LEARNING STRUCTURED METACONTROL

Anonymous authors
Paper under double-blind review

ABSTRACT

Recent results in large language model reasoning underscore a well-known truth in classical planning machine learning: the ability to intelligently control how compute is allocated is paramount. For tree search, the mechanisms for making these decisions often rely on handcrafted rules, such as running a fixed amount of iterations of search before deciding on a move. This contrasts heavily with the evaluation component of search, which has benefitted enormously over the years from the adoption of learned, rather than predefined, evaluators. Here, we present a method where a controller learns to decide whether to act or keep planning (i.e., metacontrol) based on a learned embedding of the current planning tree. We evaluate our model's performance on chess play, showing that it is a strong meta-controller and beats existing baselines on efficient allocation of tree search iterations. We examine different ways of parametrizing our idea and show that a key component is the proper integration and propagation of estimator uncertainty in the evaluator. Finally, we interpret the learned halting mechanism itself.

1 INTRODUCTION

Knowing how to efficiently allocate compute is central for optimal reasoning in complex settings. When proving a theorem, planning the implementation of a coding task, or figuring out what move to play in a game of chess, being able to optimize what to think about at any given moment is key to success. Seeking to maximize the eventual value of some future sequence of "real" actions via the optimization of immediate "thinking" actions — i.e., metareasoning, or "thinking about thinking" — requires an agent to interrogate the current state of its own reasoning in order to decide what directions are most fruitful to explore, and if further thinking is even warranted.

In this paper, we examine precisely the latter question: how do you know when to stop thinking? Historically, time allocation in search algorithms like Monte Carlo tree search (MCTS; Coulom, 2006; Kocsis & Szepesvári, 2006) has not been very sophisticated: typical approaches include running a fixed number of iterations or for a fixed wall-clock time. This stands in stark contrast to the evaluation component of search, which has benefitted enormously from the development of learned, rather than predefined, evaluators (Silver et al., 2018; Lightman et al., 2024; Mnih et al., 2013; Nasu, 2018; The LCZero Authors, 2018).

A principled system for intelligently allocating search iterations should 1) be able to interrogate the planning object directly, 2) integrate over epistemic uncertainty in the evaluator's estimate (for example, a shallow plan constructed by an expert is likely more reliable than a deep plan made by a novice), and 3) account for any costs associated with additional planning.

Here, we present a method that achieves all three of these desiderata. Using chess as a testbed, we operationalize a plan as a PUCT tree generated with the LeelaChessZero (lc0; The LCZero Authors, 2018) chess engine. Importantly, we use a Graph Neural Network (GNN; Scarselli et al., 2009; Li et al., 2017; Kokkinos & Potamianos, 2017; Veličković et al., 2018) to embed the search tree continuously throughout the planning process. The GNN is pretrained on a representation-learning objective that promotes the propagation of value distributions throughout the tree. The tree embedding is then passed to a separate controller network that predicts the relative value of halting versus continuing to plan, with successive planning actions incurring increasing costs. We show that our method, which we call GRAph MetaReasoning (GRAMR), strongly outperforms a litany of stopping rules from the literature. We inspect several ways of instantiating GRAMR and show that incorporating epistemic uncertainty into the representational objective, and prospectively estimating

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.

the outcomes of future search, each measurably increase performance when planning. Finally, we show that the controller network uses the value representations encoded in the tree embedding, as well as the budget, in order to make its decision.

2 Related work

We begin with a quick summary of related work in this area.

Adaptive computation time. Dynamically allocating thinking time has been considered in various forms throughout the field. One notable line considers the problem of letting RNNs allocate varying numbers of recurrent steps per input token (Graves, 2017; Banino et al., 2021), though this is somewhat different than controlling an external search process. More relevantly, Pascanu et al. (2017) encode a plan as a vector in a high-dimensional space, though in their hands this is done by passing real and imagined experiences through an LSTM that sequentially updates a latent representation; how to apply this approach to tree-structured data is unclear (Guez et al., 2018; Hamrick et al., 2017; Pascanu et al., 2017). Lan et al. (2020) tackles the problem of learning to stop for MCTS. Their model predicts the resolvability of given board states from either the state itself or from a collection of search-derived features, which is similar to our lookahead encoding, but they 1) do not process the entire tree, instead focusing only on the root-children actions, 2) do not account for any cost of planning, and 3) do not account for estimator noise. Dynamic Thinker (DT; Wang et al., 2025; Chung et al., 2023) notably altered the Thinker architecture to incorporate a halting action; DT thus, like us, has the capability to terminate search when advantageous, and is optimized to maximize expected real rewards minus planning costs. Unlike us, the internal agent in DT operates only on statistics drawn from two nodes: the root, and a currently focused node. Our GNN-based approach allows us to holistically consider the entire tree at once.

Metareasoning. The foundations of metareasoning can be found in Russell & Wefald (1991), which contends that agents should treat their own computations as a sort of "meta-action" that they can choose, and over which they ought to plan to maximize future reward. Callaway et al. (2018) derived an algorithm for solving these meta-level MDPs. The uncertainty-tracking component of our work was directly inspired by Bayesian Q-learning (Dearden et al., 1998), which tackles the problem of how to integrate parametrized uncertainty into an exploration policy for Q-learning. Sezener et al. (2019) explore the same idea in a simple search context under slightly different and stronger statistical assumptions.

Modern results. Recent impressive results in LLM reasoning have spurred new research into how to dynamically allocate inference-time compute. Policy-guided tree search (Li, 2025) explores the problem of learning how to expand and navigate a search tree. Their approach involves maintaining a cursor that can be moved around the tree, and that defines where search-related actions (expansion, termination, etc.) may take place. Like us, they use a graph neural network for embedding and optimize over predefined costs for certain computations. In contrast, their method must be fit online with the generator in the loop; we use an offline RL approach that can be trained much more efficiently. Others have investigated predicting the amount of benefit a prompt would gain from additional compute (Damani et al., 2024), the integration of process reward model uncertainty into search (Song et al., 2026), online detection of reasoning stabilization (Fu et al., 2025), and even fine-tuning LLMs on a metareasoning objective (Sabbata et al., 2025).

