Searching Latent Program Spaces

Clément Bonnet
clement.bonnet16@gmail.com

Matthew V Macfarlane
University of Amsterdam
m.v.macfarlane@uva.nl

Abstract

Program synthesis methods aim to automatically generate programs restricted to a language that can explain a given specification of input-output pairs. While purely symbolic approaches suffer from a combinatorial search space, recent methods leverage neural networks to learn distributions over program structures to narrow this search space significantly, enabling more efficient search. However, for challenging problems, it remains difficult to train models to perform program synthesis in one shot, making test-time search essential. Most neural methods lack structured search mechanisms during inference, relying instead on stochastic sampling or gradient updates, which can be inefficient. In this work, we propose the Latent Program Network (LPN), a general algorithm for program induction that learns a distribution over latent programs in a continuous space, enabling efficient search and test-time adaptation. We explore how to train these networks to optimize for test-time computation and demonstrate the use of gradient-based search both during training and at test time. We evaluate LPN on ARC-AGI, a program synthesis benchmark that evaluates performance by generalizing programs to new inputs rather than explaining the underlying specification. We show that LPN can generalize beyond its training distribution and adapt to unseen tasks by utilizing test-time computation, outperforming algorithms without test-time adaptation mechanisms.

1 Introduction

Program synthesis aims to automatically generate programs that satisfy a given specification, typically as input-output examples [Summers, 1977, Biermann, 1978]. Although symbolic approaches have shown success in limited domains [Gulwani, 2011, Albarghouthi et al., 2013, Osera and Zdancewic, 2015, Feser et al., 2015, Frankle et al., 2016], they fail to scale to modern challenges involving large search spaces and complex patterns like in ARC-AGI [Chollet, 2019, Chollet et al., 2024]. To handle the complexity and exponential search space of such difficult problems, neural approaches have emerged that aim to learn algorithms and programs from data [Graves, 2014, Kurach et al., 2015, Reed and De Freitas, 2015, Zaremba et al., 2016, Gaunt et al., 2016, Bunel et al., 2016, Bošnjak et al., 2017]. Such methods are required as they can significantly narrow down the programmatic search space by leveraging biases learned through training, enabling more efficient search. Large language models (LLMs) have emerged as a particularly strong architecture for program synthesis tasks, with language pre-training helping to narrow the search space before any further problem-specific fine-tuning is performed [Austin et al., 2021]. This can be particularly useful if generating problem-specific data is too expensive or limited. However, models trained on a specific program distribution will likely only generalize to problems close to this distribution and perform poorly on problems such as ARC-AGI, which are specifically designed to be out of distribution for LLMs [Gendron et al., 2023]. Such generative neural methods lack mechanisms for systematic search at test time, with models usually resorting to stochastic sampling [Chen et al., 2021] or heuristic search [Zhang et al., 2023]. Hottung et al. [2021b], Li et al. [2024a] and the ARC Prize 2024 leading team MindsAI explore fine-tuning a

Submitted to the ARC Prize 2024 competition [Chollet et al., 2024]. The code used in this research is open source and available at https://github.com/clement-bonnet/lpn.

model on a test task. However, such fine-tuning is computationally expensive and highly prone to overfitting. This requires creating synthetic datasets on the fly, making fine-tuning a less scalable approach.

To address these limitations, we introduce a general and scalable algorithm, Latent Program Network (LPN), to perform program induction learning [Sun et al., 2018]. LPN builds a mechanism for test-time adaptation directly into the neural architecture, without the need for parameter updates, and utilizes a training objective suitable to test-time search. To perform adaptation, LPN leverages a continuous latent space to model a wide range of potential programs, which a neural decoder can then use to execute that program on a specific input. Note that our decoder directly generates outputs pixel by pixel instead of writing Python code that would execute the task. Contrary to most existing works, we train the neural architecture from scratch, as opposed to building on top of large-scale models such as LLMs. Instead, our goal is to generalize purely through a test-time adaptation mechanism, with as few priors as possible. Our key innovation is the ability to search through a structured latent space during both training and inference, enabling efficient adaptation at test time. We leverage gradient-based optimization in this latent space to find the latent program that best explains the given specification. This latent program can then be executed on new inputs to generate their corresponding outputs. Since this space is a highly compressed representation of input-output pairs, we can perform gradient ascent in this space without encountering potential overfitting that parameter-based fine-tuning methods face and therefore we do not require synthetic expansion. Our training objective ensures that we learn a structured latent space that is smooth and performs well with gradient-based optimization, allowing for more efficient program discovery. First, to assess the benefits of our latent-search architecture, we evaluate it on a simple subset of ARC-type programs. Second, we benchmark on ARC-AGI, a difficult program synthesis problem with a public train and evaluation set and hidden test set. In this work, we specifically choose to not enhance our results by using additional synthetic datasets, human or LLM generated, as we believe it is in the spirit of the ARC-AGI competition to only use priors from the training set. Specifically, we only use `re-arc` [Hodel, 2024] to generate input-output pairs that follow the programs of the training set. By training only on train set problems, we ensure no possibility of data leakage from the evaluation dataset, which likely occurs in methods leveraging pre-trained LLMs or LLM-generated programs [Li et al., 2024a].

Our works' main contributions are:

1. We introduce Latent Program Network (LPN) which directly builds in test time adaption into the architecture by learning a latent space representing the space of possible programs, enabling test time adaption of an output decoder by moving in this latent space. We train this with a novel training objective for learning the latent space that prevents encoding the output directly into the latent space. Instead, we encode an input-output pair to the latent space but train this representation to decode the output of a different input-output pair, which prevents the latent space from representing the output space instead of the program space.

2. We demonstrate that gradient-based search on the given specification in the latent space significantly improves the performance of LPN at test time compared to performance without latent search.

3. We show that adding gradient-based latent search during training enables the latent space itself to be trained such that gradient optimization in the space works well, resulting in significant improvement in sample efficiency.

4. We do not make use of any pre-trained models or LLM / human-generated synthetic data when evaluating our work in the ARC domain, aside from generating input-output pairs using `re-arc` [Hodel, 2024] based on the training set. This makes our method highly scalable and could be quickly applied to a different domain without requiring a domain-specific language, synthetic data, or pre-trained model. LPN can be applied to any problem for which a large enough number of input-output pairs from a given set of programs is available. In our current setting, we do not even apply enough compute during training to observe convergence indicating that improved results on ARC-AGI could be found simply by scaling training compute resources/parameter counts.

5. Our work directly refutes recent claims [Li et al., 2024b] that vision transformer architectures [Dosovitskiy, 2020] struggle to solve individual arc problems. We show this is not a

bottleneck in making progress on ARC-AGI using purely neural approaches that predict outputs pixel by pixel.

2 Related Work

Early approaches to program synthesis focused on fully symbolic methods for LISP program construction, such as Summers [1977], Biermann [1978], which aimed to infer LISP programs from examples using symbolic reasoning. Such approaches were extended to domains such as string programming [Gulwani, 2011] with applications to Microsoft Excel spreadsheets. However, these methods were limited in their scalability and adaptability to more complex domains, often relying on enumerative search with domain-specific heuristics to reduce the search space [Albarghouthi et al., 2013, Osera and Zdancewic, 2015]. With the advent and successes of Deep Learning [LeCun et al., 2015], there has been a trend to either train fully end-to-end neural networks for program synthesis or combine neural networks with symbolic methods. Neural programmers-interpreters [Reed and De Freitas, 2015] proposed a full end-to-end recurrent neural network approach. They found that such architecture could learn 21 different programs using the same core inference module, inferring programs to execute at test time. Our model extends such approaches by learning a latent program representation instead of learning only a fixed discrete set of programs. We also do not require program IDs during training, which are inferred by our encoder during training. A popular neuro-symbolic approach called DreamCoder [Ellis et al., 2021] tackles the limitations of a domain-specific language (DSL) by building up a library of useful programs that can be reused progressively making search more efficient by being able to search over higher-level programs. LPN works by predicting outputs from specifications via the latent space, and is therefore distinct from methods leveraging DSLs to synthesize programs that map inputs to outputs. By directly predicting outputs, LPN does not restrict the types of programs we can execute, which is a consequence of using a DSL.

Recently, much of the work in neural program synthesis has shifted toward leveraging pre-trained large language models (LLM). These approaches are significantly different from ours in terms of what priors come into the model since these LLMs are pre-trained on data that likely have non-zero mutual information with program synthesis problems, allowing them to tighten any generalization gap. This branch of approaches aims to improve generalization by reducing the required gap between train and test, by expanding the dataset directly or via a pre-trained model. CodeIt [Butt et al., 2024] iteratively samples programs from a pre-trained CodeT5 [Wang et al., 2023] model, performs hindsight relabeling, and then fine-tunes the model, helping tackle the sparsity of rewards in program synthesis. Kim et al. [2024] create ARCLE an environment consisting of fixed actions according to core knowledge priors in which Reinforcement Learning can be performed. This is not as scalable and relies on human-extracted primitives for training and restricts the space of possible programs that can be executed to solve ARC tasks. Gendron et al. [2023] investigate the zero-shot performance of LLMs on various reasoning tasks, including ARC-AGI, showing limited but non-zero performance without fine-tuning, which suggests some degree of generalization. Mitchell et al. [2023] report comparable findings on Concept-ARC [Moskvichev et al., 2023]. Li et al. [2024a] investigate distinct induction and transduction methods using LLM-based synthetic datasets for training. However, following a trend with ARC-based research, they evaluate on the public evaluation dataset instead of the hidden test dataset. Since they leverage synthetic LLM-based programs and human-crafted prompts, there is a strong possibility for data leakage. LPN is classified as an induction-based approach since it tries to find a latent program that explains each of the input-output pairs in the specification to then predict the test input.

A training approach that is most similar to LPN is Kolev et al. [2020], one of the few attempts to learn ARC program synthesis end-to-end using a neural approach without a DSL or pre-trained language models. They use an autoencoder to learn grid embeddings, embedding the entire instruction set in one step to predict the output grid directly using a transformer [Vaswani et al., 2017] architecture, also making use of spectral norm regularization [Yoshida and Miyato, 2017]. There are still significant differences in architecture as they do not learn a latent space of programs that can then be used for search, instead opting to directly predict the output from the specification. Such approaches do not leverage test-time adaptation, likely struggle to generalize across varying specification sizes, and require careful architectures to be permutation invariant neither of which are problems present in the LPN architecture.

