Levin Tree Search with Context Models

Laurent Orseau[1] , **Marcus Hutter**[1] , **Levi H. S. Lelis**[2]

[1]Google DeepMind
[2]Department of Computing Science, University of Alberta, Canada
and Alberta Machine Intelligence Institute (Amii), Canada
{lorseau,mhutter}@google.com, levi.lelis@ualberta.ca

Abstract

Levin Tree Search (LTS) is a search algorithm that makes use of a policy (a probability distribution over actions) and comes with a theoretical guarantee on the number of expansions before reaching a goal node, depending on the quality of the policy. This guarantee can be used as a loss function, which we call the LTS loss, to optimize neural networks representing the policy (LTS+NN). In this work we show that the neural network can be substituted with parameterized context models originating from the online compression literature (LTS+CM). We show that the LTS loss is convex under this new model, which allows for using standard convex optimization tools, and obtain convergence guarantees to the optimal parameters in an online setting for a given set of solution trajectories — guarantees that cannot be provided for neural networks. The new LTS+CM algorithm compares favorably against LTS+NN on several benchmarks: Sokoban (Boxoban), The Witness, and the 24-Sliding Tile puzzle (STP). The difference is particularly large on STP, where LTS+NN fails to solve most of the test instances while LTS+CM solves each test instance in a fraction of a second. Furthermore, we show that LTS+CM is able to learn a policy that solves the Rubik's cube in only a few hundred expansions, which considerably improves upon previous machine learning techniques.

1 Introduction

We[1] consider the problem of solving a set of deterministic single-agent search problems of a given domain, by starting with little prior domain-specific knowledge. We focus on algorithms that learn from previously solved instances to help solve the remaining ones. We consider the satisficing setting where solvers should (learn to) quickly find a solution, rather than to minimize the cost of the returned solutions.

Levin Tree Search (LevinTS, LTS) is a tree search algorithm for this setup that uses a policy, *i.e.*, a probability distri-

---

[1]Extended version of the IJCAI 2023 paper. Source code at: https://github.com/deepmind/levintreesearch_cm.

bution over actions, to guide the search [Orseau *et al.*, 2018]. LTS has a guarantee on the number of search steps required before finding a solution, which depends on the probability of the corresponding sequence of actions as assigned by the policy. Orseau and Lelis [2021] showed that this guarantee can be used as a loss function. This LTS loss is used to optimize a neural-network (NN) policy in the context of the Bootstrap search-and-learn process [Jabbari Arfaee *et al.*, 2011]: The NN policy is used in LTS (LTS+NN) to iteratively solve an increasing number of problems from a given set, optimizing the parameters of the NN when new problems are solved to improve the policy by minimizing the LTS loss.

One constant outstanding issue with NNs is that the loss function (whether quadratic, log loss, LTS loss, etc.) is almost never convex in the NN's parameters. Still, most of the time NNs are trained using online convex optimization algorithms, such as stochastic gradient descent, Adagrad [Duchi *et al.*, 2011], and its descendants. Such algorithms often come with strong convergence or regret guarantees that only hold under convexity assumptions, and can help to understand the effect of various quantities (number of parameters, etc.) on the learning speed [Zinkevich, 2003; Hazan, 2016; Boyd and Vandenberghe, 2004]. In this paper we present parameterized context models for policies that are convex with respect to the model's parameters for the LTS loss. Such models guarantee that we obtain an optimal policy in terms of LTS loss for a given set of training trajectories — a guarantee NNs do not have.

The context models we introduce for learning policies are based on the models from the online data compression literature [Rissanen, 1983; Willems *et al.*, 1995]. Our context models are composed of a set of contexts, where each context is associated with a probability distribution over actions. These distributions are combined using product-of-experts [Hinton, 2002] to produce the policy used during the LTS search. The expressive power of product-of-experts comes mainly from the ability of each expert to (possibly softly) veto a particular option by assigning it a low probability. A similar combination using geometric mixing [Mattern, 2013; Matthew, 2005] (a geometrically-parameterized variant of product-of-experts) in a multi-layer architecture has already proved competitive with NNs in classification, regression and density modelling tasks [Veness *et al.*, 2017; Veness *et al.*, 2021; Budden *et al.*, 2020]. In our work the context distributions

are fully parameterized and we show that the LTS loss is convex for this parameterization.

In their experiments, Orseau and Lelis [2021] showed that LTS+NN performs well on two of the three evaluated domains (Sokoban and The Witness), but fails to learn a policy for the 24-Sliding Tile Puzzle (STP). We show that LTS with context models optimized with the LTS loss within the Bootstrap process is able to learn a strong policy for all three domains evaluated, including the STP. We also show that LTS using context models is able to learn a policy that allows it to find solutions to random instances of the Rubik's Cube with only a few hundred expansions. In the context of satisficing planning, this is a major improvement over previous machine-learning-based approaches, which require hundreds of thousands expansions to solve instances of the Rubik's Cube.

We start with giving some notation and the problem definition (Section 2), before describing the LTS algorithm, for which we also provide a new lower bound on the number of node expansions (Section 3). Then, we describe parameterized context models and explain why we can expect them to work well when using product-of-experts (Section 4), before showing that the LTS loss function is convex for this parameterization (Section 5) and considering theoretical implications. Finally we present the experimental results (Section 6) before concluding (Section 7).

2 Notation and Problem Definition

A table of notation can be found in Appendix I. We write $[t] = \{1, 2, \ldots t\}$ for a natural number t. The set of nodes is $\mathcal{N}$ and is a forest, where each tree in the forest represents a search problem with the root being the initial configuration of the problem. The set of children of a node $n \in \mathcal{N}$ is $\mathcal{C}(n)$ and its parent is $\mathrm{par}(n)$; if a node has no parent it is a root node. The set of ancestors of a node is $\mathrm{anc}(n)$ and is the transitive closure of $\mathrm{par}(\cdot)$; we also define $\mathrm{anc}_+(n) = \mathrm{anc}(n) \cup \{n\}$. Similarly, $\mathrm{desc}(n)$ is the set of the descendants of n, and $\mathrm{desc}_+(n) = \mathrm{desc}(n) \cup \{n\}$. The depth of a node is $d(n) = |\mathrm{anc}(n)|$, and so the depth of a root node is 0. The root $\mathrm{root}(n)$ of a node n is the single node $n_0 \in \mathrm{anc}_+(n)$ such that n_0 is a root. A set of nodes $\mathcal{N}'$ is a tree in the forest $\mathcal{N}$ if and only if there is a node $n^0 \in \mathcal{N}'$ such that $\bigcup_{n \in \mathcal{N}'} \mathrm{root}(n) = \{n^0\}$. Let $\mathcal{N}^0 = \bigcup_{n \in \mathcal{N}} \mathrm{root}(n)$ be the set of all root nodes. We write $n_{[j]}$ for the node at depth $j \in [d(n)]$ on the path from $\mathrm{root}(n) = n_{[0]}$ to $n = n_{[d(n)]}$. Let $\mathcal{N}^* \subseteq \mathcal{N}$ be the set of all *solution* nodes, and we write $\mathcal{N}^*(n) = \mathcal{N}^* \cap \mathrm{desc}_+(n)$ for the set of solution nodes under n. A *policy* π is such that for all $n \in \mathcal{N}$ and for all $n' \in \mathcal{C}(n) : \pi(n' \mid n) \geq 0$ and $\sum_{n' \in \mathcal{C}(n)} \pi(n' \mid n) \leq 1$. The policy is called *proper* if the latter holds as an equality. We define, for all $n' \in \mathcal{C}(n)$, $\pi(n') = \pi(n)\pi(n' \mid n)$ recursively and $\pi(n) = 1$ if n is a root node.

Edges between nodes are labeled with *actions* and the children of any node all have different labels, but different nodes can have overlapping sets of actions. The set of all edge labels is $\mathcal{A}$. Let $a(n)$ be the label of the edge from $\mathrm{par}(n)$ to n, and let $\mathcal{A}(n)$ be the set of edge labels for the edges from node n to its children. Then $n \neq n' \wedge \mathrm{par}(n) = \mathrm{par}(n')$ implies $a(n) \neq a(n')$.

Starting at a given root node n^0, a tree search algorithm expands a set $\mathcal{N}' \subseteq \mathrm{desc}_+(n^0)$ until it finds a solution node in $\mathcal{N}^*(n^0)$. In this paper, given a set of root nodes, we are interested in parameterized algorithms that attempt to minimize the cumulative number of nodes that are expanded before finding a solution node for each root node, by improving the parameters of the algorithm from found solutions, and with only little prior domain-specific knowledge.

3 Levin Tree Search

Levin Tree Search (LevinTS, which we abbreviate to LTS here) is a tree/graph search algorithm based on best-first search [Pearl, 1984] that uses the cost function[2] $n \mapsto d(n)/\pi(n)$ [Orseau *et al.*, 2018], which, for convenience, we abbreviate as $\frac{d}{\pi}(n)$. That is, since $\frac{d}{\pi}(\cdot)$ is monotonically increasing from parent to child, LTS expands all nodes by increasing order of $\frac{d}{\pi}(\cdot)$ (Theorem 2, Orseau *et al.* [2018]).

Theorem 1 (LTS upper bound, adapted from Orseau *et al.* [2018], Theorem 3). *Let π be a policy. For any node $n^* \in \mathcal{N}$, let $\overline{\mathcal{N}}(n^*) = \{n \in \mathcal{N} : \mathrm{root}(n) = \mathrm{root}(n^*) \wedge \frac{d}{\pi}(n) \leq \frac{d}{\pi}(n^*)\}$ be the set of nodes within the same tree with cost at most that of n^*. Then*

$$|\overline{\mathcal{N}}(n^*)| \leq 1 + \frac{d(n^*)}{\pi(n^*)} \, .$$

Proof. Let $\mathcal{L}$ be the set of leaves of $\overline{\mathcal{N}}(n^*)$, then

$$|\overline{\mathcal{N}}(n^*)| \leq 1 + \sum_{n \in \mathcal{L}} d(n) = 1 + \sum_{n \in \mathcal{L}} \pi(n)\tfrac{d}{\pi}(n)$$
$$\leq 1 + \sum_{n \in \mathcal{L}} \pi(n)\tfrac{d}{\pi}(n^*) \leq 1 + \tfrac{d}{\pi}(n^*) \, ,$$

where we used Lemma 10 (in Appendix) on the last inequality. □

The consequence is that LTS started at $\mathrm{root}(n^*)$ expands at most $1 + \frac{d}{\pi}(n^*)$ nodes before reaching n^*.

Orseau and Lelis [2021] also provides a related lower bound showing that, for any policy, there are sets of problems where any algorithm needs to expand $\Omega(\frac{d}{\pi}(n^*))$ nodes before reaching some node n^* in the worst case. They also turn the guarantee of Theorem 1 into a loss function, used to optimize the parameters of a neural network. Let $\mathcal{N}'$ be a set of solution nodes whose roots are all different, define the *LTS loss function*:

$$L(\mathcal{N}') = \sum_{n \in \mathcal{N}'} \tfrac{d}{\pi}(n) \tag{1}$$

which upper bounds the total search time of LTS to reach all nodes in $\mathcal{N}'$. Equation (1) is the loss function used in Algorithm 2 (Appendix A) to optimize the policy — but a more precise definition for context models will be given later. To further justify the use of this loss function, we provide a lower bound on the number of expansions that LTS must perform before reaching an (unknown) target node.

---

[2] Orseau *et al.* [2018] actually use the cost function $(d(n) + 1)/\pi(n)$. Here we use $d(n)/\pi(n)$ instead which is actually (very) slightly better and makes the notation simpler. All original results can be straightforwardly adapted.

