How Should We Search in Neural Language Spaces? A Study on Test-Time Search

Anonymous authors
Paper under double-blind review

Abstract

Recently, learning neural languages and their interpreters using data has been shown to be effective in generalizing to out-of-distribution tasks. In particular, the structure of neural languages can be leveraged via test-time search in a paradigm like programming by example (PBE). Several existing works use the differentiability of neural languages and their interpreters to search the language space in a gradient-guided fashion. We examine existing frameworks and study their effectiveness when searching the language space in a policy-guided fashion, where the encoder that induces a program in the neural language to solve a task acts as a policy. We show that frameworks enabling policy-guided search in the language space naturally exhibit length generalization via test-time search. Moreover, our results indicate that, depending on the nature of the neural language and its interpreter, using policy-guided search techniques such as Levin Tree Search (LTS) during test time both improves out-of-distribution generalization and requires significantly less computational overhead.

1 Introduction

Program induction offers a compelling route to out-of-distribution generalization because programs can recombine known operations in novel ways and execute with more computation than the programs observed during training. Recent work shows that such program spaces can be learned from data, yielding neural languages that support generalization to novel compositions and program lengths (Macfarlane & Bonnet, 2025; Macfarlane et al., 2026; Baek et al., 2026). This raises a fundamental question: *how should we search these learned program spaces at test time?*

A key component of this paradigm is *test-time search*. A learned language defines a space of candidate programs, and solving a new task means finding a program in this space consistent with the available input-output examples. Because architectures such as Latent Program Network (LPN) and Neural Language Interpreter (NLI) are differentiable, previous work approached this problem through gradient-based optimization (Macfarlane & Bonnet, 2025; Macfarlane et al., 2026). In NLI, for example, the discrete program representation is relaxed through the Gumbel-Softmax distribution (Jang et al., 2017), allowing gradients to propagate through both the program and its learned interpreter. At test time, these gradients can refine an initial program proposed by the encoder.

In this paper, we argue that while differentiability enables learning, it need not determine how the learned language should be searched at test time to express new programs. Once a model has learned a discrete language, searching through a continuous relaxation of that language potentially ignores some of its most useful structure. Gradient optimization takes many small steps in a continuous parameter space before those changes translate into different discrete programs. In contrast, each step of a discrete search procedure can directly replace one candidate program with another. We show that treating the learned language as what it represents, a space of discrete symbols, substantially improves the efficiency of test-time search. In our experiments, searching directly in program space reduces the computational cost of discovering solutions by about an order of magnitude.

We then consider a second form of structure that the original NLI architecture does not exploit. Instead of producing the program as a complete collection of symbols, we introduce a recurrent variant in which programs are generated autoregressively, with each symbol conditioned on the prefix generated so far. This formulation is closely related to autoregressive and execution-guided

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.

neural program synthesis (Chen et al., 2019). Importantly, the encoder now has a second interpretation: it defines a policy over the next program symbol. Test-time program synthesis can therefore be formulated as a tree-search problem in which nodes correspond to program prefixes and the learned encoder provides a policy for expanding them. This allows us to apply policy-guided search algorithms such as Levin Tree Search (LTS) (Orseau et al., 2018) directly in the learned neural language.

The recurrent formulation brings an additional advantage. Search no longer needs to fill out all program slots predetermined in the architecture before evaluating a candidate. Instead, programs can be constructed incrementally in a tree-like expansion and terminated as soon as a solution is obtained. This favors shorter programs and avoids unnecessary computation. Combined with policy-guided tree search, this yields roughly another order-of-magnitude reduction in test-time computation. Overall, our methods reduce the number of floating-point operations required for test-time search by up to two orders of magnitude relative to the gradient-based search used in previous work.

We evaluate these ideas on sequence-transformation problems used in prior work, where systematic generalization requires executing novel compositions and programs of lengths outside the training distribution, and on inducing programs for cellular automata. Our experiments show that the learned neural language supports substantially more efficient search than previous work. More broadly, our results highlight an important distinction for models that learn structured representations end-to-end: the mechanism used to learn a representation need not be the best mechanism for searching it. Differentiable learning can discover a discrete compositional language, while classical discrete and policy-guided search can provide a far more effective test-time search scheme.

2 Problem Statement

We follow the programming-by-example (PBE) formulation of Lieberman (2001). Each task is defined by an unknown program p drawn from a distribution over a program space $\mathcal{P}$. The model observes a specification $S = \{(x_i, y_i)\}_{i=1}^{n}$ of n input-output pairs generated by p, i.e. $y_i = p(x_i)$, together with a query input x_{n+1}, and must predict $p(x_{n+1})$. Information about p is available only through its behavior on S; the model never observes a representation of p, either during training or at test time. Training tasks are generated by sampling $p \sim P_{\text{train}}$ and inputs $\{x_i\}_{i=1}^{n+1} \sim P(X \mid p)$, while evaluation tasks are drawn from P_{test}, which may differ from P_{train}. We are primarily interested in this out-of-distribution (OOD) regime, where solving a test task requires programs that are longer than, or composed differently from, those seen during training.

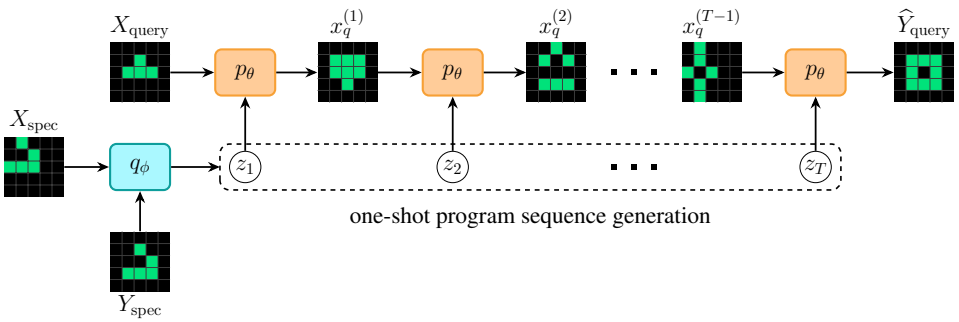
We consider models that solve PBE tasks through a learned *neural language*. An inductor q_ϕ maps a specification to a program $z = (z_1, \ldots, z_t)$ of sequence length $t \leq T$, where each symbol z_j belongs to a learned vocabulary $\mathcal{V}$ of size K. An interpreter p_θ executes z on an input to produce a distribution over outputs, $p_\theta(y \mid x, z)$. We write $\mathcal{Z} = \bigcup_{t=1}^{T} \mathcal{V}^t$ for the set of all programs expressible in the language. Since the size of $\mathcal{Z}$ grows exponentially with the maximum program length T, finding a solution by naively enumerating $\mathcal{Z}$ is infeasible. Both components are trained end-to-end on tasks from P_{train}. In this paper, we focus on how to use a given pair (q_ϕ, p_θ) at test time.

