  to be a neural network (FNN or CNN) because of its capability to learn complex patterns and features from raw data with SGD through the backpropagation for differentiation. As a main constraint in this research is to be completely unbiased, this is a necessary condition.

All that remains to be found are appropriate training samples  $(s, \hat{z})$  given the histories of simulated episodes during self play, where  $\hat{z}$  is preferably the best possible estimation for the true outcome. In the presented system,  $\hat{z}$  is calculated with game tree search TD-learning methods, presented in section IV.

#### IV. TD-LEARNING

TD algorithms are ideal in an SL framework, because of its natural embedding of the optimization of the MSE cost function in its update rule [16]. An additional advantage specific to chess is that no policy needs to be learned to make it work, which would be hard considering the diverse possibilities of legal moves in chess positions.

The core idea of TD-learning is to update the weights of the value function in the direction of what is supposed to be a better estimate found out one time step later. The observed error is called the temporal difference

$$\delta_t = V_\theta(s_{t+1}) - V_\theta(s_t) \quad (9)$$

where we suppose  $V_\theta(s_T) = z_H$  from now on if  $s_T$  is a terminal state. The episodic update in pure TD-learning (also called TD(0)) is

$$\theta \leftarrow \theta - \frac{\alpha}{T} \sum_{t=0}^{T-1} \nabla V_\theta(s_t) \delta_t \quad (10)$$

##### A. TD( $\lambda$ )

By introducing an additional exponential weight decay parameter  $0 \leq \lambda \leq 1$  credit can be given to more distant states. The resulting update after the simulation of an episode corresponds to

$$\theta \leftarrow \theta - \frac{\alpha}{T} \sum_{t=0}^T \nabla V_\theta(s_t) \sum_{n=t}^{T-1} \lambda^{n-t} \delta_t \quad (11)$$

From equations 8 and 11 we derive a formula to set the target values  $\hat{z}_t$  for a SL framework which can be learned with (mini-batch) SGD on bigger datasets after simulating a whole iteration of episodes:

$$\hat{z}_t = V_\theta(s_t) + \sum_{n=t}^{T-1} \lambda^{n-t} \delta_t \quad (12)$$

##### B. TD-Leaf( $\lambda$ )

With the understanding of the machinery in TD( $\lambda$ ), we have weaponed ourselves to extend the algorithm to include game tree search. Call  $l(s_t, d)$  the node at depth  $d$  in the PV. TD-Leafs goal is then to modify the value function of  $l(s_t, d)$  to  $l(s_{t+1}, d)$ . Hence, the equations can be written out as

$$\delta_t = V_\theta(l(s_{t+1}, d)) - V_\theta(l(s_t, d)) \quad (13)$$

$$\theta \leftarrow \theta - \frac{\alpha}{T} \sum_{t=0}^T \nabla V_\theta(l(s_t, d)) \sum_{n=t}^{T-1} \lambda^{n-t} \delta_t \quad (14)$$

Instead of updating the encountered state as in TD( $\lambda$ ), the leaf nodes are updated, ensuring the collected data to be less correlated. Furthermore, minimax search is included in the algorithm. A downside to this algorithm is the influence of the outcome of an episode to the update of a state which was not part of that episode, as it could be that the updated leaf state was only encountered in the minimax search and not at all in the true simulation. We will call this an update into the direction of 'wrong belief', credit has not been given where it is due.

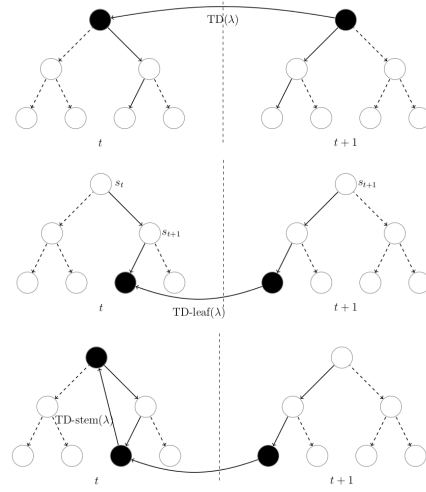


Fig. 1: Comparison between the 3 discussed TD-learning algorithms, the PV are indicated with solid edges. In classical TD-learning the updates are between successive states. TD-leaf updates are between successive leaf nodes and in TD-stem both ideas are combined.

##### C. TD-Stem( $\lambda$ )

The novel proposal is to include depth information gathered from experience into the value function. Although this may have the issue of including more noise into the value function and making it less smooth complicating SGD, it could speed up training by self play as depth information is propagated faster from terminal positions.

The way this would work is to update the static evaluation function of the encountered state to the temporal difference between the successive leaves at the PV (equation 13). The target values for the encountered states are given by equation 12.

A visual comparison between the algorithms is given in figure 1.

#### V. NETWORK ARCHITECTURE

In conventional chess engines, the heuristic evaluation function is linear, but also reinforcement learning chess programs often use a linear value function [7], [9]. In this dissertation, we switched gears and used a neural network model instead, as they have the ability to model a larger range of functions and extract additional features from the data. Additionally, they perform quite good in practice although they do not necessarily converge under TD algorithms [17]. Here, there has been experimented with a CNN architecture using raw image planes as input, which is a first to our best knowledge. The activation function between all hidden units is an ordinary rectifier linear unit (ReLU) and the weights are initialized with the tips given by Bengio et al. [19].

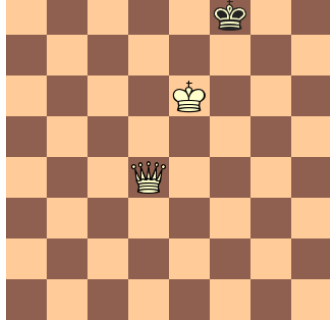


Fig. 2

TABLE I: 8 Bitboards corresponding to the chess position in figure 2. Piece maps indicate the cells where pieces reside. Mobility maps show where pieces can go to.

piece	piece map	mobility map
	0 1 0	0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1 0 1 0 0 0 0 0 1 1 1 0
	0 1 0	0 0 0 1 0 0 0 1 1 0 0 1 0 0 1 0 0 1 0 1 0 1 0 0 0 0 1 1 1 0 0 0 1 1 1 0 1 1 1 1 0 0 1 1 1 1 0 0 0 1 0 1 0 1 0 0 1 0 0 1 0 0 1 0
	0 0 0 0 0 1 0	0 0 0 0 1 0 1 0
	0 0	0 0

Chess boards are separated in two sets of features:

- bitboards: raw image planes (8x8)
- global features: objective observations about the board

An example of how to extract bitboards from a chess position is shown in table I, where the chess board in figure 2 is vectorized. Feature maps are extracted from the binary image planes which are fed into a CNN. The global features come together with a flattened version of the feature maps at the output of the final convolutional layer in a final fully connected network (FCN) in order to obtain the static evaluation.

## VI. EXPERIMENTS

We performed two experiments on basic endgame problems by letting 2 agents with the same value function play episodes

TABLE II: Hyper-parameters modifiable in between stages.

$\lambda$	Parameter for TD-learning methods
$d_r$	Depth at which immediate wins are sought
$d_v$	Depth at which the value network is called
$f_\epsilon$	Decay function for exploration parameter $\epsilon = f(i)$ . i increments every iteration.
$I$	Number of iterations
$i_0$	First iteration number, to initialize $\epsilon$ with the $f_\epsilon$
$K$	The number of states that are used to calculate the $\lambda$ -return from an episode
$M$	The maximal amount of moves made in an episode
$N$	How many games are played during an iteration
$R$	The number of additional random moves played on the board position extracted from the dataset

TABLE III: Hyper-parameters of the stages in the experiments

Exp	Stage	$N$	$I$	$d_v$	$\lambda$	$i_0$	$K$	$d_r$
1	1	5000	20	1	0.5	1	50	0
	2	5000	20	1	0.5	2	50	0
	3	5000	20	1	0.7	2	50	0
	4	500	20	3	0.8	2	50	0
	5	250	30	3	0.8	2	50	0
	6	250	30	3	0.8	2	50	0
	7	250	50	3	0.8	2	50	0
2	1	500	20	1	0.5	0	10	3
	2	250	20	3	0.6	20	20	3
	3	250	20	3	0.7	40	30	3
	4	250	50	3	0.8	40	30	5
	5	250	20	3	0.8	45	30	5

against each other starting from a random position sampled from a generated dataset and compared the final performances of TD-Leaf and TD-Stem by playing 2000 games against an optimal player making perfect moves after training. The optimality is obtained by probing a Gaviota tablebase, containing the exact depth to mate (DTM) and win draw loss (WDL) information [18]. We propose 3 metrics, the win conversion rate (WCR), win efficiency (WE) and loss holding score (LHS) to assess the quality of a model:

$$\begin{aligned}
 \text{WCR} &= \frac{\text{games model won}}{\text{games model should win}} \\
 \text{WE} &= \frac{\text{average DTM of won games}}{\text{average length of won games}} \\
 \text{LHS} &= \frac{\text{average length of lost games}}{\text{average DTM of lost games}}
 \end{aligned}$$

The complete simulation is performed through stages, which are manually controllable. The process has been further optimized with the addition of hyper-parameters (see table II), modifiable through stages. In table III the configurations for the stages conducted in the experiments are laid out.

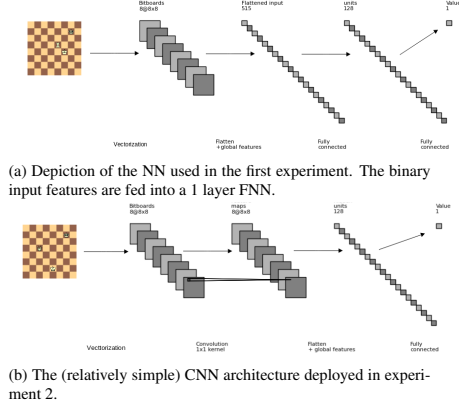


Fig. 3: Network architectures used for the value function during the experiments.

TABLE IV: Performance comparison TD-Leaf( $\lambda$ ) and TD-Stem( $\lambda$ ),  $N$  is the total number of episodes played in the simulation.

	TD-Leaf( $\lambda$ )	TD-Stem( $\lambda$ )
<b>WCR</b>	0.48	<b>0.85</b>
<b>WE</b>	<b>0.87</b>	0.86
<b>LHS</b>	0.80	<b>0.91</b>
<b>N</b>	353 500	<b>304 500</b>

#### A. Experiment 1: king rook king endgame

The first simulations we ran were on the king rook king (K RK) endgame, where the obvious goal for the player holding the rook is to checkmate the opponent as efficiently as possible. As only kings and rooks are set on the board, the input channels are limited to 8 bitboards. The network is trained with a learning rate of  $\alpha = 10^{-4}$ , its architecture is represented in figure 3a. After every iteration during a stage, the exploration parameter  $\epsilon$  is decreased according to the function  $f_{\epsilon}(i) = (i)^{-3/4}$ . Table IV confirms the hypothesis of TD-Stem being a faster learner than TD-Leaf. With the value function learned in TD-Stem, we succeed at winning about 85% of the positions, clearly outperforming TD-Leaf. When the engine is at the losing side, it defends itself better with the novel variant.

#### B. Experiment 2: king queen king endgame

We try to confirm our conjecture of TD-Leaf being at least equivalent to TD-Stem in performance by analyzing an even easier problem, the king queen king (K Q K) endgame. The learning rate of the network depicted in figure 3b is  $\alpha = 10^{-4}$ . The exploration parameter  $\epsilon$  obeys the decay function  $f_{\epsilon}(i) = 1 - 0.02i$ .

A first glance at the learning curve in figure 4 can trick one in believing that TD-Leaf learns quicker than TD-Stem. However, the sudden dive in the curve starting from episode 30000

TABLE V: Performance comparison TD-Leaf( $\lambda$ ) and TD-Stem( $\lambda$ )

	3 stages		5 stages	
	TD-Leaf( $\lambda$ )	TD-Stem( $\lambda$ )	TD-Leaf( $\lambda$ )	TD-Stem( $\lambda$ )
<b>WCR</b>	0.65	<b>0.77</b>	0.90	0.90
<b>WE</b>	<b>0.67</b>	0.64	0.89	0.89
<b>LHS</b>	0.89	0.89	0.95	<b>0.97</b>
<b>N</b>	<b>26000</b>	27000	<b>43500</b>	44500

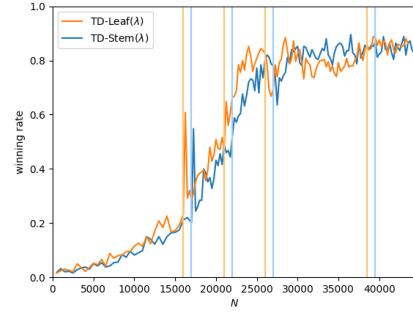


Fig. 4: The learning curve of the second experiment, divided in 5 stages.

corresponding with TD-Leaf hints to a contrary belief. An explanation for this behavior is that the winning rate in the plot represents the agents strength against itself, and not against the optimal agent. Hence, the decrease is a sign of an improvement at the losing side.

We can back this up with the observations noted in table V, where we can see how both models still largely improve in the upcoming stages after the plateau in the learning curve is reached. Furthermore, TD-Stem plays stronger than TD-Leaf at the point in the plateau, confirming our belief of our proposal being the faster learner.

We provide two possible reasons why TD-Stem outperforms TD-Leaf in our experiments:

1. The influence of positive rewards propagates faster in updates, because depth plays a fundamental part to the learned value function at the states and their leaf nodes and so on.
2. The wrong belief effect in TD-Leaf slows down learning

## VII. CONCLUSION

In this study, a different approach from existing chess engines has been tried by not including any human bias to the system. To attain this property, only raw bitboard features and objective information about chess positions are fed into an NN. The value function is learned through self play in combination with a novel RL algorithm we called TD-Stem( $\lambda$ ). We have compared this proposal with TD-Leaf( $\lambda$ ) and the outcome of the experiments indicate that our variant learns faster.

More research can and needs to be carried out to obtain valuable knowledge about strategic game play in chess. Just as in backgammon, there might valuable positional knowledge in the game which has not been discovered yet. To explore these possibilities with RL however, it is important to only make use of objective information (like tablebases for instance).

Possibilities to continue research are for example by learning on bigger endgame examples, deeper CNNs, more bitboards extracted from chess positions and learning policy networks.

#### REFERENCES

- [1] Claude E. Shannon, *Programming a computer for playing chess*, Philosophical Magazine, 41(314) 1950
- [2] Alan Turing, *Chess*, Digital Computers applied to Games, 1953
- [3] Murray Campbell, Joseph Hoane Jr, Feng-hsiung Hsu *Deep Blue*, Artificial Intelligence, Vol. 134, p57-83, 2002
- [4] *CCRL 40/40 Rating List* [http://www.computerchess.org.uk/ccrl/4040/cgi/compare/\\_engines.cgi?print=Rating+list+%28text%29&class=all+engines&only\\_best\\_in\\_class=1](http://www.computerchess.org.uk/ccrl/4040/cgi/compare/_engines.cgi?print=Rating+list+%28text%29&class=all+engines&only_best_in_class=1) 2017
- [5] Arthur L. Samuel *Some Studies in Machine Learning Using the Game of Checkers*, IBM Journal of Research and Development, Vol. 3, p210-229, 1959
- [6] Gerald Tesauro *Temporal Difference Learning and TD-Gammon*, Communications of the ACM, Vol. 38, p58-68, 1995
- [7] Jonathan Baxter, Andrew Tridgell and Lex Weaver *TDLeaf( $\lambda$ ): Combining Temporal Difference Learning with Game-Tree Search*, CoRR, 1999
- [8] Matthew Lai *Giraffe: Using Deep Reinforcement Learning to Play Chess*, diploma thesis Imperial College London, 2015
- [9] Joel Veness, David Silver, Alan Blair and William Uther *Bootstrapping from Game Tree Search*, Advances in Neural Information Processing System Vol. 22 p.1937-1945, 2009
- [10] Omid E. David, Nathan S. Netanyahu, Lior Wolf *DeepChess: End-to-End Deep Neural Network for Automatic Learning in Chess*, Artificial Neural Networks and Machine Learning ICANN 2016
- [11] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra and Martin Riedmiller *Playing Atari with Deep Reinforcement Learning*, NIPS Deep Learning Workshop 2013
- [12] Ziyu Wang, Nando de Freitas and Marc Lanctot *Dueling Network Architectures for Deep Reinforcement Learning*, CoRR 2015
- [13] David Silver, Aja Huang, Chris J. Madison et al. *Mastering the game of Go with deep neural networks and tree search*, Nature Vol. 529 p484-489, 2016
- [14] Richard S. Sutton and Andrew G. Barton, *Introduction to Reinforcement Learning*, 1998
- [15] George T. Heineman, Gary Pollice and Stanley Selkow (2008), *Algorithms in a Nutshell*, p217223, 1998
- [16] Csaba Szepesvári *Algorithms for Reinforcement Learning*, 2010
- [17] J.N. Tsitsikilis, B.V. Roy *An Analysis of Temporal Difference Learning with Function Approximation* IEEE Transactions on Automatic Control, Vol.42(5), p674690 1996
- [18] Galen Huntington, Guy Haworth, *Depth to Mate and the 50-Move Rule*. ICGA Journal, Vol. 38, No. 2
- [19] Xavier Glorot, Yoshua Bengio *Understanding the difficulty of training deep feedforward neural networks* Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics, PMLR 9: p249-256, 2010

# Contents

<b>Preface</b>	<b>ii</b>
<b>Abstract</b>	<b>iv</b>
<b>Extended abstract</b>	<b>v</b>
<b>List of Figures</b>	<b>xiv</b>
<b>List of Tables</b>	<b>xvi</b>
<b>List of Algorithms</b>	<b>xvii</b>
<b>1 Introduction</b>	<b>1</b>
<b>2 Conventional Chess Engines</b>	<b>4</b>
2.1 Search . . . . .	4
2.1.1 Minimax . . . . .	5
2.1.2 $\alpha\beta$ -pruning . . . . .	6
2.1.3 Principal Variation Search . . . . .	7
2.1.4 Quiescence Search . . . . .	8
2.1.5 Other Search Improvements . . . . .	10
2.2 Evaluation . . . . .	11
2.3 Databases . . . . .	12
2.3.1 Opening Books . . . . .	12
2.3.2 Endgame Tablebases . . . . .	12
2.4 Conclusion . . . . .	13
<b>3 Deep Learning Background</b>	<b>16</b>
3.1 Machine Learning . . . . .	16
3.2 Neural Network . . . . .	17
3.2.1 Feedforward Neural Network . . . . .	17
3.2.2 Convolutional Neural Network . . . . .	22
3.2.3 Deep learning . . . . .	25
3.3 (Deep) Reinforcement Learning . . . . .	25
3.3.1 Markov Decision Process . . . . .	26

3.3.2	Value Prediction . . . . .	28
3.3.3	Control . . . . .	31
3.3.4	Value and Policy Iteration . . . . .	32
<b>4</b>	<b>Deep Reinforcement Learning in Chess (and other games)</b>	<b>34</b>
4.1	Feature Space . . . . .	34
4.2	Static Evaluation Function . . . . .	37
4.3	TD algorithms . . . . .	37
4.3.1	TD( $\lambda$ ) . . . . .	39
4.3.2	TD-directed( $\lambda$ ) . . . . .	41
4.3.3	TD-root( $\lambda$ ) . . . . .	41
4.3.4	TD-leaf( $\lambda$ ) . . . . .	42
4.3.5	TD-stem( $\lambda$ ) . . . . .	43
4.4	Bootstrapping Tree Search . . . . .	44
4.5	Monte Carlo Tree Search . . . . .	45
4.5.1	AlphaGo . . . . .	46
4.6	Policy Gradient Reinforcement Learning . . . . .	47
4.7	Monte Carlo Value Initialization . . . . .	48
<b>5</b>	<b>Experiments</b>	<b>51</b>
5.1	Goal . . . . .	51
5.2	Metrics . . . . .	51
5.2.1	Learning Curves . . . . .	52
5.2.2	Optimal Opponent Evaluation . . . . .	52
5.2.3	Value Curves . . . . .	54
5.2.4	Speed . . . . .	54
5.3	Experimental Setup . . . . .	54
5.3.1	Search . . . . .	55
5.3.2	Evaluation . . . . .	55
5.3.3	Self Play . . . . .	56
5.3.4	Implementation . . . . .	59
<b>6</b>	<b>Results</b>	<b>63</b>
6.1	Experiment 1: king rook king with uniform sampling . . . . .	63
6.2	Experiment 2: king queen king with sampling from dataset . . . . .	68
6.3	Experiment 3: 3 pieces endgames with sampling from dataset . . . . .	70
6.4	Conclusion . . . . .	73
<b>7</b>	<b>Conclusion</b>	<b>74</b>
	<b>References</b>	<b>76</b>
<b>A</b>	<b>Chess</b>	<b>81</b>
A.1	Rules . . . . .	81

---

A.1.1	Chess Pieces . . . . .	81
A.1.2	Special Positions . . . . .	81
A.1.3	Special Moves . . . . .	82
A.1.4	Termination . . . . .	82
A.2	Evaluation of Strength . . . . .	84
A.3	Chess Jargon . . . . .	87



# List of Figures

2.1	Search tree . . . . .	5
2.2	Importance quiescence search . . . . .	10
2.3	Mate in 549 . . . . .	13
2.4	Toy example . . . . .	15
3.1	General feedforward neural network (FNN) architecture . . . . .	18
3.2	Hidden units . . . . .	19
3.3	Connections FNN and convolutional neural network (CNN) . . . . .	24
3.4	Convolutional layer flow diagram . . . . .	24
3.5	reinforcement learning (RL) paradigm . . . . .	25
4.1	Chess Position . . . . .	36
4.2	Network architecture value function . . . . .	37
4.3	Comparison between TD-root( $\lambda$ ) and TD( $\lambda$ ) . . . . .	43
4.4	TD-leaf( $\lambda$ ) . . . . .	44
4.5	TD-stem( $\lambda$ ) . . . . .	45
4.6	Monte Carlo tree search (MCTS) . . . . .	46
4.7	AlphaGo . . . . .	47
4.8	Two examples leading to bad <i>MC</i> evaluations. . . . .	50
5.1	learning curve . . . . .	52
5.2	End games . . . . .	57
5.3	High level software architecture of the simulation system. . . . .	62
6.1	neural network (NN) used in the first experiment . . . . .	64
6.2	Learning curve experiment 1 . . . . .	65
6.3	Performance comparison between TD-Stem( $\lambda$ ) and TD-Leaf( $\lambda$ ) experiment 1 . . . . .	66
6.4	value curve experiment 1 . . . . .	67
6.5	CNN architecture deployed in experiment 2 . . . . .	69
6.6	Learning curve of the second experiment . . . . .	70
6.7	Performance comparison between TD-Stem( $\lambda$ ) and TD-Leaf( $\lambda$ ) . . . . .	71
6.8	Value curves experiment 2 . . . . .	72
6.9	CNN architecture deployed in experiment 2 . . . . .	72

---

A.1	Chessboard layout . . . . .	82
A.2	Piece mobility . . . . .	83
A.3	Chess starting position . . . . .	84
A.4	Special chess positions . . . . .	85
A.5	Castling . . . . .	86
A.6	En passant capture . . . . .	86
A.7	Promotion . . . . .	87

# List of Tables

2.1	Comparison $\alpha\beta$ -pruning and regular minimax . . . . .	8
2.2	Strongest chess engines . . . . .	14
3.1	Comparison between activation functions . . . . .	20
4.1	Bitboards . . . . .	36
5.1	Hyper-parameters modifiable in between stages. . . . .	60
6.1	Hyper-parameters of the stages in Experiment 1 . . . . .	65
6.2	Performance comparison TD-Leaf( $\lambda$ ) and TD-Stem( $\lambda$ ) . . . . .	65
6.3	Hyper-parameters of the stages in Experiment 2 . . . . .	68
6.4	Performance comparison TD-Leaf( $\lambda$ ) and TD-Stem( $\lambda$ ) . . . . .	69
6.5	Hyper-parameters of the stages in Experiment 2 . . . . .	71
6.6	Performance of the trained model in experiment. . . . .	73
A.1	Glossary with chess terms . . . . .	88

# List of Algorithms

1	Minimax . . . . .	6
2	$\alpha\beta$ Minimax . . . . .	7
3	PVS . . . . .	9
4	Minibatch stochastic gradient descent . . . . .	21
5	Episodical EMVC . . . . .	28
6	TD(0) . . . . .	29
7	Tabular TD( $\lambda$ ) . . . . .	30
8	supervised TD algorithm . . . . .	42
9	Monte Carlo Value Initialization . . . . .	49
10	$\alpha\beta$ -Result . . . . .	56

# Acronyms

<b>AI</b>	artificial intelligence.
<b>ANN</b>	artificial neural network.
<b>CNN</b>	convolutional neural network.
<b>DDQN</b>	double deep Q-network.
<b>DTM</b>	depth to mate.
<b>EVMC</b>	every visit Monte Carlo.
<b>FCN</b>	fully connected network.
<b>FNN</b>	feedforward neural network.
<b>MC</b>	Monte Carlo.
<b>MCTS</b>	Monte Carlo tree search.
<b>MCVI</b>	Monte Carlo value initialization.
<b>MDP</b>	Markov decision process.
<b>ML</b>	machine learning.
<b>MSE</b>	mean squared error.
<b>NN</b>	neural network.
<b>PV</b>	principal variation.
<b>PVS</b>	principal variation search.
<b>QS</b>	quiescence search.
<b>ReLU</b>	rectifier linear unit.
<b>RL</b>	reinforcement learning.

**SGD** stochastic gradient descent.

**TD** temporal difference.

# Chapter 1

## Introduction

Ever since computer science started to emerge as a proper field, researchers have been attracted to the problem of letting computers play the game of chess. This is mainly out of theoretical interest for artificial intelligence (AI), as it is a human task. Chess is a complex game (it has been estimated that around  $10^{120}$  unique board positions exist (Shannon, 1950)), which makes brute force algorithms completely unscalable. In the early 50's, notable scientists such as Turing and Shannon created a theoretical chess program on paper (Shannon, 1950; Turing, 1953). Remarkable is they both already included the notions of an evaluation function, minimax and quiescent search. These concepts are still used today in state of the art chess programs.

In the second half of the 20th century, computer chess evolved further and further. Continuous minor adjustments were made to the heuristics of the evaluation function, computation became more and more parallelized and search methods were improved with more efficient pruning. This led to the victory of *Deep Blue* against world champion at that time Gary Kasparov in 1997 (Murray Campbell, 2002), an event that received a lot of attention from the media, mainly due to the symbolical significance of machine catching up with human intelligence (Krauthammer, 1997).

Since that event, computer chess was studied less and less and the *Drosophila*<sup>1</sup> of AI research became go, a game that is, despite its simplicity of rules, extremely complex. Search techniques classically used in chess engines are less usable for go engines mainly due to the size of the board and large amount of possible moves every ply. Recently in 2016, a *Google* research team succeeded at beating a 9-dan<sup>2</sup> professional player (David Silver, 2016). The researchers achieved this by using a combination of tree search algorithms and deep RL. This success lead to a nomination by *Science* magazine for *Breakthrough of the Year* (scb, 2016).

In this master thesis, we look into how reinforcement learning and other machine learning tech-

---

<sup>1</sup>*Drosophila* species are extensively used as model organisms in genetics for testing new techniques. It is used as a metaphor in this context.

<sup>2</sup>Highest rating achievable in *go*.

niques could be used on chess as well. Through self play and deep neural networks for the evaluation function we investigate what machines can learn about the chess game without any prior knowledge except the rules. This lack of inserting human knowledge into the system makes the learning agent as unbiased as possible, which is necessary to truly reveal the power of RL. In a similar manner by a huge amount of stochastic self play combined with a TD learning algorithm <sup>3</sup>, *TD-gammon* was able to discover new strategies humans never thought about before (Tesauro, 1995).

The move decision of current chess computers is based on looking ahead as far as possible. They build a game tree and evaluate positions at the bottom of the tree with a static evaluation function. The move choice is then the one that would lead to the best evaluation at the bottom of the tree under the assumption that both players try to make the best moves. the search is heavily optimized as an attempt to reduce the influence of the exponentially increasing complexity when looking more moves ahead. The height of the search tree is the most crucial factor for the performance of an engine.

The evaluation function is usually a heuristic based where game technical knowledge is used, which has been gathered during the history of the game by expert chess players. These rules are not objective, as they were determined by humans. On top of that, chess computers have difficulties at playing intuitively.. They still show signs of weakness in the endgame, and it takes a lot of effort to design or improve them. Furthermore, porting these technique to other games requires redesigning most of the heuristics, and the resulting AIs do not always achieve a similar high level of play for more strategical games.

All current machine learning approaches to learn the static evaluation function use the helping hand of human knowledge about the chess game in their learning system. This makes it harder to actually improve on the classical state of the art approaches with heuristics. The purpose of this thesis is to research how a machine could teach itself to play chess with the rules as only knowledge. Choosing for a completely unbiased system could make the chess engine learn strategies humans never thought about before, just like what happened with *TD-Gammon*.

There have been recent and less recent efforts to incorporate machine learning into chess engines. Recently in 2015, an end-to-end machine learning chess engine incorporating self play and reinforcement learning has reached the FIDE International Master level, while Grandmaster level can be achieved with the use of millions of database games (Lai, 2015; Omid E. David, 2016).