Theorem 2 (Informal lower bound). *For a proper policy π and any node n^*, the number of nodes whose $\frac{d}{\pi}$ cost is at most that of n^* is at least $[\frac{1}{\bar{d}}\frac{d}{\pi}(n^*) - 1]/(|\mathcal{A}| - 1)$, where $\bar{d} - 1$ is the average depth of the leaves of those nodes.*

A more formal theorem is given in Appendix B.

Example 3. *For a binary tree with a uniform policy, since $\bar{d} = d(n^*) + 1$, the lower bound gives $2^d d/(d+1) - 1$ nodes for a node n^* at depth d and of probability 2^{-d}, which is quite tight since the tree has $2^d - 1$ nodes. The upper bound $1 + d2^d$ is slightly looser.*

Remark 4. *Even though pruning (such as state-equivalence pruning) can make the policy improper, in which case the lower bound does not hold and the upper bound can be loose, optimizing the parameters of the policy for the upper bound still makes sense, since pruning can be seen as a feature placed on top of the policy — that is, the policy is optimized as if pruning is not used. It must be noted that for optimization Orseau and Lelis [2021] (Section 4) use the log gradient trick to replace the upper bound loss with the actual number of expansions in an attempt to account for pruning; as the results of this paper suggest, it is not clear whether one should account for the actual number of expansions while optimizing the model.*

4 Context Models

Now we consider that the policy π has some parameters $\beta \in \mathcal{B}$ (where $\mathcal{B} \subseteq \mathbb{R}^k$ for some k, which will be made more precise later) and we write $\pi(\cdot; \beta)$ when the parameters are relevant to the discussion. As mentioned in the introduction, we want the LTS loss function of Eq. (1) to be convex in the policy's parameters, which means that we cannot use just any policy — in particular this rules out deep neural networks. Instead, we use context models, which have been widely used in online prediction and compression (*e.g.*, [Rissanen, 1983; Willems *et al.*, 1995; Matthew, 2005; Veness *et al.*, 2021]).

The set of contexts is $\mathcal{Q}$. A context is either active or inactive at a given node in the tree. At each node n, the set of active contexts is $\mathcal{Q}(n)$, and the policy's prediction at n depends only on these active contexts.

Similarly to patterns in pattern databases [Culberson and Schaeffer, 1998], we organize contexts in sets of mutually exclusive contexts, called *mutex sets*, and each context belongs to exactly one mutex set. The set of mutex sets is $\mathcal{M}$. For every mutex set $M \in \mathcal{M}$, for every node n, at most one context is active per mutex set. In this paper we are in the case where *exactly* one context is active per mutex set, which is what happens when searching with multiple pattern databases, where each pattern database provides a single pattern for a given node in the tree. When designing contexts, it is often more natural to directly design mutex sets. See Figure 1 for an example, omitting the bottom parts of (b) and (d) for now.

To each context $c \in \mathcal{Q}$ we associate a *predictor* $p_c : \mathcal{A} \to [0, 1]$ which is a (parameterized) categorical probability distribution over edge labels that will be optimized from training data — the learning part will be explained in Section 5.1.

To combine the predictions of the active contexts at some node n, we take their renormalized product, as an instance of product-of-experts [Hinton, 2002]:

$$\forall a \in \mathcal{A}(n) : p_\times(n, a) = \frac{\prod_{c \in \mathcal{Q}(n)} p_c(a)}{\sum_{a' \in \mathcal{A}(n)} \prod_{c \in \mathcal{Q}(n)} p_c(a')} \quad (2)$$

We refer to the operation of Eq. (2) as *product mixing*, by relation to geometric mixing [Mattern, 2013], a closely related operation. Then, one can use $p_\times(n, a)$ to define the policy $\pi(n'|n) = p_\times(n, a(n'))$ to be used with LTS.

The choice of this particular aggregation of the individual predictions is best explained by the following example.

Example 5 (Wisdom of the product-of-experts crowd). *Figure 1 (a) and (b) displays a simple maze environment where the agent is coming from the left. The only sensible action is to go Up (toward the exit), but no single context sees the whole picture. Instead, they see only individual cells around the agent, and one context also sees (only) the previous action (which is Right). The first two contexts only see empty cells to the left and top of the agent, and are uninformative (uniform probability distributions) about which action to take. But the next three contexts, while not knowing what to do, know what* not *to do. When aggregating these predictions with product mixing, only one action remains with high probability: Up.*

Example 6 (Generalization and specialisation). *Another advantage of product mixing is its ability to make use of both general predictors and specialized predictors. Consider a mutex set composed of m contexts, and assume we have a total of M data points (nodes on solution trajectories). Due to the mutual exclusion nature of mutex sets, these M data points must be partitioned among the m contexts. Assuming for simplicity a mostly uniform partitioning, then each context receives approximately M/m data points to learn from. Consider the mutex sets in Fig. 1 (b): The first 4 mutex sets have size 3 (each context can see a wall, an empty cell or the goal) and the last one has size 4. These are very small sizes and thus the parameters of the contexts predictors should quickly see enough data to learn an accurate distribution. However, while accurate, the distribution can hardly be* specific*, and each predictor alone is not sufficient to obtain a nearly-deterministic policy — though fortunately product mixing helps with that. Now compare with the 2-cross mutex set in Fig. 1 (d), and assume that cells outside the grid are walls. A quick calculation, assuming only one goal cell, gives that it should contain a little less than 1280 different contexts. Each of these contexts thus receives less data to learn from on average than the contexts in (b), but also sees more information from the environment which may lead to more specific (less entropic) distributions, as is the case in situation (c).*

Remark 7. *A predictor that has a uniform distribution has no effect within a product mixture. Hence, adding new predictors initialized with uniform predictions does not change the policy, and similarly, if a context does not happen to be useful to learn a good policy, optimization will push its weights toward the uniform distribution, implicitly discarding it.*

Hence, product mixing is able to take advantage of both general contexts that occur in many situations and specialised contexts tailored to specific situations — and anything in-between.
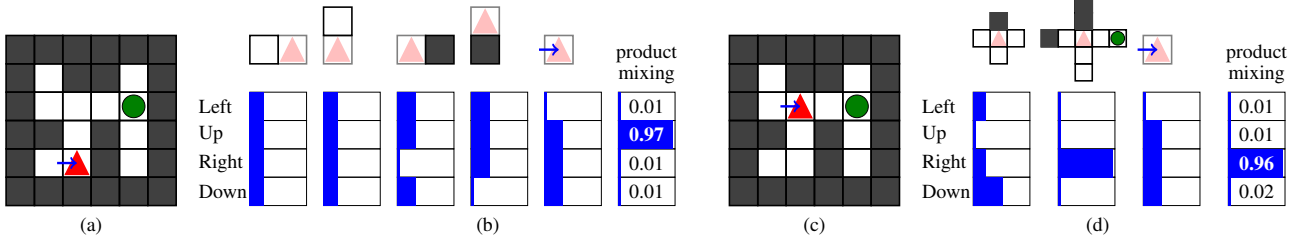
Figure 1: (a) A simple maze environment. The dark gray cells are walls, the green circle is a goal. The blue arrow symbolizes the fact that the agent (red triangle) is coming from the left. (b) A simple context model with five mutex sets: One mutex set for each of the four cells around the triangle, and one mutex set for the last chosen action. Each of the first four mutex sets contains three contexts (wall, empty cell, goal), and the last mutex set contains four contexts (one for each action). The 5 active contexts (one per mutex set) for the situation shown in (a) are depicted at the top, while their individual probability predictions are the horizontal blue bars for each of the four actions. The last column is the resulting product mixing prediction of the 5 predictions. No individual context prediction exceeds 1/3 for any action, yet the product mixing prediction is close to 1 for the action Up. (c) Another situation. (d) A different set of mutex sets for the situation in (c): A 1-cross around the agent, a 2-cross around the agent, and the last action. The specialized 2-cross context is certain that the correct action is Right, despite the two other contexts together giving more weight to action Down. The resulting product mixing gives high probability to Right, showing that, in product mixing, specialized contexts can take precedence over less-certain more-general contexts.

Our LTS with context models algorithm is given in Algorithm 1, building upon the one by Orseau and Lelis [2021] with a few differences. As mentioned earlier, it is a best-first search algorithm and uses a priority queue to maintain the nodes to be expanded next. It is also budgeted and returns `"budget_reached"` if too many nodes have been expanded. It returns `"no_solution"` if all nodes have been expanded without reaching a solution node — assuming safe pruning or no pruning. Safe pruning (using `visited_states`) can be performed if the policy is Markovian [Orseau *et al.*, 2018], which is the case in particular when the set of active contexts $\mathcal{Q}(n)$ depends only on `state(n)`. The algorithm assumes the existence of application-specific `state` and `state_transition` functions, such that `state(n′) = state_transition( state(n), a(n′))` for all $n′ \in \mathcal{C}(n)$. Note that with context models the prediction $\pi(n′ \mid n)$ depends on the active contexts $\mathcal{Q}(n)$ but *not* on the state of a child node. This allows us to delay the state transition until the child is extracted from the queue, saving up to a branching factor of state transitions (see also [Agostinelli *et al.*, 2021]).

Remark 8. *In practice, usually a mutex set can be implemented as a hashtable as for pattern databases: the active context is read from the current state of the environment, and the corresponding predictor is retrieved from the hashtable. This allows for a computational cost of $O(\log |M|)$ per mutex set M, or even $O(1)$ with perfect hash functions, and thus $O(\sum_{M \in \mathcal{M}} \log |M|)$ which is much smaller than $|\mathcal{Q}|$. Using an imperfect hashtable, only the contexts that appear on the paths to the found solution nodes need to be stored.*

5 Convexity

Because the LTS loss in Eq. (1) is different from the log loss [Cesa-Bianchi and Lugosi, 2006] (due to the sum in-between the products), optimization does *not* reduce to maximum likelihood estimation. However, we show that convexity in the log loss implies convexity in the LTS loss. This

means, in particular, that if a probability distribution is log-concave (such as all the members of the exponential family), that is, the log loss for such models is convex, then the LTS loss is convex in these parameters, too.

First we show that every sequence of functions with a convex log loss also have convex *inverse* loss and LTS loss.

Theorem 9 (Log loss to inverse loss convexity). *Let $f_1, f_2, \dots f_s$ be a sequence of positive functions with $f_i : \mathbb{R}^n \to (0, \infty)$ for all $i \in [s]$ and such that $\beta \mapsto -\log f_i(\beta)$ is convex for each $i \in [s]$, then $L(\beta) = \sum_k \frac{1}{\prod_t f_{k,t}(\beta)}$ is convex, where each (k, t) corresponds to a unique index in $[s]$.*

The proof is in Appendix E.1. For a policy $\pi(\cdot; \beta)$ parameterized by β, the LTS loss in Eq. (1) is $L_{\mathcal{N}'}(\beta) = \sum_{k \in \mathcal{N}'} d(n^k)/\pi(n^k; \beta)$, and its convexity follows from Theorem 9 by taking $f_{k,0}(\cdot) = 1/d(n^k)$, and $f_{k,t}(\beta) = \pi(n^k_{[t]}|n^k_{[t-1]}; \beta)$ such that $\prod_{t=1}^{d(n^k)} f_{k,t}(\beta) = \pi(n^k; \beta)$.

Theorem 9 means that many tools of compression and on-line prediction in the log loss can be transferred to the LTS loss case. In particular, when there is only one mutex set ($|\mathcal{M}| = 1$), the f_i are simple categorical distributions, that is, $f_i(\beta) = \beta_{j_t}$ for some index j_t, and thus $-\log f_i$ is a convex function, so the corresponding LTS loss is convex too. Unfortunately, the LTS loss function for such a model is convex in β only when there is only one mutex set, $|\mathcal{M}| = 1$. Fortunately, it becomes convex for $|\mathcal{M}| \geq 1$ when we reparameterize the context predictors with $\beta \rightsquigarrow \exp \beta$.