Given a trained language and a test task (S, x_{n+1}), there are two broad ways to produce a prediction. The first is *amortized inference*: a single forward pass of the inductor proposes a program $\hat{z} = q_\phi(S)$, which the interpreter executes on the query to give $\hat{y}_{n+1} \sim p_\theta(\cdot \mid x_{n+1}, \hat{z})$. This is cheap, but its quality is bounded by how well the inductor generalizes. Because q_ϕ is trained only on P_{train}, it can fail to propose a correct program for an OOD task even when one exists in $\mathcal{Z}$.

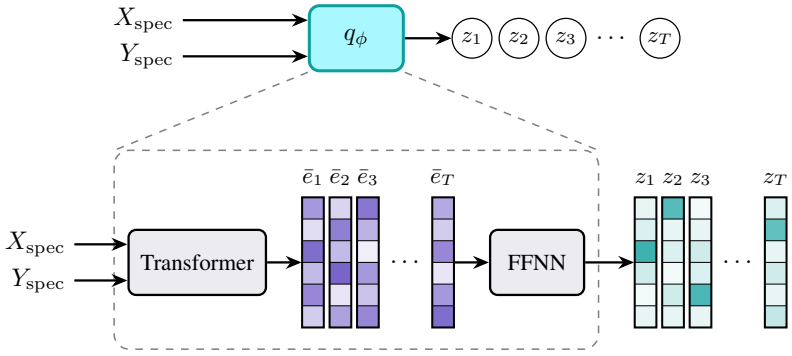
The second is *test-time search*, which spends additional computation to explore $\mathcal{Z}$ to propose a solution. Test-time search has been shown to substantially improve OOD generalization over amortized inference with the same trained model (Macfarlane et al., 2026; Macfarlane & Bonnet, 2025). This suggests that a learned language can often represent OOD solutions that the inductor fails to propose, and that performance is limited by the ability to find them. Our focus is therefore not on how to learn a better language, but on how to make the fullest use of a learned one.

3 Searching in Neural Language Spaces

This section describes the neural architectures we study and the test-time search techniques we apply to them. We briefly describe NLI and its training process. We then introduce a variant of NLI that



(a) NLI rollout example.



(b) Inner workings of the inductor in NLI.

Figure 1: (a) Illustration describing the flow of program sequence generation in NLI, evolving the given input grid for 4 ticks in `GameofLife` cellular automaton. For simplicity, $|S|$ is shown to be 1. (b) Illustration showing how program sequence is generated one-shot in NLI

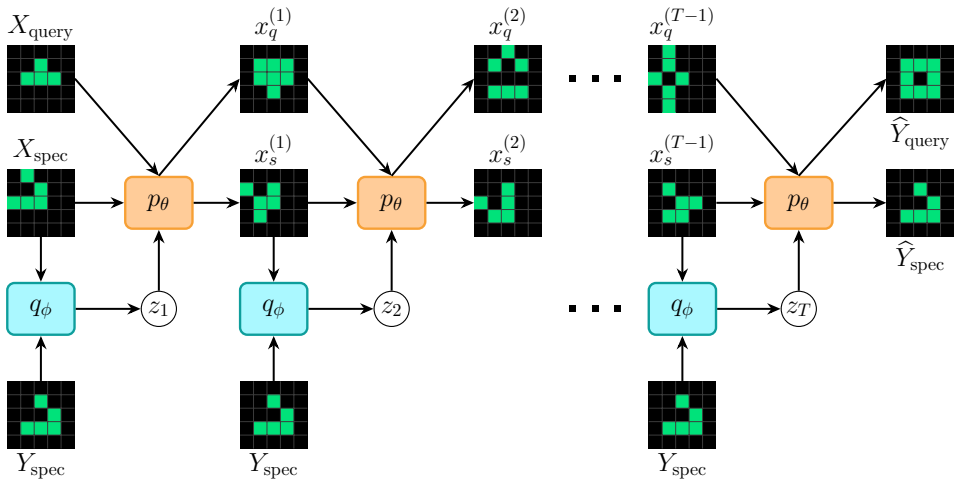
generates program sequences autoregressively and accommodates diverse search techniques during test time. We also discuss various test-time search techniques we study and describe them by their search space, guidance signal, and termination condition.

3.1 Neural Architectures

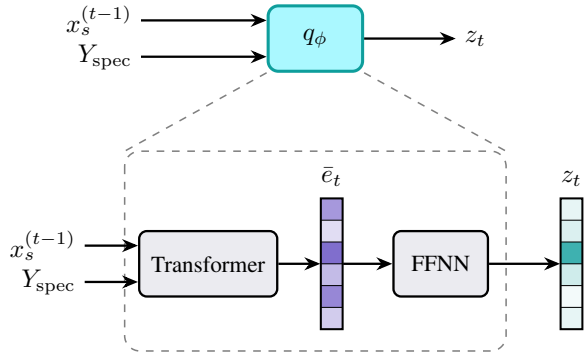
NLI. NLI (Macfarlane et al., 2026) is an architecture that learns an end-to-end differentiable neural language that is discrete and symbolic. As described in Sec. 2, the architecture consists of an *inductor* (denoted by q_ϕ) and an *interpreter* (denoted by p_θ). The inductor generates a program sequence $z_{1:T}$ conditioned on the specification set S, which is then sequentially processed by the interpreter, along with the inputs (either specification or query) to generate the predicted output. The program sequence length T is baked into the architecture of q_ϕ so that one-shot program generation is possible. Fig. 1a shows the entire flow of NLI solving a given task. To train NLI, Macfarlane et al. (2026) use a reconstruction loss combined with *token reuse regularization* that incentivizes token reuse across solutions in a batch of size B, encouraging compositional generalization (Eq. 1). To ensure variable program sequence length, NLI implements a manual *no-op symbol* in p_θ and q_ϕ. NLI also uses the Gumbel-Softmax trick (Jang et al., 2017) during training to make the learned language discrete.

$$\mathcal{L}_{\text{train}}(\phi,\theta) = \underbrace{-\mathbb{E}_S\left[\log p_\theta(Y_{\text{query}} \mid X_{\text{query}}, q_\phi(S))\right]}_{\mathcal{L}_{\text{recon}}(\phi,\theta)} + \lambda \underbrace{\sum_{k=1}^{|\mathcal{V}|}\left(1 - \prod_{b=1}^{B}\prod_{t=1}^{T}(1 - q_\phi(S)_{b,t,k})\right)}_{\mathcal{L}_{\text{reuse}}(\phi)} \quad (1)$$

NLI-R. In this paper, we introduce a variant of NLI called *Recurrent Neural Language Interpreter* (NLI-R). Like NLI, NLI-R is an architecture that learns an end-to-end differentiable neural language



(a) NLI-R rollout example.



