Learning Where to Search: Subtree Exploration in Tree-Search

Anonymous authors
Paper under double-blind review

Abstract

In planning with tree search, how search budget is allocated across the tree determines the final decision. Standard Monte Carlo Tree Search (MCTS) allocates this computation implicitly through repeated local tree-traversal decisions that do not globally prioritize reaching the most informative region of the tree to explore next. We introduce Learning to Allocate Search Resources (LASER), a framework that separates *where* to search from *how* to search. Our method uses a graph neural network to explicitly select a node in the current search tree from which to further explore by a bounded local MCTS. We formulate this process as a metalevel decision problem and train the node selector from action-level supervision, without requiring annotations of where search should occur. On Sokoban and sliding puzzles, LASER consistently outperforms standard MCTS and alternative subtree-selection strategies under matched search budgets. We further integrate our approach into EfficientZero by training the selector concurrently with a world model, value function, and policy during reinforcement learning, and observe improved aggregate performance on Atari 100k over baseline search methods.

1 Introduction

> *"The crux in problem solving is selectivity in search."*
>
> *In Search of Insight* (Kaplan & Simon, 1990)

From superhuman game-playing agents (Silver et al., 2016; 2017; Schrittwieser et al., 2020) to recent advances in mathematical reasoning (Hubert et al., 2026), search and planning have been central to progress in artificial intelligence. With additional computation at test time, search enables agents to explore and evaluate alternative decisions before acting. Tree-search methods, particularly Monte Carlo Tree Search (MCTS), provide a general framework for planning by constructing a tree of future outcomes to refine the decision at its root. Under a finite search budget, however, the effectiveness of search for timely decisions depends not only on how much computation is available, but also on how that computation is allocated.

MCTS allocates this budget through recursive traversal of a tree rooted at the current state, where action selection at each node is treated as a multi-armed bandit problem. Selection rules such as UCT balance the exploitation of actions with high estimated value against the exploration of less-visited alternatives (Kocsis & Szepesvári, 2006). Repeated application of these local selections determines which regions of the tree receive further exploration and, consequently, how the overall search budget is distributed.

This allocation mechanism, however, is not directly aligned with the objective of planning. While the bandit formulation underlying conventional selection rules aims to minimize the cumulative regret of not having selected the true optimal action during search, planning ultimately aims to maximize the value of the final action selected at the root. This mismatch motivates a metareasoning perspective, in which computation is selected according to its value for improving the final decision (Hay & Russell, 2011). From this perspective, different regions of the search tree may have substantially different value to explore, raising the question: *given the tree constructed so far, where should the search allocate its next computation?*

We address this question by separating *where to search* from *how to search*. To this end, we propose **L**earning to **A**llocate **SE**arch **R**esources (LASER), which learns to allocate computation across

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

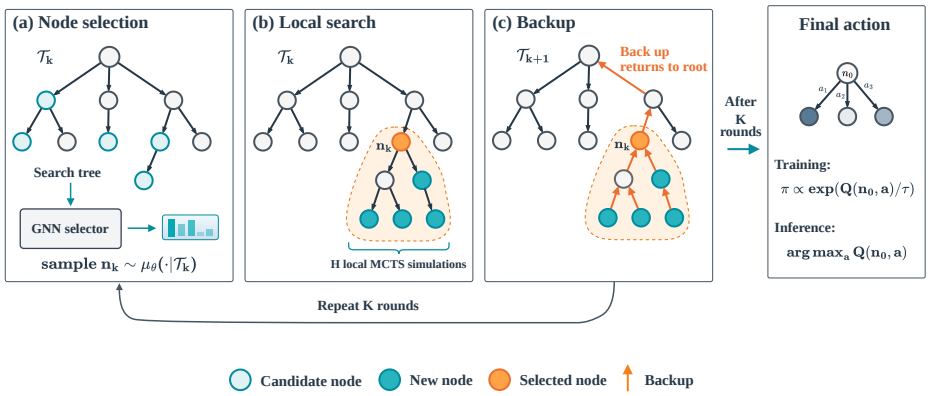


Figure 1: Overview of LASER. (a) A GNN selector processes the current search tree $\mathcal{T}_k$, scores candidate nodes, and samples a starting node n_k. (b) MCTS explores the selected subtree through H simulations from n_k. (c) Each simulation's return is backed up to the global root n_0 and updates search statistics along the branch. This process repeats for K rounds, and the tree and its statistics persist across rounds. The total budget is KH simulations. The final action maximizes the root action-value estimate $Q(n_0, a)$; selector training uses a softmax readout of these values for a smooth objective.

regions of the search tree. Instead of determining search allocation solely through local traversal, LASER uses a global selection mechanism that identifies which region of the current tree should receive further computation. The selected region is subsequently explored using a local search procedure, allowing global compute allocation and local tree exploration to be handled at distinct levels. This formulation makes search allocation an explicit decision over the current tree, rather than an implicit consequence of successive local action selections, as in MCTS.

Concretely, LASER maintains a search tree and learns a global selector that chooses among its expandable nodes, effectively selecting a subtree for further exploration. We parameterize the selector with a graph neural network (GNN) that uses message passing to evaluate candidate nodes based on their local search statistics and context within the evolving tree. Planning alternates between selecting a node and performing a bounded local MCTS within its subtree, with the resulting information backed up to the global root (see Fig. 1). The selector then observes the updated tree and chooses where to allocate the next block of compute. We formulate this iterative process as a metalevel decision problem and train the selector using action-level demonstrations to improve the final decision at the root, without supervision over which subtrees to select.

We evaluate our method in two settings. First, we study planning with pretrained value function and policy on the benchmarks of Sokoban and sliding puzzles to isolate the effect of search. Under the same simulation budget, LASER consistently outperforms standard MCTS and alternative selection strategies, with solve-rate gains over standard MCTS of up to 35.2 percentage points in Sokoban with UCT and an increase from 93.5% to 100% on the 24-puzzle with PUCT. Second, we evaluate LASER as a planning component for reinforcement learning by integrating it into EfficientZero (Ye et al., 2021) on the Atari 100k benchmark. The selector learns through self-imitation from the agent's own experience, without external demonstrations. It achieves the highest mean among the evaluated methods. These gains support the effectiveness of learned search allocation within model-based reinforcement learning, where the selector is trained alongside a world model, a value function, and a policy, and planning helps to shape data collection. Ablation studies show that using a tree context improves the selector's performance over scoring nodes independently.

2 Related Work

Prior work has explored MCTS variants to improve search efficiency and decision quality. UCT balances exploration and exploitation (Kocsis & Szepesvári, 2006). PUCT incorporates policy priors (Rosin, 2011; Silver et al., 2016), and RAVE improves early estimates through shared action statistics (Gelly & Silver, 2011). Other work targets final-action quality through simple-regret

exploration (Tolpin & Shimony, 2012) and policy improvement under small search budgets (Danihelka et al., 2022). More recently, DeepSearch combines global frontier selection with local search for LLM mathematical reasoning (Wu et al., 2026). Its structure resembles ours, but it uses a deterministic heuristic based on local node and parent statistics and assumes terminal-solution verification in the Reinforcement Learning from Verifiable Rewards (RLVR) setting. Our selector is instead a neural model, it uses the entire search tree, and it can be supervised by the root actions of an expert or a reinforcement-learning actor.

Learning to plan includes differentiable architectures such as Value Iteration Networks (Tamar et al., 2016) and TreeQN and ATreeC (Farquhar et al., 2017), which embed planning in policy or value prediction. Other approaches learn to plan through interactions using imaginary actions and resets (Chung et al., 2023), or exhibit planning-like behavior through model-free recurrent computation (Guez et al., 2019). Neural tree-search approaches are particularly close to our learning setup. MCTSnets (Guez et al., 2018) jointly learn evaluation and backup operations and a search policy for tree traversal. D-TSN (Mittal et al., 2025) replaces TreeQN's exhaustive expansion with best-first search, sampling frontier nodes according to their estimated returns and backpropagating root Q-prediction errors to improve its models. Our learning approach is similar to MCTSnets and D-TSN, but targets a policy over search-tree nodes to improve the final root decision. The local MCTS planner and its evaluators remain fixed, and our training scheme must overcome the unavailability of gradients through the black-box, non-differentiable local search procedure.

When no simulator is available, planning can proceed in a learned world model. MuZero searches in learned latent dynamics (Schrittwieser et al., 2020), and EfficientZero improves its sample efficiency (Ye et al., 2021). Our work is orthogonal to the choice of model: given a transition model and fixed evaluators, we learn where additional search computation should be spent at test time.

Selecting promising nodes from a search tree has also been studied in branch-and-bound optimization (He et al., 2014; Labassi et al., 2022; Zhang et al., 2025). These methods aim to accelerate solving by learning which open node to process next, using oracle node preferences or rewards based on solver progress. LASER, in contrast, selects starting nodes for local MCTS to improve the final root decision, learning from expert environment actions without requiring node-selection labels.

3 Background

3.1 Setting and notation

We consider planning in a deterministic environment formulated as an MDP $\mathcal{M} = (S, A, f, r, \gamma)$ with state set S, finite action set A, deterministic transition function $f : S \times A \to S$, reward $r : S \times A \to \mathbb{R}$, and discount $\gamma \in [0, 1)$. Deterministic dynamics is the standard setting for tree-search planners (Silver et al., 2017; Guez et al., 2018; Schrittwieser et al., 2020; Chung et al., 2023; Hubert et al., 2026).

