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(λ), and compare it with the state of the art, TD-Leaf(λ).**

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 20th 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(λ) 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(λ) 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(λ) 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 $s \in \mathcal{S}$.
- $r_A : \mathcal{S} \times \mathcal{A}_\mathcal{S} \mapsto \mathbb{R}$ 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 episodical, meaning it has terminal states (checkmate, stalemate, ...). An episode is represented with its history from whites perspective

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

where we assume $a_i \in \mathcal{A}_{s_i}, \forall i \in 0, \cdots, T-1$, 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: $R_t = \sum_{t'=t}^{T-1} \gamma^t r_{t'+1}$. 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:

$$V(s_t) = \mathrm{E}\left[R_t | s\right] \tag{1}$$

$$= \mathrm{E}\left[\sum_{t'=t}^{T-1} \gamma^t r_{t'+1} | s\right] \tag{2}$$

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:

$$z_H = r_T \tag{3}$$

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

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

$$V(s) = \mathrm{E}\left[z | s\right] \tag{5}$$

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)

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

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 ϵ-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_θ is differentiable with respect to its tunable parameters θ, 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 α by calculation of the gradient of the loss function:

$$\theta \leftarrow \theta - \alpha \nabla L_\theta \tag{7}$$

Every episode individually corresponds to the update

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

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

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

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

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

A. TD-(λ)

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

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

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 (minibatch) SGD on bigger datasets after simulating a whole iteration of episodes:

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

B. TD-Leaf(λ)

With the understanding of the machinery in TD(λ), 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

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

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

Instead of updating the encountered state as in TD(λ), 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.

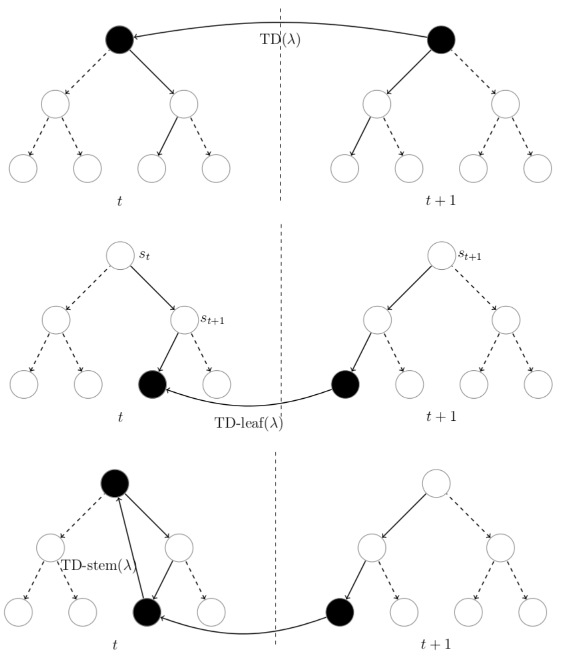


Fig. 1: Comparison between the 3 discussed TD-learning algorithms, the PV are indicated with solid edges. In classical TD-learning the updates are between successive states. TD-leaf updates are between successive leaf nodes and in TD-stem both ideas are combined.

C. TD-Stem(λ)

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 leafs 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].