(b) Inner workings of the inductor in NLI-R.

Figure 2: (a) Illustration describing the flow of program sequence generation in NLI-R, evolving the given input for 4 ticks in `GameofLife` cellular automaton. At each symbol generation step, q_ϕ is conditioned on the output executed so far. Unlike NLI, NLI-R must follow the specification execution to generate the program sequence. For simplicity, $|S|$ is shown to be 1. (b) Illustration showing how a symbol is generated in NLI-R

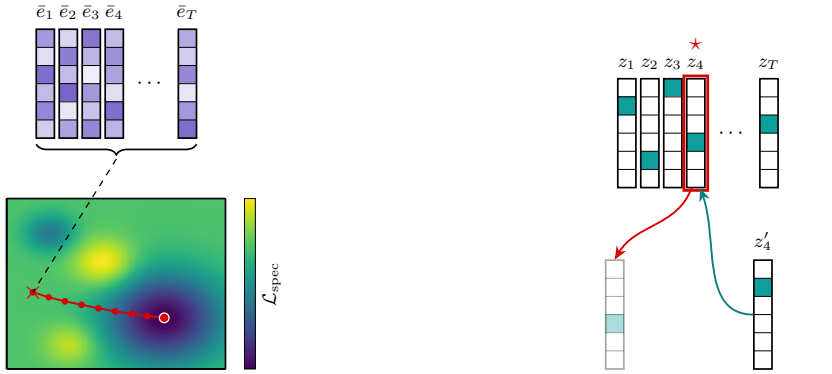
that is discrete and symbolic. The main difference between the two architectures is how they generate the program sequence. Unlike NLI, NLI-R generates the program sequence one symbol at a time, conditioning each symbol on the interpreter's output. Fig. 2a shows the flow of program sequence generation while solving a given task. This makes the symbol generation *autoregressive*. This simple change provides two hypothesized advantages: the program sequence generation is now conditioned on the behavior of program prefixes, ensuring that the inductor chooses symbols with feedback from the interpreter, and disentangling program sequence length from the architecture of the inductor enables unbounded program sequence generation during test time. In this paper, we focus on the latter hypothesis and design experiments accordingly. To stay as close to NLI as possible, this architecture retains every other design aspect from NLI: training objective (Eq. 1), manual no-op symbol usage, Gumbel-Softmax trick (Jang et al., 2017), and the sequential execution of symbols using a learned interpreter.

3.2 Test-Time Search Techniques

We characterize a test-time search technique by three components: a **search space**, a **guidance signal**, and a **termination condition**. Given these, the technique traverses the search space, using the guidance signal to decide which candidates to visit next, until it meets the termination condition. It then returns a program $\hat{z}$, which is executed on x_{n+1} to produce the prediction. Tab. 1 showcases

the search techniques we study and their components. The *search space* is the set of objects the technique moves between. It may be $\mathcal{Z}$ itself, where each candidate is a complete discrete program; the continuous space of embeddings from which the inductor derives the symbols of a program; or the set of program prefixes, where candidates are built one symbol at a time. The *guidance signal* is the information used to propose or prioritize candidates. Because the specification outputs are observed, one such signal is the specification reconstruction loss $\mathcal{L}_{\text{spec}}(z; S) = -\sum_{(x_i,y_i)\in S} \log p_\theta(y_i \mid x_i, z)$, which measures how well a candidate explains S and is differentiable with respect to the embeddings when the search space is continuous. Another is the inductor, whose distribution over symbols can be sampled from or used as a policy over the next symbol. The *termination condition* determines when the search stops, e.g., after a fixed number of steps or as soon as a candidate explains S.

Continuous search. *Gradient Search* (Macfarlane et al., 2026) uses the perturbed embeddings directly as its candidates. It performs gradient descent on $\mathcal{L}_{\text{spec}}$ with step size η, annealing the Gumbel-Softmax temperatures of the inductor and interpreter. It stops after n_{steps} steps and returns the candidate with the lowest $\mathcal{L}_{\text{spec}}$. a variant, *Gradient Search w/ Early Stopping*, which additionally stops the search when the program sequence derived from the embedding vectors (Fig. 1b and Fig. 2b) reproduces every output in S.



(a) Gradient Search (b) Greedy Coordinate Update

Figure 3: Illustration of how Gradient Search and GCU function within their search space. (a) Gradient Search operates on the embedding vectors and is shown guiding them to the minimum of $\mathcal{L}_{\text{spec}}$ (b) GCU operates on program sequences and is shown identifying the symbol with maximum gradient norm and performing a replacement that is minimal in $\mathcal{L}_{\text{spec}}$

Complete discrete search. We introduce *Greedy Coordinate Update* (GCU), which searches directly over discrete programs and uses gradients only to decide where to edit. Each perturbed embedding sequence is converted into a discrete program by choosing the most probable symbol at every position. At each step, GCU computes $\nabla_z \mathcal{L}_{\text{spec}}$, selects the position with the largest gradient norm, and replaces that symbol with whichever symbol in $\mathcal{V}$ most reduces $\mathcal{L}_{\text{spec}}$. The search stops once a program reproduces every output in S or after n_{swaps} swaps. GCU is related to gradient-guided token substitution methods such as HotFlip (Ebrahimi et al., 2018) and Greedy Coordinate Gradient (Zou et al., 2023). Those methods use gradients to rank candidate replacements; GCU uses the gradient to choose the position and evaluates all replacements exactly.

Sample-and-Vote (Baek et al., 2026) draws programs from the inductor at sampling temperature κ and stops once n_{rollouts} programs have been drawn. Only programs that reproduce every output in S receive a vote, and the program with the most votes is returned. For NLI, symbols are sampled independently at each position; for NLI-R, programs are sampled by autoregressive rollouts.

Discrete tree search. The remaining two techniques require a policy over the next symbol, so they apply only to NLI-R. Both use an ϵ-mixture of the inductor and the uniform distribution over $\mathcal{V}$, which follows the uniform policy with probability ϵ and the inductor otherwise, and we write $\pi(n)$ for the probability this policy assigns to the path from the root to node n.

Beam Search advances the tree one level at a time. At each advancement, it retains the b prefixes with the highest cumulative log-probability $\log \pi(n)$. The search stops once a prefix reproduces every output in S or after n_{adv} advancements.

