INDUCING UNSEEN PROGRAMS AS TRANSFORMATIONS IN A LEARNED REPRESENTATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Program induction uses input–output examples to infer how an unknown program behaves on further inputs. We ask whether a shared representation of data can make this inference possible for programs absent from training. We introduce Joint-Embedding Program Inductive Architecture (JEPIA), which encodes a task's input–output examples and fits a transformation between their embeddings. The fitted transformation predicts output embeddings for further inputs. Errors on these predictions train the shared encoder through the fitting procedure, so the representation is learned for the inference it must support. We instantiate JEPIA with rotations and reflections, which admit an analytical fit, and regularize the embeddings against collapse. In modular addition, digit embeddings acquire a ring structure in numerical order, providing an interpretation of how the model predicts held-out additions. On cube transformations, the model transfers from two-move training programs to programs generated by shorter and longer sequences. Additional context examples and broader exposure to training programs improve held-out prediction. These results support learning data representations that allow a common fitting procedure to infer new programs from examples.

1 INTRODUCTION

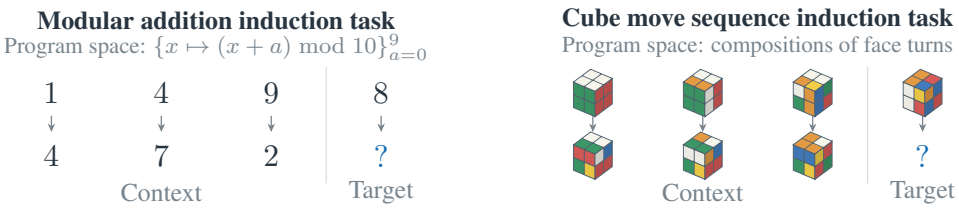


Figure 1: **Examples of program induction tasks.** Each task is constructed by sampling a program from the program space and collecting input–output examples, partitioned into context and target sets. The goal is to predict the target outputs given the context pairs and target inputs.

Given a few input–output examples from an unknown program, a program induction task asks a model to predict that program's outputs on further inputs (Alet et al., 2021). Figure 1 illustrates two such tasks. In the setting studied here, a model is trained on a collection of tasks and evaluated on tasks generated from programs absent from training. Memorizing the training programs is therefore insufficient: the model must use examples to infer how an unfamiliar program behaves. Related problems have been studied across fields under different names and assumptions; Appendix A discusses their relationship to our setting.

Program induction can be viewed as a search for a program that explains the given examples and predicts outputs on further inputs. The effectiveness and efficiency of this search depend jointly on how the program space is structured and how the search procedure uses that structure. Learning can help by building a library of reusable program components (Ellis et al., 2021), or by constructing a latent program space that supports gradient-based search at inference time (Macfarlane & Bonnet, 2025). Yet a vast program space can make search prohibitively expensive, and its learned structure

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

must support programs absent from training. Moreover, a few examples may be consistent with several programs that disagree on further inputs. Restricting the program space can make search tractable and reduce this ambiguity, but may exclude programs of interest. The challenge is to make inference tractable while retaining the capacity to infer unfamiliar programs.

We introduce Joint-Embedding Program Inductive Architecture (**JEPIA**), which learns to organize data representations so that programs can be inferred and applied as simple transformations of embeddings (Figure 2). A shared encoder maps each input and output to an embedding. For each task, we fit a *latent program*, a transformation from a prescribed family, to the context embeddings. We then apply it to target input embeddings to predict the corresponding output embeddings. Following Joint-Embedding Predictive Architectures (JEPA), we measure prediction error in representation space (Assran et al., 2023). This error trains the encoder through the context fit, alongside representation regularization. Training thus encourages a geometry in which a latent program fitted to a few examples also predicts further examples. At evaluation, the encoder is fixed, and each task supplies a new context from which to induce a latent program.

Modular addition makes this mechanism interpretable. We train on a subset of additions modulo ten and evaluate on held-out additions, using the same ten digits throughout. As JEPIA learns to solve the training tasks, the digit embeddings develop a numerically ordered ring structure. This geometry makes additions interpretable as rotations and supports induction of held-out additions without further encoder training. The experiment shows both that the learned representation supports unseen programs and how its organization enables this transfer.

We next use Rubik's cube move sequences to examine generalization within and across program composition lengths. Cube moves compose into programs of controlled length, and their effect depends on move order. Training on a subset of two-move programs supports prediction on held-out programs generated by sequences of the same, shorter, and longer lengths. We also examine how prediction improves with more context examples per task and a larger proportion of the program space used during training, demonstrating that JEPIA can use additional information when available. These experiments support learning shared data representations as a basis for induction of unseen programs.

