TD-Leaf(λ)

Giraffe: Using Deep Reinforcement Learning to Play Chess

Stefan Lüttgen

Motivation

- Learn to play chess
- Computer approach different than human one
- Humans search more selective:
  Kasparov (3-5 positions per second)
- Computers much more exhaustive:
  Deep Blue (200 million positions per second)
- How can humans still compete with computers?

Goal

- No complex and/or handcrafted evaluation function
- Learn evaluation function in a hierarchical fashion using deep learning and reinforcement learning
- Derive own rules through self-play in evaluating position
- Only provide piece values
- Only use basic features

Overview

- Introduction
- Current conventional chess engines & related work
- Deep learning framework
- TD-Leaf(λ)
- Experiments and Results
- Conclusion & Outlook

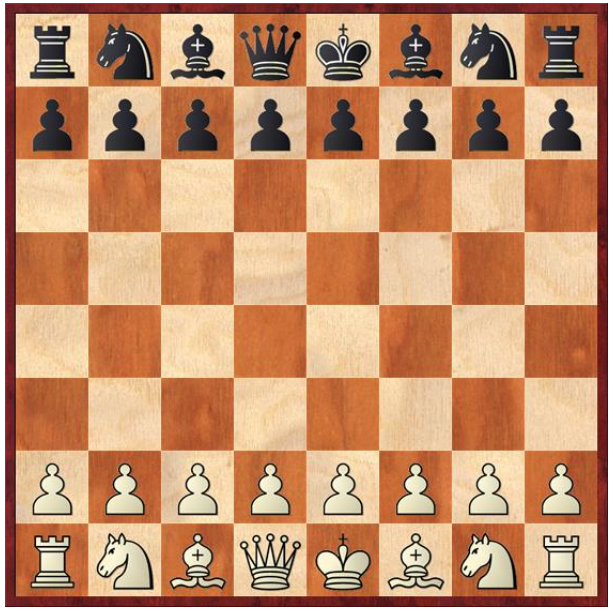
Introduction

Evaluating a chess position:

- assign a score δ that corresponds to the chance of winning for the side to move
- δ must be monotonically increasing with respect to the chance of winning (if the side to move plays error free)
- Score centered around 0.00 (50 % chance of winning or draw position):

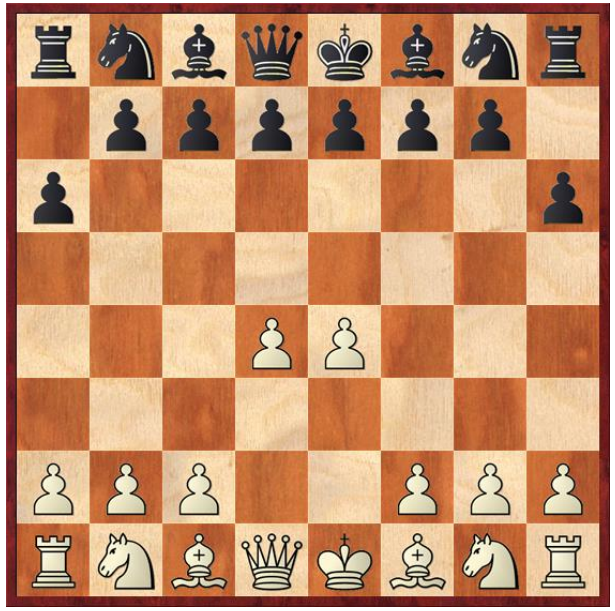
0.30 maps to = (white's 1st move advantage)
0.60 maps to +/= : Slight advantage for white
0.90 maps to +/- : Clear advantage for white
1.30 and above maps to +- : White is winning

Example Evaluations:



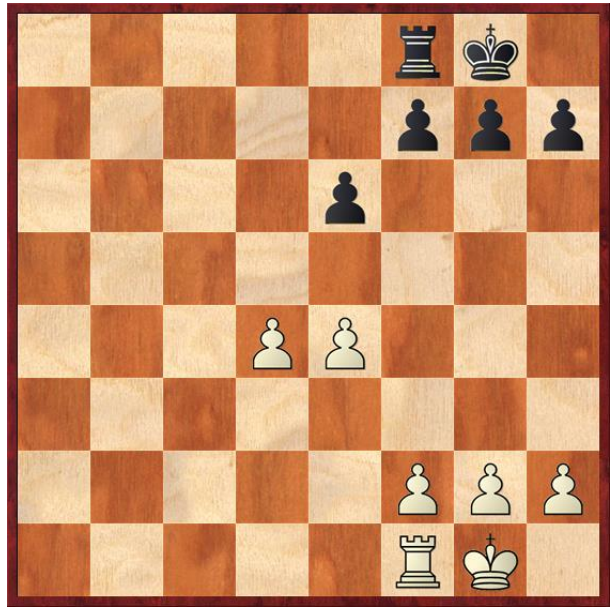
+ (0.30): "1st move advantage "