A search tree $\mathcal{T}$ is a set of nodes, each node n associated with a state $s(n) \in S$, a depth $d(n)$ in the tree, and a visitation count $N(n)$. The edges of the tree are previously explored transitions of the environment. Each edge (n, a) to the next-state node n' with $s(n') = f(s(n), a)$ carries a backed-up mean return $Q(n, a)$ and a visitation count $N(n, a)$. We write n_0 for the root, $\mathcal{V}(\mathcal{T})$ for the nodes, $\mathrm{par}(n)$ for the parent of $n \neq n_0$, $\mathcal{T}(n)$ for the subtree rooted at n, and s for $s(n)$ wherever the node is clear from the context.

3.2 Monte Carlo Tree Search

MCTS incrementally builds a search tree by repeated iterations from the root. Each iteration follows a tree-traversal policy until it reaches a node with terminal state or selects an edge not yet been added to the tree. In the latter case, the following state is added as n_L, and its reward-to-go is estimated by a given value function $\hat{v}(s(n_L))$. This estimate is then backed up along the traversed path as

$$G_i = r(s(n_i), a_i) + \gamma G_{i+1}, \tag{1}$$

with $G_L = \hat{v}(s(n_L))$, updating visit counts and mean returns by $Q(n_i, a) \leftarrow Q(n_i, a) + \frac{G_i - Q(n_i,a)}{N(n_i,a)}$.

The tree-traversal policy treats each action selection as a multi-armed bandit problem, balancing exploitation of high-value actions with exploration. UCT (Kocsis & Szepesvári, 2006) selects the

action by maximizing its estimated return plus an exploration bonus that favors less-visited actions:

$$a^* = \operatorname*{argmax}_a \left[Q(n,a) + c_{\text{UCT}} \sqrt{\frac{\log N(n)}{N(n,a)}} \right], \tag{2}$$

PUCT (Rosin, 2011; Silver et al., 2017) additionally guides exploration with a policy prior $P(s,a)$:

$$a^* = \operatorname*{argmax}_a \left[Q(n,a) + c_{\text{PUCT}} P(s,a) \frac{\sqrt{N(n)}}{1 + N(n,a)} \right]. \tag{3}$$

The positive constants c_{UCT} and c_{PUCT} control exploration. After the search budget is exhausted, the final action is selected using the search statistics at the root, typically by maximizing visitation count or estimated action value (Browne et al., 2012).

4 LASER: Learning to Allocate Search Resources

Standard MCTS allocates search resources through a sequence of local action selections starting from the root, making access to any subtree dependent on decisions at its ancestors. These selections rely on local search statistics rather than directly comparing the potential benefit of further computation across the tree. Moreover, UCT-style exploration rules are motivated by cumulative-regret minimization, whereas planning seeks a good final action under a finite search budget (Tolpin & Shimony, 2012). This suggests allocating computation according to how much exploring different regions of the tree could improve the final decision at the root. Subtrees provide a natural unit for this allocation: selecting a node specifies where further search begins, and the subtree rooted at that node defines the continuations to explore. We therefore learn an allocation policy that selects subtrees according to their potential to improve the final root decision.

The main idea behind LASER is to score each node in the tree based on how promising it is to explore that node further. To this end, we split the available test-time compute budget across K local searches of H iterations each. At the beginning of each local search, a neural network processes the entire current search tree and scores candidate nodes based on their search statistics and context within the tree. These scores define a distribution from which we sample the starting point of the local search. We then run MCTS from this node, backing up the resulting returns to the global root. The tree persists across searches, so that later rounds build on the statistics accumulated by earlier ones. The selector's scoring network is trained end-to-end to improve the final action selected at the global root.

4.1 Subtree exploration

Concretely, LASER starts from a tree $\mathcal{T}_1$ containing only the root n_0 and performs the following steps at each round $k = 1, \ldots, K$:

1. a *selector* chooses a node n_k from the candidate set $\mathcal{F}(\mathcal{T}_k)$ (defined below);
2. the local search runs H simulations rooted at n_k, expanding $\mathcal{T}_k$ into $\mathcal{T}_{k+1}$; and
3. the simulated returns are backed up along the full branch to the original root n_0, updating search statistics along the way.

The total budget is therefore KH simulations, split evenly across K local searches. With $K = 1$, the local search starts at the original root and recovers standard MCTS up to the readout.

MCTS as the local search. We use MCTS with UCT or PUCT for action selection. The local search procedure is unchanged, except that the standard MCTS backup rule (1) is applied along the extended branch to the original root n_0, rather than stopping at the local-search root n_k. Thus, computation allocated anywhere in the tree updates the root action values used for the final decision.

Candidate set. A node is eligible for selection if it is already present in the tree, has a nonterminal state, and has at least one unvisited action:

$$\mathcal{F}(\mathcal{T}) = \{\, n \in \mathcal{V}(\mathcal{T}) : \; s(n) \text{ is non-terminal}, \; \exists a \text{ unvisited at } n \,\}.$$

Candidates can include both leaves and *partially explored* internal nodes. Selecting an internal node allocates additional computation to a subtree that already contains search statistics and explored branches. Once all actions at a node have been visited, that node is excluded from the candidate set, although its descendants may remain eligible.

Policy readout. After the search terminates, LASER selects the root action with the highest backed-up value, $\operatorname{argmax}_a Q(n_0, a)$. We use Q instead of the more commonly used root visitation counts (Silver et al., 2017; Schrittwieser et al., 2020; Yu et al., 2025), since in our method these counts reflect *where* the selector allocated computation rather than what it found.

For training, we replace the hard argmax with a softmax at temperature $\tau > 0$:

$$\pi(a \mid s_0) = \frac{\exp\big(Q(n_0, a)/\tau\big)}{\sum_{a'} \exp\big(Q(n_0, a')/\tau\big)}.$$

This relaxation allows the training objective to respond smoothly to changes in relative root Q-values, so search allocations receive a learning signal even when the maximizing action remains unchanged. As $\tau \to 0$, the policy recovers the greedy action. Since a limited search budget may leave some root actions unvisited, in practice, we use completed Q-values (Danihelka et al., 2022) to define the readout over all actions, as described in Appendix B.

4.2 Learning the selector

We train LASER's selector using expert demonstrations $\mathcal{D} = \{(s_i, a_i^*)\}_{i=1}^N$. Although this formulation assumes access to expert data, it requires only action-level demonstrations, without the annotations of which subtrees to explore that some prior work requires (He et al., 2014; Labassi et al., 2022). Existing planners can therefore supply the training targets, for example by running search with a larger computational budget at training time. Our experiments further show that a relatively small demonstration dataset suffices to train the selector. Additionally, we present a strategy that instead constructs training targets from the agent's highest-performing past interactions. Details are provided in Section 5.

Given a state s and an expert action a^*, we train the selector to allocate search so that the resulting policy assigns high probability to a^* in expectation. We therefore minimize the expected cross-entropy between the expert and the search-induced policy:

$$\ell(s, a^*) = -\,\mathbb{E}[\log \pi(a^* \mid s)].$$

The expectation is over the choice of starting nodes sampled by the selector and any randomness in local search. Denoting the selector by μ_θ, we seek parameters θ that minimize the above loss. Since the mapping from selector decisions to the search-induced policy π involves sampling discrete nodes and is thus non-differentiable, we estimate the gradient using REINFORCE (Williams, 1992):

$$\nabla_\theta \ell(s, a^*) = -\,\mathbb{E}\left[\sum_{k=1}^{K} \nabla_\theta \log \mu_\theta(n_k \mid \mathcal{T}_k) \log \pi(a^* \mid s)\right].$$

To understand how individual allocation decisions contribute to the objective, let π_k denote the policy read out from $\mathcal{T}_k$, before round k, and define its loss as $\ell_k = -\log \pi_k(a^* \mid s)$. After K rounds, the final loss is ℓ_{K+1}, whose expectation gives the training objective $\ell(s, a^*)$. We can decompose this loss into the changes produced by successive local searches:

$$\ell_{K+1} = \ell_1 + \sum_{k=1}^{K} (\ell_{k+1} - \ell_k).$$

The initial loss ℓ_1 is computed from root action values initialized independently of θ. Minimizing the expected final loss is therefore equivalent to maximizing the expected sum of terms $\ell_k - \ell_{k+1}$. Following Guez et al. (2018), we then define the reward for round k as

$$r_k = \ell_k - \ell_{k+1} = \log \pi_{k+1}(a^* \mid s) - \log \pi_k(a^* \mid s).$$

This yields a finite-horizon metalevel decision problem, where states are search trees, actions select starting nodes, and transitions execute H local-search simulations. The reward r_k measures the

reduction in loss after exploring the subtree rooted at n_k in $\mathcal{T}_k$. Using the reward-to-go instead of the total loss reduces the variance of the REINFORCE gradient estimator (Greensmith et al., 2004):

$$\nabla_\theta \ell(s, a^*) = -\,\mathbb{E}\left[\sum_{k=1}^{K} \nabla_\theta \log \mu_\theta(n_k \mid \mathcal{T}_k)\, R_k\right], \qquad R_k = \sum_{j=k}^{K} r_j = \ell_k - \ell_{K+1}.$$

As in Guez et al. (2018), we introduce a discount factor $\gamma_c \in [0, 1]$ to trade off bias and variance. Discounting emphasizes near-term loss reductions, since later improvements depend on additional selections and are harder to attribute to the current decision:

$$\hat{g}_{\gamma_c} = -\sum_{k=1}^{K} \nabla_\theta \log \mu_\theta(n_k \mid \mathcal{T}_k)\, R_k^{\gamma_c}, \qquad R_k^{\gamma_c} = \sum_{j=k}^{K} \gamma_c^{\,j-k} r_j. \tag{4}$$

The estimator is unbiased when $\gamma_c = 1$; smaller values can reduce variance but introduce bias. Finally, we train the selector by updating its parameters θ using the gradient estimator in (4).

