

A Datasets

A.1 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 4) 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, meaning it is the same across different pairs but varies from specification to specification. This task demonstrates how deep learning methods without test-time computation may still make errors on such tasks. We then extend this task to study an out-of-distribution setting in section 5.4.

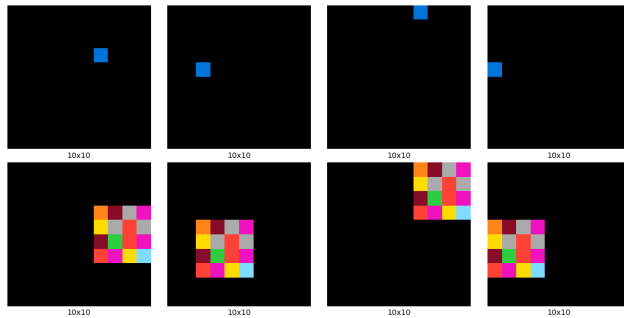


Figure 4: Example of input (top row) and output (bottom row) pairs of a specification sampled from the *Pattern* task. Each sample is a batch of 4 pairs that share the same pattern.

A.2 ARC-AGI

We evaluate LPN on the ARC-AGI benchmark by training it on the `re-arc` dataset [Hodel, 2024], a dataset designed to be in distribution relative to the ARC training set, previously used for training large language models and ARC specific transformers to solve ARC-AGI Li et al. [2024b], Akyürek et al. [2024]. Most previous works however expand beyond this dataset, usually synthetically adding additional programs [Butt et al., 2024], or crafted datasets. We restrict training to tasks from only this dataset with the goal of assessing generalization instead of closing a performance gap with more data. This also ensures LPN solely learns from core knowledge priors without data leakage from the evaluation set. The evaluation set is known to be significantly out of distribution and therefore represents a challenging generalization experiment. We train a 178M-parameter LPN with a 256-dim latent space for 100k steps (batch size 256) for 2 days on a TPU v4-32, see Appendix E for further details, in terms of data, this amounts to 51M I/O pairs. Leveraging insights from smaller scale experiments we make use of gradient in LPN during training. We train LPN in Grad 0 mode for 95k steps and then fine tune in Grad 1 for a further 5k steps. We outline our hyperparameter procedure for both LPN and TTT for ARC-AGI in Appendix E.

B Expanded Experiments

B.1 LPN Training Extended

We provide an expanded version of Table 1 with additional ablations for both the training and inference axes.

Training	Inference					
	Grad 0	Grad 1	Grad 5	Grad 20	Grad 100	Sample 250
Grad 0	3.2 (2.7)	3.6 (3.0)	18.8 (14.4)	52.5 (25.0)	67.5 (20.0)	3.2 (2.7)
Grad 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)
Grad 1 **	0.6 (0.1)	13.7 (3.0)	60.2 (7.5)	88.9 (6.0)	94.1 (3.8)	0.7 (0.2)
Grad 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)
Grad 5 **	0.0 (0.0)	0.0 (0.0)	9.9 (2.9)	87.1 (6.0)	95.1 (3.3)	0.0 (0.0)
Sample 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)
Sample 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 4: Ablation study of LPN training and inference methods on the *Pattern* task. Rows represent training methods: LPN Grad [N] for N gradient steps, LPN Grad [N] ** with gradient flow through latent optimization (analogous to meta-learning), and Sample [N] for N random samples. Columns represent inference methods: Grad [N] for N gradient ascent steps and Sample [N] for random search with N samples. Training was performed for 20k steps with 3 seeds, with performance reported as mean (standard deviation) over 3 runs. Bold values indicate the best performance per inference method.

B.2 Adapting Out-Of-Distribution

We provide expanded results for the out-of-distribution (OOD) experiments outlined in Section 5.4. We include results for varying levels of OOD starting with in-distribution, weak OOD, and strong OOD.

Training	Inference		
	Grad 0	Grad 10	Grad 100
In-Context	15.3 (0.6)	-	-
Test Time Training	15.3 (0.6)	17.0 (1.7)	0.78 (0.69)
LPN Grad 0	30.2 (15.7)	72.8 (29.5)	82.2 (21.3)
LPN Grad 1	26.6 (7.4)	98.0 (0.6)	99.2 (0.6)
LPN Grad 2	14.4 (3.5)	97.2 (1.6)	98.9 (1.2)
LPN Grad 3	1.0 (0.7)	85.7 (6.3)	98.3 (1.9)
LPN Grad 1 **	10.6 (6.4)	93.8 (5.1)	97.4 (1.9)

In-distribution

Training	Inference		
	Grad 0	Grad 10	Grad 100
In-Context	2.1 (0.9)	-	-
Test Time Training	2.1 (0.9)	2.1 (1.1)	0.00 (0.00)
LPN Grad 0	7.6 (5.6)	51.3 (35.7)	62.8 (38.3)
LPN Grad 1	7.4 (4.4)	93.1 (4.3)	97.7 (2.2)
LPN Grad 2	3.0 (2.6)	86.1 (6.0)	95.9 (2.1)
LPN Grad 3	0.0 (0.0)	55.0 (7.8)	93.9 (3.8)
LPN Grad 1 **	1.3 (1.0)	81.7 (13.3)	91.5 (5.5)

Weakly out-of-distribution

Training	Inference		
	Grad 0	Grad 10	Grad 100
In-Context	0.0 (0.0)	-	-
Test Time Training	0.0 (0.0)	1.8 (0.7)	0.3 (0.1)
LPN Grad 0	0.3 (0.5)	18.8 (14.5)	41.1 (29.6)
LPN Grad 1	0.0 (0.0)	59.9 (11.6)	88.0 (5.3)
LPN Grad 2	0.0 (0.0)	38.5 (13.0)	81.8 (10.9)
LPN Grad 3	0.0 (0.0)	11.3 (9.3)	72.0 (14.0)
LPN Grad 1 **	0.0 (0.0)	40.9 (19.8)	71.1 (14.3)

Strongly out-of-distribution

Table 5: 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.

611 B.3 Validating the Decoder

612 Training deep networks from scratch to solve ARC-like tasks has been challenging [Li et al., 2024c].
 613 If it is the case that neural networks struggle even to learn to execute single programs this represents a
 614 significant bottleneck to training models from scratch on a broad distribution of programs. Therefore,
 615 before training LPN end-to-end, we conclusively show that our decoder architecture does not suffer
 616 from such a bottleneck, and can learn individual programs.

617 We show 5 of the 400 total tasks from the ARC-AGI training set, and for each of these tasks,
 618 we train a small LPN architecture of 800k parameters (except for the last task which required a
 619 bigger model with 8.7M parameters) on the corresponding task generator from re-arc [Hodel,
 620 2024]. Specifically, we select the first five tasks from the arc-agi_training_challenges json
 621 file (007bbfb7, 00d62c1b, 017c7c7b, 025d127b, 045e512c) shown in figure Figure 5.

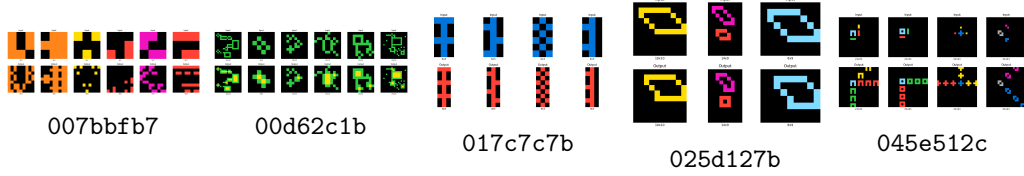


Figure 5: Overfit training on the first 5 ARC-AGI training tasks. The captions correspond to task IDs. For each task, the top row contains the input grids, and the bottom row the output grids. Each task consists of observing all pairs but the first and inferring the output of the leftmost pair given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

622 We evaluate both the distribution of re-arc generated tasks on which it is trained and on the true
 623 task from the ARC-AGI training set in figure Figure 6. We show that for each task, the small LPN
 624 decoder-only model successfully learns individual programs and manages to solve the corresponding
 625 ARC-AGI task. Therefore, our model outperforms previously reported results in Li et al. [2024c],
 626 and concludes that our architecture does not suffer from a decoder bottleneck. Note that the encoder
 627 is not helpful in this experiment since the task is always the same. Our later results on ARC-AGI
 628 Section 5.6 take this a step further and show that we can learn a single transformer architecture
 629 capable of executing over 270 programs in the ARC training dataset.

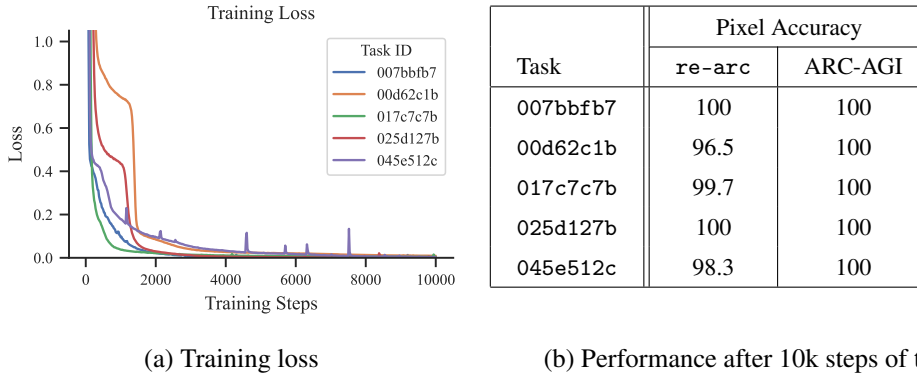


Figure 6: Training loss and performance of LPN training on 5 of the re-arc distributions. For each task, only samples from the re-arc generators are used for training. The corresponding ARC-AGI tasks are never seen during training.

