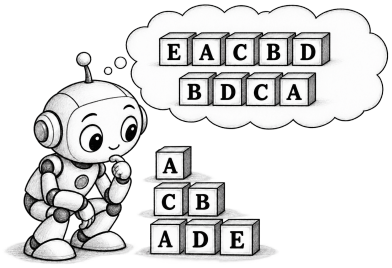
A BEGINNING OF INFINITY: GENERALIZATION BY DREAMING WITH NEURAL LANGUAGES

Anonymous authors
Paper under double-blind review

ABSTRACT

Generalization in neural networks is typically bounded by the amount and quality of the data used to train them. Recent neural models that learn a discrete language and its interpreter can represent programs far beyond their training distribution, but recovering these programs requires expensive search at test time. In this work, we show how this generalization capability can be transferred into a System 1 network, fully end to end. Once the learner has acquired a neural language, it can continue learning by dreaming. That is, it composes the primitives of its learned language into novel programs and executes them on inputs drawn from the training input distribution, generating new problems whose programs serve as exact training labels. The language thus becomes a source of new training problems, allowing the learner to acquire concepts that were never encoded in its original training set, without any domain-specific language, hand-written generator, or test-time search. If the learned language is universal for a problem class, dreaming can in principle reach every problem in it, even when the training data contains only a small fraction of the problems in that class. We evaluate our approach on several programming-by-example benchmarks and show that dreaming enables strong out-of-distribution generalization, matching or outperforming test-time search methods at orders of magnitude lower inference cost.

1 INTRODUCTION



The most powerful ideas have reached far beyond the problems that produced them; a few rules of arithmetic cover infinitely many sums. Neural networks, by contrast, rarely reach beyond the data that they were trained on. Despite this limitation, generalizing in-distribution has led to considerable progress via expanding the training data and model capacity (Kaplan et al., 2020), which continues to produce remarkable capabilities, as shown by large language models (LLMs). However, networks still fail at compositional and out-of-distribution (OOD) generalization (Lake & Baroni, 2018; Dziri et al., 2023): they learn concepts that should let them solve new problems, yet they remain constrained to the training distribution. One example is length generalization. No matter how long LLMs are trained on multiplication, there is always a length or difficulty of multiplication they cannot answer correctly (Anil et al., 2022; Dziri et al., 2023).

Recently, a class of neural networks that perform latent adaptation has shown interesting OOD generalization properties, representing programs that lie out of distribution relative to the training set. Latent Program Networks (LPN; Macfarlane & Bonnet, 2025) applied these ideas to difficult abstract reasoning puzzles (Chollet, 2019) and highlighted specific instances of compositional generalization that had emerged. However, this generalization was not always reliable, and some concepts could not be composed. Neural Language Interpreters (NLI; Macfarlane et al., 2026) expanded upon this work and achieved much stronger generalization, the primary shift being the use of a discrete neural language learnt end-to-end.

The key to generalization in these networks is learning a compressed latent space in which search can be performed at test time to explain the given data, with the compression limiting overfitting

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.

and making search efficient. However, this generalization comes at the cost of test-time search and compute, which is expensive to run for every new problem. In this work, we investigate how to transfer this generalization capability into the encoder, fully end-to-end, in order to build strong generalization into the neural network itself via learned latent spaces.

A natural way to move beyond the training data is for a learner to generate its own. If a system already knows a set of primitives, it can compose them into new problems, solve them, and store the solutions in a library, continually building more and more complex problems from simpler ones. This process, often called dreaming, has a long lineage, from the sleep phase of the wake-sleep algorithm (Hinton et al., 1995) to DreamCoder (Ellis et al., 2018; 2021), which alternates between growing a library of abstractions and training a recognition model on programs dreamed from it. Neural program synthesizers are likewise trained on randomly composed programs (Balog et al., 2017; Devlin et al., 2017), and recent approaches to abstract reasoning generate training tasks from procedural or LLM-written programs (Hodel, 2024; Li et al., 2025; Butt et al., 2024). However, a key problem limiting scalability is that these methods rely on a given language, complex search procedures, human priors, and hand-scripted functions and selection procedures.

In this work, we take the idea of composing new programs from primitives and apply it to learned neural languages in a fully end-to-end learned way, without hand-holding or injecting human priors. We then use these problems as additional synthetic training data for the System 1 network, allowing it to solve complex problems it has never seen before from simple primitives it has seen before. We build this idea on top of the NLI architecture in two stages. NLI consists of a program inductor, a learnable codebook, and a language interpreter. Given an input-output specification, the inductor produces a neural program over the codebook, which the interpreter then executes to produce an output.

In stage 1, we learn an end-to-end neural language. We then compose new programs from this neural language and execute them with the frozen interpreter to create input-output data. Because the interpreter is frozen, every composed program has a fixed meaning and serves as an exact label for the data it generates. The inductor is then trained to solve these imagined problems, using a cross-entropy loss on the teacher-forced tokens of the generating programs. Crucially, this requires no DSL, no generation function, no hand-specified primitives, and no complex search method. The only assumption is that the initial training dataset is composed of primitives that can be extracted and used to create new problems.

The result is a dataset-in, encoder-out procedure that generalizes to OOD problems out of the box, through only deep learning training and automatic, systematic problem generation. Furthermore, if the induced language is universal for a problem class, dreaming can in principle reach every problem in it. By introducing a new recurrent inductor to NLI, we can express unbounded programs, compared to the bounded NLI inductor. Combined with a token reuse loss, the recurrent inductor also encourages the model to break programs down into simple, reusable primitives, which is exactly what makes them composable during dreaming. We find that this training method instills OOD generalization in the encoder, matching or outperforming a range of search methods, but at multiple orders of magnitude lower FLOPs. We show that the recurrent NLI can express very long programs, far beyond the capability of search and the base NLI encoder. Finally, we investigate the impact of the sampled languages and training targets in dreaming.

2 Background

2.1 Programming by Examples

We formalize our task under the programming-by-example (PBE) setting. Each task is defined by a specification set of input/output (I/O) examples $S = \{(x_i, y_i)\}_{i=1}^{m}$, where $y_i = p(x_i)$ is generated by a latent program p sampled from a distribution P over a space of programs. Crucially, p itself is never directly observed; it is accessible only through the generated I/O examples in S. The goal is to infer a program M that satisfies S, and its performance is measured by the loss between the prediction $M(x_{m+1})$ and the ground truth $p(x_{m+1})$ on a held-out query input x_{m+1}. This is to ensure that the inferred program generalizes to unseen inputs rather than merely memorizing the specification. To test for generalization, the program distribution P may differ between training and test-time, which leads to in-distribution (ID) and out-of-distribution (OOD) settings.

