Evaluating Variance-Aware PUCT in Neural Chess Engines

Axel Ahlqvist
Lund University
ax5810ah-s@student.lu.se

Oscar Näslund Cuesta
Lund University
os3540na-s@student.lu.se

Abstract

Game-playing algorithms often face a trade-off between analyzing moves that currently look promising and exploring alternatives whose value is uncertain. AlphaZero-style chess engines combine neural network evaluation with Monte Carlo tree search to selectively explore the game tree. We evaluate a variance-aware modification of the PUCT tree policy on chess puzzles and find that it underperforms standard PUCT across different networks and search budgets under certain conditions. We suggest explanations for this discrepancy and possible improvements to the implementation of the variance-aware tree policy.

1 Introduction

Chess engines have evolved significantly since IBM's Deep Blue defeated the reigning chess world champion Garry Kasparov in 1997 (Campbell et al., 2002). Deep Blue relied on a classical approach based on large-scale game tree search combined with handcrafted position evaluation functions. In contrast, modern chess engines incorporate machine learning methods that enable the engine to learn evaluation functions and policies that guide the game-tree search. This project evaluates variance-aware prior-based tree policies performance in chess engines, compared to well-established prior-based policies (Silver et al., 2017). We also analyze the resulting differences in search behaviour and decision-making to better understand when variance-aware exploration helps or hurts move selection.

2 Background

Chess is a deterministic perfect-information game in which each board configuration corresponds to a state and legal moves define transitions between states. A chess engine is designed to explore the game tree of possible moves and select moves that lead to the highest expected outcome. Classical chess engines normally evaluate positions using an efficient handcrafted heuristics-based evaluation function paired with a brute force search through a large volume of moves. More recently game playing models have found great success in board games such as chess and Go, which has been viewed as a benchmark for game-playing programs. These models use neural nets that evaluate positions and give a policy for different actions to take. These functions are paired with some variant of Monte Carlo tree search to explore the game tree.

2.1 Monte Carlo Tree Search

Monte Carlo tree search (MCTS) is an algorithm for determining the optimal action from a certain state, by building a tree of the resulting states after actions. In its most general form, the algorithm consists of four steps. We initialize the tree with a root node. The following procedure is then repeated a fixed number of times to expand the tree:

1. **Selection.** Starting at the root node, the algorithm repeatedly selects a child according to a *tree policy* until it reaches a leaf node.
2. **Expansion.** The selected leaf node is expanded by adding child nodes corresponding to the legal actions from that state.
3. **Evaluation.** The leaf state is evaluated using an evaluation function, and the resulting value is stored at the node.
4. **Backpropagation.** The statistics of the traversed nodes are updated: visit counts are incremented and value estimates are updated using the leaf evaluation.

When this process terminates, the selected action is usually the move corresponding to the child of the root with the highest visit count (Browne et al., 2012). Figure 1 illustrates the four-step MCTS cycle.