While pre-trained models may solve some problems either in one shot or with minimal sampling, certain solutions are highly unlikely to be generated by the model, and no amount of search or sampling will uncover them within a reasonable time. Fine-tuning [Hottung et al., 2021b, Hübotter et al., 2024, Li et al., 2024a] for a new problem is one way to break out of this, but it is very expensive and inefficient to perform on large models. A branch of work that has the potential to overcome some of these challenges is conditioning models on a latent space, where moving through the latent space impacts the distribution of outputs predicted by the decoder. Most of the existing work in this field with the strongest similarities to our work does not investigate the task of program synthesis. Gómez-Bombarelli et al. [2018] investigate latent space optimization for molecule generation, and on the Traveling Salesman Problem Hottung et al. [2021a] show that learning a latent space can be more sample-efficient than simply sampling from a standard transformer model. Compass [Chalumeau et al., 2023] leverage a latent space in online Reinforcement Learning applied to combinatorial optimization, using the Poppy objective [Grinsztajn et al., 2022] to encourage diversity of the latent space. This idea has also been explored in black-box optimization [Yu et al., 2024] to increase the diversity of input design modes, levering an energy-based model to search the latent space. In symbolic mathematics, latent space optimization has been used to find an equation that best explains the data, balancing a trade-off between low complexity and high accuracy [Meidani et al., 2023]. In program synthesis, similar concepts have been explored, such as in Hong et al. [2021], which focuses on discrete representations of programs for string transformation tasks, by training an auto-encoder [Hinton and Salakhutdinov, 2006] on the full specification. However, they do not leverage gradient search in the latent space at test time but instead use beam search [Furcy and Koenig, 2005] and simply train the model to find a program in the DSL that can explain the specification instead of generalizing to a new input. Policy sketches [Murali et al., 2017, Nye et al., 2019] can also be seen as learning a discrete latent space which is then more compressed and therefore potentially more efficient to search. However, while discrete spaces can be useful for interpretability and potentially compositional generalization [Shi et al., 2023], they still suffer from the drawback that they are harder to search relative to smooth continuous latent spaces that can leverage a gradient signal [Bartunov et al., 2020]. LEAPS [Trivedi et al., 2021] learn a latent space over programs for the Karel domain [Pattis, 1994], and then use gradient-free optimization to search this space for high-performing programs, however, Carvalho et al. [2024] show this method is outperformed by hill climbing in the programmatic space.

3 Background

Program Synthesis aims to generate deterministic programs in a target language, such that outputs generated from inputs are consistent with the given specification. Typically, the problem space Y consists of programs formulated within a domain-specific language (DSL). Each task is defined by a specification set X, where each specification, $X_m \in X$, is described by a set of input/output (I/O) examples:

$$X_m = \{(x_1^m, y_1^m), \ldots, (x_n^m, y_n^m)\} \tag{1}$$

A program $f \in Y$ is considered to solve the task associated with X_m if it satisfies:

$$\forall j \in [1, n], \quad f(x_j^m) = y_j^m \tag{2}$$

This definition requires that the program exactly replicates the output for each input in its specification. We denote F_m to represent the true function that generates the input-output pairs.

Program Synthesis Generalization. In this work, we consider the problem where we are given a specification set of input-output examples generated by a program F_m (not necessarily constrained to a DSL), along with an additional input x_{n+1}^m:

$$P_m = \{(x_1^m, y_1^m), \ldots, (x_n^m, y_n^m), x_{n+1}^m\}. \tag{3}$$

The goal is not to be able to explain the specification, but to apply the program shown in the specification to a new input example x_{n+1}^m, demonstrating generalization. This can be done via induction

or transduction [Li et al., 2024a]. If we do not limit the Kolmogorov complexity of programs [Kolmogorov, 1965, Solomonoff, 1964], we can find an explanation for any given specification whether or not it corresponds to the true program that underlies the specification. Evaluating generalization performance tests not only the ability to explain a specification but also to successfully infer the underlying program in its generality and apply it to a new input. This problem bears a resemblance to few-shot learning with the difference of having only one task. A recent problem posed in the Artificial Intelligence research community that falls under this formulation is the ARC dataset [Chollet, 2019] that contains programs on 2D grids of 30x30 cells that can take any 10 colors.

Variational Auto-Encoders (VAEs) [Kingma, 2013] provide a framework for approximate Bayesian inference. Given i.i.d. samples x from a dataset, the goal is to estimate the posterior $p(z|x)$, where z represents a latent variable. Since directly computing this posterior is generally intractable, the evidence lower bound (ELBO) is introduced to approximate it. In this approach, a parameterized encoder $q_\phi(z|x)$ is used to approximate the true posterior distribution.

The ELBO can be formulated as follows:

$$\log p(x) \geq \mathbb{E}_{z\sim q_\phi(z|x)}\left[\log p_\theta(x|z)\right] - D_{KL}(q_\phi(z|x)\|p(z)), \tag{4}$$

where the first term captures the likelihood of reconstructing x from the latent variable z, and the second term is a Kullback-Leibler divergence that regularizes $q_\phi(z|x)$ to be close to the prior distribution $p(z)$.

4 Latent Program Network

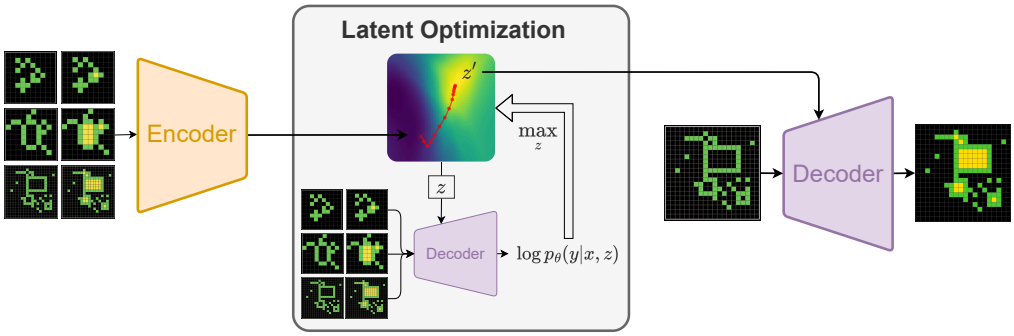


Figure 1: Inference of the Latent Program Network (LPN) model. (Left): the encoder maps I/O pairs to a latent space of encoded programs. (Middle): the latent program is refined during an optimization process to best explain the given I/O pairs. (Right): the decoder executes the latent program to generate the desired output to a newly given input. The latent optimization figure in the middle comes from the experiment described in the appendix, figure 16.

We propose the Latent Program Network (LPN), an algorithm that trains a neural network end to end to take a specification of input-output pairs and generate the output of a newly given input. With a focus on abstraction, search, and synthesis, LPN is designed with the ability to perform test-time adaptation explicitly built into the architecture. LPN is composed of three components (see figure 1): (1) a neural network encoder that takes in a specification X_m and outputs an abstracted latent program, (2) an optimization process that refines the latent to best explain the data, (3) a decoder neural network that executes the latent program to generate an output.

Prior work has focused on directly training models to maximize the likelihood of decoding the correct output given a specification [Kolev et al., 2020]. However, we diverge from such transduction-based methods, which condition on all input-output pairs in the specification to predict an output, as these approaches face challenges when scaling to specifications of different sizes and generalizing to unseen tasks. They also offer no inherent ability to adapt at test time. Instead, by explicitly factorizing our system into abstraction generation, latent program synthesis, and output prediction, we aim to build an induction machine that can utilize test-time computation to adapt to unseen instances.

4.1 Latent Program Inference

The encoder and decoder play similar roles as in a VAE, while the latent optimization process attempts to solve a dynamic optimization problem. Unlike VAEs, we can utilize our encoder at test time as a starting point for latent program search, which we find crucial for performance.

Encoder: The probabilistic encoder is trained to approximate the Bayesian posterior over programs. Specifically, it maps an input-output pair (x, y) to a distribution in the latent space $q_\phi(z|x, y)$, representing possible programs that could explain the given input-output mapping. Using a variational approach is important because, for any given input-output pair, there exists a broad range of possible programs that map the input to the output, even when restricting to e.g. programs of low Kolmogorov complexity [Solomonoff, 1964, Kolmogorov, 1965]. Intuitively, the encoder is trained to learn an abstract representation of programs in a continuous latent space, by implicitly encoding input-output pair examples. In practice, we use a multivariate normal distribution whose mean μ and diagonal covariance Σ parameters are inferred by the encoder. To take advantage of hardware parallelization, the encoder can process all the I/O pairs in a given specification in parallel. It should be noted that LPN is permutation invariant to the specification order by encoding each pair independently, contrary to a naive sequence model over the concatenation of I/O pairs.

Decoder: The probabilistic decoder is responsible for mapping a latent program and an input to the space of outputs, directly predicting the output pixel by pixel instead of via a DSL. It models the distribution of possible outputs y given an input x and a latent z. Note that even if the underlying I/O mappings are deterministic, we still use a probabilistic decoding framework $p_\theta(y|x, z)$ to be compatible with maximum likelihood learning. Figure 2 shows the decoder generating different outputs by keeping the input fixed but varying the latent program, which in this figure represents a specific grid pattern to reproduce. In a real task, the aim of this encoder-decoder system is to learn a compressed representation of the space of possible programs we care about (e.g. in the case of ARC-AGI, this would correspond to programs that use the Core Knowledge priors [Chollet, 2019]).

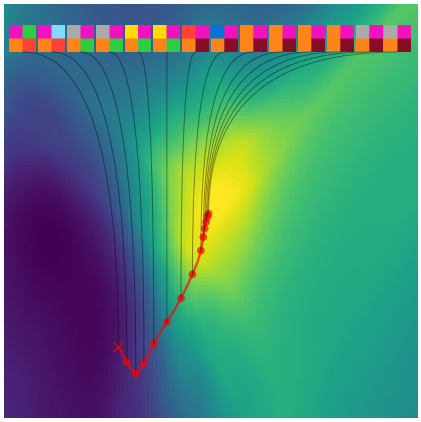


Figure 2: Conditioning the decoder on different points of the latent space leads to different outputs being generated. Experiment detailed in figure 16.