3 Model

3.1 Intuition

We begin with a high level description of our method (schematized at Figure 1). We operationalize a "plan" as a tree T, where the root node corresponds with the current state of the "real world" and further nodes represent imagined states. In the constructed tree, each edge corresponds to a valid action and each node corresponds to the successor state of that action; in our case each edge is a valid move and each node the resulting chess position. Each node contains a set of features that summarize that node's value. We use a Graph Neural Network (Scarselli et al., 2009; Li et al., 2017; Kokkinos & Potamianos, 2017; Veličković et al., 2018) to embed the tree T. This embedding compresses the information contained within the nodes of the tree into a single, high-dimensional vector h_{root} in a

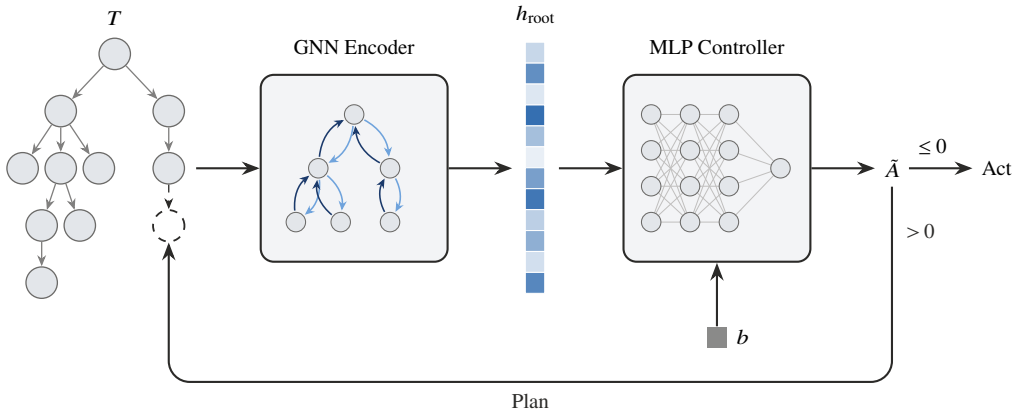


Figure 1: **Illustration of GRAMR architecture.** A planning tree is first embedded with a GNN into a high-dimensional space. The encoding of the root specifically is passed to the controller network, which estimates the expected advantage of planning over acting, given the current budget.

rich representational space. A separate controller network accepts h_{root}, as well as a budget variable b indicating how much time the agent has on the clock (running out of time is an automatic loss), and predicts the advantage of planning over halting $A(T, b) = Q(T, b, \text{plan}) - Q(T, b, \text{act})$, where $Q(T, b, a)$ denotes the expected value of performing a with current planning tree T and budget b. The encoder is initially pretrained with supervised learning (details below) to produce an embedding that we hypothesize will be particularly useful for planning; it is subsequently frozen and used to train the controller with reinforcement learning (details below) on actual episodes.

For training and evaluation, we sampled $100,000$ positions from human chess games via the Lichess open database. For each one, we constructed a deep planning tree (on the order of ten thousand nodes) with Monte-Carlo tree search using the value head from LeelaChessZero[1] (The LCZero Authors, 2018) as an evaluator. Both encoder pretraining and controller training leverage these pre-constructed trees.

3.2 Encoding

3.2.1 Graph neural networks

The current planning tree T is encoded using a GNN (Scarselli et al., 2009; Li et al., 2017; Kokkinos & Potamianos, 2017; Veličković et al., 2018). In brief, GNNs operate by projecting each node of a graph into a high-dimensional space and then performing k rounds of message-passing between the projected node representations. They are composed of four networks applied at each node n:

1. An embedding network that projects the features x_n of every node n to a high-dimensional space $h_0^n = f_{\text{embed}}(x_n)$. We took f_{embed} to be a small MLP.
2. A message-passing network that accepts the representation of every neighbour of n and aggregates them $M_i^n = f_{\text{msg}}\Big(\{h_i^{n'}; n' \in \mathcal{N}(n)\}\Big)$, where i is the current round index. We took f_{msg} to be multi-headed attention (Vaswani et al., 2017).
3. A state-update network that accepts the current representation of n, as well as the message from its neighbours M_n^i and produces an updated representation $h_{i+1}^n = f_{\text{state}}(h_i^n, M_i^n)$. We took f_{state} to be a small GRU (Cho et al., 2014).
4. An output network that operates only after the message-passing rounds are completed, $o = f_{\text{out}}(h_k^n)$. We took f_{out} to be a small MLP.

[1]Given a chess position, the lc0 value head returns an estimated outcome distribution $(p_{\text{win}}, p_{\text{draw}}, p_{\text{loss}})$.

Architectural hyperparameters for all these networks are given in Appendix A. Typically, GNNs are run "synchronously"; within a round, representations are treated as frozen. As a result, k sets the information radius of the model: nodes can integrate information only from nodes a maximum distance k within the graph. For tree search, where the graph grows with every iteration of search, this leads to undesirable consequences: the root of the tree would be blind to any nodes added below depth k. We instead structure our message passing rounds "asynchronously" from the leaves towards the root (Kokkinos & Potamianos, 2017). Thus every parent sees its children's post-update representations, rather than their pre-update representations. This allows information to flow from every node to the root in a single pass — and, correspondingly, we set $k = 1$.

While in principle the GNN could be trained end-to-end with the controller, we have found that in practice it is best to pretrain it on a representational objective that ensures it adequately aggregates information from the tree before it is introduced to the controller. The central question is thus: what is the right representation? We identify two principal axes along which this representation may vary: epistemic uncertainty encoding and prospective encoding.

Uncertainty encoding. The value estimates emitted by evaluators may be thought of as noisy approximations of a ground truth. In reality, a given chess position is either a win, a draw, or a loss with best play. The lc0 value head outputs a distribution over those outcomes, $(p_{\text{win}}, p_{\text{draw}}, p_{\text{loss}})$. In turn, that estimate itself may have an associated uncertainty arising from how well calibrated the value head is. Most state-evaluators do not quantify this uncertainty; they just output the point estimate. However, as we motivate in the next section, an optimal reasoner ought to integrate this uncertainty into its computations. We will refer to encoders that account for this uncertainty as "distributional" encoders, whereas the more traditional point-estimate version will be "point" encoders.

Prospective encoding. A natural approach is that each node in the tree should carry its own raw evaluation (i.e., the immediate output of the value head on the corresponding position); we would then require the encoder to learn to propagate those values throughout the tree, consolidating them to create an accurate summary. In essence, it would learn learn a backup operator: given the set of values for a node's children, compute the value of its parent, recursively towards the root. (A grid world analogy might be that it sees the vector of rewards associated with each state and from that predicts the value function induced by those rewards.) In this setting, a node's features are static in time: it maintains only its initial evaluation and does not get updated as search proceeds. An alternative approach would be to have each node store its *search-based* features (e.g., visit count, estimated value) explicitly, and for the GNN to instead predict how they will evolve with more iterations. That is, given the search features at iteration t, predict what those features will be at a time $t+\tau$. (The analogy here would be mapping from the *current* value function after t trajectories on the grid world to the value function τ iterations later.) In contrast to the first approach, which endeavours only to summarize the tree at the current moment, this approach seeks to prospectively evaluate where future search could lead. We will refer to the predictive kind of encoder as "lookahead" and the static backup encoders as "backup".