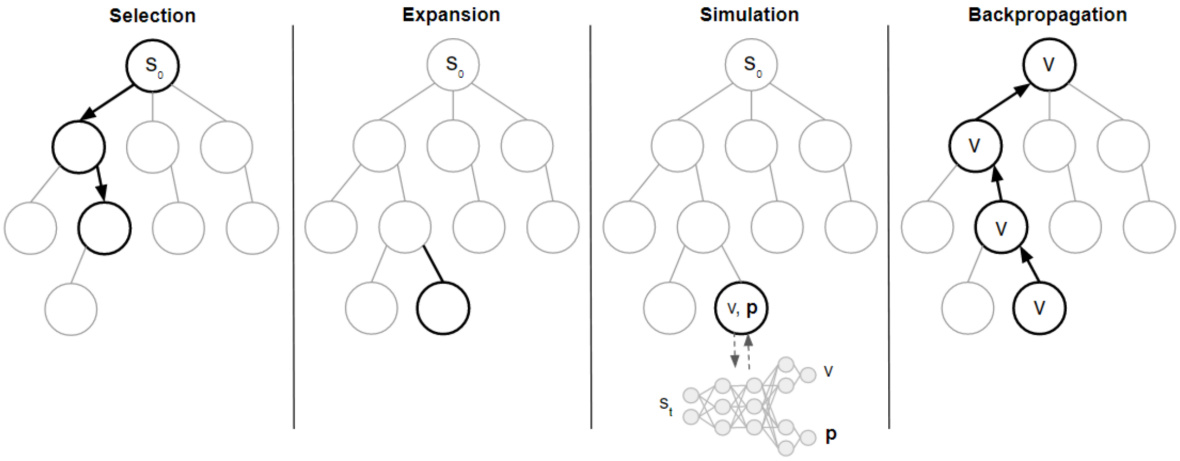

Figure 1: Illustration of the four steps of Monte Carlo tree search.

2.2 PUCT

One implementation of MCTS found in many of the modern game playing engines is Predictor + Upper Confidence bounds applied to Trees (PUCT). PUCT specifies a tree policy for selecting which child node to explore during MCTS. In modern neural-network-based engines, this selection rule is combined with an evaluation function and policy prior produced by a neural network.

The tree policy selects for a node with state s with children (s_i, a_i) the action a_i which maximizes the selection function S. The selection function normally consists of two parts; Q which represents the expected value of an action, and U which is large for less explored actions.

$$S(s,a) = Q(s,a) + U(s,a)$$

The version of PUCT that will be used as a baseline in this paper uses the selection function

$$S(s,a) = Q_a + c_{puct} \cdot \pi(a) \frac{\sqrt{N}}{1+n_a}$$

where Q_a is the expected value of the action, π is the policy function, N the number of visits of the parent node, n_a the number of times the action has been tried from the parent node state and $c_{puct} = \frac{3}{2}$.

In order to further guide the exploration towards interesting areas of the game tree, one could incorporate the variance in evaluation of a state in the exploration term. In chess, humans often put more attention on calculating the sharpest lines, where there might only be one deciding move, compared to the more stable lines. This idea motivates the introduction of the variance-aware PUCT algorithm, which is the central area of our research.

2.3 Variance-Aware PUCT

The tree policy PUCT-V is a variance-aware tree policy that is similar to standard PUCT, but additionally takes into account the estimated variance in the evaluation of a state in its selection formula. Prior work has showed that PUCT-V can outperform PUCT in several bandit and reward tree evaluations (Weichart, 2026). We extend this evaluation to games with a much larger action space, namely chess, and compare PUCT-V's performance in MCTS with standard PUCT. We have implemented PUCT-V according to Weichart. The selection function S takes the form

$$S(s,a) = Q_a + c_1 \cdot \pi(a) \hat{\sigma}_a \frac{\sqrt{N}}{1+n_a} + c_2 \cdot \pi(a) \frac{log(N)}{1+n_a}$$

where Q_a is the expected value of the action, π is the policy function, $\hat{\sigma}_a$ the estimated standard deviation of the value of the action, N the number of visits of the parent node, n_a the number of times the action has been tried from the parent node state, $c_1 = \sqrt{2}$ and $c_2 = 3$. The estimated standard deviation $\hat{\sigma}_a$ is computed as the square root of the sample variance of the evaluations of the nodes in the subtree that has state s_a as its root, where s_a is the state resulting from taking action a. This value is stored at each node, and is initialized as 0 when a new node is added to the tree. Its value is updated in the backpropagation step of MCTS using Welford's online algorithm (Welford, 1962).

3 Method

We have built a puzzle-solving evaluation pipeline using the lczerolens library[1], which provides a PyTorch wrapper around pretrained Leela Chess Zero networks together with a Monte Carlo tree search implementation and utilities for evaluating and analyzing chess puzzles. For each puzzle position, both search methods use the same neural network outputs: a value estimate for the position and policy priors over legal moves. This allows us to isolate the effect of changing the tree policy from PUCT to PUCT-V.