Latent Optimization: The encoder is trained to approximate the posterior over programs and may not encode the right abstraction given an I/O pair. Especially if the task is very novel, the encoder may fail at producing the right latent program, which, fed to the decoder, would generate the wrong output. Therefore, we include a middle stage of latent optimization where, starting from the encoder's prediction z, we search for a better latent program z', one that would better explain the observed data according to the decoder p_θ. The search process is generally denoted $z' = f(p_\theta, z, x, y)$ and can be implemented in several ways (c.f. section 4.2). Analogous to system 1/system 2 thinking [Kahneman, 2011], we can think of the encoder generating an intuitive first guess as to what the observed program may be (system 1), and the latent optimization process executing a search for hypotheses that would better explain the observations (system 2).

$$\begin{array}{lll} \text{Encoder} & \text{Latent Optimization} & \text{Decoder} \\ z \sim q_\phi(z|x, y) & z' = f(p_\theta, z, x, y) & \hat{y} \sim p_\theta(y|x, z') \end{array} \tag{5}$$

4.2 Search Methods for Latent Optimization

Given n input-output pairs $\{(x_i, y_i)\}_{i=1\ldots n}$, the search process $z' = f(p_\theta, z, x, y)$ attempts to find a z' that satisfies:

$$z' \in \arg\max_z \sum_{i=1}^{n} \log p_\theta(y_i|x_i, z) \tag{6}$$

This means we search for the latent that would most likely make the decoder generate the right outputs given the corresponding inputs. By finding a latent that can explain all the input-output pairs, the latent solution to the optimization problem is more likely to generalize to a new input-output pair. We describe here two instantiations of the search process, namely a *random search* and a *gradient ascent* algorithm, both acting in the latent space of programs. We leave for future work the exploration of other search methods like evolutionary strategies [Hansen and Ostermeier, 2001, Chalumeau et al., 2023] that could better trade-off exploration and exploitation of the latent space.

Random Search An initial version of the latent search process is to sample some latents from either the prior distribution $p(z)$ or around the approximate Bayesian posterior $q_\phi(z|x_i, y_i)$ and select the latent that gives the highest log likelihood of the input-output pairs according to the decoder.

$$\forall k \in [1, K],\ z_k \sim p(z) \qquad z' \in \arg\max_{z_k} \sum_{i=1}^{n} \log p_\theta(y_i|x_i, z_k) \tag{7}$$

Random search asymptotically converges to the true maximum likelihood latent (equation 6) and can prove useful when the function to optimize (here, the decoder log-likelihood) is not differentiable or smooth. However, the efficiency of random search decreases exponentially with the dimension of the latent space, which makes it impractical for most applications.

Algorithm 1 LPN Test-Time Inference with Gradient Ascent Latent Optimization

Input: n input-output pairs (x_i, y_i), a test input x_{n+1}, the number of gradient steps K
for $i = 1, \dots, n$ **do** ▷ Can be done in parallel
  Sample $z_i \sim q_\phi(z|x_i, y_i)$
end for
Initialize latent $z' = \frac{1}{n}\sum_{i=1}^{n} z_i$
for $k = 1, \dots, K$ **do** ▷ Perform gradient ascent
  $z' = z' + \alpha \cdot \nabla_z \sum_{i=1}^{n} \log p_\theta(y_i|x_i, z)|_{z=z'}$
end for
Generate output: $y_{n+1} \sim p_\theta(y|x_{n+1}, z')$

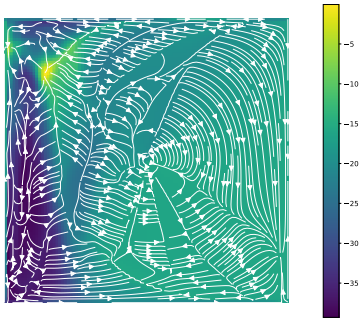


Figure 3: Gradient field of the decoder log-likelihood $f(z) = \log p_\theta(y|x, z)$

Gradient Ascent Since the decoder is a differentiable neural network, its log-likelihood $\log p_\theta(y|x, z)$ is also differentiable with respect to z and one can use first-order methods like gradient-ascent to efficiently search through the latent space for a solution to the latent optimization problem (equation 6). Figure 3 shows the gradient field and the landscape of the decoder log-likelihood of a 2D latent space and displays how gradient-ascent can be performed to find a local optimum in the latent space. This visualization highlights that only a small portion of the latent space can explain all the input-output pairs, corresponding to high decoding likelihood. Notably, poor initialization can lead the search to converge to different local minima, highlighting the importance of amortized inference from the encoder.

$$z'_0 = \frac{1}{n}\sum_{i=1}^{n} z_i \qquad \forall k \in [1, K],\ z'_k = z'_{k-1} + \alpha \cdot \nabla_z \sum_{i=1}^{n} \log p_\theta(y_i|x_i, z)|_{z=z'_{k-1}} \tag{8}$$

The series $(z_k)_{k\in[1,K]}$ should exhibit increasing decoding likelihood if the step size α is small enough. In practice, we generate the output with the best latent found during the gradient ascent algorithm, which may not always be the one that is obtained after taking the last gradient step (z_K).

4.3 Training

To train the LPN system end-to-end, we assume to have a dataset of tasks, where a task is defined as n input-output pairs (x_i, y_i) generated by the same program. To simulate the test conditions of predicting a new input from a given specification, we design the training procedure to reconstruct

each of the outputs y_i from their inputs x_i and all the $n-1$ other pairs $(x_j, y_j)_{j \neq i}$. We emphasize that we do not use the specific pair (x_i, y_i) to reconstruct y_i, which would otherwise lead to the encoder directly compressing the output y_i as a shortcut without learning program-related abstractions.

When reconstructing output y_i, we first sample latents z_j from the encoder $q_\phi(z|x_j, y_j)$ for all $j \neq i$. We then aggregate them by computing their mean $\frac{1}{n-1} \sum_{j \neq i} z_j$, then we perform latent optimization using e.g. gradient ascent to obtain z'_i. Finally, we compute the negative log-likelihood of the right output y_i using its corresponding input x_i and the refined latent z'_i. In practice, we compute the cross-entropy loss of the decoder logits $p_\theta(\hat{y}_i|x_i, z'_i)$ and the labels y_i, which derives from maximizing the likelihood of a categorical distribution. The full training pipeline is detailed in algorithm 2.

Specifically, we compute the reconstruction loss $\mathcal{L}_{\text{rec}}$ and the KL loss $\mathcal{L}_{\text{KL}}$ between the approximate posterior and the prior:

$$\mathcal{L}_{\text{rec}}(\phi, \theta) = \sum_{i=1}^{n} -\log p_\theta(y_i|x_i, z'_i) \quad (9) \qquad \mathcal{L}_{\text{KL}}(\phi) = \sum_{i=1}^{n} D_{\text{KL}}\left(q_\phi(z|x_i, y_i) \parallel \mathcal{N}(0, I)\right) \quad (10)$$

The dependence of the reconstruction loss $\mathcal{L}_{\text{rec}}(\phi, \theta)$ in ϕ arises from using the reparameterization trick [Kingma, 2013] when sampling each latent z_j. Indeed, we first sample a normal random vector $\epsilon_j \sim \mathcal{N}(0, I)$, then we infer the mean μ_j and diagonal covariance Σ_j using the encoder and recompose the latent $z_j = \mu_j + \epsilon_j \cdot \Sigma_j$. Then, z'_i is used by the decoder to reconstruct the output. Note that we can decide whether to let the decoder gradient flow through the latent update. Indeed, it is more computationally efficient to stop the gradient through the update, by changing line 10 of algorithm 2 with $z'_i = z'_i + \alpha \cdot \overline{g'_i}$, where $g'_i = \nabla_z \sum_{j \neq i} \log p_\theta(y_j|x_j, z)|_{z=z'_i}$, with $\overline{x}$ notating a stop-gradient on x.

We denote β the weighting factor that balances the reconstruction and KL terms [Burgess et al., 2018], which gives the combined training objective:

$$\mathcal{L}_{\text{total}}(\phi, \theta) = \mathcal{L}_{\text{rec}}(\phi, \theta) + \beta \mathcal{L}_{\text{KL}}(\phi) \quad (11)$$

Algorithm 2 LPN Training with Gradient Ascent Latent Optimization

1: **Input:** encoder parameters ϕ, decoder parameters θ
2: **for** $t = 1, \ldots,$ num_training_steps **do**
3: Sample n input-output pairs (x_i, y_i) from the same program
4: **for** $i = 1, \ldots, n$ **do** ▷ Can be done in parallel
5: Sample $z_i \sim q_\phi(z|x_i, y_i)$ ▷ Using the reparameterization trick
6: **end for**
7: **for** $i = 1, \ldots, n$ **do** ▷ Can be done in parallel
8: $z'_i = \frac{1}{n-1} \sum_{\substack{j=1 \\ j \neq i}}^{n} z_j$
9: **for** $k = 1 \ldots K$ **do** ▷ Perform gradient ascent in the latent space
10: $z'_i = z'_i + \alpha \cdot \nabla_z \sum_{\substack{j=1 \\ j \neq i}}^{n} \log p_\theta(y_j|x_j, z)|_{z=z'_i}$ ▷ Optional stop-gradient on the update
11: **end for**
12: $\mathcal{L}_i = -\log p_\theta(y_i|x_i, z'_i) + \beta \cdot D_{\text{KL}}(q_\phi(z|x_i, y_i) \parallel \mathcal{N}(0, \text{I}))$
13: **end for**
14: $\mathcal{L} = \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}_i$ ▷ Total loss for all pairs
15: Update ϕ and θ via gradient descent on $\mathcal{L}$
16: **end for**

This training procedure offers some freedom on the latent optimization, i.e. how to compute z'_i from z_i. Training with gradient ascent latent optimization (as detailed in algorithm 2) incurs a significant compute overhead due to the costly latent gradient computation through the decoder. Therefore, although we may use a high compute budget at test-time, we propose to use a small number of gradient ascent steps during training, ranging from 0 to 5 steps. Specifically, we call *mean* training when we train without latent optimization (i.e. 0 steps of gradient ascent), since in this case we use the mean latent to make a prediction.