In essence, we have two axes along which we can instantiate our encoder: distributional vs. point, and lookahead vs. backup. We pretrained separate encoders on all four combinations of these axes. In each case, the objective was child-value decoding. In particular, we required:

$$f_{\text{out}}(h_n, c) \approx \tilde{v}_c \tag{1}$$

where $\tilde{v}_c$ denotes a representation of the value of the cth child of node n. In the point-estimate case, $\tilde{v}_c$ is simply an outcome triplet $(p^c_{\text{win}}, p^c_{\text{draw}}, p^c_{\text{loss}})$, whereas in the distributional case, $\tilde{v}_c$ is a CDF over $p^c_{\text{win}} - p^c_{\text{loss}}$ accompanied by a "mass" channel estimating the number of leaves supporting the estimate. In the backup case, $\tilde{v}_c$ is the relevant value quantity of c when consolidated by search (e.g., the visit-averaged outcome distribution in the point-estimate case). In the lookahead case, the search-consolidated value quantity is given as a feature and instead $\tilde{v}_c$ is taken as the same quantity but after τ additional iterations of search. In all cases, the GNN may be understood as being trained to represent node n in a manner that facilitates decoding of its *children's* value targets. This ensures that h_{root} carries information about the actions the agent may take.

3.2.2 Uncertainty propagation

As a motivating example, consider repeatedly choosing between two bandits A and B. If one's confidence in their value estimates is low (Figure 2, left), the choice may be hard; one wants to balance

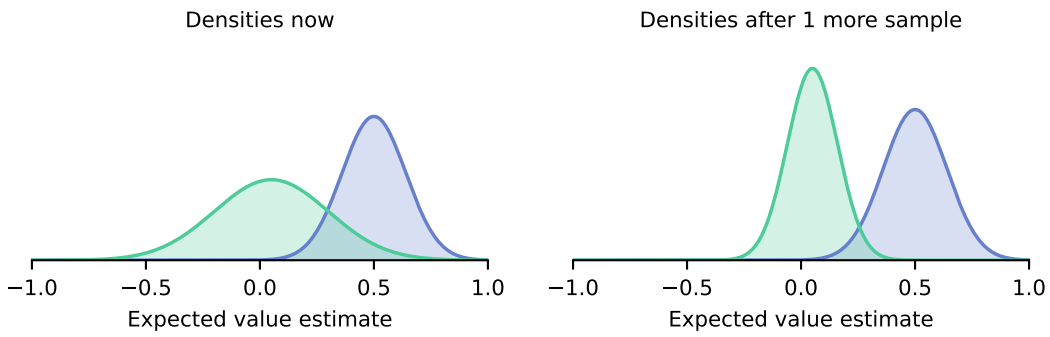


Figure 2: **Hypothetical inferred distributions over the expected return of two bandits.** Left: the current estimates. Right: the estimates after one additional sample of the non-preferred bandit.

information gathering (i.e., exploration) with reward accrual (i.e., exploitation). The appropriate exploratory choice would help reduce these uncertainties, making the choice potentially much easier (see Figure 2, right), at the cost of some reward. Bayesian Q-learning (Dearden et al., 1998) solves this problem by treating Q-values as normally distributed with latent mean and variance, updating a normal-gamma prior over them with data for uncertainty integration. Here, we apply the same principle to tree search.

Consider a chess position with two legal moves, A and B. Say, hypothetically, that A is estimated with high certainty to most likely be a draw but certainly not a loss, e.g. $(0.1, 0.9, 0)$. In contrast, B is believed to be very sharp so that either a loss or a win is possible, e.g. $(0.5, 0, 0.5)$. We ought to, in some sense, be able to say that the outcome distribution of the *parent* be something like "at least a draw, maybe a win". We now outline how we perform this computation.

Recall that the lc0 value head outputs its estimates in the form of a categorical distribution $(p_{\text{win}}, p_{\text{draw}}, p_{\text{loss}})$. In principle, we could model the uncertainty *around* this estimate using its conjugate prior, the Dirichlet distribution. However, in tree search, we generally want the value of a node to be the max over its children, and a max of Dirichlets is not itself Dirichlet, so the computational benefit of conjugacy would be lost after one layer of updates.

Instead, we represent each node as a discretized CDF over $p_{\text{win}} - p_{\text{loss}}$, evaluated at $\{-1, -0.8, \ldots, 0.8, 1\}$. A parent's induced CDF can be computed efficiently as simply the pointwise product of its children's CDFs (Baum & Smith, 1996). This process is seeded at the leaves by the lc0 value estimate, which we treat as the mean of a Dirichlet distribution and then approximate the CDF of the induced $p_{\text{win}} - p_{\text{loss}}$ distribution numerically. We separately also propagate the size of the subtree supporting a node's estimate, which is also a proxy for epistemic uncertainty. The in-depth training procedure for the encoders can be found in Appendix A.

3.3 Control

At a high level, the decision to act or keep planning is fundamentally about reward: additional planning (in expectation) increases the reward the agent hopes to get but incurs an associated cost (implicitly or explicitly). Thus, a sophisticated controller ought to estimate the expected value of planning relative to immediately acting (i.e., the "advantage") $A(T, b) = \mathbb{E}\Big[Q(T, b, \text{plan}) - Q(T, b, \text{act})\Big]$ and only keep planning if this difference is positive.

We estimate this quantity using a separate neural network controller, a small MLP, that accepts the embedded representation of the tree root generated by the encoder, h_{root}, as well as the current budget, b. We train the controller to approximate the advantage

$$f_{\text{control}}(h_{\text{root}}, b) \approx A(T, b) \tag{2}$$

by training it on episodes in which the agent earns reward by choosing to either plan or halt given a tree-budget pair.

Recall that our dataset consists of chess positions sampled from the Lichess open database, and then expanded into deep trees via MCTS. The controller sees (embeddings of) successive prefixes of this full tree; starting with the root, every time it selects the plan action, another MCTS expansion is played out, unveiling a new set of nodes and growing the subtree the controller sees. In addition, the controller is given a budget b; choosing to plan subtracts 1 from the budget and incurs a cost that increases as b approaches 0 (details in Appendix B). Following the convention of actual chess games, running out of budget results in a loss. Thus when choosing to plan, the controller must balance the expected gain versus the cost.

On the other hand, when the agent decides to act rather than plan, it earns a reward equal to the value of the move it plays, as scored by the full tree. Formally, let $V^T(m) = p^T_{\text{win}} - p^T_{\text{loss}}$ be the value of a given move m, as estimated using search tree T. Then, when the agent halts with subtree T_a and makes the corresponding move, it earns reward as scored by the expert tree T^*:

$$R = V^{T^*}\left(\arg\max_m V^{T_a}(m)\right) \tag{3}$$

To ensure that the deep tree indeed remains an "expert" scorer of the agent's moves, we cap the number of successive plan actions the agent can take at $L = 48$. As a (notable) byproduct, this allows to compute, via backward induction, the optimal value function at every depth up to L. If halting is forced at $L = 48$ then the value of halting at L is known: it is simply the expert evaluation of whatever move tree search picks after 48 planning steps. At move 47, then, the optimal value is:

$$\begin{aligned} V(T_{:47}, b) &= \max\{Q(T_{:47}, \text{act}, b), Q(T_{:47}, \text{plan}, b)\} \\ &= \max\{Q(T_{:47}, \text{act}, b), Q(T_{:48}, \text{act}, b-1) - c(b)\} \end{aligned}$$

where $c(b)$ is the planning cost and $T_{:x}$ denotes the x-expansion subtree the controller sees after x successive planning actions. This backward-induction logic may be repeated to compute the optimal value, and therefore the optimal action, at every available prefix of the tree.