Reward Interpretation. To gain intuition for the objective, we plug the induced policy into the reward and rearrange terms. Writing $Q_k(a)$ for the root action-value estimate $Q(n_0, a)$ in $\mathcal{T}_k$, we have

$$r_k = \frac{Q_{k+1}(a^*) - Q_k(a^*)}{\tau} + \log \frac{\sum_a \exp\big(Q_k(a)/\tau\big)}{\sum_a \exp\big(Q_{k+1}(a)/\tau\big)}.$$

When the selected node is below the root, local search can change only the root action value associated with its branch. If the node lies in the subtree corresponding to a^*, increasing that action's estimated value yields positive reward via the first term. Conversely, if it lies in a competing subtree, decreasing the corresponding action's value yields positive reward via the second term. Thus, the selector is rewarded for directing search to support a^* or refute competing alternatives. When the root is selected, multiple values may change, and the reward depends on their combined effect.

Architecture. We parameterize the selector with a graph neural network (GNN) to learn allocation decisions from the evolving search tree. Through message passing, the network combines node features with structural information from the tree and produces a scalar score $z_\theta(n; \mathcal{T})$ for each node. Normalizing these scores over eligible nodes defines the selection policy:

$$\mu_\theta(n \mid \mathcal{T}) \propto \exp\big(z_\theta(n; \mathcal{T})\big), \qquad n \in \mathcal{F}(\mathcal{T}).$$

The graph representation and architecture details are provided in Appendix C.2.

5 Experiments

Our experiments examine whether learning where to search improves decisions under a limited test-time simulation budget. We study LASER in two settings: planning with pretrained components and incorporating planning into model-based reinforcement learning training. The first scenario evaluates search using an exact simulator and fixed evaluators; the second evaluates the selector in a reinforcement learning setting where the selector, dynamics model, value function, and policy are trained concurrently on experience collected using LASER, and the value-function targets use planning-derived value estimates.

5.1 Experimental Setup

We evaluate on Sokoban, sliding puzzles, and Atari 100k. These domains are established benchmarks for evaluating search and planning in combinatorial puzzles (Guez et al., 2019; Agostinelli et al., 2019) and sample-efficient model-based reinforcement learning (Ye et al., 2021). Detailed descriptions of the environments are provided in Appendix A.1.

In Sokoban and sliding puzzles, we use the environment simulator to isolate search performance from world modeling. The value function and, where applicable, policy prior are pretrained and remain fixed during selector training and evaluation (Appendix A.2). For Sokoban, we save policy checkpoints $C_{1:4}$ at four successive stages of reinforcement learning and train a value function for

Table 1: Solve rates (%) on Sokoban and sliding puzzles. All methods use 50 simulations per decision. Results are means over three seeds with 95% confidence intervals, except for MCTS and DeepSearch under UCT, which are deterministic in this setting and evaluated once.

	UCT				PUCT			
Environment	MCTS	Random	DeepSearch	LASER	MCTS	Random	DeepSearch	LASER
Sokoban, C_1	18.1	$30.3_{\pm 0.4}$	18.0	$\mathbf{43.2}_{\pm 0.7}$	$41.7_{\pm 0.1}$	$47.1_{\pm 0.5}$	$50.3_{\pm 0.1}$	$\mathbf{54.8}_{\pm 0.1}$
Sokoban, C_2	18.7	$35.8_{\pm 0.1}$	19.1	$\mathbf{47.6}_{\pm 0.2}$	$57.9_{\pm 0.1}$	$62.5_{\pm 0.7}$	$66.4_{\pm 0.2}$	$\mathbf{69.2}_{\pm 0.1}$
Sokoban, C_3	18.9	$40.7_{\pm 0.2}$	19.4	$\mathbf{54.1}_{\pm 0.9}$	$74.4_{\pm 0.2}$	$75.7_{\pm 0.7}$	$79.6_{\pm 0.3}$	$\mathbf{81.3}_{\pm 0.1}$
Sokoban, C_4	21.6	$41.7_{\pm 0.2}$	17.9	$\mathbf{56.8}_{\pm 0.9}$	$83.9_{\pm 0.3}$	$83.1_{\pm 0.9}$	$85.8_{\pm 0.2}$	$\mathbf{87.2}_{\pm 0.2}$
15-puzzle	86.8	$0.0_{\pm 0.0}$	3.2	$\mathbf{92.5}_{\pm 0.5}$	$94.7_{\pm 0.1}$	$99.3_{\pm 0.1}$	$96.0_{\pm 0.6}$	$\mathbf{99.9}_{\pm 0.1}$
24-puzzle	84.8	$0.0_{\pm 0.0}$	0.0	$\mathbf{89.4}_{\pm 0.0}$	$93.5_{\pm 0.3}$	$45.3_{\pm 1.1}$	$88.4_{\pm 0.4}$	$\mathbf{100.0}_{\pm 0.0}$
35-puzzle	81.0	$0.0_{\pm 0.0}$	0.0	$\mathbf{90.1}_{\pm 2.0}$	$96.9_{\pm 0.4}$	$7.7_{\pm 0.3}$	$64.3_{\pm 0.4}$	$\mathbf{99.4}_{\pm 0.1}$

each checkpoint to examine how evaluator quality affects learned search allocation. In sliding puzzles, we train the value function via value iteration and learn the policy prior with behavior cloning. The policy prior and selector use the same demonstrations to test whether learned allocation improves policy-guided search without additional expert data. Appendix A.3 describes how we obtain expert demonstrations, Appendix B details the MCTS implementation, and Appendix C.3 describes the selector training procedure.

In Atari, we integrate LASER into EfficientZero (Ye et al., 2021), which jointly learns a world model, a policy, and a value function through reinforcement learning. It performs MCTS with the learned model and uses the policy and value function to guide search. This setting evaluates our method when the components used for planning are learned alongside the selector. Without external expert demonstrations, we train the selector through self-imitation using actions from the agent's best-performing episodes collected so far. Details of selector training in this setting are provided in Appendix C.1 and EfficientZero integration details are in Appendix D.4.

We evaluate planning with both UCT and PUCT in both puzzle domains. UCT uses value estimates and visitation statistics (Eq. (2)), while PUCT additionally uses a pretrained policy to guide the search (Eq. (3)). In most experiments, we use $K = 5$ local searches of $H = 10$ simulations each, totaling 50 simulations per decision. We report puzzle solve rates and aggregate human-normalized Atari scores. We run each experiment with three independent random seeds for Sokoban and sliding puzzles and four for Atari 100k. We report mean performance with 95% confidence intervals.

5.2 Baselines

We compare LASER with standard MCTS, random subtree selection, and heuristic subtree selection under matched simulation budgets. Standard MCTS starts each simulation at the original root. The random and heuristic selectors use the same subtree-search structure as LASER but replace its learned selector. The random selector chooses a candidate node uniformly. The heuristic selector, which is based on DeepSearch (Wu et al., 2026), chooses the candidate node with the highest heuristic score:

$$h(n) = \lambda_1 \tanh(Q(\mathrm{par}(n), a)) + \lambda_2 \mathcal{H}(\pi(\cdot \mid s(\mathrm{par}(n)))) + \lambda_3 \sqrt{\frac{d(n)}{D_{\max}}}.$$

Here, a is the action that led to n, $\mathcal{H}$ denotes policy entropy, and $d(n)/D_{\max}$ is the normalized node depth. We tune the coefficients $\lambda_{1:3}$ separately for each setting, as described in Appendix A.4.

5.3 Planning with Pretrained Evaluators

Table 1 reports solve rates across the puzzle domains. LASER significantly outperforms all tested baselines under both UCT and PUCT selection rules. The gains are particularly larger under UCT, where search has no policy prior. PUCT provides stronger guidance through the pretrained policy and leaves less room for improvement. The learned selector nevertheless provides further gains, even with the strongest pretrained evaluator.



Figure 2: Aggregate human-normalized scores across the 26 Atari 100k games, with 95% stratified-bootstrap confidence intervals over four seeds.

DeepSearch performs poorly under UCT, particularly in sliding puzzles. One possible explanation is its sensitivity to value-function errors. Its deterministic heuristic can repeatedly favor subtrees with overestimated values and commit substantial computation to them. Random selection distributes search uniformly and outperforms DeepSearch in Sokoban under UCT. Under PUCT, however, DeepSearch generally performs better than random selection, possibly because the prior improves local search, thereby alleviating some of the systematic overestimation that can trap DeepSearch.

Random selection generally outperforms standard MCTS in Sokoban. These gains support the benefit of the search structure shared by random selection and LASER, even without learning the selector in some domains. However, random selection's weaker performance on sliding puzzles indicates that this structure alone may not be sufficient for consistent improvements across different tasks. Within the same search structure, the learned selector consistently outperforms both random and heuristic selection. These additional gains support the benefit of learning where to allocate computation.