5 ARC-AGI Experiments

In this work, we consider the ARC-AGI 2024 challenge [Chollet et al., 2024] as our testing domain for the proposed method. It is a challenging program synthesis dataset that encompasses a wide variety of unique tasks. Because very little training data is available (400 training tasks), the benchmark is resistant to memorization and rather tests for adaptation and out-of-distribution generalization. As justified in Chollet [2019], it is a benchmark that tests developer-aware generalization, which means one possesses limited information regarding the test tasks. Indeed, developers attempting to solve ARC-AGI cannot see the private test set but are given a set of Core Knowledge priors upon which all tasks are built. Such priors include objectness, goal-directedness (agency), simple arithmetic (numbers and counting), and basic geometry and topology.

5.1 LPN Architecture

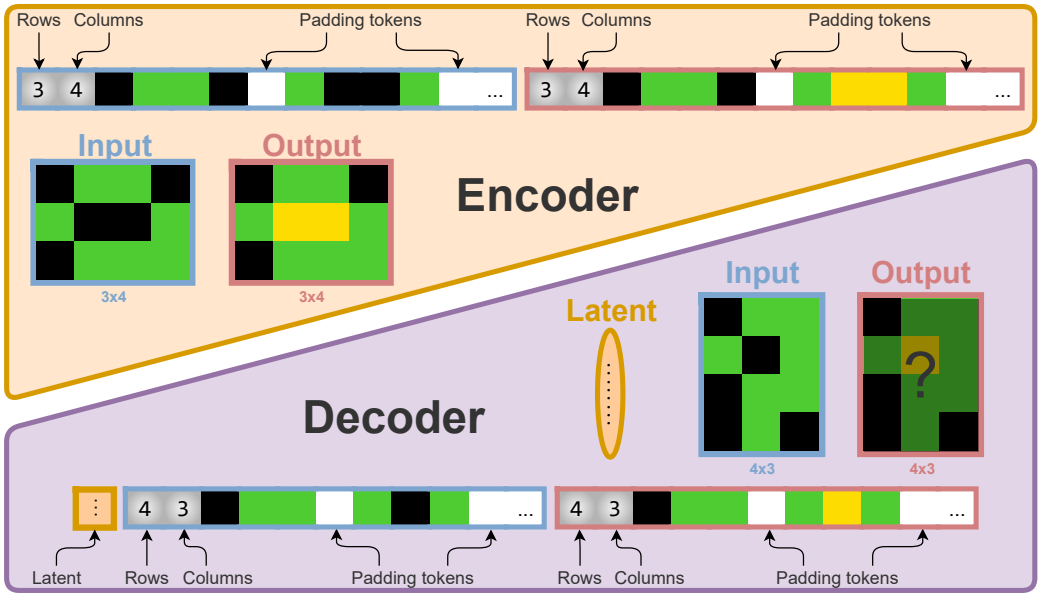


Figure 4: LPN architecture for ARC-AGI. Both the encoder and the decoder are small transformers that take in flattened padded grids as inputs. The actual number of rows and columns are prefixed to each sequence.

In all our experiments, programs are defined in the input-output space of ARC-AGI, i.e. 2D grids whose cells can take 10 different values and shapes are (n, m) with $n, m \in [1, 30]$. To train a high-performing latent space on such input-output pairs, it is critical to design an encoder that can process both input and output grids to infer a distribution over possible programs and a decoder that can execute a large number of programs on input grids.

We implement both the encoder and decoder as small transformers [Vaswani et al., 2017] specifically designed for this benchmark, in contrast to the more general large language models (LLMs) typically used [Wang et al., 2023]. By foregoing the transfer learning benefits of large pre-trained models, we aim to investigate the effectiveness of learning programs in this narrow domain and evaluate the potential of test-time latent search. The code used in this research is open source and available at `https://github.com/clement-bonnet/lpn`. Our codebase is implemented in JAX [Bradbury et al., 2018] and uses neural network building blocks from the Flax library [Heek et al., 2024].

We model the input and output images as 2D grids that we pad and flatten in a raster-scan fashion to form sequences of pixel values each of size $30 * 30 = 900$ (see figure 4). We prefix each grid sequence with shape information, namely 2 extra values for the number of rows and columns, leading to sequences of 902 values.

Encoder The encoder takes both the input and output grids from a given pair and returns a distribution of inferred program latents that underlie the task. More specifically, it returns the mean and the diagonal covariance of a multivariate normal distribution from which one can sample. Each grid sequence contains 902 values, and we add an extra CLS token for the output embedding. Therefore, the total sequence length of the encoder transformer is 1805. To process such sequences, we implement the encoder as a standard transformer [Vaswani et al., 2017] with pre-layer normalization [Baevski and Auli, 2019, Xiong et al., 2020] and multi-head attention. To account for spatial information, we factorize positional embeddings to be of the following form: $pe(i, j, c) = pr(i) + pc(j) + emb(c)$ with $i, j \in [1, 30]$, $c \in \{0, 1\}$ the channel index (0 for input and 1 for output), pr are the row embeddings, pc are the column embeddings. All embeddings are in $\mathbb{R}^H$ with H being the embedding size of the transformer. All 1800 color values between 0 and 9, the 4 shape values between 1 and 30, and the cls token are separately embedded into $\mathbb{R}^H$ using look-up tables. We mask the padded tokens given by the shape values and feed the sequence to multiple transformer blocks (more details on the architecture hyper-parameters can be found in the appendix section A). The attention mask is non-causal as all non-padded tokens can attend to all other non-padded tokens for the encoding computation. The CLS embedding then goes through a layer-norm and two parallel dense layers to output the mean and diagonal log-covariance of the multivariate normal distribution over latents. Sampled program latents are of dimension d which may differ from H.

Decoder The decoder takes an input grid and a latent program and auto-regressively generates an output grid. The decoder is designed similarly to the encoder with a few differences. First, it prefixes the flattened sequence with (a projection of) the latent embedding. Then, because the decoder auto-regressively generates the output, its attention mask is made causal on the second half of the sequence, which is the output grid. The attention mask on the output sequence is also a function of the predicted output shapes, dynamically masking padding tokens. The sequence embeddings corresponding to the output are then extracted and projected to either shape logits for the first two embeddings, or grid logits for the 900 other output grid embeddings. Note that an output token embedding maps to logits for the next token in a raster-scan fashion. However, due to padding at each line, we map the last embedding of each row to the first token of the next row instead.

5.2 Validating the Decoder

Training deep networks from scratch to solve ARC-like tasks has been challenging [Li et al., 2024b]. If it is true that such networks struggle even to learn to execute single programs, then this would represent a significant bottleneck to models training from scratch on a broad distribution of programs. Therefore, before training LPN end-to-end, we conclusively show that our decoder architecture does not suffer from such a bottleneck, and is able to learn individual programs.

We take 5 of the 400 tasks from the ARC-AGI training set, and for each of these tasks, we train a small LPN architecture of 800k parameters (except for the last task which required a bigger model with 8.7M parameters) on the corresponding task generator from `re-arc` [Hodel, 2024]. Specifically, we select the first five tasks from the `arc-agi_training_challenges` json file (`007bbfb7`, `00d62c1b`, `017c7c7b`, `025d127b`, `045e512c`) shown in figure 5.

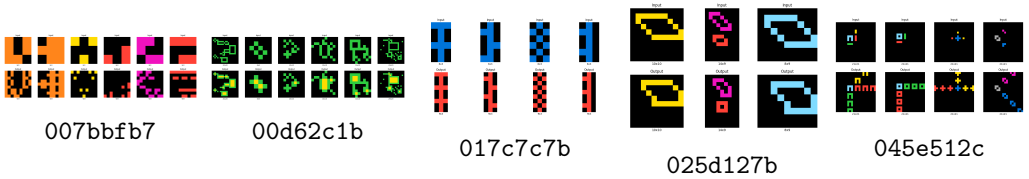
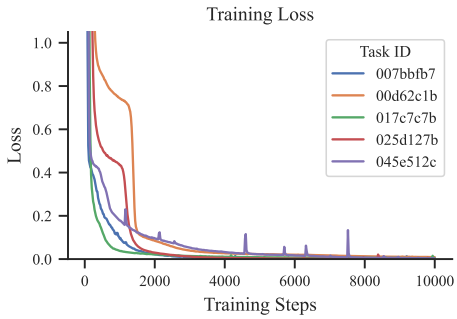


Figure 5: Overfit training on the first 5 ARC-AGI training tasks. The captions correspond to task IDs. For each task, the top row contains the input grids, and the bottom row the output grids. Each task consists of observing all pairs but the first and inferring the output of the leftmost pair given its input. Each curve corresponds to a separate training run.

We evaluate both the distribution of `re-arc` generated tasks on which it is trained and on the true task from the ARC-AGI training set in figure 6. We show that for each task, the small LPN decoder-only model successfully learns individual programs and manages to solve the corresponding ARC-AGI task. Therefore, our model outperforms previously reported results in Li et al. [2024b], and concludes

that our architecture does not suffer from a decoder bottleneck. Note that the encoder is not helpful in this experiment since the task is always the same. Our later results on ARC-AGI section 5.6 take this a step further and show that we can learn a single transformer architecture capable of executing over 180 programs in the ARC training dataset.



(a) Training loss

Task	Pixel Accuracy	
	`re-arc`	ARC-AGI
`007bbfb7`	100	100
`00d62c1b`	96.5	100
`017c7c7b`	99.7	100
`025d127b`	100	100
`045e512c`	98.3	100

(b) Performance after 10k steps of training

Figure 6: Training loss and performance of LPN training on 5 of the `re-arc` distributions. For each task, only samples from the `re-arc` generators are used for training. The corresponding ARC-AGI tasks are never seen during training.

5.3 Pattern Task

The ARC-AGI challenge contains many diverse programs leveraging different knowledge priors. Injecting these priors into LPN by training the model to master ARC-like tasks requires significant compute resources when training from scratch, without an LLM-based initialization. Therefore, to investigate the training dynamics and properties of LPN before such a large-scale training, we develop a simpler task called *Pattern* task (see figure 7) within the same domain, but using a narrow distribution of pattern-like programs. This specific task always generates fully-black 10x10 inputs with a single blue pixel at a random location that defines where the output pastes a 4x4 pattern sampled from a uniform distribution. The pattern is program-specific, which means it is the same across different pairs but it varies from one specification to another. This task enables us to demonstrate how deep learning methods may still make errors on such tasks without test-time computation. We later extend this task to study an out-of-distribution setting in section 5.5.