3.1 Networks

We evaluate the search methods using two pretrained Leela-compatible neural networks. First, we use maia-1900[2], a human-like network trained to predict moves by players ranked around 1900 Elo rating. The Maia architecture is relatively small, with 6 residual blocks and 64 filters (McIlroy-Young et al., 2020). This makes it useful for quick experimental iterations, and for testing whether the search methods behave differently with a weaker, human-like policy prior.

Second, we use T1-256x10-distilled[3], a stronger Leela Chess Zero network with 256 filters and 10 residual blocks. We use this network as an approximation of state of the art chess networks, while also limiting our computational costs. The T1-256x10 network is trained via distillation from a stronger network, which in turn is trained using self-play.

3.2 Puzzles

We use the Lichess puzzle dataset through the lczerolens Hugging Face dataset[4]. Each puzzle contains a starting position in FEN notation, a sequence of correct moves, a puzzle rating, and theme tags such as `sacrifice`, `fork`, `endgame`, or `defensiveMove`.

For each puzzle, the engine is placed at the starting position of a puzzle of appropriate difficulty and must choose the correct move sequence under a fixed search budget. In the main experiments, we use puzzles rated above 2100 for maia-1900 and the highest-rated puzzles in the dataset for the stronger T1-256x10 network.

We score each run as *success*, *partial*, or *fail*. A success means that the engine follows the full solution line. A partial result means that it finds only some of the correct moves in the sequence. A fail means that the first engine move is incorrect. We also report a numerical score between 0 and 1, given by the fraction of correct moves in the solution sequence.

3.3 Compute

Experiments with the maia-1900 network were run locally on a laptop with an Apple M2 CPU. Experiments with the T1-256x10 network were run on the COSMOS high-performance computing cluster at LUNARC, Lund University, using NVIDIA A40 GPUs.

4 Results

4.1 Human-like maia-1900 network

We evaluated PUCT-V against the standard PUCT selection rule on a dataset of 200 Lichess puzzles with rating > 2100. We ran the comparison across a range of simulation budgets and report representative numbers at 800 and 3200 simulations per move, illustrating both a moderate and a generous search budget. PUCT-V yielded a lower success rate (full puzzle solved) and a higher failure rate (first move incorrect) than standard PUCT at both budgets, and the gap did not close as the search budget grew.

	800 sims		**3200 sims**	
	PUCT	**PUCT-V**	**PUCT**	**PUCT-V**
Success (%)	58.0	46.5	68.0	59.0
Partial (%)	14.5	19.5	11.5	15.0
Fail (%)	27.5	34.0	20.5	26.0
Mean score	0.642	0.554	0.726	0.657

Table 1: Puzzle outcome rates for vanilla PUCT and PUCT-V on 200 Lichess puzzles with rating > 2100 (range 2101–2991, mean 2345) using maia-1900.

	800 sims	**3200 sims**
PUCT-V wins	5	8
PUCT-V losses	32	27
Ties	163	165

Table 2: Paired puzzle-level comparison of PUCT-V against vanilla PUCT on the same 200-puzzle maia-1900 evaluation set.

[1] https://github.com/Xmaster6y/lczerolens

[2] https://lczero.org/play/networks/sparring-nets/

[3] https://lczero.org/play/networks/bestnets/

[4] https://huggingface.co/datasets/lczerolens/lichess-puzzles

Figure 2 shows how the success rate for both methods changes as the simulation budget increases.

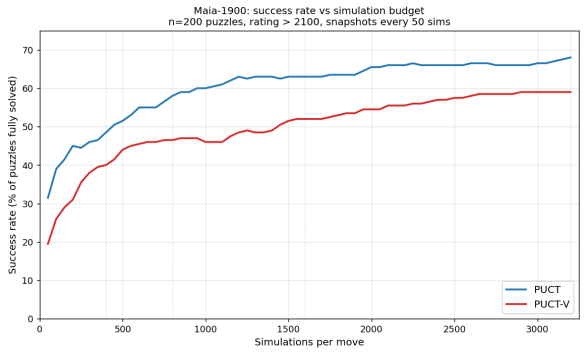

Figure 2: Success rate plotted against number of MCTS iterations per move for maia-1900 on 200 puzzles rated above 2100. PUCT consistently solves more puzzles than PUCT-V, although both methods improve as the search budget increases.