We further investigate the selector's preferences in Appendix C.5. The analysis suggests that shallower nodes are preferred in sliding puzzles, but uniform random selection is more likely to choose deeper nodes, which might explain random selection's weaker performance under UCT. To illustrate how these preferences affect search allocation, Appendix D.3 compares the search trees formed by LASER and standard MCTS in Sokoban.

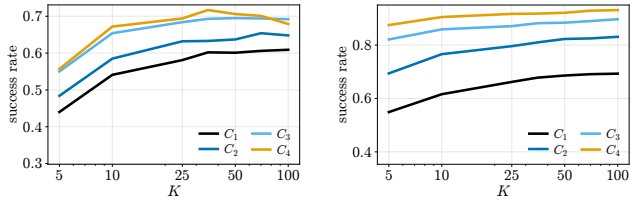
In sliding puzzles, the PUCT policy prior is trained by behavior cloning on the same expert demonstrations used to train the selector. One might expect that distilling these demonstrations into a policy prior would be sufficient to guide search. Yet learning a separate selector from the very same data improves planning further. The demonstrations offer more than a policy to follow: they also provide a basis for learning how to allocate computation beyond that policy's guidance.

5.4 Planning in Model-Based Reinforcement Learning

Fig. 2 summarizes performance across the 26 Atari 100k games using human-normalized scores. We report the mean, median, interquartile mean (IQM), and optimality gap from human-level performance (lower is better) with 95% stratified-bootstrap confidence intervals following Agarwal et al. (2021). Per-game scores are reported in Appendix D.4. LASER achieves the highest mean, median, and IQM among the evaluated methods. Its improvement over MCTS is most pronounced in the mean, with smaller gains, within statistical error, in the median and IQM and a comparable optimality gap. DeepSearch and Random both underperform in terms of IQM and optimality gap relative to MCTS. These results indicate that replacing root-initiated search with subtree search does not consistently improve performance without an effective allocation rule. The benefits of a selector may also depend on the quality of the agent's self-imitation data. When the agent rarely discovers successful behavior, its episodes provide limited guidance for selector training. As more successful episodes become available, the selector can learn to direct search toward actions that produced better outcomes. This dependence offers a possible explanation for the uneven gains across games, where gains tend to be larger when MCTS already performs well, and thus for the larger improvement in the mean than in the IQM.

5.5 Analysis and Ablations

To assess the importance of tree context, Appendix C.4 compares alternative selector architectures: independent node scoring does not perform as well as the GNN in Sokoban; while augmenting the GNN with state observation embeddings yields no consistent gains. Exploration within local search also affects selector learning. Appendix D.1 shows that adding noise under PUCT generally benefits



(a) UCT (b) PUCT

Figure 3: Test-time scaling in Sokoban: increasing K beyond its training value of 5, with fixed H and no retraining.

Table 2: Planning time per environment decision in milliseconds.

Environment	MCTS (ms)	LASER (ms)
Sokoban	21.0	18.7
15-puzzle	14.6	14.0
24-puzzle	14.7	13.3
35-puzzle	14.6	10.3
Atari	74.2	80.5

LASER more than standard MCTS, suggesting that making search outcomes more responsive to node selection provides a more informative learning signal.

Test-time scaling. We evaluate test-time scaling in Sokoban by increasing the number of local searches K while keeping the number of simulations per local search H fixed. Fig. 3 reports solve rates for a selector trained with $K = 5$ and evaluated with larger values of K without further training. Increasing K exposes the selector to larger search trees and potentially different feature distributions from those encountered during training. Despite this mismatch, more rounds of local search generally improve performance, demonstrating that LASER can benefit from additional computation at test time without retraining.

To investigate how best to divide computation between selecting starting nodes and searching from them, we vary K and H under a fixed budget of $KH = 50$ in Appendix D.2. PUCT performs best at $K = 5$ or 10, which balances selection frequency and local search effort. UCT favors $K = 25$, suggesting that more frequent subtree selection may be beneficial without policy guidance.

Planning cost. Table 2 reports the planning time per environment decision for standard MCTS and LASER. Overall, the two methods have comparable latency. Our method is slightly faster in Sokoban and sliding puzzles but somewhat slower on average in Atari. Starting local searches in deeper nodes avoids repeatedly traversing the path from the original root to the selected subtree. This can reduce traversal time, particularly with smaller action sets where candidate nodes tend to be deeper in the tree. However, it incurs overhead from processing the tree and evaluating the selector network K times per decision. Empirically, LASER achieves its performance gains with planning times comparable to those of standard MCTS.

6 Conclusion

We introduced LASER, which learns to allocate search computation across subtrees to improve the final decision at the root. A GNN selector processes the search tree and its statistics to choose where local MCTS searches begin, and their returns are backed up to the global root. The selector is trained using either expert demonstrations or the best of the learning agent's own interaction, with no annotations of which subtrees to explore. Experiments on Sokoban and sliding puzzles show improvements over standard MCTS and alternative allocation rules under matched simulation budgets, both with and without policy priors. Within EfficientZero, LASER also improves aggregate Atari 100k performance through self-imitation from the agent's own experience.

One limitation of LASER is that each local search uses a fixed simulation budget, raising the opportunity to allocate compute unevenly depending on the potential gain for each subtree. Moreover, selector training depends on the quality of action-level supervision, and self-imitation may provide limited guidance when the agent rarely discovers successful behavior. Future work could address these limitations by jointly learning subtree selection and local-search budgets and exploring a curriculum that gradually increases task difficulty as the agent acquires successful experience for selector training.

AI USE STATEMENT

AI tools were used to assist with revising the manuscript and improving its clarity and presentation. They were also used as coding assistants. All code was reviewed by the authors.

REFERENCES

Rishabh Agarwal, Max Schwarzer, Pablo Samuel Castro, Aaron C Courville, and Marc Bellemare. Deep reinforcement learning at the edge of the statistical precipice. *Advances in neural information processing systems*, 34:29304–29320, 2021.

Forest Agostinelli, Stephen McAleer, Alexander Shmakov, and Pierre Baldi. Solving the rubik's cube with deep reinforcement learning and search. *Nature Machine Intelligence*, 1(8):356–363, 2019.

Marc G Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of artificial intelligence research*, 47:253–279, 2013.

Shaked Brody, Uri Alon, and Eran Yahav. How attentive are graph attention networks? *arXiv preprint arXiv:2105.14491*, 2021.

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

Kyunghyun Cho, Bart Van Merriënboer, Çağlar Gulçehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder–decoder for statistical machine translation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pp. 1724–1734, 2014.

Stephen Chung, Ivan Anokhin, and David Krueger. Thinker: Learning to plan and act. *Advances in Neural Information Processing Systems*, 36:22896–22933, 2023.

Ivo Danihelka, Arthur Guez, Julian Schrittwieser, and David Silver. Policy improvement by planning with gumbel. In *International Conference on Learning Representations*, 2022.

Lasse Espeholt, Hubert Soyer, Remi Munos, Karen Simonyan, Vlad Mnih, Tom Ward, Yotam Doron, Vlad Firoiu, Tim Harley, Iain Dunning, et al. Impala: Scalable distributed deep-rl with importance weighted actor-learner architectures. In *International conference on machine learning*, pp. 1407–1416. PMLR, 2018.

Gregory Farquhar, Tim Rocktäschel, Maximilian Igl, and Shimon Whiteson. Treeqn and atreec: Differentiable tree-structured models for deep reinforcement learning. *arXiv preprint arXiv:1710.11417*, 2017.

Sylvain Gelly and David Silver. Monte-carlo tree search and rapid action value estimation in computer go. *Artificial Intelligence*, 175(11):1856–1875, 2011.

Evan Greensmith, Peter L Bartlett, and Jonathan Baxter. Variance reduction techniques for gradient estimates in reinforcement learning. *Journal of Machine Learning Research*, 5(Nov):1471–1530, 2004.

Arthur Guez, Théophane Weber, Ioannis Antonoglou, Karen Simonyan, Oriol Vinyals, Daan Wierstra, Rémi Munos, and David Silver. Learning to search with mctsnets. In *International conference on machine learning*, pp. 1822–1831. PMLR, 2018.

Arthur Guez, Mehdi Mirza, Karol Gregor, Rishabh Kabra, Sébastien Racanière, Théophane Weber, David Raposo, Adam Santoro, Laurent Orseau, Tom Eccles, et al. An investigation of model-free planning. In *International conference on machine learning*, pp. 2464–2473. PMLR, 2019.

Nicholas Hay and Stuart J Russell. Metareasoning for monte carlo tree search. Technical report, Technical Report UCB/EECS-2011-119, EECS Department, University of California, Berkeley, 2011.

He He, Hal Daumé, and Jason Eisner. Learning to search in branch and bound algorithms. *Advances in neural information processing systems*, 27, 2014.

Thomas Hubert, Rishi Mehta, Laurent Sartran, Miklós Z Horváth, Goran Žužić, Eric Wieser, Aja Huang, Julian Schrittwieser, Yannick Schroecker, Hussain Masoom, et al. Olympiad-level formal mathematical reasoning with reinforcement learning. *Nature*, 651(8106):607–613, 2026.