Starting position



+ (0.59): „slight advantage„

White in the center
Black passive



+ (0.93): "clear advantage "

Extra pawn

-> Position simple, still undecided!

Example Evaluations:

+ (1.60): "winning position"

Materially equal, but:

- Space
- Bishop pair
- Piece activity
- Initiative

-> Evaluating a chess position is complex!

Overview

- Introduction
- Current conventional chess engines & related work
- Deep learning framework
- TD-Leaf(λ)
- Experiments and Results
- Conclusion & Outlook

Conventional Chess Engines

- Depth-limited minimax() with α-β pruning and q-search
- Chess:
  - Avg. branching factor 35
  - Avg. game length 80 plies
  - Tree size ~10^{46} (w/o repititions)

-> Employ fixed depth with static evaluation at end of sub-tree

Drawback: Horizon effect

-> Tackle using q-search

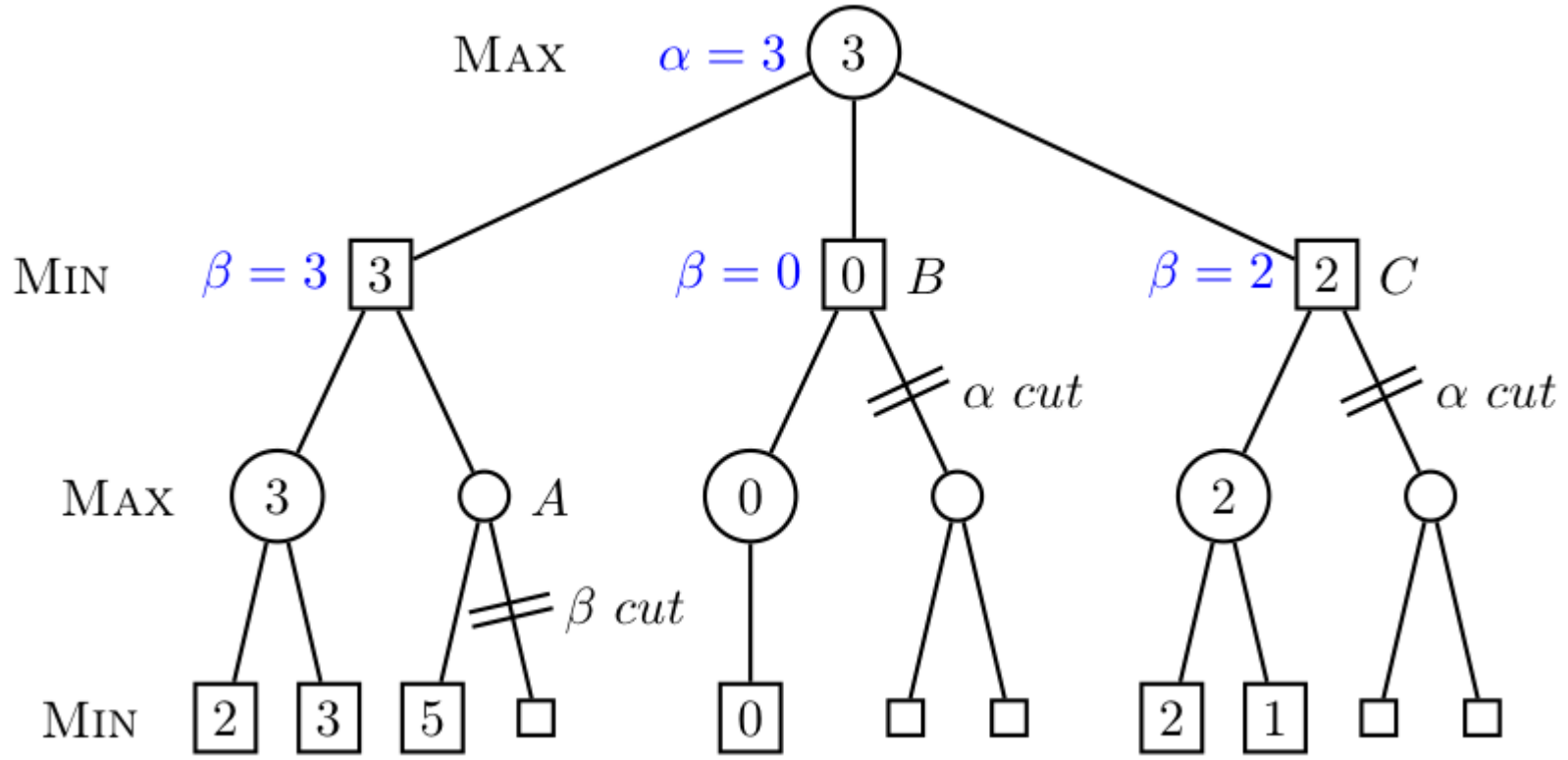
α-β Pruning

- Check lower bound α and upper bound β “window” before calling minimax() at move/node within the tree
- Only explore nodes that can potentially be useful
- Optimal move ordering can reduce branching factor to the square root

-> twice as many searches in the same time

- Heuristics for move ordering (killer heuristic)

α-β Pruning Example



At Max Node: if v ≥ β → β-cut
At Min Node: if v ≤ α → α-cut

Evaluation Function

- Assign score to a position without looking ahead:
  - Material
  - Pawn Structure
  - Piece-specific Evaluation
  - Mobility
  - King Safety

$$f_1 \cdot material + f_2 \cdot mobility + f_3 \cdot king\ safety + \ldots$$

-> Complicated, hand-crafted, many parameters to tune

Related Work: KnightCap vs. Giraffe