Deep RL can help us out with our quest at perfecting chess play and strategy. In RL, agents are released in an environment where they have to find out by exploration which actions generate the best cumulative rewards. To improve their play, the agents make use of the gained experience. In this case, the board is the environment and white and black are the agents. By only incorporating basic and objective game elements into the framework the agents can discover by

---

<sup>3</sup>A RL algorithm, further discussed in chapter 3



themselves what is good and what not, without additional biased knowledge. By using self play in a Deep RL framework a value function can be constructed with only information observed by the agents themselves. As this would all be learned by the agents experience, the final static evaluation could introduce additional depth information, making deeper searches less necessary.

Deep RL has already proven its worth in games, *Atari* and *Go* being most noteworthy. The fact that sensory image data is being processed during playing *Atari* games made it the first game where CNNs were proven successful (Mnih et al., 2013). These video games are an excellent toy to mess around and experiment with new deep RL algorithms. AI game play on *Atari* games has evolved further and new architectures came along the way (Wang et al., 2015).

In this master's thesis we will set the first step in the quest to let machines master the complete chess game through self play only and gaining experience without biased supervision. More specifically, we introduce new algorithms that can be used along the way. Due to time constraints and complexity of the overall problem, it has not been possible to implement and examine all covered techniques, but we compare a new temporal difference (TD) learning algorithm, TD-Stem( $\lambda$ ), with a state of the art mechanism to improve chess play with RL. We also limit ourselves to easy chess endgame problems, as they give the opportunity for objective quality assessments. The learned models could be used to learn more complex chess positions as well from the bottom up.

The main questions we try to answer in this dissertation are:

- How would one program a chess engine without bias with deep RL?
- What is the best TD learning algorithm for this cause?

This book is organized as follows. We start off in chapter 3 by exploring how current chess computers achieve their strength and further discuss the remaining weaknesses. In chapter 3 an overview is given of deep learning. The knowledge gained in these two chapters are combined in chapter 4, where existing and new ideas are combined to propose possible architectures and techniques for programming a chess computer with deep RL. The setup of the carried out experiments are further discussed in chapter 5. Subsequently, the results of these experiments are presented and discussed in chapter 6. This thesis is finalized in a conclusion in chapter 7. For the readers whom are less familiar with the game of chess, we provide a brief introduction in appendix A.

## Chapter 2

# Conventional Chess Engines

State of the art chess computers search into a tree which is branched by the possible moves the players can make. The chosen move is then the move where the most plausible leaf in the tree yields the highest evaluation. Most engines choose to set up this evaluation heuristically, using expert knowledge. Tree search is computationally speaking a very complex operation, which is why branches are pruned as much as possible and heuristics are kept simple to provide fast evaluations. Generally speaking, due to the very time consuming calculations, code is optimized intensively. These optimizations happen in software as well in hardware. To gain computer power, the engines take up most of the CPUs when possible to branch through the search tree, as these operations are highly parallelizable.

In this chapter, we will look at the principal techniques used in current chess engines. These methods are typically optimized as much as possible to reduce computational overhead. This chapter is divided into four sections, each one describing an important aspect about the determination of the value function of the current board position. This value function is the mapping of a chess board to a score to indicate which side has the best chances. These methods can give an insight in how to improve Deep RL algorithms with respect to chess.

First, we will look at how the size of the search tree of possible future moves is reduced in section 2.1. Next we consider how to evaluate a board position in section 2.2, after which we conclude the chapter by looking into how we can use data intensive knowledge to improve play in section 2.3.

## 2.1 Search

Due to the tactical nature of chess (i.e. every slightest modification in board configuration can drastically change the evaluation), it seems to be nearly impossible to find an algorithm generating the best move in a blink of an eye. This is why some lookahead is needed to evaluate positions reliably. The search to the best move is done by branching through a tree structure where edges represent moves, and the nodes connected by the edges the positions before and after the move. An example of a search tree is depicted in 2.1.

One problem is that the search tree grows exponentially for every move made. Suppose we want

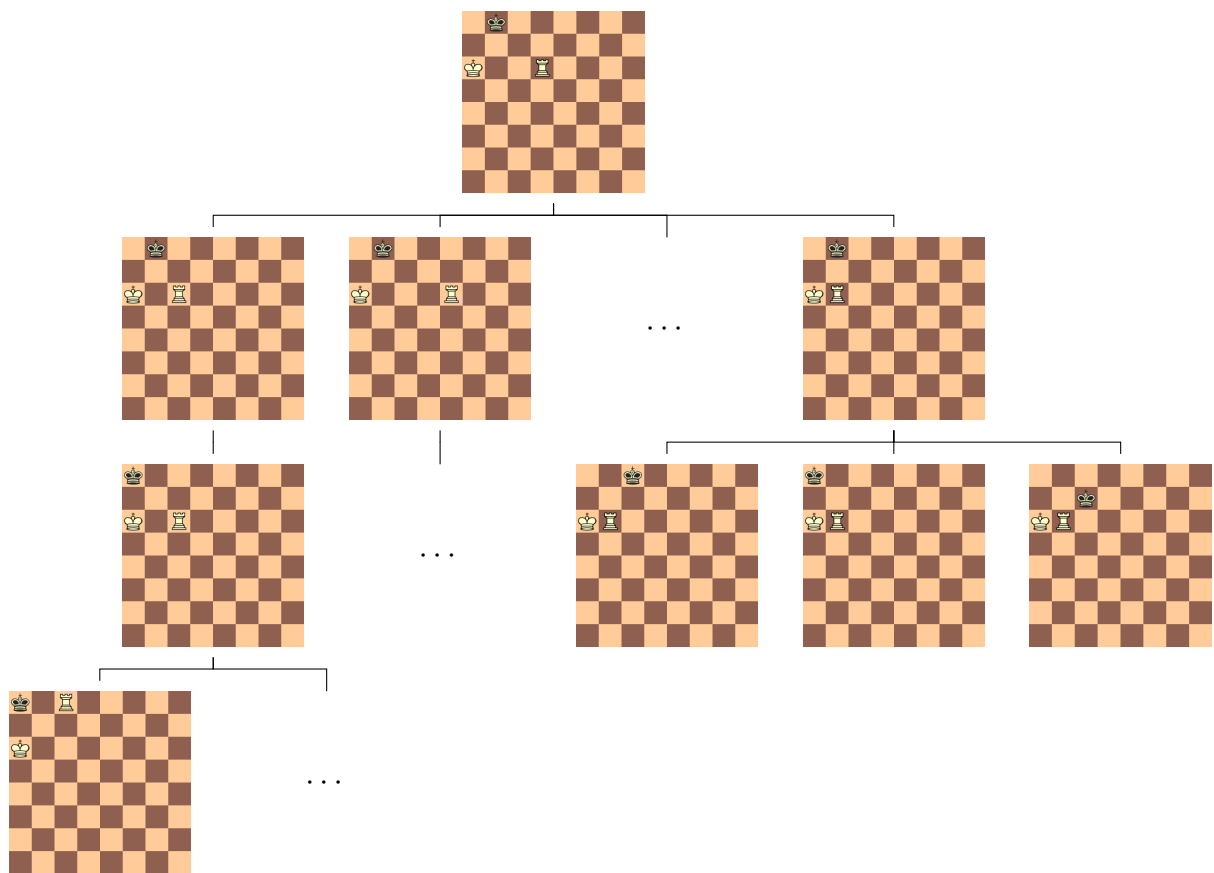


Figure 2.1: An illustration of a search tree up to  $d = 3$ .

to look at a depth  $d$  (the lateral length of the tree, in this context the maximal number of moves thought ahead) and we can make on average the branching factor  $b$  moves every ply, the size of the search tree is  $b^d$ . As chess' branching factor is around 35 (Levinovitz, 2014), the depth is limited very rapidly. At the same time, the deeper an engine can look ahead, the stronger it plays. This is why researchers try to reduce as many branches in the search tree as possible.

### 2.1.1 Minimax

Minimax is the algorithm describing how to evaluate the root position given the static evaluations of the leaf positions in games. Basically, when we look deeper into the tree, we have to alternately choose between the score of whites and blacks best move. In game theory terminology white and black are called the max-player and min-player, respectively. As the name suggests, the max-player tries to maximize the score while the min-player tries to minimize it. Due to the simplicity of zero-sum games like chess (they are symmetrical, both players try to achieve exactly the same goal, namely checkmating the opponent), the evaluation of a position is equivalent to the negation of the evaluation from the opponent's viewpoint:

$$V^{(w)}(s) \equiv -V^{(b)}(s) \quad (2.1)$$

with  $V^{(w)}(s)$  and  $V^{(b)}(s)$  the value functions from white and blacks point of view respectively. Value functions indicate the expected outcome of position (an estimated or exact score appropriate to the position). If one side is winning, the other one is losing. This simplification of minimax with the zero-sum property (also called negamax in literature) is listed as pseudo code in algorithm 1.

---

**Algorithm 1** Minimax
 

---

**Input:** board, depth

**Output:** max

```

// max-player should call Minimax(board,depth)
// first call min-player should be  $-Minimax$ (board,depth)
if depth  $\leftarrow$  0 then
    return Evaluation(board) // static evaluation of a leaf node
end if
max  $\leftarrow$   $-\infty$ 
for move in LegalMoves(board) do
    newboard  $\leftarrow$  doMove(board, move)
    score  $\leftarrow$   $-Minimax$ (newboard, depth - 1)
    if score > max then
        max  $\leftarrow$  score
    end if
end for
return max

```

---

### 2.1.2 $\alpha\beta$ -pruning

Minimax requires traversing the entire search tree, which is not at all feasible as explained before due to the exponential growth of the search tree. Luckily, an improvement can be made to minimax which allows to safely prune certain subtrees. This improvement is called  $\alpha\beta$ -pruning, and was already proposed in 1955 (John McCarthy, 1955). The number of nodes to be evaluated is reduced by not traversing nodes further in a branch-and-bound manner when a refutation has been found. In the (recursive) algorithm, two values are stored:  $\alpha$  and  $\beta$ .  $\alpha$  represents the minimal score for the max-player in a variation, while  $\beta$  represents the maximal score for the min-player. In other words, they are the provisional worst case scenario evaluations for both parties. Based on  $\alpha$  and  $\beta$ , variations can be cut off if proven to involve inferior game play by one of the players. A formal algorithm for games with the zero-sum property is provided in algorithm 2. Completely worked out examples enhance the understanding significantly, therefore we encourage the reader to look at the cited website (Kamko, 2016).

In the worst case, we always evaluate bad leafs and end up with the same complexity of minimax,  $O(b^d)$ . In the best case scenario, edges corresponding with good moves are traversed first,

resulting in hard to beat  $\alpha$  and  $\beta$  values. By always choosing to branch through the best moves in the search tree we achieve the square root of the original time complexity of minimax  $O(b^{\frac{d}{2}})$  (Russel J. Stuart, 2003). From this reasoning, it follows that the move ordering of branching is very important, as we could potentially gain twice the depth in our search in the same timespan (see table 2.1 for a comparison). Heuristics as well as machine learning techniques like neural networks can be used to determine the move order of the search (Robert Hyatt, 2005; van den Herik, 2002).

---

**Algorithm 2**  $\alpha\beta$ Minimax

---

**Input:** board, depth,  $\alpha$ ,  $\beta$ 
**Output:**  $\alpha$ 

```

// max-player should call Minimax(board,depthLeft, $-\infty,\infty$ )
// min-player should call  $-Minimax$ (board,depthLeft, $-\infty,\infty$ )
if depthLeft  $\leftarrow$  0 then
    return Evaluation(board) // static evaluation of a leaf node
end if
for move in LegalMoves(board) do
    newboard  $\leftarrow$  doMove(board, move)
    score  $\leftarrow -\alpha\beta Minimax$ (newboard, depthLeft - 1,  $-\beta, -\alpha$ ) // Exchanging because
    of zero-sum property
    if score  $\geq \beta$  then
        return  $\beta$  //  $\beta$ -cutoff
    end if
    if score  $> \alpha$  then
         $\alpha \leftarrow$  score //  $\alpha$  is the max from Minimax (Algorithm 1)
    end if
end for
return  $\alpha$ 

```

---

### 2.1.3 Principal Variation Search

principal variation searches (PVSs) (which in literature is often put equal to the almost identical *negascout* algorithm) is a further improvement to the  $\alpha\beta$ -pruning minimax algorithm which in general leads to a save in computation time when the move ordering is done right.

The idea is the following: we base our calculations on the principal variation, which is the path in the tree corresponding to the intuitively best move (this could be calculated for example with shallow searches of limited depth). The first node typically traversed is part of the principal variation, after which we continue with regular  $\alpha\beta$ -pruning. For all the other moves, we set up a small window based (i.e. we suppose  $\beta$  to be close to  $\alpha$ ) on the  $\alpha$ -value found in the principal variation. The immediate consequence is a growing number of  $\beta$ -cutoffs. If we were to find a

Table 2.1: The exact number of explored nodes in the best case scenario is  $b^{\lceil \frac{d}{2} \rceil} + b^{\lfloor \frac{d}{2} \rfloor} - 1$  (Daniel Edwards, 1961), while it is  $b^d$  in regular minimax, which is the worst case for  $\alpha\beta$ -pruning.

depth	worst case	best case
0	$1 \cdot 10^0$	$1 \cdot 10^0$
1	$3.5 \cdot 10^1$	$3.5 \cdot 10^1$
2	$1.225 \cdot 10^3$	$6.9 \cdot 10^1$
3	$4.287 \cdot 10^4$	$1.259 \cdot 10^3$
4	$1.500 \cdot 10^6$	$2.449 \cdot 10^3$
5	$5.252 \cdot 10^7$	$4.409 \cdot 10^4$
6	$1.838 \cdot 10^9$	$8.574 \cdot 10^4$
7	$6.433 \cdot 10^{10}$	$1.543 \cdot 10^6$
8	$2.251 \cdot 10^{12}$	$3.001 \cdot 10^6$
9	$7.881 \cdot 10^{13}$	$5.402 \cdot 10^7$
10	$2.758 \cdot 10^{15}$	$1.050 \cdot 10^8$

variation leading to a score in the window, we know our initial assumption was wrong, and we have to search through the same node once more with general  $\alpha\beta$ -pruning. This is why the move ordering is crucial for this to work. See algorithm 3 for pseudo code.

PVS and negascout yield the same result as  $\alpha\beta$ -pruning (Reinefeld, 1989). The algorithm is heavily based on the original *Scout* algorithm, which was the first algorithm to outperform  $\alpha\beta$ -pruning (Pearl, 1980).

Another minimax search algorithm has been implemented, proven to outperform PVS and negascout in practice, namely *MTD(f)* (Plaat et al., 1996). The window calls from PVS only return a bound on the minimax value. In contrast, *MTD(f)* calls a full  $\alpha\beta$ -search a number of times, converging towards the exact value. Overhead of searching through the same nodes is covered by transition tables in memory.

### 2.1.4 Quiescence Search

A remaining problem after the already presented search algorithms is the fixed depth. This can cause the horizon effect, the evaluation at the leaf node can be suboptimal due to the possible presence of tactical moves completely changing the static evaluation. An example of how problematic this can be is shown in figure 2.2.

Typical tactical moves that are considered are piece captures and checks. The search goes on until the node seems 'quiet' enough, or by finding a new lower bound to the minimax score with a static evaluation (Beal, 1990). This enhancement is based on the null move observation, which

---

**Algorithm 3** PVS

---

**Input:** board, depth,  $\alpha$ ,  $\beta$ **Output:** max*// max-player should call  $PVS(\text{board}, \text{depth}, -\infty, \infty)$* *// min-player should call  $-PVS(\text{board}, \text{depth}, -\infty, \infty)$* **if** depth  $\leftarrow$  0 **then**    **return** *Evaluation*(board)**end if***// First, calculate score of principal variation with full  $\alpha\beta$ -search*pv  $\leftarrow$  *PV*(board)newboard  $\leftarrow$  *doMove*(board, pv) $\alpha \leftarrow -PVS(\text{newboard}, \text{depth}-1, -\beta, -\alpha)$ *// Look at other moves in a reduced window  $[\alpha, \alpha + 1]$  (Fishburn, 1984)***for** move **in** *LegalMoves*(board) **and** move  $\neq$  pv **do**    newboard  $\leftarrow$  *doMove*(board, move)    score  $\leftarrow -PVS(\text{newboard}, \text{depth} - 1, -\alpha - 1, -\alpha)$     **if**  $\alpha < \text{score} < \beta$  **then**        score  $\leftarrow -PVS(\text{newboard}, \text{depth} - 1, \beta, -\alpha)$  *// Wrong guess of pv, full research*    **end if**    **if** score  $\geq \beta$  **then**        **return**  $\beta$  *//  $\beta$ -cutoff*    **end if**    **if** score  $> \alpha$  **then**         $\alpha \leftarrow$  score    **end if****end for****return**  $\alpha$ 

---

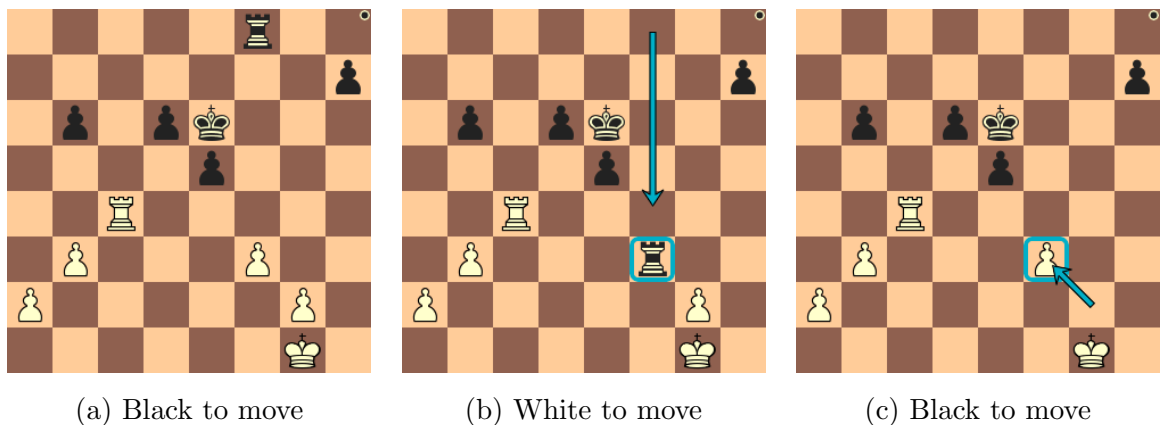


Figure 2.2: Suppose the remaining depth after figure 2.2b is 0, then a static evaluation of the position is required. Without any quiescence search (QS), a typical engine would first of all count all the pieces on the board and note that black is a pawn up, hence yield an evaluation in favor of black. This is utterly wrong, as white is winning in this position, due to the tactical change at the next move `gxf3`. Now White is a Rook up in exchange of a pawn. The figure 2.2c shows the resulting position after the exchange. There are no remaining tactical threats in this position, hence this is a quiet position. The engine should have computed this as well to provide a reliable minimax evaluation.

expresses that it is (almost <sup>1</sup>) always better for the side to move to play than to do nothing.

### 2.1.5 Other Search Improvements

In this subsection, we briefly discuss other possible (some of them obligatory) enhancements to regular  $\alpha\beta$ -search.

While traversing through the search tree, stakes are high chess engines encounter the same states again and again (but achieved through different paths). This is why a transposition table is stored in memory, which is essentially a large hash table storing information about all encountered positions (Kishimoto, 2002). Typical characteristics that are stored in an entry are

- best move
- highest calculated depth
- score

Another almost obligatory enhancement on the basic methodology is iterative deepening. Because of the exponential complexity and uncertainty about the duration of searching (e.g. in QS), time management is required. Some time is allocated for the search of a best move, and systematically the calculations go deeper and deeper. In case of time running out, the engine can still fall back to the best move according to current calculations, this as opposed to

---

<sup>1</sup>In rare cases, no good moves are available, and a player is forced to weaken its position. These positions are called zugzwang positions



to depth first searches. Usually, the depth first algorithms like  $\alpha\beta$ -pruning (as we have seen in previous sections) are fit into a best first framework (Korf, 1985; Reinefeld and Marsland, 1994).

In section 2.1.3 we saw that a more narrow window was used in PVS as a way to achieve more cutoffs in  $\alpha\beta$ -search. Aspiration windows use the same idea, they make a guess about the size of the window. If the true score appears to be outside this window, a costly re-search has to be made. To mitigate this downside, some programs change the window size according to fails. For example, if the windows fail by estimating  $\beta$  too low, an engine will typically increase the upper bound.

## 2.2 Evaluation

How do we give a static evaluation to a leaf node in the search tree? The exact value is unknown unless we have arrived (close to) a terminal state, so we have to make an approximation. In conventional chess engines heuristics are applied to perform these approximations. This comes at the cost of being biased by human expert knowledge, which is difficult to prove to be entirely correct. Examples can be found where the computer does not seem to find the best move or interpret the position correctly.

We are not going to discuss this in great detail, but generally a big amount of features is extracted from the position, and every feature gets a weight. Engines use these weights to describe the dynamics between these features and their relative importance. Most evaluation functions in existing chess engines use a linear value function

$$V(s) = \sum_{f \in \mathcal{F}} w_i f \quad (2.2)$$

where  $s$  denotes the state of the board (the chess position). The weights represent the importance of every feature to the global evaluation, e.g. the value of the pieces on the board will have a larger weight than structural characteristics. Nonlinear evaluation functions can be used as well, but are due to speed less common at the moment of writing. Typical features considered are among other more complex ones:

- material
- pawn structure
- piece mobility
- connectivity
- king safety

The weights of features also depend on the game phase. Most evaluation functions make a distinction between the opening, middlegame and endgame. These are all hand crafted characteristics about the chess game humans discovered by themselves, and each player from beginner to expert tries to learn the dynamics between the importance of these features through game play.

## Tuning

Tuning is the subject about how to automatically adjust the weight to achieve the highest possible playing strength. This can be done with supervised learning (section 3.1), reinforcement learning (section 3.3), genetic algorithms as used in *Falcon* (David-Tabibi et al., 2009), but also with other less mathematically sound mechanisms. *Stockfish* for example assures that the weights of the most important features reach their optimal values rapidly based on game play between engines differentiated through the variables, which accompanies a gain of strength. One issue is that after the optimization of the important variables, the less significant variables take a random walk and playing strength may decrease (Kiiski, 2011).

To perform reliable automated tuning, a change in playing strength should somehow be measured. This is faced with some difficulties, as chess is a game containing a lot of diversity. How does one generate an adequate set of test positions? How are we certain all aspects of the game have been covered? The solution is to play as many randomized (preferably realistic in practice) positions between engine opponents. The standard way to do this is to play many games with a small time control, where games can end prematurely. Strength of an engine (or chess player in general) is calculated with the ELO rating system (section A.2).

## 2.3 Databases

### 2.3.1 Opening Books

Probably the most complex phase of a chess game is the opening. Strategically and tactically it is a very challenging part of the game, and grandmasters often lose games due to slight inaccuracies in opening play. This is why grandmasters prepare their games by revising their opening play depending on the typical opening choices of their opponents. If the chess player is well prepared, he may automate the first moves and save a considerable amount of time, which may come out beneficially at time control.

An identical argument can be made in favor for an engine storing an opening book, which is basically a tree of chess moves starting from the initial position. All lines stored in an opening book are expert knowledge, chess theory evolved throughout history.

Chess theory may contain bad lines, this is why book learning is implemented in *Crafty* (Hyatt, 1999; Kloetzer, 2011). When many games through the same leaf node of the opening tree are lost, this node is removed. Hence, this is a trial and error mechanism. This bad book move signaling can remove bad lines.

### 2.3.2 Endgame Tablebases

Since August 2012, chess has been solved completely up to seven pieces (kings included) by a supercomputer at *Moscow State University*. The idea of a tablebase is to store a database containing the depth to mate (DTM) (number of moves up until checkmate) from all winning

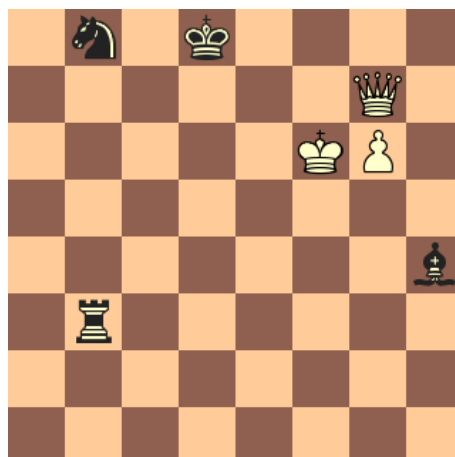


Figure 2.3: White to move can mate black in a forced way in an exuberant number of 549 moves. No engine can find the right sequence. This problem is left as an exercise to the reader

positions. All absent positions in the database are theoretical draws. By using a tablebase, we only have to maximize the DTM of all possible moves to achieve optimal play. Tablebases can be uploaded into engines, which will look up the position into the database and play the best move. Although they do not contain all positions up until seven pieces (this takes 130 000 GB on a hard drive), it greatly simplifies computations for plenty endgame scenarios (lom, 2012).

Tablebases have been generated with retrograde analysis, from all possible checkmate positions calculations worked backwards. This technique has unveiled winning sequences which are beyond human comprehension. The coming of tablebases has put the 50-move rule under pressure, although most 50+ winning sequences are a set of seemingly random piece dancing. An example has been provided in Figure 2.3.

## 2.4 Conclusion

To conclude this chapter, we analyze the characteristics of these approaches. Although chess engines today are much stronger than professional human players (compare the rating of the world champion which is around 2800 with table 2.2), we can never be sure with this approach if the engines are actually making the best decisions. These engines are designed in such a way that they do not play intuitively, but calculate as much variations as possible and base their decision upon that. The choices of variations to look at are based on human expert knowledge, hence the state of the art chess computers have been taught to choose actions in their search tree by human guidelines.

These choices have especially their consequences for positional and endgame play, which are far from optimal right now. The static position evaluations have been decided heuristically, which may not have been generalized enough for all chess positions (an example over-coloring this

Table 2.2: The 12 strongest chess engines, tested by letting them play against each other. These are the playing conditions: Ponder off, General book (up to 12 moves), 3-4-5 piece EGTB Time control: Equivalent to 40 moves in 40 minutes on Athlon 64 X2 4600+ (2.4 GHz), about 15 minutes on a modern Intel CPU. Computed on May 27, 2017 with Bayeselo based on 715'938 games (CC, 2017a). The strongest FIDE chess rating ever recorded (May 2014) attained by the world champion Magnus Carlsen (at the moment of writing) is included in the ranking for comparison.

Rank	Name	Rating	95% Confidence
1	Komodo 10.4	3392	[3372, 3412]
2	Stockfish 8	3390	[3373, 3407]
3	Houdini 5.01	3387	[3368, 3406]
4	Deep Shredder 13	3288	[3268, 3308]
5	Fire 5	3270	[3248, 3292]
6	Fizbo 1.9	3249	[3225, 3273]
7	Andscacs 0.89	3240	[3216, 3264]
8	Chiron 4	3208	[3184, 3232]
9	Gull 3	3194	[3183, 3205]
10	Equinox 3.20	3186	[3174, 3198]
11	Booot 6.1	3178	[3157, 3199]
12	Fritz 15	3171	[3158, 3184]
	...		
58	Magnus Carlsen	2882	

aspect is illustrated in figure 2.4).

Due to the tree operations in the search algorithm, there is often overhead by traversing bad lines. A lot of effort has been put in the selection of lines when branching through the tree by introducing heuristics into the move order. If the engines would somehow have better intuition in this idea, the computational effort could decrease.

Globally, these methods have already been optimized so much with parallelization and small optimizations, that it may be time to look for different methods to learn more about the beautiful game.

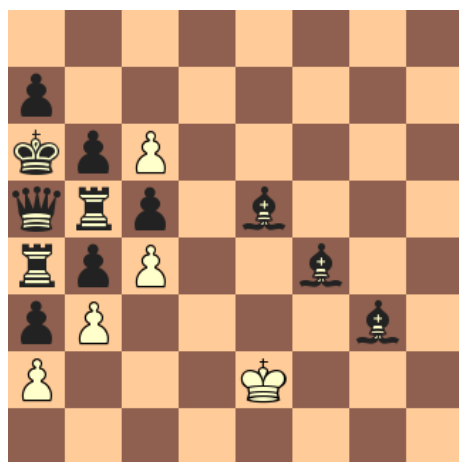


Figure 2.4: A toy example where chess engines fail to indicate the right evaluation. Because the static evaluations are favoring the material value count against the structure in an unbalanced way in this position, the computers indicate black will win almost certainly. In reality, white can actually force this game to be a draw relatively easily (Penrose, 2017).

## Chapter 3

# Deep Learning Background

In this chapter, we review the mathematical foundations for the methods used in this dissertation. We start off with a walk-through in the world of machine learning (section 3.1) and neural networks (section 3.2), after which we expand on the concept of reinforcement learning in section 3.3. These machine learning methods are combined into a bigger concept, namely deep reinforcement learning.