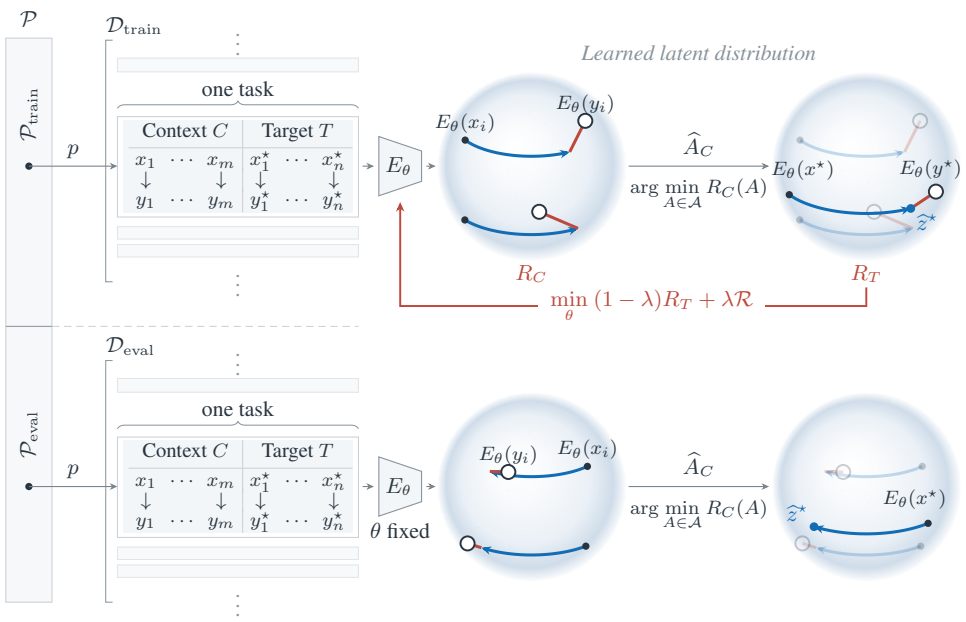


Figure 2: **JEPIA training and evaluation.** Disjoint training and evaluation program spaces generate the respective datasets. For each task, a shared encoder embeds the inputs and outputs. A latent program induced from the context embeddings maps target input embeddings to predicted output embeddings. Target prediction error and regularization train the encoder. Evaluation keeps the encoder fixed and tests induction on programs unseen during training.

2 BACKGROUND

2.1 PROGRAM INDUCTION

We consider program induction through prediction from input–output examples, a general setting used to encompass explicit program inference and direct output prediction (Devlin et al., 2017; Alet et al., 2021). Here we describe the tasks and datasets used in this work; Appendix A relates this formulation to other settings adopted across the literatures.

Let $\mathcal{X}$ be a data space and $\mathcal{P}$ a space of programs, each specifying a mapping $p : \mathcal{X} \to \mathcal{X}$. A task is constructed by sampling a program and a collection of its input–output pairs (x, y), with $y = p(x)$. We divide the examples into a *context set* C and a *target set* T:

$$C = \{(x_i, y_i)\}_{i=1}^{m}, \qquad T = \{(x_j^\star, y_j^\star)\}_{j=1}^{n}. \tag{1}$$

The model receives the context pairs and target inputs, and predicts the target outputs:

$$\widehat{y_j^\star} = f(C, x_j^\star). \tag{2}$$

The context is the model's evidence about the implied program. Target outputs supply supervision during training and are used to score predictions during evaluation. In the addition example from the introduction, $1 \mapsto 4$ and $4 \mapsto 7$ could form the context, while $8 \mapsto 1$ is a target pair: the target input is 8 and the correct output is 1. The addend identifies the underlying program, while the examples provide the information used to infer it.

A dataset is a collection of tasks. Training and evaluation datasets are constructed from disjoint subsets of programs,

$$\mathcal{P}_{\text{train}} \cap \mathcal{P}_{\text{eval}} = \emptyset. \tag{3}$$

For modular addition, this means holding out addends: the digit 3 can occur in training data even when the program "add three modulo ten" is reserved for evaluation. The split therefore tests new relationships among data values that may already be familiar. Within each evaluation task, the model must also predict target outputs from the available context. Appendix C gives the sampling formulation and the treatment of equivalent programs.

2.2 JOINT-EMBEDDING PREDICTIVE ARCHITECTURES

To learn representations from these relationships, we build on prediction in representation space. In a Joint-Embedding Predictive Architecture (JEPA), an encoder maps an observation to an embedding, and a predictor estimates the embedding of a related observation (LeCun, 2022). The prediction is compared with an encoding of the observed target. For example, I-JEPA predicts embeddings of target image regions from a context region, using the target locations to specify what to predict (Assran et al., 2023). The resulting loss measures prediction error between representations.

Because these representations are learned, the prediction objective admits a trivial solution: every observation could be mapped to the same vector, which would be easy to predict. This is *representation collapse*. A useful learning objective must also encourage embeddings that retain distinctions between observations.

LeJEPA combines a prediction loss with a regularizer called Sketched Isotropic Gaussian Regularization (SIGReg) (Balestriero & LeCun, 2025). SIGReg acts on a collection of embeddings, encouraging their distribution to match a d-dimensional standard isotropic Gaussian, $\mathcal{N}(0, I_d)$. It compares the embeddings projected onto sampled directions with a one-dimensional standard Gaussian. The prediction loss constrains the relationships between paired observations, while the regularizer encourages the collection of embeddings to remain spread across the representation space.

The geometry of the Gaussian reference will also guide our choice of latent programs. It has no preferred direction: rotating or reflecting the space leaves the distribution unchanged. In high dimensions, most of its probability mass lies in a shell whose thickness is small relative to its radius, approximately $\sqrt{d}$. Rotations and reflections preserve this radial structure, giving a family of transformations compatible with the reference distribution. We return to this choice after introducing the general architecture. Figure 2 summarizes the prediction and regularization principles; JEPIA uses them with a predictor inferred from the context of each task.

3 JOINT-EMBEDDING PROGRAM INDUCTIVE ARCHITECTURE

JEPIA learns a shared representation of data and fits a new latent program within that representation for each task. We first describe how a task is solved, then how predictions across tasks train the encoder.