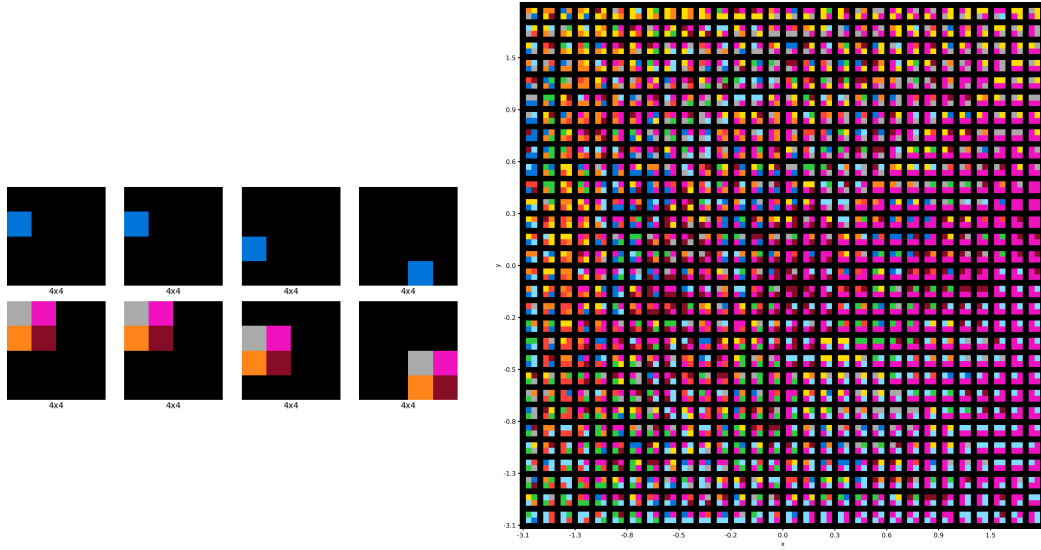
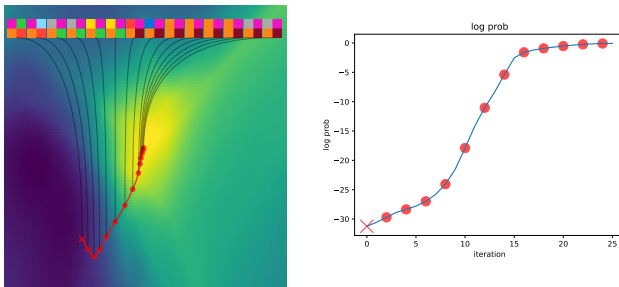


Figure 7: (Left) A 2D pattern task with inputs containing marker points where patterns should be placed, with patterns varying for each program. (Right) The latent traversal visualizes the effect of traversing the latent space, on the predicted pattern by the decoder at marker points.

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 a small LPN model until convergence with a latent space of dimension 2 to easily visualize it in figure Figure 7. 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 $\text{CDF}(z) = (x, y)$, or equivalently, $z = (\text{PPF}(x), \text{PPF}(y))$ using the percent-point function (PPF). 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.



The figures on the left show a random initialization in the latent space and a corresponding latent trajectory following the gradient ascent algorithm. The log-likelihood for each latent is shown during optimization. The decoded pattern is traced until convergence in the latent space, giving insights into how LPN performs latent optimization at test time to find the optimal latent program.

642 B.5 Scaling Specification Size

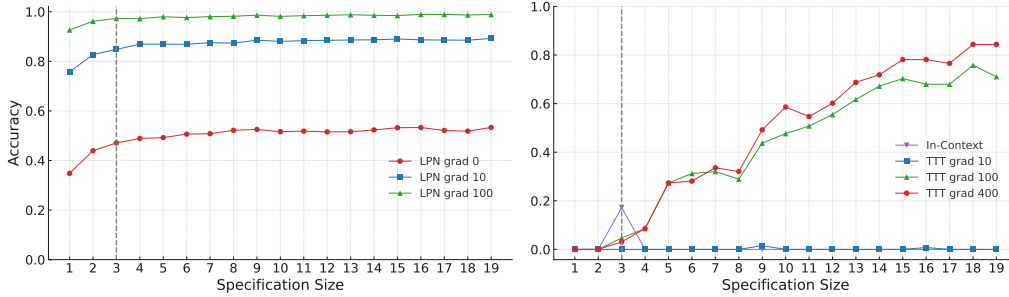


Figure 8: Accuracy of Latent Program Network (LPN) and Test-Time Training (TTT) models as we scale specification size. Higher specification sizes improve accuracy across different training methods. The dashed vertical line at specification size 3 indicates the fixed size used during training.

643 To analyze the effect of scaling specification size, we conduct an ablation using models trained on
 644 the pattern task for a single seed. We compare both in-context learning and LPN. We then evaluate
 645 different inference strategies on the in-distribution dataset and analyze their performance as the test
 646 specification size increases.

647 Our results indicate that LPN generalizes effectively across varying specification sizes. This robust-
 648 ness is primarily due to the use of mean pooling, which prevents overfitting to specific context lengths
 649 and enables the model to gracefully handle both slightly larger and significantly larger specifica-
 650 tion sizes. Furthermore, mean pooling encourages representations of the same program to remain
 651 structurally similar, improving generalization.

652 In contrast, in-context learning exhibits strong overfitting to the training specification size, with a
 653 noticeable drop in performance as the specification size deviates from the training distribution. How-
 654 ever, we observe that fine-tuning-based inference can scale effectively with increasing specification
 655 sizes, provided that a sufficient amount of data is available. The effectiveness of fine-tuning is highly
 656 dependent on the choice of hyperparameters, requiring substantial amounts of data to maintain strong
 657 performance as the specification size grows.

658 B.5.1 Performance Across Training Sizes

659 To further explore the impact of training specification size on model performance, we present results
 660 for training sizes 1, 7, and 11, evaluated across test sizes 1, 3, 7, 11, 15, and 19 on the pattern task.
 661 The following tables report accuracy for LPN with and without gradient ascent, In-Context Learning,
 662 and Test-Time Training (TTT). Notably, In-context learning consistently overfits to specific test sizes
 663 that align closely with the training size. In contrast, LPN, particularly with 100 gradient ascent
 664 steps, demonstrates robust generalization, maintaining high accuracy across all test sizes and training
 665 sizes. TTT shows improved performance as test sizes increase, approaching LPN’s accuracy at larger
 666 specification sizes when also trained on large specification sizes.

Method	Test Size					
	1	3	7	11	15	19
LPN grad 0	38.9	44.1	47.5	44.7	48.8	45.9
LPN grad 100	89.8	92.1	94.1	94.1	95.3	96.3
In-Context	2.1	0.0	0.0	0.0	0.0	0.0
TTT grad 100	7.8	19.9	50.9	72.4	81.8	90.4

Table 6: Performance (top-2 accuracy, percentage) for training size 1 on the pattern task.

Method	Test Size					
	1	3	7	11	15	19
LPN grad 0	14.6	27.7	32.8	33.4	33.2	35.1
LPN grad 100	87.7	89.3	92.7	91.0	92.6	93.6
In-Context	0.0	0.0	78.3	0.0	0.0	0.0
TTT grad 100	11.3	34.7	77.9	90.6	94.7	93.2

Table 7: Performance (top-2 accuracy, percentage) for training size 7 on the pattern task.

Method	Test Size					
	1	3	7	11	15	19
LPN grad 0	11.1	26.6	40.8	43.2	50.0	48.8
LPN grad 100	91.4	97.1	97.5	96.9	98.0	97.9
In-Context	0.0	0.0	0.1	94.1	0.1	0.0
TTT grad 100	0.1	16.2	56.1	78.3	93.7	92.2

Table 8: Performance (top-2 accuracy, percentage) for training size 11 on the pattern task.

667 B.6 Measuring Floating Point Operations

668 To evaluate the computational efficiency of different methods, we measure the number of floating-point operations (FLOPs) required for performing inference on a single task. The reported FLOP
669 measurements correspond to the cost of generating an entire 30x30 grid given 3 input-output pairs
670 (specification size).
671

672 While gradient-based approaches do not require an additional computational budget in terms of trial
673 attempts, their inference mechanisms involve performing gradient updates, either in parameter space
674 or latent space, which adds a computational overhead. A key distinction between TTT and LPN is
675 their computational cost: LPN updates only the latent space by backpropagating through the decoder,
676 whereas TTT requires backpropagation through all model parameters. This makes TTT significantly
677 more expensive and less efficient for real-time applications.