Lukasz Kaiser, Mohammad Babaeizadeh, Piotr Milos, Blazej Osinski, Roy H Campbell, Konrad Czechowski, Dumitru Erhan, Chelsea Finn, Piotr Kozakowski, Sergey Levine, et al. Model-based reinforcement learning for atari. *arXiv preprint arXiv:1903.00374*, 2019.

Craig A Kaplan and Herbert A Simon. In search of insight. *Cognitive psychology*, 22(3):374–419, 1990.

Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *European conference on machine learning*, pp. 282–293. Springer, 2006.

Abdel Ghani Labassi, Didier Chételat, and Andrea Lodi. Learning to compare nodes in branch and bound with graph neural networks. *Advances in neural information processing systems*, 35: 32000–32010, 2022.

Dixant Mittal, Liwei Kang, and Wee Sun Lee. Learning to search from demonstration sequences. In *The Thirteenth International Conference on Learning Representations*, 2025.

Christopher D Rosin. Multi-armed bandits with episode context. *Annals of Mathematics and Artificial Intelligence*, 61(3):203–230, 2011.

Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, et al. Mastering atari, go, chess and shogi by planning with a learned model. *Nature*, 588(7839):604–609, 2020.

David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.

David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *nature*, 550(7676):354–359, 2017.

Aviv Tamar, Yi Wu, Garrett Thomas, Sergey Levine, and Pieter Abbeel. Value iteration networks. *Advances in neural information processing systems*, 29, 2016.

David Tolpin and Solomon Shimony. Mcts based on simple regret. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 26, pp. 570–576, 2012.

Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3):229–256, 1992.

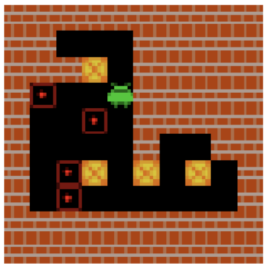
Fang Wu, Weihao Xuan, Heli Qi, Aaron Tu, Ximing Lu, Li Erran Li, and Yejin Choi. Deepsearch: Overcome the bottleneck of reinforcement learning with verifiable rewards via tree-based search. In *The Fourteenth International Conference on Learning Representations*, 2026.

Weirui Ye, Shaohuai Liu, Thanard Kurutach, Pieter Abbeel, and Yang Gao. Mastering atari games with limited data. *Advances in neural information processing systems*, 34:25476–25488, 2021.

Xiao Yu, Baolin Peng, Vineeth Vajipey, Hao Cheng, Michel Galley, Jianfeng Gao, and Zhou Yu. Exact: Teaching ai agents to explore with reflective-mcts and exploratory learning. In *International Conference on Learning Representations*, volume 2025, pp. 65157–65184, 2025.

Sijia Zhang, Shuli Zeng, Shaoang Li, Feng Wu, and Xiangyang Li. Learning to select nodes in branch and bound with sufficient tree representation. In *The Thirteenth International Conference on Learning Representations*, 2025.

Albert L Zobrist. A new hashing method with application for game playing. *ICGA Journal*, 13(2): 69–73, 1990.



(a) Sokoban

1	2	18	10	3
13	16	15	5	9
6	4		24	8
11	22	19	17	7
12	21	23	14	20

(b) Sliding puzzle



(c) Atari

Figure 4: Illustrations of the environments used in our experiments: (a) an example Sokoban level, (b) 24-tile sliding puzzle, and (c) a subset of Atari 100k games.

A Experiment Details

A.1 Environments

Sokoban. Sokoban is a puzzle game in which the player pushes boxes onto designated targets. Boxes can only be pushed, never pulled. The player must therefore plan carefully to avoid irreversible deadlocks that prevent the level from being solved. We use Boxoban levels (Guez et al., 2019) rendered on a 10×10 grid with four boxes and four targets. Rewards follow the standard Boxoban convention: -0.01 per step, $+1$ for pushing a box onto a target, -1 for pushing a box off a target, and $+10$ for solving the level.

Sliding puzzle. The sliding puzzle requires rearranging numbered tiles into a target configuration. Each move slides a tile into an adjacent blank space. A state is a $d \times d$ grid containing a permutation of $0, \ldots, d^2 - 1$, where 0 denotes the blank. In the goal configuration, tiles appear in row-major sorted order, with the blank in the bottom-right corner. The four actions move the blank up, right, down, or left, subject to the grid boundaries. Each move incurs a reward of -1, with no terminal bonus.

Atari 100k. Atari 100k (Kaiser et al., 2019) is a benchmark for sample-efficient reinforcement learning that uses a subset of 26 Atari games from the Arcade Learning Environment (ALE) (Bellemare et al., 2013). For each game, the agent must learn to play from pixel observations with a budget of 100k environment interactions. With an action repeat of four, this budget corresponds to 400k emulator frames. The games span diverse control and decision-making challenges and test agents' ability to learn effective policies from limited experience.

A.2 Pretraining Value Functions and Policy Priors

Sokoban We train an IMPALA agent (Espeholt et al., 2018) from scratch on grid observations for 5×10^8 environment steps, following the setup described by Chung et al. (2023). We save four checkpoints at 10M, 20M, 90M, and 500M steps and denote them by C_1–C_4, respectively. Sampling actions directly from these policies, without search, yields solve rates of 28.0%, 50.1%, 71.1%, and 83.1% on the evaluation pool.

For each checkpoint, we train a separate value network through regression on Monte Carlo rollouts. We collect rollouts using ε-greedy mixtures of the saved policy, for $\varepsilon \in \{0, 0.33, 0.66, 1\}$, and use the realized discounted returns-to-go as regression targets. This data collection broadens state coverage beyond trajectories generated by the checkpointed policy alone, and the value networks cover out-of-distribution values of the IMPALA critic that has only trained on its policy rather than for being accurate for the search.

Data for value-function and selector training are drawn from the 900,000-level unfiltered Boxoban training split. Evaluation uses a fixed pool of 1,000 levels from the held-out test split.

Sliding puzzles We learn a cost-to-go function using deep approximate value iteration (Agostinelli et al., 2019). The network is trained to minimize Bellman errors and consists of 4-blocks of residual MLPs with batch normalization.

Training data is generated by applying m random moves to the goal state, where m is sampled uniformly from $\{1, \ldots, M\}$ with $M = 500$. Evaluation puzzles are generated using the same procedure with $M = 1000$. Here, M bounds the number of scrambling moves, which is not necessarily equal to the shortest solution length. We train the value function until MCTS@50 reaches approximately 80% solve rate on validation puzzles. Further details of the approximate value iteration procedure are given by Agostinelli et al. (2019).

For the PUCT experiments, we train a policy network to provide the prior $P(s, a)$ using the standard behavior-cloning objective. The policy network is a smaller version of the residual MLP used for the value function. Policy training uses exactly the same expert demonstration dataset as selector training.

A.3 Expert Data

Sokoban We obtain expert demonstrations from a DRC agent (Guez et al., 2019) trained separately for 10^8 environment steps. The agent is evaluated on 20,000 levels from the unfiltered Boxoban training split, and each visited state is recorded together with the action taken.

Sliding puzzles We combine 500 published puzzles and their solutions from Agostinelli et al. (2019) with 500 additional solved trajectories generated via MCTS@200 from sampled scrambled states. Episodes that do not reach the goal are discarded.

A.4 Baseline Hyperparameter Tuning

The coefficients of the DeepSearch heuristic function are tuned separately for each setting via grid search. We fix $\lambda_1 = 1$, vary λ_2 and λ_3, and evaluate each configuration on a small set of task instances. The best-performing coefficients are $(\lambda_1, \lambda_2, \lambda_3) = (1, 0, -4)$ for both Sokoban and sliding puzzles, except for Sokoban under PUCT, where $(1, -0.5, -4)$ performs best. For Atari 100k, grid search is not feasible, so we use the default coefficients reported by Wu et al. (2026): $(\lambda_1, \lambda_2, \lambda_3) = (0.4, 0, 0.01)$.

B MCTS implementation

Algorithm 1 provides pseudocode for our MCTS implementation. We employ a transposition table keyed by state hashes. In Sokoban, states are encoded by the positions of the agent and boxes. In sliding puzzles, we use Zobrist hashing (Zobrist, 1990).

To keep the value and exploration terms on comparable scales across environments, we apply min–max normalization following MuZero (Schrittwieser et al., 2020). The minimum and maximum backed-up values, $q_{\min}$ and $q_{\max}$, are tracked to normalize action values for tree selection as $\frac{Q(s,a)-q_{\min}}{q_{\max}-q_{\min}}$.

For unvisited actions, we complete the missing Q-values using the mixed-value approximation of Danihelka et al. (2022):

$$Q_{\text{completed}}(s, a) = \begin{cases} Q(s, a), & N(s, a) > 0, \\ v_{\text{mix}}(s), & \text{otherwise.} \end{cases} \tag{5}$$

The mixed value combines the evaluator prediction $\hat{v}(s)$ with a policy-weighted average of visited action values:

$$v_{\text{mix}}(s) = \frac{1}{1 + N_s}\left[\hat{v}(s) + N_s \frac{\sum_{a:N(s,a)>0} \pi(a \mid s) Q(s, a)}{\sum_{a:N(s,a)>0} \pi(a \mid s)}\right], \tag{6}$$

where $N_s = \sum_b N(s, b)$. When no action has been visited, we set $v_{\text{mix}}(s) = \hat{v}(s)$.