KnightCap:

- TD(λ) (TD-Gammon) adjusted to TD-Leaf(λ): evaluate terminal nodes
- Went from 1650 to 2150 within three days or 308 games online
- With opening book even better around 2400-2500

Giraffe:

- No play against humans for self discovery
- Less features/ parameters to optimize: 363 vs. 5872
- Having a deep network to hierarchically learn connections between features
- Smoother representation than bitboards

Overview

- Introduction
- Current conventional chess engines & related work
- Deep learning framework
- TD-Leaf(λ)
- Experiments and Results
- Conclusion & Outlook

Hierarchical Feature Extraction: Deep Learning

- Deep learning where knowledge is inherently hierarchical
- Categorize features and combine them later on ("modalities")
- Leave enough space for self-discovery
- Choose more humanly feature representation

Approach: TD-Leaf(λ) with high level feature extraction through deep neural networks given as less hand-crafted features as possible

-> Try to derive positional understanding related to the modalities and their relation to each other

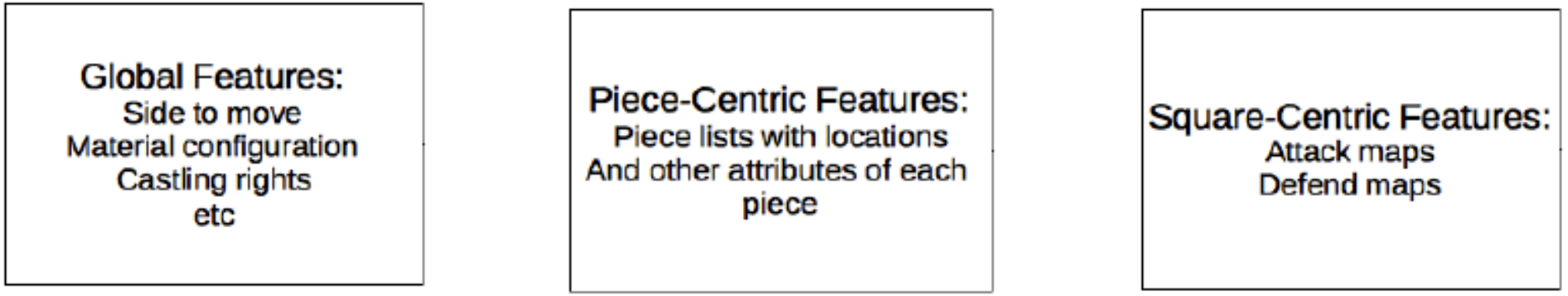
Feature Representation

- Low level features to let the network discover knowledge about a position
- Smooth in how input maps to output: positions that are close together in feature space should have similar evaluations („NN friendly“)
- Slot system to ensure consistent feature length:
  - Piece Lists: slots for existence and coordinates of each potential piece
  - Side to move
  - Castling rights
  - Material Configuration
  - Sliding piece mobility
  - Attack/ Defend Map