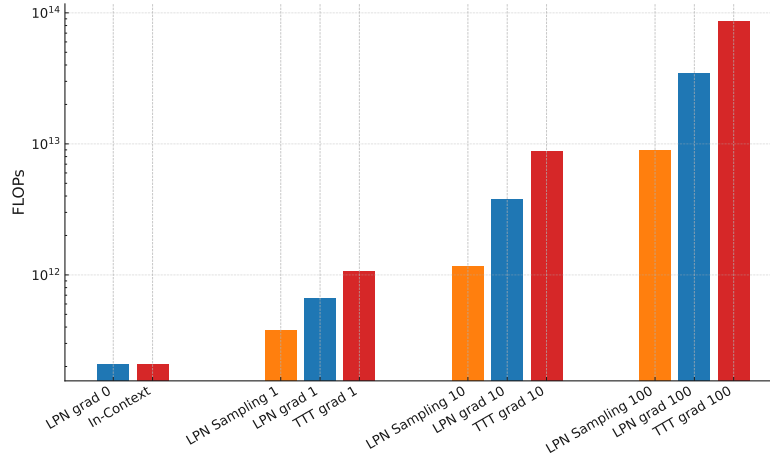


Figure 9: Floating-point operations (FLOPs) required for test-time inference across different methods.

678 The LPN Sampling results can be understood as ablation of LPN without gradient-based search, but
679 only sampling multiple latents from the encoder computing the decoder likelihood for each, avoiding
680 the computational cost of backpropagating through the decoder. However, this efficiency gain comes
681 at a significant cost in performance, as shown in the pattern task results Table 1.

B.7 Sequence Ablation

B.7.1 Dataset

To investigate the generalizability of our results beyond environments with significant spatial structure, we perform an ablation on a synthetic sequence task and replicate the analysis previously conducted on the pattern dataset. This synthetic dataset features a vast program space, with over 100 million unique programs, each defined by composing 3 to 5 parameterized rules that transform sequences of numbers (ranging from 0 to 4). Programs are composed of 3 to 5 rules with specific integer-based parameters (e.g., thresholds or operation values). These rules are then applied in their chosen order: for each rule, we process the sequence from left to right, transforming each position according to the rule’s condition and operation, producing an intermediate sequence that serves as input to the next rule. This sequential, rule-by-rule application, with each rule scanning left to right, enables complex transformations and vast numbers of programs with meaningfully different outputs. The core rules are defined as follows:

- If a number is greater than its right neighbor by a threshold k , decrease it by a value m , modulo 5.
- If a number has identical neighbors on both sides, multiply its value by a factor n and clip the result to 4.
- If a number is greater than its left neighbor by a threshold k , add a value m , modulo 5.
- If a number is less than its right neighbor by a threshold k , subtract a value m , modulo 5.
- Replace the number with a function of its neighbors (e.g., sum, average, maximum, or minimum), modulo 5.

These parameterized rules may interact at the same position across their sequential applications, creating cascading effects that result in complex transformations. While a single input-output pair may not uniquely identify the underlying program, multiple pairs typically provide sufficient information to determine it. The table below presents accuracy metrics for models evaluated on this task.

B.7.2 Results

Our experiments demonstrate that Test-Time Training (TTT) exhibits overfitting, as evidenced by their performance degradation when gradient fine tuning is performed. In contrast, LPN maintains robust performance without overfitting. Furthermore, incorporating a single gradient step during training enhances LPN performance marginally at higher inference gradients.

Training	Inference (Avg \pm Std)		
	Grad 0	Grad 10	Grad 100
In-Context	0.78 (0.01)	-	-
Test-Time Training	0.78 (0.01)	0.73 (0.01)	0.12 (0.01)
LPN Grad 0	0.76 (0.02)	0.83 (0.01)	0.81 (0.01)
LPN Grad 1	0.71 (0.00)	0.86 (0.01)	0.82 (0.02)
LPN Grad 2	0.63 (0.01)	0.85 (0.00)	0.80 (0.01)
LPN Grad 3	0.49 (0.04)	0.85 (0.00)	0.82 (0.02)
LPN Grad 1 **	0.68 (0.01)	0.85 (0.00)	0.81 (0.01)

Table 9: Study of model performance across different inference gradients for methods: In-Context, Test-Time Training with learning rate of $1e-4$, LPN Grad 0, LPN Grad 1, LPN Grad 2, LPN Grad 3 (all with learning rates of $1e-1$), LPN Grad 1 **. Performance is averaged over 3 training runs, with standard deviation in parentheses.

712 **B.8 LPN fine tuning**

713 We also ablate LPN training by training in full Grad 0 mode for 100k steps. We show the results
 714 compared to fine tuning for 5k steps in Grad 1 mode.

FLOPs	LPN Grad 1 Tune	LPN Grad 0 Tune
2E+11	7.75	8.25
2E+12	10.25	10.25
2E+13	15.25	13.60
2E+14	15.50	15.10
2E+15	15.50	15.10

Table 10: Performance of LPN with and without gradient tuning on ARC-AGI for 10k steps

715 We find a marginal performance gain at higher computational budgets from fine-tuning with one
 716 gradient step, so we adopt this approach in the main method. However, we note a slight perfor-
 717 mance drop in Grad 0 inference, which is expected as the network begins to optimize with gradient
 718 adaptation.

719 **B.9 ARC-AGI Solution Analysis**

720 In this section we investigate whether there are differences between the types of tasks in ARC-AGI
721 evaluation dataset that LPN solves vs Test-Time Training. We run both methods at the budget of
722 $2E+14$ and analyze the problems solved. We first analyze the overlap between problems solved
723 between the two methods. We see that there is a spread of the different problems being solved by the
724 different architectures, a roughly even split between problems only solved by TTT, problems solved
725 only by LPN and problems solved by both.

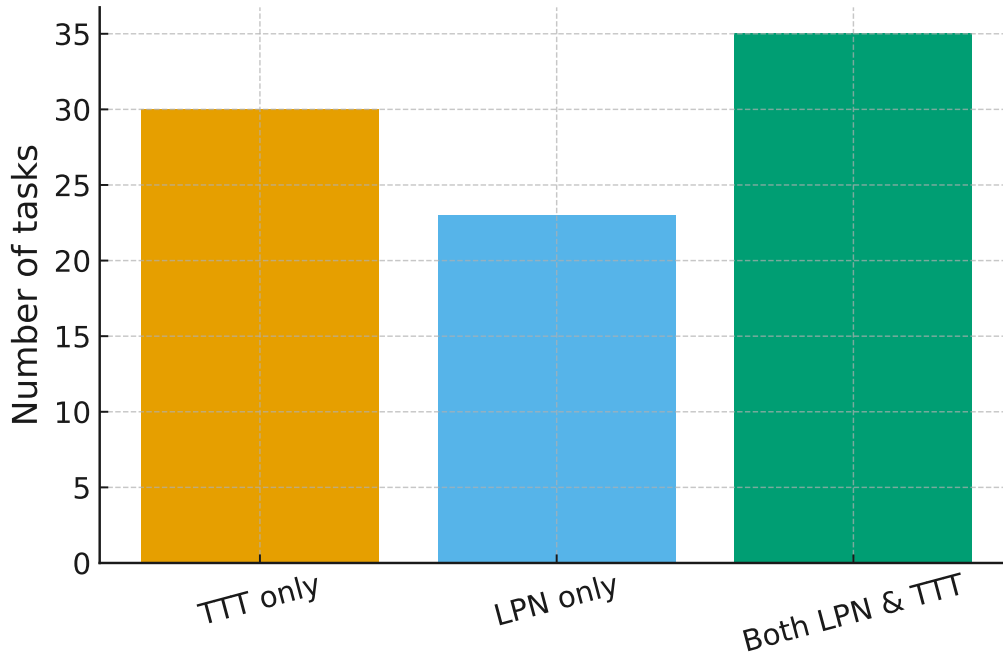


Figure 10: Bar Chart showing Problems Uniquely solved by LPN and TTT and tasks solved when at least one solves the problem

726 If we measure the accuracy when either LPN or TTT solves an ARC puzzle. We achieve a combined
727 score of 22%. We also show 3 examples of problems solved only by LPN and 3 examples only solved
728 by Test-Time Training to understand the differences between the problems each method solved.