Search follows the UCT and PUCT selection rules described in Section 3.2. The exploration coefficient is set to $c = 1.25$ for Sokoban and $c = 0.1$ for sliding puzzles, with the latter chosen by tuning standard MCTS on a small set of puzzles. For standard MCTS, the final action corresponds to the most-visited root child. This readout performed better on average across the evaluated benchmarks than selecting the action with the highest Q-value.

Algorithm 1 MCTS with a learned value function.

Require: root state s; simulation budget n_{sim}; value network v

initialize $N(\cdot,\cdot) \leftarrow 0$ and $Q(\cdot,\cdot) \leftarrow 0$ at the root s
for $i = 1 \dots n_{\text{sim}}$ **do**
  $n \leftarrow s, \quad \tau \leftarrow \langle\,\rangle$
  while not DONE(n) **do**
    $Q(n,\cdot) \leftarrow$ completed Q (Eq. 5), normalized with $q_{\min}$ and $q_{\max}$
    $a \leftarrow \arg\max_{a' \in \mathcal{A}(n)} \; Q(n,a') + u(n,a')$ ▷ u from UCT or PUCT
    $r \leftarrow R(n,a)$
    $\tau \leftarrow \tau \cup \langle (n,a,r) \rangle$
    if CHILD(n,a) exists **then**
      $n \leftarrow$ CHILD(n,a)
    else
      $n \leftarrow$ EXPAND(n,a)
      break
  if KEY(n) already in the table **then**
    bind n to the existing entry's statistics
    $V \leftarrow V_{\text{mix}}(n)$
  else
    register KEY(n) in the table
    $V \leftarrow 0$ if n terminated, else $v(n)$
  for $(n,a,r) \in$ REVERSE(τ) **do**
    $V \leftarrow r + \gamma V$
    $N(n,a) \leftarrow N(n,a) + 1$
    $Q(n,a) \leftarrow Q(n,a) + \dfrac{V - Q(n,a)}{N(n,a)}$
    update $q_{\min}, q_{\max}$ with $Q(n,a)$
return $\arg\max_{a \in A} N(s,a)$

Following AlphaGo Zero (Silver et al., 2017), exploration under PUCT is encouraged by mixing the policy prior with Dirichlet noise:

$$P(s,a) = (1-\varepsilon)\pi(a \mid s) + \varepsilon\eta_a, \qquad \boldsymbol{\eta} \sim \text{Dirichlet}(\alpha \mathbf{1}), \tag{7}$$

where the noise vector is defined over legal actions. We use $\alpha = 2.0$ to scale it inversely with the number of legal actions and $\varepsilon = 0.25$. Standard MCTS applies this noise at the root. In LASER, we apply it at the starting node of each local search to encourage exploration within the selected subtree.

C Selector training and architecture

C.1 Self-Imitation Learning

For Atari 100k, we train the selector through self-imitation using the agent's own best episodes, without external demonstrations. At each selector update round, we rank all episodes collected so far by return and select the top 5%, capped at 20 episodes. The collection is updated from the complete training history at every update round, so newly collected episodes with higher returns can replace earlier demonstrations.

Selector updates are interleaved with world-model training. We begin after 10,000 world-model gradient steps and perform one selector update round every 1,000 world-model gradient steps thereafter. This initial delay allows the agent to collect more informative demonstrations before selector training begins. At each round, we update the selector using a batch sampled uniformly from the demonstrations in the selected episodes.

C.2 Architecture

The frontier selector is a graph neural network that operates on the search tree. Each tree node is represented as a vertex, and each parent–child link is represented by two directed edges, one in

each direction. Node features are linearly projected into $d = 64$-dimensional hidden representations and updated through $T = 2$ rounds of message passing. Each round uses a GATv2 attention layer (Brody et al., 2021) with $n_{\text{heads}} = 4$ attention heads, each of dimension $d_{\text{head}} = 16$. Edge features are incorporated into the attention logits. The resulting message updates each node's hidden representation through a GRU (Cho et al., 2014), followed by layer normalization. After T rounds, a two-layer MLP maps each frontier node's representation to a scalar logit. The starting node for the next local search is sampled from the softmax distribution over these logits. The selector contains approximately 3.8×10^4 parameters.

The selector uses only information available in the search tree. For a node n corresponding to state s, the node features are

$$\left[\frac{d(n)}{d_{\max}},\ \log(1 + N(\text{par}(n))),\ \hat{v}(s),\ \mathbb{1}_{n\in\mathcal{F}},\ \mathbb{1}_{n=\text{root}}\right],$$

where $d(n)$ is the node's depth, $d_{\max}$ is the maximum depth of the current tree, and $\mathcal{F}$ is the set of frontier nodes. For a parent node with state s and a child reached by action a, both directed edges carry the features

$$[\text{onehot}(a),\ Q(s,a),\ \log(1 + N(s,a)),\ \text{direction}].$$

The direction indicator is 1 for the parent-to-child edge and 0 for the reverse edge; all other features are identical. These features give the selector access to the tree structure and accumulated search statistics without privileged information about the environment.

C.3 Training

For training the selector, we add an exploration bonus term to the objective, giving us

$$\mathcal{L}(s, a^*) = -\sum_{k=1}^{K}\left[\log \mu_\theta(n_k \mid \mathcal{T}_k)\, R_k^{\gamma_c}\right] - \beta H(\mu_\theta(\cdot \mid \mathcal{T}_k)),$$

where $H(.)$ is the Shannon entropy. The rest of hyperparameter used for training the selector is listed in Table 3.

Fig. 5 shows the selector's success rate against the number of training states. At zero states, the selector is untrained, so each curve starts from the random-selection baseline. The dashed lines mark the final performance when trained on the whole dataset, and the marker shows where the running mean of past few evaluations first reaches it. Most of the gain appears within the first ten percent of the data, after which the curves flatten: in seven of the eight settings the selector matches its full-data performance using between 20% and 57% of the training states. Curves end where training was stopped once the criterion was met, so their right-hand endpoints reflect the stopping rule rather than any limit on further improvement.

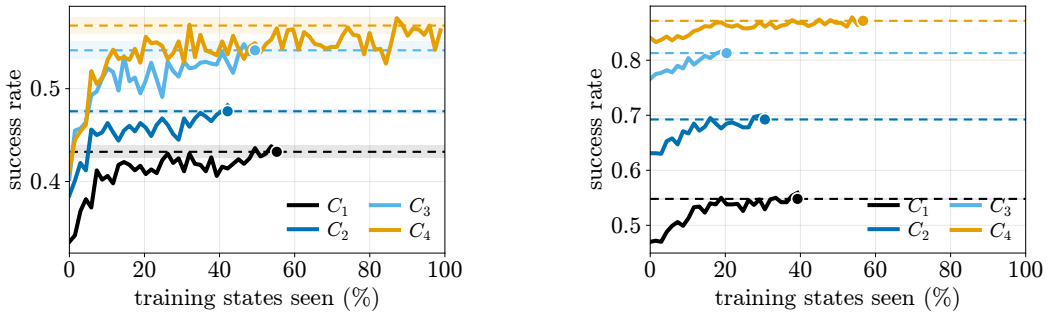
C.4 Selector Ablations

GNN versus MLP. We evaluate the contribution of the GNN by replacing it with an MLP while retaining the same node and edge features. The MLP scores each candidate independently using its

Table 3: Hyperparameters for selector training.

Hyperparameter	**Value**
Learning rate	1e-4†
Batch size	256
Number of epochs	1†
Credit discount	0.9
Entropy coefficient	0.1
gradient-norm clip	1.0
Root policy temperature τ	1.0‡

†Sliding puzzle: 5e-5, 3 epochs ‡Atari: 1e−4



(a) UCT (b) PUCT

Figure 5: Selector learning curves in Sokoban, plotted against the percentage of training states seen. Dashed lines indicate full-data performance, and markers show when a running mean of recent evaluations first reaches this level.

Table 4: Selector architecture ablation. Results are means with 95% confidence intervals.

	UCT			PUCT		
Environment	MLP	GNN	GNN + emb.	MLP	GNN	GNN + emb.
Sokoban, C_1	$39.0_{\pm 0.7}$	$43.2_{\pm 0.7}$	$42.2_{\pm 1.4}$	$54.6_{\pm 0.3}$	$54.8_{\pm 0.1}$	$53.9_{\pm 0.7}$
Sokoban, C_2	$45.3_{\pm 0.4}$	$47.6_{\pm 0.2}$	$47.7_{\pm 1.1}$	$68.7_{\pm 0.5}$	$69.2_{\pm 0.1}$	$68.5_{\pm 0.2}$
Sokoban, C_3	$49.8_{\pm 0.3}$	$54.1_{\pm 0.9}$	$52.8_{\pm 0.8}$	$81.3_{\pm 0.3}$	$81.3_{\pm 0.1}$	$81.1_{\pm 0.3}$
Sokoban, C_4	$52.6_{\pm 0.5}$	$56.8_{\pm 0.9}$	$55.6_{\pm 0.3}$	$86.5_{\pm 0.1}$	$87.2_{\pm 0.2}$	$86.9_{\pm 0.2}$
15-puzzle	$89.9_{\pm 0.2}$	$92.5_{\pm 0.5}$	$90.5_{\pm 0.4}$	$97.8_{\pm 0.1}$	$99.9_{\pm 0.1}$	$100.0_{\pm 0.0}$
24-puzzle	$88.9_{\pm 1.1}$	$89.4_{\pm 0.0}$	$90.7_{\pm 0.7}$	$98.5_{\pm 0.3}$	$100.0_{\pm 0.0}$	$100.0_{\pm 0.0}$
35-puzzle	$79.9_{\pm 0.6}$	$90.1_{\pm 2.0}$	$89.9_{\pm 1.8}$	$93.0_{\pm 0.8}$	$99.4_{\pm 0.1}$	$99.0_{\pm 0.5}$

node features and the features of its incoming parent edge. Table 4 shows that the GNN outperforms the MLP under UCT. Selection therefore benefits from information beyond the individual candidate node. Under PUCT, however, the MLP performs only slightly worse than the GNN. We hypothesize that strong policy guidance within local searches reduces the importance of choosing the best starting node, particularly when the search budget is limited. This could narrow the performance gap between selection rules that use different amounts of tree context.