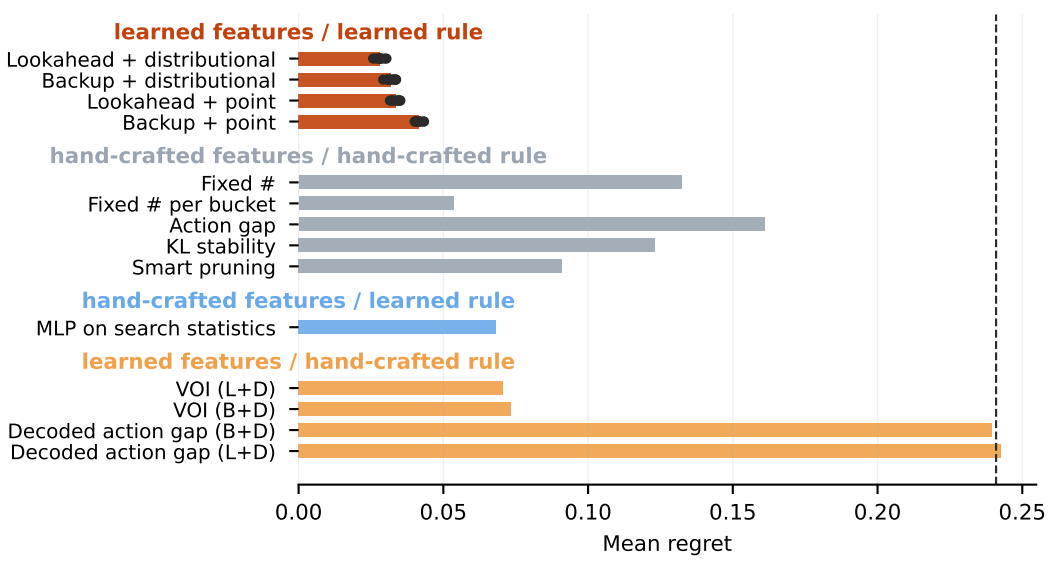


Figure 3: **GRAMR achieves substantially lower regret on real episodes than all baselines.** Mean regret on plan-vs-act decisions compared to choices made by an oracle. The learned features / learned rule group is exclusively various instantiations of GRAMR. All other categories are baselines from the literature. The dashed, vertical line indicates the regret achieved by always halting immediately. Full descriptions of each baseline can be found in Appendix C. L/B correspond to "Lookahead" or "Backup" encoders, P/D correspond to "Point" or "Distributional" encoders. Black dots in the GRAMR group are the results across 10 different seeds.

4 RESULTS

We trained four controllers (one for each encoder regime) and measured their regret compared to optimal behaviour on a heldout subset of trees. As baseline comparisons, we also measured the regret of several handcrafted rules (e.g., smart pruning, KL-divergence stabilization, fitting a fixed expansion count per budget bucket, among others) as well as some learned ones (e.g., optimizing an MLP on the root children statistics). A full description of each is available at Appendix C. All four of our controllers achieve significantly lower regret compared to all baselines (Figure 3).

Within the four encoder regimes, we find that accounting for uncertainty and prospective encoding each significantly increases the controller's performance (Figure 4). That is, distributional controllers are stronger than point controllers, and lookahead controllers are stronger than backup controllers. Our best controller, therefore, was the lookahead, distributional controller.

In principle, these results are theoretically grounded. On the distributional front, tracking the uncertainty in the value estimate for a given root child would hypothetically enable a sophisticated planner to compute the expected benefit of further planning, which is the basis of metareasoning. Similarly, the prospective encoding regime transforms the encoder from a backup approximator to an actual search-efficacy predictor — yet another ingredient in rational metareasoning.

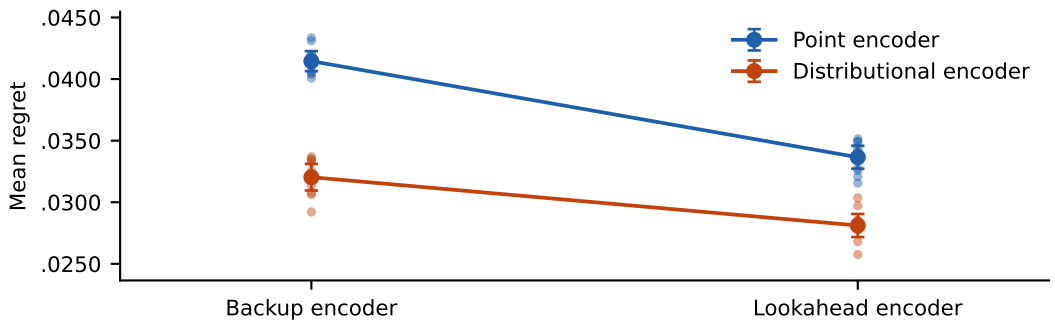


Figure 4: **Predictivity encoding and a distributional readout in the encoder reduce regret.** Regret versus optimal policy for controllers trained with each encoder representation regime. All four combinations were trained with ten different seeds. Error bars indicate 95% confidence intervals over the mean.

Next, we interrogated the mechanism that the controllers learned. First, we wanted to verify that the controller was actually reading the value representations associated with the root-children. To test this, we took a decoding approach: we trained linear probes to predict the value representations (histograms in the distributional case, WDL triplets in the point case) associated with the top 4 children in the tree from the controller activations at each layer (Figure 5, top). We found that these quantities were indeed decodable throughout the network, in a manner that decayed with depth. We also found that budget was almost perfectly decodable, and that the action gap (i.e., the difference in value between the top two children), despite not appearing in our representation learning scheme, was also significantly decodable at most layers (Figure 5, bottom).

Second, we validated that the best controller indeed uses the children's representations via an ablation approach. In particular, we took the tree encoding and linearly projected out directions associated with various quantities: (i) the value representations of the top three children separately, (ii) the top two jointly, (iii) the worst child, and (iv) the leaf count channel for the distributional encoders (Figure 6). We measured the induced gain in regret relative to the controller's baseline. We found that the lookahead, distributional encoder significantly suffered when the count channel is ablated, more than doubling its regret. Its response profile was broadly well-aligned with predictions from classical metareasoning: ablating the worst child mostly doesn't matter, and ablating high-rank children does quite matter. Notably, ablating rank 1 is less important than ablating rank 2. This latter prediction arises from the fact that, generally, thinking about the current favorite is (in expectation) unlikely to change future behavior, and is therefore less valuable than thinking about its immediate alternatives (Dearden et al., 1998; Russell & Wefald, 1991; Sagiv et al., 2025).