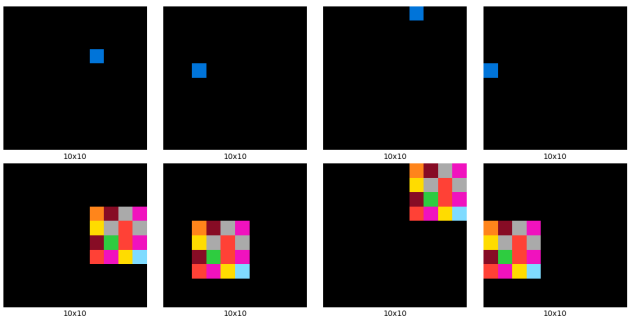


Figure 7: Example of input (top row) and output (bottom row) pairs of a specification sampled from the *Pattern* task. Each sample is a batch of 4 pairs that share the same pattern.

We compare a variety of training and inference methods in Table 1. We train 1M parameter models with each method for 20k steps with a batch size of 128. Then, we evaluate each training mode with different choices for latent optimization. Note that for all inference methods, we still only evaluate performance with a budget of 1, any inference budget only happens in the latent space before making a single output prediction. We repeat each experiment 3 times with different random seeds and report the mean performance and standard variation in parentheses.

Training	Inference					
	Mean	GA 1	GA 5	GA 20	GA 100	RS 250
Mean	3.2 (2.7)	3.6 (3.0)	18.8 (14.4)	52.5 (25.0)	67.5 (20.0)	3.2 (2.7)
GA 1	8.6 (4.4)	**44.6** (10.9)	**85.4** (7.6)	**98.4** (1.4)	**99.5** (0.5)	10.2 (5.3)
GA 1 (g)	0.6 (0.1)	13.7 (3.0)	60.2 (7.5)	88.9 (6.0)	94.1 (3.8)	0.7 (0.2)
GA 5	0.0 (0.0)	0.4 (0.3)	31.9 (11.2)	88.5 (11.9)	98.1 (2.1)	0.5 (0.4)
GA 5 (g)	0.0 (0.0)	0.0 (0.0)	9.9 (2.9)	87.1 (6.0)	95.1 (3.3)	0.0 (0.0)
RS 5	6.1 (4.4)	8.2 (6.5)	27.7 (21.6)	56.3 (27.5)	72.2 (21.2)	6.1 (4.4)
RS 25	**10.8** (8.0)	13.3 (10.1)	39.9 (21.4)	72.3 (18.5)	87.9 (9.2)	**10.8** (8.0)

Table 1: Ablation of different LPN training and inference methods on the *Pattern* task. For each training method, training was done for 20k steps with 3 different seeds, and performance is aggregated over the 3 runs with the standard deviation in parentheses. Methods with a (g) indicate that the gradient over parameters flows through the latent gradient ascent (analog to meta-learning). Other methods stop the parameter gradient during latent optimization. Two latent optimization methods are compared. GA [N] stands for gradient ascent with N steps, RS [X] means random search with a budget of X samples.

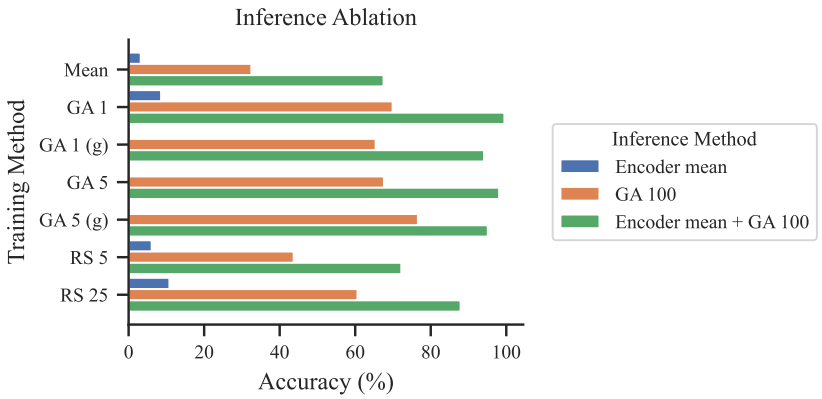


Figure 8: Ablation on the initialization of latent optimization and the role of the encoder. *Encoder mean* signifies that latent optimization is initialized using the encoder mean latents. GA 100 stands for 100 steps of gradient ascent during latent optimization. The results demonstrate the importance of using the encoder to find a good starting point before doing latent optimization.

The first result demonstrated on this relatively simple benchmark is that inference using mean latents (no latent optimization) performs poorly across all training methods. This shows that inferring the correct latent in one shot is challenging and that some form of latent program search is needed for this task and model size. Then, we see that all training methods show increasing performance when given more budget for latent optimization. Indeed, from using the encoder mean prediction to using 1 to 100 steps of gradient ascent, accuracy on the *Pattern* task keeps increasing, showing the ability of LPN to utilize compute to adapt to a new task at test time. Also, we observe that training with 1 gradient ascent step of latent optimization shows higher returns than training with mean latents when scaling the inference budget. With 100 steps of gradient ascent at inference time, mean training gets an accuracy of 67.5% when training with one gradient step reaches 99.5%. This demonstrates the benefits of training the latent space with the awareness that gradient ascent will be performed at test time, an important inductive bias for the LPN architecture. Lastly, we observe that gradient ascent vastly outperforms random search, validating that search without using the gradient signal is highly inefficient.

In addition, we ablate the impact of initializing latent optimization with the encoder versus using the prior. Specifically, we compare $z \sim p(z)$ and $z \sim q_\phi(z|x, y)$. Figure 8 shows that initializing search with the encoder is critical for performance across all training methods validating the intuition that

LPN can perform fast system 1-like reasoning using the encoder and then narrow down the search space during latent optimization, simulating system 2 reasoning.

5.4 Analyzing the Latent Space

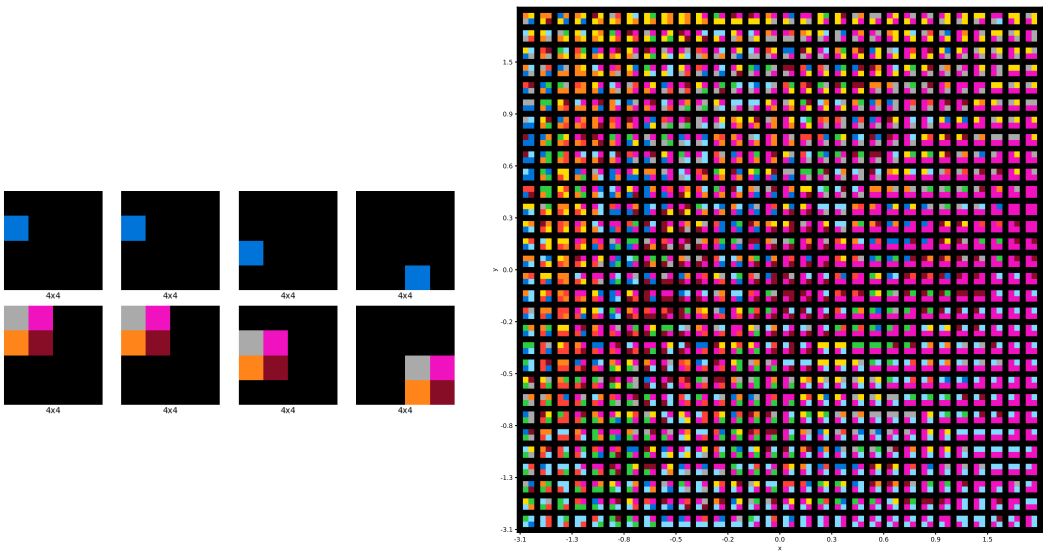


Figure 9: (Left) The 2D pattern task with inputs containing marker points where patterns should be placed, with patterns varying for each different program. (Right) The latent traversal visualizes the effect of traversing the latent space, on the predicted pattern by the decoder at marker points. Visualization in higher resolution in fig. 17

To validate that the encoder is learning programs in its latent space, we design an even simpler task with small 4x4 grids that have 2x2 patterns. We train an LPN model until convergence with a latent space of dimension 2 to easily visualize it in figure 9. Due to the simplicity of the task we train the model with *mean* training, i.e. no latent optimization. Because we are using a 2D Gaussian prior for the latent space, we can convert $\mathbb{R}^2$ to the unit square using the normal cumulative distribution function (CDF), and then plot on the unit square at coordinates (x, y) the decoder's output when conditioned by the latent $z = (\text{CDF}(x), \text{CDF}(y))$. These results demonstrate the diversity and smoothness of the latent space, showing structure in terms of color patterns, which motivates performing gradient ascent latent optimization in more complex tasks. This shows that the latent space can encode a wide range of diversity in its latent space which is especially important for adapting to unseen patterns.

5.5 Adapting Out-Of-Distribution

We study the out-of-distribution (OOD) behavior of different training methods in table 2. We use the same *Pattern* task as in section 5.3 but with different levels of color density for the patterns to reproduce. Specifically, we train on a density of 50% (which means half the patterns are black cells), and evaluate on 50%, 75% (weakly OOD), and 100% (strongly OOD). We observe that LPN equipped with gradient ascent at inference time can recover strong performance in the OOD setting (88% accuracy) whereas the prediction from the mean latent essentially gets an accuracy of 0%. Moreover, all methods suffer from a performance drop when increasing the OOD setting, but training with gradient ascent gets the lowest drop in accuracy, demonstrating a latent space better behaved for search at inference time. This motivates training LPN on some distribution of programs and then using the model equipped with gradient ascent at inference time given a novel task.