4.2 Strong T1-256x10 network

To test whether the underperformance of PUCT-V was specific to the weaker, human-like maia-1900 network, we repeated the comparison with the stronger Leela network T1-256x10-distilled on the 500 hardest puzzles in the dataset.

Table 3 shows that standard PUCT outperforms PUCT-V across all tested search budgets. Increasing the number of simulations improves both methods, but the gap remains large. At 800 simulations, PUCT solves 64.2% of the puzzles compared to 36.6% for PUCT-V. At 3200 simulations, the success rates increase to 73.2% and 48.2%, respectively. The mean score follows the same pattern, increasing with more search for both methods while remaining consistently higher for PUCT.

The two methods produced identical scores on 330 of the 500 puzzles (for 3200 simulations). Of the 170 divergent puzzles, vanilla PUCT scored higher on 151 and PUCT-V on 19. Table 4 in appendix B breaks down the divergent puzzles by Lichess theme tag, restricted to themes with at least $n = 2$ divergent occurrences. We report the PUCT-V win–loss count and the mean per-puzzle score difference for all themes.

4.3 Qualitative analysis of divergent decisions

To understand the mechanism behind PUCT-V's underperformance on the themes in Table 4, we selected six puzzles from the T1-256x10 evaluation for further analysis: three puzzles where PUCT-V underperformed PUCT, and three where it outperformed.

For each engine-move position, we reran both methods and recorded the root visit counts at simulation checkpoints of 50, 100, 200, 400, and 800 simulations. Since the final move is selected as the legal move with the largest root visit count, these trajectories show how each method allocates search effort over time.

Figure 3 shows a representative example. In puzzle ibcAh, the correct first engine move is the knight sacrifice $e5c4$ (Figure 4 in Appendix A). Vanilla PUCT assigns most of its visits to this move by the end of the search, whereas PUCT-V spreads visits more evenly across several candidate moves and ultimately selects $a6a5$. This suggests that PUCT-V did not concentrate enough simulations on the tactical sacrifice to make its value estimate improve.

	800 sims		**1600 sims**		**3200 sims**	
	PUCT	PUCT-V	PUCT	PUCT-V	PUCT	PUCT-V
Success (%)	64.2	36.6	68.0	42.0	73.2	48.2
Partial (%)	20.2	34.6	18.2	30.8	15.2	27.2
Fail (%)	15.6	28.8	13.8	27.2	11.6	24.6
Mean score	0.724	0.495	0.752	0.536	0.787	0.582
Paired (PUCT-V vs PUCT):						
PUCT-V wins	21		23		19	
PUCT-V losses	170		162		151	
Ties	309		315		330	

Table 3: Puzzle outcomes for vanilla PUCT and PUCT-V on the 500 hardest puzzles (rating range 3034–3331).