3.1 GENERAL ARCHITECTURE

Let $E_\theta : \mathcal{X} \to \mathbb{R}^d$ be an encoder applied independently to each input and output, with parameters shared across tasks. We choose a family $\mathcal{A}$ of transformations $A : \mathbb{R}^d \to \mathbb{R}^d$, called *latent programs*. For a pair (x, y) from a program, the relationship we seek in the representation space is

$$A(E_\theta(x)) \approx E_\theta(y). \tag{4}$$

The encoder places data from all tasks in one representation, where different programs can relate the same embeddings through different transformations.

Within a task, the context pairs C provide evidence about which latent program to use. We fit one by minimizing its empirical prediction error on these m context pairs:

$$\widehat{A}_C \in \arg\min_{A \in \mathcal{A}} \frac{1}{m} \sum_{(x_i, y_i) \in C} \ell_{\mathcal{Z}}\big(A(E_\theta(x_i)), E_\theta(y_i)\big), \tag{5}$$

where latent loss $\ell_{\mathcal{Z}}$ measures error between embeddings. We then apply this fitted latent program $\widehat{A}_C$ to each target-input embedding. A shared decoder $D_\phi : \mathbb{R}^d \to \mathcal{X}$ converts the predicted embedding $\widehat{z}_j^\star$ into an output $\widehat{y}_j^\star$:

$$\widehat{z}_j^\star = \widehat{A}_C\big(E_\theta(x_j^\star)\big), \qquad \widehat{y}_j^\star = D_\phi(\widehat{z}_j^\star). \tag{6}$$

Thus, solving a task consists of encoding its examples, fitting a latent program from context, and using that fit to predict and decode the target outputs. The same fitted program is used for all targets in the task.

Learning across tasks should make this procedure useful beyond the pairs used for fitting. We therefore train the encoder with the error on n target pairs excluded from the context fit:

$$\mathcal{L}_{\text{pred}}(C, T) = \frac{1}{n} \sum_{(x_j^\star, y_j^\star) \in T} \ell_{\mathcal{Z}}\big(\widehat{A}_C(E_\theta(x_j^\star)), E_\theta(y_j^\star)\big). \tag{7}$$

Changing the encoder changes both the embeddings and the latent program fitted to the context. We differentiate through this fit, as well as through the target-input and target-output encodings. An encoder update can therefore change how subsequent context fits extend to target inputs. Across tasks, the objective encourages a representation in which programs can be inferred from context and used to predict further examples. Appendix B.1 describes the assumptions on the fitting procedure and its differentiation.

To discourage representation collapse, we add a representation regularizer $\mathcal{R}$:

$$\mathcal{L}_{\text{enc}} = (1 - \lambda)\mathcal{L}_{\text{pred}} + \lambda \mathcal{R}(Z), \qquad 0 < \lambda < 1. \tag{8}$$

Here Z contains the encoded inputs and observed outputs from the context and target pairs. Target prediction trains the encoder to make a context fit useful on further examples, while regularization discourages collapse of the embedding collection. Algorithm 1 summarizes encoder training for individual tasks. Appendix B specifies the batch loss and regularization used in the implementation.

An auxiliary decoder is trained separately to recover outputs from their observed embeddings, with its gradients stopped at the encoder. The encoder is updated by Equation 8, so its representation is learned through latent prediction and regularization. At evaluation, both encoder and decoder parameters are fixed; each new context supplies a fresh latent fit. Figure 2 summarizes the architecture, and Appendix B gives the decoder objective.

Algorithm 1 Training JEPIA with orthogonal latent programs

Require: Training tasks $\mathcal{D}_{\text{train}}$; encoder $E_\theta : \mathcal{X} \to \mathbb{R}^d$; weight λ; learning rate η

for all tasks $(C, T) \in \mathcal{D}_{\text{train}}$ **do**

$z_x \leftarrow E_\theta(x)$, $z_y \leftarrow E_\theta(y)$ for each pair $(x, y) \in C \cup T$ ▷ Shared encoder

$X_C \leftarrow [z_x]_{(x,y)\in C}$, $Y_C \leftarrow [z_y]_{(x,y)\in C}$ ▷ Context embeddings as columns

$U\Sigma V^\top \leftarrow \mathrm{SVD}(Y_C X_C^\top)$, $\widehat{Q}_C \leftarrow UV^\top$ ▷ Fit latent program to context

$\widehat{z}_y \leftarrow \widehat{Q}_C\, z_x$ for each $(x, y) \in T$ ▷ Apply it to target inputs

$\mathcal{L}_{\text{pred}} \leftarrow \dfrac{1}{|T|} \displaystyle\sum_{(x,y)\in T} \dfrac{1}{d} \big\|\widehat{z}_y - z_y\big\|_2^2$ ▷ Target error in latent space

$\mathcal{L} \leftarrow (1-\lambda)\,\mathcal{L}_{\text{pred}} + \lambda\ \mathrm{SIGReg}\big(\{z_x, z_y\}_{(x,y)\in C\cup T}\big)$ ▷ Prevent collapse

$\theta \leftarrow \theta - \eta \nabla_\theta \mathcal{L}$ ▷ Backpropagate through the fit

end for

3.2 An instance with orthogonal latent programs

We instantiate the architecture with SIGReg as the representation regularizer $\mathcal{R}$ (Balestriero & LeCun, 2025) and a family of latent programs compatible with its isotropic Gaussian reference. Rotations and reflections move embeddings while preserving their distances from the origin and from one another. They also preserve the Gaussian reference distribution described in Section 2.2. Compositions of rotations and reflections form the *orthogonal* family. For column-vector embeddings, these transformations are matrices Q satisfying $Q^\top Q = I_d$, acting as $z \mapsto Qz$. We choose this family and squared prediction error:

$$\mathcal{A} = \{z \mapsto Qz : Q^\top Q = I_d\}, \qquad \ell_{\mathcal{Z}}(u, v) = \frac{1}{d}\|u - v\|_2^2. \tag{9}$$

SIGReg encourages the aggregate embedding distribution to remain spread out. The prediction objective encourages arrangements in which an orthogonal transformation fitted to context also predicts target pairs.