Levin Tree Search (LTS) (Orseau et al., 2018) expands nodes in best-first order of the cost $d(n)/\pi(n)$, where $d(n)$ is the depth of node n. The search stops once a node reproduces every output in S or after n_{exp} expansions. For any solution node $n^\star$, LTS finds a solution within at most $d(n^\star)/\pi(n^\star)$ expansions (Orseau et al., 2018). The mixture ensures $\pi(n) \geq (\epsilon/K)^{d(n)}$, which keeps this bound finite even when the inductor assigns vanishing probability to a necessary symbol.

Table 1: Test-time search techniques and their components (Sec. 3.2). All techniques support NLI and NLI-R except Beam Search and LTS, which require NLI-R.

Technique	**Search space**	**Guidance signal**	**Termination**	**Parameters**
Gradient Search	Continuous embeddings	$\nabla_{\bar{e}}\mathcal{L}_{\text{spec}}$	n_{steps} exhausted	$n_{\text{starts}}, \eta, n_{\text{steps}}$
Gradient Search w/ ES	Continuous embeddings	$\nabla_{\bar{e}}\mathcal{L}_{\text{spec}}$	Spec. match or n_{steps} exhausted	$n_{\text{starts}}, \eta, n_{\text{steps}}$
Sample-and-Vote	Complete programs	Spec. match	n_{rollouts} exhausted	$n_{\text{rollouts}}, \kappa$
GCU	Complete programs	$\nabla_z\mathcal{L}_{\text{spec}}$, $\mathcal{L}_{\text{spec}}$	Spec. match or n_{swaps} exhausted	$n_{\text{starts}}, n_{\text{swaps}}$
Beam Search	Program prefixes	ϵ-mixture policy	Spec. match or n_{adv} exhausted	$b, \epsilon, n_{\text{adv}}$
LTS	Program prefixes	ϵ-mixture policy	Spec. match or n_{exp} exhausted	ϵ, n_{exp}

4 EXPERIMENTS

Our experiments address two questions raised in Sec. 1, both assume a language is given.

RQ1. Does discrete search generalize better, and more efficiently, than continuous search?

RQ2. Given a compute cost, is autoregressive symbol generation better than one-shot generation?

Benchmarks. We evaluate on four benchmarks. Three come from the suite of Macfarlane et al. (2026), in which each task transforms a fixed-length sequence. In *Shift-L*, models are trained on left shifts of $k \in \{1, \ldots, 5\}$ and tested on larger shifts $k \in \{6, \ldots, 10\}$, which requires longer programs. *Shift-P* inverts this split: models are trained on shifts $k \in \{7, 8, 9\}$ and tested on $k \in \{1, 2, 3\}$, which requires extracting shorter primitives from longer programs. In *Comp-I*, models are trained on programs consisting of a single primitive drawn from a library of more than 20 sequence operations, and tested on compositions of exactly two primitives. We also introduce *Cell-Auto*, a benchmark of cellular automata on 5×5 binary grids. Training tasks apply a single tick of one of three rules: Game of Life, Morley, or Day & Night. OOD tasks apply two ticks, composing the training rules, so solving them requires chaining learned single-tick behavior into longer programs. In all benchmarks, in-distribution (ID) test tasks are drawn from P_{train} and OOD test tasks from P_{test}.

Candidates. Each candidate is a pair consisting of a neural architecture from Sec. 3.1 and a test-time search technique from Sec. 3.2. For NLI, we evaluate Sample-and-Vote, Gradient Search, Gradient Search w/ Early Stopping, and GCU. For NLI-R, we evaluate the same four techniques together with Beam Search and LTS, which require a policy over the next symbol. We also report amortized inference for both architectures, in which the inductor is executed without search.

Training. Since our focus is on test time, we follow standard practice for training and model selection and hold the resulting checkpoints fixed. For each benchmark, NLI and NLI-R are trained with the objective in Eq. 1 across a sweep of hyperparameters, including the program length T, with 3 random seeds per configuration. After training, we select the checkpoint with the highest accuracy under amortized inference, without any test-time search, on a validation set whose tasks are drawn from P_{train} but whose input–output pairs are disjoint from those seen during training. Selection therefore favors no search technique. The hyperparameter sweep is listed in App. A. All search techniques are applied to the same selected checkpoint, so differences between candidates of the same architecture are due to search alone.

Evaluation. A test task is solved if the program returned by the search reproduces the query output exactly. Search techniques observe only the specification S; the query output is used only for evaluation. We report accuracy on the ID and OOD test splits, together with the test-time computation each candidate spends. Because individual steps of different techniques differ greatly in cost, we measure this computation in floating-point operations (FLOPs), counting the matrix multiplications performed during search, including those in the backward passes of gradient-based techniques. App. C details the accounting. For the tree searches, intermediate interpreter states are cached, so each expansion is charged for a single inductor forward pass and a single interpreter step per child. Technique-specific parameters are listed in App. B.

Vocabulary filtering. We also study *vocabulary filtering*, which restricts the symbols available during search to a subset $\mathcal{A} \subseteq \mathcal{V}$. The subset $\mathcal{A}$ is selected using how often each symbol appears in programs that solve in-distribution tasks, controlled by a threshold τ. Increasing τ reduces the search space but also increases the risk of removing the solution from the search space, making the choice of τ difficult. We use the filtered vocabulary in all our experiments. App. D describes the filtering procedure and choice of τ.

4.1 OOD Generalization of Search Techniques

Table 2: OOD accuracy of each framework–technique pair. Best result per benchmark in bold.

Framework	Technique	Comp-I	Shift-L	Shift-P	Cell-Auto
NLI	Amortized Inference	0.000	0.002	0.000	0.038
	Sample-and-Vote	0.670	0.753	0.000	0.262
	Greedy Coordinate Update	**1.000**	0.815	**1.000**	0.530
	Gradient Search	0.685	0.351	0.067	0.446
	Gradient Search w/ ES	0.989	0.461	0.104	0.406
NLI-R	Amortized Inference	0.433	0.811	0.239	0.514
	Levin Tree Search	**1.000**	**1.000**	**1.000**	**0.980**
	Beam Search	0.939	**1.000**	0.365	0.892
	Sample-and-Vote	0.955	0.840	0.624	**0.980**
	Greedy Coordinate Update	**1.000**	0.840	0.640	**0.980**
	Gradient Search	0.675	0.840	0.292	0.802
	Gradient Search w/ ES	0.991	0.840	0.333	0.840

Tab. 2 uses a fixed dataset containing OOD tasks in each domain. We evaluate each framework-technique combination on the fixed dataset, sweeping search-specific parameters over 5 seeds. We report the best performance configuration, with performance averaged across 5 seeds. Amortized Inference is deterministic once the model is trained because Gumbel-Softmax is no longer used in generating the z-vectors. The search sweep details are in App. B.