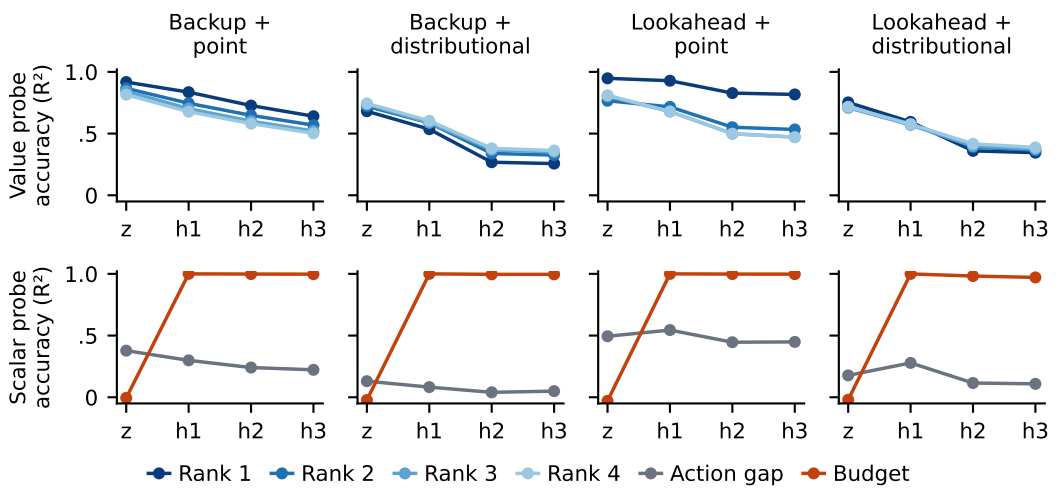


Figure 5: **Children values, action gap, and current budget decode linearly from the controller activations.** Decoding accuracy, measured as R^2, for linear probes trained to estimate various target variables from controller layer activations, as well as the root embedding. Top: the top 4 root children based on decoded value. Bottom: the action gap (i.e., the difference in value between the best and second-best option) and the budget.

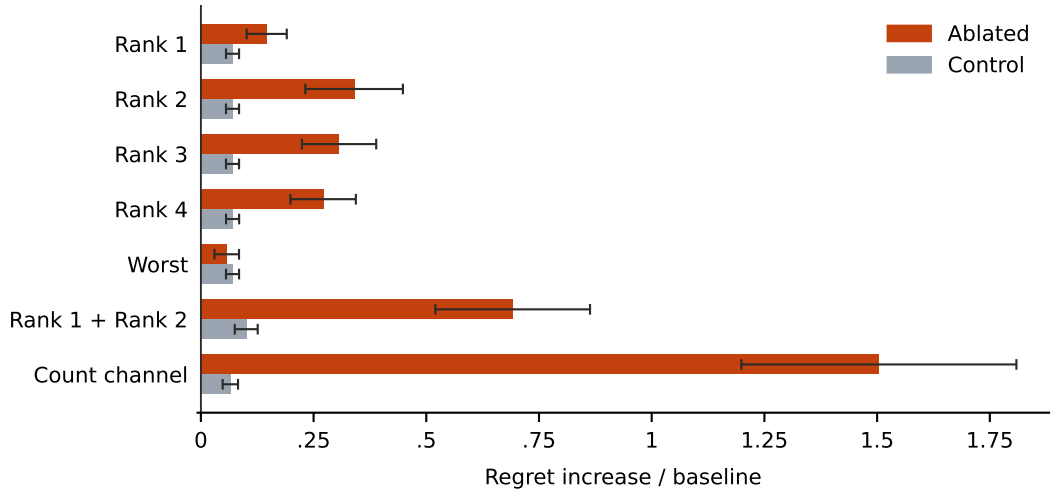


Figure 6: **Linearly projecting out decision variables from the tree encoding increases regret in the lookahead, distributional controller.** Increased regret relative to baseline when subspaces associated with various variables are linearly projected out of the root encoding. "Rank x" denotes the rank x child by decoded value. "Worst" conversely is the worst child. "Count channel" refers to the channel that estimates the number of leaves in a subtree beneath the given node. Controls are ablations on randomly-selected subspaces of equal dimension.

Finally, we examined how the network encoded the budget variable. In particular, we ablated the budget in two different ways: a mean ablation, always setting it to the mean budget allocation in the dataset, and a shuffling ablation that permuted budget labels. Both manipulations significantly degrade performance (Figure 7, left). Unsurprisingly, if the budget variable is misspecified, the model cannot correctly account for the upcoming cost of its actions and it massively underperforms. We then wanted to test the sensitivity of our controller to the budget. By construction, the planning cost function is very steep at low budget and quite permissive at high budget. So, we computed the change in advantage when a given tree-budget pair has its budget adjusted either up or down

(Figure 7, right). The controller was well-calibrated: reductions in budget reduce the advantage of planning, gains increase it, and the effect is significantly stronger when the initial budget is low.

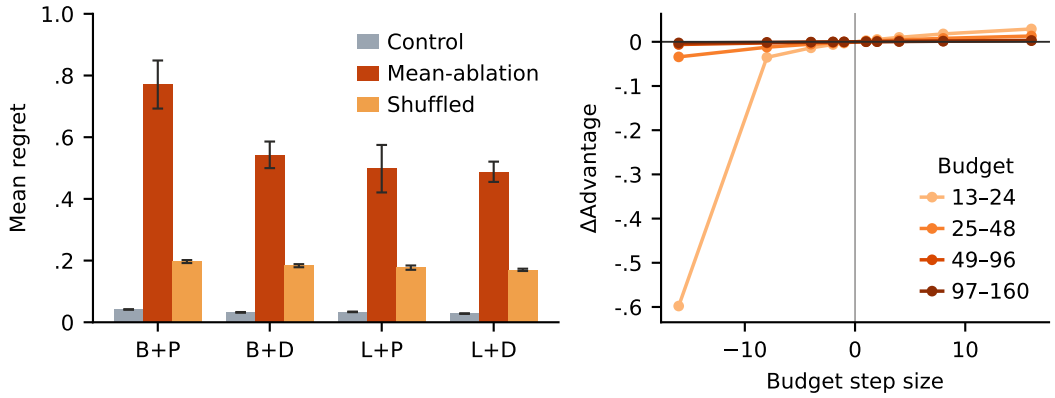


Figure 7: **The controller is well-calibrated on the latent planning cost function.** Left: mean and shuffle ablations of the budget input strongly impair control capabilities. Right: given a tree-budget pair, the change in advantage when that budget is adjusted up or down, across budget buckets.

5 Discussion

The real world is rife with domains where planning exhaustively is intractable and where it is not always possible to plan all the way to a well-defined goal state. In these cases, knowing when to stop planning is a critical question, as planning itself comes at a cost. Historically, in machine learning, this decision has been made with the use of handcrafted rules over predefined metrics. Here, we presented GRAMR, a method for learning how to estimate the advantage of planning over acting from data. GRAMR embeds the current planning tree into a bespoke representational space, and then processes that embedding through a learned controller to estimate advantage. Our method strongly outperforms a large set of baselines drawn from models in public use for chess play (e.g., the lc0 chess engine) as well as other metrics from the cognitive science and machine learning literatures.