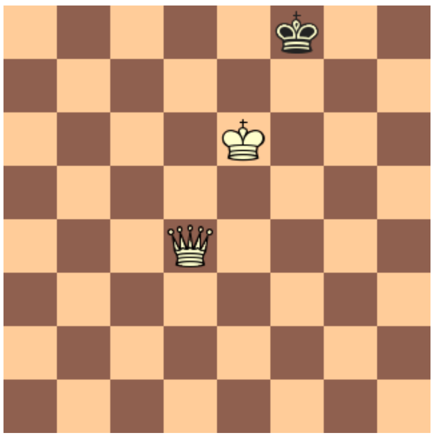


Fig. 2

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
♔	$\begin{matrix}0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\\0&0&0&0&1&0&0&0\\0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\end{matrix}$	$\begin{matrix}0&0&0&0&0&0&0&0\\0&0&0&1&0&0&0&0\\0&0&0&1&0&1&0&0\\0&0&0&1&1&1&0&0\\0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\end{matrix}$
♕	$\begin{matrix}0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\\0&0&0&1&0&0&0&0\\0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\end{matrix}$	$\begin{matrix}0&0&0&1&0&0&0&1\\1&0&0&1&0&0&1&0\\0&1&0&1&0&1&0&0\\0&0&1&1&1&0&0&0\\1&1&1&0&1&1&1&1\\0&0&1&1&1&0&0&0\\0&1&0&1&0&1&0&0\\1&0&0&1&0&0&1&0\end{matrix}$
♚	$\begin{matrix}0&0&0&0&0&1&0&0\\0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\end{matrix}$	$\begin{matrix}0&0&0&0&1&0&1&0\\0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\end{matrix}$
♛	$\begin{matrix}0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\end{matrix}$	$\begin{matrix}0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\end{matrix}$

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 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:

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

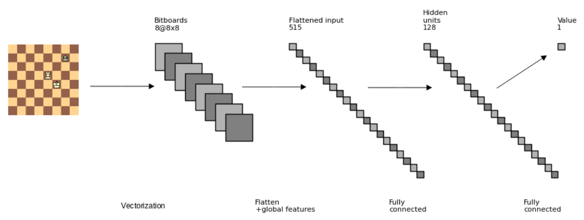
The complete simulation is performed through stages, which are manually controllable. The process has been further optimized with the addition of hyper-parameters (see table II), modifiable through stages. In table III the configurations for the stages conducted in the experiments are laid out.

TABLE II: Hyper-parameters modifiable in between stages.

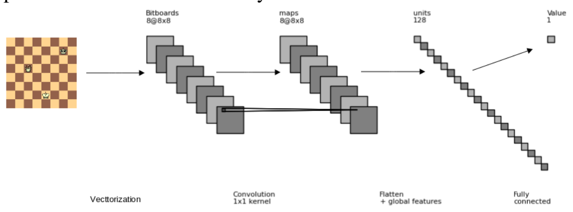
λ	Parameter for TD-learning methods
d_r	Depth at which immediate wins are sought
d_V	Depth at which the value network is called
f_ϵ	Decay function for exploration parameter $\epsilon = f(i)$. i increments every iteration.
I	Number of iterations
i_0	First iteration number, to initialize ϵ with the f_ϵ
K	The number of states that are used to calculate the λ-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	λ	i_0	K	d_r
	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
1	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
	1	500	20	1	0.5	0	10	3
	2	250	20	3	0.6	20	20	3
2	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



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



(b) The (relatively simple) CNN architecture deployed in experiment 2.

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

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

	TD-Leaf(λ)	**TD-Stem(λ)**
WCR	0.48	**0.85**
WE	**0.87**	0.86
LHS	0.80	**0.91**
N	353 500	**304 500**

A. *Experiment 1: king rook king endgame*

The first simulations we ran were on the king rook king (KRK) 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 ϵ 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 (KQK) endgame. The learning rate of the network depicted in figure 3b is $\alpha = 10^{-4}$. The exploration parameter ϵ 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 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.

TABLE V: Performance comparison TD-Leaf(λ) and TD-Stem(λ)

	3 stages		**5 stages**	
	TD-Leaf(λ)	**TD-Stem(λ)**	**TD-Leaf(λ)**	**TD-Stem(λ)**
WCR	0.65	**0.77**	0.90	0.90
WE	**0.67**	0.64	0.89	0.89
LHS	0.89	0.89	0.95	**0.97**
N	**26000**	27000	**43500**	44500

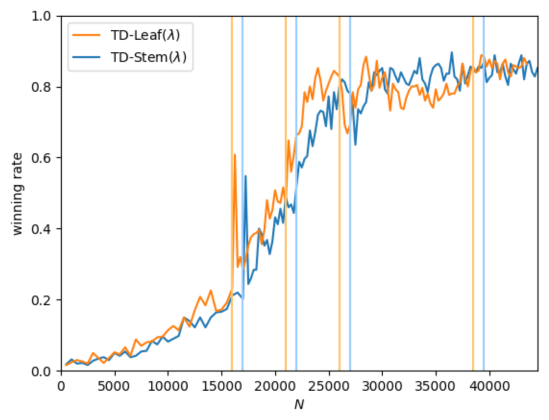


Fig. 4: The learning curve of the second experiment, divided in 5 stages.

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(λ). We have compared this proposal with TD-Leaf(λ) 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` 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 andStanley 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