Tab. 2 shows that discrete search outperforms continuous search on OOD tasks across benchmarks and frameworks. LTS achieves near 100% accuracy across benchmarks consistently, showing a stark difference between continuous search and discrete, tree-based search. Additionally, when comparing frameworks with the technique held fixed, NLI-R outperforms NLI 17 out of 20 times in the table. This shows that autoregressive program generation is a better choice than one-shot program generation consistently across benchmarks.

4.2 Cost of Generalization in Search Techniques

When we vary the parameters of the termination condition for a search technique, the amount of computation to produce the solution changes. By varying the termination condition parameters for each search technique (with the remaining parameters fixed at their best values according to Tab. 2), we obtain Fig. 4. The plots show that continuous search techniques are mostly at the far-right corner whereas discrete search techniques are not. When comparing LTS (solid yellow line) with Gradient Search (dashed purple line) across plots, we see a consistent $100\times$ difference in computational cost for equal or better performance from LTS. This clear distinction between continuous searches and discrete searches is an evidence that motivates discrete languages and searching in discrete spaces.

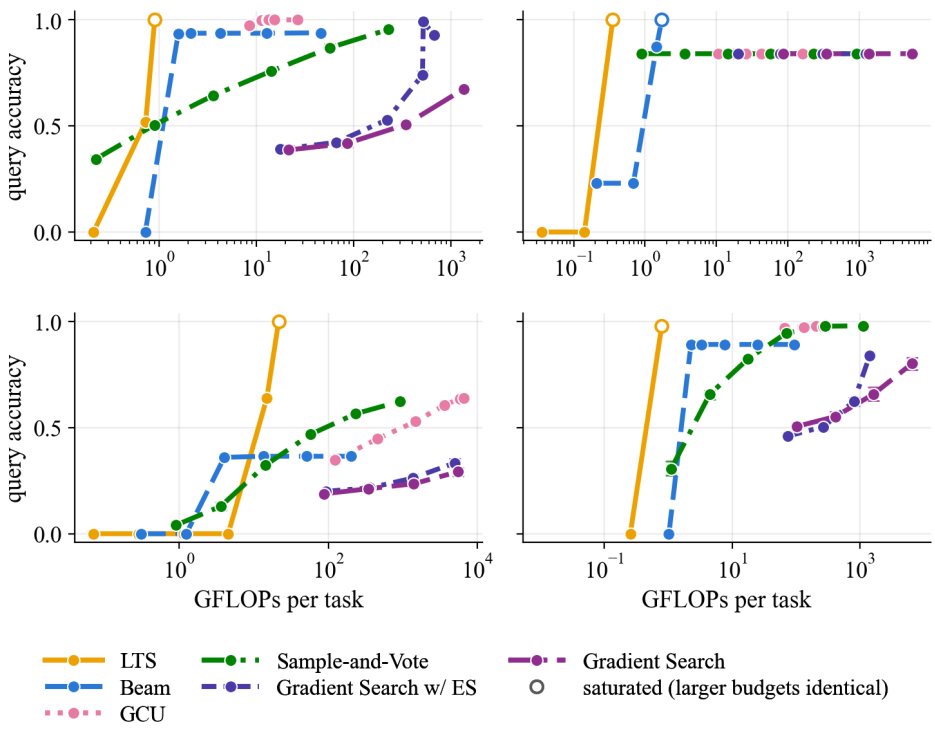


Figure 4: Performance (measured in query accuracy) vs Computational Cost (measured in GFLOPs/task) in Comp-I [top-left], Shift-L [top-right], Shift-P [bottom-left], Cell-Auto [bottom-right] when using NLI-R framework.

4.3 Cost of Autoregressive Program Search

Fig. 5 is generated similarly to Fig. 4 by varying the termination condition parameters and fixing the search parameters to their best values according to Tab. 2. In Fig. 5, we can see that NLI-R curves stay above NLI for a given search technique consistently (the exception being GCU in Shift-P). We also see that, for a given search technique and domain pair, NLI-R upper-bounds NLI performance across computational costs 9 out of 12 times. This shows that autoregressive program sequence generation can outperform one-shot generation across benchmarks and search techniques.

5 Related Work

In this section, we discuss related work not addressed elsewhere in the paper.

Neurally guided synthesis. Neural models have long guided program search in predefined domain-specific languages by predicting program properties (Balog et al., 2017) or jointly learning abstractions and a search policy (Ellis et al., 2021). Execution-guided synthesis conditions generation on the results of executing partial programs (Chen et al., 2019), combined with a value function and sequential Monte Carlo search in Ellis et al. (2019) and with predicted execution subgoals in ExeDec (Shi et al., 2024). NLI-R follows this approach, but in a learned, differentiable neural language.

Continuous search and program representations. Derivative-free optimizers offer alternatives to gradient-based search in continuous spaces, such as the cross-entropy method (de Boer et al., 2005), which LEAPS applies to learned program embeddings (Trivedi et al., 2021), and CMA-ES (Hansen, 2016). Since prior work on learned languages uses gradient-based search, we do not compare against these optimizers. Carvalho et al. (2024) find that direct search in a programmatic space can outperform search in learned latent spaces for programmatic policies; our results reach a similar conclusion for learned neural languages in PBE.