2.2 Latent Program Networks

Latent Program Networks (LPN; Macfarlane & Bonnet, 2025) treat PBE as inference over a learned space of programs rather than search over a hand-designed DSL. The architecture consists of an encoder and a decoder. The encoder maps the specification S to a continuous latent program z, and the decoder executes that latent on new inputs through $p_\theta(y \mid x, z)$. The decoder never sees S, so z must carry everything specific to the task at hand. Inference in LPN is semi-amortized (Kim et al., 2018). The encoder provides a fast initial guess, which is then refined by gradient ascent on the likelihood the decoder assigns to S, so that additional compute at test time can correct errors in the amortized proposal. This search is what drives the out-of-distribution performance of LPN, but it is expensive and operates over a continuous space with no explicit compositional structure.

2.3 Neural Language Interpreters

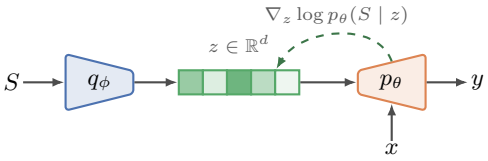


Figure 1: Latent Program Networks. The encoder q_ϕ maps the specification S to a continuous latent program $z \in \mathbb{R}^d$, which the decoder p_θ executes on a new input x to predict y. At test time z is refined by gradient ascent on the likelihood of S (dashed).

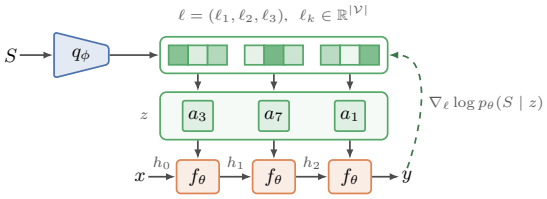


Figure 2: Neural Language Interpreters. The inductor q_ϕ maps S to latents ℓ, each projected to a token from a learned vocabulary $\mathcal{V}$ to form the program z. A shared interpreter f_θ executes one step per token, mapping x to y. At test time ℓ is refined by gradient ascent on the likelihood of S (dashed).

Neural Language Interpreters (NLI; Macfarlane et al., 2026) replace the continuous latent of LPN with a discrete one. An inductor maps S to a short sequence of tokens drawn from a learned codebook, and an interpreter executes this latent program by applying a single shared network once per token to update an intermediate state. Since every token is read by the same mechanism wherever it appears, tokens behave as reusable primitives of a learned language, and a longer program is simply more applications of the same interpreter. Gumbel-Softmax relaxations on both the inductor and the interpreter keep the whole system differentiable, which makes gradient-based search over the discrete program space possible. With this search, NLI recovers programs built from primitives that were never combined during training, and removing either the recurrence or the discreteness collapses this out-of-distribution accuracy to near zero. This generalization, however, still comes at the price of extensive test-time search, with thousands of initializations per task.

3 Method

Building on the compositional structure of NLI, we introduce *dreaming*, a second stage training that follows standard training that learns the neural language. During the first stage, the model learns a library of reusable tokens along with an interpreter. A dreamer then composes the learned tokens into programs of varying length and composition, and executes them with the learned interpreter to generate new I/O pairs. By retraining the inductor directly on these composed programs, dreaming naturally enables OOD generalization and removes the need for the expensive test-time search.

3.1 Universal Languages

We define a universal language with respect to a target problem class. That is, a language is universal if, for every problem in the class, there exists a program in the space defined by the language that can solve it. In our PBE setting, a problem is a function $f : \mathcal{X} \to \mathcal{Y}$ from an input space $\mathcal{X}$ to an output space $\mathcal{Y}$, and a problem class is a set $\mathcal{F}$ of such functions. The specification S provides examples of a problem's behavior; solving the problem means computing its outputs on every valid input.

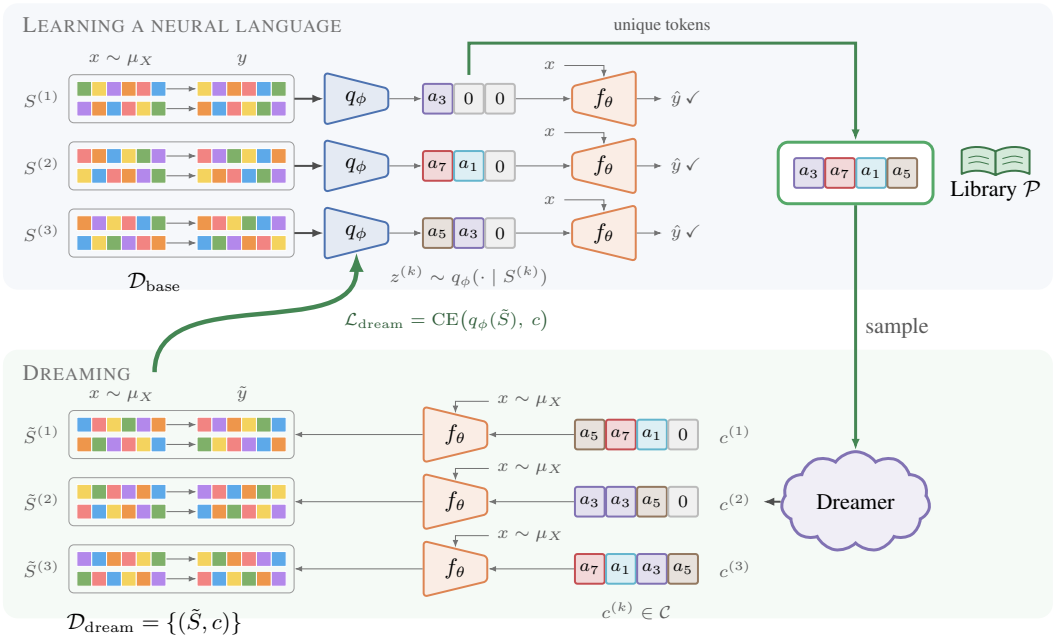


Figure 3: Overview of dreaming. In the first stage the inductor q_ϕ and decoder f_θ are trained on $\mathcal{D}_{\text{base}}$, and the unique tokens of programs that solve their tasks form the library $\mathcal{P}$. In the second stage a dreamer samples from $\mathcal{P}$ to write new programs $c \in \mathcal{C}$, which the frozen decoder applies to inputs from the same distribution μ_X to produce $\mathcal{D}_{\text{dream}}$. The inductor is then trained to recover each generating program c from its synthetic specification $\tilde{S}$ with a cross-entropy loss.

A language is a pair $\mathcal{L} = (\mathcal{Z}, I)$, where $\mathcal{Z}$ is a set of valid programs one can write with $\mathcal{L}$ and $I : \mathcal{Z} \times \mathcal{X} \to \mathcal{Y}$ is an interpreter defining the semantics of the programs. For the languages we can learn with NLI, $\mathcal{Z} = \mathcal{V}^*$ is the set of all finite symbol sequences over a finite learned vocabulary $\mathcal{V}$. In a learned neural language, the semantics of these symbols are determined by the interpreter parameters, which are held fixed when assessing universality.