We showed that the control allocation problem is particularly aided by a specific set of representational primitives: encoding the epistemic uncertainty around a value estimate, and estimating the outcomes of future search. These jointly ensure that the controller network has access to the basic computational ingredients for metareasoning; namely, knowing which of its value estimates are most uncertain, and having a model for how that uncertainty might get reduced with further search.

Our approach leaves a number of open questions. Most obviously, our encoders and controllers do not see the board itself; they see only value estimates computed from board states. This suggests an interesting possibility that the learned model is applicable outside the domain of chess and may generalize directly to other decision-making regimes (e.g., playing Go or proving theorems). Understanding the limits of this generalization and the extent to which this method can be used as a plug-and-play module for any value-guided tree search is an open question. Conversely, it is possible that our method would be greatly aided by having access to the board states, as right now it can only make decisions on the basis of value estimates and their uncertainties in a generic sense. Furthermore, although the current work is restricted to the question of halting, a natural question is whether it can be extended to guide the search policy itself.

6 FINAL

AI USE STATEMENT

In this work, we used generative AI tools primarily for method implementation – that is, transforming the algorithm design into a concrete implementation, suggesting and implementing performance improvements to increase throughput during training, codebase maintenance over the course of the project (e.g., refactoring for simplicity or to account for new functionality), and other tasks of that nature. In one specific case, the AI contributed to the design of the method; namely, numerical approximation of the marginal $p_{\text{win}} - p_{\text{loss}}$ from the leaf Dirichlets in the distributional encoders is accomplished via an identity supplied by the AI (and verified by the authors). We have not used generative AI tools for dataset generation, formulating or proving mathematical claims, proposal/refinement of hypotheses or experiments, translation, data cleaning, analysis support, or interpretation. We additionally used generative AI tools for refining created figures based on supplied prototypes, and for identification of relevant literature.

REPRODUCIBILITY STATEMENT

The authors are committed to ICLR's reproducibility mission. To support this, source code supporting the entire analytical pipeline – dataset generation, model implementation and training pipelines, analysis – will be uploaded to GitHub pending acceptance of the submission. In lieu of this, in-depth descriptions of e.g., architectural hyperparameters, training procedure, and so on are provided in the Appendix.

ACKNOWLEDGMENTS

This work was supported by NSF award number 2312374.

REFERENCES

Andrea Banino, Jan Balaguer, and Charles Blundell. PonderNet: Learning to Ponder, September 2021. URL http://arxiv.org/abs/2107.05407. arXiv:2107.05407 [cs.LG].

Eric B Baum and Warren D Smith. A Bayesian approach to relevance in game playing I. 1996.

Frederick Callaway, Sayan Gul, Paul M. Krueger, Thomas L. Griffiths, and Falk Lieder. Learning to select computations, August 2018. URL http://arxiv.org/abs/1711.06892. arXiv:1711.06892 [cs.AI].

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.

Rémi Coulom. Efficient selectivity and backup operators in monte-carlo tree search. In *International conference on computers and games*, pp. 72–83. Springer, 2006.

Mehul Damani, Idan Shenfeld, Andi Peng, Andreea Bobu, and Jacob Andreas. Learning How Hard to Think: Input-Adaptive Allocation of LM Computation, October 2024. URL http://arxiv.org/abs/2410.04707. arXiv:2410.04707 [cs.LG].

Richard Dearden, Nir Friedman, and Stuart Russell. Bayesian Q-Learning. 1998.

Yichao Fu, Junda Chen, Siqi Zhu, Zheyu Fu, Zhongdongming Dai, Yonghao Zhuang, Yian Ma, Aurick Qiao, Tajana Rosing, Ion Stoica, and Hao Zhang. Efficiently Scaling LLM Reasoning with Certaindex, May 2025. URL http://arxiv.org/abs/2412.20993. arXiv:2412.20993 [cs.LG].

Alex Graves. Adaptive Computation Time for Recurrent Neural Networks, February 2017. URL http://arxiv.org/abs/1603.08983. arXiv:1603.08983 [cs.NE].

Arthur Guez, Théophane Weber, Ioannis Antonoglou, Karen Simonyan, Oriol Vinyals, Daan Wierstra, Rémi Munos, and David Silver. Learning to Search with MCTSnets, July 2018. URL http://arxiv.org/abs/1802.04697. arXiv:1802.04697 [cs.AI].

Jessica B. Hamrick, Andrew J. Ballard, Razvan Pascanu, Oriol Vinyals, Nicolas Heess, and Peter W. Battaglia. Metacontrol for Adaptive Imagination-Based Optimization, May 2017. URL http://arxiv.org/abs/1705.02670. arXiv:1705.02670 [cs.LG].

Nicholas Hay, Stuart Russell, David Tolpin, and Solomon Eyal Shimony. Selecting Computations: Theory and Applications, July 2012. URL http://arxiv.org/abs/1207.5879. arXiv:1207.5879 [cs.AI].

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

Filippos Kokkinos and Alexandros Potamianos. Structural Attention Neural Networks for improved sentiment analysis, January 2017. URL http://arxiv.org/abs/1701.01811. arXiv:1701.01811 [cs.CL].

Li-Cheng Lan, Meng-Yu Tsai, Ti-Rong Wu, I.-Chen Wu, and Cho-Jui Hsieh. Learning to Stop: Dynamic Simulation Monte-Carlo Tree Search, December 2020. URL http://arxiv.org/abs/2012.07910. arXiv:2012.07910 [cs.AI].

Yang Li. Policy Guided Tree Search for Enhanced LLM Reasoning, February 2025. URL http://arxiv.org/abs/2502.06813. arXiv:2502.06813 [cs.LG].

Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. Gated Graph Sequence Neural Networks, September 2017. URL http://arxiv.org/abs/1511.05493. arXiv:1511.05493 [cs.LG].

Hunter Lightman, Vineet Kosaraju, Yura Burda, Harri Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. LET'S VERIFY STEP BY STEP. 2024.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing Atari with Deep Reinforcement Learning, December 2013. URL http://arxiv.org/abs/1312.5602. arXiv:1312.5602 [cs.LG].

Yu Nasu. Efficiently updatable neural-network based evaluation functions for computer shogi. Technical report, Ziosoft Computer Shogi Club, 2018. URL https://github.com/ynasu87/nnue/blob/master/docs/nnue.pdf. In Japanese with English abstract.

Razvan Pascanu, Yujia Li, Oriol Vinyals, Nicolas Heess, Lars Buesing, Sebastien Racanière, David Reichert, Théophane Weber, Daan Wierstra, and Peter Battaglia. Learning model-based planning from scratch, July 2017. URL http://arxiv.org/abs/1707.06170. arXiv:1707.06170 [cs.AI].

Stuart Russell and Eric Wefald. Principles of metareasoning. *Artificial Intelligence*, 49(1-3):361–395, May 1991. ISSN 00043702. doi: 10.1016/0004-3702(91)90015-C. URL https://linkinghub.elsevier.com/retrieve/pii/000437029190015C.