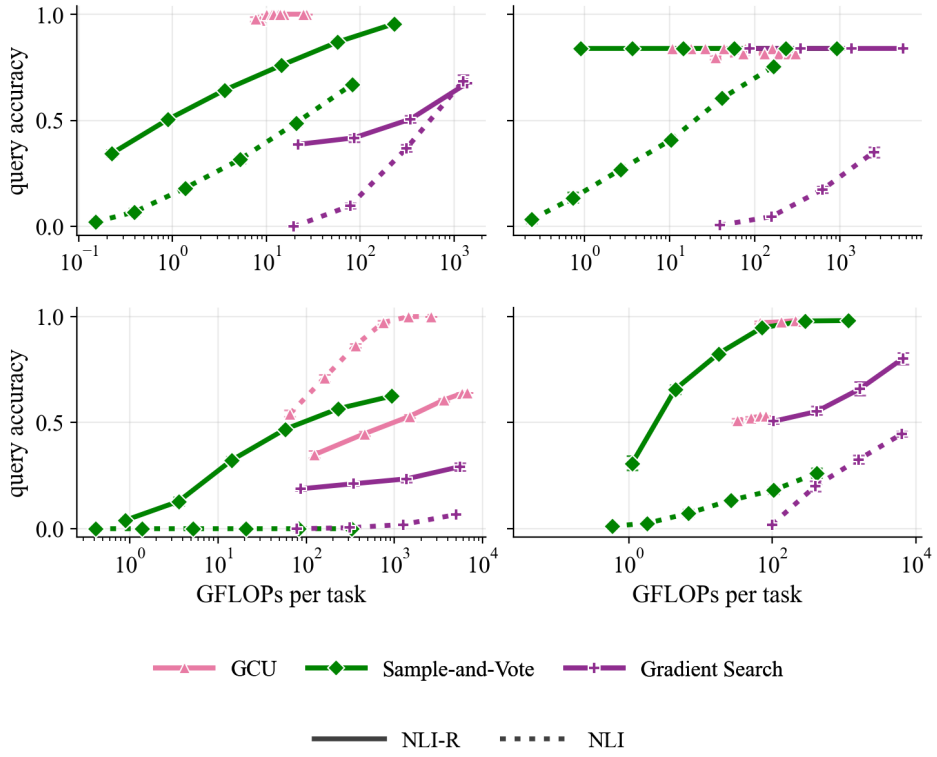


Figure 5: Performance vs Computational Cost in four benchmarks (following the same order as Fig. 4) for different categories of search techniques using NLI and NLI-R

Policy-guided tree search. Levin Tree Search (Orseau et al., 2018) bounds search effort by the probability a policy assigns to a solution path, and PHS extends it with a heuristic (Orseau & Lelis, 2021). NLI-R makes these methods applicable to learned languages by providing a policy over the next symbol and an interpreter for evaluating each prefix.

Test-time search. Recent work samples and aggregates reasoning paths (Wang et al., 2023), searches reasoning trees (Yao et al., 2023), allocates test-time compute for language-model reasoning (Snell et al., 2025), or updates model parameters on task examples (Akyürek et al., 2025). We instead compare search procedures over executable programs in a neural language.

6 Conclusion

In this paper, we studied the performance of NLI, a one-shot program generation architecture, and NLI-R, an autoregressive program generation architecture, augmented with several test-time search techniques. We found that using autoregressive program generation provides a significant performance gain at test time for a given computational cost (Sec. 4.3). Additionally, using discrete search techniques, especially tree search techniques, provides a two-orders-of-magnitude computational advantage for a given performance level (Sec. 4.2). These results suggest that although languages can be learned in a differentiable, continuous fashion, searching them discretely can be more effective. They also point to a clear direction for future work to develop methods that learn a language that can be utilized to its fullest with discrete test-time search.

Future work includes expanding the benchmark set to include larger, more complex domains. Showing the same conclusions on more complex benchmarks will further cement the insights from this study. Including more families of search techniques, such as CMA-ES (Hansen, 2016), will expand the library of search techniques studied and sharpen the decision boundary between gradient-guided and non-gradient-guided searches. Finally, future work should study the effects of vocabulary filtering in detail to see whether the insights translate across less expressive languages as well.

REFERENCES

Ekin Akyürek, Mehul Damani, Adam Zweiger, Linlu Qiu, Han Guo, Jyothish Pari, Yoon Kim, and Jacob Andreas. The surprising effectiveness of test-time training for few-shot learning. In *International Conference on Machine Learning*, 2025.

Doojin Baek, Gyubin Lee, Junyeob Baek, Hosung Lee, and Sungjin Ahn. Learning to theorize the world from observation. In *International Conference on Machine Learning*, 2026.

Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. DeepCoder: Learning to write programs. In *International Conference on Learning Representations*, 2017.

Tales Carvalho, Kenneth Tjhia, and Levi H. S. Lelis. Reclaiming the source of programmatic policies: Programmatic versus latent spaces. In *International Conference on Learning Representations*, 2024.

Xinyun Chen, Chang Liu, and Dawn Song. Execution-guided neural program synthesis. In *International Conference on Learning Representations*, 2019.

Pieter-Tjerk de Boer, Dirk P. Kroese, Shie Mannor, and Reuven Y. Rubinstein. A tutorial on the cross-entropy method. *Annals of Operations Research*, 134(1):19–67, 2005. doi: 10.1007/s10479-005-5724-z.

Javid Ebrahimi, Anyi Rao, Daniel Lowd, and Dejing Dou. Hotflip: White-box adversarial examples for text classification. In *Association of Computational Linguistics*, 2018.

Kevin Ellis, Maxwell Nye, Yewen Pu, Felix Sosa, Joshua B. Tenenbaum, and Armando Solar-Lezama. Write, execute, assess: Program synthesis with a REPL. In *Advances in Neural Information Processing Systems*, 2019.

Kevin Ellis, Catherine Wong, Maxwell Nye, Mathias Sablé-Meyer, Lucas Morales, Luke Hewitt, Luc Cary, Armando Solar-Lezama, and Joshua B. Tenenbaum. DreamCoder: Bootstrapping inductive program synthesis with wake-sleep library learning. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pp. 835–850, 2021. doi: 10.1145/3453483.3454080.

Nikolaus Hansen. The CMA evolution strategy: A tutorial, 2016.

Eric Jang, Shixiang Gu, and Ben Poole. Categorical reparameterization with gumbel-softmax. In *International Conference on Learning Representations*, 2017.

Henry Lieberman. *Your Wish is My Command: Programming by Example*, volume 1. Morgan Kauffmann, 2001.

Matthew V. Macfarlane and Clément Bonnet. Searching latent program spaces. In *Advances in Neural Information Processing Systems*, 2025.

Matthew V. Macfarlane, Clément Bonnet, Herke van Hoof, and Levi H. S. Lelis. Gradient-based program synthesis with neurally interpreted languages. In *International Conference on Learning Representations*, 2026.

Laurent Orseau and Levi H. S. Lelis. Policy-guided heuristic search with guarantees. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 2021.

Laurent Orseau, Levi H. S. Lelis, Tor Lattimore, and Theophane Weber. Single-agent policy tree search with guarantees. In *Advances in Neural Information Processing Systems*, 2018.

Kensen Shi, Joey Hong, Yinlin Deng, Pengcheng Yin, Manzil Zaheer, and Charles Sutton. ExeDec: Execution decomposition for compositional generalization in neural program synthesis. In *International Conference on Learning Representations*, 2024.

Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. Scaling LLM test-time compute optimally can be more effective than scaling parameters for reasoning. In *International Conference on Learning Representations*, 2025.

Dweep Trivedi, Jesse Zhang, Shao-Hua Sun, and Joseph J. Lim. Learning to synthesize programs as interpretable and generalizable policies. In *Advances in Neural Information Processing Systems*, 2021.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V. Le, Ed H. Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. In *International Conference on Learning Representations*, 2023.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. In *Advances in Neural Information Processing Systems*, 2023.