Definition 1 *We say that $\mathcal{L}$ is universal for $\mathcal{F}$ if $\forall f \in \mathcal{F}, \exists z \in \mathcal{Z}, \forall x \in \mathcal{X}, I(z, x) = f(x)$.*

Thus, every problem in the class has at least one program that solves it across all inputs. Universality is a property of the language's expressiveness, and recovering a program that solves the problem from an observed specification is the role of the inductor.

Example 1 *Let us consider the problem class of shifting integers in a list. For example, shifting by two positions could yield the following input-output example: $[1, 2, 3, 4] \to [3, 4, 1, 2]$. Fix a list length $n \geq 1$, and let $R : \mathbb{Z}^n \to \mathbb{Z}^n$ cyclically shift a list one position to the right. Consider the class $\mathcal{F}_{\text{shift}} = \{R^k : k \in \mathbb{N}_0\}$, where R^0 is the identity. A language with vocabulary $\mathcal{V} = \{s\}$, whose interpreter applies R once per token, is universal for this class: for every k, the program s^k consisting of k repetitions of s computes R^k. The empty program computes the identity.*

In our definition, the language $\mathcal{V}^*$ allows arbitrary finite program lengths. NLI's original architecture is restricted to T program slots. In this paper, we show a recurrent version of NLI's inductor (Section 3.2) which in principle allows program length to grow unbounded and encode $\mathcal{V}^*$.

3.2 Learning a Neural Language

NLI consists of a program inductor q_ϕ, a language interpreter f_θ, and a codebook E_v. Given a specification, the inductor produces, in a one-shot manner, a program distribution z over the codebook spanning all T time steps. The interpreter then executes the selected tokens on a query input x one step at a time, decoding a latent state h_t into a candidate output $\tilde{\pi}_t$ (see Figure 2 for visualizations).

The first codebook entry serves as a skip token: its probability mass $z_{t,\text{skip}}$ linearly interpolates between the previous output π_{t-1} and the candidate output $\tilde{\pi}_t$, controlling how much of the previous output carries into the current one:

$$\pi_t = z_{t,\text{skip}} \cdot \pi_{t-1} + (1 - z_{t,\text{skip}}) \cdot \tilde{\pi}_t. \tag{1}$$

A main limitation of NLI is that the program length T is fixed in advance, so any problem requiring more than T steps is unsolvable. To address this, we introduce NLI-R, a recurrent variant of NLI in which the inductor generates the program one token at a time. Rather than producing the entire program at once, NLI-R's inductor leverages intermediate state information and infers one program token at a time. Conditioned on the current state $x_{\text{spec},t}$ and the goal state y_{spec}, the inductor produces the next program token z_t, which the interpreter then executes on $x_{\text{spec},t}$ to produce the next state $x_{\text{spec},t+1}$. Because program tokens are generated and executed sequentially, the program length is no longer fixed: NLI-R can execute programs of arbitrary length, allowing it to solve problems that exceed NLI's T-step ceiling.

In contrast with NLI, the intermediate state information tells the model how far the current execution state is from the goal, allowing it to select program tokens more reliably. However, this comes at the cost of one inductor call per step, compared to a single call for the entire program in NLI.

Both NLI and NLI-R are trained to minimize the sum of two losses: a reconstruction loss and a token reuse loss. The reconstruction loss is to ensure the inferred program, once interpreted, produces the correct output. However, the model could trivially minimize the reconstruction loss by learning a macro for every task, never learning to compose reusable tokens at all. The auxiliary reuse loss prevents this: it measures the expected number of distinct tokens used across batches b and time steps t, encouraging the model to reuse as many tokens in the program as possible, and thereby encouraging it to solve tasks using compositions of reusable tokens in a compact library.

$$\mathcal{L}_{\text{train}}(\phi, \theta, E_V) = \underbrace{-\mathbb{E}_S \left[\log f_\theta(y \mid x, q_\phi(S), E_v)\right]}_{\mathcal{L}_{\text{recon}}(\phi,\theta,E_v)} + \lambda \underbrace{\sum_{k=1}^{|E_v|} \left(1 - \prod_{b=1}^{B} \prod_{t=1}^{T} (1 - q_\phi(S)_{b,t,k})\right)}_{\mathcal{L}_{\text{reuse}}(\phi)}. \tag{2}$$

Token reuse is essential for dreaming, which extrapolates to novel tasks by recombining learned tokens. Dreaming can only generate problems that the learned language can express, so its coverage is bounded by how close the language is to being universal for the problem class. Task-specific macros can still be composed into new programs, but if the vocabulary lacks the underlying primitives, some functions in the class have no corresponding program and remain out of reach. Learning primitives rather than macros therefore brings the language closer to universality, allowing dreaming to cover the problem class more broadly.

3.3 DREAMING

One main reason neural architectures fail to generalize OOD without extensive test-time search is that they never learn from the I/O examples generated by unseen compositions. Dreaming supplies this missing information through a three-step recipe: (1) the dreamer *extracts* the reusable tokens learned during standard training; (2) it *composes* them into longer and more diverse programs than those that appeared in the training data, and *executes* them with the learned interpreter to synthesize a diverse set of I/O examples; and (3) the model's inductor *learns* directly from the compositions in the dream. The execution process is detailed in Algorithm 1.

Extract reusable codes. The dreamer first evaluates the model on ID tasks and collects the programs that solve them correctly, with skip and other identity tokens stripped out. Since a primitive is not necessarily represented by a single token, a Byte-Pair Encoding (BPE) tokenizer (Sennrich et al., 2016) is applied to identify frequently occurring subsequences of tokens. Subsequences that appear more often than a frequency threshold form a library $\mathcal{P}$.

Compose and synthesize dreams. Using the library $\mathcal{P}$, the dreamer recombines tokens $p \in \mathcal{P}$ into compositions up to a maximum depth D. All compositions are constrained to length T: those

Algorithm 1 Dreaming Stage

Require: Trained model M, ID dataset $\mathcal{D}_{\text{ID}}$, frequency threshold ϵ, maximum dream depth D