### 3.1 Machine Learning

Machine learning is a family of methods used to perform automatic data analysis gaining insight by finding patterns without having to be explicitly programmed. Due to the incredible growth of data everywhere, machine learning methods are booming in many fields, one of them being AI.

Machine learning problems have been classified into three categories, from which hybrids are possible:

- **Supervised learning.** The machine has access to labeled data. The labels are true solutions from practice mapped from the given input.
- **Unsupervised learning.** In this case, we have no access to labeled data, and the goal is to find some sort of structure in the data points, which we call clusters.
- **Reinforcement learning.** An agent has the task to discover an environment. The environment returns feedback signals (rewards and punishments) to the agent. The goal of the program is to find out how the agent can maximize its rewards. We will dive deeper into this topic in section 3.3.

#### Supervised Learning

In a supervised learning model, we have access to a dataset  $\{(x_1, y_1), \dots, (x_N, y_N)\}$ , with  $x_i \in \mathcal{X}$  a feature vector and  $y_i \in \mathcal{Y}$  the label corresponding to  $x_i$ . With the model, we associate a function  $f : \mathcal{X} \times \mathcal{W} \mapsto \mathcal{Y}$  with  $w \in \mathcal{W}$  denoted as the parameters or weights of the model. This function maps input data to a new label. The goal of a supervised learning algorithm is then

to optimize the weights in such a way to minimize a certain loss function  $L : \mathcal{Y} \times \mathcal{Y} \mapsto \mathbb{R}_{\leq 0}$ . In regression problems, where we want to estimate for every new data point  $x_i$  a label  $\hat{y}_i = f(x_i, w)$ , the most common loss function is the mean squared error (MSE):

$$L = \frac{1}{N} \sum_{i=1}^N (\hat{y}_i - y_i)^2 \quad (3.1)$$

By using a learning algorithm, we try to minimize the loss over the dataset. We then hope that the dataset was a good and diverse representation, such that  $f$  generalized well to the real world.

Here, we will use combinations of supervised and reinforcement learning to let the computer analyze how chess play can be improved. The data will be collected by an agent in the chess environment, and a supervised learning algorithm can be used on this data.

## 3.2 Neural Network

A neural network is a mathematical model typically used in machine learning (ML) tasks. The model describes how data points are converted into a class label (in case of classification) or a real value (regression). Then a supervised or unsupervised learning algorithm is applied to a dataset. This learning algorithm tunes the variables in the model in such a way, that it minimizes a certain cost or loss function. In case of supervised (which is what we will use to the highest degree in this thesis), this loss function is a measure of error between the output of the model and a true value from the dataset corresponding with the input. These true values can be seen as the target of the model to reach, as described in section 3.1.

Neural networks are a supervised learning model constructed in such a way that they more or less imitate the series of interconnected neurons in the human brain, with the philosophy to let a machine solve recognition and decision tasks in the same fashion as a human being would. We will use this model to estimate the static evaluation function of a chess position.

### 3.2.1 Feedforward Neural Network

#### Feedforward Propagation

An artificial neural network (ANN) is a NN in its most general form, where all nodes in the network can be connected. An FNN is an ANN without the possibility of loops and cycles, as shown schematically in figure 3.1. The computation of the output vector of a given input is called feedforward propagation. Inputs are fed linearly via a series of weights to every node in the first layer. The linear combination of inputs then undergoes a nonlinear transformation called the activation, which at its turn is fed forward into all the layers of the next node and so on and so forth up until the output layer. Nodes in the hidden layer are called hidden units, and are slackly analogous to axons in the human brain. The variables of this model are the weights of the linear combinations, every edge thus corresponds to a tunable parameter.

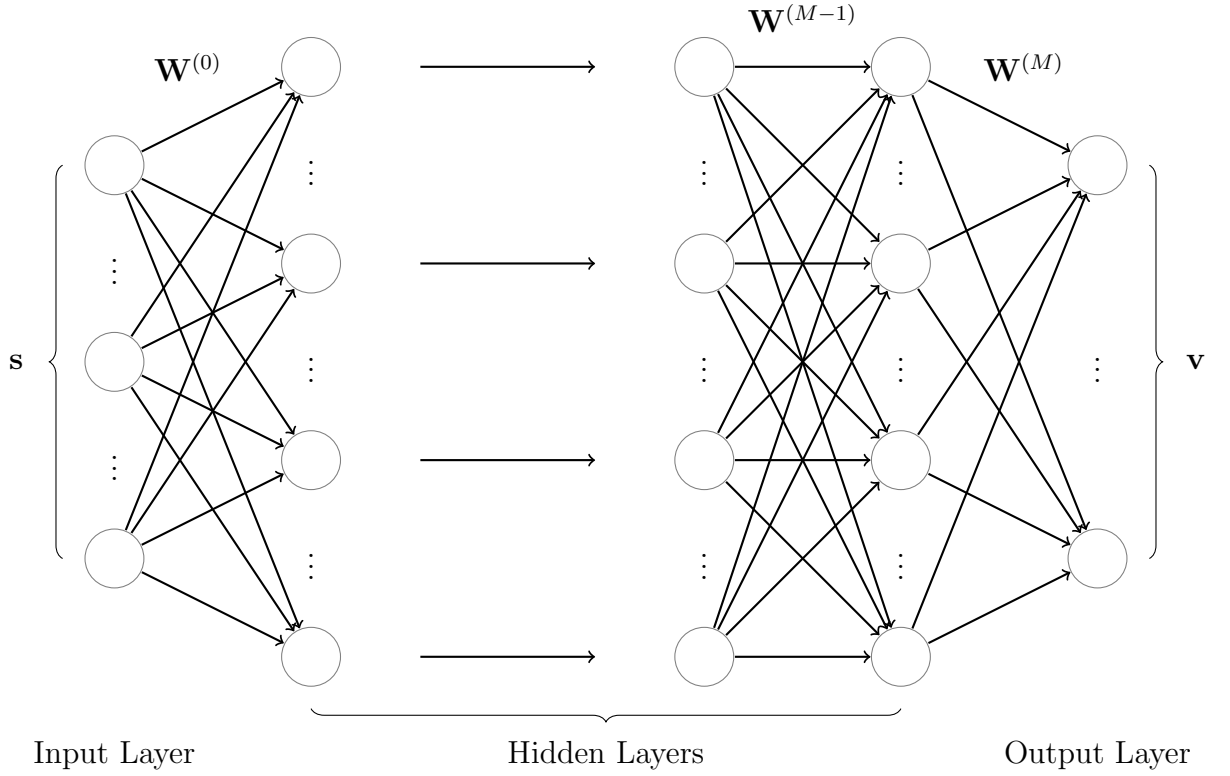


Figure 3.1: The general architecture of an FNN, also called an fully connected network (FCN) in literature.

Suppose we have  $\mathbf{s} = (s_1, s_2, \dots, s_D)$  as input vector, the resulting activation at a node  $n \in \{1, \dots, N_1\}$  in the first hidden layer is then

$$a_n^{(1)} = \sum_{d=1}^D w_{dn}^{(0)} s_d \quad (3.2)$$

where  $w_{dn}^{(0)}$  represents the weight between input node  $s_d$  and hidden unit  $a_n$ . We can rewrite all activation equations into matrix form with

$$\mathbf{a}^{(1)} = \mathbf{W}^{(0)\top} \mathbf{s} \quad (3.3)$$

where  $\mathbf{W}_{dm}^{(0)} = w_{dm}^{(0)}$ . If we call the nonlinear transformation of the activation  $\sigma(a)$ , the output of every hidden unit yields

$$z_n^{(1)} = \sigma(a_n^{(1)}) \quad (3.4)$$

$$= \sigma \left( \sum_{d=1}^D w_{dn}^{(0)} s_d \right) \quad (3.5)$$



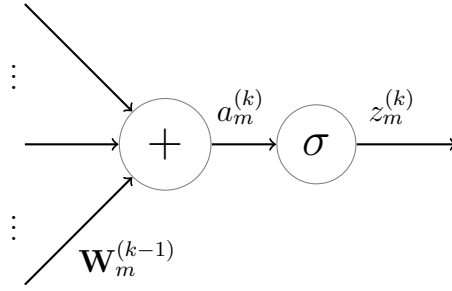


Figure 3.2: Hidden unit

so we obtain the matrix equation

$$\mathbf{z}^{(1)} = \sigma \left( \mathbf{W}^{(0)\top} \mathbf{s} \right) \quad (3.6)$$

By introducing an additional bias vector at the first layer, allowing translation of every activation function to the desired position, we obtain

$$\mathbf{z}^{(1)} = \sigma \left( \mathbf{W}^{(0)\top} \mathbf{s} + \mathbf{b}^{(0)} \right) \quad (3.7)$$

These calculations are performed in a chain-wise fashion, the outputs of the hidden units in every layer are propagated forward until we arrive at the output layer. Hence, for an arbitrary layer  $k$ , the activation equation looks like this (Figure 3.2):

$$\mathbf{z}^{(k)} = \sigma \left( \mathbf{W}^{(k-1)\top} \mathbf{s} + \mathbf{b}^{(k-1)} \right) \quad (3.8)$$

The final nonlinear equation describing the feedforward propagation of an NN is (supposing we have  $M$  hidden layers in total)

$$\hat{\mathbf{v}} = \sigma \left( \mathbf{W}^{(M)\top} \mathbf{s} + \mathbf{b}^{(M)} \right) \quad (3.9)$$

which is the estimate of a value from the glsNN, given an input sample.

### Nonlinearity

Generally, we want the nonlinear transformation of the activation to be some sort of threshold function (mimicking the biological decision of a neuron to let the information propagate further), but most importantly, it should be a differentiable function. This is because the most commonly used learning algorithm, backpropagation, uses differentiation in its optimization of the NN. Due to the recursive nature of the network function and backpropagation, we want the derivative of the activation function to be simple as well. Common choices are the sigmoid, tanh and rectifier linear unit (ReLU) function. They are compared in 3.1. From these functions, the ReLU seems to provide the best performances in most practical examples (Mishkin et al., 2016). It should be mentioned that some alternatives like the noisy and leaky ReLU can be used with some success (Nair and Hinton, 2010; Maas et al., 2014).

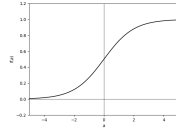
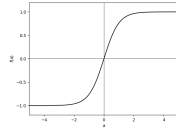
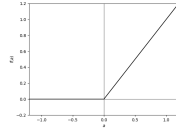
name	$\sigma(a)$	$\sigma'(a)$	chart
sigmoid	$\frac{1}{1+e^{-a}}$	$\sigma(a)(1 - \sigma(a))$	
tanh	$\tanh(a)$	$1 - \tanh(a)^2$	
ReLU	$\max(a, 0)$	$\begin{cases} 1 & \text{if } a > 0 \\ 0 & \text{if } a \leq 0 \end{cases}$	

Table 3.1: Comparison between activation functions

### Learning Algorithm

The goal of the learning algorithm is to tune all model parameters (the weights and bias) such that a global loss function on the training data is minimized. Supposing we have  $N$  data samples available, we write

$$L(w) = \frac{1}{N} \sum_{i=1}^N L_i(w) \quad (3.10)$$

so we want to minimize the sum of all errors on all individual samples. The choice of  $L_i(w)$  depends on the chosen problem.

To find a local optimum, most researchers use gradient descent, a first-order iterative optimization algorithm. It may not find the global optimum, but it is computationally efficient, because of the ease at finding the first partial derivatives of all weights with backpropagation. By updating the weights in the direction of the steepest descent (the gradient of the loss function with respect to all parameters) at each iteration, we gradually reduce the global loss.

$$w \leftarrow w - \alpha \nabla L(w) \quad (3.11)$$

$$= w - \alpha \sum_{i=1}^N \frac{\nabla_w L_i(w)}{N} \quad (3.12)$$

We observe the introduction of a new training parameter, the step size in the steepest direct also called the learning rate  $\alpha$ . It should be chosen wisely. The higher the learning rate, the faster the model will learn a local optimum, but if it is too big it might oscillate around the ravine. In contrast, the learning algorithm will be slower but more accurate for a smaller step size.

As our problem to approximate the static evaluation function of a chess position is a regression problem, we will use the MSE from section 3.1. By calculating the derivative we find the update rule for the weights to be

$$w \leftarrow w - \alpha \sum_{i=1}^N \frac{\nabla_w L_i(w)}{N} \quad (3.13)$$

$$= w - \frac{\alpha}{N} \sum_{i=1}^N \nabla_w (\hat{y}_i(w) - y_i)^2 \quad (3.14)$$

$$= w - \frac{\alpha}{N} \sum_{i=1}^N (\hat{y}_i(w) - y_i) \nabla_w \hat{y}_i(w) \quad (3.15)$$

The issue with using gradient descent as described above, is that it can be computationally expensive to perform a single step, as we have to run through the whole training set. By updating the model parameters in an online manner, i.e. update the model along every sample of the dataset, we achieve a faster convergence rate. This has to happen stochastically (we need to shuffle the dataset at every iteration). With the theories of convex minimization and stochastic approximation, it has been proven that almost surely the objective function will converge to a local minimum (Saad, 1998). The name of this online learning algorithm is stochastic gradient descent (SGD)

One remaining problem is that computing every update independently is slower than computing them in batch with vectorization libraries (eg. LAPACK). The solution to this problem is straightforward, we can make a compromise between gradient descent and SGD by doing the weight updates in minibatches. A high-level pseudo code description is laid out in Algorithm 4

---

**Algorithm 4** Minibatch stochastic gradient descent

---

**Input:**  $S, V, n$  //  $n$  is size minibatch,  $1 \rightarrow \text{SGD}$ ,  $N \rightarrow \text{GD}$

Initialize  $w$  (randomly),  $\alpha$

**repeat**

  shuffle samples

**for** all minibatches **do**

$$w \leftarrow w - \alpha \sum_{i=1}^n \frac{\nabla L_i(w)}{n}$$

**end for**

**until** local minimum is achieved

---

### Further optimizations

We initialize the weights  $w$  with random values. It is intuitive to understand that at the start of learning, we are allowed to make bigger steps into the right direction. Then, we could gradually decrease the value of the learning. In other words, it makes sense to use adaptive learning rates. Two notable and widely accepted adapted gradient mechanisms are *AdaGrad* (which bases its learning rate updates on weight changes so far) and *RMSProp* (less aggressive form of *AdaGrad*) (Duchi et al., 2011; Sutskever et al., 2012).

To reduce the oscillation of the random fluctuations of online learning, researchers came up with the momentum method, which (just like a real object) follows the steepest descent until it has enough velocity to deviate from it into the previous direction. The most famous momentum gradient descent learning algorithm is the *Nesterov* momentum. There are a lot more variations on standard minibatch SGD, for which I encourage the reader to read *Deep Learning* (Goodfellow et al., 2016a).

## Backpropagation

As described in the previous subsection, we need to calculate the gradient of the loss function with respect to the weights of all connections. The backpropagation algorithm is an efficient recursive algorithm achieving this goal. As the name suggests, we start by calculating the derivatives in the last layer, and let the information we get from calculating these derivatives propagate backwards to compute the weight derivatives of previous layers all the way back to the input layer. This is done with the chain rule of differentiation. More specific details and a proof can be found in *Deep Learning*.

### 3.2.2 Convolutional Neural Network

To further simulate the functioning of the brain and apply it to machine learning tasks, researchers created convolutional neural networks, a feedforward ANN where weights are shared between spatially correlated neurons, to give the network understanding of translation invariance. The connectivity pattern of neurons is inspired by the visual cortex. They have been applied successfully especially in the field of computer vision, but also in other domains. The main resource for this section is *Deep Learning* (Goodfellow et al., 2016a).

Convolutional layers are the core building blocks of CNNs, after which an FCN computes the output. These layers are stacks of feature maps. The deeper we go into the architecture, the larger the receptive field of every hidden unit, i.e. the number of input parameters that had an influence on the hidden node. The connections between neurons are filters. The idea is thus to learn the optimal filters we have to apply onto the receptive fields in order to achieve the best features possible in the next layer. Feedforward computations are accomplished with the convolution operation, hence the name.

We will later see in the next chapter how this architecture may be beneficial for the performance of board games like *go*, and maybe chess.

## Convolution

The convolution <sup>1</sup> operation is the application of weighted averaging of correlated measurements to account input noise into the output. We do this averaging for every input variable around its center. Suppose we have a 1-D tensor of discrete measurements, we write this down

---

<sup>1</sup>The presented convolution is actually a cross correlation in mathematics terminology

mathematically as

$$y(i) = (w * x)(i) \quad (3.16)$$

$$= \sum_n w(n)x(i + n) \quad (3.17)$$

For 2-D data (as images for instance) this convolution is

$$Y(i, j) = (W * X)(i, j) \quad (3.18)$$

$$= \sum_{m,n} W(m, n)X(i + m, j + n) \quad (3.19)$$

which are actually 2 1-D convolutions at the same time. The weight matrix is also called the kernel in machine learning terminology, while the filters are called kernels.

Why can using the convolution operation be so beneficial for machine learning tasks? It provides three important characteristics: sparse interactions, parameter sharing and equivariant representations (Goodfellow et al., 2016b).

The sparse interactions are attained by making the kernel smaller than the input, resulting in having to store less parameters.

Secondly, tied weights are used, which means that all windows of feature maps are transformed by the same kernels into the next layer. This as well reduces the size of the parameter space, but has also the consequence that instead of learning a separate set of parameters for every location of the input like in a simple FNN, the parameters will be shared location independently. Hence, the model becomes translation invariant. The parameter sharing and sparse interactions are visualized in 3.3.

The equivariance relationship ( $f$  and  $g$  are equivariant functions if and only if  $f(g(x)) \equiv (g(f(x)))$ ) is therefore attained for translation, but it can be achieved for other relations (eg. scale, rotation) as well by using different mechanisms.

## Convolutional Layer Architecture

Convolutional layers typically consist out of 3 main stages as shown in figure 3.4. The input is fed into the convolutional stage, which executes the convolution operations. This is followed by a nonlinear transformation just as in section 3.2. The convolutional layer can be finalized by an optional pooling stage if the model is used for computer vision purposes. The output of the last stage is then the input is then fed into the next convolutional layer, making CNNs a feedforward architecture.

Pooling is a method to introduce extra invariances by downsampling the information. Additional information can be found in the references, but this method is not from crucial importance to this master dissertation (Boureau et al., 2011).

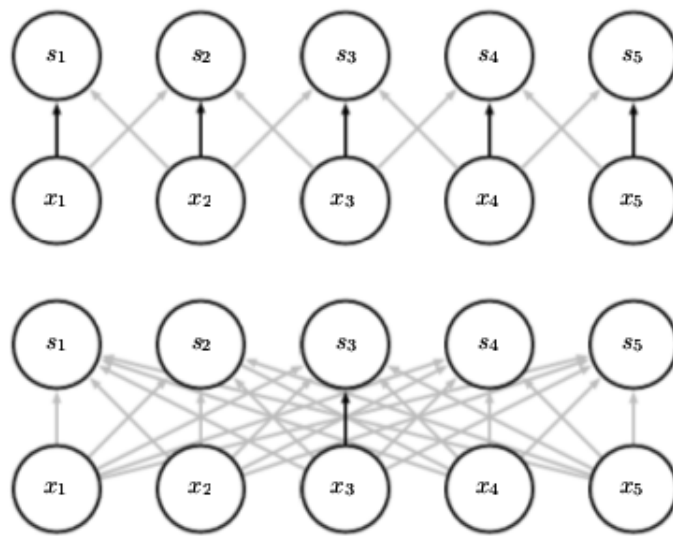


Figure 3.3: In an FCN, all units between layers are connected to each other, but in a CNN the nodes are only connected by the nodes from its receptive field in the previous layer. This makes the number of connections considerably smaller. The weights are also equal for every connection set between input nodes.

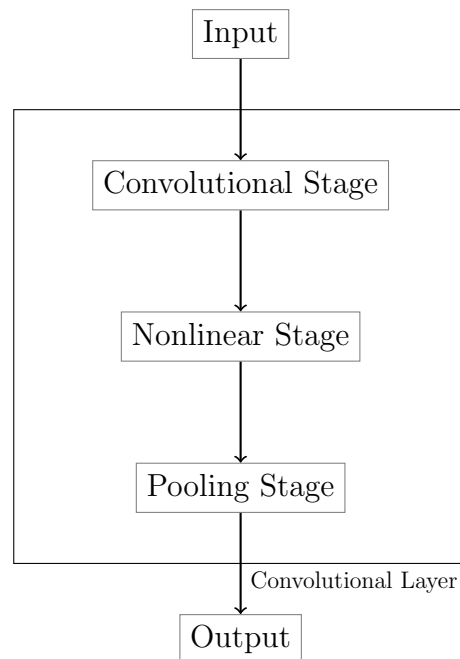


Figure 3.4: Convolutional layer flow diagram

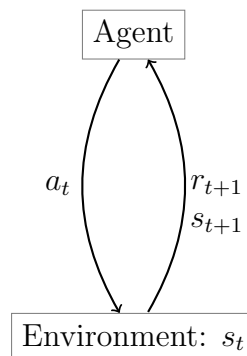


Figure 3.5: The general framework for RL, an agent is performing an action  $a_t$  in state  $s_t$ . By doing this action, it will receive its new state  $s_{t+1}$  with the reward  $r_{t+1}$  corresponding to the action at the next time step.

### 3.2.3 Deep learning

Deep learning algorithms are (as the name suggests) neural networks using an architecture with many hidden layers. The conditions for an architecture to be 'deep' are that (Deng and Yu, 2014):

- it is a composition of successive layers, using the output of previous layer as input
- it is based on hierarchical learning of feature representations of the data. The features learned in deeper layers are more complex and are derived from the low level features learned in earlier stages of the network.
- learn multiple representations on different abstractions

## 3.3 (Deep) Reinforcement Learning

Just like neural networks are inspired by the functioning of the brain, RL is a field in machine learning inspired by behaviorist psychology. The principal idea of solving problems in RL is to find from any given situation in an environment the optimal sequence of actions such that an agent in the environment maximizes its reward. The bigger the rewards, the closer the agent is to achieving its final goal. The agent has to find this sequence by exploring the environment, where actions in given states lead to certain rewards. To better understand the environment and increase future rewards, the agent has to balance between exploiting current knowledge and exploring new areas of the environment where he knows less about, this is the exploration-exploitation dilemma.

We start of this section with an explanation about Markov decision processes, the mathematical framework used for studying RL. We will find out that for an agent to interpret its environment better, it should sense which states are best to reside in. Background on these problems are provided in section 3.3.2. Similarly, we want to know which actions the agent should take to understand its environment to the best of his powers, this is examined in 3.3.3. This section is

concluded with algorithms using value prediction as well as control in section 3.3.4.

### 3.3.1 Markov Decision Process

Notations are inspired on the survey *Algorithms for Reinforcement Learning* (Szepesvári, 2010).

**Definition 1.** *An Markov decision process (MDP) is a 5-tuple  $\mathcal{M} = (\mathcal{S}, \mathcal{A}, \mathcal{P}, R, \gamma)$  with*

- $\mathcal{S}$ : state space
- $\mathcal{A}$ : action space
- $\mathcal{P} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \mapsto \mathbb{R}$ : state transition probability kernel
- $R : \mathcal{S} \times \mathcal{A} \mapsto \mathbb{R}$ : the expected immediate reward function
- $\gamma \in \mathbb{R} \cap (0, 1)$ : discount factor

With MDPs we can define sequential decision making problems in RL as follows. Suppose  $t \in \mathbb{N}$  denotes an instant of time, and we interact according MDP  $\mathcal{M}$ . When an agent chooses action  $a_t \in \mathcal{A}$  in state  $s_t \in \mathcal{S}$ , it gets fed back a reward  $r_{t+1}$  and arrives in a new state  $s_{t+1}$  with a probability  $P(s_{t+1}|s_t, a_t)$ . The expected reward at time step  $t$  was  $R(s_t, a_t)$ , but from now on we assume this reward to be deterministic given  $a_t$  and  $s_t$  to simplify the framework.

From this point on, the same decision process goes on and on until the agent arrives at a terminal state  $s_T$ . A full sequence  $\{s_1, s_2, \dots, s_T\}$  is called an episode. The goal of the agent is to maximize its return over the course of an episode. The return starting from  $t$  is defined as the exponentially discounted sum of the rewards obtained:

$$\mathcal{R} = \sum_{t=0}^{\infty} \gamma^t r_{t+1} \quad (3.20)$$

In this manner, MDPs can balance between being discounted ( $\gamma < 1$ ) and undiscounted ( $\gamma = 1$ ). The agent bases its future behavior on the history up until we arrived at  $s_t$  is defined by the sequence of tuples  $(s_0, a_0, r_1), (s_1, a_1, r_2), \dots, (s_{t-1}, a_{t-1}, r_t)$ .

From this point on, we will assume the MDP to be deterministic, i.e. only one state transition  $s_{t+1}$  is possible given  $(s_t, a_t)$ . To formally describe this, define the successor function  $\Sigma : \mathcal{S} \times \mathcal{A} \mapsto \mathcal{S}$ . The state transition probability kernel simplifies to

$$P(s_{t+1}|s_t, a_t) = \begin{cases} 1 & \Sigma(s_t, a) = s_{t+1} \\ 0 & \text{else} \end{cases} \quad (3.21)$$

In the next chapter, we will see how chess can easily fit into this framework of MDPs as a tool to learn to play chess.

### Value Function

Before we continue, we define the concept of a policy.

**Definition 2.** *A policy is a probability function  $\pi : \mathcal{S} \times \mathcal{A} \mapsto \mathbb{R}$ . An agent is following a policy  $\pi$  if it draws its next action  $a \in \mathcal{A}$  in state  $s \in \mathcal{S}$  randomly with probability  $\pi(a|s)$*



To find the optimal behavior in an MDP, value functions are used. Value functions denote how good it is for an agent to be in certain states. For chess boards, this is equivalent to the static evaluation function.

**Definition 3.** *The value function  $V^\pi : \mathcal{S} \mapsto \mathbb{R}$  with underlying policy  $\pi$  is defined by*

$$V^\pi(s) = \mathbb{E}[\mathcal{R}|s] = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t r_{t+1} | s\right] \quad (3.22)$$

*with the understanding that all triplets  $(s_t, a_t, r_{t+1})$  came up by following  $\pi$ .*

In contrast to value functions, action value functions stand for the profit achieved by performing an action in a certain state.

**Definition 4.** *The action value function  $Q^\pi : \mathcal{S} \times \mathcal{A} \mapsto \mathbb{R}$  with underlying policy  $\pi$  is defined by*

$$Q^\pi(s, a) = \mathbb{E}[\mathcal{R}|s, a] = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t r_{t+1} | s, a\right] \quad (3.23)$$

*with the understanding that all triplets  $(s_t, a_t, r_{t+1})$  came up by following  $\pi$ .*

An optimal policy can be derived easily supposing we have found the optimal value for every state in the state space, which is the highest achievable return starting from a state. The optimal value function  $V^* : \mathcal{S} \mapsto \mathbb{R}$  attains this most favorable mapping. We define the optimal action value function  $Q^* : \mathcal{S} \times \mathcal{A}$  analogously. The policy performing greedy actions only with respect to  $Q^*$  is an optimal policy. As we can relate  $V^*$  and  $Q^*$  to each other by the coming equations:

$$V^*(s) = \max_{a \in \mathcal{A}} Q^*(s, a) \quad (3.24)$$

$$Q^*(s, a) = R(s, a) + \gamma V^*(\Sigma(s, a)) \quad (3.25)$$

we can also act optimally according to  $V^*$  by acting greedily with respect to the next state's value function.

The remaining question is now on how to find these optimal functions. A tool used for this are the recursive Bellman equations (Bertsekas and Shreve, 2007).

**Theorem 1.** *The optimal value function satisfies*

$$V^*(s) = \max_{a \in \mathcal{A}} \{R(s, a) + \gamma V^*(\Sigma(s, a))\} \quad (3.26)$$

*The optimal action value function satisfies*

$$Q^*(s, a) = R(s, a) + \gamma \max_{a' \in \mathcal{A}} Q^*(\Sigma(s, a), a') \quad (3.27)$$

### 3.3.2 Value Prediction

#### Monte Carlo method

Let us consider the problem of the estimation of the value function. The closer we can approach the optimal value function with our estimate, the better our policy will be. We defined the value function earlier as the expected return starting from a certain state 3.22. This definition gives us the insight that a possible algorithm to create approximations might be a Monte Carlo algorithm, averaging out all returns.

Suppose we have got an MDP  $\mathcal{M} = (\mathcal{S}, \mathcal{A}, \mathcal{P}, R, \gamma)$  that generated  $N$  episodes following a certain policy, with each episode having an index  $n$  and history

$$H_n = \{(s_0, a_0, r_1), (s_1, a_1, r_2), \dots, (s_{T_n-1}, a_{T_n-1}, r_{T_n})\}$$

Store the observed returns for every state in time in every episode

$$\mathcal{R}_n(t) = \sum_{t'=t}^{T_n-1} \gamma^{t'-t} r_{t'+1} \quad (3.28)$$

We can now sensibly update the value function after the termination of an episode with a gradient descent kind of optimization step with the following update function

$$\hat{V}(s_t) \leftarrow \hat{V}(s_t) + \alpha (\mathcal{R}_n(t) - \hat{V}(s_t)) \quad (3.29)$$

where  $\alpha$  is the learning rate and  $\mathcal{R}_n(t) - \hat{V}(s_t)$  is the gradient of the MSE between the actual observed return and the estimated return. We can find a nice recursive relationship between observed returns in an episode:

$$\begin{aligned} \mathcal{R}_n(T-1) &= r_T \\ \mathcal{R}_n(t) &= r_{t+1} + \gamma \mathcal{R}_n(t+1) \end{aligned}$$

This recursive relation is used in Algorithm 5, where the history of an episode is interpreted to update the value function. The presented algorithm is called the every visit Monte Carlo (EVMC) algorithm. At the moment, we are assuming tabular methods, where the value function is stored for each distinct state independently in an array.