Andy Zou, Zifan Wang, Nicholas Carlini, Milad Nasr, J. Zico Kolter, and Matt Fredrikson. Universal and transferable adversarial attacks on aligned language models, 2023.

AI USE STATEMENT

In this work, we used generative AI tools for implementing the methods as executable code. We have not use generative AI tools to help conceptualize frameworks, propose or refine hypothesis, design research methodology or experiments. The rest of the required disclosure tasks are not applicable to this work. Additionally, we used generative AI tools to create the software code for plots, brain storming, identifying relevant literature, and format the research paper. We have reviewed all AI-assisted work when implementing the methods, formatting the paper, suggesting relevant literature, writing the plotting scripts. We take responsibility for the final content of this work, including text, claims, or artifacts produced with the aid of generative AI.

A Architecture Details

Table 3: Hyperparameters for NLI training.

Hyperparameter	Comp-I	Shift-L	Shift-P	Cell-Auto
Model Architecture				
Model Dimension (d_{model})	128	128	128	128
Number of Heads (n_{head})	8	8	8	8
Feed-Forward Dimension (d_{ff})	512	512	512	512
Encoder Layers	2	2	2	2
Decoder Layers	2	2	2	2
Positional Embedding	sinusoidal	sinusoidal	sinusoidal	learned grid
Program Generation				
Vocabulary Size (K)	512	512	512	512
Program Length (T)	2	4	8	4
Training				
Batch Size	512	512	512	512
Num Batches	100,000	100,000	100,000	150,000
Optimizer	Adam	Adam	Adam	Adam
Learning Rate	1×10^{-3}	1×10^{-4}	1×10^{-3}	8×10^{-4}
LR Schedule	cosine	cosine	cosine	cosine
Gradient Clip Norm	2.0	2.0	2.0	1.0
Gumbel-Softmax Sampling (Inductor)				
Use Gumbel	Yes	Yes	Yes	Yes
Start Temperature	8.0	8.0	8.0	8.0
End Temperature	0.1	0.1	0.1	0.1
Annealing Batches	100,000	100,000	100,000	150,000
Annealing Strategy	pointwise	pointwise	pointwise	linear
Straight-Through	No	No	No	No
Gumbel-Softmax Sampling (Interpreter)				
Use Gumbel	Yes	Yes	Yes	Yes
Start Temperature	2.0	2.0	2.0	2.0
End Temperature	0.1	0.1	0.1	0.1
Annealing Batches	100,000	100,000	100,000	150,000
Annealing Strategy	pointwise	pointwise	pointwise	linear
Straight-Through	No	No	No	No
Regularization & Losses				
Token Reuse Coefficient (λ)	1×10^{-5}	1×10^{-5}	1×10^{-5}	1×10^{-5}

Table 4: Hyperparameters for NLI-R training.

Hyperparameter	**Comp-I**	**Shift-L**	**Shift-P**	**Cell-Auto**
Model Architecture				
Model Dimension (d_{model})	128	128	128	128
Number of Heads (n_{head})	8	8	8	8
Feed-Forward Dimension (d_{ff})	512	512	512	512
Encoder Layers	2	2	2	2
Decoder Layers	2	2	2	2
Positional Embedding	sinusoidal	sinusoidal	sinusoidal	learned grid
Program Generation				
Vocabulary Size (K)	512	512	512	512
Program Length (T)	2	8	8	4
Training				
Batch Size	512	512	512	512
Num Batches	100,000	100,000	100,000	150,000
Optimizer	Adam	Adam	Adam	Adam
Learning Rate	5×10^{-4}	1×10^{-3}	5×10^{-4}	1×10^{-4}
LR Schedule	cosine	cosine	cosine	cosine
Gradient Clip Norm	2.0	2.0	2.0	1.0
Gumbel-Softmax Sampling (Inductor)				
Use Gumbel	Yes	Yes	Yes	Yes
Start Temperature	8.0	8.0	8.0	8.0
End Temperature	0.1	0.1	0.1	0.1
Annealing Batches	100,000	100,000	100,000	150,000
Annealing Strategy	pointwise	pointwise	pointwise	linear
Straight-Through	No	No	No	No
Gumbel-Softmax Sampling (Interpreter)				
Use Gumbel	Yes	Yes	Yes	Yes
Start Temperature	2.0	2.0	2.0	2.0
End Temperature	0.1	0.1	0.1	0.1
Annealing Batches	100,000	100,000	100,000	150,000
Annealing Strategy	pointwise	pointwise	pointwise	linear
Straight-Through	No	No	No	No
Regularization & Losses				
Token Reuse Coefficient (λ)	1×10^{-5}	1×10^{-5}	1×10^{-5}	1×10^{-5}

B TEST-TIME SEARCH DETAILS

Table 5: Test-time search hyperparameters swept for NLI. Ladder values are the grid searched; the vocabulary-filtering threshold τ is reported separately.

Hyperparameter	**Comp-I**	**Shift-L**	**Shift-P**	**Cell-Auto**
		Shared		
Held-out Test Tasks	200	200	200	100
Search Seeds	5	5	5	5
Gumbel Sampling at Search Time	No	No	No	No
Program Length (T)	2	4	8	4
		Sample-and-Vote		
Samples (budget)		$\{1, 4, 16, 64, 256, 1024\}$		
Vote Temperature (κ)	10.0	10.0	10.0	10.0
		Greedy Coordinate Update		
Restarts (S)	$\{1..1024\}$	$\{1..1024\}$	$\{1..1024\}$	$\{1..64\}$
Update Steps	100	100	100	100
Patience	1	1	1	1
Restart Noise (σ)	7.5	7.5	7.5	7.5
		Gradient Search		
Restarts (S)	$\{1, 4, 16, 64\}$	$\{1, 4, 16, 64\}$	$\{1, 4, 16, 64\}$	$\{1, 4, 16, 64\}$
Optimization Steps	100	100	100	100
Step Size	1×10^{-3}	1×10^{-3}	1×10^{-3}	1×10^{-3}
Restart Noise (σ)	7.5	7.5	7.5	7.5
Query Evaluation	hard	hard	hard	hard
Search Annealing (τ_e, τ_d)		$1.0 \rightarrow 0.1$ over 50 steps, linear		
		Gradient Search w/ Early Stopping		
Restarts (S)	$\{1..1024\}$	$\{1..64\}$	$\{1..64\}$	$\{1..64\}$
Optimization Steps	100	100	100	100
Step Size	1×10^{-3}	1×10^{-3}	1×10^{-3}	1×10^{-3}
Restart Noise (σ)	7.5	7.5	7.5	7.5
Query Evaluation	hard	hard	hard	hard
Search Annealing	None	None	None	None

Table 6: Test-time search hyperparameters swept for NLI-R. Ladder values are the grid searched; the vocabulary-filtering threshold τ is reported separately. Beam width is set to the filtered vocabulary size K_{filt}, which differs per domain.