5.6 ARC-AGI 2024

Data Generation To test LPN on the ARC-AGI benchmark, we design a training phase where we aim at injecting the Core Knowledge priors into our LPN system. In this work, we use the `re-arc`

Training	Inference		
	Mean	GA 10	GA 100
Mean	**30.2** (15.7)	72.8 (29.5)	82.2 (21.3)
GA 1	26.6 (7.4)	**98.0** (0.6)	**99.2** (0.6)
GA 2	14.4 (3.5)	97.2 (1.6)	98.9 (1.2)
GA 3	1.0 (0.7)	85.7 (6.3)	98.3 (1.9)
GA 1 (g)	10.6 (6.4)	93.8 (5.1)	97.4 (1.9)

Training distribution

Training	Inference		
	Mean	GA 10	GA 100
Mean	**7.6** (5.6)	51.3 (35.7)	62.8 (38.3)
GA 1	7.4 (4.4)	**93.1** (4.3)	**97.7** (2.2)
GA 2	3.0 (2.6)	86.1 (6.0)	95.9 (2.1)
GA 3	0.0 (0.0)	55.0 (7.8)	93.9 (3.8)
GA 1 (g)	1.3 (1.0)	81.7 (13.3)	91.5 (5.5)

Weakly out-of-distribution

Training	Inference		
	Mean	GA 10	GA 100
Mean	**0.3** (0.5)	18.8 (14.5)	41.1 (29.6)
GA 1	0.0 (0.0)	**59.9** (11.6)	**88.0** (5.3)
GA 2	0.0 (0.0)	38.5 (13.0)	81.8 (10.9)
GA 3	0.0 (0.0)	11.3 (9.3)	72.0 (14.0)
GA 1 (g)	0.0 (0.0)	40.9 (19.8)	71.1 (14.3)

Strongly out-of-distribution

Table 2: Study of the out-of-distribution (OOD) performance on the *Pattern* task. Models are trained on patterns that have a density of 50% (half black, half colored), then evaluated on the same distribution, on a density of 75% (weakly OOD) and 100% (strongly OOD). Performance is averaged over 3 training runs with different seeds, with standard deviation in parentheses.

generators from Hodel [2024] to generate as many input-output pairs as needed during training in an online fashion. These 400 generators solely implement the programs from the ARC-AGI training set. Being able to perfectly execute this set of 400 programs is a necessary yet not sufficient condition for solving ARC-AGI. By limiting the training set to only the core knowledge priors shown in the training dataset we ensure that there is no data leakage from other sources such as the evaluation datasets. Therefore any performance on the evaluation set we demonstrate in this work is purely due to generalization.

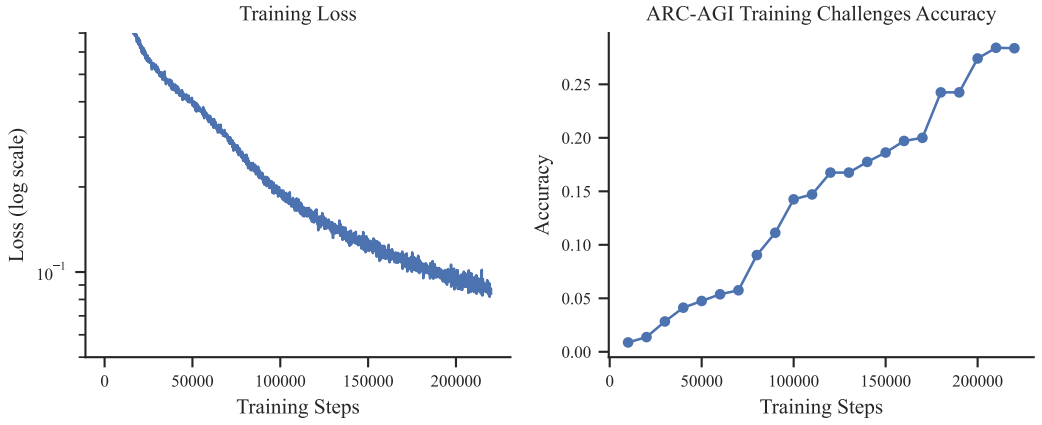


Figure 10: Training LPN on the `re-arc` generators from the ARC-AGI training set. The run took 5 days on a TPU v3-8 without converging. (Left): training loss in log scale. (Right): top-2 accuracy using *mean* inference mode (no latent optimization) on the 400 tasks from the ARC-AGI training set.

Training We train an LPN model of 39M parameters with a latent space of dimension 256, using *mean* training, i.e. no latent optimization during training. We performed 220k training steps with a batch size of 128 on a TPU v3-8, which took 5 days. We then fine-tuned the model for 2k steps

using 1 step of gradient-ascent latent optimization. Figure 6 shows the training loss and the model's top-2 accuracy on the 400 tasks from the training set, using mean latents for inference (no latent optimization). We note that training was stopped before convergence due to computational limits.

Inference	Training (400 tasks)	Evaluation (400 tasks)	Test (100 tasks)
Mean	24.83	3.25	-
GA 1	33.21	3.50	-
GA 5	38.25	6.12	-
GA 25	42.58	5.87	-
GA 100	44.58	8.38	-
GA 200	43.88	9.88	3.00
GA 300	46.13	9.38	-
GA 400	46.00	9.50	-

Table 3: Performance after 220k training steps (5 days on a TPU v3-8). Top-2 accuracy on the different sets, results are given in percentages. The *Test* results correspond to the leaderboard score, i.e. performance on the private test set. *GA* N stands for gradient ascent with N gradient steps, used for latent optimization.

Results We analyze scaling test-time latent optimization in table 3. For gradient ascent latent optimization we used the Adam [Kingma and Ba, 2017] optimizer with $\beta_1 = \beta_2 = 0.9$ and a cosine decay learning rate schedule starting at a learning rate of 1.0. Here as well, we observe performance increasing as we scale the number of gradient ascent steps performed in the latent space at test time, with convergence occurring around 400 steps. On the training dataset, gradient ascent nearly doubles the top-2 accuracy from 24.83% to 46.13% with 300 gradient steps, representing over 184 distinct programs generated with pixel-perfect accuracy. We show in fig. 11 a T-SNE visualization of the learned latent space and examples of generation on the training set using the trained checkpoint.

We observe a surprisingly high generalization to the evaluation dataset, even though it is known to include significantly more complex programs than the training set. The generalization performance sees similar increases when scaling gradient steps with performance increasing more than 3x between Mean and GA 200 inference. Since the ARC test set is private, we evaluated only the highest-performing inference method according to the evaluation set and scored 3.00% on the test dataset (leaderboard score). Given that there cannot be any data leakage from the evaluation dataset, this suggests that the generalization gap between training and testing could be slightly higher than that of training and evaluation. Importantly we highlight that figure 10 clearly shows that training is far from convergence, therefore we can expect both higher training accuracy and likely generalization with higher compute.

6 Conclusion

In this work, we introduce Latent Program Network (LPN), a novel approach to inductive program synthesis that leverages continuous latent space representations to enable efficient search and test-time adaptation in the space of latent programs. Unlike previous methods, LPN directly integrates the ability for test-time adaptation into the architecture, rather than relying on extensive sampling from the model. Our results demonstrate that, for program synthesis tasks over grids, LPN can use test-time adaptation to boost performance significantly by refining representations in the latent space. Specifically, we find that gradient-based search is the most efficient adaptation method for LPN, with step size and number of steps being important hyper-parameters for performance. Additionally, this adaptive approach enables LPN to generalize beyond its training distribution, which is particularly challenging for methods lacking test-time adaptation. On the ARC-AGI challenge, we show that LPN can generalize from the training set to the evaluation set without additional synthetic datasets, relying solely on test-time adaptation. We believe this to be a scalable approach as an alternative to methods that improve neural architecture performance simply by expanding the training set to reduce the generalization gap required. Our findings also indicate that LPN scales effectively with increased compute, suggesting that, with additional resources, LPN would achieve higher performance on the

ARC-AGI evaluation dataset and subsequently the test set. Lastly, we show that our LPN architecture for ARC-AGI outperforms previous vision transformer methods Li et al. [2024b] that struggled to overfit to single ARC tasks, conclusively showing that such grid-based program tasks are not a challenge for existing architectures. In particular, we train a network from scratch capable of executing over 180 programs on the arc training dataset. Therefore, future work should focus on other challenges that limit performance.

Limitations and Future Work Despite its strengths, LPN faces limitations. An initial limitation of our work is that, despite access to TPUs for training, our main ARC-AGI training run has not been trained to convergence yet. Training networks from scratch has the downside of long training times, so future work would need to scale compute to understand the convergence properties of LPN when training on RE-ARC, both in training time and parameter counts. Secondly, while we show that gradient ascent can be used to boost test time performance to a significant extent, gradient ascent as an optimization method may encounter local optima, which could restrict LPN's capacity to find the best latent solution. Future work could investigate different optimization procedures in the latent space to overcome this challenge, such as alternatives to initializing search and procedures for updating latents according to the gradient. Hybrid approaches combining evolution strategies (e.g., COMPASS [Chalumeau et al., 2023]) with gradient-based methods might improve search efficacy in future iterations. Another limitation is the challenge of representing complex, discrete programs within a continuous latent space, which may restrict expressivity for certain tasks or compositional generalization Shi et al. [2023]. Future work could explore discrete program representations, though this would require addressing the complexities of discrete space search.

In summary, LPN represents a step forward in adaptive program synthesis, demonstrating effective test-time adaptation, scalability, and potential for generalization. This work underscores the value of structured search and adaptive latent representations in advancing program synthesis capabilities.

Acknowledgments

We thank Google's TPU Research Cloud (TRC) for supporting this research. We are also grateful to Nathan Grinsztajn, Natasha Butt, Levi Lelis, and Jessica Hu for their feedback on the early versions of the paper.

References

Aws Albarghouthi, Sumit Gulwani, and Zachary Kincaid. Recursive program synthesis. In *Computer Aided Verification: 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings 25*, pages 934–950. Springer, 2013.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

Alexei Baevski and Michael Auli. Adaptive input representations for neural language modeling, 2019. URL `https://arxiv.org/abs/1809.10853`.

Sergey Bartunov, Vinod Nair, Peter Battaglia, and Tim Lillicrap. Continuous latent search for combinatorial optimization. In *Learning Meets Combinatorial Algorithms at NeurIPS2020*, 2020.

Alan W Biermann. The inference of regular lisp programs from examples. *IEEE transactions on Systems, Man, and Cybernetics*, 8(8):585–600, 1978.

Matko Bošnjak, Tim Rocktäschel, Jason Naradowsky, and Sebastian Riedel. Programming with a differentiable forth interpreter. In *International conference on machine learning*, pages 547–556. PMLR, 2017.