---

**Algorithm 5** Episodical EMVC

---

**Input:**  $H = \{(s_0, a_0, r_1), (s_1, a_1, r_2), \dots, (s_{T_n-1}, a_{T_n-1}, r_{T_n})\}, V(s)$   
 $\mathcal{R}_n \leftarrow 0$   
**for**  $t \leftarrow T-1$  **downto** 0 **do**  
     $\mathcal{R}_n \leftarrow r_{t+1} + \gamma \mathcal{R}_n$   
     $V(s_t) \leftarrow V(s_t) + \alpha (\mathcal{R}_n - V(s_t))$  // Gradient Descent update step  
**end for**

---

In practice, EVMC stabilizes for the tabular method (which is only feasible for a limited amount of states) almost surely to the optimum. The issue is the variance between the calculated returns

over episodes, which can be very high because we take every visited state into account (this is a general problem of Monte Carlo (MC) simulations in general (Robert and Casella, 2005)), resulting in poor estimates at first and a long training time.

### TD(0)

To address the issues of EVMC, we will limit the influence of states further away in the episode with temporal difference learning, the most drastic one being TD(0), where we only use the value of the next state in our estimate. As in EVMC, we will use an online gradient descent algorithm. The update function is now (under the same MDP conditions as in the MC method described earlier)

$$\hat{V}(s_t) \leftarrow \hat{V}(s_t) + \alpha \left( r_{t+1} + \gamma \hat{V}(s_{t+1}) - \hat{V}(s_t) \right) \quad (3.30)$$

So basically, we have slightly modified the target for a state to a linear approximation of the true return in the episode. The TD-error is  $r_{t+1} + \gamma \hat{V}(s_{t+1}) - \hat{V}(s_t)$ . States further away than the next state do not influence the target anymore. These were exactly the states entailing the largest amount noise in MC measurements. Another advantage over EVMC is its suitability for incremental learning, meaning we can update our value function after every action, as in algorithm 6.

---

#### Algorithm 6 TD(0)

---

**Input:**  $s_t, r, s_{t+1}, V$

$V(s_t) \leftarrow V(s_t) + \alpha (r_{t+1} + \gamma V(s_{t+1}) - V(s_t))$  // gradient descent update step

---

Just like EVMC, TD(0) converges almost surely to  $V^*$ . One may wonder which algorithm works best and in which cases, as they both have their own merits. An example where TD(0) is slower than EVMC is a long chain of states where the agent only gets a positive reward at the final action, because the TD-error has to propagate back to the first states. So the longer the chain, the longer it will take until the initial states have learned which paths to follow.

### TD( $\lambda$ )

The attentive reader may have wondered why temporal difference learning is called TD(0). The reason behind the zero in the name is because it is actually a special case of TD( $\lambda$ ), an algorithm unifying EVMC and TD(0). First of all, let us define the  $n$ -step return, which is a prediction of the actual return taking  $n$  states ahead into account.

$$G_n(t) = \left( \sum_{i=0}^{n-1} \gamma^i r_{t+i} \right) + \gamma^n V(s_{t+n}) \quad (3.31)$$

We notice  $G_1(t)$  to be the target return in TD(0) and  $G_{T-t}(t)$  the real return value given  $T$ , the length of the episode. We construct the  $\lambda$ -return with all  $n$ -step returns with exponential decay as mixing weights.

$$G_\lambda(t) = (1 - \lambda) \sum_{n=1}^{\infty} \lambda^{n-1} G_n(t) \quad (3.32)$$

where it can easily be checked that  $(1 - \lambda)$  is a normalization factor.  $\lambda$  is the trace decay parameter, and controls to what extent the algorithm resembles on TD(0) or a Monte Carlo algorithm.  $\lambda = 0$  gives TD(0). The finite version of equation 3.32 is

$$G_\lambda(t) = (1 - \lambda) \sum_{n=1}^{T-t} \lambda^{n-1} G_n(t) + \lambda^{T-t} G_t \quad (3.33)$$

where we define  $G_t$  to be all the subsequent  $n$ -step returns after the terminal state.

An incremental algorithm can be designed by using accumulating eligibility traces, which can be seen as some sort of history of past states. The TD-error propagates back to all past states when the incremental update occurs. Suppose  $z$  is the eligibility trace, appropriate incremental updates are then (Sutton and Barto, 1998)

$$z(s_t) \leftarrow 1 + \gamma \lambda z(s_t) \quad (3.34)$$

$$V(s_t) \leftarrow V(s_t) + \alpha (r_{t+1} + \gamma V(s_{t+1}) - V(s_t)) z(s_t) \quad (3.35)$$

Algorithm 7 gives pseudocode for a tabular TD( $\lambda$ ) algorithm

---

**Algorithm 7** Tabular TD( $\lambda$ )

---

**Input:**  $s_t, r_{t+1}, s_{t+1}, V, z$

**for**  $s \in \mathcal{S}$  **do**

$$z(s) \leftarrow \gamma \lambda z(s) + \mathbb{1}_{\{s=s_t\}}$$

$$V(s) \leftarrow V(s) + \alpha (r_{t+1} + \gamma V(s_{t+1}) - V(s_t)) z(s_t)$$

**end for**

---

In summary, TD( $\lambda$ ) tries to assemble the advantages of both TD-learning and Monte Carlo methods. Just as these algorithms, it converges to  $V^*$ . Finding the right value of  $\lambda$  for quicker convergence is often a process of trial and error, and it has even been proved that changing  $\lambda$  during training does not impact convergence under the right circumstances (eg. eligibility trace update function) (Bertsekas and Tsitsiklis, 1996).

### Function approximation

The methods considered so far consider a tabular value function, but many problems like chess have such a large state space that saving all their values is infeasible. We choose to describe all states with feature vectors. A state  $s \in \mathcal{S}$  is transformed to its feature vector of dimension  $D$  by a feature extraction method  $\phi : \mathcal{S} \mapsto (\mathbb{R})^D$ . The value function can then be constructed around a mathematical model operating on these feature vectors  $\phi(s)$ . This model introduces a new set of parameters, the weights  $w$  of the model.

Some examples of function approximation methods are

- linear:  $V_w(s) = w^T \phi(s)$
- neural networks (see Section 3.2)
- other mathematical models for machine learning

Deep RL is the class of RL algorithms using deep neural networks as function approximation method. From now on it is assumed neural networks are used for this task. The tabular methods we have discussed so far now needs to be generalized to algorithms with function approximation. As Monte Carlo algorithms and TD(0) can be seen as special cases of TD( $\lambda$ ), we will limit our examination to TD( $\lambda$ ). An example of a non incremental episodic learning algorithm works as follows:

1. run episode and get history  $\mathcal{H}_n$
2. calculate the  $\lambda$ -returns for all states in  $\mathcal{H}_n$
3. use the  $\lambda$ -returns as target values for supervised learning on a neural network.
4. update weights of neural network with gradient descent

As we want to learn a continuous value function, this is a regression problem. A classical loss function to optimize in regression problems is the Mean Squared Error (MSE):  $L_w(\hat{V}, V) = (\hat{V} - V)^2/2$ .

When using non-linear value function approximators, convergence is no longer guaranteed (Baird, 1995). However, the algorithm generally still finds good results in practice under the right choice of hyper-parameters.

### 3.3.3 Control

We primarily focus on learning problems where the agent can influence its observations, which is interactive learning. The agent is making decisions changing the state of the environment and improving its policy. There are two ways for the agent to improve its policy. It can use active learning, where it will control the samples such that it maximizes finding a good policy or it can use online learning, where the online performance is optimized.

#### Closed-loop Online Learning

It is from crucial importance in online learning to occasionally choose actions that at the time might look suboptimal, to augment the future payoff. This question is then how frequent the agent must balance its choice to explore or exploit the current environment. This means that the methods described in this section can be used independently from which value functions are used.

**$\epsilon$ -greedy policy** This is a very simple strategy to balance exploitation and exploration, choose a fixed parameter  $\epsilon \in \mathbb{R} \cap [0, 1]$  and at every state choose a random action with probability  $\epsilon$ . Although this method is really simple, it can be competitive with more sophisticated algorithms if we some time decay is employed and tuned. Time decay is beneficial as normally the policy should improve, hence we should exploit more as time goes on.

**Boltzmann exploration strategy** With  $\epsilon$ -greedy, actions are drawn uniformly with a probability  $\epsilon$ . We may want to draw seemingly worse actions with a lower probability. To attain this

property, we can use the distribution

$$\pi(a|s) = \frac{e^{\beta J(s,a)}}{\sum_{a' \in \mathcal{A}_s} e^{\beta J(s,a')}} \quad (3.36)$$

$\beta$  controls how greedy the action will be, while  $J : \mathcal{S} \times \mathcal{A} \mapsto \mathbb{R}$  is a function indicating how good the action is, like  $V(\Sigma(s))$  or  $Q(s, a)$ .

### Q-learning

By trying to learn  $Q^*$  immediately, a greedy policy may be close to optimal.

With tabular Q-learning, an estimate for every  $(s, a)$ -tuple is stored and updated incrementally. Just as in TD-learning, this update happens in a gradient descent manner, by moving the Q-value in the direction of the last error between the target and previous estimate. Upon the observation of  $(s_t, a_t, r_{t+1}, s_{t+1})$ , the Q-value changes to

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left( r_{t+1} + \gamma \max_{a' \in \mathcal{A}} \{Q(s_{t+1}, a')\} - Q(s_t, a_t) \right) \quad (3.37)$$

Analogous to TD-learning, the target value of our estimate is  $r_{t+1} + \gamma \max_{a' \in \mathcal{A}} \{Q(s_{t+1}, a')\}$ .

Q-learning is very attractive due to its simplicity, and its convergence to  $Q^*$ . The policy followed by the agent could be  $\epsilon$ -greedy or *Boltzmann* exploration.

Incremental Deep Q-learning algorithms can then be generated with the extension to function approximation for  $Q_w$ :

$$w \leftarrow w + \alpha \left( r_{t+1} + \gamma \max_{a' \in \mathcal{A}} \{Q_w(s_{t+1}, a')\} - Q_w(s_t, a_t) \right) \nabla_w Q_w(s_t, a_t) \quad (3.38)$$

This update rule is used plenty of times in practice. However, its convergence properties are doubtful to say at least, especially when non linear approximation methods like deep learning are used.

Analogous to our deep TD( $\lambda$ ) algorithm, we can use episodic experience to learn the optimal weights of our network with regression. Good empirical results have been obtained by adding new samples to the data used for the previous iteration (Riedmiller, 2005).

#### 3.3.4 Value and Policy Iteration

The presented Q-learning and TD-learning algorithms are value iteration algorithms, by exploring the environment the value function changes up until the point it has converged. There are two weaknesses to applying value iteration only:

- may take a long time to converge
- we actually try to find the optimal policy, the value function is just a tool to do so

To incorporate the updates of policies, we can iterate over value iteration and policy iteration.

1. pick a random policy  $\pi$

2. update  $V^\pi$  under the policy
3. improve  $\pi$
4. go back to step 2 if satisfied

We actually already discussed a value and policy iteration algorithm, namely the use of the *Boltzmann* distribution to choose the next action. In that example, we updated the policy in an incremental fashion, as there is an immediate relationship between the policy and the value function.

## Chapter 4

# Deep Reinforcement Learning in Chess (and other games)

In this chapter, we will explore how we can use deep RL and mechanisms discussed in previous chapters to let a machine play games, and chess especially as it is the main focus of this dissertation. As we will have to feed chess positions into the neural networks, we will have to transform them into a vector representation. One should not forget that the starting point of the machine should be only knowledge about the game, and no expert knowledge. How this is done and how we can do it in chess is discussed in section 4.1. We need a function transforming this vector data into an evaluation, this is considered in 4.2. The application of  $TD(\lambda)$  and more sophisticated versions of TD-learning to games is further examined in section 4.3. A method that has shown recent success is presented in section 4.5 and we conclude this chapter by also considering how we could introduce policy iterations into chess in section 4.6.

### 4.1 Feature Space

Because the state space of most games is excessively big, tabular methods have no chance of being successful. This fact justifies using function approximation on a vectorized representation of game states. The feature vector is then fed into a machine learning framework of choice, often (deep) neural networks and CNNs.

The goal of our feature vector should be general and low level enough, to avoid being biased by the features. Let us examine the chess representations used by other machine learning engines for inspiration.

In section 2.2 we already described how most engines represent the chess position with hand-crafted expert features among features that only contain knowledge strictly characteristic to the model of the game. Also *Meep*, a RL based chess program, maintains

- material
- piece square tables



- pawn structure
- mobility
- king safety

information in a mostly binary fashion, which makes it a sparse representation (Veness et al., 2009). To arrive to a unbiased and general representation of the chess board, we will combine ideas from computer vision and another board game, *go*.

Nowadays, most object recognition and category learning tasks in computer vision are performed with the help of deep neural networks, especially CNNs. These networks take raw image data over several channels (often a separate channel for each color channel) as input, and the actual knowledge is learned over the course of several layers (Krizhevsky et al., ). This is in contrast of using a different model with features obtained by image analysis like *histogram of gradients* and a *bag of words* representation extracted with help of clustering over a large dataset (Lowe, 2004). This idea propagated further into the RL field with *Atari* games, where deep Q-learning is applied on raw image data as state representations (Mnih et al., 2013). The climax was reached when researchers implemented *AlphaGo*. This engine mastered *go* through self play and feeding board positions as a stack of binary image data into a CNN to learn the evaluation function with supervised learning (David Silver, 2016).

We can try to do the same in chess as in *go*, by stacking up feature planes describing the position and mobility of the pieces in a binary fashion. We will call these maps the piece maps and mobility maps. The idea used is the same as the concept of bitboards, a data structure commonly used in chess engines to describe a board state. A bitboard is defined as an array with the dimension of a chessboard ( $8 \times 8$ ), where each element describes the state of the square with a 1 or a 0. By stacking bitboards for every piece type separately with an indication of their positions and where they can go to, we create piece maps and mobility maps respectively. An example for the position in figure 4.1 that occurred in a famous game is provided in table 4.1. The important aspect to remember here is that no other information than the rules of chess are applied. We will call these features the bitboard features in the rest of this document.

Next to this, we can add additional (redundant) information to the features to help the model in question to learn patterns. Examples of additional features are

- side to move
- indication for each castle if it is still possible
- possibility to do en passant for each of the 16 squares where this could be a possibility
- piece count for each piece type
- piece mobility: number of squares each piece can go to
- for each piece (type) the number of pieces it attacks
- for each piece (type) the number of pieces it defends

We will call this data global features from now on. Some of these could be encoded with image maps as well. For example, every cell could denote the number of times it is attacked and

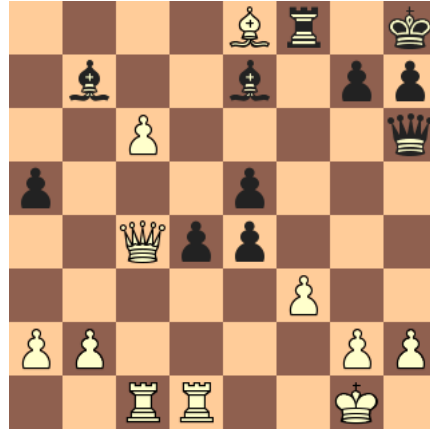


Figure 4.1: chess position coming out of the game *Alexander McDonnell vs Louis Charles Mahe De La Bourdonnais*, London 1834 after 24. c6













piece	piece map	mobility map	piece	piece map	mobility map
	0 1 0	0 1 0 0 0 0 0 0 0 1 0 1		0 0 0 0 0 0 0 1 0	0 0
	0 1 0	0 0 0 0 0 0 1 0 0 0 0 0 0 1 0 0 1 0 0 0 1 0 0 0 0 1 1 1 0 0 0 0 1 1 0 1 0 0 0 0 0 0 1 1 0 0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 0 1 0 0		0 1 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 0 0 0 0 0 0 0 1 1 0 0 0 0 0 1 0 1 0 0 0 0 1 0 0 1 0 0 0 1 0 0 0 1 0 0 1 0 0 0 0 0
	0 1 1 0 0 0 0	0 1 0 0 0 0 0 0 1 1 0 0 0 0 0 0 1 1 0 0 0 0 1 1 0 0 1 1 0 0		0 0 0 0 0 1 0	0 0 0 0 1 0 1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1 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 1 0	0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 1 0		0 0 0 0 0 0 0 0 0 1 0 0 1 0	1 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 1 0 1 0 0 0 0 1 0 0 0 1 0 0 1 0 0 0 0 0 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 1 0 1 0 0 1 1 0 0 0 0 1 1 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 1 1 0 1 1 0 0 1 1 1 1 1 1 0 0 0 0 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 1 1 0 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 0 0 0 1 1 0	0 1 0 0 0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Table 4.1: An example of how the chessboard from figure 4.1 can be transformed to a stack of binary image channels

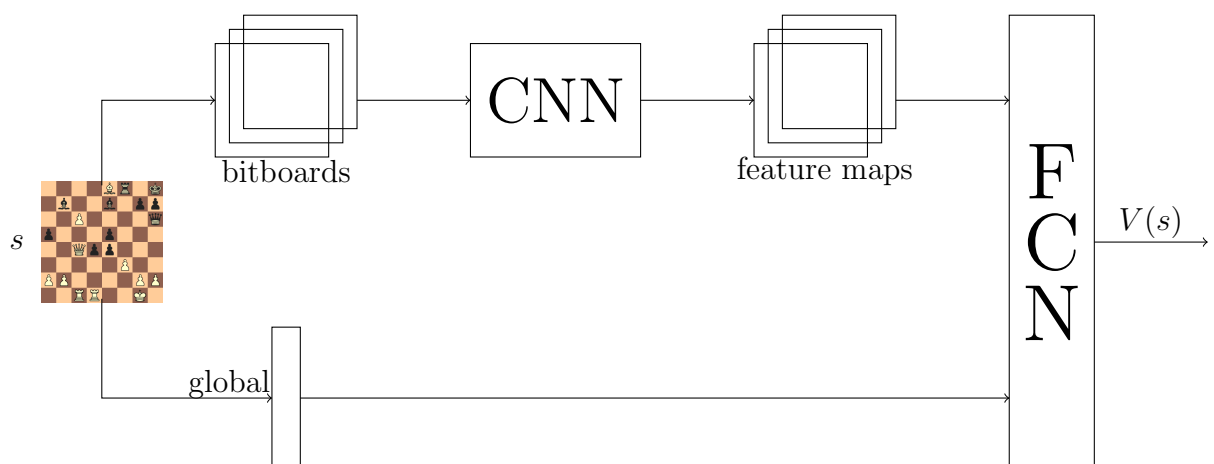


Figure 4.2: Schematic overview of a possible network architecture for chess using CNNs

defended.

## 4.2 Static Evaluation Function

In conventional chess engines, the heuristic evaluation function is linear, but also reinforcement learning chess programs often use a linear function:  $V_w(s) = \sum w f(s)$  (Baxter et al., 1999; Veness et al., 2009). In this dissertation, we switch gears and use a neural network model instead, as they have the ability to model a larger range of functions and extract additional features from the data.

The chess engine *Giraffe*, which uses a TD-learning algorithm, uses an FNN architecture with 2 hidden layers (Lai, 2015). A totally different approach has been taken in *DeepChess*, where the evaluation function is a deep siamese network learned on a big dataset of games between grandmasters. Two positions are given as input and the network returns which position is best. Playing is then performed with a comparison-based  $\alpha\beta$ -search. Admirably, *DeepChess* succeeded at reaching a grandmaster level of play (David et al., 2016).

To our best knowledge, no one has experimented with a CNN architecture to evaluate chess positions yet. The feature space presented in the previous section forms an intuitive basis to start with. The distinction between bitboard and global features demands for an architecture where the channel stack is the input of a CNN, where some form of feature extraction is carried out. Next, the resulting feature maps are combined with the global feature vector into a FCN with regression, as shown in figure 4.2.

## 4.3 TD algorithms

As investigated in section 3.3.2, TD algorithms are ideal in a machine learning framework, because of its natural embedding of the optimization of the MSE cost function in its update rule.

We saw that a  $TD(\lambda)$  approach is often the best choice, as it combines the best of both MC methods and classical TD-learning. We shortly review  $TD(\lambda)$  in section 4.3.1 and analyze its shortcomings, after which we consider alternatives using both the ideas of  $TD(\lambda)$  and minimax search (for which it is of course more efficient to apply the  $\alpha\beta$ -search). These methods slightly differ in terms of the episodic weight updates.

Before we continue, we need to revise the RL framework. In literature, the RL framework for boardgames is often a simplification of the classical one represented in section 3.3. The reward function  $r : \mathcal{S} \times \mathcal{A} \mapsto \mathbb{R}$  in boardgames is often chosen to be

$$r(s_t, a_t) = \begin{cases} W(s_{t+1}) & \text{if } s_{t+1} \text{ is terminal} \\ 0 & \text{if else} \end{cases} \quad (4.1)$$

where  $W(s_{t+1})$  denotes the objective evaluation of the terminal state of the game (do not confuse with the static evaluation function). For chess, this objective evaluation can be (assuming  $s$  is a terminal state)

$$W(s) = \begin{cases} 1 & \text{win} \\ 0 & \text{draw} \\ -1 & \text{loss} \end{cases} \quad (4.2)$$

Notice how this definition of the reward function meets the zero sum property.

Previous research in RL in games mainly do not take the discount factor into account when performing a TD-learning. This is rationalized by considering the return  $\mathcal{R} = \sum_{t=0}^T \gamma^t r(s_t, a_t)$ . As  $r(s_t, a_t)$  is zero everywhere except at the final state, it may be beneficial to assign the final reward as the return of every action, especially in TD-learning algorithms. The reason for this is a remark made in section 3.3.2 about the otherwise slow convergence properties in chaining.

In what follows, we will consider two main ideas to compute a target:

- episodic updates of the weights. After each episode the weights can be updated with an update rule. We could say this is a way of doing incremental learning over episodes. Here it is important to notice that the updates are heavily correlated, which may be the cause of bad generalization if we were to use them incrementally. The update rules make use of the assumptions about the simplified reward function and neutralized discount factor.
- translation to supervised learning framework. What is needed, are a set of tuples  $(s, \mathcal{R})$  where the return operates as a target value to learn  $V(s)$ . By using this approach, we can choose our favorite supervised learning algorithm from section 3.2.1. We can gather these tuples over the course of many episodes and shuffle them before applying mini-batch SGD (which is in general beneficial for the performance in software implementations) to improve generalization.

### 4.3.1 TD( $\lambda$ )

TD( $\lambda$ ) was the first RL algorithm that proved to be effective for learning to play a game. *TD-Gammon* used a neural network with one fully connected sigmoidal hidden layer to learn a value function for backgammon (Tesauro, 1995). Assume from now on

$$V_{est}(s_t) = \begin{cases} r_t & \text{if } t = T \\ \hat{V}(s_t) & \text{else} \end{cases} \quad (4.3)$$

Under the conditions  $\gamma = 0$  and equation 4.1 the temporal difference is defined to be

$$\delta_t = V_{est}(s_{t+1}) - V_{est}(s_t) \quad (4.4)$$

If we use a learning rate  $\alpha$ , we can now update the parameters after each episode with a history  $(s_0, a_0, r_1), (s_1, a_1, r_2), \dots, (s_{T-1}, a_{T-1}, r_T)$  as follows:

$$w \leftarrow w - \alpha \sum_{t=0}^{T-1} \nabla_w V_{est}(s_t) \sum_{n=t}^{T-1} \lambda^{n-t} \delta_n \quad (4.5)$$

This is equivalent to an episodic gradient descent update minimizing the MSE with the  $\lambda$ -return as target in supervised learning context. Remember that the general regression supervised learning paradigm can be summarized for a target  $z$  with the formula

$$\Delta w = \alpha \nabla_w V_{est}(s_t)(z_t - V_{est}(s_t)) \quad (4.6)$$

Over the course of an episode  $[s_0, r_1, s_1, \dots, r_T, s_T]$ , the total update amounts to

$$\Delta w = \alpha \sum_{t=0}^{T-1} \nabla_w V_{est}(s_t)(z_t - V_{est}(s_t)) \quad (4.7)$$

Hence, if equation 4.5 were a gradient descent update, following equality should hold:

$$z_t - V_{est}(s_t) = \sum_{n=t}^{T-1} \lambda^{n-t} \delta_n \quad (4.8)$$

Working this out yields

$$\begin{aligned}
z_t &= \sum_{n=t}^{T-1} \lambda^{n-t} [V_{est}(s_{n+1}) - V_{est}(s_n)] + V_{est}(s_t) \\
&= \lambda^0 [V_{est}(s_{t+1}) - V_{est}(s_t)] \\
&\quad + \lambda [V_{est}(s_{t+2}) - V_{est}(s_{t+1})] \\
&\quad + \dots \\
&\quad + \lambda^{T-1-t} [V_{est}(s_T) - V_{est}(s_{T-1})] + V_{est}(s_T) \\
&= \sum_{n=0}^{T-2-t} (\lambda^n - \lambda^{n+1}) V_{est}(s_{t+1+n}) \lambda^{T-1-t} V_{est}(s_T) \\
&= (1 - \lambda) \sum_{n=1}^{T-1-t} \lambda^{n-1} V_{est}(s_{t+n}) + \lambda^{T-1-t} V_{est}(s_T) \\
&= (1 - \lambda) \sum_{n=1}^{T-1-t} \lambda^{n-1} V_{est}(s_{t+n}) + \lambda^{T-1-t} (1 - \lambda) V_{est}(s_T) + \lambda^{T-t} V_{est}(s_T) \\
&= (1 - \lambda) \sum_{n=1}^{T-t} \lambda^{n-1} V_{est}(s_{t+n}) + \lambda^{T-t} V_{est}(s_T)
\end{aligned}$$

Combining the defined reward function,  $\gamma = 1$  and equation 3.31 for the n-step return, we notice how the n-step return is equal to the approximated value function n steps ahead:  $G_n(s_t) = V_{est}(s_{t+n})$ . The return after the terminal state is trivially estimated to be the value function at the terminal state:  $V(s_T)$ . Hence we achieve (using equation 3.33) our final result

$$z_t = (1 - \lambda) \sum_{n=0}^{T-t} \lambda^{n-1} G_n(s_t) + G_t \quad (4.9)$$

$$= G_\lambda(s_t) \quad (4.10)$$

The amazing thing about *TD-Gammon* is that its level surpassed that of humans by using strategies that were considered bad at that time. These techniques were later incorporated into humans play. One characteristic about its evaluations was that objectively better positions were estimated a smaller value than worse positions. Still, the engine was able to choose the best move. *Tesauro* explains this by the huge similarity between consecutive positions for the neural network, resulting in a smaller difference. For choosing the right move, the relative error is much more important than the absolute error.

Another key to the success of *TD-Gammon* is the stochasticity of the game. Thanks to the stochastic noise the agent is able to discover new strategies much quicker than it would in a deterministic environment.

In section 3.3.2 a supervised learning algorithm using  $z_t$  as target was laid out. It is key to remember that the update rule from equation 4.5 is essentially a supervised learning regression

algorithm, but generalizing for  $G_{\lambda}(s_t)$  may be beneficial, as we can study the influence of the discount factor and reward function in this manner. To resume and summarize, the target values for the network are equal to

$$G_{\lambda}(s_t) = (1 - \lambda) \sum_{n=1}^{T-t} \lambda^{n-1} \left[ \sum_{i=0}^{n-1} (\gamma^i r_{k+i}) + \gamma^n V_{est}(s_{k+n}) \right] \quad (4.11)$$

in the standard  $TD(\lambda)$  method. The techniques from the following sections will only differ in how we define our target  $z_t = G_{\lambda}(s_t)$ .

Algorithm 8 is the supervised learning framework, independent of the definition of  $z_t$ , hence independent of the implemented TD algorithm and conditions. It consists of following noteworthy elements:

- random initialization of the network's parameters.
- playing episodes with the current network under some policy. Let us assume it is  $\epsilon$ -greedy for now.
- calculating the matching returns from all played games and adding the tuples  $(s_t, z_t)$  to replay memory  $\mathcal{D}$ . In algorithm 8 the replay memory gets refreshed every iteration, but more sophisticated methods to have a representable dataset may be applied.
- splitting  $\mathcal{D}$  in a training set  $\mathcal{D}_{training}$  and validation set  $\mathcal{D}_{test}$
- doing experience replay: random sampling of minibatches from  $\mathcal{D}$ , from which the weights will be updated with gradient descent.
- learn until the loss on the validation set stops decreasing, to avoid the network from overfitting.