C. Nicolò De Sabbata, Theodore R. Sumers, Badr AlKhamissi, Antoine Bosselut, and Thomas L. Griffiths. Rational Metareasoning for Large Language Models, June 2025. URL http://arxiv.org/abs/2410.05563. arXiv:2410.05563 [cs.CL].

Yotam Sagiv, Thomas Akam, Ilana B Witten, and Nathaniel D Daw. Between planning and map building: Prioritizing replay when future goals are uncertain. *Neuron*, 113(24):4278–4292, 2025.

F. Scarselli, M. Gori, Ah Chung Tsoi, M. Hagenbuchner, and G. Monfardini. The Graph Neural Network Model. *IEEE Transactions on Neural Networks*, 20(1):61–80, January 2009. ISSN 1045-9227, 1941-0093. doi: 10.1109/TNN.2008.2005605. URL http://ieeexplore.ieee.org/document/4700287/.

Can Eren Sezener, Amir Dezfouli, and Mehdi Keramati. Optimizing the depth and the direction of prospective planning using information values. *PLOS Computational Biology*, 15(3):e1006827, March 2019. ISSN 1553-7358. doi: 10.1371/journal.pcbi.1006827. URL https://dx.plos.org/10.1371/journal.pcbi.1006827.

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. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science*, 362(6419):1140–1144, December 2018. ISSN 0036-8075, 1095-9203. doi: 10.1126/science.aar6404. URL https://www.science.org/doi/10.1126/science.aar6404.

Zeen Song, Zihao Ma, Wenwen Qiang, Changwen Zheng, and Gang Hua. Adaptive Uncertainty-Aware Tree Search for Robust Reasoning, February 2026. URL http://arxiv.org/abs/2602.06493. arXiv:2602.06493 [cs.LG].

The LCZero Authors. Leelachesszero, 2018. URL https://lczero.org/. Software, https://github.com/LeelaChessZero/lc0.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (eds.), *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017. URL https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf.

Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph Attention Networks, February 2018. URL http://arxiv.org/abs/1710.10903. arXiv:1710.10903 [stat.ML].

Kevin A Wang, Jerry Xia, Stephen Chung, and Amy Greenwald. Dynamic thinker: Optimizing decision-time planning with costly compute. In *The Seventeenth Workshop on Adaptive and Learning Agents*, 2025.

A Appendix: Encoder training procedure

The training procedures for all four GNN regimes are largely identical up to differences in loss function associated with the different targets. In all cases, we take a planning tree T from our MCTS corpus and, for each parent-child pair (p, c) compute the corresponding value target $\tilde{v}_c$. We apply the output head to the embedding of every parent and require it to output $f_{\text{out}}(h_p, c) = \tilde{v}_c$ with losses that vary according to the uncertainty regime.

The distributional encoders optimize a weighted combination of three primary losses and an auxiliary loss:

$$\mathcal{L}_{\text{dist}} = \mathcal{L}_{\text{Cramér}} + 0.25\mathcal{L}_{\text{CE}} + \mathcal{L}_{\text{mean}} + \mathcal{L}_{\text{mass}} \tag{4}$$

$\mathcal{L}_{\text{Cramér}}$ is a CDF-matching loss:

$$\mathcal{L}_{\text{Cramér}} = \Delta \sum_{i=1}^{w} (P_i - Q_i)^2 \tag{5}$$

where Δ is the bin-width (0.2 in our case), w is the number of bins, and P_i, Q_i are the discretized CDFs evaluated at bin i. $\mathcal{L}_{\text{CE}}$ is soft-target cross-entropy, used as an auxiliary to ensure that mass ends up at precisely the right bin in additional to being roughly distributionally correct. The mean loss $\mathcal{L}_{\text{mean}}$ shapes the representation to be particularly accurate about the histogram mean:

$$\mathcal{L}_{\text{mean}} = (\hat{v} - \bar{q})^2 \tag{6}$$

where $\hat{v}$, the estimated mean value of $p_{\text{win}} - p_{\text{loss}}$, is output by a separate mean-value head and $\bar{q}$ is the mean of the target value distribution. Finally, the distributional encoders, in addition to the value histogram, separately support a "count channel" that propagates the size of the subtree underneath

a given node. In particular, each node maintains a small accumulator state a_n that it updates as a function of the accumulator states and the graph embeddings of its children:

$$a_0^p = \text{softplus}(W_0 x_p + b_0) \tag{7}$$

$$a^p = a_0^p + \sum_c \sigma(W_g h^c + b_g) \odot a^c \tag{8}$$

where a_0^p is an initial count embedding based only on the features of p and a^p is the count updated from the children of p. This accumulator state is then injected into the graph embedding of p:

$$h^p \leftarrow h^p + W_I \log(1 + a^p) \tag{9}$$

Jointly, this ensures a parallel path through which subtree size can accumulate in the encoder without struggling against the bounded operations in the state GRU, and then be injected directly into the graph representation. This count channel is targetted by the final loss $\mathcal{L}_{\text{mass}}$.

$$\mathcal{L}_{\text{mass}} = (\log \hat{N} - \log N)^2 \tag{10}$$

where $\log \hat{N}$ is decoded by a specific count head from the embedding of h_{parent}.

In contrast to the distributional regimes, the point regimes are fairly straightforward and just use the soft-target cross-entropy loss on the output outcome distributions.

$$L_{\text{point}} = \mathcal{L}_{\text{CE}} \tag{11}$$

All four encoders were trained to convergence, at roughly 1200 epochs each. Below is a table of other relevant hyperparameters.

Table 1: Encoder training hyperparameters.

Hyperparameter	Value
Embedding dimension	128 (point), 512 (distributional)
Message dimension	128 (point), 512 (distributional)
Attention heads $\times$ head dimension	4×32
Count-channel dimension	8 (distributional), none (point)
Value-head hidden width	128 (point), 512 (distributional)
f_{embed}	2-layer MLP, hidden width 128, ReLU
f_{state}	GRU cell, input size is message dimension, hidden size is embedding dimension
f_{msg}	Attention over children, 4×32
f_{out}	2-layer MLP, hidden width 128 (point) or 512 (distributional), ReLU
Count and value auxiliary heads	2-layer MLPs, same width as the child-value head
Optimizer	Adam
Peak learning rate	10^{-3} (point), 5×10^{-4} (distributional)
Batch size	32
Epochs trained	1500

B Appendix B: Controller training procedure

We start again with our 100k tree corpus. We pair each tree with a budget for decision-making by creating five buckets: scramble (6-12), small (13-24), medium (25-48), large (49-96), and very large (97-160). We convert each tree into ten episodes by sampling two budget counts uniformly from each bucket, for a starting total of 1M episodes. We then filter this dataset on two categories. First is triviality; the lc0 value head is quite strong, even without search, and so its immediately-preferred action is often the best one. To ensure that the controller does not simply fit this trivial baseline, we filter all trees where the difference between maximum possible reward and minimum possible reward (excluding cost) is less than 0.1.