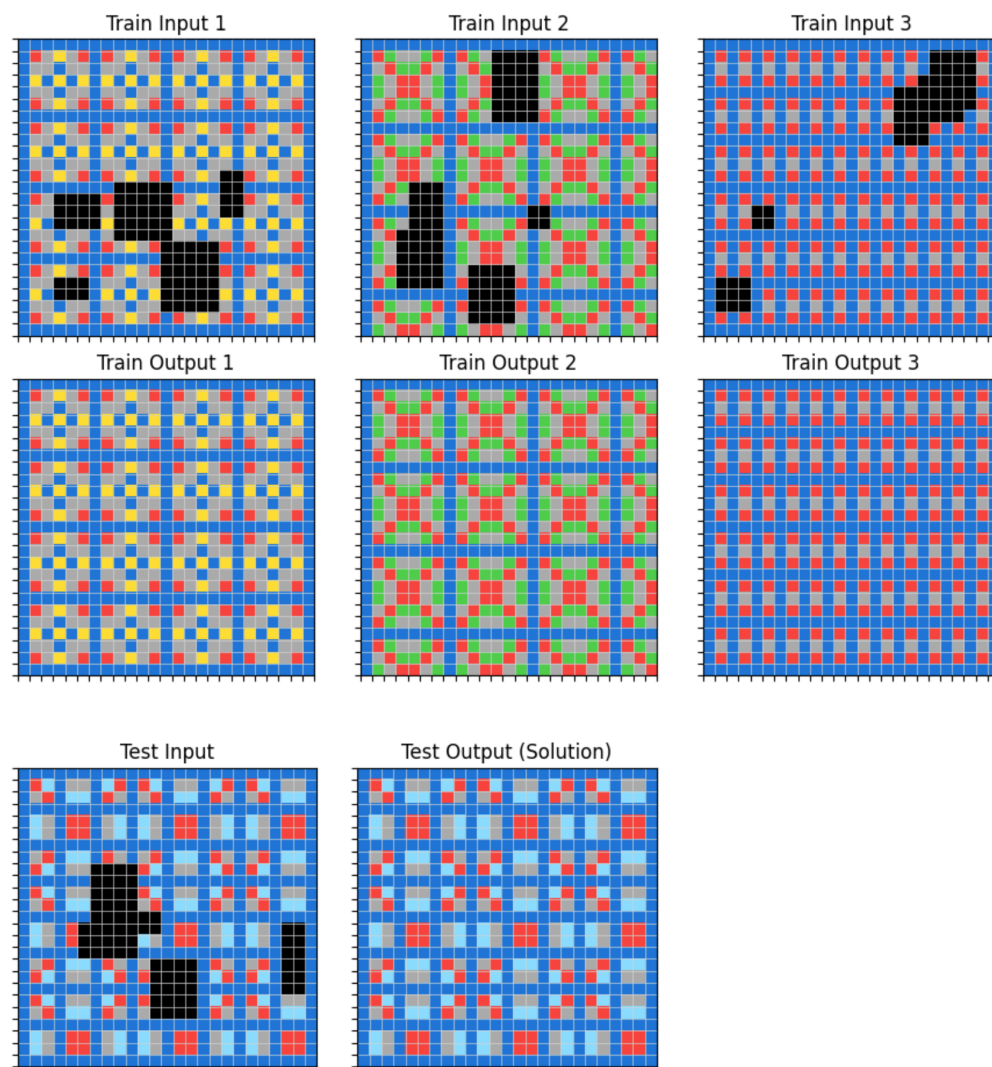


Figure 11: LPN Example 1

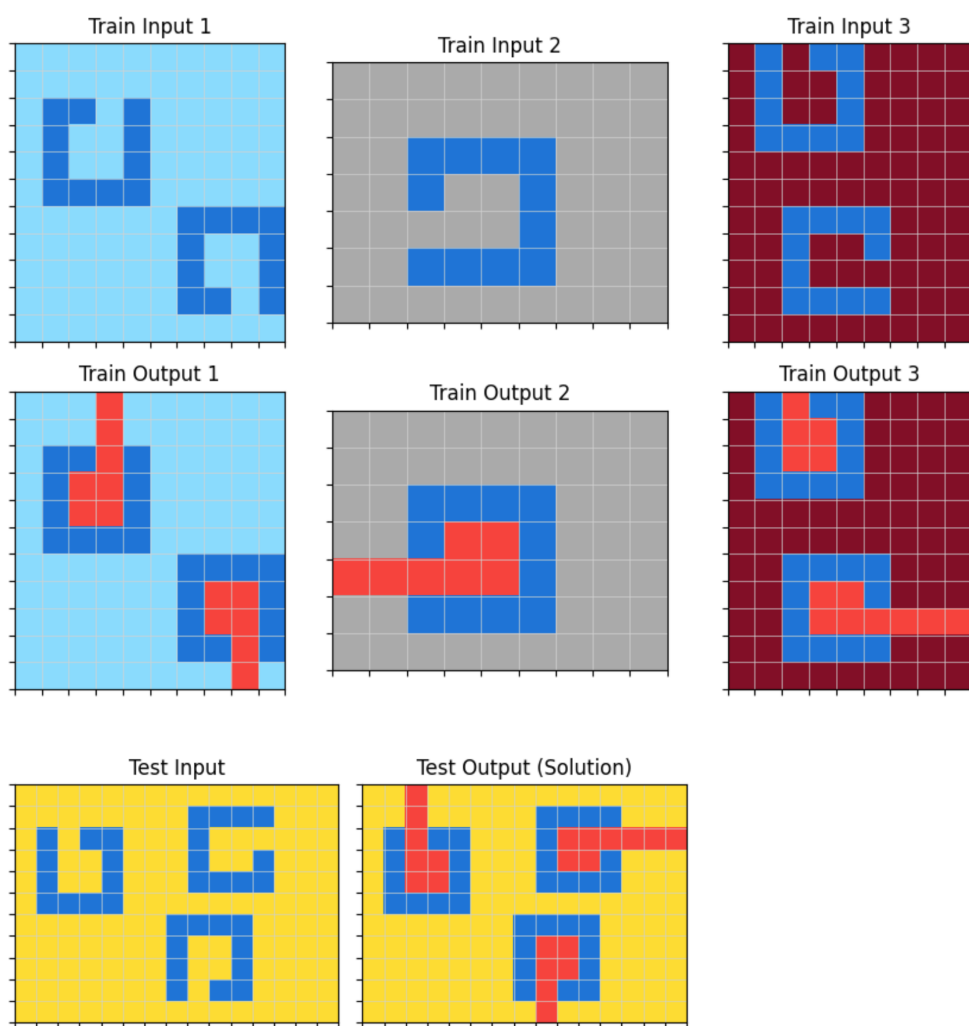


Figure 12: LPN Example 2

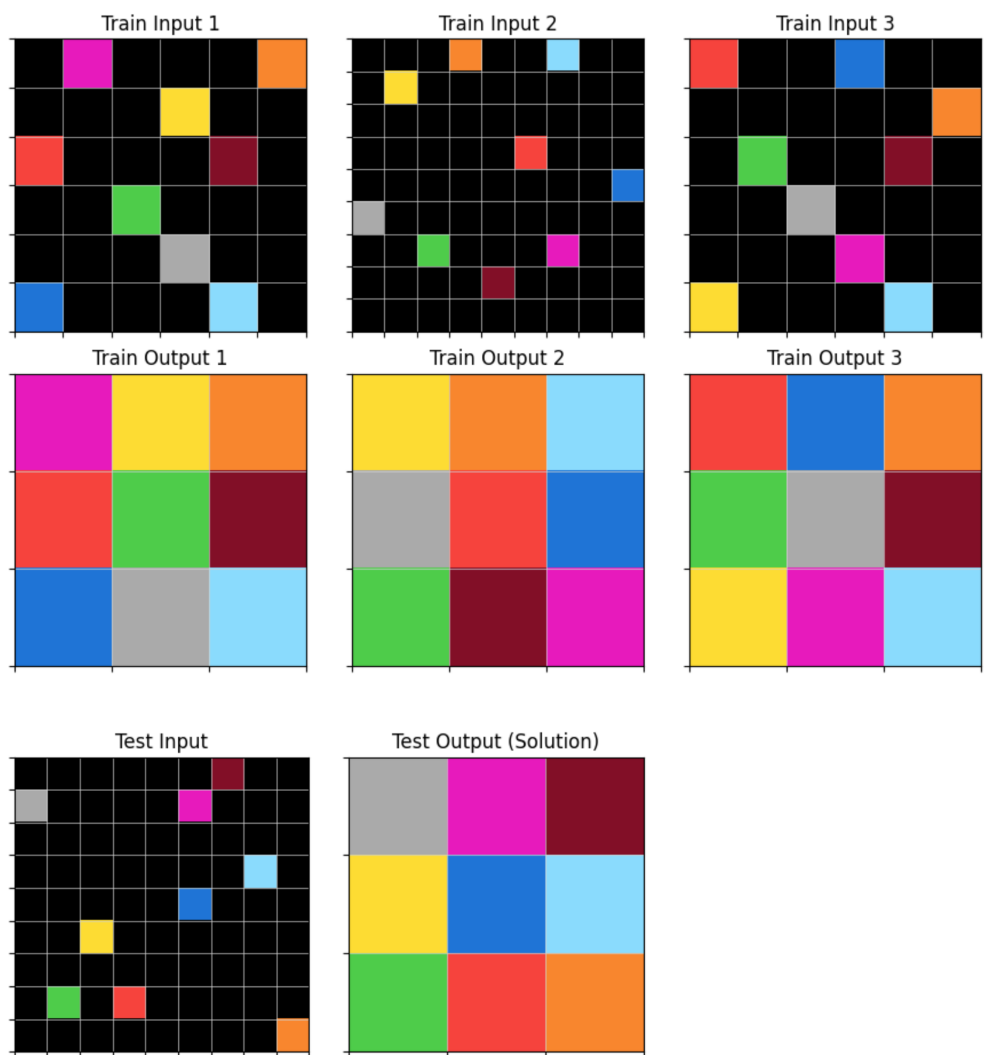


Figure 13: LPN Example 2

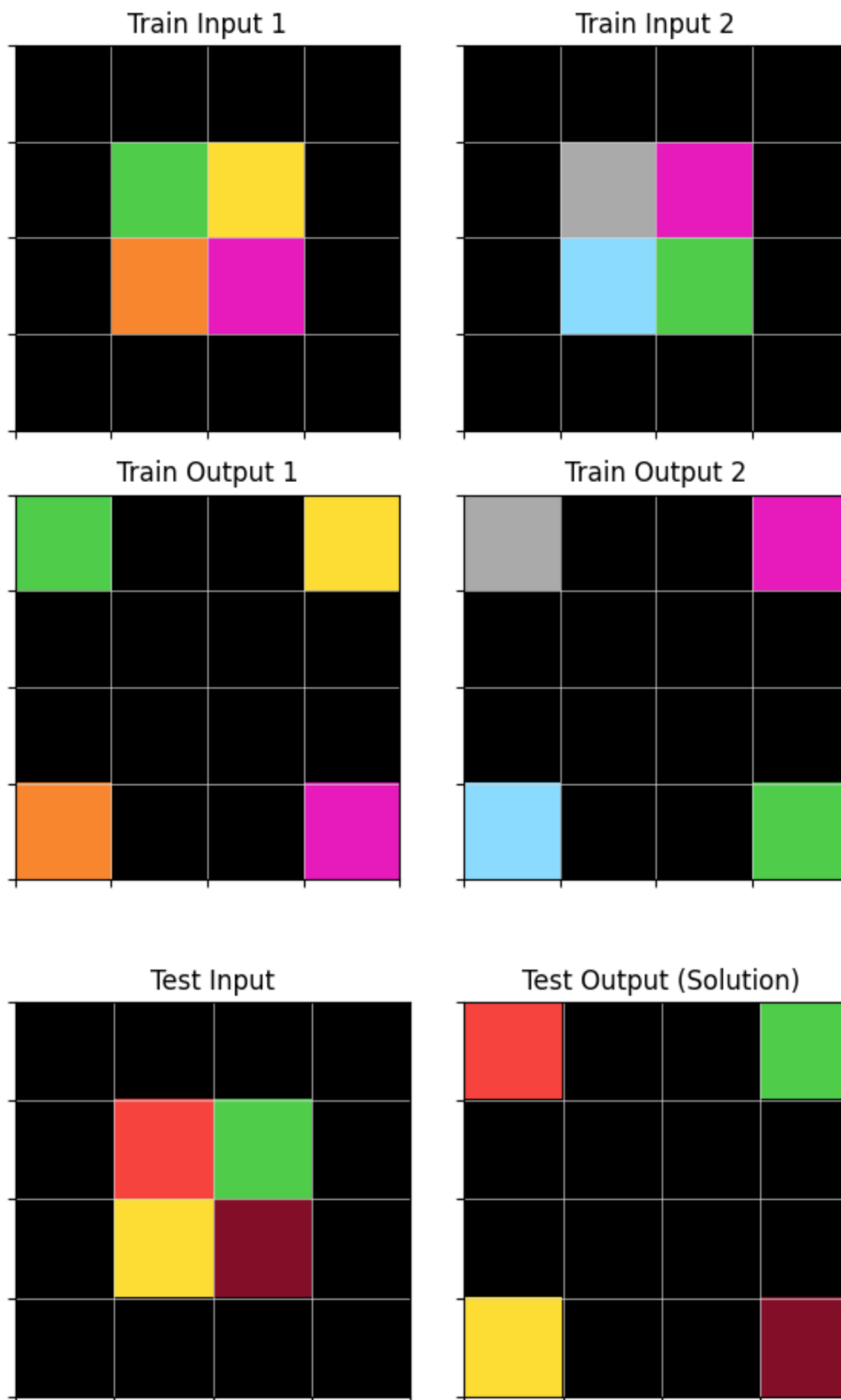


Figure 14: TTT Example 1

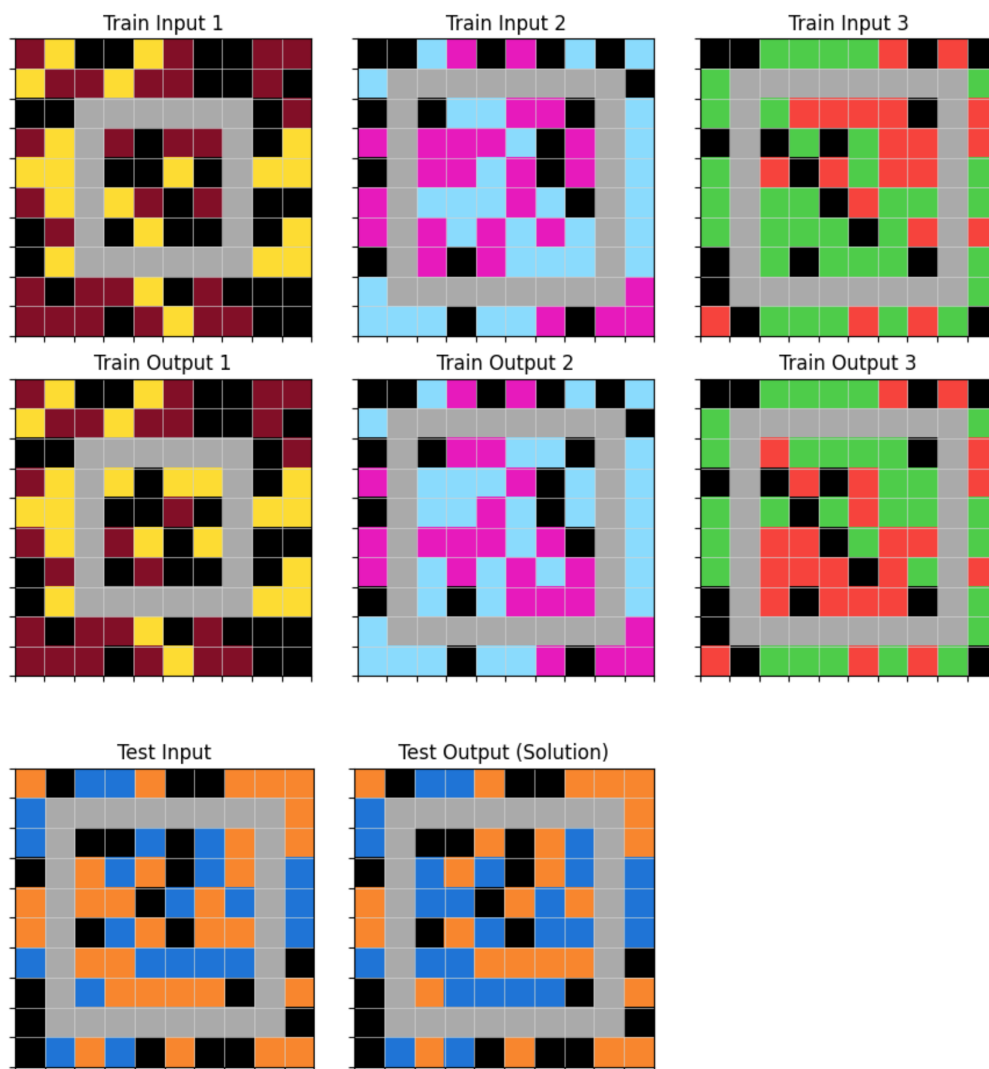


Figure 15: TTT Example 2

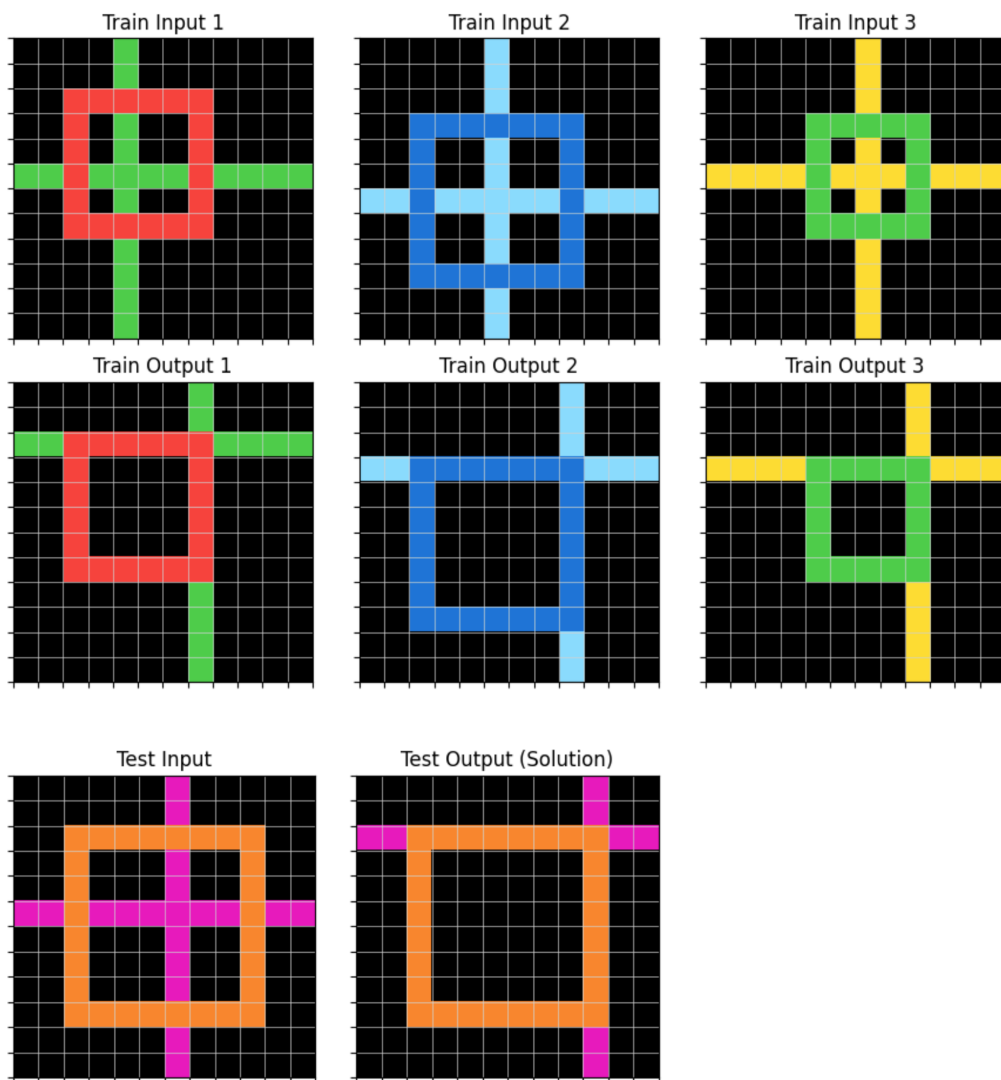


Figure 16: TTT Example 3

729 C LPN Algorithm

730 Below we outline two algorithms: first, LPN test-time inference (Algorithm 1) and its mechanism
 731 for performing inductive inference. Second, we provide the full algorithm for LPN during training
 732 (Algorithm 2).

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

Require: n input-output pairs (x_i, y_i) , a test input x_{n+1} , number of gradient steps K

- 1: **for** $i = 1, \dots, n$ **do** ▷ Can be done in parallel
- 2: Sample $z_i \sim q_\phi(z|x_i, y_i)$
- 3: **end for**
- 4: Initialize latent $z' \leftarrow \frac{1}{n} \sum_{i=1}^n z_i$
- 5: **for** $k = 1, \dots, K$ **do** ▷ Perform gradient ascent
- 6: $z' \leftarrow z' + \alpha \cdot \nabla_z \sum_{i=1}^n \log p_\theta(y_i|x_i, z)|_{z=z'}$
- 7: **end for**
- 8: **return** $y_{n+1} \sim p_\theta(y|x_{n+1}, z')$

Algorithm 2 LPN Training with Gradient Ascent Latent Optimization

Require: Encoder parameters ϕ , decoder parameters θ

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

733 D Variational Inference

734 Latent Program Networks (LPNs), when operated in grad 0 mode, perform a form of variational
735 inference. Updating the latent representation using gradients from the encoder constitutes semi-
736 amortized variational inference [Kim et al., 2018, Marino et al., 2018].

737 In prior work, such as LEAPs Trivedi et al. [2021], it is assumed during training that the full
738 representation of a program f is observable. In this setting, variational inference on programs can be
739 conducted using a standard Variational Autoencoder (VAE) restricted to the program space. LEAPs
740 introduce a form of variational inference via an Execution Evidence Lower Bound (ELBO), where
741 function reconstruction is based on correctly executing the function for a given input rather than
742 reconstructing the full program representation. The Execution ELBO is defined as:

$$\mathbb{E}_{z \sim q_\phi(z|f)} [\mathbb{E}_{x \sim p(x)} [\log p_\theta(y = f(x) | x, z)]] - \text{KL}(q_\phi(z|f) \parallel p(z)).$$

743 This formulation is advantageous because it enables learning a function that directly executes the
744 program without requiring explicit reconstruction of the program followed by execution. In LPNs,
745 this is critical, as the differentiable parameterization of the function executor allows backpropagation
746 through the executor to perform program search in the latent space.

747 In this work, we assume that the functions generating the underlying input-output pairs are not