This choice has an analytical solution for the empirical risk minimizer on context fit. Let $X_C, Y_C \in \mathbb{R}^{d\times m}$ contain the context input and output embeddings as columns. Equation 5 becomes the orthogonal Procrustes problem (Schönemann, 1966),

$$\widehat{Q}_C \in \operatorname*{arg\,min}_{Q^\top Q = I_d} \|QX_C - Y_C\|_F^2. \tag{10}$$

A singular value decomposition $Y_C X_C^\top = U\Sigma V^\top$ gives a minimizer and its target prediction as

$$\widehat{Q}_C = UV^\top, \qquad \widehat{z}_j^\star = \widehat{Q}_C E_\theta(x_j^\star). \tag{11}$$

The fit aligns the two configurations of context embeddings using a single orthogonal transformation. Its predictions on the target inputs then train the encoder through the same objective as in the general architecture. Appendix B.2 derives the solution and discusses contexts that leave parts of the transformation undetermined.

Products of orthogonal transformations are again orthogonal, and the order of multiplication can affect the result. The family therefore accommodates latent actions whose compositions depend on order, as in the cube tasks below. Each task uses one fitted matrix for its complete input–output relationship.

4 Experiments

The experiments examine three aspects of the proposed approach. Modular addition tests induction of held-out programs and makes the learned representation interpretable. Cube transformations test generalization within and across program composition lengths. Varying the number of context pairs and training programs tests whether JEPIA can use additional information at inference and during learning. All JEPIA experiments use the orthogonal instance, with encoder and decoder parameters fixed during evaluation. Appendix C gives further details of task construction and evaluation.

4.1 INDUCTION OF UNSEEN PROGRAMS THROUGH LEARNED REPRESENTATIONS

Modular addition allows us to examine both prediction and the geometry that supports it. The data are the digits $\mathcal{X} = \{0, \ldots, 9\}$, and each program adds a fixed value modulo ten:

$$p_a(x) = (x + a) \bmod 10, \qquad a \in \{0, \ldots, 9\}. \tag{12}$$

Each run trains on nine addends and holds out the remaining one. We evaluate all ten choices of held-out addend with three training seeds each; tasks contain eight context pairs and one target pair. Digits are supplied as one-hot vectors, which identify them without providing numerical order. All ten digits can occur during training; evaluation tests the model's ability to infer new additions relating these familiar values. We measure the fraction of target digits predicted correctly.

Held-out prediction improves as the representation is learned. Figure 3 pairs this result with two-dimensional principal component analysis (PCA) views of the digit embeddings, where the digits become arranged in numerical order around a ring. Fitting and prediction use the full learned embeddings; PCA provides a view of their organization. An ideal circular representation illustrates why this geometry is useful:

$$\bar{z}(x) = \begin{pmatrix} \cos(2\pi x/10) \\ \sin(2\pi x/10) \end{pmatrix}, \qquad \bar{z}(p_a(x)) = R_{2\pi a/10}\bar{z}(x), \tag{13}$$

where R_α is a planar rotation by angle α. Every addition has the same form of latent action and differs only in the angle of rotation. The arrangement therefore makes all addends expressible in the same family, including those absent from training.

On this ideal ring, the context pairs from the opening example, $1 \mapsto 4$ and $4 \mapsto 7$, determine a rotation by three digit positions. Applying that rotation to the embedding of 8 reaches the embedding of 1. This explains how context can identify the action of a held-out addition once a suitable shared geometry is available. In the learned model, held-out prediction tests whether fitting to context is useful on further inputs. Together, the prediction results and the observed geometry support the idea that learning the arrangement of data values can support induction of new programs.

Comparison with latent program search. We compare JEPIA with the Latent Program Network (LPN; Macfarlane & Bonnet (2025)), the closest neural approach in problem formulation: it infers a latent program from input–output pairs without a symbolic language, applies it to further inputs, and refines it at inference time by gradient ascent in latent space. LPN encodes the context pairs jointly into a distribution over a continuous latent space, in which a point represents a candidate program, and a decoder predicts each target output from the target input and that point. The two architectures therefore differ in what a program is and in what trains the representation. In LPN a program is

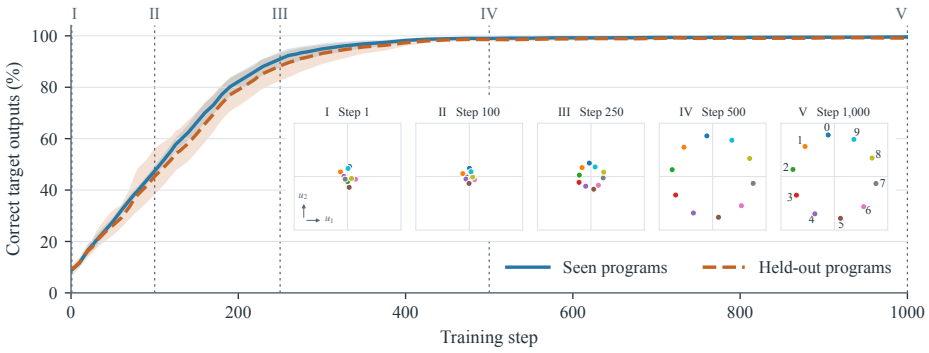


Figure 3: **Held-out addition prediction improves as the digit embeddings organize into a ring.** Curves show correct target outputs on the training task bank and on held-out programs, averaged over program partitions and training seeds with pointwise 95% bootstrap intervals. Insets show two-dimensional PCA projections of the learned digit embeddings from one selected training run at the checkpoints marked I–V, with colors indicating digits (labeled at step 1,000). As accuracy rises, the digits spread into a ring ordered by numerical value, providing a geometric interpretation of transfer to held-out addends. Latent fitting and output prediction use the full learned embeddings.