Let $\beta_{c,a} \in [\ln \varepsilon_{\text{low}}, 0]$ be the value of the parameter of the predictor for context c for the edge label a. Then the prediction of a context c is defined as

$$\forall a \in \mathcal{A}(n) : p_c(a; \beta) = \frac{\exp(\beta_{c,a})}{\sum_{a' \in \mathcal{A}(n)} \exp(\beta_{c,a'})}. \quad (3)$$

We can also now make precise the definition of $\mathcal{B}$: $\mathcal{B} = [\ln \varepsilon_{\text{low}}, 0]^{|\mathcal{Q}| \times A}$, and note that $p_c(a; \beta) \geq \varepsilon_{\text{low}}/|\mathcal{A}(n)|$. Similarly to geometric mixing [Mattern, 2013; Mattern, 2016], it can be proven that context models have a convex log loss,

Algorithm 1 Budgeted LTS with context models. Returns a solution node if any is found, or `"budget_reached"` or `"no_solution"`.

```
# n⁰: root node
# B: node expansion budget
# β: parameters of the context models
def LTS+CM(n⁰, B, β):
  q = priority_queue()
  # tuple: {d/π, d, π_n, node, state, action}
  tup = {0, 0, 1, n⁰, state(n⁰), False}
  q.insert(tup) # insert root node/state
  visited_states = {} # dict: state(n) -> π(n)
  repeat forever:
    if q is empty: return "no_solution"
    # Extract the tuple with minimum cost d/π
    d/π n, d, π_n, n, s_parent, a = q.extract_min()
    if n ∈ N*: return n # solution found
    s = state_transition(s_parent, a) if a else
        s_parent
    π_s = visited_states.get(s, default=0)
    # Note: BFS ensures d/π(n_s) ≤ d/π(n); s = state(n_s)
    # Optional: Prune the search if s is better
    if π_s ≥ π_n: continue
    else: visited_states.set(s, π_n)
    # Node expansion
    expanded += 1
    if expanded == B: return "budget_reached"

    Z = ∑_{a∈A(n)} ∏_{c∈Q(n)} p_c(a;β) # normalizer
    for n' ∈ C(n):
      a = a(n') # action
      # Product mixing of the active contexts'
          predictions
      p_×,a = (1/Z) ∏_{c∈Q(n)} p_c(a;β) # See Eq. (3)
      # Action probability, ε_mix ensures π_n' > 0
      π_n' = π_n((1-ε_mix)p_×,a + ε_mix/|A(n)|)
      q.insert({(d+1)/π_n', d+1, π_n', n', s, a})
```

and thus their LTS loss is also convex by Theorem 9. In Appendix E.2 we provide a more direct proof, and a generalization to the exponential family for finite sets of actions.

Plugging (3) into Eq. (2) and pushing the probabilities away from 0 with $\varepsilon_{\mathrm{mix}} > 0$ [Orseau *et al.*, 2018] we obtain the policy's probability for a child n' of n (*i.e.*, for the action $a(n')$ at node n) with parameters β:

$$p_\times(n, a; \beta) = \frac{\exp(\sum_{c\in\mathcal{Q}(n)} \beta_{c,a})}{\sum_{a'\in\mathcal{A}(n)} \exp\left(\sum_{c\in\mathcal{Q}(n)} \beta_{c,a'}\right)}, \quad (4)$$

$$\pi(n' \mid n; \beta) = (1-\varepsilon_{\mathrm{mix}})p_\times(n, a(n'); \beta) + \frac{\varepsilon_{\mathrm{mix}}}{|\mathcal{A}(n)|}. \quad (5)$$

5.1 Optimization

We can now give a more explicit form of the LTS loss function of Eq. (1) for context models with a dependency on the

parameters β, for a set of solution nodes $\mathcal{N}'$ assumed to all have different roots:

$$L(\mathcal{N}', \beta) = \sum_{n\in\mathcal{N}'} \ell(n, \beta), \quad (6)$$

$$\ell(n, \beta) = \frac{d(n)}{\pi(n; \beta)} = \frac{d(n)}{\prod_{j=0}^{d(n)-1} \pi(n_{[j+1]}|n_{[j]}; \beta)} \quad (7)$$

$$= d(n) \prod_{j=0}^{d(n)-1} \sum_{a'\in\mathcal{A}(n_{[j]})} \exp\left(\sum_{c\in\mathcal{Q}(n_{[j]})} \beta_{c,a'} - \beta_{c,a(n_{[j+1]})}\right)$$

where $a(n_{[j+1]})$ should be read as the action chosen at step j, and the last equality follows from Eqs. (4) and (5) where we take $\varepsilon_{\mathrm{mix}} = 0$ during optimization. Recall that this loss function L gives an upper bound on the total search time (in node expansions) required for LTS to find all the solutions $\mathcal{N}'$ for their corresponding problems (root nodes), and thus optimizing the parameters corresponds to optimizing the search time.

5.2 Online Search-and-Learn Guarantees

Suppose that at each time step $t = 1, 2\ldots$, the learner receives a problem n_t^0 (a root node) and uses LTS with parameters $\beta^t \in \mathcal{B}$ until it finds a solution node $n_t \in \mathcal{N}^*(n_t^0)$. The parameters are then updated using n_t (and previous nodes) and the next step $t + 1$ begins.

Let $\mathcal{N}_t = (n_1, \ldots, n_t)$ be the sequence of found solution nodes. For the loss function of Eq. (6), after t found solution nodes, the optimal parameters *in hindsight* are $\beta_t^* = \operatorname{argmin}_{\beta\in\mathcal{B}} L(\mathcal{N}_t, \beta)$. We want to know how the learner fares against β_t^* — which is a moving target as t increases. The *regret* [Hazan, 2016] at step t is the cumulative difference between the loss incurred by the learner with its time varying parameters $\beta^i, i = 1, 2, \ldots, t$, and the loss when using the optimum parameters in hindsight β_t^*:

$$\mathcal{R}(\mathcal{N}_t) = \sum_{i\in[t]} \ell(n_i, \beta^i) - L(\mathcal{N}_t, \beta_t^*).$$

A straightforward implication of the convexity of Eq. (7) is that we can use Online Gradient Descent (OGD) [Zinkevich, 2003] or some of its many variants such as Adagrad [Duchi *et al.*, 2011] and ensure that the algorithm incurs a regret of $\mathcal{R}(\mathcal{N}_t) = O(|\mathcal{A}| |\mathcal{Q}| G\sqrt{t} \ln \frac{1}{\varepsilon_{\mathrm{low}}})$, where G is the largest observed gradient in infinite norm[3] and when using quadratic regularization. Regret bounds are related to the learning speed (the smaller the bound, the faster the learning), that is, roughly speaking, how fast the parameters converge to their optimal values for the same sequence of solution nodes. Such a regret bound (assuming it is tight enough) also allows to observe the impact of the different quantities on the regret, such as the number of contexts $|\mathcal{Q}|$, or $\varepsilon_{\mathrm{low}}$.

OGD and its many variants are computationally efficient as they take $O(d(n)|\mathcal{A}| |\mathcal{M}|)$ computation time per solution node n, but they are not very data efficient, due to the *linearization* of the loss function — the so-called 'gradient

---

[3]The dependency on the largest gradient can be softened significantly, *e.g.*, with Adagrad and sporadic resets of the learning rates.

trick' [Cesa-Bianchi and Lugosi, 2006]. To make the most of the data, we avoid linearization by sequentially minimizing the full regularized loss function $L(\mathcal{N}_t, \cdot) + R(\cdot)$ where $R(\beta)$ is a convex regularization function. That is, at each step, we set:

$$\beta^{t+1} = \underset{\beta \in \mathcal{B}}{\operatorname{argmin}} \, L(\mathcal{N}_t, \beta) + R(\beta) \qquad (8)$$

which can be solved using standard convex optimization techniques (see Appendix C) [Boyd and Vandenberghe, 2004]. This update is known as (non-linearized) Follow the Leader (FTL) which automatically adapts to local strong convexity and has a fast $O(\log T)$ regret without tuning a learning rate [Shalev-Shwartz, 2007], except that we add regularization to avoid overfitting which FTL suffers from. Unfortunately, solving Eq. (8) even approximately at each step is too computational costly, so we amortize this cost by delaying updates (see below), which of course incurs a learning cost, *e.g.*, [Joulani *et al.*, 2013].

6 Experiments

As with previous work, in the experiments we use the LTS algorithm with context models (Algorithm 1) within the search-and-learn loop of the Bootstrap process [Jabbari Arfaee *et al.*, 2011] to solve a dataset of problems, then test the learned model on a separate test set. See Appendix A for more details. Note that the Bootstrap process is a little different from the online learning setting, so the theoretical guarantees mentioned above may not carry over strictly — this analysis is left for future work.,

This allows us to compare LTS with context models (LTS+CM) in particular with previous results using LTS with neural networks (LTS+NN) [Guez *et al.*, 2019; Orseau and Lelis, 2021] on three domains. We also train LTS+CM to solve the Rubik's cube and compare with other approaches.

LTS+NN's domains. We foremost compare LTS with context models (LTS+CM) with LTS with a convolutional neural network [Orseau and Lelis, 2021] (LTS+NN) on the three domains where the latter was tested: (a) Sokoban (Boxoban) [Guez *et al.*, 2018] on the standard 1000 test problems, a PSPACE-hard puzzle [Culberson, 1999] where the player must push boxes onto goal positions while avoiding deadlocks, (b) The Witness, a color partitioning problem that is NP-hard in general [Abel *et al.*, 2020], and (c) the 24 (5×5) sliding-tile puzzle (STP), a sorting problem on a grid, for which finding short solutions is also NP-hard [Ratner and Warmuth, 1986]. As in previous work, we train LTS+CM on the same datasets of 50 000 problems each, with the same initial budget (2000 node expansions for Sokoban and The Witness, 7000 for STP) and stop as soon as the training set is entirely solved. Training LTS+CM for these domains took less than 2 hours each.

Harder Sokoban. Additionally, we compare algorithms on the Boxoban 'hard' set of 3332 problems. Guez *et al.* [2019] trained a convLSTM network on the medium-difficulty dataset (450k problems) with a standard actor-critic setup — not the LTS loss — and used LTS (hence LTS+NN) at test time. The more recent ExPoSe algorithm [Mittal *et al.*,

2022] updates the parameters of a policy neural network[4] *during* the search, and is trained on both the medium set (450k problems) and the 'unfiltered' Boxoban set (900k problems) with solution trajectories obtained from an A* search.

Rubik's Cube. We also use LTS+CM to learn a fast policy for the Rubik's cube, with an initial budget of $B_1 = 21000$. We use a sequence of datasets containing 100k problems each, generated with a random walk of between m and $m' = m + 5$ moves from the solution, where m increases by steps of 5 from 0 to 50, after which we set $m' = m = 50$ for each new generated set. DeepCubeA [Agostinelli *et al.*, 2019] uses a fairly large neural network to learn in a supervised fashion from trajectories generated with a backward model of the environment, and Weighted A* is used to solve random test cubes. Their goal is to learn a policy that returns solutions of near-optimal length. By contrast, our goal is to learn a fast-solving policy. Allen *et al.* [2021] takes a completely different approach (no neural network) by learning a set of 'focused macro actions' which are meant to change the state as little as possible so as to mimic the so-called 'algorithms' that human experts use to solve the Rubik's cube. They use a rather small budget of 2 million actions to learn the macro actions, but also use the more informative goal-count scoring function (how many variables of the state have the correct value), while we only assume access to the more basic solved/unsolved function. As with previous work, we report solution lengths in the quarter-turn metric. Our test set contains 1000 cubes scrambled 100 times each — this is likely more than enough to generate random cubes [Korf, 1997] — and we expect the difficulty to match that of previous work.