748 fully observable, aligning with real-world scenarios where the data-generating functions are rarely
749 fully known. Instead, we observe only partial data for each function, represented as a dataset
750 $X_d = \{(x_{i,d}, y_{i,d})\}_{i=1}^{N_d}$, where each dataset d corresponds to a set of input-output pairs generated by
751 a particular unseen function. Across D such datasets, the following objective serves as a practical
752 approximation to the Execution ELBO:

$$\sum_{d=1}^D \sum_{i=1}^{N_d} \mathbb{E}_{z \sim q_\phi(z|X_d)} [\log p_\theta(y_{i,d} | x_{i,d}, z)] - \text{KL}(q_\phi(z|X_d) \parallel p(z)), \quad (7)$$

753 where D is the number of datasets, each representing input-output pairs from a distinct unseen
754 function, and N_d is the number of samples in dataset d . However, this standard objective is flawed
755 because the encoder $q_\phi(z|X_d)$ can “cheat” by encoding specific details of the pair $(x_{i,d}, y_{i,d})$ it needs
756 to reconstruct, leading to memorization rather than capturing the general function f . To address this,
757 we propose the Leave-One-Out (LOO) objective:

$$\sum_{d=1}^D \sum_{i=1}^{N_d} \mathbb{E}_{z \sim q_\phi(z|X_{-i,d})} [\log p_\theta(y_{i,d} | x_{i,d}, z)] - \text{KL}(q_\phi(z|X_d) \parallel p(z)), \quad (8)$$

758 where $X_{-i,d} = X_d \setminus \{(x_{i,d}, y_{i,d})\}$ denotes the dataset X_d excluding the i -th input-output pair.
759 This objective prevents memorization by denying the encoder access to $(x_{i,d}, y_{i,d})$ when producing
760 the latent representation z used for its reconstruction. Consequently, q_ϕ must infer the underlying
761 function from the remaining data $X_{-i,d}$, making the LOO objective a better functional approximation
762 of the Execution ELBO’s intent by directly promoting generalization. Training optimizes the encoder
763 $q_\phi(z|X_d)$ and decoder $p_\theta(y | x, z)$ across multiple datasets, resulting in an amortized inference
764 model q_ϕ that efficiently proposes an approximate posterior for any given dataset X_d .

765 For optimal performance on a specific test dataset X_{test} , we employ semi-amortized variational
766 inference. First, the trained amortized encoder $q_\phi(z|X_{\text{test}})$ rapidly generates an initial latent rep-
767 resentation z_0 . Then, with the encoder parameters ϕ and decoder parameters θ fixed, we perform
768 instance-specific optimization starting from z_0 to find a refined latent representation z^* . This is
769 achieved by directly maximizing the ELBO via latent z . This combination of an efficient amortized
770 proposal and instance-specific ELBO refinement yields a superior latent program representation z^*
771 tailored to the test problem. This procedure, known as semi-amortized variational inference, balances
772 computational efficiency with high-quality inference.