If we were to implement  $TD(\lambda)$  onto the chess model, it would be hard for the agent to learn to play well. This is due to these main reasons:

- in contrast to backgammon, chess is a deterministic game, which will make exploration harder under a non adapted policy
- chess is a very tactical game. This fact requires some sort of tree search. With  $TD(\lambda)$  we are only looking one move ahead under an  $\epsilon$ -greedy policy.

The policy problem is somewhat hard to solve (we will look further into it in section 4.6), but we can introduce tree search in different fashions as we will see next.

### 4.3.2 TD-directed( $\lambda$ )

This is the easiest minimax variant proposed by the developers of *KnightCap* (Baxter et al., 1999). The same update rules and target are applied as in  $TD(\lambda)$ . The one difference lies in the policy. Play is guided by the minimax value at a depth  $d$ .

### 4.3.3 TD-root( $\lambda$ )

A first algorithm introducing a minimax search into TD-learning is  $TD\text{-root}(\lambda)$ , its idea already dates back to 1959(!), where the agent was able to reach an amateur level in checkers (Samuel,

**Algorithm 8** supervised TD algorithm

---

```

Initialize  $w$  (randomly)
repeat
  play  $N$  episodes  $\rightarrow (\mathcal{H}_1, \dots, \mathcal{H}_N)$ 
   $\mathcal{D} = \{\}$  // create a dataset to store experience from episodes
  for  $i \leftarrow 1$  to  $N$  do
    for  $s_t \in \mathcal{H}_i$  do
       $\mathcal{D} \leftarrow \mathcal{D} \cup \{(s_t, z_t)\}$  // calculate return
    end for
  end for
  shuffle samples
  select  $\mathcal{D}_{training}$  and  $\mathcal{D}_{test}$  from  $\mathcal{D}$ 
  repeat
    for all minibatches  $\in \mathcal{D}_{training}$  do
       $w \leftarrow w - \alpha \sum_{i=1}^n \frac{\nabla L_i(w)}{n}$ 
    end for
  until loss on  $\mathcal{D}_{test}$  stops decreasing
until satisfying convergence

```

---

1959). The idea is to update the value functions in positions into the direction of the minimax values, as they are better representation of the actual value at that state, as shown in figure 4.3. By doing this, the values at states will have the tree search incorporated into it. The value at the roots change into the direction of the values at the leafs of the principal variations.

To continue, we define the function  $l : \mathcal{S} \times \mathbb{N}_{>0} \mapsto \mathcal{S}$  that projects a state to its leaf state given a depth following the path of the principal variation. To incorporate this idea into TD learning, we only need to slightly modify equation 4.4 for a certain depth  $d$  to

$$\delta_t = V_{est}(l(s_{t+1}, d)) - V_{est}(s_t) \quad (4.12)$$

The update rule with the simplified conditions and TD-error from equation 4.12 is then equation 4.5. The corresponding target value for every state is consequently

$$z_t = \sum_{n=t}^{T-1} \lambda^{n-t} \delta_n + V_{est}(s_T) \quad (4.13)$$

#### 4.3.4 TD-leaf( $\lambda$ )

An issue with  $TD(\lambda)$  and  $TD-root(\lambda)$  is that they only update according the line of play, which correlates the data. By using the same minimax idea as before, but now updating the leaf along the principal variation (PV) to the leaf along the PV of the next position, we obtain data from positions not played. This is because the PV up until a certain depth is not necessarily the



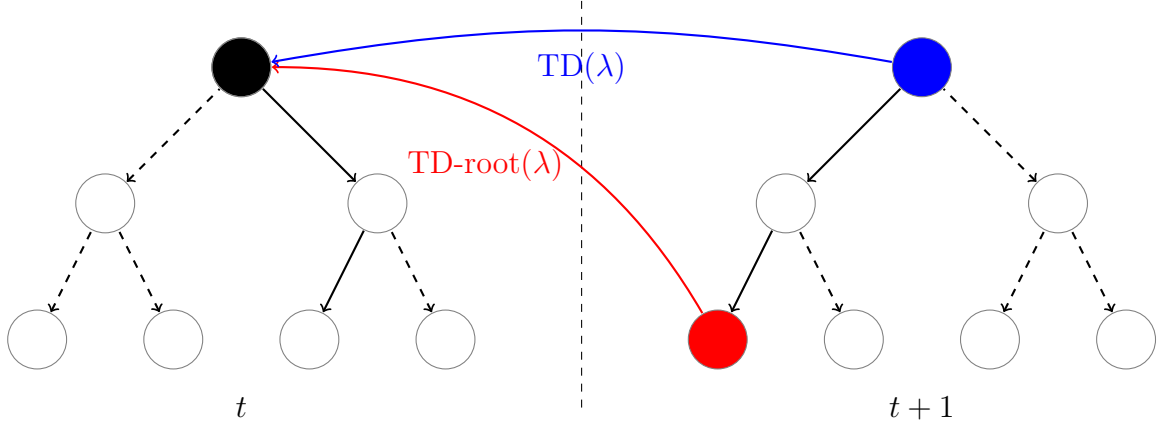


Figure 4.3: Comparison between  $\text{TD-root}(\lambda)$  and  $\text{TD}(\lambda)$ . In  $\text{TD}(\lambda)$ , the root nodes get feedback (an error signal) from the root at the next time step, while in  $\text{TD-root}(\lambda)$ , the root receives its feedback signal from the principal leaf node at the next state.

same path as in the following position. This idea is named  $\text{TD-leaf}(\lambda)$ , and was used to train against expert chess opponents in *KnightCap* and *Giraffe* (Baxter et al., 1999; Lai, 2015). Also with  $\text{TD-leaf}(\lambda)$ , a superhuman level in checkers has been achieved (Schaeffer et al., 1996). The update rule is depicted at figure 4.4.

The temporal difference and corresponding update rule defining this update are

$$\delta_t = V_{\text{est}}(l(s_{t+1}, d)) - V_{\text{est}}(l(s_t, d)) \quad (4.14)$$

$$w \leftarrow w - \alpha \sum_{t=0}^{T-1} \nabla_w V_{\text{est}}(l(s_t, d)) \sum_{n=t}^{T-1} \lambda^{n-t} \delta_n \quad (4.15)$$

Hence, our tuple for our supervised learning framework consists out of  $l(s_t)$  and

$$z_t = \sum_{n=t}^{T-1} \lambda^{n-t} \delta_n + V_{\text{est}}(l(s_T, d)) \quad (4.16)$$

which analogously as in section 4.3.1 can be transformed to

$$z_t = (1 - \lambda) \sum_{n=0}^{T-t} \lambda^{n-1} V_{\text{est}}(l(s_{t+n}, d)) + \lambda^{T-t} V_{\text{est}}(s_T) \quad (4.17)$$

$$= G_\lambda(s_t) \quad (4.18)$$

using the property that the leaf of a terminal state is itself, or put otherwise  $l(s_T) = s_T$ . With equation 4.18, we can define the  $\lambda$ -return for general discount rates and reward functions. The attentive reader noticed it was impossible to generalize the target to  $G_\lambda(s_t)$  for  $\text{TD-root}(\lambda)$ , because factors can not be canceled out when comparing leaves to roots.

### 4.3.5 TD-stem( $\lambda$ )

To conclude this section about TD algorithms, we will examine an own contribution (to the best of our knowledge):  $\text{TD-stem}(\lambda)$ . The idea is to simply combine the ideas of  $\text{TD-leaf}(\lambda)$  and  $\text{TD-root}(\lambda)$ .

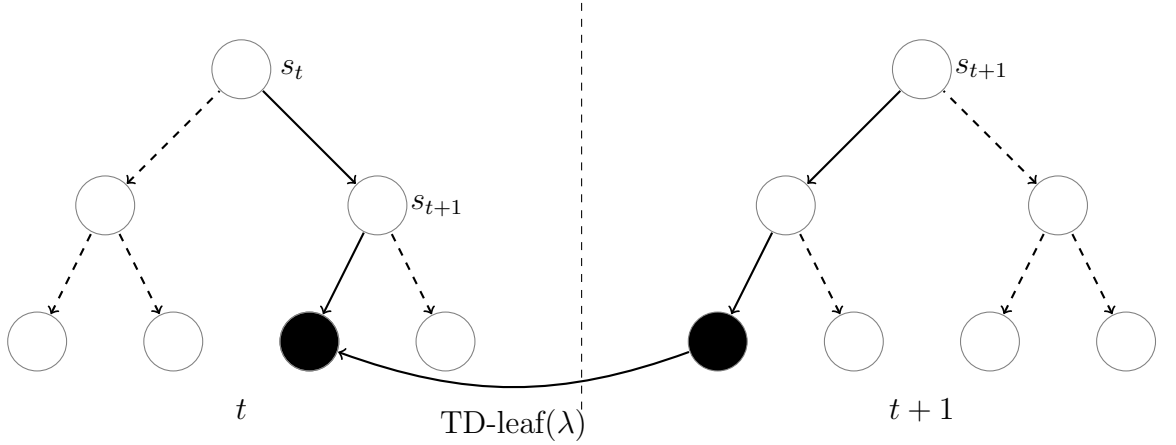


Figure 4.4: How updates in TD-leaf( $\lambda$ ) work. The PVs are denoted by the non dashed paths in the search tree. Notice how the paths may be different in the next state. The updates happen from leaf to leaf, which not necessary need to be played.

root( $\lambda$ ) by defining the error as the difference between the leaves, but updating the root state. This makes sense, as it seems to annihilate disadvantages of the former methods. In TD-leaf( $\lambda$ ) the update can be from another path (resulting in being erroneous), while TD-root( $\lambda$ ) can not be generalized to a  $\lambda$ -return. Essentially, the update is in the direction of the error between the minimax values of consecutive states. An additional bonus could be the incorporation of depth into the static evaluation function. However, this could also be a disadvantage, as this could make the evaluation more noisy. the main idea is depicted in 4.5.

To formalize this idea into mathematics:

$$\delta_t = V_{est}(l(s_{t+1}, d)) - V_{est}(l(s_t, d)) \quad (4.19)$$

$$w \leftarrow w - \alpha \sum_{t=0}^{T-1} \nabla_w V_{est}(s_t) \sum_{n=t}^{T-1} \lambda^{n-t} \delta_n \quad (4.20)$$

The generalization to  $\lambda$ -return is straightforward, with retrospect to equation 4.18.

## 4.4 Bootstrapping Tree Search

There are three drawbacks to the TD algorithms with search included discussed in previous section:

1. Only one node gets updated for every tree search, making the search somewhat inefficient.
2. The positions may not be representative or too coherent with each other, as they are all based on actual occurrences or computed best line of play (PV).
3. The target search is based on move played. Hence, this method is much more effective when the player is already strong.

Especially the third point is a major turn off in our point of view, as we want to inject as less expert domain knowledge as possible into our model.

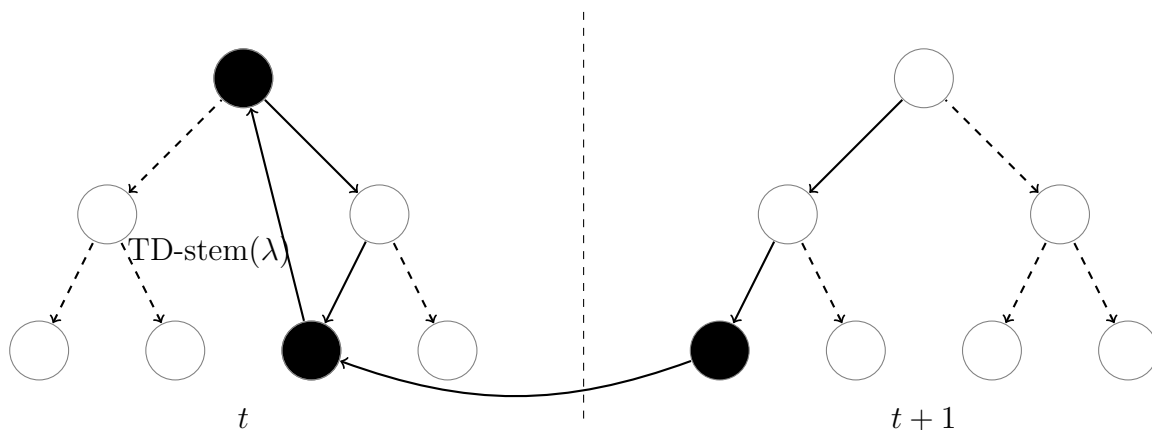


Figure 4.5: In  $\text{TD-stem}(\lambda)$ , the root is updated based on values of the consecutive leaf node values.

To address these problems, we could make more samples by using all information we got from the tree search. Additionally, we do not use the search from future, but rather use the minimax value directly as target. These ideas have been implemented in the chess engine *Meep*, reaching an expert rating with self play (Veness et al., 2009).

## 4.5 Monte Carlo Tree Search

We briefly look into a RL method, that has been proven to be widely successful, especially in Go, but from which the application to chess seems to be a hard nut to crack. It is an algorithm where researchers can show off their creativity by extending the basic methodology, which is why we should not deaden its potential contribution to chess. Because the *AlphaGo* paper is a huge influence for this master thesis, we will investigate its MCTS implementation a little more in detail.

The basic idea is simple, by building a search tree in an asymmetric manner we optimally try to balance out exploitation and exploration. The search tree is expanded along simulations, from which the final result is propagated back into earlier encountered search nodes. These final results have an impact on the future search.

The base algorithm consists out of four phases for each iteration as also illustrated in figure 4.6 (Browne and Powley, 2012):

1. **Selection.** starting from the root state, a tree policy is followed to branch through the tree until arriving at a (current) leaf node. One important characteristic about the tree policy is its exploration exploitation balance by embedding the state visit
2. **Expansion.** an additional node is added to the search tree with the tree policy
3. **Simulation.** from the expanded node on, a MC roll-out with a default policy is performed until arriving at a final state.
4. **Backpropagation.** the final result of the simulation is used to update statistics and values of the nodes in the search tree

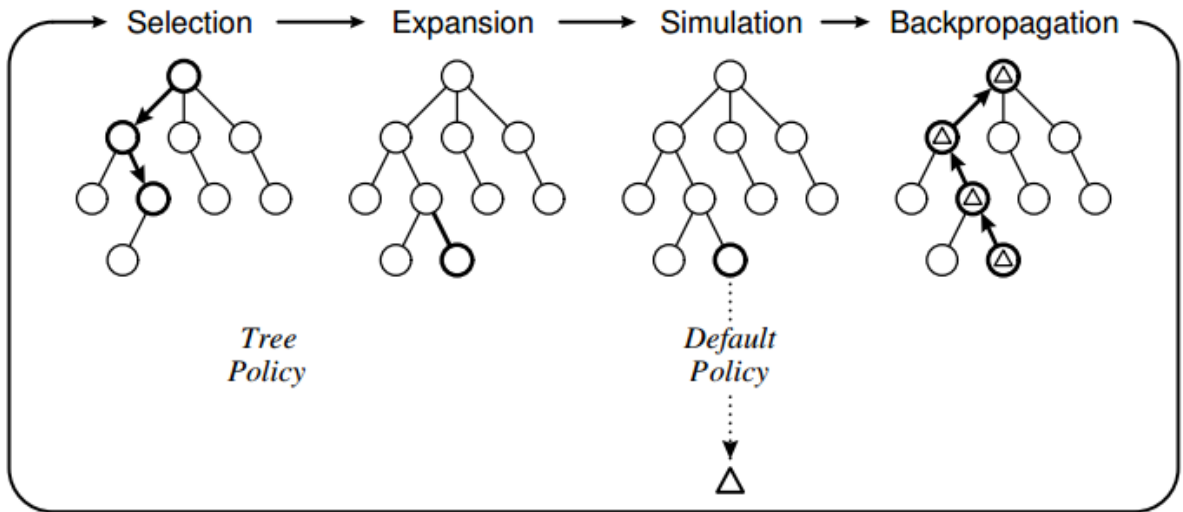


Figure 4.6: Illustration of how MCTS algorithms function.

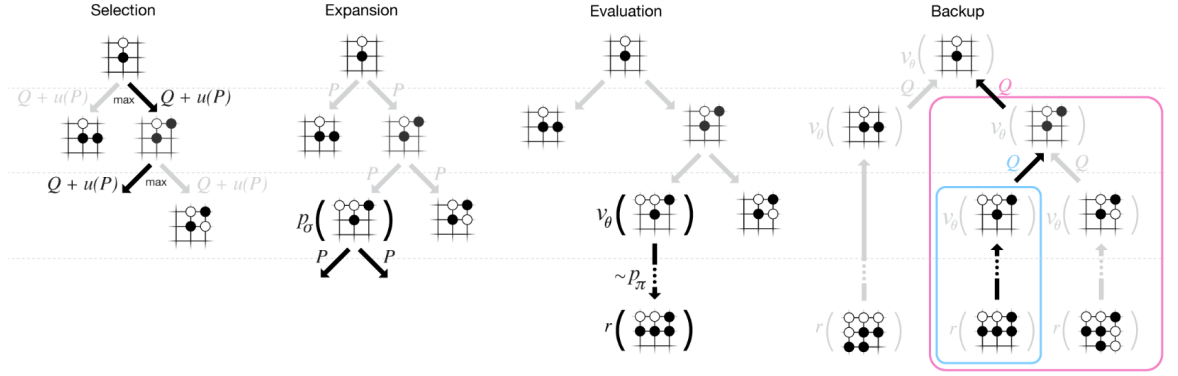
Typically, these phases are executed iteratively until a certain computational budget has been reached. At this point the final decision about the next action is returned.

The main things this method tries to achieve are to approximate the value of actions through random simulations and to use values efficiently for a best-first strategy. While the agent in TD algorithms tries to learn a long term value for each state, its purpose in MCTS is to estimate temporary state values to decide the play of the next move. In this respect, it is an algorithm competing with  $\alpha\beta$ -search for deciding the next move. In general, minimax is considered to be an unsuitable choice if no reliable heuristic exists. For large branch factors MCTS's ideas are also superior to minimax, as it is easier to gain depth. However, it is argued that MCTS performs poorly in tactical situations where the smallest mistake may be decisive (Ramanujan et al., 2011). These considerations lead to the insight that it may be hard for MCTS to improve chess play, especially in comparison with minimax. On the other hand, as we are trying to assume no additional domain specific knowledge, the simulation aspect of MCTS could help things out. It feels like more research into this topic needs to be conducted.

### 4.5.1 AlphaGo

Here, we discuss how a MCTS algorithm has been implemented in *AlphaGo*, as it may gain more insight how it could be beneficial to self play in a RL game environment (David Silver, 2016). The phases are constructed in the following way (figure 4.7):

1. **Selection.** Each edge contains statistics about the results obtained previously under the tree and roll-out policy. The stored values for an edge corresponding with the action  $a$  from state  $s$  are the action value  $Q(s, a)$ , the visit count (the number of iteration passing through that node)  $N(s, a)$  and a prior probability  $P(s, a)$  representing the policy learned with a policy network, see section 4.6. The tree policy for the selection phase is then

Figure 4.7: Illustration of the four phases of MCTS in *AlphaGo*.

defined to be

$$a_t = \operatorname{argmax}_a \{Q(s_t, a_t) + u(s_t, a_t)\} \quad (4.21)$$

where  $u(s_t, a_t) \propto \frac{P(s, a)}{1 + N(s, a)}$ , hence  $u(s, a)$  encourages exploration.

2. **Expansion.** From the leaf node on, the default policy (a faster version of the previous described network) is applied. If the child of the leaf has been visited a sufficient number of times, it will be added to search tree.
3. **Simulation.** A MC roll-out yields the final result  $r$  of the game under the default policy.
4. **Backpropagation.** The value that will be propagated back to through the edges of the tree is

$$V(s_L) = (1 - \mu)\hat{V}(s_L) + \mu r \quad (4.22)$$

where  $\hat{V}(s_L)$  is the estimation by feeding the leaf node through the neural network. All edges through the path of the search tree are updated according to the following rule:

$$N(s, a) \leftarrow N(s, a) + 1 \quad (4.23)$$

$$Q(s, a) \leftarrow (1 - \frac{1}{N(s, a)})Q(s, a) + \frac{1}{N(s, a)}V(s_L) \quad (4.24)$$

where equation 4.24 is a moving average over all encountered leaf values.

## 4.6 Policy Gradient Reinforcement Learning

Only approximating the value function has a big downside for deterministic games, namely that the smallest difference in value between two actions can cause a change in selection between moves (Sutton et al., 2000). A common theme in learning to play games is to iterate between learning a value function and policy iteratively, as discussed in section 3.3.4. For very large state spaces, which is the most common in game theory, this equals iterating between learning an approximation function  $V_w(s)$  and a probability distribution  $\pi_w(a|s)$ .

Just as a deep neural network can be used to learn  $V_w(s)$ , the same is true for  $\pi_w(a|s)$ , where the difference in architecture between both is a softmax layer at the end instead of a single value

learned with regression. Policy networks have proven their strengths in Go (David Silver, 2016).

Making policy networks a viable option for Go is the encoding of the actions, which is simply setting a stone at a certain square. Actions can thus easily be encoded with an output map. For chess, this encoding is much harder, as of all the legal actions possible in the game, at each state only a few are possible. An interesting research question is how policy networks can be incorporated into learning to play chess by finding decent encodings, especially for a CNN framework. A possible idea is the use of 2 output maps, one for the start position and one for the final position of a piece, hence encoding a uci move. This results in the encoding of  $64^2$  actions, from which most of them are not legal, especially from a certain state where the average breadth is around 20.

Another potential idea is recent research into Atari games, where a new CNN architecture has been proposed to estimate the optimal advantage function, defined to be

$$A(s, a) = Q(s, a) - V(s) \quad (4.25)$$

The dueling double deep Q-network (DDQN) learns  $Q(s, a)$  by separately learning the advantage and the value in the same network. By estimating the advantage function, we can significantly reduce the variance on the policy, which is very state dependent, hence this network could be a better choice than separating with an individual policy network. The dueling DDQN is the state of the art in playing Atari games (Wang et al., 2015).

## 4.7 Monte Carlo Value Initialization

All current RL techniques in chess use some sort of expert knowledge for the time being. *Knight-Cap* performed its training by playing humans and initializing its weights to piece value count, *Giraffe* initialized its network weights in the same way and used grandmaster positions to start with. *Meep* used handcrafted expert chess features. For the vision of this thesis, all these methods are forbidden, as they introduce some expert bias. At the same time, some initialization might be necessary to have a good point to start from and speed up training, as TD learning might take a very long time due to its nature to propagate back from the ending evaluation. The trick that made some of the previous chess engines quite successful was the initialization to a piece value count, as in this fashion, it is not necessary to branch out episodes up until the total end, the checkmate, which would be time consuming.

An own idea came to mind, trying to solve this issue. We will call the algorithm the Monte Carlo value initialization (MCVI), as it heavily relies on MC roll-outs. The idea is to set up random positions, and create a raw estimate of the value function based on the statistical results of the roll-out from that board position. As the statistics provide a value between 0 and 1, we transform it to be zero sum with the following equation:

$$V \leftarrow \frac{2w}{w+l} - 1 \quad (4.26)$$

which provides as a number between -1 and 1, hence mimicking the optimal value function. As positions with less pieces are on average objectively closer to a terminal state with optimal play and easier to finish, we start off the algorithm with less pieces. After this, we can continue for more pieces and play until a certain confidence is reached over the positions by evaluating them through the network, as the network already learned how to handle these with raw evaluations for less pieces. This (obviously) saves out time, as we do not have to execute the simulations until a terminal state has reached, and stop preemptively. A rudimentary algorithm is depicted in 9.

It is important to note how draws are not accounted in the calculations. We drop these simulations, as statistically speaking, by performing completely random actions without any peculiar policy, the relative difference between wins and losses would be minimized. The obtained values would not be a good indication about who got the better opportunities if play would be better.

---

**Algorithm 9** Monte Carlo Value Initialization

---

**Input:**  $P, M, N$

---

```

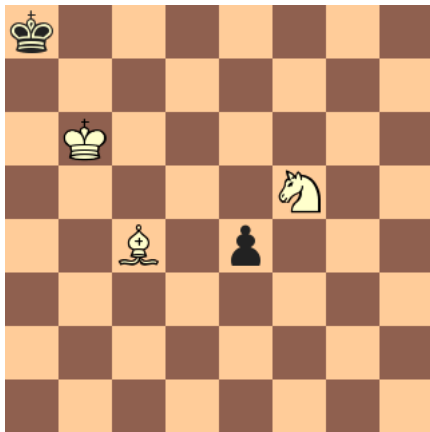
for  $i \leftarrow 1$  to  $P$  do
   $\mathcal{D} \leftarrow \{\}$ 
  for  $m \leftarrow 1$  to  $M$  do
     $B \leftarrow$  random position with  $P$  pieces
    for  $n \leftarrow 1$  to  $N$  do
       $w \leftarrow 0$ 
       $l \leftarrow 0$ 
       $r \leftarrow$  MC-rollout( $B$ ) // until certain confidence from end result is reached
      if  $r$  is a win then
         $w \leftarrow w + 1$ 
      else if  $r$  is a loss then
         $l \leftarrow l + 1$ 
      end if
    end for
     $V \leftarrow \frac{2w}{w+l} - 1$ 
     $\mathcal{D} \leftarrow \mathcal{D} \cup \{(B, V)\}$ 
  end for
  fit  $\mathcal{D}$  to network
end for

```

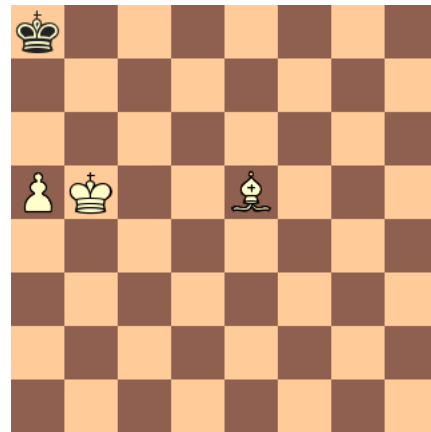
---

There are still some pitfalls that could ruin the joy:

- in some positions it is clear how play should continue, and how the side with the material advantage can win, but random play may benefit the other side. An example is illustrated in figure 4.8a.



(a) white is on the winning hand here, but black is more probable to win with random moves. The probability is sufficiently high that the black pawn will succeed at promoting to a queen, while setting the mating net is not straightforward to accomplice with the white pieces.



(b) To hold a draw, black can just move back and forward from a8 and b7. The problem is that black will make mistakes by playing randomly, and white will achieve an evaluation of 1 with equation 4.26.

Figure 4.8: Two examples leading to bad *MC* evaluations.

- Some positions are a draw, but random play may let the side that is the most probable to make a mistake do foolish actions, while the strategy maintaining the draw is straightforward (figure 4.8b).
- To yield a good first rough estimate, many simulations may still be needed for every  $P$ .

We may hope, that these strange positions are exceptions, and that the law of big numbers may weaken their influence.

To conclude this section, let us make a final remark about our vision in this document. We assume the agent should know nothing more than the rules at the beginning of the game. If we change this vision, to creating a chess engine that learns to play chess by only accepting objective facts, we could modify our MCVI algorithm in such a way to create a satisfying initialization with more certainty, by simply looking at the evaluation of tablebases (see section 2.3.2). If the position drawn from a MC roll-out seems to be in the tablebase, a correct result can be returned instantly, heavily improving the algorithm.



# Chapter 5

## Experiments

Every theoretical detail may look cool, but it has to show its worth in practice. In this chapter it is presented how experiments are conducted, from which the results are discussed further in chapter 6. To start off, we narrow down our experimental research from the bag of ideas discussed in previous chapter in section 5.1. The metrics to objectively (which is crucial to retain an unbiased system) evaluate our models are shown in section 5.2. Finally, section 5.3 gives an overview of the total system with its design choices. Furthermore, all hyper-parameters are summarized.

### 5.1 Goal

The holy grail is, as explained before, in this document the ability of machines to master the complete chess game through self play only, i.e. by gaining experience without a biased supervisor. This is however way too big for a master's thesis scope, which is why the constraint to relatively easy endgames was chosen. As we will see, this has the nice benefit it can be compared with optimal play. These models could be used to learn more complex chess positions as well from the bottom up.

A main difference between what we have discussed so far and other research is the use of bit-board features as input for a CNN. In the end it would be interesting to study whether this is a promising idea.

The current state of the art TD-learning algorithm for tactical board play is TD-Leaf( $\lambda$ ). We compare this algorithm with the in section 4.3.5 presented TD-Stem( $\lambda$ ).

### 5.2 Metrics

We show in this section how we can qualitatively assess and compare performances of learned models.

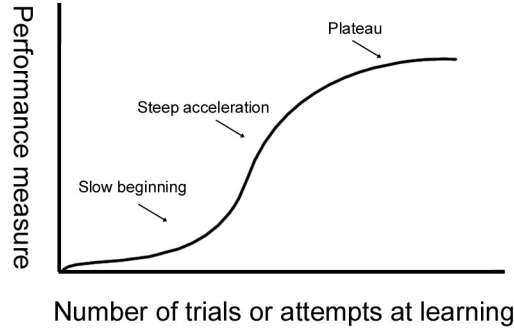


Figure 5.1: The typical form of a learning curve, which looks like a logistic function. A slow start is followed by a steep acceleration in learning after which a plateau is reached. Interestingly, this image comes from a psychology book (Dewey, 2007).