-> 363 features in total (KnightCap 5872)

Network Architecture

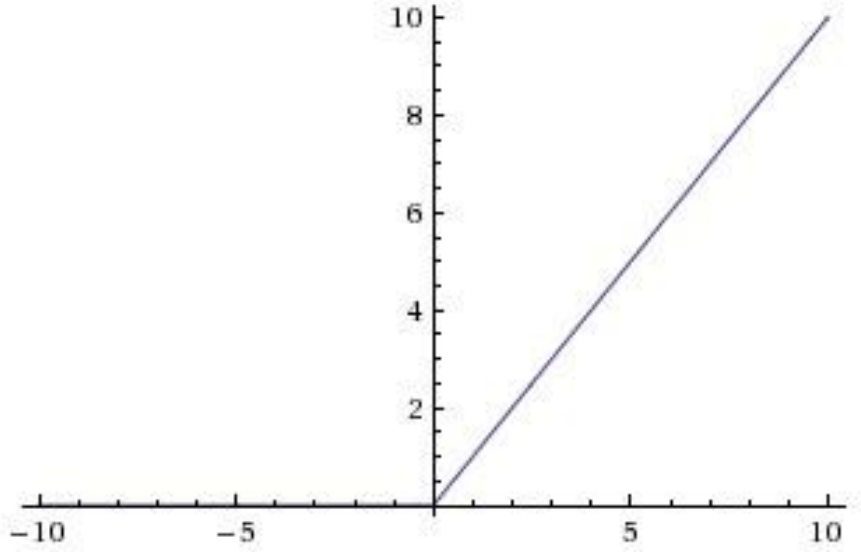
- 3 layer network (2 hidden layer + output layer)
- Rectified Linear activation (ReLU) for hidden nodes: $f(x) = \max(0, x)$
- Hyperbolic tangent function for output layer: maps to $[-1, 1]$
- Three modalities:



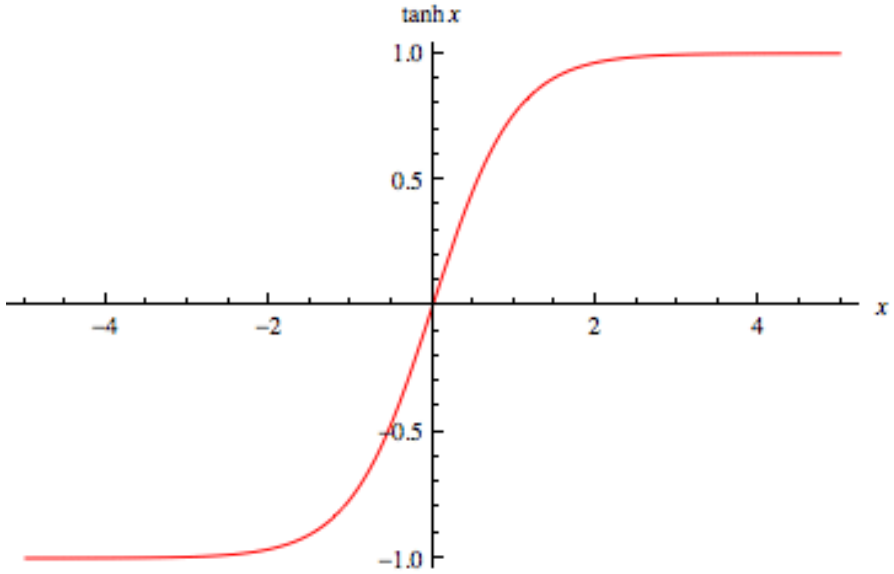
- Modalities seperated in first two layers (avoid overfitting)
- Combine in last two layers (capture interactions between high level concepts derived from first two layers)