James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs, 2018. URL `http://github.com/jax-ml/jax`.

Rudy R Bunel, Alban Desmaison, Pawan K Mudigonda, Pushmeet Kohli, and Philip Torr. Adaptive neural compilation. *Advances in Neural Information Processing Systems*, 29, 2016.

Christopher P Burgess, Irina Higgins, Arka Pal, Loic Matthey, Nick Watters, Guillaume Desjardins, and Alexander Lerchner. Understanding disentangling in β-vae. *arXiv preprint arXiv:1804.03599*, 2018.

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. *arXiv preprint arXiv:2402.04858*, 2024.

Tales H Carvalho, Kenneth Tjhia, and Levi HS Lelis. Reclaiming the source of programmatic policies: Programmatic versus latent spaces. *arXiv preprint arXiv:2410.12166*, 2024.

Felix Chalumeau, Shikha Surana, Clément Bonnet, Nathan Grinsztajn, Arnu Pretorius, Alexandre Laterre, and Tom Barrett. Combinatorial optimization with policy adaptation using latent space search. *Advances in Neural Information Processing Systems*, 36:7947–7959, 2023.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Francois Chollet, Mike Knoop, Bryan Landers, Greg Kamradt, Hansueli Jud, Walter Reade, and Addison Howard. Arc prize 2024. `https://kaggle.com/competitions/arc-prize-2024`, 2024. Kaggle.

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

Alexey Dosovitskiy. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020.

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*, pages 835–850, 2021.

John K Feser, Swarat Chaudhuri, and Isil Dillig. Synthesizing data structure transformations from input-output examples. *ACM SIGPLAN Notices*, 50(6):229–239, 2015.

Jonathan Frankle, Peter-Michael Osera, David Walker, and Steve Zdancewic. Example-directed synthesis: a type-theoretic interpretation. *ACM Sigplan Notices*, 51(1):802–815, 2016.

David Furcy and Sven Koenig. Limited discrepancy beam search. In *IJCAI*, pages 125–131, 2005.

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.

Gaël Gendron, Qiming Bao, Michael Witbrock, and Gillian Dobbie. Large language models are not strong abstract reasoners. *arXiv preprint arXiv:2305.19555*, 2023.

Rafael Gómez-Bombarelli, Jennifer N Wei, David Duvenaud, José Miguel Hernández-Lobato, Benjamín Sánchez-Lengeling, Dennis Sheberla, Jorge Aguilera-Iparraguirre, Timothy D Hirzel, Ryan P Adams, and Alán Aspuru-Guzik. Automatic chemical design using a data-driven continuous representation of molecules. *ACS central science*, 4(2):268–276, 2018.

Alex Graves. Neural turing machines. *arXiv preprint arXiv:1410.5401*, 2014.

Nathan Grinsztajn, Daniel Furelos-Blanco, and Thomas D Barrett. Population-based reinforcement learning for combinatorial optimization. *arXiv preprint arXiv:2210.03475*, 2022.

Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. *ACM Sigplan Notices*, 46(1):317–330, 2011.

Nikolaus Hansen and Andreas Ostermeier. Completely derandomized self-adaptation in evolution strategies. *Evolutionary Computation*, 9(2):159–195, 2001. doi: 10.1162/106365601750190398.

Jonathan Heek, Anselm Levskaya, Avital Oliver, Marvin Ritter, Bertrand Rondepierre, Andreas Steiner, and Marc van Zee. Flax: A neural network library and ecosystem for JAX, 2024. URL `http://github.com/google/flax`.

Geoffrey E Hinton and Ruslan R Salakhutdinov. Reducing the dimensionality of data with neural networks. *science*, 313(5786):504–507, 2006.

Michael Hodel. Addressing the abstraction and reasoning corpus via procedural example generation, 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 *International Conference on Machine Learning*, pages 4308–4318. PMLR, 2021.

André Hottung, Bhanu Bhandari, and Kevin Tierney. Learning a latent search space for routing problems using variational autoencoders. In *International Conference on Learning Representations*, 2021a.

André Hottung, Yeong-Dae Kwon, and Kevin Tierney. Efficient active search for combinatorial optimization problems. *arXiv preprint arXiv:2106.05126*, 2021b.

Jonas Hübotter, Sascha Bongni, Ido Hakimi, and Andreas Krause. Efficiently learning at test-time: Active fine-tuning of llms. *arXiv preprint arXiv:2410.08020*, 2024.

Daniel Kahneman. *Thinking, fast and slow*. 1st ed. New York : Farrar, Straus and Giroux, [2011] ©2011, 2011. URL `https://search.library.wisc.edu/catalog/9910114919702121`. Includes bibliographical references (pages 447-448) and index.

Yunho Kim, Jaehyun Park, Heejun Kim, Sejin Kim, Byung-Jun Lee, and Sundong Kim. Diffusion-based offline rl for improved decision-making in augmented arc task. *arXiv preprint arXiv:2410.11324*, 2024.

Diederik P Kingma. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017. URL `https://arxiv.org/abs/1412.6980`.

Victor Kolev, Bogdan Georgiev, and Svetlin Penkov. Neural abstract reasoner. *arXiv preprint arXiv:2011.09860*, 2020.

Andrei Nikolaevich Kolmogorov. Three approaches to the definition of the concept "quantity of information". *Problemy peredachi informatsii*, 1(1):3–11, 1965.

Karol Kurach, Marcin Andrychowicz, and Ilya Sutskever. Neural random-access machines. *arXiv preprint arXiv:1511.06392*, 2015.

Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436–444, 2015.

Wen-Ding Li, Keya Hu, Carter Larsen, Yuqing Wu, Simon Alford, Caleb Woo, Spencer M Dunn, Hao Tang, Michelangelo Naim, Dat Nguyen, et al. Combining induction and transduction for abstract reasoning. *arXiv preprint arXiv:2411.02272*, 2024a.

Wenhao Li, Yudong Xu, Scott Sanner, and Elias Boutros Khalil. Tackling the abstraction and reasoning corpus with vision transformers: the importance of 2d representation, positions, and objects, 2024b. URL `https://arxiv.org/abs/2410.06405`.

Kazem Meidani, Parshin Shojaee, Chandan K Reddy, and Amir Barati Farimani. Snip: Bridging mathematical symbolic and numeric realms with unified pre-training. *arXiv preprint arXiv:2310.02227*, 2023.

Melanie Mitchell, Alessandro B Palmarini, and Arseny Moskvichev. Comparing humans, gpt-4, and gpt-4v on abstraction and reasoning tasks. *arXiv preprint arXiv:2311.09247*, 2023.

Arseny Moskvichev, Victor Vikram Odouard, and Melanie Mitchell. The conceptarc benchmark: Evaluating understanding and generalization in the arc domain. *arXiv preprint arXiv:2305.07141*, 2023.

Vijayaraghavan Murali, Letao Qi, Swarat Chaudhuri, and Chris Jermaine. Neural sketch learning for conditional program generation. *arXiv preprint arXiv:1703.05698*, 2017.

Maxwell Nye, Luke Hewitt, Joshua Tenenbaum, and Armando Solar-Lezama. Learning to infer program sketches. In *International Conference on Machine Learning*, pages 4861–4870. PMLR, 2019.

Peter-Michael Osera and Steve Zdancewic. Type-and-example-directed program synthesis. *ACM SIGPLAN Notices*, 50(6):619–630, 2015.

Richard E Pattis. *Karel the robot: a gentle introduction to the art of programming*. John Wiley & Sons, 1994.

Scott Reed and Nando De Freitas. Neural programmer-interpreters. *arXiv preprint arXiv:1511.06279*, 2015.

Kensen Shi, Joey Hong, Yinlin Deng, Pengcheng Yin, Manzil Zaheer, and Charles Sutton. Exedec: Execution decomposition for compositional generalization in neural program synthesis. *arXiv preprint arXiv:2307.13883*, 2023.

Ray J Solomonoff. A formal theory of inductive inference. part i. *Information and control*, 7(1):1–22, 1964.

Phillip D Summers. A methodology for lisp program construction from examples. *Journal of the ACM (JACM)*, 24(1):161–175, 1977.

Shao-Hua Sun, Hyeonwoo Noh, Sriram Somasundaram, and Joseph Lim. Neural program synthesis from diverse demonstration videos. In *International Conference on Machine Learning*, pages 4790–4799. PMLR, 2018.

Dweep Trivedi, Jesse Zhang, Shao-Hua Sun, and Joseph J Lim. Learning to synthesize programs as interpretable and generalizable policies. *Advances in neural information processing systems*, 34: 25146–25163, 2021.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in Neural Information Processing Systems*, 2017.

Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi DQ Bui, Junnan Li, and Steven CH Hoi. Codet5+: Open code large language models for code understanding and generation. *arXiv preprint arXiv:2305.07922*, 2023.

Ruibin Xiong, Yunchang Yang, Di He, Kai Zheng, Shuxin Zheng, Chen Xing, Huishuai Zhang, Yanyan Lan, Liwei Wang, and Tie-Yan Liu. On layer normalization in the transformer architecture, 2020. URL `https://arxiv.org/abs/2002.04745`.

Yuichi Yoshida and Takeru Miyato. Spectral norm regularization for improving the generalizability of deep learning. *arXiv preprint arXiv:1705.10941*, 2017.

Peiyu Yu, Dinghuai Zhang, Hengzhi He, Xiaojian Ma, Ruiyao Miao, Yifan Lu, Yasi Zhang, Deqian Kong, Ruiqi Gao, Jianwen Xie, et al. Latent energy-based odyssey: Black-box optimization via expanded exploration in the energy-based latent space. *arXiv preprint arXiv:2405.16730*, 2024.

Wojciech Zaremba, Tomas Mikolov, Armand Joulin, and Rob Fergus. Learning simple algorithms from examples. In *International conference on machine learning*, pages 421–429. PMLR, 2016.

Shun Zhang, Zhenfang Chen, Yikang Shen, Mingyu Ding, Joshua B Tenenbaum, and Chuang Gan. Planning with large language models for code generation. *arXiv preprint arXiv:2303.05510*, 2023.