E Hyperparameters

In this section, we outline our approach to hyperparameter search and provide full documentation of all the hyperparameters used in all reported experiments.

Hyperparameter Search To ensure a fair comparison between LPN and in-context learning, which share the same core architecture for embedding inputs and generating outputs, we kept all architectural parameters identical across methods. We conducted hyperparameter testing to determine whether to use rotational embeddings. We performed testing by repeating the decoder validation experiment with and without rotational embeddings. Rotational embeddings improved performance across all individual tasks and so was selected to be used in both the encoder and decoder of LPN.

For the ARC-AGI results, we performed a hyperparameter search over the learning rate for test time adaptation for both test-time tuning and LPN. Since these methods operate in different spaces, they are unlikely to require similar parameters, making it fairer to tune this parameter independently for each baseline. We validated on a held-out test set of unseen problems from the RE-ARC dataset. We searched over learning rates ranging from 10^{-1} to 10^{-7} . Performance was evaluated by measuring average accuracy across five FLOP measures (ranging from $2E+11$ to $2E+15$). For the baseline pattern task in Table 1 we choose a learning rate of 0.1 without performing hyperparameter optimisation as the experiment is simply to understand the behavior of LPN and not to optimize performance. For pattern OOD task we perform grid search for LPN and TTT over a dataset of from the strongly OOD task for 10 gradient steps, filtering out learning rates that do not decrease the test-time loss by at least 1.0%. We use these learning rates for the scaling specification size ablation also.

Validating the Decoder In section B.3, we train an LPN model on each of the re-arc generators corresponding to the first 5 tasks from the training set. For each task, we train the model for 10k gradient steps with a batch size of 128 and 4 pairs per specification, resulting in 5,120,000 input-output pairs. We gather all hyperparameters in table 11, common to all the tasks except 045e512c which had 4 encoder layers, 8 heads, 32 embedding dimensions per head, an MLP factor of 2.0, for a total of 8.7M parameters.

Component	Hyperparameter	Value
Encoder Transformer	Number of Layers	0
	Number of Heads	6
	Embedding Dimension per Head	16
	Latent Dimension	32
	RoPE	False
Decoder Transformer	Number of Layers	3
	Number of Heads	6
	Embedding Dimension per Head	16
	MLP Dimension Factor	1.0
	RoPE	False
Training	Number of Parameters	829k
	Training Steps	10k
	Batch Size	128
	Optimizer	AdamW
	Gradient Clipping Norm	1.0
	Learning Rate	4e-4
	Number of Rows & Columns	30, 30

Table 11: Hyperparameters for the experiments from section B.3, validating the decoder.