### 5.2.1 Learning Curves

A typical plot used for assessment of RL models is the learning curve. With these curves we can compare how fast agents are learning to optimize their rewards. The number of episodes  $N$  are plotted against the running average of cumulative reward per episode  $\mu_r$ . These plots typically look similar to the plot in figure 5.1.

In this thesis, some modifications of the classical learning curve with respect to the performance measure are used as alternatives. When we know for instance that most positions are a priori won, we can assess the performance of the model by checking the winning rate, which is

$$Q_w = \frac{\text{number of games won}}{N}$$

Like all averages, this can be tracked with a running averages.

Similarly, some positions should lead to checkmate faster and faster over the course of the learning process to prove that the agent learned to play more efficiently. Another performance measure is thus the average length. Formally, with  $L_i$  the length of episode  $i$ , this measure is represented as follows:

$$Q_l = \frac{1}{N} \sum_{i=0}^N L_i$$

### 5.2.2 Optimal Opponent Evaluation

Suppose for the sake of argument that we have an optimal player  $O$ , always making the right move. As an additional bonus, the final result is a priori known from the initial state  $s$  of an episode  $i$  taking optimal play from both players into account. We call this quantity the win draw loss (WDL). From the perspective of a player  $A$

$$R_{wdl,A}(s) = \begin{cases} 1 & \text{theoretical win } A \\ 0 & \text{theoretical draw} \\ -1 & \text{theoretical loss } A \end{cases}$$

If we were to have these optimal players, the depth to mate is available as well. We call this quantity  $L_{dtm} : \mathcal{S} \mapsto \mathbb{N}_{\geq 0}$  and it is the length of a game between optimal players.

Suppose we have designed a player  $A$  using a model learned through self play on positions where we know the theoretical outcome and DTM. If we let it play at its full strength some amount of games against an optimal player, we subsequently encounter  $N$  states with full a priori knowledge. By storing the statistics about every final result  $R_A(s)$  and game length (in plies)  $L_A(s)$  associated with the states, some useful strength indications of  $A$  can be derived. To make notations easier:

$$\begin{aligned} W &= \{s | R_{wdl,A}(s) = 1\} \\ W_A &= \{s | R_A(s) = 1\} \\ D &= \{s | R_{wdl,A}(s) = 0\} \\ D_A &= \{s | s \in D \wedge R_A(s) = 0\} \end{aligned}$$

- The **win conversion rate (WCR)** indicates how frequent the player is able to finish off a won position against a strong opponent.

$$Q_{wcr,A} = \frac{|W_A|}{|W|}$$

- With the **win efficiency (WE)**, we learn how optimal the player converts the won position to checkmate.

$$Q_{we,A} = \frac{1}{|W_A|} \sum_{s \in W_A} \frac{L_{dtm}(s)}{L_A(s)}$$

It is easy to see how  $0 \leq Q_{we,A} \leq 1$ , as the optimal length of a won game is always smaller or equal than the actual observed length.

- The **draw conversion rate (DCR)** is very similar to the WCR, and quantifies the ability of  $A$  to hold a draw against an optimal opponent.

$$Q_{dcr,A} = \frac{|D_A|}{|D|}$$

- The **loss holding score (LHS)** is a signification of how strong  $A$  defends lost positions. If the optimal player had to use the maximum of  $L_{dtm}$  plies,  $A$  behaved optimally.

$$Q_{lhs,A} = \frac{1}{|(D \cup W)^c|} \sum_{s \in (D \cup W)^c} \frac{L_A(s)}{L_{dtm}(s)}$$

Where  $(D \cup W)^c$  is the complement of all drawn and won positions, hence, all lost positions. Equivalent to WE,  $0 \leq Q_{lhs,A} \leq 1$  and the equality with 1 holds under optimal play from  $A$ . This measure is the fraction of the true game duration to the duration under optimal play.

With this bag of metrics, every game disregarding the theoretical result can be used to assess the quality of a model.

At first sight, these metrics may seem useless as chess has not been solved, and no 'optimal' player exists. However, recall the discussion in section 2.3.2 where we acknowledged the existence of tablebases yielding the exact information we need to use the performance measures defined here above for chess positions up to 7 pieces. Taking the gargantuan amount of storage into account for the 7 piece tablebase, we will restrict ourselves to 5 pieces. Although the size of this set of files is still 7 GB, it remains manageable.

These metrics are far from perfect, as they depend largely on the states provided. Some positions have much simpler paths to checkmate, and others are for example a certain draw where the player tested still goes for a win (which he would never achieve against a perfect opponent). We conclude this section by proclaiming the presented performance measure are more suitable for method comparison instead of real strength indicators.

### 5.2.3 Value Curves

Considering the closer we get to mate, the better the position is for the winning side, we would expect the optimal value function  $V^*(s)$  to be inversely proportional to the DTM. From the winning player's perspective, he should always be able to improve his position. The player can only behave optimal with an  $\alpha\beta$ -search if the DTM in the best leaf node (chosen upon the static evaluation) at the chosen depth of all states is indeed smaller than the root node. Hence, plotting the value function with respect to DTM information is an indication of how much has been learned during self play.

### 5.2.4 Speed

Due to the high complexity of the problem and its associated algorithms, a thoughtful implementation with respect to the computational cost is essential. The entire algorithm relies on two major components where speed can be measured in best interest: the move simulation in episodes and fitting the network to the data generated by this. The first will be measured in moves per second (MPS) while the latter is simply represented in seconds. Speedups are measured as the division of the duration in seconds of the old method by the duration of the new method.

## 5.3 Experimental Setup

With all that has been proposed yet, it is clear that this is not a trivial system to design, as there are lot of potential algorithms and hyper-parameters that can be tuned. It was necessary to fix a lot of these choices. As this is research, most solutions remain simple (Occam's razor<sup>1</sup>), if someone were to continue research from here on, more complex systems can be designed.

---

<sup>1</sup>If there exist two explanations, the simpler one is usually better

Basically, it works as follows. A script with modifiable parameters is manually started. A fixed number of iterations are ran where the model is first playing against itself. When the model has played enough episodes and has gathered the necessary data, the learning algorithm is carried out onto the value network until the system starts to overfit.

This section contains an overview of a more detailed setup of the experiments.

### 5.3.1 Search

The search algorithm implemented is a combination of 2  $\alpha\beta$ -pruning methods. First, an  $\alpha\beta$ -search without further optimizations is performed on the outcome of the game like in algorithm 10, looking for a forced win. If this indeed returns a forced win, the next stage is omitted. Else, a secondary search is required yielding the minimax evaluation with algorithm 2 with two optimizations:

- the use of a transposition table. Although the speedup is minimal, it is consistent (1.03<sup>2</sup>), it was left in the main implementation of the search algorithm. The corresponding hash function is the default one of the programming language on a string representation of the chessboard. A transposition table with Zobrist hashing has been attempted well, but this proved to be less performant (speedup of 0.77, hence slower) (Pătraşcu and Thorup, 2012).
- stopping recursive calls when the depth left is equal to 1. All possible states one ply ahead are fed immediately in one batch into the network to obtain the static evaluation. Vectorizing input speeds up the minimax evaluation, because most software libraries (like LAPACK) are optimized for that. Although less branches (with only one value left to calculate) are cut off the speedup is significant (2.24).

The first search can be performed at a higher depth than the second, because it is computationally some orders of magnitude less expensive for leave states to check whether the chess position is checkmate than to call the value network for an evaluation.

### 5.3.2 Evaluation

#### Feature Space

The input of our vectorization function in the experiments is a string representation of the board, named extended position description (EPD). From this the bitboards are extracted like described in figure 4.1 are extracted. The only global features that were added are the side to move, whether the position is a draw and whether it is mate. It follows that our feature vector consists out of 1539 binary elements. Undoubtedly, better state representations using more bitboards and global features are possible as presented in section 4.1, but the vectorization function

---

<sup>2</sup>All speedups mentioned in this section were measured by playing 100 games with a value network as described in section 5.3.2 and measuring the average time every method needed for calculating a move. Speedup is measured as  $t_{old}/t_{new}$

---

**Algorithm 10**  $\alpha\beta$ -Result

---

**Input:** board, depth,  $\alpha$ ,  $\beta$ **Output:**  $\alpha$ 

```

if board is checkmate then
  return -1
else if depthLeft  $\leftarrow$  0 then
  return 0
else
  for move in LegalMoves(board) do
    newboard  $\leftarrow$  doMove(board, move)
    score  $\leftarrow -\frac{\alpha\beta\text{Result}(\text{newboard}, \text{depthLeft}-1, -\beta, -\alpha)}{2}$  // faster mates are better
    if score  $\geq \beta$  then
      return  $\beta$ 
    end if
    if score  $> \alpha$  then
       $\alpha \leftarrow$  score
    end if
  end for
end if

```

---

proved to be a performance bottleneck. To be able to carry out more simulations the decision was taken to restrict to this basis input vector, considering it already contains almost all game play information.

Simulations have been executed on chess problems with a limited set of possible pieces on the board like the king rook king (krk) and king queen king (kqk) end games depicted in 5.2. These require less channels as the state is less condensed with pieces.

### Network architecture

There has been experimented with different CNN architectures using the bitboards as input. The activation function between all hidden units is an ordinary ReLU and the weights are initialized with the tips given by Bengio et al. (Bengio, 2012).

#### 5.3.3 Self Play

The self play framework we employ is one with multiple agents, both black and white try to maximize their rewards. We have not experimented with the systems using one learning agent playing with the most up to date model against an older model (David Silver, 2016; Wang et al., 2015). The reason it has not been tested is the additional implementation cost to address two

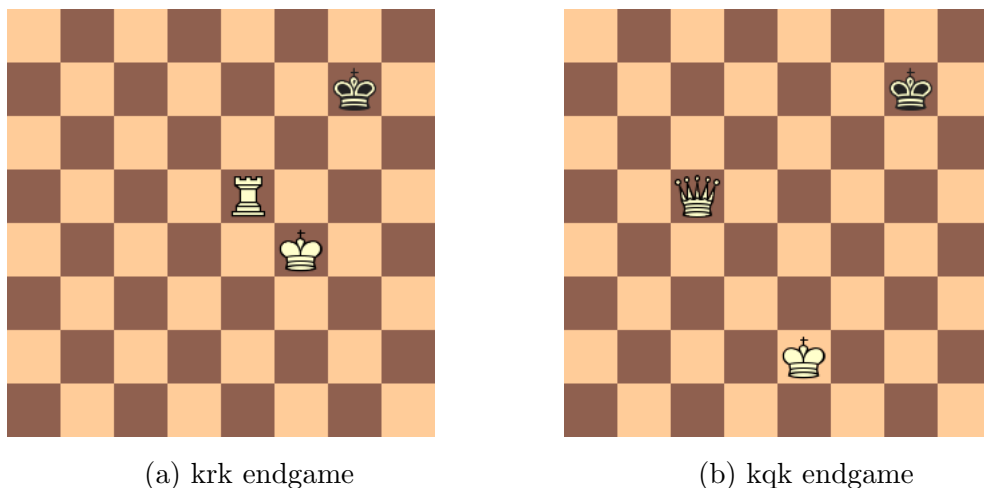


Figure 5.2: Subproblems where only two kinds of pieces are used. The dimension size of the input reduces to 515.

networks, hence two GPUs, in combination with time constraints.

In self play, episodes are set up and the model plays them up to a certain point. During self play, data is collected. This data is then used after some amount of episodes so it can be used to update the value network.

### Setup episode

Initial positions can be generated by random placing the pieces on the board and achieving a legal position. However, This method becomes more complex when more pieces need to be placed on the board. This is why in most experiments, in light of future research, the starting state is generated through a large dataset containing computer games (CC, 2017b). The large amount of EPDs that were generated through this were subsequently filtered according to the problems looked for. For example, an additional dataset only containing krk positions was generated to solve the krk problem. It is made sure that all positions in the dataset are unique. There are still two main issues with naively random sampling positions and using them as initial state:

1. Due to the fact all positions originate from games, they are all part of a sequence. States from identical games are correlated to each other, especially when they are only a few moves away. This correlation may induce overfitting in the learning process.
2. We cannot validate our positions with the exact same dataset, as this would not prove that the learned approximation is an acceptable generalization.

These problems are mitigated by making random moves before self play. An increasing amount of random moves decorrelates the input. However, the position may evolve too much by capturing pieces and pushing pawns. There was not enough time to limit the random moves to non captures and non pawn moves.

Aside from this additional randomization, we might split our datasets in a training, used for self play, and validation set to solve the second shortcoming.

## Policy

The policy of the agents during simulation has been restricted to  $\epsilon$ -greedy. As noted in section 3.3.3, this works fine and for the scope of this thesis, and it is not necessary to try more elaborated policies. To try to speed up training though, there has been some examinations of decay functions. Two different decay functions are deployed, suppose  $i$  is a integer linearly proportional to the number of simulated episodes, and  $f$  is an additional parameter:

1. Linearly adjusting the decay:  $\epsilon_i \leftarrow 1 - fi$
2. To adjust for a slow start, we can begin by using a power  $0.5 < f < 1$  to reduce  $\epsilon$  with hyperbolic decay:  $\epsilon_i \leftarrow \frac{1}{(i+1)^f}$

The greedy action is taken in the direction of the minimax search described in 5.3.1, as in classical engines. This is the same for both examined TD( $\lambda$ ) algorithms. The depth of search is for our purposes limited to 3, to guarantee fast simulations.

## Episode Termination

The game is continued until a terminal state is reached, or a maximum number of actions has been performed, after which the game and its data are stored in replay history and a new episode is generated in the same fashion.

## Self Play Termination

An iteration of self play goes on until enough games have been recorded with notable rewards (i.e. episodes ending with checkmate). The rationale behind this is that at first, most games end undecided (because random play increases the probability of exceeding the maximum number of allowed moves).

## Learning Algorithm

When a self play iteration has been completed, we perform the following steps to improve our network with mini-batch SGD:

1. **Data assimilation.** This phase is simply collecting all episodic data (featurized states and rewards). Every episode has to be examined in detail independently in the following steps.
2. **Calculation target values.** The  $\lambda$ -returns are calculated with TD-Step( $\lambda$ ) or TD-Leaf( $\lambda$ ). The discount rate is fixed to  $\gamma = 1$ , but the implementation is flexible enough to handle other values. In this manner, TD-Leaf( $\lambda$ ) as implemented in the system is equivalent to previous research.



3. **mini-batch SGD.** At the end of every iteration, the network gets an update with mini-batch SGD.

Furthermore, we have added some optimizations to this process based on observations when testing the implementation:

- When learning has just started and the agent is mostly in an exploring mood, only the last values of the episodes are kept, as the start of the episode is mostly noise. By throwing this noise away, we accelerate initial training.
- Memory problems show up (a very bad performance) when keeping all returns. Hence, it is ensured that training data does not exceed a maximum size (50000 samples). At the same time, every episode has the same contribution to this dataset. The training samples used for supervised learning later in the process are moved to a buffer object, also storing tuples from previous value network update iterations, to decorrelate the samples and thus reduce overfitting.
- The data to update the network is picked randomly from the buffer object. Samples recorded during more recent play have a higher probability of being picked. Depending on the network, a different learning rate is used. Deeper networks have the tendency of needing a smaller learning rate.

## Experimental Methodology

By running a script, the system is set up. An initialized value network learned from a previous run can be given as a parameter. These runs are ad hoc, and the decision for the parameters can change in between runs. This way of working increases the flexibility of adjustments. Every separate run from a script will be called a stage from now on. It speaks for itself that stages with the same parameters must be ran to obtain useful comparisons. The parameters that were adjusted in between the stages are laid out in table 5.1.

In general, the exponential weight decay parameter  $\lambda$  from TD learning increases over the stages, because the more training evolves, the higher our confidence in assigning credit to moves further away from checkmate. Similarly, the number of moves used to calculate the  $\lambda$ -return increases along the script. It also makes more sense to start off with a small depth, and increase it along the way for a faster decent initialization of the network. Starting off with deep searches seems like overkill and a waste of computation power.

### 5.3.4 Implementation

All experiments were carried out on GPU servers. To be specific everything was tested on either a NVIDIA GeForce GTX 680 4GB with an Intel(R) Core(TM) i7 CPU 930 @2.80GHz (6 CPUs) or NVIDIA Tesla K40c 12GB with an Intel(R) Core(TM) i7-3930K CPU @ 3.20GHz. We will call the former machine A and the latter machine B.

Table 5.1: Hyper-parameters modifiable in between stages.

---

$\lambda$	trace decay parameter for TD-learning methods
$d_r$	Depth at which final results are examined in search
$d_V$	Depth at which the value network is called
$f_\epsilon$	Decay function for exploration parameter $\epsilon = f(i)$ . i increments every iteration.
$I$	Number of iterations
$i_0$	First iteration number, to initialize $\epsilon$ with the $f_\epsilon$
$K$	The number of states that are used to calculate the $\lambda$ -return from an episode
$M$	The maximal amount of moves made in an episode
$N$	How many games are played during an iteration
$R$	The number of additional random moves played on the board position extracted from the dataset

---

All code is written in Python v2.7, with a chess engine written in C. At first, all chess specific knowledge was extracted through the Python chess library, but it became clear that performance could be enhanced enormously by using native C code (the Python chess library is still used at some parts, but rarely) (Fiekas, 2017). The neural networks are implemented with Tensorflow, an open source software library for machine intelligence.

All implementations were serialized at first, but the necessity to parallelize the code is obvious. A speedup almost linear proportional to the number of CPUs in the machine is achieved by concurrently simulating the episodes. Quite some work was put into the synchronization of all components with the Python multiprocessing module, as the network has to calculate values for all episodes. After playing  $N$  episodes, the network has to be called again, now to fit to the data.

A high level architecture of the written software <sup>3</sup> is shown in figure 5.3. The *supervisor* is created in a main process (there can only be as many processes as CPUs in the Python design) and controls all work. Before simulating episodes the supervisor creates a new process responsible for the value network, whose calculations are performed on the GPU. thereafter the following steps are executed iteratively:

1. The supervisor creates as many processes as there are CPUs available and commands them to perform  $N$  episodes in total. Next, the supervisor goes to sleep and waits until these jobs are finished.
2. The agents need to call the value network, but the GPU resource can only be taken once at a time, which is why all requests are stalled in a FIFO queue if necessary
3. Whenever an episode terminates, it dumps its recorded data in a buffer. As all episodic

---

<sup>3</sup>open source (Van den Heede, 2017)

processes are trying to drop data, this is handled in a queue as well.

4. When all  $N$  episodes have terminated, the supervisor wakes up and passes data from the buffer to the value network again with the purpose to improve the value function with SGD.
5. After fitting, these steps repeat themselves until the desired convergence has been attained.

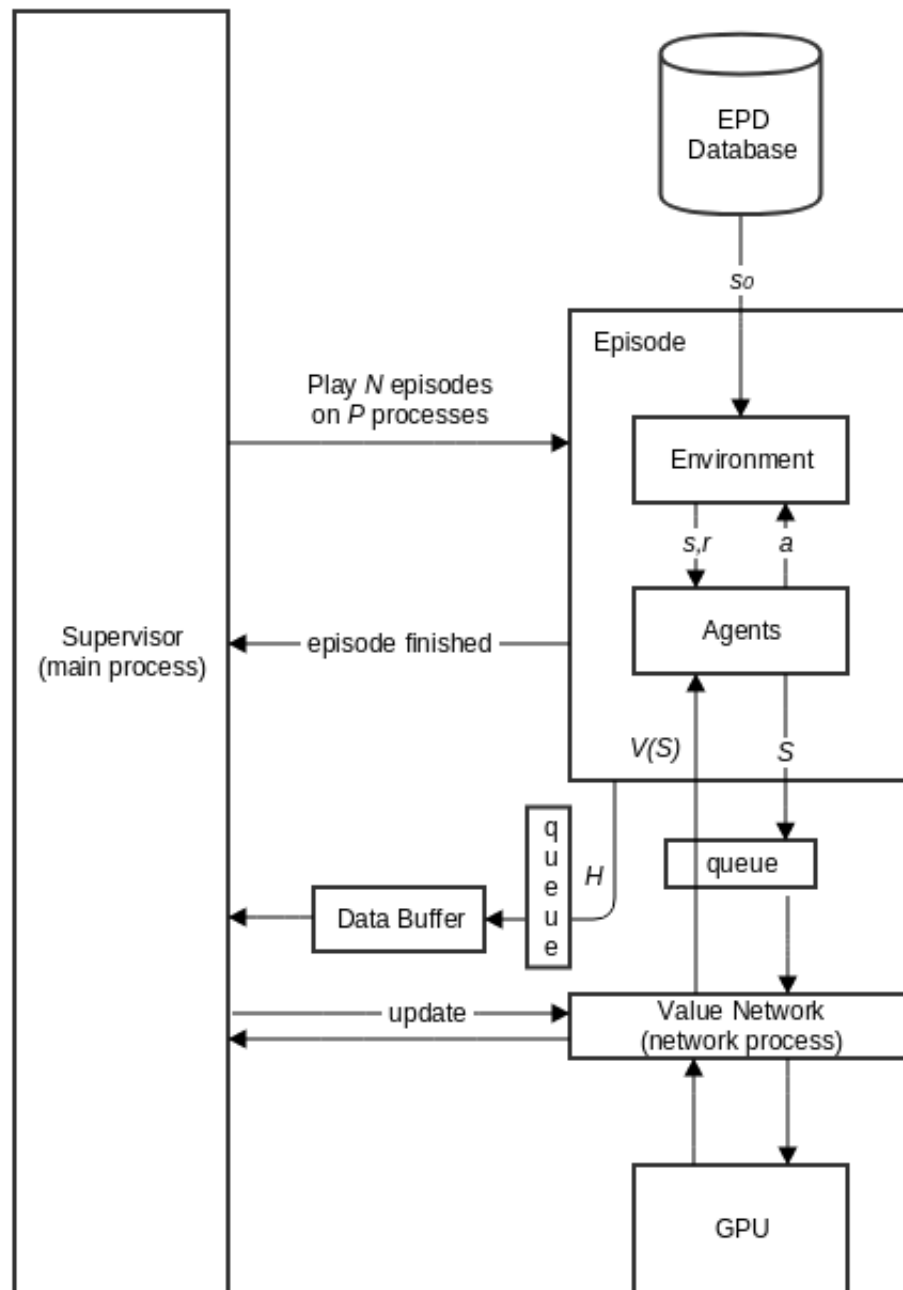


Figure 5.3: High level software architecture of the simulation system.

# Chapter 6

## Results

In this chapter, we examine the strength of our proposed algorithms and compare TD-Stem( $\lambda$ ) with TD-Leaf( $\lambda$ ) in specific. We do this by considering end games with no more than 3 pieces on the board. By limiting ourselves to these kinds of problems we can perform objective comparisons with the metrics discussed in previous chapter. The first experiment in section 6.1 deals with a krk endgame. An additional experiment with different hyper-parameters is carried out in section 6.2. Finally, a more generalized piece setup is used for the simulations discussed in 6.3.

### 6.1 Experiment 1: king rook king with uniform sampling

The first simulations we ran were all on the krk endgame, where the obvious goal for the player holding the rook is to checkmate the opponent as efficiently as possible. To facilitate the process, we started by only giving the white side the rook. If the game ends in a draw, white receives a negative reward:

$$r(s_t, a_t) = \begin{cases} 10 & \text{if } s_{t+1} \text{ is checkmate} \\ -10 & \text{if } s_{t+1} \text{ is draw} \\ 0 & \text{else} \end{cases}$$

The initial starting positions in this simulation were set by randomly placing the kings and rook on the board under the constraint of being a legal position, ensuring to be totally unbiased. The value network did not contain any convolutional layers yet, only a fully connected layer with 128 hidden units as shown in figure 6.1.

We ran the simulation script on machine A (section 5.3.4) over 7 stages identical for both TD-Leaf( $\lambda$ ) and TD-Stem( $\lambda$ ) with a learning rate  $\alpha = 10^{-4}$ . The hyper parameters during the stages are laid out in table 6.1. Furthermore,  $M = 50$  and  $f_\epsilon(i) = (i)^{-3/4}$ . Since samples are uniformly distributed,  $R = 0$ .  $K$  and  $d_r$  were not implemented yet.

The resulting learning curve from figure 6.2 gives the impression that our variant outperforms

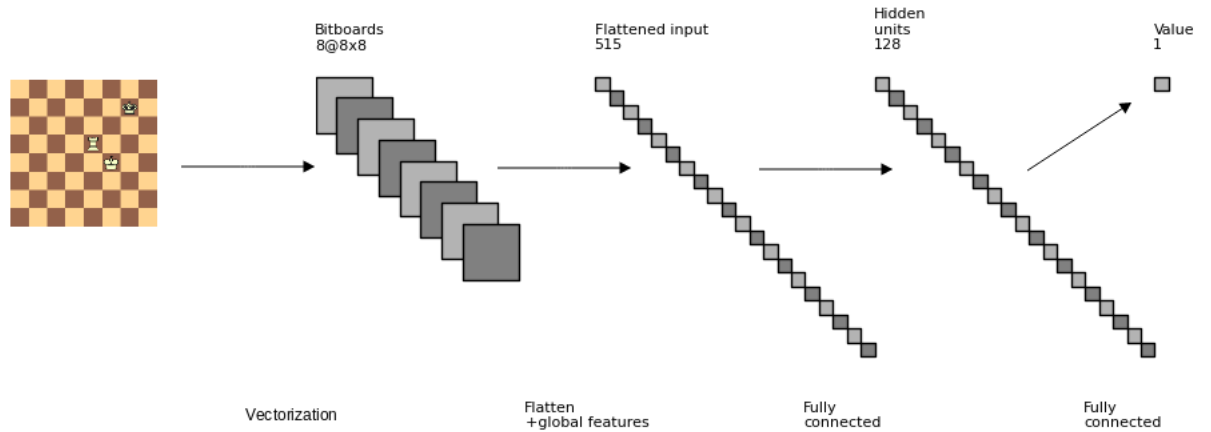


Figure 6.1: Depiction of the NN used in the first experiment. The binary input features are fed into a 1 layer FNN.

the state of the art TD-learning algorithm. One could argue that the 2nd and 3th stage have not induced additional learning. We also notice how increasing the depth boosts training. We prove this to be true by letting a greedy agent perform 2000 episodes with both methods against a perfect player using the krk tablebase as reference. With the value function learned in TD-Stem( $\lambda$ ), we succeed at winning about 85% of the positions, clearly outperforming TD-Leaf( $\lambda$ ). When winning, both techniques seem equally good. When the engine is at the losing side, it defends itself better with our variant. The reason why TD-Leaf( $\lambda$ ) evaluates moves faster during self play, is because it has more difficulties finding wins (remember from section 5.3.3 how the iteration continues until enough rewards have been collected).

Let us observe the bar plots (figure 6.3) in function of the theoretical DTM to study which positions the generated models can solve. Aside of showing the superiority of TD-Stem( $\lambda$ ) in this experiment, it indicates how TD-Learning propagates information further and further from the terminal states where the rewards are awarded. Training may have stopped somewhat early, as the winning rate of the model was still improving.

If we investigate the value function in figure 6.4, we understand why the model does not always succeed at making progress and approaching checkmate. Notice how the variance increases the further we go from a winning terminal state, indicating that although the average value function contains the right trend, we have not found enough solutions yet for these though positions. We also observe that especially close to checkmate, the variance in TD-stem( $\lambda$ ) is way smaller than in TD-leaf( $\lambda$ ).