a point, and the decoder is trained to reproduce outputs in data space; nothing in the objective constrains how points between the latents of training programs decode. In JEPIA a program is a transformation between data embeddings, and the encoder is trained on the target predictions of transformations fitted to context, so the training programs shape a geometry that the same fitting procedure can reuse. Both models are scored by exact-output accuracy on decoded predictions.

Figure 4 shows LPN on the modular-addition tasks. Accuracy on the training addends reaches nearly 100%, while accuracy on held-out addends remains at or below chance throughout training. The latent space is nonetheless structured: the latents of the training programs become cyclically ordered, and the latent of a held-out addend is placed between those of its neighbors. Appendix D shows that additional gradient steps at training or inference do not change this outcome, and traces the failure to the decoder, which maps no region of the latent space to the held-out program. Because the orthogonal family is well matched to modular addition, we read this comparison as isolating the effect of representing programs as transformations rather than as points, not as a general ranking of the two methods.

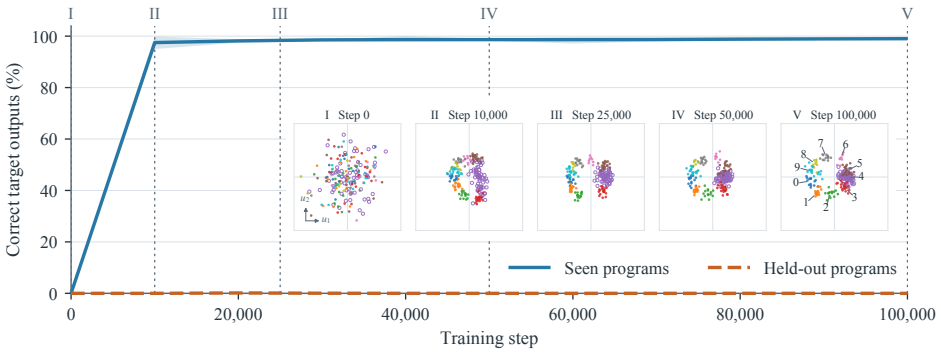


Figure 4: **LPN does not transfer to held-out additions.** Curves show correct percentage of target outputs on seen and held-out programs over training, averaged over three runs with one held-out addend each, with pointwise 95% bootstrap intervals. LPN uses five context pairs and ten targets, one gradient-ascent step during training, and mean inference during evaluation. Training lasts 100,000 updates, one hundred times the length used for JEPIA in Figure 3. Insets show two-dimensional PCA views of the latent programs from one representative run at the checkpoints marked I–V, colored by addend (labeled at step 100,000); hollow markers are tasks from the held-out addend $+4$.

4.2 Generalization within and across program composition lengths

Cube transformations let us vary the compositions used to generate programs while keeping the data format and task unchanged. A program applies a fixed sequence of moves to a Rubik's cube of size $2 \times 2 \times 2$. For example, turning the right face and then the upper face defines one program, whose context pairs show the resulting transformation of different starting states. Reversing those moves can give a different program. The domain thus provides a shared system of transformations with order-dependent composition and control over the sequences used to sample programs.

We train on 19 of the 27 distinct, nonidentity transformations generated by two quarter turns from $\{R, R^{-1}, U, U^{-1}, F, F^{-1}\}$, reserving four for validation and four for held-out evaluation. Evaluation also considers the six one-move transformations and 16 sampled transformations at each generating length from three through twenty. We identify transformations by their permutations of the 24 facelet positions, the individual colored squares on the cube, and exclude any evaluation transformation present in training. Sequence length records how the transformation was generated; several sequences, including sequences of different lengths, can have the same effect. Longer sequences are sampled without consecutive turns of the same face, and transformations are deduplicated within and across the evaluation lengths. The model fits one latent transformation from 64 context pairs and predicts eight target outputs. At each checkpoint, we evaluate the same 32 tasks per length, using held-out programs also at length two.

The learned representation supports prediction beyond the two-move training programs, including programs generated by shorter and longer sequences. Figure 5 compares performance across these lengths using facelet accuracy, the fraction of facelet colors predicted correctly across target states. The encoder, decoder, and context-fitting procedure are shared across all evaluation lengths. Varying the generating length changes how evaluation programs are sampled; the model still fits and applies a single latent transformation for each task. This experiment tests whether representations learned from two-move programs support inference of other transformations from their examples. The upper panel places these measurements in context: unrestricted sequences of these moves express 27 distinct nonidentity transformations at length two and over 1.8 million at length twenty. We obtain these counts from the exhaustive quarter-turn distance enumeration of Bryan (1993), including transformations reachable by shorter sequences of matching parity. The counts describe the full program spaces; accuracy is measured on the sampled evaluation tasks.