Machine description. We used a single EPYC 7B12 (64 cores, 128 threads) server with 512GB of RAM without GPU. During training and testing, 64 problems are attempted concurrently — one problem per CPU core. Optimization uses 128 threads to calculate the loss, Jacobian and updates.

Hyperparameters. For all experiments we use $\varepsilon_{\text{low}} = 10^{-4}$, $\varepsilon_{\text{mix}} = 10^{-3}$, a quadratic regularization $R(\beta) = 5\|\beta - \beta_0\|^2$ where $\beta_0 = (1 - 1/A) \ln \varepsilon_{\text{low}}$ (see Appendix F). The convex optimization algorithm we use to solve Eq. (8) is detailed in Appendix C.
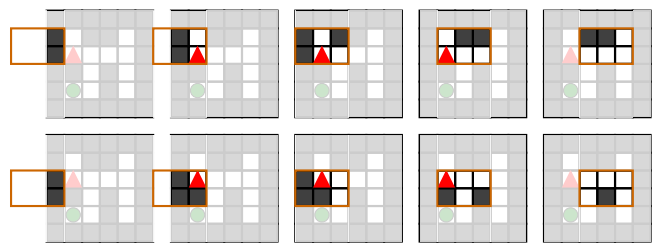

Figure 2: Example of a relative tiling of row span 2, column span 3, at maximum row distance 1 and maximum column distance 3 around the agent (red triangle). Each orange rectangle is a mutex set of at most 4^6 different contexts. A padding value can be chosen arbitrarily (such as the wall value) for cells outside the grid.

---

[4]The architecture of the neural network was not specified.

Domain	Algorithm	%solved	Length	Expansions	Time (ms)
Boxoban	LTS+CM (this work)	100.00	41.7	**2 132.3**	124
	LTS+NN [Orseau and Lelis, 2021]	100.00	**40.1**	2 640.4	19 500
The Witness	LTS+CM (this work)	100.00	15.5	**102.8**	9
	LTS+NN [Orseau and Lelis, 2021]	100.00	**14.8**	520.2	3 200
STP (24-puzzle)	LTS+CM (this work)	**100.00**	211.2	**5 667.4**	236
	LTS+NN [Orseau and Lelis, 2021]	0.90	*145.1*	*39 005.6*	*31 100*
Boxoban hard	LTS+CM (this work)	**100.00**	67.8	48 058.6	3 275
	LTS+NN [Guez *et al.*, 2019]	94.00	n/a	n/a	*3 600*
	ExPoSe [Mittal *et al.*, 2022]	97.30	n/a	n/a	n/a
Rubik's cube	LTS+CM (this work)	100.00	81.7	**498.0**	16
	DeepCubeA [Agostinelli *et al.*, 2019]	100.00	**21.5**	∼600 000.0	24 220
	GBFS(A+M) [Allen *et al.*, 2021]	100.00	378.0	†171 300.0	n/a

Table 1: Results on the test sets. The last 3 columns are the averages over the test instances. The first three domains allow for a fair comparison between LTS with context models and LTS with neural networks [Orseau and Lelis, 2021] using the same 50k training instances and initial budget. For the last two domains, comparison to prior work is more cursory and is provided for information only, in particular because the objective of DeepCubeA is to provide near-optimal-length solutions rather than fast solutions. The values for LTS+{CM,NN} all use a single CPU, no GPU (except for LTS+NN [Guez *et al.*, 2019]). DeepCubeA uses four high-end GPU cards. More results can be found in Table 2 in Appendix H. †Does not account for the cost of macro-actions.

Mutex sets. For Sokoban, STP, and The Witness we use several mutex sets of rectangular shapes at various distances around the agent (the player in Sokoban, the tip of the 'snake' in The Witness, the blank in STP), which we call *relative tilings*. An example of relative tiling is given in Fig. 2, and a more information can be found in Appendix G. For the Rubik's cube, each mutex set $\{i, j\}$ corresponds to the ordered colors of the two cubies (the small cubes that make up the Rubik's cube) at location i and j (such as the up-front-right corner and the back-right edge). There are 20 locations, hence 190 different mutex sets, and each of them contains at most 24^2 contexts (there are 8 corner cubies, each with 3 possible orientations, and 12 side cubies, each with 2 possible orientations). For all domains, to these mutex sets we add one mutex set for the last action, indicating the action the agent performed to reach the node; for Sokoban this includes whether the last action was a push. The first 3 domains all have 4 actions (up, down, left, right), and the Rubik's cube has 12 actions (a rotation of each face, in either direction).

Results. The algorithms are tested on test sets that are separate from the training sets, see Table 1. For the first three domains, LTS+CM performs better than LTS+NN, even solving all test instances of the STP while LTS+NN solves less than 1% of them. On The Witness, LTS+CM learns a policy that allows it to expand 5 times fewer nodes than LTS+NN. LTS+CM also solves all instances of the Boxoban hard set, by contrast to previous published work, and despite being trained only on 50k problems. On the Rubik's cube, LTS+CM learns a policy that is hundreds of times faster than previous work — though recall that DeepCubeA's objective of finding short solutions differs from ours. This may be surprising given how simple the contexts are — each context 'sees' only two cubies — and is a clear sign that product mixing is taking full advantage of the learned individual context predictions.

7 Conclusion

We have devised a parameterized policy for the Levin Tree Search (LTS) algorithm using product-of-experts of context models that ensures that the LTS loss function is convex. While neural networks — where convexity is almost certainly lost — have achieved impressive results recently, we show that our algorithm is competitive with published results, if not better.

Convexity allows us in particular to use convex optimization algorithms and to provide regret guarantees in the online learning setting. While this provides a good basis to work with, this notion of regret holds against any competitor that learns from the same set of *solution* nodes. The next question is how we can obtain an online search-and-learn regret guarantee against a competitor for the same set of *problems* (root nodes), for which the cumulative LTS loss is minimum across all sets of solution nodes for the same problems. And, if this happens to be unachievable, what intermediate regret setting could be considered? We believe these are important open research questions to tackle.

We have tried to design mutex sets that use only basic domain-specific knowledge (the input representation of agent-centered grid-worlds, or the cubie representation of the Rubik's cube), but in the future it would be interesting to also *learn to search* the space of possible context models — this would likely require more training data.

LTS with context models, as presented here, cannot directly make use of a value function or a heuristic function, however they could either be binarized into multiple mutex sets, or be used as in PHS* [Orseau and Lelis, 2021] to estimate the LTS cost at the solution, or be used as features since the loss function would still be convex (see Appendix C).

Acknowledgments

References

[Abel *et al.*, 2020] Zachary Abel, Jeffrey Bosboom, Michael J. Coulombe, Erik D. Demaine, Linus Hamilton, Adam Hesterberg, Justin Kopinsky, Jayson Lynch, Mikhail Rudoy, and Clemens Thielen. Who witnesses the witness? finding witnesses in the witness is hard and sometimes impossible. *Theor. Comput. Sci.*, 839:41–102, 2020.

[Agostinelli *et al.*, 2019] Forest Agostinelli, Stephen McAleer, Alexander Shmakov, and Pierre Baldi. Solving the rubik's cube with deep reinforcement learning and search. *Nature Machine Intelligence*, 1, 07 2019.

[Agostinelli *et al.*, 2021] Forest Agostinelli, Alexander Shmakov, Stephen McAleer, Roy Fox, and Pierre Baldi. A* search without expansions: Learning heuristic functions with deep q-networks, 2021. arXiv 2102.04518.

[Allen *et al.*, 2021] Cameron Allen, Michael Katz, Tim Klinger, George Konidaris, Matthew Riemer, and Gerald Tesauro. Efficient black-box planning using macro-actions with focused effects. In *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence*, pages 4024–4031, 2021.

[Boyd and Vandenberghe, 2004] Stephen Boyd and Lieven Vandenberghe. *Convex Optimization*. Cambridge University Press, Cambridge, England, 2004.

[Büchner *et al.*, 2022] Clemens Büchner, Patrick Ferber, Jendrik Seipp, and Malte Helmert. A comparison of abstraction heuristics for rubik's cube. In *ICAPS 2022 Workshop on Heuristics and Search for Domain-independent Planning*, 2022.

[Budden *et al.*, 2020] David Budden, Adam Marblestone, Eren Sezener, Tor Lattimore, Gregory Wayne, and Joel Veness. Gaussian gated linear networks. In *Advances in Neural Information Processing Systems*, volume 33, pages 16508–16519. Curran Associates, Inc., 2020.

[Cesa-Bianchi and Lugosi, 2006] Nicolo Cesa-Bianchi and Gabor Lugosi. *Prediction, Learning, and Games*. Cambridge University Press, New York, NY, USA, 2006.

[Cortés, 2006] Jorge Cortés. Finite-time convergent gradient flows with applications to network consensus. *Automatica*, 42(11):1993–2000, 2006.

[Culberson and Schaeffer, 1998] Joseph C. Culberson and Jonathan Schaeffer. Pattern databases. *Computational Intelligence*, 14(3):318–334, 1998.

[Culberson, 1999] Joseph C. Culberson. Sokoban is PSPACE-Complete. In *Fun With Algorithms*, pages 65–76, 1999.

[Duchi *et al.*, 2011] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(61):2121–2159, 2011.

[Ebendt and Drechsler, 2009] Rüdiger Ebendt and Rolf Drechsler. Weighted a* search – unifying view and application. *Artificial Intelligence*, 173(14):1310 – 1342, 2009.

[Frank and Wolfe, 1956] Marguerite Frank and Philip Wolfe. An algorithm for quadratic programming. *Naval Research Logistics Quarterly*, 3(1-2):95–110, 1956.

[Guez *et al.*, 2018] Arthur Guez, Mehdi Mirza, Karol Gregor, Rishabh Kabra, Sebastien Racaniere, Theophane Weber, David Raposo, Adam Santoro, Laurent Orseau, Tom Eccles, Greg Wayne, David Silver, Timothy Lillicrap, and Victor Valdes. An investigation of model-free planning: boxoban levels. https://github.com/deepmind/boxoban-levels/, 2018. Accessed: 2023-05-01.

[Guez *et al.*, 2019] Arthur Guez, Mehdi Mirza, Karol Gregor, Rishabh Kabra, Sebastien Racaniere, Theophane Weber, David Raposo, Adam Santoro, Laurent Orseau, Tom Eccles, Greg Wayne, David Silver, and Timothy Lillicrap. An investigation of model-free planning. In *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 2464–2473. PMLR, 2019.

[Hazan, 2016] Elad Hazan. Introduction to online convex optimization. *Foundations and Trends® in Optimization*, 2(3-4):157–325, 2016.

[Hinton, 2002] Geoffrey E. Hinton. Training Products of Experts by Minimizing Contrastive Divergence. *Neural Computation*, 14(8):1771–1800, 08 2002.

[Jabbari Arfaee *et al.*, 2011] S. Jabbari Arfaee, S. Zilles, and R. C. Holte. Learning heuristic functions for large state spaces. *Artificial Intelligence*, 175(16-17):2075–2098, 2011.

[Jaggi, 2013] Martin Jaggi. Revisiting Frank-Wolfe: Projection-free sparse convex optimization. In *Proceedings of the 30th International Conference on Machine Learning*, volume 28(1) of *Proceedings of Machine Learning Research*, pages 427–435, Atlanta, Georgia, USA, 17–19 Jun 2013. PMLR.

[Joulani *et al.*, 2013] Pooria Joulani, Andras Gyorgy, and Csaba Szepesvari. Online learning under delayed feedback. In *Proceedings of the 30th International Conference on Machine Learning*, volume 28(3) of *Proceedings of Machine Learning Research*, pages 1453–1461. PMLR, 2013.