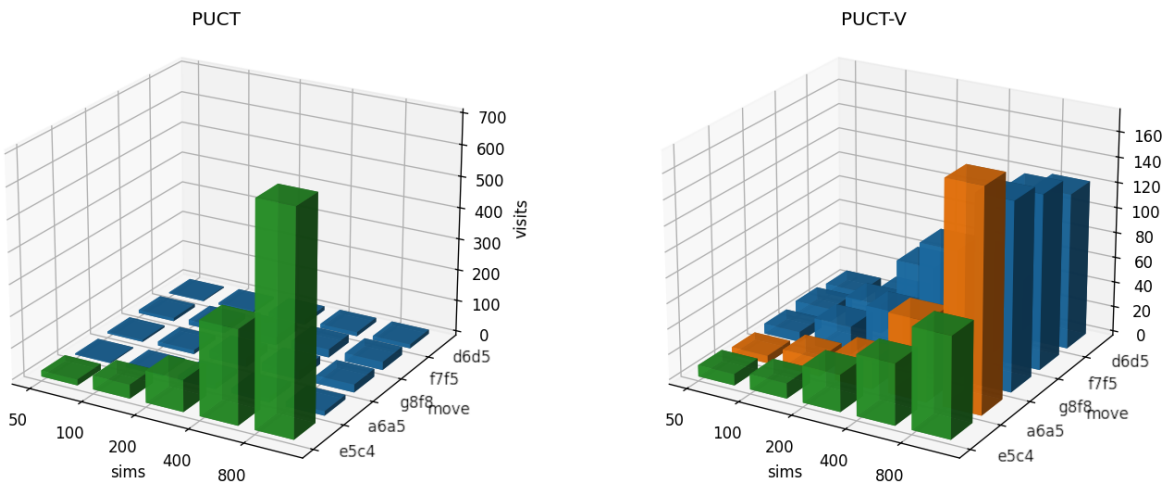

Figure 3: Visit-count evolution on the first engine move of puzzle ibcAh. Left: standard PUCT. Right: PUCT-V. Bars show root visit counts at five checkpoints for the top-5 candidate moves (combined ranking across both methods). The correct move e5–c4 is highlighted in green; PUCT-V's incorrect pick a6–a5 is highlighted in orange.

5 Discussion

Our experiments show that PUCT generally solves more puzzles than PUCT-V for both tested pre-trained networks. We identify several explanations for these results. One possible reason for this is that the pre-trained network T1-256 is distillation-trained from a network trained using the AlphaZero algorithm, which employs PUCT as its tree policy when doing MCTS. This means that the network is conditioned on PUCT, potentially making it better suited for PUCT than PUCT-V. The network maia-1900 is to our knowledge not trained using this method, so this does not explain the full discrepancies in the results. Comparing the difference in success rate between PUCT-V and PUCT for both networks (Tables 1 and 3), we observe that the difference is larger for T1-256 which might be due to this reason.

Another possible explanation for PUCT-V's worse performance is that the variance estimate might need a larger number of search iterations to converge to a meaningful value, since we are using the sample variance. Observing Figure 2 the success rate of PUCT and PUCT-V seems to plateau as the number of search iterations increases, making it uncertain if more simulations will close the gap. Related to this, the estimated variance might in general be too noisy. For any node with a low number of visited children the sample variance will be a very rough estimate of the actual variance, and might disturb the selection more than it improves it.

Another reason might be that chess as a domain is worse suited for PUCT-V compared to PUCT. Chess is a game with a very large action space, which has been a challenge for chess engines for a long time. One of the big advantages of PUCT is that the policy narrows down the search, allowing it to focus resources at the most promising moves. We have observed that PUCT-V usually spreads its search across more moves than PUCT, which potentially could worsen its performance. A representative example of this is shown in Figure 3, where the visit counts of the top candidate moves over the search time are shown for PUCT and PUCT-V respectively.

6 Conclusion and future research

We evaluate PUCT-V as a tree policy in Monte Carlo tree search, as a part of a chess engine comparing it to the classical tree policy PUCT. We implement PUCT-V using the sample variance as an estimate of the variance, and use two different pre-trained neural networks for the move policies and evaluations. We demonstrate that under these conditions PUCT-V performs worse than PUCT at solving chess puzzles, while still performing at a comparable level to PUCT. We hypothesize that the discrepancies in performance can be explained by the neural networks conditioning on PUCT in training, the sample variance being too rough of an estimate in certain situations, and possibly that the large action space of chess might not be suited to PUCT-V.

For future research, two of these hypothesized problems could be addressed by training a new neural network using the AlphaZero algorithm. This network would be trained using PUCT-V as the tree policy in its MCTS, conditioning the network to perform best with PUCT-V. Also, the network could be trained to predict the expected variance in the MCTS evaluation with a certain number of simulations, as suggested by Wu (2020). The network being trained using PUCT-V would make the comparison to PUCT being run with a PUCT-trained network more fair. Using the network's predicted variance instead of the sample variance might result in a better estimate of the variance, and would eliminate the problem of poor variance estimates when the number of visited children of a node is low.