Activation Functions

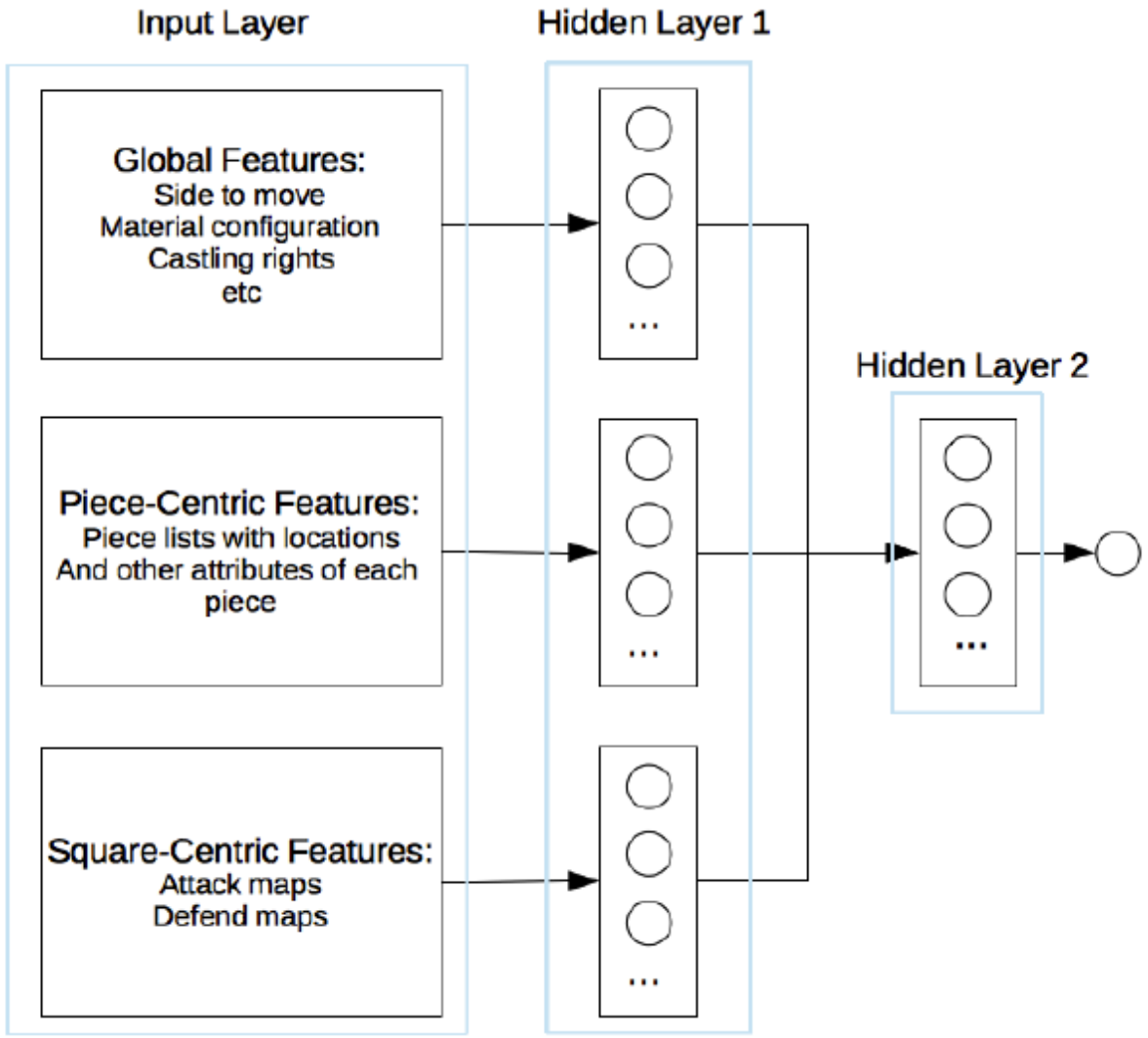
Rectifier (ReLU): $f(x) = \max(0, x)$



Hyperbolic tangent: $f(x) = \tanh(x)$



Network Architecture



Training Set Generation

- Satisfy conflicting objects:
  - High volume: large and sufficient number of training positions (excl. human-based positions, e. g. online play)
  - Correct distribution: model realistic positions
  - Variety: Still ensure learning unequal positions (appear in inner nodes when playing out lines)
- Collect 5 mio. database positions, randomly apply one legal move per position

-> 175 mio. positions in total

Network Initialization & Training

- Bootstrapping by providing basic material values

-> Training the network needs an error signal:

- Property: Local gradient of minimax is the gradient of the evaluation function at terminal nodes
- Using TD-learning: make the evaluation function a better predictor of its own