[Korf, 1997] Richard E. Korf. Finding optimal solutions to rubik's cube using pattern databases. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence and Ninth Conference on Innovative Applications of Artificial Intelligence*, AAAI'97/IAAI'97, page 700–705. AAAI Press, 1997.

[Mattern, 2013] Christopher Mattern. Linear and geometric mixtures - analysis. *Proceedings of the Data Compression Conference*, pages 301–310, 02 2013.

[Mattern, 2016] Christopher Mattern. *On Statistical Data Compression*. PhD thesis, Technische Universität Ilmenau, Fakultät für Informatik und Automatisierung, Feb 2016.

[Matthew, 2005] V Mahoney Matthew. Adaptive weighing of context models for lossless data compression. *Florida Institute of Technology CS Dept, Technical Report CS-2005-16, https://www. cs. fit. edu/Projects/tech_reports/cs-2005-16. pdf*, 2005.

[Mittal *et al.*, 2022] Dixant Mittal, Siddharth Aravindan, and Wee Sun Lee. Expose: Combining state-based exploration with gradient-based online search, 2022. arXiv 2202.01461.

[Nesterov, 1983] Yurii Nesterov. A method for solving the convex programming problem with convergence rate $o(1/k^2)$. *Proceedings of the USSR Academy of Sciences*, 269:543–547, 1983.

[Orabona and Pál, 2018] Francesco Orabona and Dávid Pál. Scale-free online learning. *Theoretical Computer Science*, 716:50–69, 2018.

[Orseau and Hutter, 2021] Laurent Orseau and Marcus Hutter. Isotuning with applications to scale-free online learning, 2021.

[Orseau and Lelis, 2021] Laurent Orseau and Levi H. S. Lelis. Policy-guided heuristic search with guarantees. *Proceedings of the AAAI Conference on Artificial Intelligence*, 35(14):12382–12390, May 2021.

[Orseau *et al.*, 2018] Laurent Orseau, Levi Lelis, Tor Lattimore, and Theophane Weber. Single-agent policy tree search with guarantees. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018.

[Pearl, 1984] Judea Pearl. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley Longman Publishing Co., Inc., USA, 1984.

[Pohl, 1970] Ira Pohl. Heuristic search viewed as path finding in a graph. *Artificial Intelligence*, 1(3):193 – 204, 1970.

[Ratner and Warmuth, 1986] Daniel Ratner and Manfred Warmuth. Finding a shortest solution for the nxn extension of the 15-puzzle is intractable. In *Proceedings of the Fifth AAAI National Conference on Artificial Intelligence*, AAAI'86, page 168–172. AAAI Press, 1986.

[Rissanen, 1983] J. Rissanen. A universal data compression system. *IEEE Transactions on Information Theory*, 29(5):656–664, 1983.

[Shalev-Shwartz, 2007] Shai Shalev-Shwartz. *Online learning: Theory, algorithms, and applications*. PhD thesis, Hebrew University, Jerusalem, 2007.

[Sutton and Barto, 1998] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning : An Introduction*. MIT Press, 1998.

[Truong and Nguyen, 2021] Tuyen Truong and Hang-Tuan Nguyen. Backtracking gradient descent method and some applications in large scale optimisation. part 2: Algorithms and experiments. *Applied Mathematics & Optimization*, 84:1–30, 12 2021.

[Veness *et al.*, 2017] Joel Veness, Tor Lattimore, Avishkar Bhoopchand, Agnieszka Grabska-Barwinska, Christopher Mattern, and Peter Toth. Online learning with gated linear networks, 2017. arXiv 1712.01897.

[Veness *et al.*, 2021] Joel Veness, Tor Lattimore, David Budden, Avishkar Bhoopchand, Christopher Mattern, Agnieszka Grabska-Barwinska, Eren Sezener, Jianan Wang, Peter Toth, Simon Schmitt, and Marcus Hutter. Gated linear networks. *Proceedings of the AAAI Conference on Artificial Intelligence*, 35(11):10015–10023, May 2021.

[Willems *et al.*, 1995] Frans M. J. Willems, Yuri M. Shtarkov, and Tjalling J. Tjalkens. The context tree weighting method: Basic properties. *IEEE Transactions on Information Theory*, 41:653–664, 1995.

[Zinkevich, 2003] Martin Zinkevich. Online convex programming and generalized infinitesimal gradient ascent. In *Proceedings of the Twentieth International Conference on Machine Learning*, pages 928–935, 2003.

A Bootstrap Algorithm Details

Algorithm 2 Bootstrap using `LTS_CM` (given in Algorithm 1), which returns `"budget_reached"` when the number of nodes expanded reaches B_t, or returns a solution node $n^* \in \mathcal{N}^*(n^0)$ if it reaches n^*, or returns `"no_solution"` if all nodes have been expanded without exhausting the budget and without reaching a solution node, which means that the problem has no solution.

```
# 𝒩⁰: set of root nodes = problems
# B₁: initial budget
# π₁: initial policy
# Returns the set of solution nodes
def Bootstrap_with_LTS(𝒩⁰, B₁, π₁):
  solns = {} # dictionary of problem -> solution
  for t = 1, 2, ...:
    for each n⁰ ∈ 𝒩⁰:
      result = LTS_CM(n⁰, Bₜ, βₜ) # search
      if result is "no_solution": 𝒩⁰ ← 𝒩⁰ \ {n⁰}
      if result is a node n*: soln[n⁰] = n*
    if len(soln) = |𝒩⁰|: return soln
    # Update the parameters of the model
    βₜ₊₁ ≈ argmin_{β∈ℬ} L(soln.values(), β) + R(β)
    choose budget Bₜ₊₁
```

Orseau and Lelis [2021] use a variant of the Bootstrap process [Jabbari Arfaee *et al.*, 2011] to iteratively solve a set of problems while improving the policy based on the solutions for the already solved problems. See Algorithm 2.

At each Bootstrap iteration t, LTS is run on each problem (even those already solved) with a budget of B_t node expansions with the context-model policy with current parameters β^t. After collecting the set of solutions $\mathcal{N}_t$, the parameters β^{t+1} are obtained from Eq. (8) to some approximation (see Appendix C), and the next bootstrap iteration $t + 1$ is started with budget B_{t+1}.

Adjusting the budget is non trivial. Keep in mind that computation time during search is proportional to the number of expansions. Solving previously solved problems usually is fast, because the policy has been optimized for them. Each problem for which a solution is newly found usually takes a large fraction of the budget (since they couldn't be solved for the previous budget), and every problem that remains unsolved consumes the whole budget. While a larger budget means that more problems can be solved, for a fixed set of parameters it is usual to see this number grow only logarithmically with the budget (since we are tackling hard problems). Hence when only few problems have already been solved, a large budget will make the algorithm spend a lot of time in yet-unsolvable problems, wasting computation time. By contrast, a too small budget will prevent finding new solutions and improving the policy, requiring more Bootstrap iterations.

Jabbari Arfaee *et al.* [2011] double the budget at each new Bootstrap iteration. This can become wasteful in computation time if learning works well, but a larger budget does not help much, in which case it may be better to use a constant budget. It may also be not fast enough during the last iterations: suppose 95% of the problems are solved, but the remaining 5% ones require to double the budget k more times: then the 95% will be resolved k more times (possibly finding different solutions), and, if the found solutions change, optimization is also performed k times before any new problem can be solved. By contrast, Orseau and Lelis [2021] use a fixed budget and double the budget only if no new problem is solved, which can also be wasteful in computation time if learning does not manage to work well enough and just one more problem is solved at each step.

To alleviate these issues, first, if more than a factor $(1 + b)$ (say $b = 1/4$) of problems are solved at iteration t compared to the previous iteration — formally, $|\mathcal{N}_t| \geq (1 + b) \left| \bigcup_{t' < t} \mathcal{N}_{t'} \right|$ — we reduce the next budget in case the current budget is too high:

$$B_{t+1} = \max\{B_1, B_t/2\} \,.$$

Otherwise, we increase B_{t+1} so as to approximately double the number of total expansions (in the worst case of no new problem solved), rather than merely doubling the budget. More precisely, at Bootstrap iteration t, say the total number of expansions of *solved* problems is T_t^+, and the number of remaining unsolved problems is $s_t^- = |\mathcal{N}^* \setminus \bigcup_{t' \leq t} \mathcal{N}_{t'}|$, then we set the next budget

$$B_{t+1} = 2B_t + T_t^+/s_t^- \,,$$

which ensures that $T_t^+ + s_t^- B_{t+1} = 2(T_t^+ + s_t^- B_t)$, where $T_t^+ + s_t^- B_t$ is the actual number of expansions used during iteration t, and $T_t^+ + s_t^- B_{t+1}$ is the probable number of expansions in case no new problem is solved, and assuming that previous problems take about the same time to be solved again (which is likely to be an overestimate due to learning).

B Formal Statement of the Lower Bound

In this section we provide a formal version of the informal lower bound of Theorem 2 on the number of node expansions required before reaching a target node n^*. This number is within a factor $(A-1)\bar{d}$ of the upper bound, showing that the upper bound is quite tight and can be meaningfully used as a loss function.

The lower bound in Theorem 11 below requires the following lemma, which shows that the probability mass at the root behaves like a liter of water that is distributed recursively (but unevenly) along all the branches, and that if we collect the water at all the leaves (assuming a finite tree) then it still amounts to one liter, as long as the policy is proper. This lemma can be found in a compact form in the proof of Theorem 3 [Orseau *et al.*, 2018].

A tree $\mathcal{N}' \subseteq \mathcal{N}$ is said to be *full* if every node of the tree either has all its children in the tree, or none of them.

Lemma 10. *Let $\mathcal{N}' \subseteq \mathcal{N}$ be a finite tree with root n_0, and let $\mathcal{L}' \subseteq \mathcal{N}'$ be its leaves. Let π be a policy with $\pi(n_0) = 1$. Then $\sum_{n \in \mathcal{L}'} \pi(n) \leq 1$. Furthermore, if the policy is proper and $\mathcal{N}'$ is a full tree, then $\sum_{n \in \mathcal{L}'} \pi(n) = 1$.*

Proof. We start with the equality case. Using the fact that the policy is proper on the second line, and the fact that the tree $\mathcal{N}'$ is full on the fourth line, we have

$$\sum_{n \in \mathcal{L}'} \pi(n) = \sum_{n \in \mathcal{N}'} \pi(n) - \sum_{n \in \mathcal{N}' \setminus \mathcal{L}'} \pi(n)$$

$$= \sum_{n \in \mathcal{N}'} \pi(n) - \sum_{n \in \mathcal{N}' \setminus \mathcal{L}'} \pi(n) \sum_{n' \in \mathcal{C}(n)} \pi(n' \mid n)$$

$$= \sum_{n \in \mathcal{N}'} \pi(n) - \sum_{n \in \mathcal{N}' \setminus \mathcal{L}'} \sum_{n' \in \mathcal{C}(n)} \pi(n')$$

$$= \sum_{n \in \mathcal{N}'} \pi(n) - \sum_{n \in \mathcal{N}' \setminus \{n_0\}} \pi(n')$$

$$= \pi(n_0) = 1 \, .$$

If the tree $\mathcal{N}'$ is not full, it suffices to assign probability 0 to children outside of $\mathcal{N}'$, which reduces to an improper policy. If the policy is not proper, it can be made proper on $\mathcal{N}'$ by renormalization of π to $\tilde{\pi}$. More precisely, if the tree $\mathcal{N}'$ is not full or the policy is not proper, define, for all $n \in \mathcal{N}' \setminus \mathcal{L}'$, for all $n' \in \mathcal{C}(n) \cap \mathcal{N}' : \tilde{\pi}(n' \mid n) = \pi(n' \mid n)/\sum_{n'' \in \mathcal{C}(n) \cap \mathcal{N}'} \pi(n'' \mid n)$, which ensures that $\tilde{\pi}(n) \geq \pi(n)$ for all nodes $n \in \mathcal{N}'$, and thus $\sum_{n \in \mathcal{L}'} \pi(n) \leq \sum_{n \in \mathcal{L}'} \tilde{\pi}(n) = 1$. $\square$

For a node n^*, define $\overline{\mathcal{N}}(n^*) = \{n \in \mathcal{N} : \text{root}(n) = \text{root}(n^*) \wedge \frac{d}{\pi}(n) \leq \frac{d}{\pi}(n^*)\}$, which is the set of nodes of the same tree of cost at most that of n^*, and $\mathcal{L}'(n^*) = \{n \in \mathcal{N} : \frac{d}{\pi}(n) > \frac{d}{\pi}(n^*) \wedge \text{par}(n) \in \overline{\mathcal{N}}(n^*)\}$, which is the set of children right outside $\overline{\mathcal{N}}(n^*)$ — $\mathcal{L}'(n^*)$ would be the 'frontier' or the contents of the priority queue in Algorithm 1, disregarding tie breaking. Let $A \geq 2$ be the maximal branching factor of the search tree $\overline{\mathcal{N}}(n^*)$, that is, for all $n \in \mathcal{N} : |\mathcal{C}(n)| \leq A$. Observe that $\overline{\mathcal{N}}(n^*)$ may not be a full tree, but that $\overline{\mathcal{N}}(n^*) \cup \mathcal{L}'(n^*)$ is a full tree.