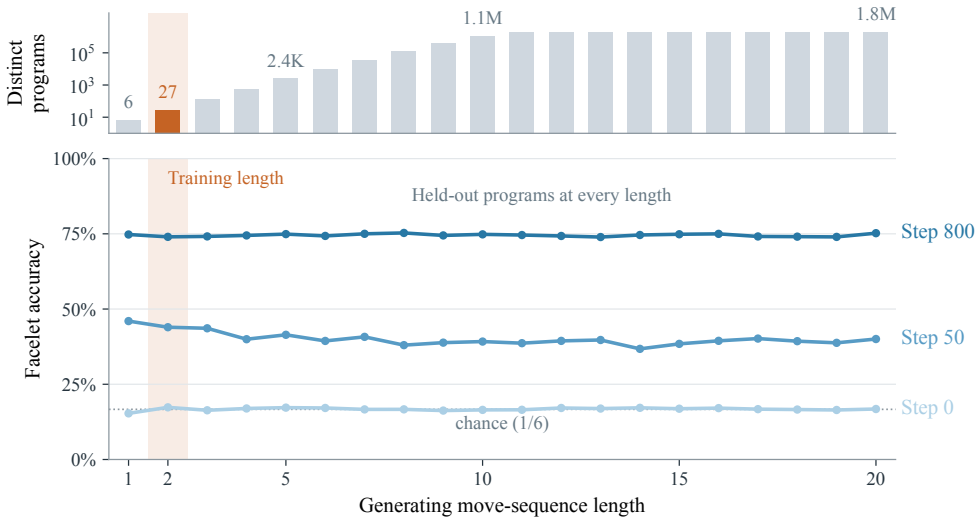


Figure 5: **Facelet prediction transfers from two-move programs to a far larger program space.** Top: number of distinct nonidentity transformations expressible by sequences of exactly the indicated length, allowing redundant sequences, from 27 at length two to 1.8 million at length twenty. Bottom: facelet accuracy on held-out programs at every length, including length two, at initialization, step 50, and step 800 of one training run. Shading marks the generating length used in training; all plotted predictions are on held-out programs. By step 800, accuracy is similar across lengths. Each task is solved by one context fit with the encoder and decoder fixed.

4.3 Using additional context and training programs

JEPIA can receive additional information at two stages. More context pairs provide observations of the program currently being inferred. More training programs provide different relationships from which to learn the shared representation. We examine both in modular addition, varying context size and the number of training addends.

To examine context size, we supply a trained model with increasingly large subsets of a context pool and keep target pairs fixed. Each context produces a new latent fit in the same representation. To examine training-program count, we train models on subsets of addends of different sizes while keeping the number of training tasks and optimizer updates fixed. Each model is evaluated on the complement of its training subset. This comparison varies the range of relationships available during training, and is averaged over program partitions because the held-out complement changes with the training subset.

The results show gains from additional context and broader training-program exposure, with diminishing gains as context grows (Figure 6). These comparisons examine two complementary capabil-

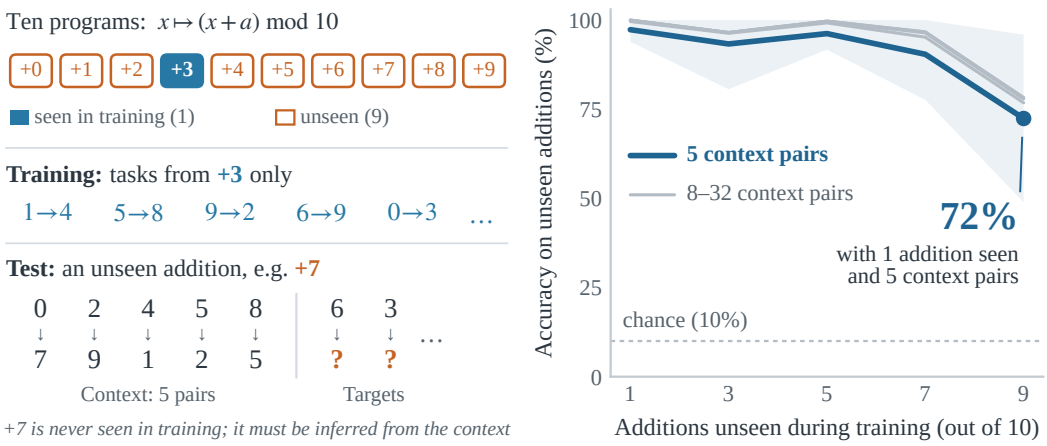


Figure 6: **Trained on a single addition, JEPIA still predicts nine unseen additions far above chance.** Left: the hardest setting. The model learns its embedding from only one of the ten additions, for example $+3$ (blue), and is then tested on the other nine (orange), none of which it has seen. Each test task provides five input–output pairs as context. Right: accuracy on unseen additions as more of the ten additions are withheld from training. The blue line uses five context pairs. Its rightmost point is the setting on the left, where accuracy is 72%, about seven times the random guess. The shaded band shows the standard deviation over five program partitions and three seeds; even one standard deviation below the mean, accuracy stays near 49%. The grey lines use 8 to 32 context pairs and stay within about six points of the blue line, so the result does not rely on a large context. Figure 7 in the appendix gives the full results.

ities: using more examples to infer a program within a fixed representation, and using experience with more programs to improve subsequent held-out prediction. They show that the architecture can benefit from both forms of additional information.

5 Discussion and Conclusion

The experiments suggest that experience with some programs can be useful for others through a shared representation of the data they relate. Modular addition makes this connection visible: organizing the digits into a suitable geometry makes different additions expressible within one simple family of transformations. Context examples then provide the information needed to fit a particular addition. The cube results examine this transfer across a larger set of transformations and across the composition lengths used to generate them.