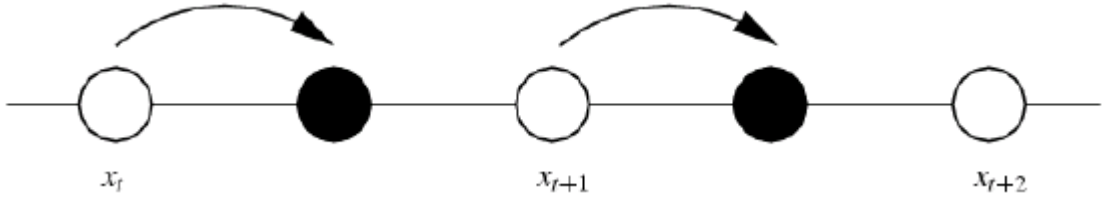
Overview

- Introduction
- Current conventional chess engines & related work
- Deep learning framework
- TD-Leaf(λ)
- Experiments and Results
- Conclusion & Outlook

Evaluation Training

- Generate error signals using TD-Leaf(λ)
- Select 256 random positions (with 1 random move) per iteration
- Let engine play against itself for 12 moves
- Add score changes weighted by decay factor λ^m (m number of moves from starting position)
  - Allow learning long term consequences
  - Prioritize short term consequences patterns

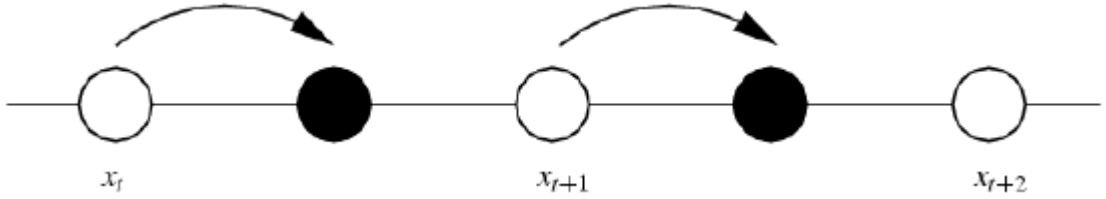
TD-Leaf(λ)



- S: set of all possible environment states
- Agent performes actions at discrete time steps $t = 1, 2 \ldots$
- At time t, agent is in state $x_t \epsilon S$, can choose action $a_t \epsilon A_{x_t}$
- a_t puts environment into state x_{t+1} with probability $p(x_t, x_{t+1}, a_t)$
- After a series of action, agent receives reward $r(x_N)$, where N is the number action in the series (e.g. $-1, 0, 1$)
- Optimal reward predicted by $J^*(x) := E_{x_N|x} r(x_N)$

-> approximate this function

TD-Leaf(λ)



- Temporal difference: $d_t := J'(x_{t+1}, w) - J'(x_t, w)$
- For J^* holds: $E_{x_N|x}[J^*(x_{t+1}) - J^*(x_t)] = 0$
- If $J'(x_t, w)$ is a good approximation, d_t shoud be close to 0
- Difference between outcome of game and penultimate move:

  $d_{N-1} = J'(x_N, w) - J'(x_{N-1}, w) = r(x_N) - J'(x_{N-1}, w)$
- Update after last move:

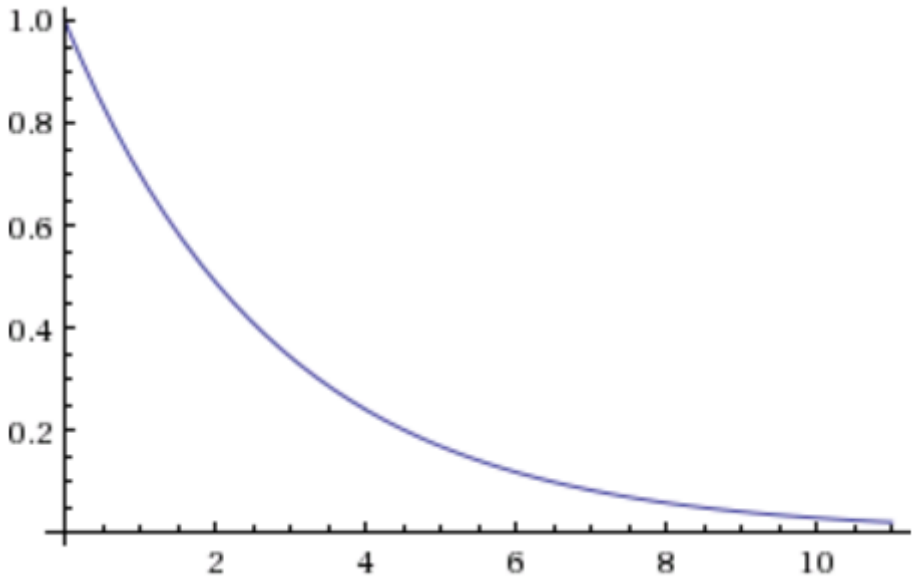
$$w := w + \alpha \sum_{t=1}^{N-1} \nabla J'(x_t, w) \left[\sum_{j=t}^{N-1} \lambda^{j-t} d_t \right]$$