$\mathcal{V} \leftarrow \emptyset$
for $(S, x_{m+1}, y_{m+1}) \in Q_{\text{ID}}$ **do**
  if $M(S, x_{m+1}) = y_{m+1}$ **then**
    $\mathcal{V} \leftarrow \mathcal{V} \cup \{q_\phi(S))\}$ ▷ Collect programs that solve ID tasks
  end if
end for
$\mathcal{V} \leftarrow \mathcal{V} \setminus \{\text{skip}, \text{identity}\}$ ▷ Strip useless tokens
$\mathcal{P} \leftarrow \text{BPE}(\mathcal{V}, \epsilon)$ ▷ Apply BPE tokenizer to extract library
$\mathcal{C} \leftarrow \text{dedup}(\mathcal{P} \cup \mathcal{P}^2 \cup \cdots \cup \mathcal{P}^D)$ ▷ Generate distinct compositions
$\mathcal{D}_{\text{dream}} \leftarrow \{\{(x_i, f_\theta(x_i, c)\}_{i=1}^{m+1}, c)\}, c \in \mathcal{C}, x \sim \mu_X$ ▷ Synthesize dreaming dataset
for $i = 1 \rightarrow n$ **do**
  $\phi_{t+1} \leftarrow \phi_t - \alpha \nabla_\phi \mathcal{L}_{\text{dream}}$ ▷ Update only the inductor
end for

exceeding T are discarded to remain compatible with NLI, while shorter ones are padded to length T with skip tokens. The set of compositions with exactly depth d is defined as

$$\mathcal{C}_d = \{\, c = p_1 \| p_2 \| \cdots \| p_d \;:\; p_i \in \mathcal{P},\ |c| \le T \,\}. \tag{3}$$

The learned interpreter is then used to deduplicate the compositions based on their behaviors to support stable dreaming. The dreamer then draws inputs at random from the input distribution, and passes each composition through the interpreter to produce the corresponding outputs. This yields a diverse set of I/O pairs far beyond those seen during training. Both the dreamed I/O pairs and their corresponding ground-truth programs are made available to the model during dreaming.

Learn from dreams. Only the program inductor parameters are updated in the dreaming phase; the interpreter is frozen so that each token is correctly interpreted as usual. Given a specification $\widetilde{S}$ from the dreaming dataset, the model is trained to directly minimize the cross-entropy loss between its predicted program and the ground-truth program c:

$$\mathcal{L}_{\text{dream}}(\phi) = -\,\mathbb{E}_{(\widetilde{S},c)} \left[\log q_\phi(c \mid \widetilde{S}) \right]. \tag{4}$$

For NLI-R, the inductor is trained with a per-step cross-entropy loss under teacher forcing: at each step, the model minimizes the cross-entropy between its predicted token and the ground-truth token, and the ground-truth token is then executed to produce the next state.

4 Experiments

We evaluate the length and compositional generalization capabilities of the models across three custom benchmarks. Specifically, our experiments assess whether the models can (1) generalize to tasks requiring programs of unseen length, and (2) decompose complex behaviors into smaller reusable ones, and recombine them to solve tasks requiring unseen compositions.

4.1 Setup

Comp-I is a one-dimensional digit-manipulation benchmark with six primitives: `shift-1`, `reverse`, `interleave`, `deinterleave`, `swap_adjacent` and `swap_halves`. Each primitive operates on a sequence of 10 digits. During training, models see only I/O pairs generated by single primitives, while OOD evaluation requires composing them into programs of length 2–4.

Cell-Auto is a two-dimensional world-modeling benchmark whose primitive set consists of three cellular automata: Game of Life, Day & Night and Morley. Each state is a 5×5 binary grid, and each cellular automaton defines a local update rule that maps one state to the next. As in Comp-I, models are trained on single primitives and evaluated on compositions of length 2–4.

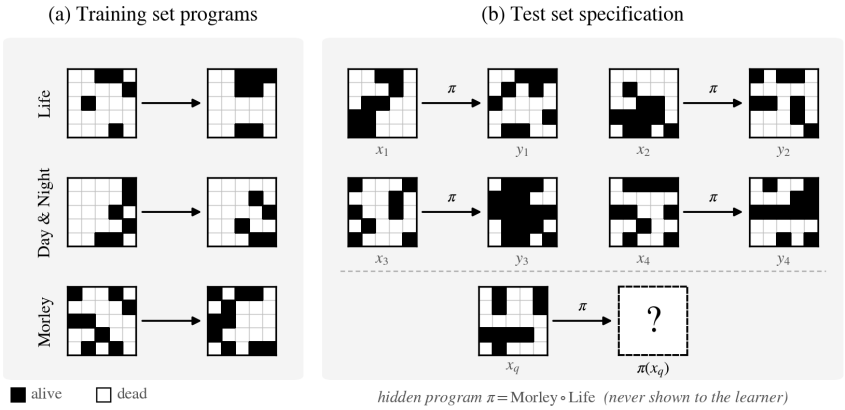


Figure 4: Cellular Automaton Benchmark. There are three Cellular Automata in the primitive set: Life, Day & Night and Morley. Each defines a rule to update a 5×5 binary grid. (a) Training specifications use single primitives. (b) Test specifications use unseen compositions, here Morley ∘ Life, from which the model must infer the hidden program.

Comp-Reverse is the opposite of Comp-I: rather than composing known primitives into longer programs, models must decompose longer programs into reusable primitives. Trained only on specifications of length 2–3, models must recover the underlying primitives and recompose them for unseen compositions (length 1 and 4) at test time.

We compare to the following baselines: LPN, NLI and NLI-R, each evaluated with several inference strategies: base inference (direct encoder output), prior search (sampling from the encoder), gradient search (gradient descent from the encoder output), and discrete search (sampling many rollouts).

4.2 Generalization through Dreaming

Test-time search shows that the learned language already contains solutions to out-of-distribution tasks. As shown in Figure 5, both NLI gradient search and NLI-R discrete search recover an increasing fraction of OOD tasks as inference compute grows, reaching 61% and 59% on Comp-I respectively. The inductor alone, however, cannot find these programs: without search, NLI achieves near-zero OOD accuracy on all benchmarks, and NLI-R reaches only 14–38% (Table 1). The information needed to generalize is therefore present in the language, but it is recovered only by paying for search on every new task.

Dreaming transfers this capability into the inductor itself. Without changing the architecture or the inference procedure, training the inductor on dreamed problems raises OOD accuracy from 0.00 to

Table 1: Results for different methods and datasets, in the custom suite. We report final accuracy for both in-distribution and out-of-distribution test splits (ID and OOD), along with inference FLOPs.

	Comp-I			**Cell-Auto**			**Comp-Reverse**		
Method	ID	OOD	FLOPs	ID	OOD	FLOPs	ID	OOD	FLOPs
LPN	1.00	0.00	7.5e7	1.00	0.03	1.8e8	1.00	0.00	7.5e7
NLI	1.00	0.00	1.1e8	1.00	0.03	2.5e8	0.76	0.01	3.8e8
NLI-R	1.00	0.14	4.4e8	1.00	0.38	1.1e9	0.93	0.33	1.7e9
LPN Prior Search	1.00	0.00	7.2e10	1.00	0.04	1.7e11	1.00	0.01	7.2e10
LPN Gradient Search	1.00	0.00	7.2e9	1.00	0.03	1.8e10	1.00	0.26	7.2e9
NLI Prior Search	1.00	0.00	2.7e11	1.00	0.04	6.7e11	0.76	0.04	5.6e11
NLI Gradient Search	1.00	0.61	5.4e13	0.95	0.70	1.4e14	0.83	0.40	1.1e14
NLI-R Discrete Search	1.00	0.59	8.5e11	0.97	0.58	2.1e12	1.00	0.56	1.7e12
NLI Dreaming	1.00	**1.00**	1.1e8	1.00	0.61	2.5e8	1.00	0.99	3.8e8
NLI-R Dreaming	1.00	0.99	4.4e8	1.00	**0.84**	1.1e9	1.00	**1.00**	1.7e9

1.00 on Comp-I and from 0.01 to 0.99 on Comp-Reverse for NLI, and from 0.14, 0.38 and 0.33 to 0.99, 0.84 and 1.00 for NLI-R across the three benchmarks. The best dreaming model matches or exceeds the OOD accuracy of every search method on all three benchmarks.