Table 6.1: Hyper-parameters of the stages in Experiment 1

Stage	$N$	$I$	$d_V$	$\lambda$	$i_0$
1	5000	20	1	0.5	1
2	5000	20	1	0.5	2
3	5000	20	1	0.7	2
4	500	20	3	0.8	2
5	250	30	3	0.8	2
6	250	30	3	0.8	2
7	250	50	3	0.8	2

Table 6.2: Performance comparison TD-Leaf( $\lambda$ ) and TD-Stem( $\lambda$ )

	TD-Leaf( $\lambda$ )	TD-Stem( $\lambda$ )
<b>WCR</b>	0.48	<b>0.85</b>
<b>WE</b>	<b>0.87</b>	0.86
<b>LHS</b>	0.80	<b>0.91</b>
<b>MPS</b>	<b>228</b>	205
<b>N</b>	353 500	<b>304 500</b>

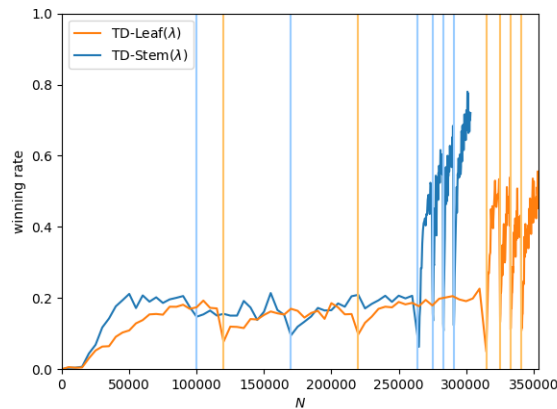
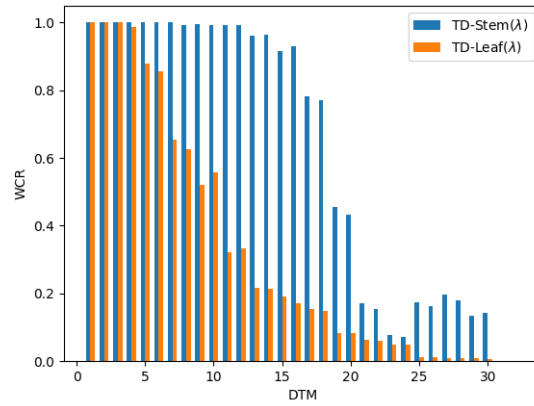
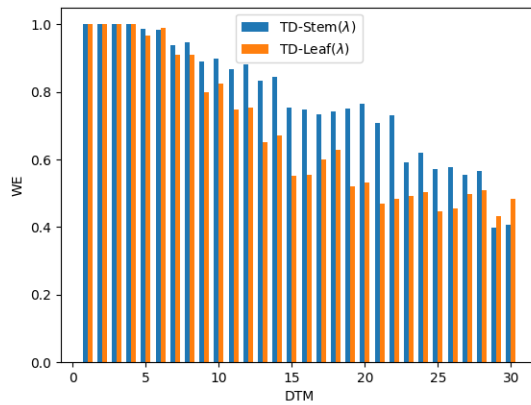


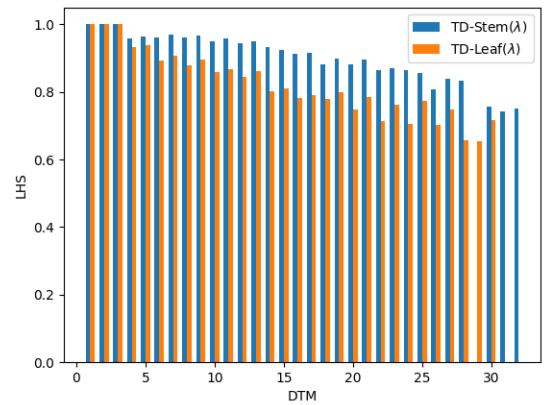
Figure 6.2: The learning curve of the experiment. The vertical lines indicate when the stages start.



(a) win conversion rate krk endgame



(b) win efficiency krk endgame



(c) loss holding score krk endgame

Figure 6.3: Performance comparison between TD-Stem( $\lambda$ ) and TD-Leaf( $\lambda$ ) through the form of bar charts with respect to the theoretical DTM. These bar plots are obtained by playing a trained model 2000 positions from a validation set against an optimal player and storing all encountered positions together with the final outcome.



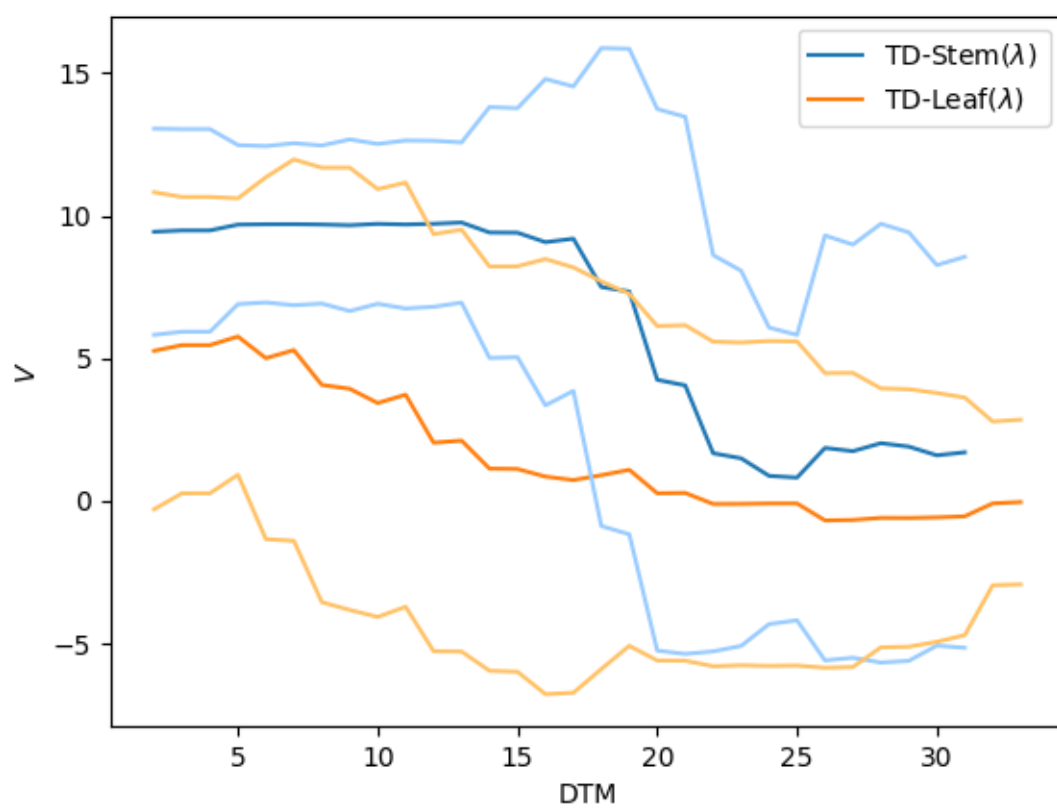


Figure 6.4: Value curve showing the average value estimate of over 15000 krk positions. To visualize the variance over these values, the 95% intervals are provided as well.

Table 6.3: Hyper-parameters of the stages in Experiment 2

Stage	$N$	$I$	$d_V$	$\lambda$	$i_0$	$K$
1	500	20	1	0.5	0	10
2	250	20	3	0.6	20	20
3	250	20	3	0.7	40	30
4	250	50	3	0.8	40	30
5	250	20	3	0.8	45	30

## 6.2 Experiment 2: king queen king with sampling from dataset

In this section, we try to confirm our conjecture of TD-Leaf( $\lambda$ ) being at least equivalent to TD-Stem( $\lambda$ ) in performance by analyzing an even easier problem, the kqk endgame. In contrast to experiment 1, both white and black can possess the queen. This trait makes the value function harder to learn, but is necessary as the goal should be to learn a symmetrical value function (remember the zero-sum property). To enforce this characteristic, data augmentation is performed as described in section 5.3.3.

Another main difference with experiment 1 is the sampling of initial positions, which happens by randomly picking boards from a dataset. In addition 5 random moves are made on these initial states with the goal of reaching an as diverse set of games as possible, which should encourage generalization by the learning algorithm.

To experiment with the idea of learning good bitboard representations, the value network is a CNN as designed in figure 6.5. The use of (1, 1) patches for the convolutional layer can be interpreted as an attempt to learn the most important squares with respect to the problem, given the piece locations and mobilities. The applied learning rate is  $\alpha = 10^{-6}$ . As in experiment 1, the maximal number of moves per episode is  $M = 50$ . However, we have changed the policy to a linear decay approach:  $f_\epsilon(i) = 1 - 0.02i$ . Changes in between the stages are represented in table 6.3. For this experiment, all simulations were ran on machine B (section 5.3.4).

A first glance at the learning curve in figure 6.6 can trick one in believing that TD-Leaf( $\lambda$ ) learns quicker than TD-Stem( $\lambda$ ). However, the sudden dive from the curve starting from episode 30000 corresponding with TD-Leaf( $\lambda$ ) hints to a contrary belief. An explanation for this behavior is that the winning rate in the plot represents the agent's strength against itself, and not against the optimal agent. Hence, the decrease is a sign of an improvement at the losing side. In retrospect, we are able to notice the exact same effect in experiment 1 in figure 6.2 around episode 330000. This idea is backed up by the fact that the strength of the models <sup>1</sup> was already analyzed after termination of the third stage. Performances of both models against the optimal player were so much worse than what the learning curve indicated, that two stages were added

<sup>1</sup>The metrics in experiment 2 have been measured in a similar fashion as experiment 1.

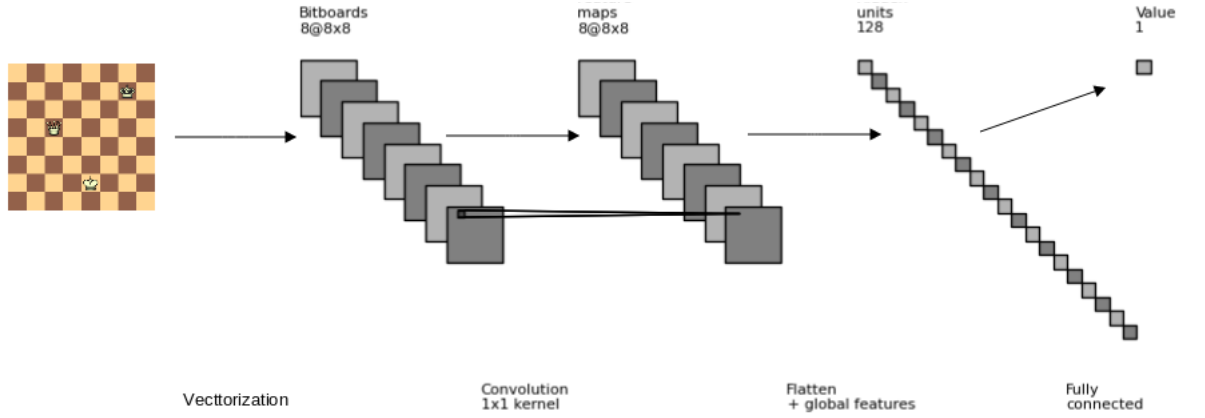


Figure 6.5: The (relatively simple) CNN architecture deployed in experiment 2.

Table 6.4: Performance comparison TD-Leaf( $\lambda$ ) and TD-Stem( $\lambda$ )

	3 stages		5 stages	
	TD-Leaf( $\lambda$ )	TD-Stem( $\lambda$ )	TD-Leaf( $\lambda$ )	TD-Stem( $\lambda$ )
WCR	0.65	<b>0.77</b>	0.90	0.90
WE	<b>0.67</b>	0.64	0.89	0.89
LHS	0.89	0.89	0.95	<b>0.97</b>
MPS	346	<b>359</b>	180	<b>188</b>
N	<b>26000</b>	27000	<b>43500</b>	44500

to the process. These result in a big improvement in play as shown in table 6.4. As TD-Stem( $\lambda$ ) does not appear to contain this subtle effect, we believe it to be probably faster in general, and more certainly it succeeds at improving the winning and losing side at the same time while TD-Leaf( $\lambda$ ) appears to improve the players one after another.

In table 6.4 and figure 6.7 it can be noticed how more equilibrated the performances of both models are in comparison to previous experiment. However, TD-Stem( $\lambda$ ) continues to outperform the state of the art with respect to the LHS, advocating in favor of our argument that TD-Stem( $\lambda$ ) improves both sides at the same time.

A possible explanation for this effect could be the following. If we review the mechanism behind TD-Leaf( $\lambda$ ) (section 4.3.4), we can notice how the final outcome of the game influences the update. However, we may be updating leaf nodes we have never encountered in the simulation. This is in contrast with TD-Stem( $\lambda$ ), where the game states are used as training samples. The insight that the states used for the network update provide a true evidence about their value (namely, the outcome of the played game) can be an explanation for this effect. On the contrary, TD-Leaf( $\lambda$ ) updates states seen in search, not necessarily in simulation, can result in updates in the direction of a 'wrong belief', as the final outcome participates to the equation (see section 4.3.4).

The value curves (figure 6.8) indicate how the variance is higher for increasing DTMs.

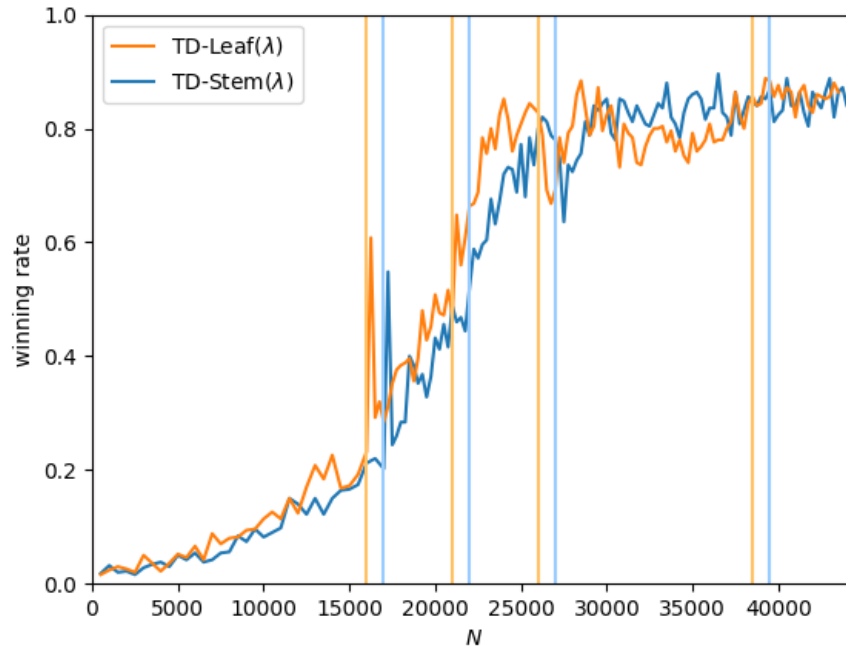


Figure 6.6: The learning curve of the second experiment, divided in 5 stages.

### 6.3 Experiment 3: 3 pieces endgames with sampling from dataset

In this experiment, we look at a much more complex problem, namely end games where every configuration is trained with 3 pieces on the board. Hence, the possible chess positions are:

- kqk
- krk
- king bishop king (kbk)
- king night king (knk)
- king pawn king (kpk)

From which knk and kbk are less relevant as they are categorized in terminal states (these positions are a draw due to insufficient material). As all piece types are seen in the dataset, the amount of bitboards in the input is now 24. As the problem seems much more complex, a deeper network was deployed as laid out in figure 6.9. One of the reasons this problem is much harder is that kpk endings can only be won after promotion. Additionally, many cases exist where it is impossible to force a pawn promotion when the opponent plays perfectly. The training setup was equivalent to the previous experiments, now with the hyper-parameters set as shown in table 6.5.

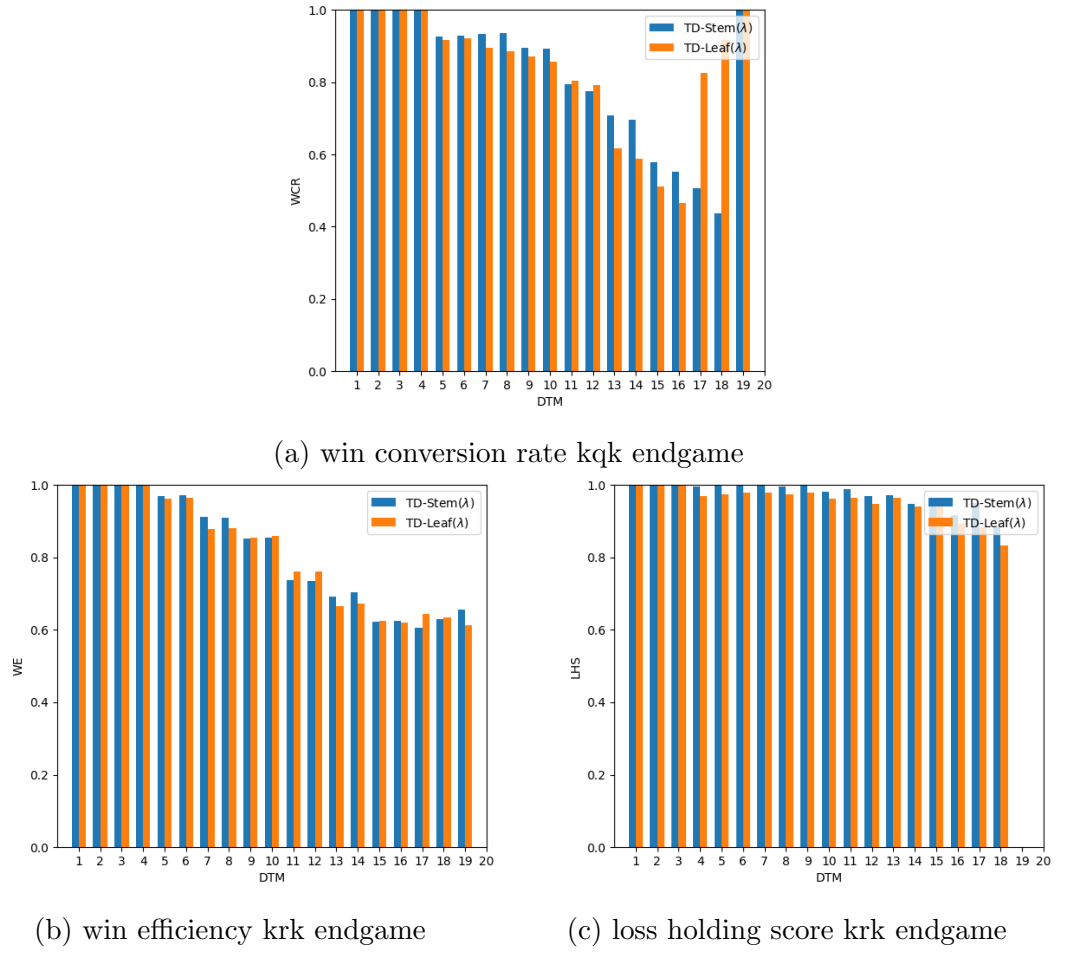


Figure 6.7: Performance comparison between TD-Stem( $\lambda$ ) and TD-Leaf( $\lambda$ ) through the form of bar charts with respect to the theoretical DTM.

Table 6.5: Hyper-parameters of the stages in Experiment 2

Stage	$N$	$I$	$d_V$	$\lambda$	$i_0$	$K$	$d_R$
1	250	100	1	0.5	0	100	3
2	250	100	3	0.7	50	100	5
3	250	60	3	0.7	95	100	5
4	250	100	3	0.8	95	100	5
5	250	100	3	0.8	95	100	5
6	250	80	3	0.7	95	100	5
7	250	100	3	0.7	95	100	5

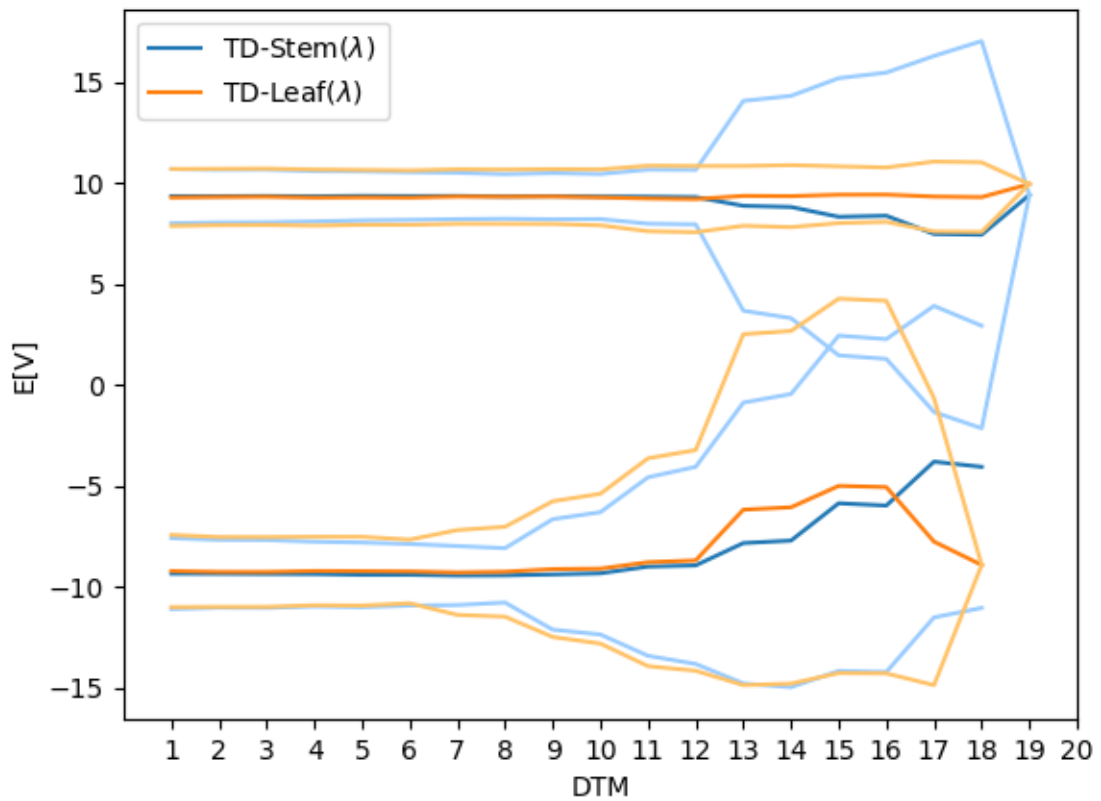


Figure 6.8: Value curves for positions where white ( $E[V] > 0$ ) and black ( $E[V] < 0$ ) hold the queen.

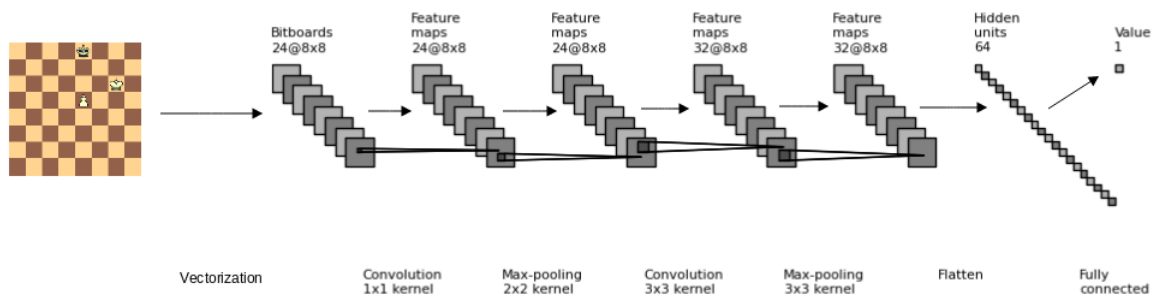


Figure 6.9: The (relatively simple) CNN architecture deployed in experiment 2.

Table 6.6: Performance of the trained model in experiment.

	3 stages		7 stages	
	TD-Leaf( $\lambda$ )	TD-Stem( $\lambda$ )	TD-Leaf( $\lambda$ )	TD-Stem( $\lambda$ )
WCR	<b>0.32</b>	0.30	0.30	0.30
WE	<b>0.86</b>	0.92	0.83	0.85
LHS	0.78	<b>0.80</b>	0.78	<b>0.79</b>
MPS	<b>39</b>	38	19	19
N	<b>133500</b>	135000	<b>183500</b>	186000

As one can see in table 6.6, the progress from stage 4 onwards is not satisfactory. Multiple reasons serve to explain this:

- One reason could be that more episodes should have been simulated before increasing the depth to  $d_V = 3$ . Additionally, the fact that many data have been thrown away due to the slow simulations when too much samples were used as training set could have had an influence. One of the powers of TD learning is the correlation between successive samples, and we might have decorrelated too much for this problem. It is also known that deep networks only have the tendency to generalize well when enough data have been seen.
- Another reason could be that the error surface is less smooth due to the more varied piece setups, which results in getting stuck in local minimums.
- Finally, it could be that the trained model tends to improve when playing more episodes. Possibly by generating games with a higher search depth. The problem is that simulating episodes is already slow when  $d_V = 3$ .

It should be studied further how these board setups could be learned through self play only without additional bias. Our opinion is that throwing less data away, albeit having much slower experiments, is the way to go. Unfortunately, there was insufficient time left at this point in the master thesis to do further research in this direction.

## 6.4 Conclusion

By doing these experiments we have learned several things. First of all, it seems that we have found a worthy and at least equivalent method for the state of the art TD learning algorithm in TD-Stem( $\lambda$ ). We discussed two possible explanations why our variant may be an improvement over TD-Leaf( $\lambda$ ): depth is included in the estimated value function and all effective outcomes correspond to an encountered board position. Secondly, in the experiments where we fixed the piece configuration to krk and kqk, the final models were able to solve the problems up until a decent level purely through self play and supervised learning. Lastly, we found that more complex variations where more configurations should be learned all at once are a bigger deal. This may be solved by using more training data and increasing the depth of search during self play at the cost of simulation speed. This is still open for examination in future research.

## Chapter 7

# Conclusion

### Summary

Conventional chess engines evaluate boards with game tree search and fast static evaluations at leaf nodes. The height of the search tree is the most crucial factor for their performance, which is why this trait has been extensively optimized. The static evaluations are carefully designed heuristics based on expert knowledge provided by humanity. Chess computers are so much stronger than humans because they use what we know about chess in a nearly perfected manner.

One might learn more about chess itself if machines were able to teach chess to themselves objectively. The machine learning tool that may help with these kind of problems is deep reinforcement learning, due to its recent successes in *Atari* games and *go*, where feature maps are learned from sensory input data like images or bitboards.

The first contribution of this thesis is that it provides techniques and algorithms for how chess can be fit in a deep RL framework. An unbiased static evaluation function is designed with a convolutional neural network. Chess positions are vectorized to bitboards only representing chess rules and visuals of the board. These bitboards are subsequently the input of the **cnn**. Multiple algorithms like TD learning, bootstrapping tree search, Monte Carlo tree search and policy networks have been addressed to learn the neural network through self play. An issue that every previous attempt to use RL in chess has solved with expert knowledge, is the network initialization issue. We have brought forward an algorithm, Monte Carlo value initialization, that tries to initialize a network objectively by exploration.

We decided to focus the experiments on the comparison between two TD learning algorithms: TD-Leaf( $\lambda$ ) and TD-Stem( $\lambda$ ). The experiments show how TD-Stem( $\lambda$ ) learns faster in these environments than the state of the art algorithm. We provided two possible reasons why TD-Stem outperforms TD-Leaf in our experiments:

1. The influence of positive rewards propagates faster in updates, because depth plays a fundamental part to the learned value function at the states and their leaf nodes and so on.



2. The wrong belief effect in TD-Leaf( $\lambda$ ), where the actual outcome of a simulation may influence states that should not take any credit, slows down learning

When trying to generalize the experiments and using deeper networks, we observed how the resulting level of the trained models was bad. This was explained by a lack of sensible data to reach enough generalization for the deep network to learn.

## Future Work

This work leaves many options open for future research. First of all, it may be interesting if someone would extend the research in this book to bigger end games, by learning models from the bottom up, i.e first training models on boards with less pieces and gradually increasing the number of pieces on the board.

Furthermore, if anyone has the time and/or more specialized hardware to its disposal with cloud computing for example (simulations were proven to be very time consuming) larger networks with more bitboard channels as input may be trained. We have the feeling that the optimal value function in chess needs a network with many connections.

Another thing to try in future research is the initialization of the value function with tablebases or MCVI, as they are objective ways to do so.

A last valuable research direction, even though we think it may not be suitable for chess; is MCTS, optionally in a hybrid with TD-learning. The same thing can be said for policy networks, we think they may be hard to set up and train, but it could be possible. In this research, the policy was limited to decaying  $\epsilon$ -greedy, but the encoding of actions in policy gradient RL learning has shown its merits in games in the past. The same thing could be true for chess.

# Bibliography

- (2012). Lomonosov endgame tablebases.
- (2016). From ai to protein folding: Our breakthrough runners-up. <http://www.sciencemag.org/news/2016/12/ai-protein-folding-our-breakthrough-runners>.
- Baird, L. (1995). Residual algorithms: Reinforcement learning with function approximation. In *In Proceedings of the Twelfth International Conference on Machine Learning*, pages 30–37. Morgan Kaufmann.
- Baxter, J., Tridgell, A., and Weaver, L. (1999). Tdleaf(lambda): Combining temporal difference learning with game-tree search. *CoRR*, cs.LG/9901001.
- Beal, D. F. (1990). A generalised quiescence search algorithm. *Artificial Intelligence*, 43:85–98.
- Bengio, Y. (2012). Practical recommendations for gradient-based training of deep architectures. *CoRR*, abs/1206.5533.
- Bertsekas, D. P. and Shreve, S. E. (2007). *Stochastic Optimal Control: The Discrete-Time Case*. Athena Scientific.
- Bertsekas, D. P. and Tsitsiklis, J. N. (1996). *Neuro-Dynamic Programming*. Athena Scientific, 1st edition.
- Boureau, Y.-L., Le Roux, N., Bach, F., Ponce, J., and Lecun, Y. (2011). Ask the locals: multi-way local pooling for image recognition. In *ICCV'11 - The 13th International Conference on Computer Vision*, Barcelone, Spain.
- Browne, C. and Powley, E. (2012). A survey of monte carlo tree search methods. *Intelligence and AI*, 4(1):1–49.
- CC (2017a). <http://www.computerchess.org.uk/ccrl/4040/index.html>. CC 40/40 Rating list.
- CC (2017b). <http://www.computerchess.org.uk/ccrl/4040/games.html>. 720000 computer games.
- Daniel Edwards, T. H. (1961). The alpha-beta heuristic. *AIM-030*. reprint available from DSpace at MIT.