The scope of the evidence is closely tied to the chosen latent family. Both domains contain reversible programs, and orthogonal transformations preserve the distinctions and distances between embeddings. A program that maps different inputs to the same output would require a different latent family to represent that behavior exactly while keeping those inputs distinguishable. A more expressive family would also give the fitting procedure more possibilities to resolve from context. The choice of representation and latent family must therefore account for both which programs can be expressed and how their effects can be inferred from the available examples.

Even within a suitable family, finite context can leave several transformations consistent with the observations. The current architecture selects one fit, and its predictions depend on what that context constrains. Additional context can reduce this ambiguity; representing uncertainty over the remaining fits is another direction, especially when they disagree on target inputs. The role of representation learning is to make the chosen inference procedure useful under these constraints.

JEPIA trains this interaction directly by learning from the target predictions of latent programs fitted to context. The results support its central premise: a learned data representation can allow the same fitting procedure to infer programs absent from training.

AI USE STATEMENT

Large language models were used to assist with coding, vector graphics generation, and revision of the writing. All artifacts from the Large Language Models were checked and verified by the authors.

REPRODUCIBILITY STATEMENT

Code for reproducing the experiments and figures is available in the anonymous repositories for JEPIA and the LPN modular-addition comparison.

REFERENCES

Ferran Alet, Javier Lopez-Contreras, James Koppel, Maxwell Nye, Armando Solar-Lezama, Tomas Lozano-Perez, Leslie Kaelbling, and Joshua Tenenbaum. A Large-Scale Benchmark for Few-Shot Program Induction and Synthesis. In *ICML*, volume 139 of *Proceedings of Machine Learning Research*, pp. 175–186, 2021.

Mahmoud Assran, Quentin Duval, Ishan Misra, Piotr Bojanowski, Pascal Vincent, Michael Rabbat, Yann LeCun, and Nicolas Ballas. Self-Supervised Learning from Images with a Joint-Embedding Predictive Architecture. In *CVPR*, pp. 15619–15629, 2023.

Randall Balestriero and Yann LeCun. LeJEPA: Provable and Scalable Self-Supervised Learning without the Heuristics. *arXiv:2511.08544*, 2025.

Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. DeepCoder: Learning to Write Programs. In *ICLR*, 2017.

Luca Bertinetto, João F. Henriques, Philip H. S. Torr, and Andrea Vedaldi. Meta-Learning with Differentiable Closed-Form Solvers. In *ICLR*, 2019.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D. Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language Models Are Few-Shot Learners. In *NeurIPS*, volume 33, pp. 1877–1901, 2020.

Jerry Bryan. God's Algorithm for the 2x2x2 Pocket Cube. Cube Lovers mailing list, December 4 1993. URL `https://www.math.rwth-aachen.de/~Martin.Schoenert/Cube-Lovers/Jerry_Bryan__God's_Algorithm_for_the_2x2x2_Pocket_Cube.html`.

François Chollet. On the Measure of Intelligence. *arXiv:1911.01547*, 2019.

Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. RobustFill: Neural Program Learning under Noisy I/O. In *ICML*, volume 70 of *Proceedings of Machine Learning Research*, pp. 990–998, 2017.

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 *PLDI*, pp. 835–850, 2021. doi: 10.1145/3453483.3454080.

Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-Agnostic Meta-Learning for Fast Adaptation of Deep Networks. In *ICML*, volume 70 of *Proceedings of Machine Learning Research*, pp. 1126–1135, 2017.

Marta Garnelo, Dan Rosenbaum, Christopher Maddison, Tiago Ramalho, David Saxton, Murray Shanahan, Yee Whye Teh, Danilo Rezende, and S. M. Ali Eslami. Conditional Neural Processes. In *ICML*, volume 80 of *Proceedings of Machine Learning Research*, pp. 1704–1713, 2018.

Sumit Gulwani. Automating String Processing in Spreadsheets Using Input-Output Examples. In *POPL*, pp. 317–330, 2011.

Yann LeCun. A Path towards Autonomous Machine Intelligence. Position paper, OpenReview, 2022. URL https://openreview.net/forum?id=BZ5a1r-kVsf.

Matthew Macfarlane and Clem Bonnet. Searching Latent Program Spaces. In *NeurIPS*, volume 38, Main Conference, pp. 103463–103520, 2025. doi: 10.52202/085713-3457.

Peter H. Schönemann. A Generalized Solution of the Orthogonal Procrustes Problem. *Psychometrika*, 31(1):1–10, 1966. doi: 10.1007/BF02289451.

Jake Snell, Kevin Swersky, and Richard Zemel. Prototypical Networks for Few-Shot Learning. In *NeurIPS*, volume 30, 2017.

A Problem setting and related formulations

We specify the sampling and evaluation assumptions for predictive program induction with disjoint training and evaluation programs, and relate this setting to formulations in other research communities.

A.1 Sampling tasks and evaluating predictions

For each program $p \in \mathcal{P}$, we assume access to a distribution D_p of input–output pairs consistent with a stateless relationship. When the program is executable, these pairs can be generated from an input distribution μ_p by

$$x \sim \mu_p, \qquad y = p(x). \tag{14}$$

The sampling formulation also permits programs for which only pairs can be obtained, without requiring execution on arbitrary inputs. In our experiments, each program is a deterministic transformation that remains fixed within a task and has no persistent hidden state or side effects.