Theorem 11 (Lower bound). *Let π be a proper policy. Then, for a node n^*, the number of nodes with cost at most that of n^* is at least*

$$|\overline{\mathcal{N}}(n^*)| \geq \frac{1}{(A-1)} \left(\frac{1}{\bar{d}} \frac{d}{\pi}(n^*) - 1 \right) \, .$$

where $\bar{d} = 1/\sum_{n \in \mathcal{L}'(n^*)} \frac{\pi(n)}{d(n)}$ is the harmonic average of the depth at the leaves $\mathcal{L}'(n^*)$.

Also observe that by the harmonic-mean – arithmetic mean inequality, $\bar{d} \leq \sum_{n \in \mathcal{L}'(n^*)} \pi(n) d(n)$, the average depth at the leaves of the search tree $\overline{\mathcal{N}}(n^*)$.

Proof. First, using the fact that $|\mathcal{L}'(n^*) \cup \overline{\mathcal{N}}(n^*)|$ is a full tree,

$$|\mathcal{L}'(n^*)| + |\overline{\mathcal{N}}(n^*)| = |\mathcal{L}'(n^*) \cup \overline{\mathcal{N}}(n^*)|$$

$$= 1 + \sum_{n \in \overline{\mathcal{N}}(n^*)} \sum_{n' \in \mathcal{C}(n)} 1 \leq 1 + A|\overline{\mathcal{N}}(n^*)| \, ,$$

and by rearranging we obtain

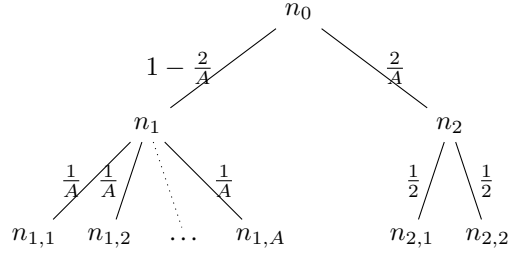$$|\overline{\mathcal{N}}(n^*)| \geq (|\mathcal{L}'(n^*)| - 1)/(A - 1) \, .$$

Figure 3: A tree showing the necessity of the factor $A - 1$ for the lower bound, with $A \geq 3$. Nodes that have fewer than A children can be completed with children of probability 0.

Now,

$$|\mathcal{L}'(n^*)| = \sum_{n \in \mathcal{L}'(n^*)} \frac{d}{\pi}(n) \frac{\pi(n)}{d(n)}$$

$$> \frac{d}{\pi}(n^*) \sum_{n \in \mathcal{L}'(n^*)} \frac{\pi(n)}{d(n)} = \frac{1}{\bar{d}} \frac{d}{\pi}(n^*) \, ,$$

Since $\mathcal{L}'(n^*)$ are the leaves of a full tree, $\sum_{n \in \mathcal{L}'(n^*)} \pi(n) = 1$ by Lemma 10 and thus $\bar{d}$ is indeed an harmonic mean of the depths of the leaves. Therefore,

$$|\overline{\mathcal{N}}(n^*)| \geq \frac{1}{(A-1)} \left(\frac{1}{\bar{d}} \frac{d}{\pi}(n^*) - 1 \right) \, . \qquad \square$$

Remark 12. *It appears that the factor $1/(A - 1)$ is necessary. Consider the tree in Fig. 3, and take $A \geq 3$. Then $\frac{d}{\pi}(n_{2,1}) = 2A$ while $\frac{d}{\pi}(n_{1,\cdot}) = 2A^2/(A - 2) > \frac{d}{\pi}(n_{2,1})$. Then $|\overline{\mathcal{N}}(n_{2,1})| = 5 = 5\frac{d}{\pi}(n_{2,1})/(2A) = O(\frac{d}{\pi}(n_{2,1})/(A - 1))$. Also note that replacing $2/A$ with $1/A$ in the tree leads to $\frac{d}{\pi}(n_{2,1}) = 4A$ and $\frac{d}{\pi}(n_{1,\cdot}) = 2A^2/(A - 1) < \frac{d}{\pi}(n_{2,1})$, which means that $|\overline{\mathcal{N}}(n_{2,1})| \geq 5 + A = \Omega(\frac{d}{\pi}(n_{2,1}))$ instead.*

C   Convex optimization algorithm

To minimize Eq. (6), many convex optimization algorithms can be considered. For a first simple implementation, we would recommend using Frank-Wolfe [Frank and Wolfe, 1956; Jaggi, 2013] while being mindful of numerical stability (see Appendix D).

In the following, we describe the convex optimization routine we use for the experiments, however we suspect better and possibly more principled algorithms might be applicable too.

For the optimizer, we use isoGD [Orseau and Hutter, 2021] projection onto $\mathcal{B}$ (see also SOLO-FTRL[Orabona and Pál, 2018]), a scale-free variant of Adagrad [Duchi *et al.*, 2011], which is adapted to use a line search. We have observed empirically that these algorithms tend to close the duality gap [Jaggi, 2013] faster than other algorithms (Frank-Wolfe [Frank and Wolfe, 1956], normalized gradient descent [Cortés, 2006], accelerated gradient descent [Nesterov, 1983]), but admittedly we did not try all variants of all algorithms). However, the well-known difficulty with Adagrad-style algorithms is that the learning rate is always smaller than

$1/G$ where G is the amplitude of largest observed gradient. But in function optimization, almost always the largest gradients are observed early and decrease significantly near the optimum — in our case, the initial gradients can be exponentially large. Hence, to reduce this dependency, we reset the learning rates on steps that are powers of 2 — this is known as the doubling trick [Cesa-Bianchi and Lugosi, 2006] and we fully expect that a regret bound can be proven with resets too, paying only a constant factor in the regret bound for a significantly milder dependency on the usually-larger initial gradients. We use one learning rate per context.

Optimization is stopped after 200 iterations: if this happens to not be enough to improve the policy significantly to solve more problems, 200 more iterations will be triggered anyway after the next Bootstrap iteration.

Optimization is also stopped early if the duality gap [Jaggi, 2013] guarantees that the loss is within a factor 2 of the optimum. Recall that this roughly means that the bound on the search time is a factor 2 away from the bound on the search time for the *optimal* parameters, which can hardly be much better. The duality gap is calculated every 20 iterations to amortize the computation cost. See also Appendix F.

For the line search, we use some ideas from Truong and Nguyen [2021]: A line search is triggered at each update iteration t where $1 \leq t \mod 20 \leq 3$, and the learning rate found by the line search is re-used for the next optimization steps as long as there is improvement — otherwise a line search is triggered too — and also as the first middle query of the line search in [0, 1]. We use a (quasi-)exact line search rather than a backtracking line search, as it can make a significant difference on the first iterations, but less so afterwards.

See also Appendix D on numerical stability.

D   Numerical Stability Considerations

One should use a numerically stable 'softmax' function for product mixing, and a stable 'log-sum-exp' (LSE) to calculate the logarithm of the LTS loss — the LTS loss can be exponentially large for untuned parameters.

$$\text{LSE}(X) = C + \log \sum_{x \in X} \exp(x - C), \quad C = \max X.$$

For a set $\mathcal{N}'$ of nodes, We can rewrite Eqs. (6) and (7), and the scaled gradient as:

$$\log \ell(n, \beta) = \log d(n) - \sum_{j=0}^{d(n)-1} \log p_\times(n_{[j]}, a(n_{[j+1]}), \beta),$$

$$\log L(\mathcal{N}', \beta) = \text{LSE}(\{\log \ell(n, \beta) \mid n \in \mathcal{N}'\})$$

$$\log(L(\mathcal{N}', \beta) + R(\beta)) = \text{LSE}(\log L(\mathcal{N}', \beta), \log R(\beta)).$$

Recall that $\varepsilon_{\text{mix}} = 0$ during optimization. During the line search, one can use $\log(L(\mathcal{N}', \beta) + R(\beta))$ but note that, while still unimodal (quasiconvex), it may not be convex anymore (despite Theorem 14) with a quadratic regularizer.

To calculate the gradients, similar caution should be used,

for example for some constant C:

$$\beta^{t+1} = \underset{\beta}{\arg\min}\, e^{-C} R(\beta) + \sum_{\tau \in \mathcal{T}} \exp(\log \ell(\tau, \beta) - C),$$

$$\nabla \exp(\log \ell(\tau, \beta) - C) = \exp(\log \ell(\tau, \beta) - C)\nabla \log \ell(\tau, \beta).$$

E   Convexity
E.1   Log loss to LTS loss
Proof of Theorem 9. We can write

$$L(x) = \sum_k \exp\left(\sum_t -\log f_{k,t}(x)\right),$$

By assumption, $-\log f_{k,t}(x)$ is convex for all k, t, and convexity is preserved by both summation and exponentiation (convex and non-decreasing) [Boyd and Vandenberghe, 2004], hence $L(x)$ is convex. □

Remark 13. *Convexity in LTS loss does not imply convexity in log loss. For example, take $f(x) = 1/x^2$ for $x > 0$, then $1/f$ is (strongly) convex, but $-\log 1/x^2 = 2\log|x|$ is not only concave on $(-\infty, 0)$ and on $(0, \infty)$ but also has a singularity at 0. In a sense, the LTS loss is 'nicer' for convex optimization than the log loss.*

Theorem 14. *The logarithm of loss function $L(\cdot)$ defined in Theorem 9 is convex.*

Proof. Following the proof of Theorem 9 we can write

$$\log L(x) = \log \sum_k \exp\left(\sum_t -\log f_{k,t}(x)\right),$$

and the result follows by observing that log-sum-exp and summation preserve convexity [Boyd and Vandenberghe, 2004]. □

E.2   LTS loss convexity of product mixing of context predictors

Theorem 15. *The function $L(\mathcal{N}_t, \beta)$ defined in Eq. (6) is convex in β.*

Proof. This function is of the form $\sum \exp \sum \log \sum \exp f(\beta)$ where f is linear. The result follows by observing that summation, exponentiation, and log-sum-exp are all preserving convexity, since they all are convex and non-decreasing [Boyd and Vandenberghe, 2004]. □

We now provide a more general result that applies if the context predictors are members of the exponential family, rather than just categorical distributions.