799 **Pattern Task** In section 5.2, we train an LPN model on a 10x10 task with 4x4 patterns. We train
800 each method (mean, gradient ascent, etc) for a total of 20k steps with a batch size of 128 and 4 pairs
801 per specification, resulting in a total of 10M input-output pairs.

Component	Hyperparameter	Value
Encoder Transformer	Number of Layers	2
	Number of Heads	6
	Embedding Dimension per Head	16
	MLP Dimension Factor	1.0
	Latent Dimension	32
	RoPE	False
Decoder Transformer	Number of Layers	2
	Number of Heads	6
	Embedding Dimension per Head	16
	MLP Dimension Factor	1.0
	RoPE	False
Training	Number of Parameters	973k
	Training Steps	20k
	Batch Size	128
	Prior KL Coeff	1e-4
	Optimizer	AdamW
	Gradient Clipping Norm	1.0
	Learning Rate	4e-4
	Number of Rows & Columns	10, 10
Testing	TTT Learning Rate	1e-4
	LPN Learning Rate	1e-1
	Optimizer	Adam

Table 12: Hyperparameters for the experiments from section 5.2, i.e. the pattern task.

802 **Analyzing the Latent Space** In section B.4, we train a small LPN model on a reduced version of the
803 *Pattern* task with grids of size 4x4 and patterns of size 2x2. We used the following hyperparameters
804 for training in table 13.

Component	Hyperparameter	Value
Encoder Transformer	Number of Layers	2
	Number of Heads	6
	Embedding Dimension per Head	12
	MLP Dimension Factor	4.0
	Latent Dimension	2
	RoPE	False
Decoder Transformer	Number of Layers	2
	Number of Heads	6
	Embedding Dimension per Head	12
	MLP Dimension Factor	4.0
	RoPE	False
Training	Number of Parameters	1M
	Training Steps	200k
	Batch Size	128
	Prior KL Coeff	1e-3
	Optimizer	AdamW
	Gradient Clipping Norm	1.0
	Learning Rate	4e-4
	Number of Rows & Columns	4, 4
Testing	TTT Learning Rate	1e-4
	LPN Learning Rate	1e-1
	Optimizer	Adam

Table 13: Hyperparameters for the experiment in section B.4, i.e. analyzing the latent space.

805 **Out-Of-Distribution** In section 5.4, we train LPN models similar to those above in the *Pattern task*
806 and evaluate them on different distributions. We gather hyperparameters used for training in table 14.

Component	Hyperparameter	Value
Encoder Transformer	Number of Layers	4
	Number of Heads	8
	Embedding Dimension per Head	8
	MLP Dimension Factor	2.0
	Latent Dimension	32
	RoPE	False
Decoder Transformer	Number of Layers	2
	Number of Heads	8
	Embedding Dimension per Head	4
	MLP Dimension Factor	1.0
	RoPE	False
Training	Number of Parameters	1M
	Training Steps	100k
	Batch Size	128
	Prior KL Coeff	1e-4
	Optimizer	AdamW
	Gradient Clipping Norm	1.0
	Learning Rate	4e-4
Testing	Number of Rows & Columns	10, 10
	TTT Learning Rate	1e-5
	LPN Learning Rate	1e-1
	Optimizer	Adam

Table 14: Hyperparameters for the experiments from section 5.4, i.e. the study of out-of-distribution performance of LPN on the Pattern task.

807 **ARC-AGI** In table 15, we finally present the hyperparameters used for experiments on ARC-AGI
808 (section 5.6).

Component	Hyperparameter	Value
Encoder Transformer	Number of Layers	8
	Number of Heads	8
	Embedding Dimension per Head	64
	MLP Dimension Factor	4.0
	Latent Dimension	256
	RoPE	True
	RoPE max freq	10
Decoder Transformer	Number of Layers	6
	Number of Heads	8
	Embedding Dimension per Head	64
	MLP Dimension Factor	4.0
	RoPE	True
	RoPE max freq	10
Training	Number of Parameters	178M
	Training Steps	100k
	Batch Size	256
	Prior KL Coeff	1e-4
	Optimizer	AdamW
	Gradient Clipping Norm	1.0
	Learning Rate	3e-4
	Number of Rows & Columns	30, 30
Testing	TTT Learning Rate	1e-4
	LPN Learning Rate	5e-2
	Optimizer	Adam

Table 15: Hyperparameters for the experiment in section 5.6, i.e. training LPN to solve the ARC-AGI benchmark.

809 F Additional Charts

810 F.1 Latent Program Embeddings

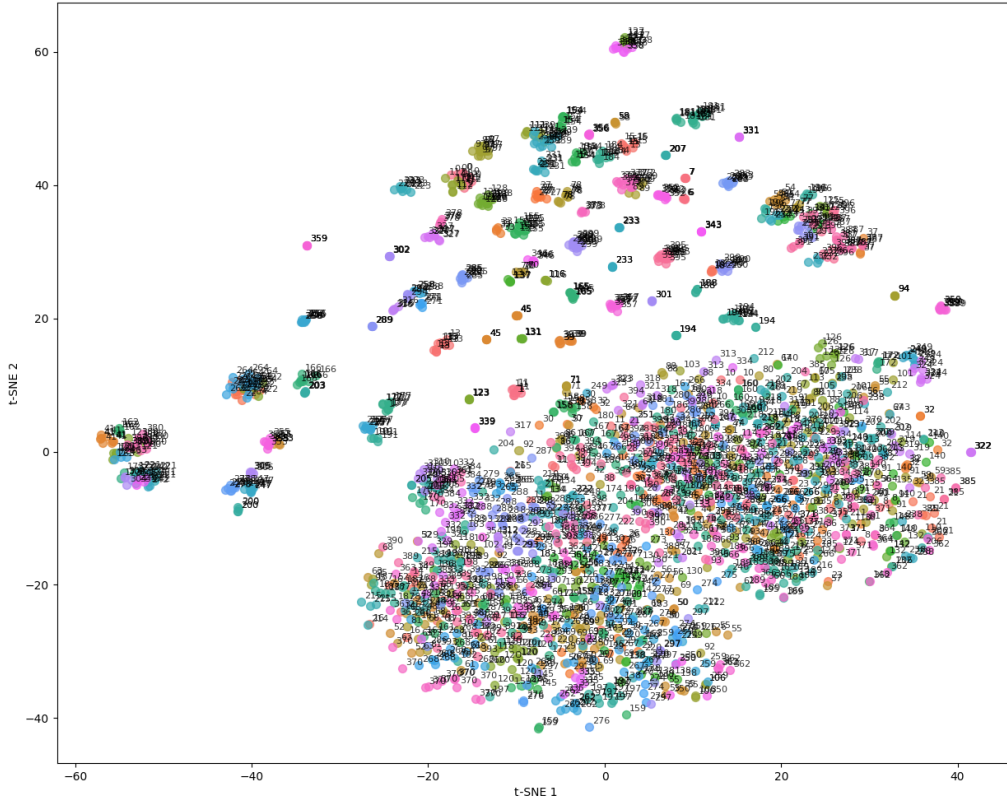


Figure 17: T-SNE visualization of the latent space of input-output pairs sampled from the re-arc generators. We see strong evidence that the latent embeddings encode information about programs with significant clustering in the latent space for the same programs across different input output pairs.

E.2 Decoder Gradient Field

Figure 18 visualizes the likelihood landscape of decoding the correct output conditioned on the given input for a single pattern task, while varying the latent input. The gradient contours overlaid on the plot illustrate the optimization dynamics when performing gradient-based updates in this space. The trajectories depict how gradient ascent seeks to maximize the decoding likelihood, revealing the structure of the landscape. Certain regions form basins of attraction that lead to valid solutions, while others correspond to local optima where the likelihood of decoding the correct output stagnates. This visualization highlights both the feasibility of optimizing the latent representation and the potential challenges of escaping suboptimal regions in the latent space.

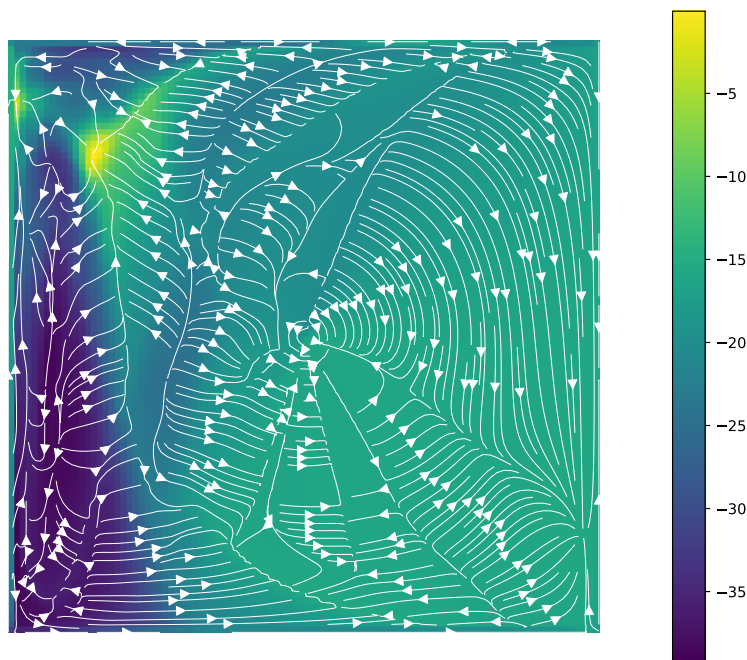


Figure 18: Visualization of the likelihood landscape for decoding the correct output conditioned on the input, as a function of the latent space.