Evaluation at Score Changes

TD-Leaf(λ) update rule:

$$w := w + \alpha \sum_{t=1}^{N-1} \nabla J'(x_t, w) \left[\sum_{j=t}^{N-1} \lambda^{j-t} d_t \right]$$

w - set of weights
α - learning rate (set to 1.0*)*
$\nabla J(x_t, w)$ *- gradient of model at t*
λ – discount factor (set to 0.7*)*
d_t *- temporal difference*



$$\mathrm{f}(x) = 0.7^x, x \in [0, 11]$$

Evaluation Example

#	Search Score	Score Change	λ	Total Error
1	10	0	0.7^0	0
2	20	10	0.7^1	7
3	20	0	0.7^2	0
4	20	0	0.7^3	0
5	-10	-30	0.7^4	-7.2
6	-10	0	0.7^5	0
7	-10	0	0.7^6	0
8	40	50	0.7^7	4.12
9	40	0	0.7^8	0
10	40	0	0.7^9	0
11	40	0	0.7^{10}	0
12	40	0	0.7^{11}	0

Total error after 12 moves: $10 \cdot 0.7^1 - 30 \cdot 0.7^4 + 50 \cdot 0.7^7 = 3.92$

Evaluation Training cont‘d

- Derive gradient of L1 loss using backpropagation
- Using stochastic gradient descent with AdaDelta update rule to train

-> separateley adjusted learning rates per weight

after each iteration based on direction of gradient

-> Takes rarely activated neurons into account, prioritizes frequently activated neurons

Overview

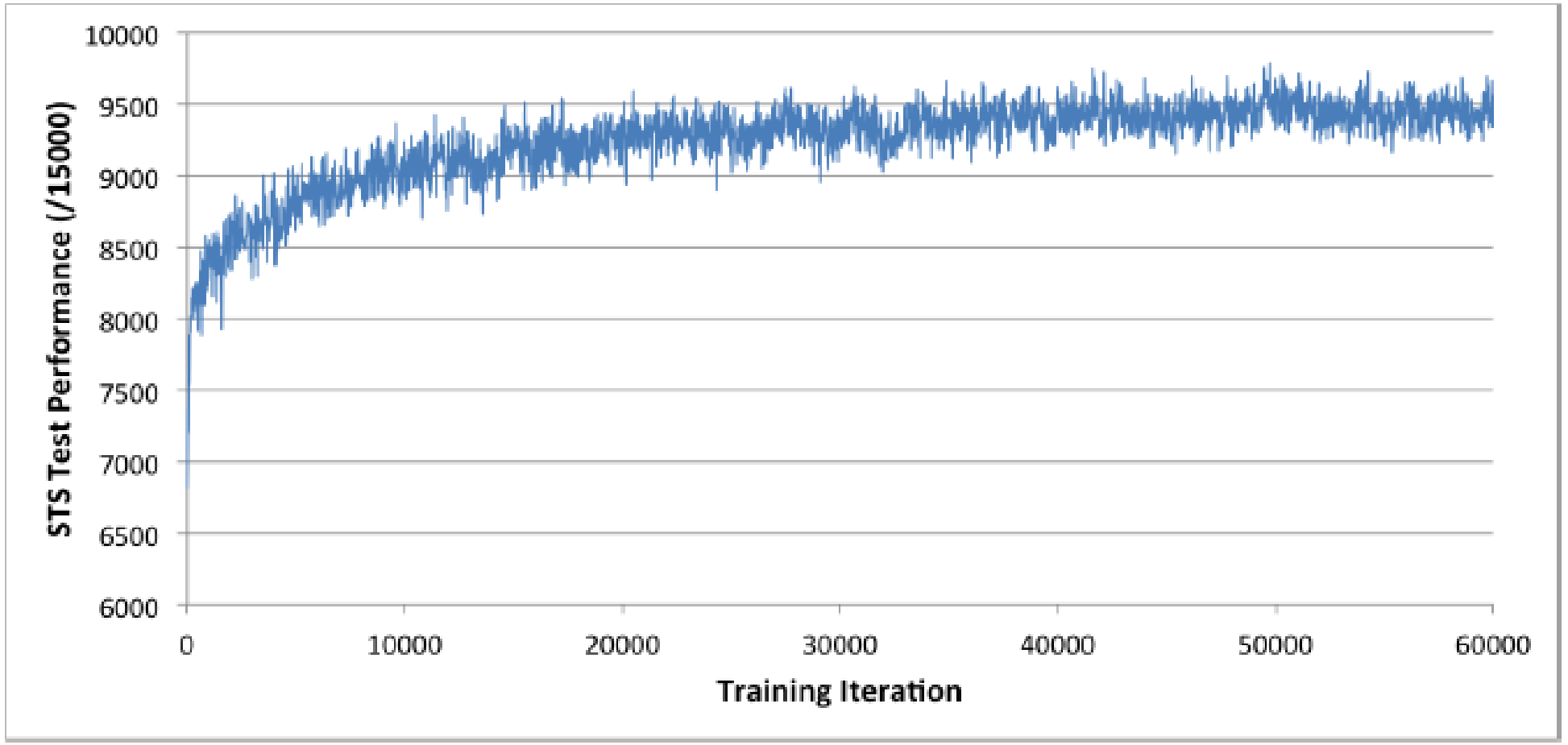
- Introduction
- Current conventional chess engines & related work
- Deep learning framework
- TD-Leaf(λ)
- Experiments and Results
- Conclusion & Outlook