Lemma 16. *Let $\mathcal{A}$ be a finite set of actions and canonical parameters $\beta \in \mathbb{R}^{\mathcal{Q} \times \mathcal{A}}$, and let $\mathcal{Q}$ be a set of predictors. Let*

$$p_c(n, a; \beta) = \exp[\beta \cdot T_c(n, a) - A_c(n, \beta) + B_c(n, a)]$$

and all node $n \in \mathcal{N}$ and all actions $a \in \mathcal{A}$, with $A_c : \mathcal{N} \times \mathcal{B} \to \mathbb{R}$ and $B_c : \mathcal{N} \times \mathcal{A} \to \mathbb{R}$ and $T_c : \mathcal{N} \times \mathcal{A} \to \mathbb{R}^{\mathcal{Q} \times \mathcal{A}}$ for

all predictors $c \in \mathcal{Q}$, be a set of members of the exponential family in canonical form, with a dependency on the current node $n \in \mathcal{N}$. Then the product mixing $p_\times(n, a; \beta)$ (Eq. (2)) of the $\{p_c\}_{c \in \mathcal{Q}}$ is also a member of the exponential family in canonical form:

$$p_\times(n, a; \beta) = \exp[\beta \cdot T(n, a) - A(n, \beta) + B(n, a)],$$

with

$$T(n, a) = \sum_{c \in \mathcal{Q}} T_c(n, a),$$

$$B(n, a) = \sum_{c \in \mathcal{Q}} B_c(n, a),$$

$$A(n, \beta) = \ln \sum_{a' \in \mathcal{A}} \exp\left[\beta \cdot T(n, a') + B(n, a')\right].$$

Proof. The result is a straightforward application of the definition of product mixing:

$p_\times(n, a; \beta)$

$= \dfrac{\prod_{c \in \mathcal{Q}} p_c(n, a; \beta)}{\sum_{a' \in \mathcal{A}} \prod_{c \in \mathcal{Q}} p_c(n, a'; \beta)}$

$= \dfrac{\exp\left[\beta \cdot \sum_{c \in \mathcal{Q}} T_c(n, a) + \sum_{c \in \mathcal{Q}} B_c(n, a)\right]}{\sum_{a' \in \mathcal{A}} \exp\left[\beta \cdot \sum_{c \in \mathcal{Q}} T_c(n, a') + \sum_{c \in \mathcal{Q}} B_c(n, a')\right]}$

$= \exp[\beta \cdot T(n, a) - A(n, \beta) + B(n, a)].$ □

Categorical context models can be expressed as members of the exponential family in canonical form, by setting $T_c(n, a)$ to a zero vector, with just a single 1 at index (c, a) for context c and action a, but only if the context c is active at node n, *i.e.*, $c \in \mathcal{Q}(n)$, that is

$$T_c(n, a)_{c', a'} = [\![c' = c]\!] \cdot [\![a' = a]\!] \cdot [\![c \in \mathcal{Q}(n)]\!],$$

where $[\![test]\!] = 1$ if $test$ is true, 0 otherwise. This implies that the vector $T(n, a)$ also has a 1 at index (c, a) for each active context c, for each action $a \in \mathcal{A}$. To select only the valid actions at node n, also set $B_c(n, a) = 0$ if $a \in \mathcal{A}(n)$ and $B_c(n, a) = -\infty$ otherwise. Then

$$p_c(n, a; \beta) = \exp(\beta \cdot T_c(n, a) - A_c(n, \beta) + B_c(n, a))$$
$$= \exp(\beta_{c,a} [\![c \in \mathcal{Q}(n)]\!] - A_c(n, \beta)) [\![a \in \mathcal{A}(n)]\!],$$

$$A_c(n, \beta) = \ln \sum_{a \in \mathcal{A}(n)} \exp(\beta_{c,a} [\![c \in \mathcal{Q}(n)]\!]),$$

that is,

$$p_c(n, a; \beta) = \begin{cases} \dfrac{\exp \beta_{c,a}}{\sum_{a' \in \mathcal{A}(n)} \exp \beta_{c,a'}} & \text{if } c \in \mathcal{Q}(n), a \in \mathcal{A}(n), \\ 0 & \text{if } a \notin \mathcal{A}(n), \\ \dfrac{1}{|\mathcal{A}(n)|} & \text{otherwise.} \end{cases}$$

Recall that uniform distributions have no effects in the product mixing, and thus a context that is not active is in effect removed from the product mixing, as in Eq. (4).

It is interesting to note that the $A_c(n, \beta)$ do not appear directly in the resulting form of the product mixing and thus do not need to be calculated.

Beyond simple context models, since the vector $T_c(n, a)$ can depend on the current node n, it can make use in particular of *features* of the corresponding state of the environment, such as a heuristic distance to the goal.

Finally, members of the exponential family in canonical form are well-known to be log-concave in their natural parameters β, that is, their log loss is convex in their natural parameters: $-\log p_c(\cdot, \cdot; \beta)$ is of the form $h(\beta) + \ln \sum \exp g(\beta)$ where h and g are linear, and since log-sum-exp is convex the result follows. Therefore, by Theorem 9, the LTS loss of the product mixing of members of the canonical exponential family is convex in their natural parameters.

F Beta-Simplex

This section describes a small but computationally helpful improvement regarding the calculation of the duality gap [Jaggi, 2013], which is used to terminate the optimization procedure.

The domain of the parameters β is defined in the main paper as $\mathcal{B} = [\ln \varepsilon_{\text{low}}, 0]^{|\mathcal{Q}| \times A}$, and wrote that $p_c(a; \beta) \geq \varepsilon_{\text{low}}/|\mathcal{C}(n)|$.

While the duality gap can be calculated on this set, it can also be calculated for a subset of $\mathcal{B}$, which more closely relates to the probability distributions of the predictors. Furthermore, the regret can still be meaningfully compared to the best probability distributions for the context predictors, rather than the optimal parameters $\beta_t^* \in \mathcal{B}$.

The highest-entropy probability distribution is the uniform distribution and can be expressed with p_c by setting all components $\beta_{c, \cdot}$ to the same value.

The lowest-entropy probability distribution that can be expressed with p_c is such that $\beta_{c,a} = 0$ for some chosen $a \in \mathcal{A}$ and $\beta_{c,a'} = \ln \varepsilon_{\text{low}}$ for $a' \neq a$, giving

$$p_c(a; \beta) = 1/(1 + (A-1)\varepsilon_{\text{low}}) \geq 1 - (A-1)\varepsilon_{\text{low}},$$
$$p_c(a'; \beta) = \varepsilon_{\text{low}}/(1 + (A-1)\varepsilon_{\text{low}}) \leq \varepsilon_{\text{low}}.$$

Consider the constrained simplex $\Delta_{\varepsilon_{\text{low}}}$ such that if $p \in \Delta_{\varepsilon_{\text{low}}}$, then for all $a \in \mathcal{A} : p(a) \geq \varepsilon_{\text{low}}$. Hence the probability distributions expressed by $\mathcal{B}$ can express at least as much as $\Delta_{\varepsilon_{\text{low}}}$ by convex combinations. Unfortunately, enforcing $\sum_a \exp \beta_{\cdot, a} = 1$ on $\mathcal{B}$ does not lead to a convex set. Instead, we define the β-simplex as a (convex) subset of $\mathcal{B}$ by constraining $\sum_{a \in \mathcal{A}} \beta_{\cdot, a} = (A-1) \ln \varepsilon_{\text{low}}$. Note that the β-simplex still contains the highest and lowest entropy distributions of $\Delta_{\varepsilon_{\text{low}}}$. The 'center' of the β-simplex is at $\beta_{\cdot, a} = ((A-1) \ln \varepsilon_{\text{low}})/A$, which we define as β_0, and explains why we use the regularization $\|\beta - \beta_0\|^2$.

Hence, instead of calculating the regret compared to $\beta^* \in \mathcal{B}$, we can also consider calculating the regret to the best distribution in $\Delta_{\varepsilon_{\text{low}}}$ or the best point in the β-simplex.

More importantly, we calculate the duality gap [Jaggi, 2013] for the β-simplex, which experimentally is easier to reduce than the duality gap for the β-hypercube $\mathcal{B}$.

G Mutex Sets: Relative Tilings

We now give a general and formal definition of the rectangular tiling example in Figure Figure 2. It is closely related

to tile coding [Sutton and Barto, 1998]. On a grid of dimension $R \times C$, relative to some position (r_0, c_0) on the grid, we call a *relative tile* $T(s_r, s_c, d_r, d_c)$ a particular mutex set with row span s_r, column span s_c, row offset d_r and column offset d_c. The ordered values of the grid rectangle between $(r_0 + d_r, c_0 + d_c)$ and $(r_0 + d_r + s_r - 1, c_0 + d_c + s_c - 1)$ identify a unique context within the relative tile. A padding value can be chosen arbitrarily for coordinates that are outside the grid.

A *relative tiling* $R_T(s_r, s_c, D_r, D_c)$ of row span s_r, column span s_c, row distance D_r and column distance D_c, relative to some position (r_0, c_0) is a set of relative tile mutex sets $\{T(s_r, s_c, d_r, d_c)\}_{d_r, d_c}$ for $d_r \in [-D_r, \ldots, D_r - s_r + 1]$ and $d_c \in [-D_c, \ldots, D_c - s_c + 1]$ — this ensures that the last position of the last relative tile is at $(r_0 + D_r, c_0 + D_c)$, while the first position of the first relative tile is at $(r_0 - D_r, c_0 - D_c)$. There are $(2D_r + 2 - s_r)(2D_c + 2 - s_c)$ mutex sets in the relative tiling.

In particular the mutex sets used in the experiments for Sokoban, The Witness, and the Sliding-Tile Puzzle are as follows: The position (r_0, c_0) is taken to be the position of the agent: the avatar in Sokoban, the blank tile in the sliding tile puzzle, and the tip of the 'snake' in The Witness.

We used the following relative tilings. For Sokoban: $R_T(3, 3, 4, 4)$, $R_T(2, 4, 2, 3)$, $R_T(4, 2, 3, 2)$ $R_T(2, 2, 2, 2)$, $R_T(1, 2, 1, 1)$, $R_T(2, 1, 1, 1)$, the number of mutex sets is $|\mathcal{M}| = 125$, and walls are used as padding value; For The Witness: $R_T(3, 3, 4, 4)$, $R_T(2, 2, 4, 4)$, $R_T(2, 1, 1, 1)$, $R_T(1, 2, 1, 1)$, $|\mathcal{M}| = 125$, with one additional padding color, and the goal location is not encoded. For the STP: $R_T(2, 2, 3, 3)$, $R_T(2, 1, 2, 2)$, $R_T(1, 2, 2, 2)$, $R_T(1, 1, 2, 2)$, $|\mathcal{M}| = 102$, with an additional padding value.

For The Witness, our implementation uses two grids: one for the (fixed) colors and one for the snake (the trajectory of the player). We perform the same relativing tiling on each grid in parallel, merging each pair of contexts into a single context.

H Extended Table of Results

While the message of the main paper is to compare LTS+CM with LTS+NN, it is also interesting to compare LTS+CM with other algorithms for the same domains. See Table 2.

In particular, Orseau and Lelis [2021] introduce the PHS* algorithm, which is based on LTS, but also uses a heuristic function to speed up the search, and they also compare with Weighted A* (WA*) [Pohl, 1970; Ebendt and Drechsler, 2009] which uses only a heuristic function, both with similar neural networks as LTS+NN. We can see LTS+CM is competitive on all three domains, while being fast (recall that tests use only one CPU, and no GPU). Moreover, LTS+CM could also be extended with a value function, either to PHS*+CM, or by using the value function as input features to each context, or by binarizing the values into multiple (possibly overlapping) contexts.