We also filter out a particular solution motif that promotes poor metacontrol. In particular, consider a tree with the following property: after a shallow search, it prefers move A. After a medium search,

it prefers move B. Finally, after a deep search, it again prefers move A. (Perhaps A looks initially attractive, then the medium search reveals a potential refutation, then the deep search in turn refutes the refutation.) In some sense — and under our backward induction labelling approach — the optimal halt point would be during the shallow search that initially preferred A, since this saves on planning cost — but intuitively, this doesn't really make sense. While A is indeed the best move, this can only be justified by the deep search, not the shallow search. It is, in essence, sort of a coincidence that the shallow and deep searches agree. As such, we filter out all trees that exhibit this ABA structure. Altogether, this leaves us with roughly 15% of the trees for controller training.

For each of the remaining tree-budget pairs, we compute the advantage, under the optimal policy, of planning versus acting $A(T_{:i}, b-i)$ at every step $i \in \{0, 1, \ldots, L\}$ and train the controller to predict it. Particularly, we use a hybrid loss:

$$\mathcal{L}_{\text{control}} = \mathcal{L}_{\text{MSE}} + 0.1\mathcal{L}_{\text{sign}} \tag{12}$$

where $\mathcal{L}_{\text{MSE}} = (A^* - \hat{A})^2$ is the standard mean square error loss on the oracle advantage A^* and the estimated advantage $\hat{A}$. On its own, it ensures that the estimated advantages are close to the true advantages. However, in many cases the true advantage is close to 0 (i.e., the decision threshold), so it is possible to have low MSE but still be on the wrong side of the threshold. We therefore augment it with an auxiliary loss that encourages the emitted advantage to also have the right sign. Letting $y = \mathbb{I}_{A^*>0}$ and σ be the sigmoid function,

$$\mathcal{L}_{\text{sign}} = -y \log \sigma(\hat{A}) - (1-y)\log(1-\sigma(\hat{A})) \tag{13}$$

that is, we apply a binary cross entropy loss treating the sign of A^* as the label and $\hat{A}$ as a logit.

We used a cost function c that scaled inversely with the budget so that planning in low-budget regimes was expensive and planning in high-budget regimes was cheap. Let:

$$\psi(b) = (b+\tau)^{-(\eta-1)} \tag{14}$$

Then the cost function for planning one step at budget b is:

$$c(b) = \lambda_c \Big[\psi(b-1) - \psi(b)\Big] \tag{15}$$

In practice, our method is robust to any reasonable choice of c. This particular one has the nice property that it telescopes: the cumulative cost of planning from budget b to some other budget $b - \Delta_b$ is just:

$$c_{\Delta_b}(b) = \lambda_c \Big[\psi(b-\Delta_b) - \psi(b)\Big]$$

Below is a table of other relevant hyperparameters.

Table 2: Controller hyperparameters.

Hyperparameter	Value
Input	$[h_{\text{root}}; T]$
Architecture	MLP, 3 hidden layers of width 256, ReLU, scalar output
Optimizer	Adam
Learning rate	10^{-3}
Batch size	18,000
Epochs	20
λ_c	18.537
η	2.8
τ	2.5

C Appendix C: Baseline comparison descriptions

Handcrafted features, handcrafted rule:

- Best fixed k: controller always halts at a specific plan-action count k, selected as the regret-minimizing k across the whole dataset.
- Best fixed k per budget bucket: as above, but k is allowed to vary for each budget bucket.
- Best smart pruning threshold: smart pruning refers to halting when the remaining number of iterations is less than the difference in visit counts between the preferred and rank 2 actions (so no amount of further search can destabilize the winner). In some cases this is applied to remaining$/f$, for f a hyperparameter. Here we swept f and picked the regret-minimizing choice.
- KLD-gain: stop when the successive difference in KL-divergence across the MCTS visit count distribution falls below a threshold (optimized on the data).
- Action gap: Halt when the difference in estimated value between the preferred and rank 2 actions is greater than a threshold (optimized on the data).

Handcrafted features, learned rule:

- MLP on search statistics: an adjusted version of DS-MCTS (Lan et al., 2020), fit an MLP directly on a predefined set of search-related feature to predict the advantage of planning over acting. The features were: visit share (fraction of all search iterations passing through the node), mean value, value variance, visit share over the last half of search iterations, mean value over the last half of search iterations, value variance over the last half of search iterations.

Learned features, handcrafted rule: We briefly introduce the concept of value of information (VOI), based on the myopic value of perfect information rule developed in Dearden et al. (1998) for Q-learning and then adapted by Hay et al. (2012) for MCTS stopping. We will provide a brief summary here – for a full accounting see those papers. The basic principle is that we can, in some sense, measure the value of the planning action in terms of the increased reward attained by the agent as a result of that planning action. As a result, the utility of planning is dependent on its capability to actually alter future behaviour: planning actions that simply reinforce the current policy have utility 0, whereas planning actions that actually result in the agent behaving better have positive utility equal to the gain.

For example, say that the agent has completed t iterations of tree search and is considering which node to expand on iteration $t+1$. Furthermore, let α be the move with the highest estimated Q-value and β be the rank 2 move – and assume that, when search is terminated, the move will be chosen as $\arg\max Q$. Since α is already the favourite, expanding α further only changes the agent's policy if it brings $Q(\alpha)$ below $Q(\beta)$. Conversely, expanding $Q(\beta)$ only changes the agent's policy if it places it above $Q(\beta)$.

Myopic VOI asks: under the assumption that the next expansion of a move m would reveal its *true* value, how much is that expansion worth? If $m \neq \alpha$, then:

$$\text{VOI}(m) = \int_{Q_\alpha}^{\infty} P\Big(Q^{\text{true}}(m) = q\Big)\Big(q - Q(\alpha)\Big)dq \tag{16}$$

In essence, what is the probability that, due to this expansion, the true value of $m \neq \alpha$ is revealed to be greater than $Q(\alpha)$ and then how much additional reward would that net the agent? Note that the lower limit of this integral starts at $Q(\alpha)$ since if m is found to have any value lower than that, the agent will not change its policy and therefore the information has 0 value.

In contrast, if $m = \alpha$ then:

$$\text{VOI}(m) = \int_{-\infty}^{Q(\beta)} P\Big(Q^{\text{true}}(m) = q\Big)\Big(\beta - q\Big)dq \tag{17}$$

In essence, what is the probability that expanding $m = \alpha$ reveals it to be worse than β (thereby changing my preferences), and how much reward do I save by now preferring β? Similarly, the upper limit of this integral is $Q(\beta)$ since the agent's behaviour only changes if it adjusts its estimate of α below that.

In both cases of the VOI computation, the distribution over the expansion outcome (i.e., the $P(Q(m) = q)$ term) is the current posterior distribution over $Q(m)$. That is, to enable this rule, the agent must be estimating Q distributionally, rather than as a point estimate.

Now, the stopping rules:

- VOI $< \lambda \times$ cost: decode the value histograms for all root children from the tree embedding, and compute the myopic VOI for each of them. Stop when the VOI falls beneath the cost of the upcoming plan action, scaled by a constant λ (optimized on the data set).
- Decoded action gap: action gap, but based on the root children values decoded from the tree embedding.