Results

- Test positional understandig via Strategic Test Suite (STS):
  - 15 scenarios of 100 positions each
  - Tactical themes are avoided
  - Score between 0 and 10 per position
- All positions are unknown to the engine
- After bootstrapping: 6000/15000
- Converging after 72 hours to approx. 9500-9700/15000

1. Undermining
2. Open Files and Diagonals
3. Knight Outposts
4. Square Vacancy
5. Bishop vs Knight
6. Re-Capturing
7. Offer of Simplification
8. Advancement of f/g/h pawns
9. Advancement of a/b/c Pawns
10. Simplification
11. Activity of the King
12. Center Control
13. Pawn Play in the Center
14. Queens and Rooks to the 7th Rank

Strategic Test Suite (STS) Results



STS Results Comparison with other Engines

Engine	Approx. Elo Rating	Avg. Nodes Searched	STS Score
Giraffe (1.0 s)	2400	258570	9641
Giraffe (0.5 s)	2400	119843	9211
Giraffe (0.1 s)	2400	24134	8526
Stockfish 5	3387	108540	10505
Senpai 1.0	3096	86711	9414
Texel 1.4	2995	119455	8494
Crafty 24.0	2801	296918	8541
GNU Chess 6	2685	58552	8307

Usually engines are inherently designed to score well in these positions

Overview

- ❖ Introduction
- ❖ Current conventional chess engines & related work
- ❖ Deep learning framework
- ❖ TD-Leaf(λ)
- ❖ Experiments and Results
- ❖ Conclusion & Outlook

Conclusion

- Trained an IM rated (2.2 %) chess position evaluator deep network using TD-Leaf(λ) for error function within 3 days
- Understand positions through self discovered evaluation function using hierarchial representation and learning
- Step away from seeing far ahead in a game tree, but rather tend towards a more humanly approach to the game
- Framework can be ported to other zero-sum board games

Outlook

- Probabilistic search: search until a certain winning probability instead of a certain depth
- Only provide chess rules and have piece values discovered [5]
- Model compression: use larger networks to train smaller ones

– Increase search speed

- Similarity Pruning: human understanding of equivalent move patterns

– Decrease branching factor even more

- Learn management for different time settings

-> make the computer create human-like positions

Resources

[1] Baxter, Tridgell, Weaver, "TDLeaf(λ): Combining Temporal Difference Learning with Game-Tree Search", 1999

[2] Baxter, Tridgell, Weaver "Learning To Play Chess Using Temporal Differences", 2001

[3] Lai, "Giraffe: Using Deep Reinforcement Learning to Play Chess", Master Thesis, 2015

[4] Zeiler, "ADADELTA: AN ADAPTIVE LEARNING RATE METHOD", 2012

[5] Droste, "Learning of Piece Values for Chess Variants", 2008