State embeddings. We also augment the GNN inputs with state embeddings produced by a small CNN to assess whether access to state features improves selection. Adding these embeddings does not improve performance in our experiments. Although state features could provide useful information, the selector's training objective may not provide a strong enough signal to learn effective representations. Identifying the best starting node may itself require substantial understanding of how the task can be solved. Learning this information from the selector's indirect supervision is challenging, and the resulting embeddings may introduce noise rather than useful guidance. Meanwhile, the Q-values themselves can be viewed as embeddings that encode task-relevant information and may already be sufficient to guide starting node selection.

Informative state features could nevertheless improve selection. For example, a reliable deadlock indicator in Sokoban could help the selector identify states from which the puzzle cannot be solved and avoid spending further computation on them.

C.5 Which Nodes Get Selected

To investigate which features influence selection, we fit a linear model to reproduce the learned selector's choices under UCT for each task. Fig. 6 reports the mean relative coefficients and their uncertainty intervals. The fitted models favor shallower nodes in sliding puzzles and penalize depth less strongly in Sokoban. One possible explanation is that the value functions trained for sliding

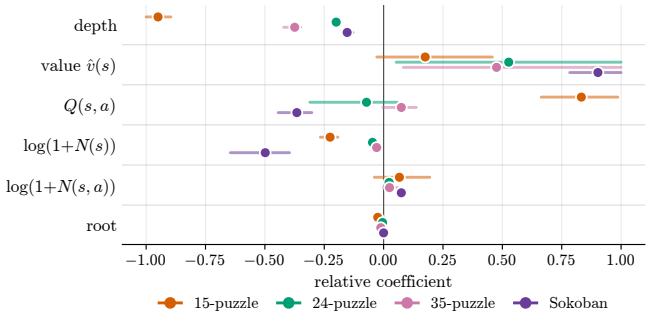


Figure 6: Relative feature coefficients of linear scoring models fitted to reproduce the learned selector's choices under UCT. Points show mean coefficients and horizontal bars indicate uncertainty intervals.

Table 5: Solve rates (%) under UCT using the learned GNN selector or a linear scoring model fitted to reproduce its choices. Sokoban results use checkpoint C_4.

	Success rate	
Environment	GNN	Linear
15-puzzle	92.5	93.5
24-puzzle	89.4	88.1
35-puzzle	90.1	85.3
Sokoban	56.8	39.8

puzzles provide more accurate guidance, so comparing shallow branches may be more useful than allocating additional search to deeper subtrees. This could also explain the weaker performance of random selection in sliding puzzles under UCT. When deeper levels contain more candidate nodes, uniform sampling allocates a larger fraction of selections to those levels, even when comparisons closer to the root would make better use of the value estimates.

The visit-count coefficients indicate how the selector balances exploration across branches with further search within them. The parent visit count receives a negative coefficient, whereas the candidate's own visit count receives a positive or near-zero coefficient. Holding other features fixed, the former favors candidates whose parents have received less search. Among candidates sharing a parent, a positive coefficient on the candidate's own visit count instead favors more frequently visited continuations. These preferences are consistent with exploring less-visited branches while reinforcing promising continuations within each branch.

Table 5 reports planning performance when the fitted linear models replace the GNN selector. The linear selector outperforms the GNN on the 15-puzzle, whereas the GNN achieves higher success rates on the larger puzzles and Sokoban. This progression suggests that simple scoring rules can suffice for easier tasks, but more complex tasks require more expressive selection mechanisms. Together with the MLP ablation, these results highlight the value of reasoning about relationships within the search tree beyond evaluating each candidate independently.

D Additional Experimental Results

D.1 Effect of Noise in PUCT

Fig. 7 shows the effect of the noise fraction ε in Eq. (7) on planning performance under PUCT. Adding noise yields larger gains for the learned selector than for standard MCTS. One possible explanation is that a strong policy prior dominates local search and limits the influence of frontier selection on the induced policy. Adding noise to the prior at the selected frontiers encourages local search to explore actions with low prior probability. The resulting search outcomes may then depend more strongly on which frontiers are selected, so the learned selector has greater influence over the induced policy.

D.2 Search Budget Allocation

Fig. 8 evaluates different allocations of K and H while keeping the total simulation budget KH fixed. Across all panels, performance improves substantially when K first increases beyond one, which supports the benefit of dividing the budget among selected subtree searches. Under UCT, performance peaks at $K = 25$, whereas PUCT achieves its highest success rates at $K = 5$ or $K = 10$.

Increasing K gives the selector more opportunities to direct computation toward different subtrees, but the corresponding decrease in H leaves fewer simulations for each local search. Without a policy prior, frequent allocation decisions may provide useful guidance that UCT alone lacks. Under PUCT,

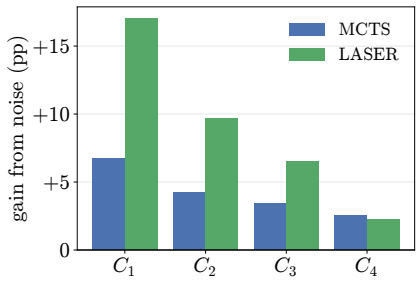
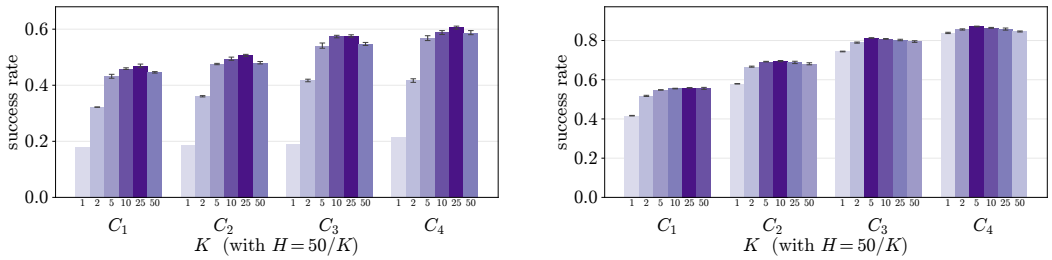


Figure 7: Effect of adding noise to the policy prior in Sokoban under PUCT. Bars show solve-rate gains in percentage points relative to search without noise.



(a) UCT (b) PUCT

Figure 8: Search budget allocation in Sokoban. We vary the number of local searches K with $H = 50/K$, keeping the total budget fixed at 50 simulations per decision. Darker bars indicate higher solve rates within each checkpoint, and error bars show 95% confidence intervals.

the prior already guides exploration within each subtree, so allocating more simulations to each local search may be more beneficial. This trade-off also affects selector training: larger K lengthens the decision-making horizon, while smaller H may make the reward associated with each selection noisier.

D.3 An Illustrative Example of Search Allocation

Fig. 9 shows an example of search trees constructed by standard MCTS and the learned selector using UCT from the same state early in a Sokoban episode . In this example, MCTS produces a more balanced tree, while the learned selector concentrates computation in selected subtrees. MCTS also spends part of its search budget expanding deadlocked branches, where further search cannot lead to reward.

Selecting a deadlocked starting node wastes the entire budget of the corresponding local search, so the selector should learn to avoid such candidates. To assess this behavior, we estimate the probability that the selector assigns a lower score to a deadlock candidate than to a non-deadlock candidate. For each tree, we compute the proportion of deadlock–non-deadlock candidate pairs in which the deadlock candidate receives a lower score. Averaging these proportions across trees yields an estimated probability of 0.85, indicating that the selector has learned to disfavor deadlocked candidates, although it can still select them.

This preference emerges without an explicit deadlock indicator or direct supervision for deadlock detection. The selector learns from expert demonstrations and relies on search statistics and their context in the tree to guide its decisions. Including informative task-specific features, such as a deadlock indicator, could help it avoid unproductive subtrees more reliably and further improve search efficiency.