On Cell-Auto, NLI Dreaming (0.61) remains below NLI gradient search (0.70), while NLI-R Dreaming reaches 0.84. We attribute this to the recurrent inductor, which conditions each token on the current intermediate state and can therefore track progress towards the goal during execution, a benefit that is most pronounced on the two-dimensional cellular automata where intermediate states are less predictable from the specification alone.

Finally, dreaming does not trade in-distribution performance for OOD generalization. All dreaming models achieve perfect ID accuracy, and on Comp-Reverse dreaming even improves ID accuracy from 0.76 to 1.00 for NLI and from 0.93 to 1.00 for NLI-R, as the dreamed compositions also cover the lengths seen during training.

4.3 Test Time Inference Efficiency

Figure 5 plots inference FLOPs per task against OOD accuracy for gradient search, discrete search and dreaming. NLI gradient search and NLI-R discrete search improve steadily as their compute budget grows, reaching around 60% on Comp-I with 10^{12}–10^{14} FLOPs. LPN gradient search, by contrast, stays near zero across its entire budget. The dreaming models reach near-perfect OOD accuracy at only 10^{8}–10^{9} FLOPs, which comes at the same cost as the base model (Table 1). This is because dreaming only requires a single forward pass of the inductor, leaving the inference unchanged. This saves three to five orders of magnitude in inference FLOPs than test-time search.

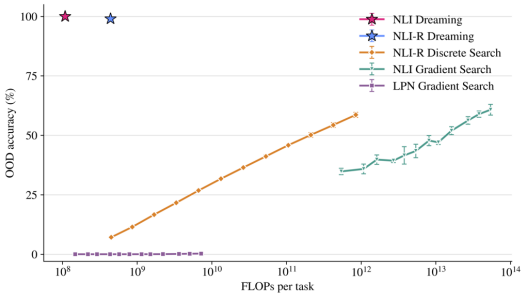


Figure 5: FLOPs scaling of test-time inference (Comp-I).

4.4 Dream to Infinity

We next test how far beyond the training distribution each model can generalize. On Comp-Reverse, models are trained on programs of length 2–3 and evaluated on compositions up to depth 16 (Figure 6). Without dreaming, NLI solves only the training lengths, as its one-shot inductor cannot produce programs outside them. NLI-R generalizes further by conditioning each token on the intermediate state, but its accuracy still falls to zero beyond depth 5.

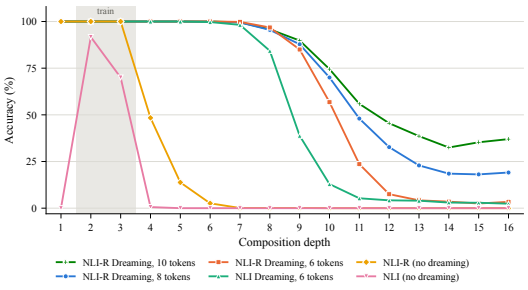


Figure 6: Length generalization of NLI and NLI-R on Comp-Reverse (single seed).

Dreaming substantially extends this range. With dreams of at most 6 tokens, NLI remains near-perfect up to depth 7, while NLI-R remains near-perfect up to depth 8 and above 50% at depth 10, generalizing beyond even the longest dreamed program. Recurrence and dreaming are complementary: dreaming supplies training signal for long compositions, and the recurrent inductor extends it past the lengths trained on.

Most strikingly, increasing the dream length to 8 tokens changes the shape of the curve. NLI-R no longer collapses: its accuracy decays to around 70% at depth 10 and stabilizes at roughly 20% from depth 13 onward, remaining non-zero at depth 16, twice the longest dreamed program and over five times the longest training program. Every other model reaches zero well before this point. Given a rich set of dreams, the recurrent inductor learns a length-general procedure rather than a fixed set

of program lengths. Since dreams are generated rather than collected, reaching longer programs requires only longer dreams, and a universal language places no bound on how far this can go.

5 Related Work

Learned neural languages. Differentiable interpreters made program execution amenable to gradient descent, learning programs within a fixed language (Gaunt et al., 2016; Bošnjak et al., 2017). Later work moved progressively more of the language into the network: LaSynth learns a latent executor (Chen et al., 2021), Latent Programmer learns discrete codes that plan generation (Hong et al., 2021), and Neural Interpreters, CompILE and PRISE learn reusable functions, sub-task codes and skills from data (Rahaman et al., 2021; Kipf et al., 2019; Zheng et al., 2024). LPN and NLI go further, learning the program space itself end to end from I/O examples: LPN as a continuous latent space executed by a neural decoder (Macfarlane & Bonnet, 2025), NLI as a discrete vocabulary of primitives given meaning by a recurrent interpreter, so programs become sequences of reusable tokens (Macfarlane et al., 2026). We build on this discrete, compositional structure, which lets learned primitives be recombined into entirely new programs.

Test-time search in learned program spaces. Learned program spaces can be searched at test time to adapt a latent program to each task. LPN runs gradient search in its continuous latent space from the encoder's proposal (Macfarlane & Bonnet, 2025); NLI searches its discrete space via a Gumbel-Softmax relaxation, recovering primitive compositions unseen in training (Macfarlane et al., 2026); and NEO builds on both with an alternative search procedure (Baek et al., 2026). Beyond program synthesis, inverting a structured decoder by search from an encoder proposal improves out-of-distribution perception (Brady et al., 2026). We take a complementary route, transferring this capability into the encoder during training so it is available in a single forward pass.

Dreaming and self-generated training problems. In dreaming, a learner uses its own knowledge to generate synthetic training problems beyond those it was given, an idea dating to the sleep phase of wake-sleep, where a recognition network learns from a generative model's samples (Hinton et al., 1995). It is central to EC^2 and DreamCoder, which grow a library of abstractions and train a recognition model on programs dreamed from it (Ellis et al., 2018; 2021), with Stitch and LILO improving abstraction discovery (Bowers et al., 2023; Grand et al., 2024). Training on sampled programs also benefits neural synthesizers (Balog et al., 2017; Devlin et al., 2017) and abstract reasoning, with tasks generated from procedural or LLM-written programs (Hodel, 2024; Li et al., 2025) or by relabelling a model's own attempts (Butt et al., 2024; Pourcel et al., 2025). Learners can also continually invent and solve their own problems (Schmidhuber, 2013; Zhao et al., 2025), and the principle extends beyond programs: generative replay recombines learned object slots, decodes them, and trains the encoder to invert the result (Brady et al., 2026). We extend this work to learned neural languages, generating new training problems from primitives discovered from data.