G Architecture

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 have shapes (n, m) with $n, m \in [1, 30]$. 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].

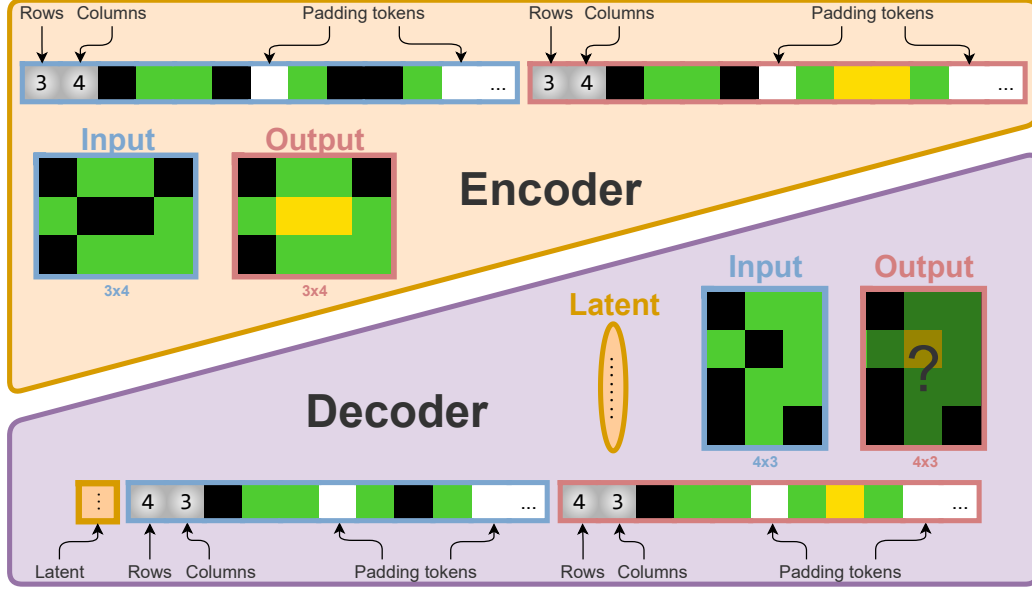


Figure 19: LPN architecture for ARC-AGI. Both the encoder and decoder are small transformers that take flattened padded grids as inputs. The actual number of rows and columns is prefixed to each sequence.

We model the input and output images as 2D grids, which we pad and flatten in a raster-scan fashion to form sequences of pixel values, each of size $30 \times 30 = 900$ (see Figure 19). Each grid sequence is prefixed with shape information, namely two extra values for the number of rows and columns, resulting in sequences of 902 values. For both the Encoder and Decoder grid positions are encoded using RoPE [Su et al., 2024]¹ for both row and column indices.

G.1 Encoder

The encoder processes both the input and output grids from a given pair and returns a distribution of inferred program latents underlying the task. Specifically, it outputs the mean and diagonal covariance of a multivariate normal distribution from which latents can be sampled. Each grid sequence contains 902 values, and we add an extra CLS token for the output embedding, resulting in a total sequence length of 1805 for the encoder transformer. The encoder is implemented as a standard transformer [Vaswani et al., 2017] with pre-layer normalization [Baevski and Auli, 2018, Xiong et al., 2020] and multi-head attention. To incorporate identify the input from the output we add an embedding $\text{emb}(c)$, $c \in \{0, 1\}$ is the channel index (0 for input, 1 for output). All 1800 color values (0 to 9), four shape values (1 to 30), and the CLS token are separately embedded into \mathbb{R}^H using lookup tables. Padded tokens, determined by the shape values, are masked, and the sequence is fed into multiple transformer blocks, see Appendix E for hyperparameter details. In the encoder the attention mask is non-causal, allowing all non-padded tokens to attend to each other during encoding. The CLS embedding is passed through a layer normalization and two parallel dense layers to output the mean and diagonal log-covariance of the multivariate normal distribution over latents. Sampled program latents have dimension d , which may differ from the embedding dimension H .

¹<https://github.com/crowsonkb/rope-flax>

847 **G.2 Decoder**

848 The decoder takes an input grid and a latent program and autoregressively generates an output grid.
849 Its design is similar to the encoder, with key differences. First, the flattened sequence is prefixed with
850 a projection of the latent embedding. Since the decoder generates the output autoregressively, the
851 attention mask is causal on the output grid portion of the sequence (the second half). The attention
852 mask also dynamically accounts for padding tokens based on the predicted output shapes. The
853 sequence embeddings corresponding to the output are extracted and projected to either shape logits
854 for the first two embeddings or grid logits for the 900 output grid embeddings. Each output token
855 embedding maps to logits for the next token in a raster-scan fashion. However, due to padding at
856 each row, the last embedding of each row is mapped to the first token of the next row.

857 H Baselines

858 H.1 Transductive Baseline

859 We compare LPN against a transductive baseline that directly conditions on the specification without
 860 explicitly constructing a latent program representation. This approach, similar to Kolev et al. [2020],
 861 Li et al. [2024a], processes the specification by encoding and concatenating each input-output pair
 862 separately.

863 **Encoder:** The encoder maps each input-output pair to an encoding vector:

$$z_i = e_\phi(x_i, y_i) \quad \forall i \in [1, n] \quad (9)$$

864 where e_ϕ is a neural network parameterized by ϕ that processes individual input-output pairs.

865 **Concatenation:** Unlike LPN which searches for a single latent program, the transductive baseline
 866 uses each encoding as a token embedding in the input sequence to the transformer. Specifically, the
 867 encodings are concatenated as:

$$z_{\text{cat}} = [z_1; z_2; \dots; z_n] \quad (10)$$

868 where $[\cdot]$ denotes sequence concatenation and each z_i serves as a token embedding in the transformer’s
 869 input sequence.

870 **Decoder:** The transformer decoder processes this sequence of embeddings along with the new input
 871 to generate the output:

$$\hat{y}_{n+1} \sim p_\theta(y|x_{n+1}, z_{\text{cat}}) \quad (11)$$

872 where the decoder attends to both the new input x_{n+1} and the sequence of specification embeddings
 873 z_{cat} .

874 By concatenating per-pair embeddings, we ensure a joint representation of all input-output pairs,
 875 allowing the decoder to access information from all grids during inference. Processing input-output
 876 pairs independently in the encoder serves as a strong prior for capturing high-level program features,
 877 reducing the risk of learning spurious correlations between pixels of different pairs. In contrast,
 878 methods that process all raw pairs jointly increase the encoder’s computational demands. Additionally,
 879 feeding all raw pairs directly to the decoder would complicate positional encodings, requiring
 880 simultaneous modeling of both within-pair and across-pair positions. Our approach mitigates this
 881 by encoding positional information separately within each pair at the encoder stage. The decoder
 882 then receives each pair’s embedding in a distinct position within the concatenated sequence, enabling
 883 clear differentiation between examples.

884 H.2 Test-Time Fine-Tuning

885 We implement the following test-time parameter tuning approach where the transductive model’s
 886 parameters are fine-tuned on the specification itself. Given a specification of n input-output pairs, we
 887 perform gradient updates on the model parameters to better predict each output given its input and
 888 the remaining pairs.

889 **Update Process:** Starting from the pre-trained parameters θ and ϕ (decoder and encoder respectively),
 890 for each pair (x_i, y_i) in the specification, we compute the loss:

$$\mathcal{L}_{\text{TT}}^i(\phi, \theta) = -\log p_\theta(y_i|x_i, z_{\text{cat}}^{-i}) \quad (12)$$

891 where $z_{\text{cat}}^{-i} = [e_\phi(x_1, y_1); \dots; e_\phi(x_{i-1}, y_{i-1}); e_\phi(x_{i+1}, y_{i+1}); \dots; e_\phi(x_n, y_n)]$ represents the con-
 892 catenated embeddings of all pairs except the i -th.

893 We then update the parameters using gradient descent:

$$\phi' = \phi - \alpha \nabla_\phi \sum_{i=1}^n \mathcal{L}_{\text{TT}}^i(\phi, \theta) \quad (13)$$

$$\theta' = \theta - \alpha \nabla_\theta \sum_{i=1}^n \mathcal{L}_{\text{TT}}^i(\phi, \theta) \quad (14)$$

895 where α is the learning rate for test-time adaptation.

896 **Inference:** After K steps of parameter updates, we use the tuned parameters ϕ' and θ' to make
 897 predictions on new inputs. The prediction process remains the same as the transductive baseline but
 898 uses the adapted parameters:

$$\hat{y}_{n+1} \sim p_{\theta'}(y|x_{n+1}, [e_{\phi'}(x_1, y_1); \dots; e_{\phi'}(x_n, y_n)]) \quad (15)$$

899