- David, O. E., Netanyahu, N. S., and Wolf, L. (2016). *DeepChess: End-to-End Deep Neural Network for Automatic Learning in Chess*, pages 88–96. Springer International Publishing, Cham.
- David Silver, Aja Huang, C. J. M. e. a. (2016). Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489.
- David-Tabibi, O., van den Herik, H. J., Koppel, M., and Netanyahu, N. S. (2009). Simulating human grandmasters: evolution and coevolution of evaluation functions. In Rothlauf, F., editor, *GECCO*, pages 1483–1490. ACM.
- Deng, L. and Yu, D. (2014). Deep learning: Methods and applications. *Found. Trends Signal Process.*, 7(3&#8211;4):197–387.
- Dewey, R. A. (2007). *Psychology: An introduction*.
- Duchi, J. C., Hazan, E., and Singer, Y. (2011). Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12:2121–2159.
- FIDE (2014). <https://www.fide.com/fide/handbook.html?id=171&view=article>.
- Fiekas, N. (2017). <https://pypi.python.org/pypi/python-chess/0.18.41>. Python chess library v0.18.4.
- Fishburn, J. (1984). *Analysis of speedup in distributed algorithms*. UMI Research Press, Ann Arbor, Mich.
- Goodfellow, I., Bengio, Y., and Courville, A. (2016a). *Deep Learning*. MIT Press. <http://www.deeplearningbook.org>.
- Goodfellow, I., Bengio, Y., and Courville, A. (2016b). *Deep Learning*. MIT Press. <http://www.deeplearningbook.org>.
- Hyatt, R. M. (1999). Book learning-a methodology to tune an opening book automatically. *ICCA JOURNAL*, 22(1):3–12.
- John McCarthy, Marvin Minsky, N. R. C. S. (1955). A proposal for the dartmouth summer research project on artificial intelligence. <http://www-formal.stanford.edu/jmc/history/dartmouth/dartmouth.html>.
- Kamko, A. (2016). Alpha-beta pruning practice. [http://inst.eecs.berkeley.edu/~cs61b/fa14/ta-materials/apps/ab\\_tree\\_practice/](http://inst.eecs.berkeley.edu/~cs61b/fa14/ta-materials/apps/ab_tree_practice/). developed for UC Berkeley.
- Kiiski, J. (2011). Stockfish’s tuning method. <http://www.talkchess.com/forum/viewtopic.php?t=40662>.
- Kishimoto, A. (2002). Transposition table driven scheduling for two-player games.

- Kloetzer, J. (2011). *Monte-Carlo Opening Books for Amazons*, pages 124–135. Springer Berlin Heidelberg, Berlin, Heidelberg.
- Korf, R. E. (1985). Depth-first iterative-deepening: An optimal admissible tree search. *Artif. Intell.*, 27(1):97–109.
- Krauthammer, C. (1997). Be afraid. Weekly Standard.
- Krizhevsky, A., Sutskever, I., and Hinton, G. E. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, page 2012.
- Lai, M. (2015). Giraffe: Using deep reinforcement learning to play chess. diploma thesis, Imperial College London.
- Levinovitz, A. (2014). The mystery of go, the ancient game that computers still can’t win. <https://www.wired.com/2014/05/the-world-of-computer-go/>.
- Lowe, D. G. (2004). Distinctive image features from scale-invariant keypoints. *Int. J. Comput. Vision*, 60(2):91–110.
- Maas, A. L., Hannun, A. Y., and Ng, A. Y. (2014). Rectifier Nonlinearities Improve Neural Network Acoustic Models.
- Mishkin, D., Sergievskiy, N., and Matas, J. (2016). Systematic evaluation of CNN advances on the imagenet. *CoRR*, abs/1606.02228.
- Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. (2013). Playing atari with deep reinforcement learning. cite arxiv:1312.5602Comment: NIPS Deep Learning Workshop 2013.
- Murray Campbell, Joseph Hoane Jr, F.-h. H. (2002). Deep blue. *Artificial Intelligence*, 134(1–2):57–83.
- Nair, V. and Hinton, G. E. (2010). Rectified linear units improve restricted boltzmann machines. In *Proceedings of International Conference on Machine Learning (ICML)*, pages 807–814.
- Omid E. David, Nathan S. Netanyahu, L. W. (2016). Deepchess: End-to-end deep neural network for automatic learning in chess. In Villa A., Masulli P., P. R. A., editor, *Artificial Neural Networks and Machine Learning ICANN 2016*, volume 9887. Springer.
- Pearl, J. (1980). Scout: A simple game-searching algorithm with proven optimal properties. In *Proceedings of the First Annual National Conference on Artificial Intelligence*. Stanford.
- Penrose, R. (2017). <http://www.telegraph.co.uk/science/2017/03/14/can-solve-chess-problem-holds-key-human-consciousness/>.
- Plaat, A., Schaeffer, J., Pijls, W., and de Bruin, A. (1996). Best-first fixed-depth minimax algorithms. *Artif. Intell.*, 87(1-2):255–293.

- Pătraşcu, M. and Thorup, M. (2012). The power of simple tabulation hashing. *J. ACM*, 59(3):14:1–14:50.
- Ramanujan, R., Sabharwal, A., and Selman, B. (2011). On the behavior of uct in synthetic search spaces.
- Reinefeld, A. (1989). *Spielbaum-Suchverfahren*. Number ISBN 3540507426. Springer-Verlag, Berlin.
- Reinefeld, A. and Marsland, T. A. (1994). Enhanced iterative-deepening search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 16(7):701–710.
- Riedmiller, M. (2005). Neural fitted q iteration first experiences with a data efficient neural reinforcement learning method. In *In 16th European Conference on Machine Learning*, pages 317–328. Springer.
- Robert, C. P. and Casella, G. (2005). *Monte Carlo Statistical Methods (Springer Texts in Statistics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.
- Robert Hyatt, A. C. (2005). The effect of hash signature collisions in a chess program. *ICGA Journal*, 28(3).
- Russel J. Stuart, P. N. (2003). *Artificial Intelligence: A Modern Approach*. Number ISBN 0-13-790395-2. Prentice Hall, Upper Saddle River, New Jersey, 2nd edition. [http:// aim a. cs. berkeley. edu/](http://aima.cs.berkeley.edu/).
- Saad, D. (1998). *On-line learning in neural networks*. Cambridge University Press, Cambridge England New York.
- Samuel, A. L. (1959). Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, 3(3):210–229.
- Schaeffer, J., Lake, R., Lu, P., and Bryant, M. (1996). CHINOOK: The world man-machine checkers champion. *The AI Magazine*, 16(1):21–29.
- Shannon, C. E. (1950). Programming a computer for playing chess. *Philosophical Magazine*, 41(314).
- Sutskever, I., Martens, J., Dahl, G., and Hinton, G. (2012). On the importance of initialization and momentum in deep learning.
- Sutton, R. S. and Barto, A. G. (1998). *Introduction to Reinforcement Learning*. MIT Press, Cambridge, MA, USA, 1st edition.
- Sutton, R. S., Mcallester, D., Singh, S., and Mansour, Y. (2000). Policy gradient methods for reinforcement learning with function approximation. In *In Advances in Neural Information Processing Systems 12*, pages 1057–1063. MIT Press.

- Szepesvári, C. (2010). *Algorithms for Reinforcement Learning*. Morgan & Claypool.
- Tesauro, G. (1995). Temporal difference learning and td-gammon. *Communications of the ACM*, 38(3):58–68.
- Turing, A. (1953). Chess. *Digital Computers Applied to Games*.
- Van den Heede, D. (2017). <https://github.ugent.be/devdnhee/warrior-chess-bot/tree/master/rookie>.
- van den Herik, L. K. W. H. M. U. P. (2002). The neural movemap heuristic in chess. *Computers and Games*, 2883:154–170.
- Veness, J., Silver, D., Blair, A., and Uther, W. (2009). Bootstrapping from game tree search. In Bengio, Y., Schuurmans, D., Lafferty, J. D., Williams, C. K. I., and Culotta, A., editors, *Advances in Neural Information Processing Systems 22*, pages 1937–1945. Curran Associates, Inc.
- Wang, Z., de Freitas, N., and Lanctot, M. (2015). Dueling network architectures for deep reinforcement learning. *CoRR*, abs/1511.06581.

# Appendix A

## Chess

In this appendix, we briefly discuss the rules of chess together with some notions used throughout the main content of the dissertation. It is only advised to those without game specific knowledge to read this chapter. First, we summarize the rules in section A.1. Chess players can be classified in strength with a rating system laid out in section A.2. To finalize, some chess specific terms used in the thesis are defined in A.3.

### A.1 Rules

Chess is a game played with two players denoted by the color of their pieces: black and white. Both sides start with 16 pieces at the start of the game. The players are allowed to play one move with one of their pieces on a checkered board every turn. Every square on this board has its own unique name as denoted in figure A.1. The allowed movements on the chessboard are further discussed in section A.1.1. The ultimate goal of both sides is to checkmate the opponent, which is explained in section A.1.1. Only the most noteworthy game specific elements are explained in this section, an official and complete explanation is provided by the FIDE (the world chess federation) (FIDE, 2014).

#### A.1.1 Chess Pieces

Every piece is allowed to take other pieces from the opponent, but is blocked by its own kind. The mobility of every piece is indicated in figure A.2. During the game, some special moves may occur, these are explained in section A.1.3 The initial setup of the board is depicted in figure A.3.

#### A.1.2 Special Positions

During a game of chess, some special positions may occur. Some examples are demonstrated in figure A.4 to clear things up.

- **check.** The king is under attack, an opponent's piece would capture the king if it were

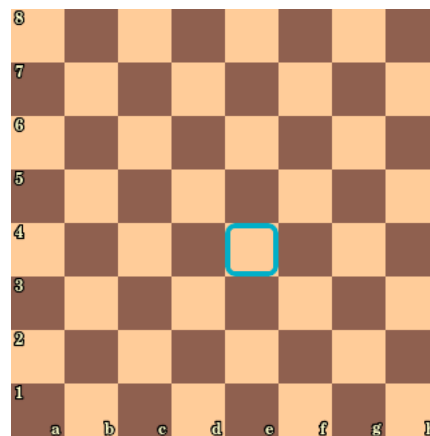


Figure A.1: The chessboard layout. The rows and columns are identified with letters and numbers respectively. For example, the marked square is e4.

not to move. The rules of chess prohibit the possibility of a king being captured. To this reason, the player is obliged to make a move escaping the assault on the king.

- **checkmate.** This is a check where no legal move can be played. This ends the game in favor of the player performing the checkmate.
- **stalemate.** There are no legal moves left, but the king is not under attack. These positions are followed by a draw (see section A.1.4).

### A.1.3 Special Moves

There exist three special moves, which happen rarely during a game of chess.

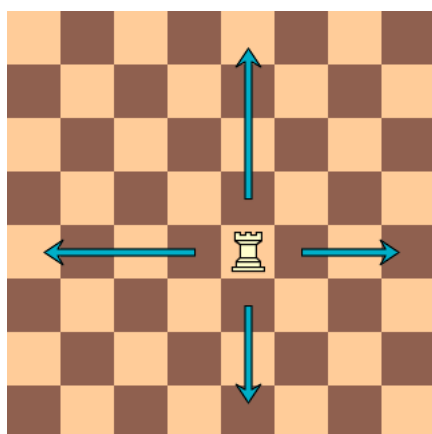
- **castling.** When the king and rook have not been moved yet during the game, castling may occur. This consists of moving the king two squares towards a rook at the base rank as indicated in figure A.5, followed by placing the rook next to the king at the opposite side of the king movement.
- **en passant capture.** To make up for the double push of pawns at their starting positions and hence missed opportunity for their adversed counterparts, pawns at the fourth rank have one and only one chance to capture these double pushed pawns.
- **promotion.** When a pawn reaches the last rank (thus it would not be able to move forward anymore), it gets promoted to one of the following pieces: queen, rook, bishop, knight. A queen promotion is shown in figure A.7.

### A.1.4 Termination

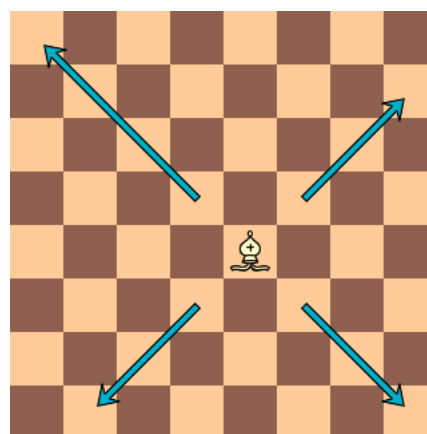
There are three possible results to a chess game:

1. white wins: 1-0
2. draw: 1/2-1/2

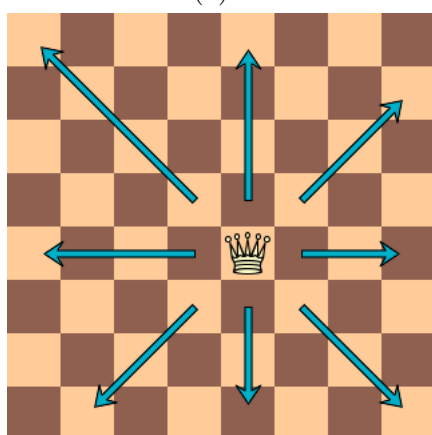




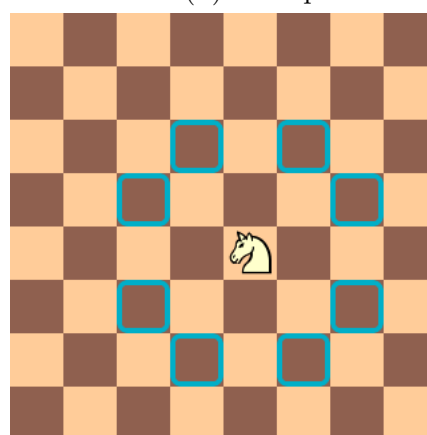
(a) Rook



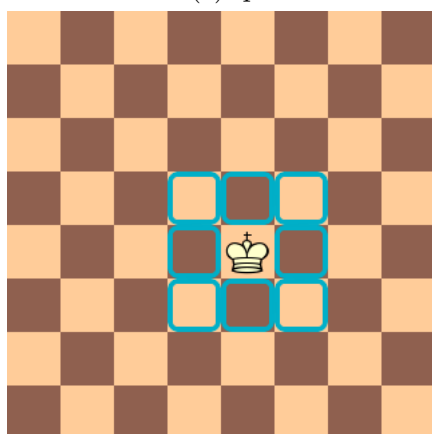
(b) Bishop



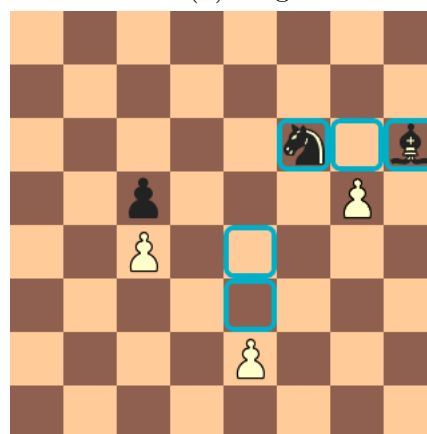
(c) queen



(d) knight



(e) king



(f) pawn

Figure A.2: Piece mobility. (a) The rook is allowed to move vertically and horizontally. (b) The bishop can move diagonally. (c) The queen's mobility is the combination of the rook's and bishop's. (d) The knight is the only piece that can 'jump' over pieces, by moving in an L-structure. (e) The king's movement is the same as the queen's, but is limited to the surrounding squares. (f) The pawn moves one square forward, but can only capture diagonally. When it is at its starting position (see figure ), it can travel 2 squares forward as well.

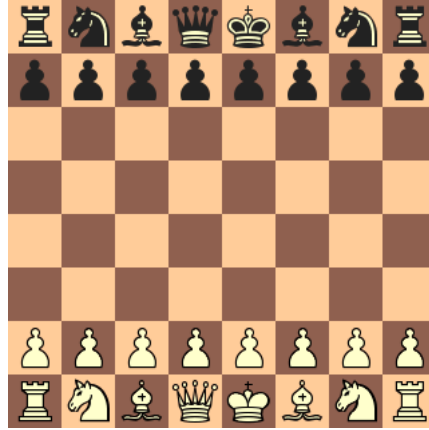


Figure A.3: The initial set up of chess. White is the first side to make a move.

### 3. black wins: 0-1

The game terminates when one of the following events occur:

- checkmate, wins the game.
- a player resigns, hence he loses.
- stalemate results in a draw
- insufficient mating material, when it is impossible to achieve a checkmate through legal play the game is drawn.
- draw by agreement
- draw by repetition can be claimed after reaching the exact same board position three times over the course of the game
- 50-move rule, when no opponent is making any progress, a draw can be claimed. This is defined as 50 moves (both black and white) without any captures and pawn moves.

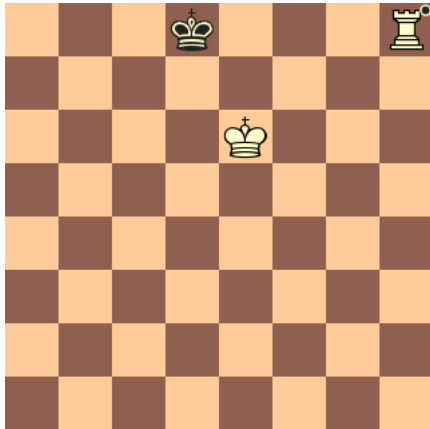
It may be interesting to note that the rules of chess are designed in such a way that every game would end eventually.

## A.2 Evaluation of Strength

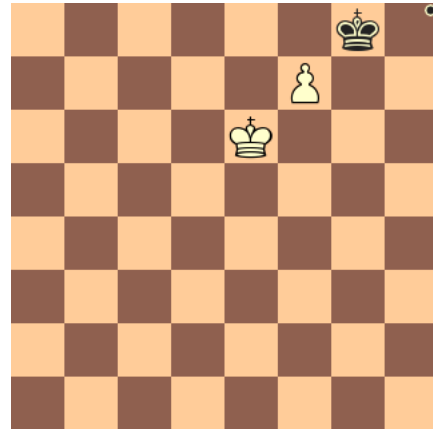
To assess the strength of chess players, the *Elo* rating system has been invented by Hungarian physicist *Arpad Elo*. It has been used as a competitive score since then for other games like scrabble. The idea is to use the rating difference as a predictor of the match. Based on the probabilistic prediction and the true outcome, the players' ratings are adjusted. This means that winning against a stronger opponent results in a higher elo gain and vice versa.

Suppose player  $A$  with rating  $R_A$  plays against player  $B$  with rating  $R_B$ . The expected outcome of the game with respect to  $A$  is then

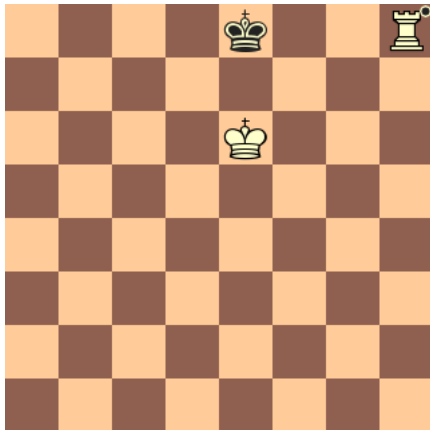
$$E_A = \frac{Q_A}{Q_A + Q_B}$$



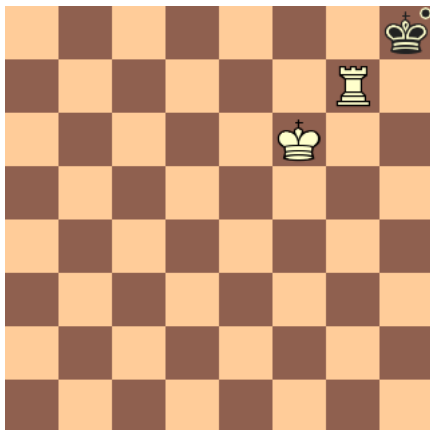
(a) Example of check



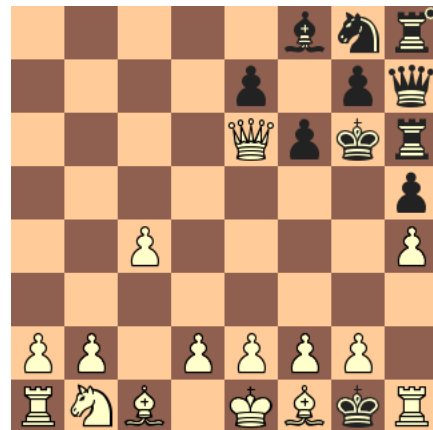
(b) Example of check



(c) Example of checkmate

(d) *Legall's* checkmate

(e) Example of stalemate



(f) Example of stalemate

Figure A.4: Special chess positions

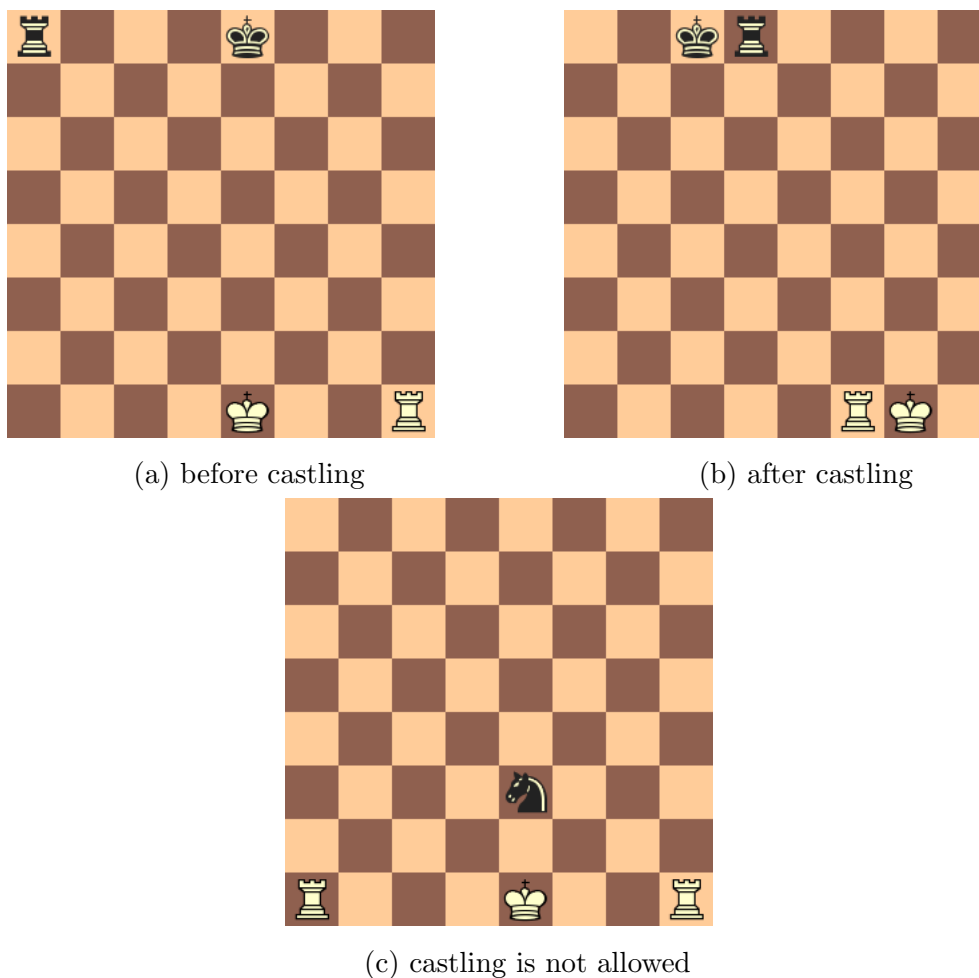


Figure A.5: (a) & (b): White performs king side castling, while black castles queen side. (c) Castling is forbidden in case the king had to pass an attacked square (Here squares f1 and d1 for king side and queen side castling respectively).

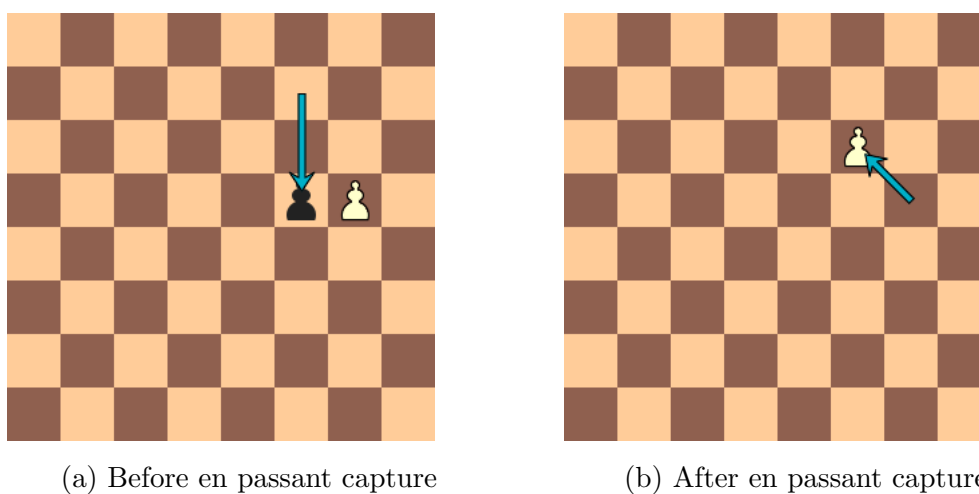


Figure A.6: Demonstration of how an en passant move is performed.

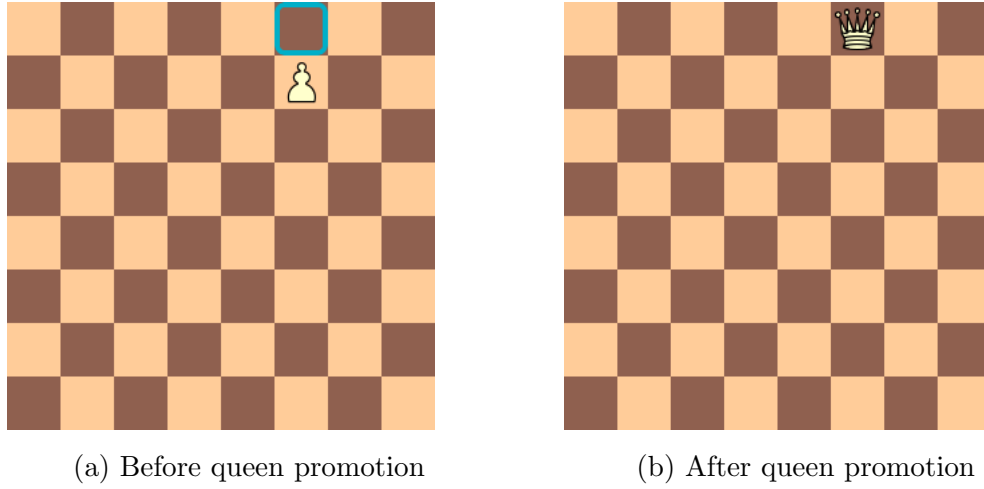


Figure A.7: Promotion of a pawn to a queen.

with

$$Q_A = 10^{R_A/400}$$

$$Q_B = 10^{R_B/400}$$

Hence the outcome is estimated with a logistic curve with base 10. This means that every rating difference of 400 points estimates the ratio of outcomes 10:1 in favor of the highest rating. The effective outcomes can be

$$S_A = \begin{cases} 1 & \text{A wins} \\ \frac{1}{2} & \text{draw} \\ 0 & \text{A loses} \end{cases}$$

The rating of  $A$  is then updated with

$$R_A \leftarrow R_A + K(S_A - E_A)$$

with  $K$  being the  $K$ -factor. In FIDE regulations,  $K$  gets smaller after playing more matches and achieving a higher rating, as the uncertainty about the real strength has decreased.

## A.3 Chess Jargon

Chess specific words that may be beneficial for understanding certain concepts within this document are presented in table A.1.

Table A.1: Glossary with chess terms

<b>attack</b>	a piece is under attack if there is a direct threat to capture it
<b>black</b>	player of the black pieces
<b>board</b>	chessboard
<b>capture</b>	a piece moves to a square occupied by the opponent
<b>castling</b>	see section A.1.3
<b>check</b>	see section A.1.3
<b>checkmate</b>	see section A.1.3
<b>draw</b>	see section A.1.4
<b>endgame</b>	the last phase possible in a chess game, where most pieces have already been captured.
<b>en passant</b>	see section A.1.3
<b>exchange</b>	an exchange denotes the balance between the captured pieces of both sides in a move sequence
<b>file</b>	vertical set of squares on the board
<b>half move</b>	one ply
<b>kingside</b>	the right half of the board (in white's perspective)
<b>mate</b>	same as checkmate
<b>middle game</b>	stage of the game between the opening and endgame
<b>minor piece</b>	a bishop or a knight
<b>move</b>	Depending on the context one or two plies or half moves.
<b>opening</b>	the first phase of the game, when most of the pieces are still on the board. More or less the first
<b>piece</b>	One of the 32 figurines of the game
<b>promotion</b>	see section A.1.3
<b>queenside</b>	the left half of the board (in white's perspective)
<b>rank</b>	a row, generally denoted by a number from 1 to 8
<b>repetition</b>	see section A.1.4
<b>stalemate</b>	see section A.1.2
<b>white</b>	player of the white pieces
<b>50 move rule</b>	see section A.1.4