6 Conclusion

We introduced dreaming, a second training stage that moves out-of-distribution generalization from test-time search into the inductor. In the first stage, the model learns a discrete vocabulary of primitives together with an interpreter. The dreamer then recombines these primitives into new programs and executes them through the frozen interpreter, so each dreamed problem comes with its program as a ground-truth label, a strong signal for retraining the inductor. Nothing is supplied from outside: the primitives are those already extracted from the training data, and no further search is involved. We also study NLI-R, whose inductor emits one token at a time conditioned on the current execution state, allowing programs of arbitrary length. On three benchmarks, dreaming reaches near-perfect OOD accuracy and matches or beats every test-time search baseline at three to five orders of magnitude less inference cost. NLI-R also solves a portion of compositions deeper than any it was trained or dreamed on, suggesting that recurrence and dreaming together can extend generalization far beyond the dreamed lengths. However, dreaming relies on extracting reusable primitives from the training data. The training compositions of our benchmarks reach only depth 3; whether clean primitives can be extracted in more complex environments remains an open question.

REFERENCES

Cem Anil, Yuhuai Wu, Anders Andreassen, Aitor Lewkowycz, Vedant Misra, Vinay Ramasesh, Ambrose Slone, Guy Gur-Ari, Ethan Dyer, and Behnam Neyshabur. Exploring length generalization in large language models. In *Advances in Neural Information Processing Systems*, volume 35, pp. 38546–38556. Curran Associates, Inc., 2022. doi: 10.52202/068431-2793. URL https://proceedings.neurips.cc/paper_files/paper/2022/hash/fb7451e43f9c1c35b774bcfad7a5714b-Abstract-Conference.html.

Doojin Baek, Gyubin Lee, Junyeob Baek, Hosung Lee, and Sungjin Ahn. Learning to theorize the world from observation. In *Proceedings of the 43rd International Conference on Machine Learning*, 2026. URL https://arxiv.org/abs/2605.03413.

Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. DeepCoder: Learning to write programs. In *International Conference on Learning Representations*, 2017. URL https://openreview.net/pdf?id=ByldLrqlx.

Matko Bošnjak, Tim Rocktäschel, Jason Naradowsky, and Sebastian Riedel. Programming with a differentiable Forth interpreter. In *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pp. 547–556. PMLR, 2017. URL https://proceedings.mlr.press/v70/bosnjak17a.html.

Matthew Bowers, Theo X. Olausson, Lionel Wong, Gabriel Grand, Joshua B. Tenenbaum, Kevin Ellis, and Armando Solar-Lezama. Top-down synthesis for library learning. *Proceedings of the ACM on Programming Languages*, 7(POPL), 2023. doi: 10.1145/3571234. URL https://doi.org/10.1145/3571234.

Jack Brady, Bernhard Schölkopf, Thomas Kipf, Simon Buchholz, and Wieland Brendel. Is generation required for data-efficient perception? In *Proceedings of the 43rd International Conference on Machine Learning*, 2026. URL https://arxiv.org/abs/2512.08854v3.

Natasha Butt, Blazej Manczak, Auke Wiggers, Corrado Rainone, David W. Zhang, Michaël Defferrard, and Taco Cohen. CodeIt: Self-improving language models with prioritized hindsight replay. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 5013–5034. PMLR, 2024. URL https://proceedings.mlr.press/v235/butt24a.html.

Xinyun Chen, Dawn Song, and Yuandong Tian. Latent execution for neural program synthesis beyond domain-specific languages. In *Advances in Neural Information Processing Systems*, volume 34, pp. 22196–22208. Curran Associates, Inc., 2021. URL https://proceedings.neurips.cc/paper/2021/hash/ba3c95c2962d3aab2f6e667932daa3c5-Abstract.html.

François Chollet. On the measure of intelligence. *arXiv preprint arXiv:1911.01547*, 2019. URL https://arxiv.org/abs/1911.01547.

Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel rahman Mohamed, and Pushmeet Kohli. RobustFill: Neural program learning under noisy I/O. In *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pp. 990–998. PMLR, 2017. URL https://proceedings.mlr.press/v70/devlin17a.html.

Nouha Dziri, Ximing Lu, Melanie Sclar, Xiang Lorraine Li, Liwei Jiang, Bill Yuchen Lin, Peter West, Chandra Bhagavatula, Ronan Le Bras, Jena D. Hwang, Soumya Sanyal, Sean Welleck, Xiang Ren, Allyson Ettinger, Zaid Harchaoui, and Yejin Choi. Faith and fate: Limits of transformers on compositionality. In *Advances in Neural Information Processing Systems*, volume 36, 2023. URL https://arxiv.org/abs/2305.18654.

Kevin Ellis, Lucas Morales, Mathias Sablé-Meyer, Armando Solar-Lezama, and Josh Tenenbaum. Learning libraries of subroutines for neurally–guided Bayesian program induction. In *Advances in Neural Information Processing Systems*, volume 31, 2018. URL https://proceedings.neurips.cc/paper_files/paper/2018/hash/7aa685b3b1dc1d6780bf36f7340078c9-Abstract.html.

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. Association for Computing Machinery, 2021. doi: 10.1145/3453483.3454080. URL https://doi.org/10.1145/3453483.3454080.

Alexander L. Gaunt, Marc Brockschmidt, Rishabh Singh, Nate Kushman, Pushmeet Kohli, Jonathan Taylor, and Daniel Tarlow. TerpreT: A probabilistic programming language for program induction. *arXiv preprint arXiv:1608.04428*, 2016. URL https://arxiv.org/abs/1608.04428.

Gabriel Grand, Lio Wong, Maddy Bowers, Theo X. Olausson, Muxin Liu, Joshua B. Tenenbaum, and Jacob Andreas. LILO: Learning interpretable libraries by compressing and documenting code. In *International Conference on Learning Representations*, pp. 30399–30446, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/hash/819cebb05f993840e8a52d7564c5c282-Abstract-Conference.html.

Geoffrey E. Hinton, Peter Dayan, Brendan J. Frey, and Radford M. Neal. The "wake-sleep" algorithm for unsupervised neural networks. *Science*, 268(5214):1158–1161, 1995. URL https://www.cs.utoronto.ca/~hinton/absps/ws.htm.

Michael Hodel. Addressing the abstraction and reasoning corpus via procedural example generation. *arXiv preprint arXiv:2404.07353*, 2024. URL https://arxiv.org/abs/2404.07353.