For the STP, while the test set used for DeepCubeA is different from the one used by the other algorithms,[5] we

still expect the results to be comparable since Orseau and Lelis [2021]'s test sets are composed of random instances rather than scrambled from the solution. The heavy cost in expansions paid by DeepCubeA shows the price of finding near-optimal solutions — which it is reported to find 97% of the time. WA* appears to give the best compromise on this problem. LTS+CM expands more nodes but, given that it is still fast in milliseconds, we can hope for better results possibly by adding more mutex sets.

DeepCubeA has also been trained on the 900k unfiltered Boxoban set [Guez *et al.*, 2018], and finds short solutions in few expansions. But because it is trained differently from the other algorithms, its results are not directly comparable. We trained LTS+CM on the 450k medium Boxoban set and obtained a slightly better average number of expansions, with a much faster algorithm in milliseconds — at the expense of solution length. The same LTS+CM also expands almost 4x fewer nodes on the Boxoban hard set than the one trained with only 50k problems.

On the Rubik's cube, we report results for LTS+CM at various stages of its training. After just 300k cubes (2 hours 10 minutes), scrambled at most 15 times (these are much easier than fully scrambled cubes) it already finds a policy that solves the whole random test set — scrambled 100 times, which is usually considered more than sufficient for generating random cubes [Korf, 1997]. At 400k cubes, the policy is already substantially faster than previous work. We trained the network for up to 9M cubes, but the average number of expansions stabilizes at around 380. The training curve is shown in Fig. 4.

We also compared LTS+CM on the same 100 hardest Rubik's Cube problems of Büchner *et al.* [2022] as used by [Allen *et al.*, 2021]. This set appears slightly simpler than the test set we used because we are able to find shorter solutions with fewer expansions in this set than in our set of problems. Note also that Allen *et al.* [2021] do not account for the length of the macro actions in the number of expansions, because they use a logical representation of the problem to 'compress' the results of macro actions — this assumes access to more information about the environment than just a simulator. The two approaches, focused macro-actions and learning a policy, are very much composable, and it would be interesting to see whether such macro actions could help make LTS+CM more efficient in training time or converge to a faster policy. As far as we are aware, LTS+CM is the first machine-learning algorithm to learn a fast policy for the Rubik's cube — while using only little domain-specific knowledge.

Finally, it must be noted that the timings reported in the table should be read with a grain of salt, as different algorithms have been tested on different machines and implemented using different libraries. LTS+CM has been implemented in Racket [6] and we report running times as a means of showing that the CM models are very fast in practice.

---

[5] The training set is also different, and although the number of problems is not clearly specified, it appears to be much larger than

50k problems.

[6] https://racket-lang.org/
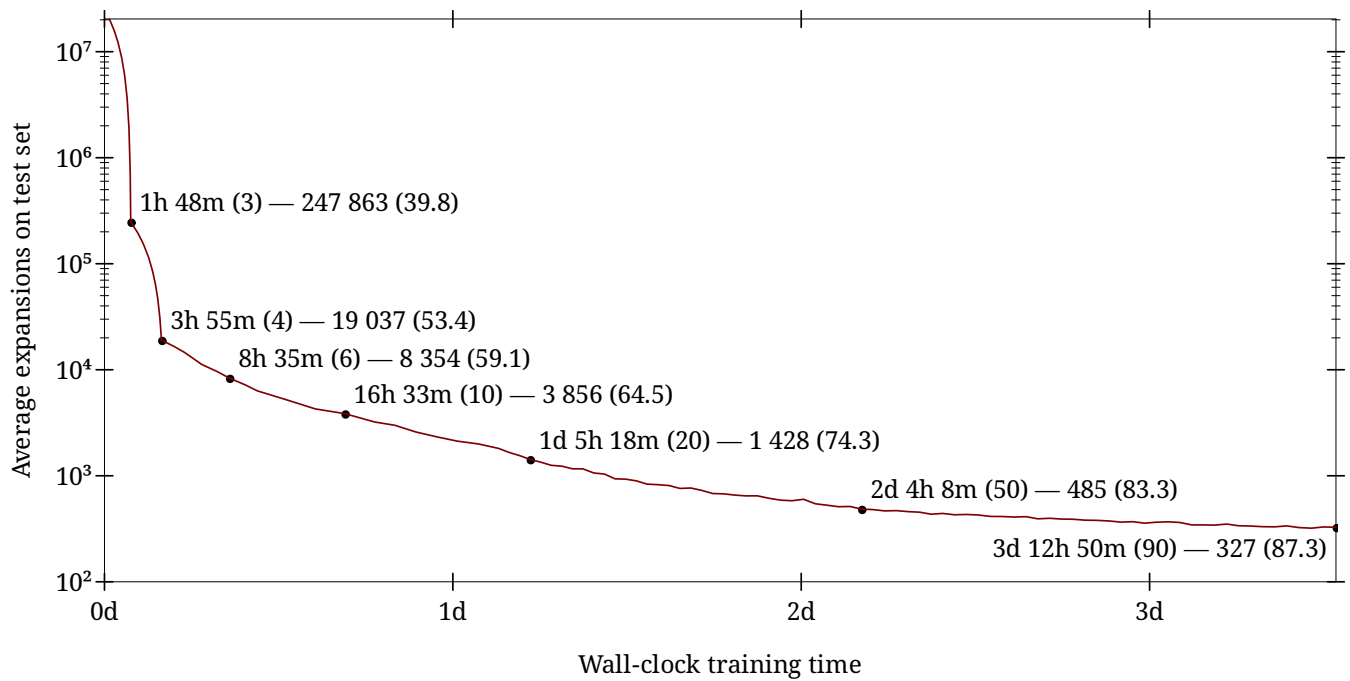
Figure 4: Rubik's cube: Average number of expansions on the test set as a function of training time. Each point label should be read: "training time (iteration) — expansions (length)", where "training time" is the wall-clock time used for both solving and optimizing since the start, "iteration" is the training set iteration (each set contains 100k cubes), "expansions" is the average number of expansions on the test set, and "length" is the average length of the solutions found on the test set.

Domain	Algorithm	%solved	Length	Expansions	Time (ms)
Boxoban	LTS+CM (this work)	100.00	41.7	2 132.3	124
	LTS+NN [Orseau and Lelis, 2021]	100.00	40.1	2 640.4	19 500
	PHS* [Orseau and Lelis, 2021]	100.00	37.6	**1 522.1**	11 300
	WA*, w=1.5 [Orseau and Lelis, 2021]	100.00	**34.5**	3 729.1	25 500
	LTS+CM (this work) @500k	100.00	48.5	**858.1**	55
	DeepCubeA [Agostinelli *et al.*, 2019]	100.00	**32.9**	1 050.0	2 350
The Witness	LTS+CM (this work)	**100.00**	15.5	**102.8**	9
	LTS+NN [Orseau and Lelis, 2021]	**100.00**	**14.8**	520.2	3 200
	PHS* [Orseau and Lelis, 2021]	**100.00**	15.0	408.1	3 000
	WA*, w=1.5 [Orseau and Lelis, 2021]	99.90	*14.6*	*18 345.2*	*71 500*
STP (24-puzzle)	LTS+CM (this work)	**100.00**	211.2	5 667.4	236
	LTS+NN [Orseau and Lelis, 2021]	0.90	*145.1*	*39 005.6*	*31 100*
	PHS* [Orseau and Lelis, 2021]	**100.00**	224.0	2 867.2	2 800
	WA*, w=1.5 [Orseau and Lelis, 2021]	**100.00**	129.8	**1 989.8**	1 600
	DeepCubeA [Agostinelli *et al.*, 2019]	**100.00**	**89.5**	6 440 000.0	19 330
Boxoban hard	LTS+CM (this work)	**100.00**	**67.8**	48 058.6	3 275
	LTS+CM (this work) @500k	**100.00**	72.5	**12 166.2**	761
	LTS+NN [Guez *et al.*, 2019]	94.00	n/a	n/a	*3 600*
	ExPoSe [Mittal *et al.*, 2022]	97.30	n/a	n/a	n/a
Rubik's cube	LTS+CM (this work) @300k	100.00	39.8	247 862.4	18 949
	LTS+CM (this work) @400k	100.00	53.3	20 647.9	950
	LTS+CM (this work) @5M	100.00	81.7	**498.0**	16
	DeepCubeA [Agostinelli *et al.*, 2019]	100.00	**21.5**	∼600 000.0	24 220
	LTS+CM (this work) @5M	100.00	**78.6**	**431.7**	16
	GBFS(A+M) [Allen *et al.*, 2021]	100.00	378.0	†171 300.0	n/a

Table 2: More test results. See Table 1 and the text in Appendix H for more information. The line splits in Boxoban and STP are because the second group uses different training sets from the rest. The test set used by DeepCubeA for STP is different from that of LTS+{CM,NN}, but we expect the comparison to be meaningful anyway. †Does not account for the cost of macro-actions.

I   Table of Notation

$\mathcal{N}$	Set of all nodes, may contain several root nodes
n	A node in $\mathcal{N}$
$d(n)$	Depth of the node n
$\mathcal{C}(n)$	Children of n
$\text{par}(n)$	Single parent of n, may not exist
$\text{root}(n)$	Topmost ancestor of n, has no parent
$\text{anc}(n)$	Set of ancestors of n
$\text{anc}_+(n)$	$\text{anc}(n) \cup \{n\}$
$\text{desc}(n)$	Descendants of n
$\text{desc}_+(n)$	$\text{desc}(n) \cup \{n\}$
$n_{[j]}$	Node at depth j on the path from $\text{root}(n) = n_{[0]}$ to $n = n_{[d(n)]}$
$\overline{\mathcal{N}}(n)$	Set of nodes of cost $d(\cdot)/\pi(\cdot)$ at most that of n
$\mathcal{N}_t$	Set of nodes after t problems
$\mathcal{N}^*$	Set of solution nodes n^*
$\mathcal{N}^*(n)$	$= \mathcal{N}^* \cap \text{desc}_+(n)$ et of solution nodes below n
$\mathcal{N}^0$	Set of root nodes (problems)
$\mathcal{L}(n)$	Leaves of the tree $\mathcal{N}(n)$
π	Policy
$\pi(n)$	Probability of the node n according to the policy π
$\pi(n' \mid n)$	$\pi(n')/\pi(n)$, assuming $n' \in \mathcal{C}(n)$.
$\frac{d}{\pi}(n)$	$d(n)/\pi(n)$
$\ell(n, \beta)$	Loss function for a single node $= \frac{d}{\pi}(n)$ for parameters β
$L(\mathcal{N}', \beta)$	Cumulative loss over a set of nodes and parameters β
$\mathcal{Q}$	Set of contexts
$\mathcal{Q}(n)$	Set of contexts active at node n
$\mathcal{M}$	Set of mutex sets
M	A mutex set
$p_c(a)$	Probability of the action label a according to a context predictor p_c
$p_\times(n, a)$	Product mixing of context predictors at node n for action (edge label) a
β	Parameters of the context predictors
$\mathcal{B}$	$= [\varepsilon_{\text{low}}, 0]^{\lvert \mathcal{Q} \rvert \times A}$, set of all possible parameter values for β
B_t	Budget used at Bootstrap iteration t
$\mathcal{A}$	Set of actions (edge labels)
$\mathcal{A}(n)$	Set of edge labels at node n, possible actions at n
a	An action, edge label
$a(n)$	Edge label (action) from $\text{par}(n)$ to n
$\mathcal{R}(\mathcal{N}_t)$	Regret of the learner compared to the optimal parameters for a set of solution nodes $\mathcal{N}_t$
$[\![test]\!]$	$=1$ if $test$ is true, 0 otherwise