Hyperparameter	**Comp-I**	**Shift-L**	**Shift-P**	**Cell-Auto**
Shared				
Held-out Test Tasks	200	200	200	100
Search Seeds	5	5	5	5
Gumbel Sampling at Search Time	No	No	No	No
Program Length (T)	2	8	8	4
Levin Tree Search				
Node Expansions (budget)	$\{1, 4, 16, 64, 256, 1024, 4096\}$			
Policy Mixing (ϵ)	1.0	1.0	1.0	1.0
Exclude Skip Token	Yes	Yes	Yes	Yes
Search Seeds	1	1	1	1
Beam Search				
Rounds (budget)	$\{1, 4, 16, 64, 256, 1024\}$			
Beam Width (K_{filt})	7	2	3	4
Policy Mixing (ϵ)	0.75	0.75	0.75	0.75
Sample-and-Vote				
Samples (budget)	$\{1, 4, 16, 64, 256, 1024\}$			
Vote Temperature (κ)	10.0	10.0	10.0	10.0
Greedy Coordinate Update				
Restarts (S)	$\{1..1024\}$	$\{1..1024\}$	$\{1..1024\}$	$\{1..64\}$
Update Steps	100	100	100	100
Patience	1	1	1	1
Restart Noise (σ)	7.5	7.5	7.5	7.5
Gradient Search				
Restarts (S)	$\{1, 4, 16, 64\}$	$\{1, 4, 16, 64\}$	$\{1, 4, 16, 64\}$	$\{1, 4, 16, 64\}$
Optimization Steps	100	100	100	100
Step Size	5×10^{-4}	5×10^{-4}	5×10^{-4}	5×10^{-4}
Restart Noise (σ)	7.5	7.5	7.5	7.5
Query Evaluation	hard	hard	hard	hard
Search Annealing (τ_e, τ_d)	$1.0 \rightarrow 0.1$ over 50 steps, linear			
Gradient Search w/ Early Stopping				
Restarts (S)	$\{1..1024\}$	$\{1..64\}$	$\{1..64\}$	$\{1..64\}$
Optimization Steps	100	100	100	100
Step Size	5×10^{-4}	5×10^{-4}	5×10^{-4}	5×10^{-4}
Restart Noise (σ)	7.5	7.5	7.5	7.5
Query Evaluation	hard	hard	hard	hard
Search Annealing	None	None	None	None

C FLOP COUNTING

Measuring compute. We report the number of floating-point operations performed by matrix multiplications during search, using the convention $2mnk$ for an $m \times k$ by $k \times n$ product (one multiply and one add per accumulation), which is the convention PyTorch's own FLOP registry uses. The count is *measured, not modelled*: a `TorchDispatchMode` intercepts every ATen operation actually dispatched inside the search call, looks up a per-operator formula, and accumulates. Nothing about the architecture is assumed, so the count automatically includes the matrix multiplications that automatic differentiation emits in the *backward* pass of the gradient-based procedures; these are counted exactly as forward ones are, rather than approximated by a fixed multiple of the forward cost. The procedures that never differentiate (beam search, Levin tree search, sample-and-vote) therefore incur no backward term at all, whereas for gradient search the backward pass is the dom-

inant cost. Because a single run at the largest swept budget determines every smaller one—the trajectory up to iteration b never depends on the total budget requested—we record the running total at each iteration, so one pass over the dataset yields an accuracy–compute curve rather than a single point.

Corrections to the stock tooling. Two corrections were necessary, and we note them because either would have silently corrupted the numbers. First, PyTorch's `flop_registry` has no formula for `aten::_transformer_encoder_layer_fwd`, the fused kernel into which `nn.TransformerEncoderLayer` compiles its attention, projection and feed-forward work. Since that kernel *is* essentially the whole model here, an unmodified counter reports only the multiplications occurring outside the transformer: on a small beam-search fixture it yields 0.745 MFLOP against a true 57.98, missing 98.7%, and it does so without warning. We supply the missing formula, deriving the shapes from the layer's actual weight tensors rather than from an assumed feed-forward width, together with formulas for the CPU scaled-dot-product-attention kernels, which are likewise unregistered. Second, we validate by *decomposition* rather than against a second counter: the identical workload is re-run with the fused path disabled, so the same arithmetic reappears as primitive `mm`/`bmm`/ `sdpa` operations that the registry has long supported, and the totals are compared. Two counters sharing one blind spot would agree to the digit on the wrong number. As a standing guard, any operator whose name indicates matrix-multiplication work but which carries no formula raises an error rather than contributing zero. Counting overhead is $1.02\times$ for beam search, $1.07\times$ for greedy coordinate update and $1.13\times$ for gradient search, so instrumenting every run is affordable and no analytic model is needed.

[1]

D VOCABULARY FILTERING

Table 7: vocabulary-filtering threshold used for the reported results. τ is the value each framework's selected checkpoint was searched under, and K_{filt} the resulting number of allowed tokens out of $K = 512$. Where the two frameworks were searched at different thresholds, those thresholds select identical token sets on the checkpoints used, so a single value labels the figure.

Filtering	**Comp-I**	**Shift-L**	**Shift-P**	**Cell-Auto**
Threshold searched (τ)				
NLI	0.1	0.1	0.01	0.33
NLI-R	0.1	0.5	0.5	0.33
Allowed vocabulary size (K_{filt})				
NLI	7	5	4	3
NLI-R	7	2	3	4
Reported				
Threshold shown in figures	0.1	0.1	0.01	0.33

[1]Three caveats apply to these figures. (i) Matrix multiplications only: softmax, layer normalization, element-wise operations and Gumbel sampling are excluded—a few percent of the total, and the standard convention for this kind of report, but not "all arithmetic". (ii) FLOPs are not wall-clock: at these batch sizes GPU utilisation is latency-bound, so two procedures with equal FLOPs can differ severalfold in seconds, and FLOPs are the half that transfers between machines. (iii) The model weights are not frozen during search. Only the program embedding requires gradients and only it is updated, but the parameters retain `requires_grad`, so weight gradients are computed and discarded. Those operations are real and are counted; measured on this model's encoder they account for roughly a third of the forward-plus-backward total, so the gradient-based procedures' counts are inflated by about that much relative to an implementation that freezes the interpreter.