Joey Hong, David Dohan, Rishabh Singh, Charles Sutton, and Manzil Zaheer. Latent programmer: Discrete latent codes for program synthesis. In *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pp. 4308–4318. PMLR, 2021. URL https://proceedings.mlr.press/v139/hong21a.html.

Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*, 2020. URL https://arxiv.org/abs/2001.08361.

Yoon Kim, Sam Wiseman, Andrew Miller, David Sontag, and Alexander Rush. Semi-amortized variational autoencoders. In *International Conference on Machine Learning*, pp. 2678–2687. PMLR, 2018.

Thomas Kipf, Yujia Li, Hanjun Dai, Vinicius Zambaldi, Alvaro Sanchez-Gonzalez, Edward Grefenstette, Pushmeet Kohli, and Peter Battaglia. CompILE: Compositional imitation learning and execution. In *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pp. 3418–3428. PMLR, 2019. URL https://proceedings.mlr.press/v97/kipf19a.html.

Brenden Lake and Marco Baroni. Generalization without systematicity: On the compositional skills of sequence-to-sequence recurrent networks. In *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pp. 2873–2882. PMLR, 2018. URL https://proceedings.mlr.press/v80/lake18a.html.

Wen-Ding Li, Keya Hu, Carter Larsen, Yuqing Wu, Simon Alford, Caleb Woo, Spencer M. Dunn, Hao Tang, Wei-Long Zheng, Yewen Pu, and Kevin Ellis. Combining induction and transduction for abstract reasoning. In *International Conference on Learning Representations*, 2025. URL https://openreview.net/pdf/a53e248e943d28f22b7a38f38bc90f74a5f0016b.pdf.

Matthew V. Macfarlane and Clément Bonnet. Searching latent program spaces. In *Advances in Neural Information Processing Systems*, volume 38, pp. 103463–103520. Curran Associates, Inc., 2025. doi: 10.52202/085713-3457. URL https://proceedings.neurips.cc/paper_files/paper/2025/hash/95738b4062a7fed00bb9db468475ae53-Abstract-Conference.html.

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. URL https://openreview.net/pdf/5d002a8ef4e8e50fea8316c33fce273be890022b.pdf.

Julien Pourcel, Cédric Colas, and Pierre-Yves Oudeyer. Self-improving language models for evolutionary program synthesis: A case study on ARC-AGI. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 49659–49688. PMLR, 2025. URL https://proceedings.mlr.press/v267/pourcel25a.html.

Nasim Rahaman, Muhammad Waleed Gondal, Shruti Joshi, Peter Gehler, Yoshua Bengio, Francesco Locatello, and Bernhard Schölkopf. Dynamic inference with neural interpreters. In *Advances in Neural Information Processing Systems*, volume 34, pp. 10985–10998. Curran Associates, Inc., 2021. URL https://proceedings.neurips.cc/paper/2021/hash/5b4e9aa703d0bfa11041debaa2d1b633-Abstract.html.

Jürgen Schmidhuber. PowerPlay: Training an increasingly general problem solver by continually searching for the simplest still unsolvable problem. *Frontiers in Psychology*, 4:313, 2013. doi: 10.3389/fpsyg.2013.00313. URL https://www.frontiersin.org/journals/psychology/articles/10.3389/fpsyg.2013.00313/full.

Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 1715–1725. Association for Computational Linguistics, 2016. doi: 10.18653/v1/P16-1162. URL https://aclanthology.org/P16-1162/.

Andrew Zhao, Yiran Wu, Tong Wu, Quentin Xu, Yang Yue, Matthieu Lin, Shenzhi Wang, Qingyun Wu, Zilong Zheng, and Gao Huang. Absolute zero: Reinforced self-play reasoning with zero data. In *Advances in Neural Information Processing Systems*, volume 38, pp. 105816–105879. Curran Associates, Inc., 2025. doi: 10.52202/085713-3534. URL https://proceedings.neurips.cc/paper_files/paper/2025/hash/9837dc00ff67d176373268ed48042d49-Abstract-Conference.html.

Ruijie Zheng, Ching-An Cheng, Hal Daumé, III, Furong Huang, and Andrey Kolobov. PRISE: LLM-style sequence compression for learning temporal action abstractions in control. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 61267–61286. PMLR, 2024. URL https://proceedings.mlr.press/v235/zheng24b.html.

AI USE STATEMENT

In this work, we used generative AI tools for writing code. We have not used generative AI tools for generating experimental results or data, and the remaining required disclosure tasks are not applicable to this work. Additionally, we used generative AI tools for brainstorming and improving the clarity and language of the manuscript. We have reviewed all AI-assisted work. AI-generated code was reviewed, tested, and verified for correctness by the authors before being used in any experiment. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

A BASELINES

A.1 LPN & NLI WITH PRIOR SEARCH

Given a specification $S = \{(x_i, y_i)\}_{i=1}^m$ and a query input x, both baselines sample n latent programs z and compute the negative log-likelihood (NLL) on the specification S for each program z: $\ell(z) = -\sum_{i=1}^m \log f_\theta(y_i|z, x_i)$. They then select the program that achieves the lowest NLL $z^* = \arg\min_z \ell(z)$ to execute on x.

A.2 LPN & NLI WITH GRADIENT SEARCH

Rather than fully relying on the inductor's initial guess, LPN generates a starting program z_0 and then performs gradient descent with respect to the NLL on the specification: $z_{t+1} = z_t - \gamma \nabla \ell(z_t)$. NLI explores a wider range of the program space by drawing n starting points from a Gaussian distribution centered at z_0: $z \sim \mathcal{N}(z_0, \sigma^2 I)$, and running gradient descent from each. After k steps, both baselines select the program with the lowest NLL to execute on x.

A.3 NLI-R WITH DISCRETE SEARCH

NLI-R with discrete search samples n program rollouts τ and selects the best one to execute on x. At each sampling step, a temperature κ is applied to the logits ℓ_t to encourage exploration over a wider range of the program space: $\tau_t \sim \text{Softmax}(\ell_t/\kappa)$. The interpreter then executes each sampled program on the specification inputs x_i, and those that correctly produce the target outputs y_i are collected into a pool of candidate programs. The final program is then selected from this pool by frequency, i.e., the most common program among the n rollouts.

B HYPERPARAMETERS

Table 2: Hyperparameters for NLI stage 1 training.