A Experiments

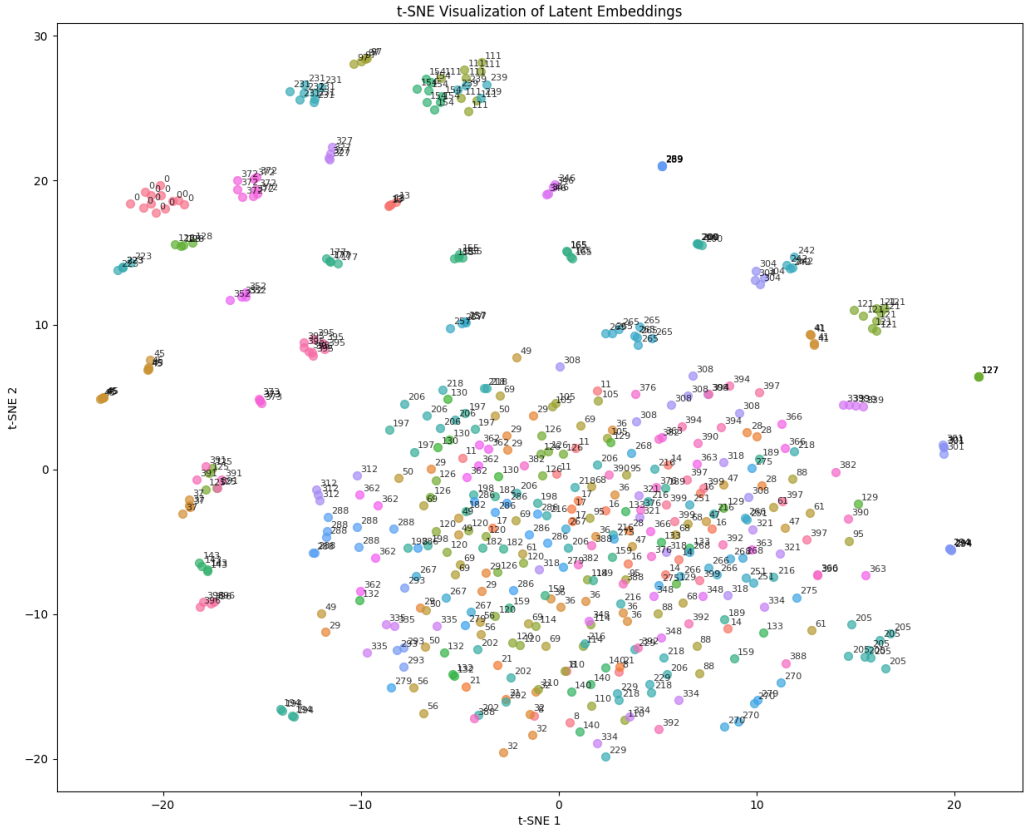


Figure 11: T-SNE of different program embeddings from pairs sampled from the `re-arc` generators. IDs correspond to different generators and hence represent specific training tasks.

Figure 11 shows a T-SNE visualization of the latent space of input-output pairs sampled from the `re-arc` generators. Tasks that are well mastered by the decoder seem to be clustered in the latent space while tasks that aren't learned yet are part of a common bag of programs yet to be learned.

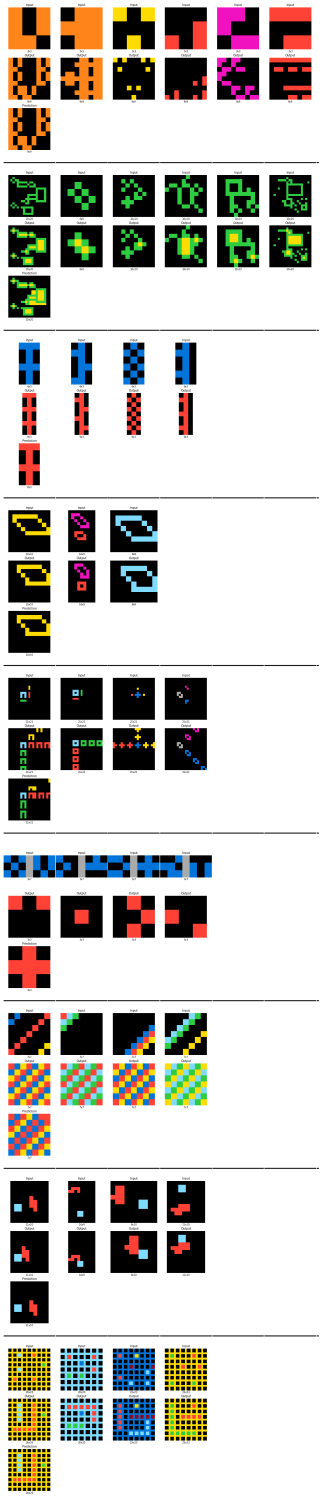


Figure 12: Random examples from the ARC-AGI training set with positive and negative cases. For each task, the top row contains the inputs, the middle row the outputs, and the bottom row is the model's prediction.

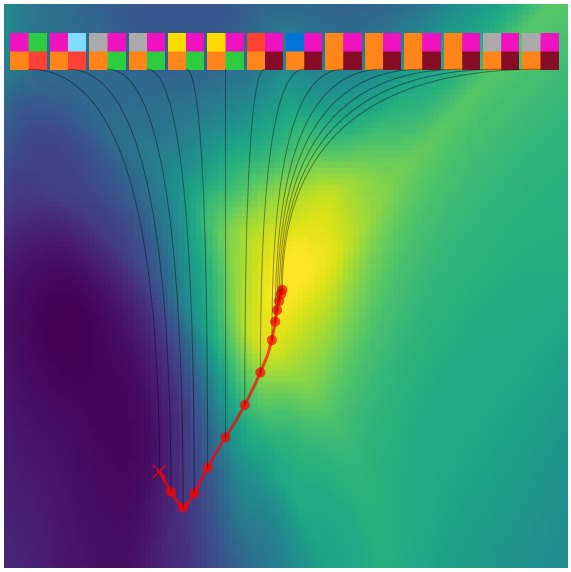


Figure 13: Gradient ascent trajectory as shown in figure 1.

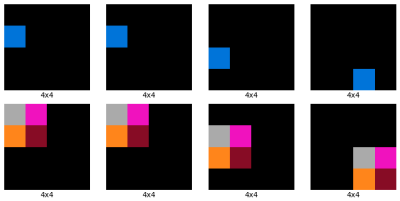


Figure 14: Small pattern task

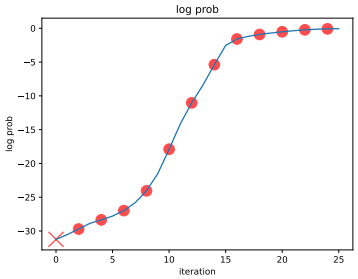


Figure 15: Log prob maximization during latent search

Figure 16: Gradient ascent trajectory detailed

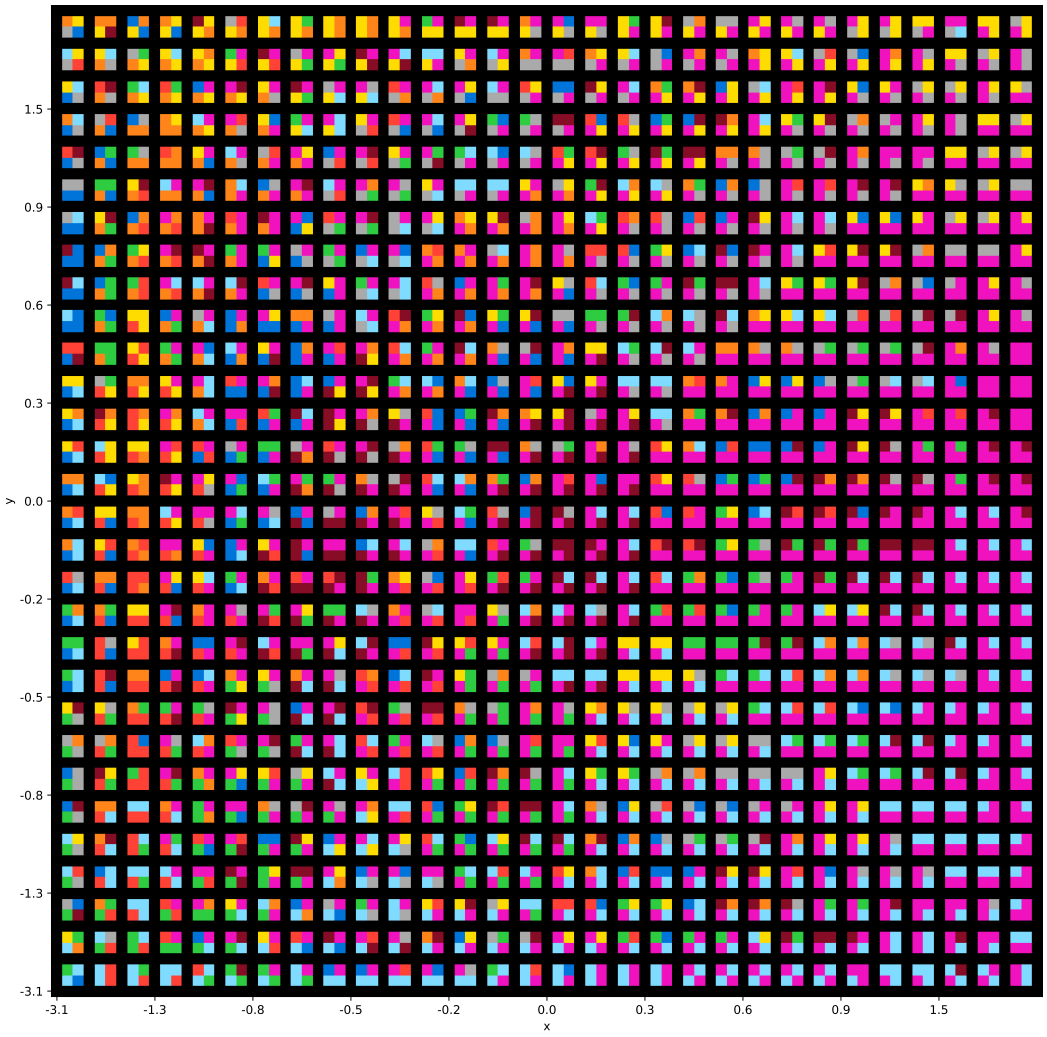


Figure 17: 2D latent space of patterns, zoomed in version of figure 9.