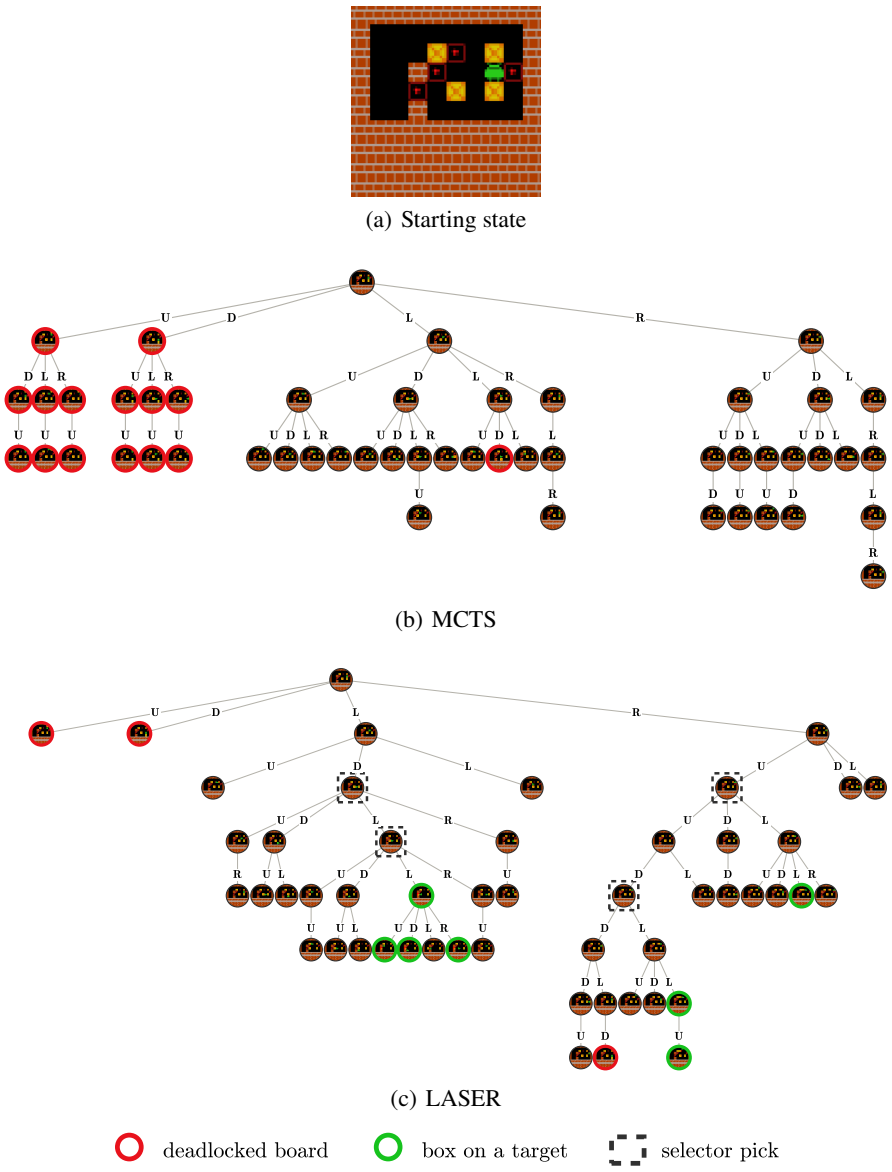


Figure 9: Search trees constructed by MCTS and LASER under UCT from the same Sokoban state. Edge labels denote actions.

D.4 Atari Details

Table 6 reports per-game scores for all 26 Atari 100k games using EfficientZero's open-source implementation. LASER replaces MCTS in all stages: during data collection, policy reanalysis, and evaluation. We use the same MCTS implementation in EfficientZero as the local search. Each search comprises $K = 5$ local searches of $H = 10$ simulations each, matching the baseline's budget of 50 latent-model expansions. We only apply Q-value completion (see Appendix B) to LASER's policy readout which does not affect the baseline. Selector training and self-imitation procedures are described in Section C.1.

For all methods, we disable `use_root_value` in EfficientZero implementation to reduce search computation during reanalysis by nearly half. With this setting, value targets bootstrap from the target network's value predictions rather than root values obtained through additional search. Policy reanalysis and the adaptive TD horizon remain enabled, so only the search-based component of off-

Table 6: Per-game scores on Atari 100k, reported as means over four seeds with 95% confidence intervals. The bottom rows summarize human-normalized scores (HNS) and z-scores computed using MCTS as the reference for each game.

Game	MCTS	Random	DeepSearch	LASER
Alien	$685.2_{\pm 46.6}$	$757.0_{\pm 45.9}$	$588.4_{\pm 50.8}$	$\mathbf{757.8}_{\pm 114.5}$
Amidar	$137.6_{\pm 31.8}$	$155.8_{\pm 16.8}$	$80.4_{\pm 10.7}$	$\mathbf{156.7}_{\pm 7.5}$
Assault	$1{,}403_{\pm 89}$	$1{,}106_{\pm 24}$	$919.9_{\pm 50.6}$	$\mathbf{1{,}426}_{\pm 15}$
Asterix	$18{,}237_{\pm 300}$	$17{,}476_{\pm 630}$	$\mathbf{18{,}295}_{\pm 794}$	$17{,}021_{\pm 1{,}023}$
BankHeist	$243.0_{\pm 22.1}$	$192.2_{\pm 49.3}$	$201.1_{\pm 64.8}$	$\mathbf{465.9}_{\pm 176.8}$
BattleZone	$\mathbf{7{,}844}_{\pm 1{,}091}$	$6{,}109_{\pm 872}$	$6{,}727_{\pm 650}$	$7{,}414_{\pm 272}$
Boxing	$40.5_{\pm 8.5}$	$45.3_{\pm 14.4}$	$44.6_{\pm 11.3}$	$\mathbf{49.0}_{\pm 2.7}$
Breakout	$383.5_{\pm 6.1}$	$437.7_{\pm 24.8}$	$\mathbf{485.5}_{\pm 14.3}$	$461.7_{\pm 32.1}$
ChopperCommand	$1{,}232_{\pm 141}$	$1{,}164_{\pm 39}$	$1{,}235_{\pm 65}$	$\mathbf{1{,}355}_{\pm 126}$
CrazyClimber	$\mathbf{85{,}273}_{\pm 7{,}740}$	$70{,}557_{\pm 6{,}772}$	$75{,}333_{\pm 3{,}335}$	$83{,}898_{\pm 4{,}561}$
DemonAttack	$7{,}991_{\pm 477}$	$8{,}347_{\pm 583}$	$\mathbf{9{,}467}_{\pm 1{,}172}$	$7{,}637_{\pm 492}$
Freeway	$\mathbf{16.5}_{\pm 5.5}$	$\mathbf{16.5}_{\pm 5.5}$	$5.5_{\pm 5.5}$	$\mathbf{16.5}_{\pm 5.5}$
Frostbite	$269.4_{\pm 5.8}$	$252.1_{\pm 2.2}$	$\mathbf{643.0}_{\pm 388.0}$	$298.4_{\pm 22.9}$
Gopher	$\mathbf{2{,}838}_{\pm 170}$	$1{,}902_{\pm 328}$	$1{,}965_{\pm 142}$	$2{,}416_{\pm 104}$
Hero	$\mathbf{10{,}354}_{\pm 1{,}172}$	$9{,}591_{\pm 968}$	$9{,}172_{\pm 1{,}021}$	$9{,}835_{\pm 692}$
Jamesbond	$\mathbf{453.1}_{\pm 66.7}$	$262.5_{\pm 37.5}$	$108.2_{\pm 45.8}$	$367.6_{\pm 80.5}$
Kangaroo	$\mathbf{1{,}717}_{\pm 292}$	$282.8_{\pm 111.0}$	$445.3_{\pm 140.5}$	$568.8_{\pm 153.8}$
Krull	$7{,}160_{\pm 511}$	$5{,}840_{\pm 586}$	$6{,}623_{\pm 822}$	$\mathbf{7{,}259}_{\pm 212}$
KungFuMaster	$20{,}125_{\pm 2{,}930}$	$20{,}751_{\pm 1{,}549}$	$20{,}920_{\pm 2{,}505}$	$\mathbf{24{,}872}_{\pm 2{,}239}$
MsPacman	$1{,}297_{\pm 42}$	$\mathbf{1{,}381}_{\pm 69}$	$1{,}023_{\pm 90}$	$1{,}368_{\pm 199}$
Pong	$14.5_{\pm 1.1}$	$20.0_{\pm 0.2}$	$14.2_{\pm 4.9}$	$\mathbf{20.5}_{\pm 0.2}$
PrivateEye	$54.7_{\pm 26.4}$	$37.0_{\pm 17.8}$	$-201.4_{\pm 235.9}$	$\mathbf{100.0}_{\pm 0.0}$
Qbert	$\mathbf{13{,}559}_{\pm 891}$	$11{,}830_{\pm 911}$	$9{,}812_{\pm 1{,}232}$	$13{,}030_{\pm 608}$
RoadRunner	$11{,}320_{\pm 1{,}237}$	$18{,}239_{\pm 1{,}310}$	$\mathbf{22{,}307}_{\pm 3{,}677}$	$20{,}533_{\pm 1{,}148}$
Seaquest	$985.6_{\pm 26.2}$	$673.4_{\pm 63.0}$	$801.6_{\pm 127.0}$	$\mathbf{1{,}244}_{\pm 105}$
UpNDown	$6{,}269_{\pm 784}$	$9{,}166_{\pm 2{,}504}$	$\mathbf{10{,}012}_{\pm 2{,}071}$	$6{,}452_{\pm 791}$
Mean HNS (%)	167.2	165.1	174.0	**183.1**
Median HNS (%)	72.1	76.8	50.8	**79.0**
Mean z	0.00	-0.33	0.75	**0.89**
Median z	0.00	-0.33	-0.58	**0.21**

policy value correction is removed. This reduced-compute configuration follows the implementation's defaults but differs from the setting used for EfficientZero's published results, which our experiments do not aim to reproduce.