Hyperparameter	Comp-I	Cell-Auto	Comp-Reverse
Model Architecture			
Model Dimension (d_{model})	128	128	128
Number of Heads (n_{head})	8	8	8
Feed-Forward Dimension (d_{ff})	512	512	512
Encoder Layers	2	2	2
Decoder Layers	2	2	2
Positional Embedding	Sinusoidal	2D Learned	Sinusoidal
Program Generation			
Codebook Size	512	512	512
Program Length	4	4	4
Training			
Batch Size	512	512	512
Num Batches	100k	150k	300k
Optimizer	Adam	Adam	Adam
Learning Rate	1e-4	8e-4	2e-4
LR Schedule	cosine to 0	cosine to 0	const. to 80k, cosine to 5e-6 at 200k
Gradient Clip Norm	2.0	1.0	1.0
Gumbel-Softmax Sampling (Inductor)			
Use Gumbel	True	True	True
Start Temperature	8.0	8.0	1.0
End Temperature	0.1	0.1	0.5
Annealing Batches	100k	150k	150k
Annealing Strategy	Pointwise	Linear	Exponential
Straight-Through	False	False	False
Gumbel-Softmax Sampling (Interpreter)			
Use Gumbel	True	True	True
Start Temperature	2.0	2.0	1.0
End Temperature	0.1	0.1	0.5
Annealing Batches	100k	150k	150k
Decay Strategy	Pointwise	Linear	Exponential
Straight-Through	False	False	False
Regularization & Losses			
Encoder Loss Coefficient	1e-5	1e-5	1e-4

Table 3: Hyperparameters for NLI dreaming.

Hyperparameter	**Comp-I**	**Cell-Auto**	**Comp-Reverse**
	Primitive Extraction		
Stage-1 Tasks Solved for Extraction	2000	2000	2000
	Dream Generation		
Composition Method	Permutation BFS	Permutation BFS	Permutation BFS
Max Dream Program Length	4	4	6
Program Slots	4	4	6
Number of Dream Tasks	500k	500k	4M
Sampling Balance	By program length	By program length	By program length
	Encoder Retraining		
Trained Parameters	Encoder only	Encoder only	Encoder only
Extra Encoder Layers	0	0	4
Causal Head Layers	0	0	2
Loss	Cross-entropy	Cross-entropy	Teacher-forced cross-entropy
Optimizer	Adam	Adam	Adam
Learning Rate	1e-3	1e-3	1e-3
LR Schedule	Cosine to 0	Cosine to 0	Cosine to 0
Batch Size	512	512	512
Num Batches	5k	200k	200k
Gradient Clip Norm	1.0	1.0	1.0
Precision	fp32	fp32	bf16

Table 4: Model Hyperparameters for NLI-R stage 1 training.

Hyperparameter	Comp-I	Cell-Auto	Comp-Reverse
Model Architecture			
Model Dimension (d_{model})	128	128	128
Number of Heads (n_{head})	8	8	8
Feed-Forward Dimension (d_{ff})	512	512	512
Encoder Layers	2	2	2
Decoder Layers	2	2	2
Positional Embedding	Sinusoidal	2D Learned	Sinusoidal
Program Generation			
Program Vocabulary Size	512	512	512
Program Length (Training)	4	4	4
Training			
Batch Size	512	512	512
Num Batches	100k	150k	300k
Optimizer	Adam	Adam	Adam
Learning Rate	1e-3	1e-4	2e-4
LR Schedule	cosine to 0	cosine to 0	const. to 80k, cosine to 5e-6 at 200k
Gradient Clip Norm	1.0	1.0	10.0
Roll-out Gradient	BPTT	BPTT	Detached
Gumbel-Softmax Sampling (Inductor)			
Use Inductor Gumbel	True	True	True
Start Temperature	8.0	8.0	1.0
End Temperature	0.1	0.1	0.5
Annealing Batches	100k	150k	150k
Decay Strategy	Linear	Linear	Exponential
Straight-Through	False	False	False
Gumbel-Softmax Sampling (Interpreter)			
Use Interpreter Gumbel	True	True	True
Start Temperature	2.0	2.0	1.0
End Temperature	0.1	0.1	0.5
Annealing Batches	100k	150k	150k
Decay Strategy	Linear	Linear	Exponential
Straight-Through	False	False	True
Regularization & Losses			
Encoder Loss Coefficient	1e-5	1e-5	1e-4

Table 5: Hyperparameters for NLI-R dreaming.

Hyperparameter	Comp-I	Cell-Auto	Comp-Reverse
Primitive Extraction			
Stage-1 Tasks Solved for Extraction	2000	2000	2000
Dream Generation			
Composition Method	Permutation BFS	Permutation BFS	Permutation BFS
Max Dream Program Length	4	4	6
Program Slots	4	4	6
Number of Dream Tasks	500k	500k	4M
Sampling Balance	By program length	By program length	By program length
Encoder Retraining			
Trained Parameters	Encoder only	Encoder only	Encoder only
Extra Encoder Layers	0	0	4
Causal Head Layers	0	0	2
Loss	Teacher-forced CE	Teacher-forced CE	Teacher-forced CE
Optimizer	Adam	Adam	Adam
Learning Rate	1e-3	1e-3	1e-3
LR Schedule	Cosine to 0	Cosine to 0	Cosine to 0
Batch Size	512	512	512
Num Batches	20k	100k	200k
Gradient Clip Norm	1.0	1.0	1.0
Precision	bf32	bf32	bf16

Table 6: Model Hyperparameters for LPN training.

Hyperparameter	Comp-I	Cell-Auto	Comp-Reverse
Model Architecture			
Model Dimension (d_{model})	128	128	128
Number of Heads (n_{head})	8	8	8
Feed-Forward Dimension (d_{ff})	512	512	512
Latent Dimension	64	64	64
Encoder Layers	2	2	2
Decoder Layers	2	2	2
Positional Embedding	Sinusoidal	2D Learned	Sinusoidal
Training			
Batch Size	512	512	512
Num Batches	150k	150k	150k
Optimizer	Adam	Adam	Adam
Learning Rate	5e-4	5e-4	5e-4
LR Schedule	cosine to 0	cosine to 0	cosine to 0
Gradient Clip Norm	1.0	1.0	1.0
Regularization & Losses			
KL Loss Coefficient	1e-4	1e-4	1e-4

Table 7: Hyperparameters for NLI test-time search.

Hyperparameter	Comp-I	Cell-Auto	Comp-Reverse
Prior Search			
Number of Samples	2048	2048	2048
Gradient Search			
Number of Initializations	2048	2048	2048
Std for Initialization	15	15	7.5
Optimizer	Adam	Adam	Adam
Gradient Steps	100	100	100
Learning Rate	1e-1	1e-1	1e-2
Use Gumbel	True	True	True
Start Temperature	1.0	1.0	1.0
End Temperature	0.1	0.1	0.1
Annealing Batches	100	100	100
Decay Strategy	Linear	Linear	Linear
Straight-Through	False	False	False

Table 8: Hyperparameters for NLI-R test-time search.

Hyperparameter	Comp-I	Cell-Auto	Comp-Reverse
Discrete Search			
Number of Rollouts	2048	2048	2048
Temperature	5.0	5.0	5.0

Table 9: Hyperparameters for LPN test-time search.

Hyperparameter	Comp-I	Cell-Auto	Comp-Reverse
Prior Search			
Number of Samples	2048	2048	2048
Gradient Search			
Optimizer	-	-	-
Gradient Steps	100	100	100
Learning Rate	1e-1	1e-1	1e-1