Acknowledgments

We would like to thank Prof. Pierre Nugues for his support and supervision of our project.

References

Cameron B. Browne, Edward Powley, Daniel Whitehouse, Simon M. Lucas, Peter I. Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. 2012. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43.

Murray Campbell, A. Joseph Hoane Jr., and Feng hsiung Hsu. 2002. Deep blue. *Artificial Intelligence*, 134(1–2):57–83.

Reid McIlroy-Young, Siddhartha Sen, Jon Kleinberg, and Ashton Anderson. 2020. Aligning superhuman ai with human behavior: Chess as a model system. *Preprint*, arXiv:2006.01855.

David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. 2017. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *Preprint*, arXiv:1712.01815.

Maximilian Weichart. 2026. Variance-aware prior-based tree policies for Monte Carlo tree search. *Preprint*, arXiv:2512.21648.

B. P. Welford. 1962. Note on a method for calculating corrected sums of squares and products. *Technometrics*, 4(3):419–420.

David J. Wu. 2020. Accelerating self-play learning in go. *Preprint*, arXiv:1902.10565.

A Appendix - Divergent puzzle ibcAh

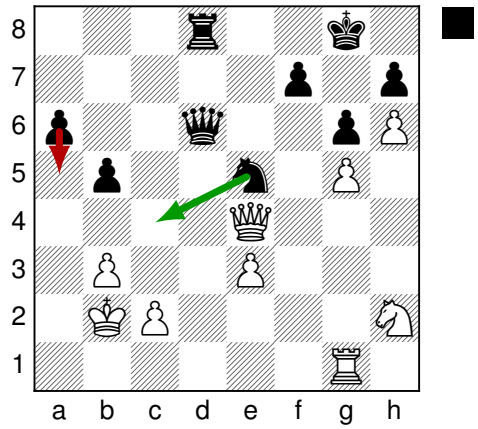

Figure 4: Puzzle ibcAh, black to move. PUCT finds the correct move Nc4, with the main line Nc4 bxc4 bxc4 Qxc4 Qe5. PUCT-V plays a5 instead.

B Appendix - Puzzle theme breakdown

Theme	n	Win–Loss	Δ-score
castling	2	0–2	−1.000
enPassant	4	0–4	−0.938
discoveredAttack	4	0–4	−0.917
promotion	5	0–5	−0.857
sacrifice	24	0–24	−0.765
fork	8	0–8	−0.735
attraction	7	1–6	−0.735
intermezzo	3	0–3	−0.733
rookEndgame	10	0–10	−0.728
long	50	4–46	−0.720
opening	2	0–2	−0.700
middlegame	84	7–77	−0.672
advancedPawn	18	2–16	−0.666
queenEndgame	3	0–3	−0.650
hangingPiece	6	0–6	−0.649
advantage	46	6–40	−0.620
kingsideAttack	11	2–9	−0.613
crushing	124	13–111	−0.594
pawnEndgame	15	1–14	−0.588
deflection	14	1–13	−0.572
short	7	0–7	−0.571
exposedKing	14	2–12	−0.565
pin	19	4–15	−0.553
veryLong	113	15–98	−0.550
master	9	1–8	−0.535
clearance	9	1–8	−0.532
quietMove	58	7–51	−0.528
endgame	84	12–72	−0.528
defensiveMove	35	6–29	−0.494
zugzwang	13	2–11	−0.474
bishopEndgame	5	2–3	−0.320
knightEndgame	3	1–2	−0.197
skewer	2	1–1	−0.075
queensideAttack	2	1–1	0.000

Table 4: Per-theme breakdown of the 170 puzzles where PUCT-V and PUCT produced different scores, themes with $n \geq 2$. T1-256x10 network, 3200 simulations.