For a distribution π_{train} supported on $\mathcal{P}_{\text{train}}$, sampling a task means

$$p \sim \pi_{\text{train}}, \qquad ((x_k, y_k))_{k=1}^{m+n} \sim D_p^{\otimes(m+n)}. \tag{15}$$

The first m pairs form C and the remaining n form T, equivalently to a random partition independent of pair values. Conditional independence of the sampled pairs permits repeated observations within and across context and targets; requiring distinct inputs would change the sampling distribution.

Let $X_T = (x_j^\star)_{j=1}^n$ collect the target inputs and let $F(C, X_T)_j$ be the prediction for target j. For a program distribution π, predictive risk is

$$R_\pi(F) = \mathbb{E}_{p\sim\pi}\, \mathbb{E}_{C\sim D_p^{\otimes m},\, T\sim D_p^{\otimes n}} \left[\frac{1}{n} \sum_{j=1}^{n} \ell_{\text{out}}\big(F(C, X_T)_j, y_j^\star\big) \right]. \tag{16}$$

Context and targets are sampled independently conditional on p. Exact-output error uses $\ell_{\text{out}}(\hat{y}, y) = \mathbf{1}\{\hat{y} \neq y\}$; facelet error averages the color-mismatch indicator over 24 positions. Expected accuracy is $1 - R_\pi(F)$. JEPIA trains the encoder with a latent surrogate, while a decoder permits this output-space evaluation.

Program identifiers are used to construct the split but are not supplied to the model as inputs or supervised labels, nor are generating sequences or primitive decompositions. Target outputs determine training losses and evaluation scores without participating in context fitting. Training repeatedly presents tasks from a fixed, seeded task bank. The figure probes also use fixed task collections, as specified in Appendix C.

A.2 Program ambiguity and evaluation partitions

For deterministic candidates, finite context restricts the program to a version space,

$$\mathcal{V}(C) = \{p \in \mathcal{P} : p(x_i) = y_i \text{ for every } i\}. \tag{17}$$

Programs in this set agree on the context but can disagree on target outputs. Since predictive evaluation cannot distinguish implementations that agree on all relevant inputs, the program partition separates input–output relationships rather than their names or descriptions.

We require $\mathcal{P}_{\text{train}} \cap \mathcal{P}_{\text{eval}} = \emptyset$, with evaluation programs drawn from a distribution supported on $\mathcal{P}_{\text{eval}}$. The same data values may occur in both partitions. For cubes, distinct move sequences can implement identical transformations, so a split of sequence strings alone is insufficient. We audit equivalence using the induced facelet permutations. Generating length describes how a transformation is sampled and need not be the shortest length that implements it.

Conceptually, the two partitions can share an underlying primitive set $\mathcal{P}_0$ whose finite, valid compositions generate the program space,

$$\mathcal{P} = \langle \mathcal{P}_0 \rangle_{\circ}. \tag{18}$$

The shared primitive set is a conceptual assumption about the program space, without requiring a specified library, primitive-level supervision, or training examples of individual primitives. Evaluation programs need not belong to the compositional closure of the *training programs*: both partitions belong to the space generated by the underlying primitives. For example, training on two-move cube transformations can be followed by evaluation on one-move transformations. Such shared structure provides a possible basis for generalization, without guaranteeing it from finitely many training tasks.

A.3 Connections across research communities

Program induction, synthesis, and programming by examples. Programming by examples specifies desired behavior through demonstrations, as in string-program synthesis (Gulwani, 2011). Synthesis methods seek an executable program in an allowed language; DeepCoder and DreamCoder learn to assist this process through search or library construction (Balog et al., 2017; Ellis et al., 2021). Predictive induction instead evaluates the inferred behavior through output predictions, a distinction studied in ProgReS (Alet et al., 2021). Our output-based evaluation permits both approaches, while JEPIA fits an executable latent transformation without recovering a symbolic source program.

Few-shot learning. In episodic few-shot classification, a model predicts query labels from a small support set, with methods such as Prototypical Networks learning embeddings for classification by class means (Snell et al., 2017). Context and targets have analogous roles in our setting, where the outputs are transformed data values and the required context count depends on the latent operator family, dimension, and embedding geometry.

Meta-learning. JEPIA follows the meta-learning setting by optimizing shared parameters across tasks for inference on a new task. Whereas MAML learns an initialization for gradient-based adaptation (Finn et al., 2017), differentiable analytical solvers train features through a prescribed fitting procedure (Bertinetto et al., 2019). JEPIA likewise retains task-specific fitting at evaluation, estimating an operator from context with fixed encoder parameters.

In-context learning. Predictions conditioned on demonstrations without updates to pretrained weights are commonly studied as in-context learning, including language-model few-shot evaluation (Brown et al., 2020). JEPIA shares this interface, with conditioning implemented through a fit to independently encoded pairs. The formulation does not require text, autoregression, or attention. Withholding programs is an additional constraint on the training and evaluation distributions.

Neural processes. Conditional Neural Processes predict an unknown function by aggregating encoded context observations into a representation that conditions a predictor (Garnelo et al., 2018). JEPIA conditions its predictions by fitting a transformation to paired data embeddings and applying

it to target-input embeddings. The orthogonal fit yields a point estimate, without representing a posterior over programs or calibrated uncertainty under ambiguous context.

We use program induction to denote prediction of an implied program's outputs from demonstrations, with disjoint program partitions specifying the generalization evaluated here.

B Fitting latent programs and training the representation

B.1 Empirical risk minimization in the learned representation

For fixed encoder parameters and context, the empirical risk of a latent operator is

$$\widehat{R}_C(A;\theta) = \frac{1}{m}\sum_{i=1}^{m} \ell_{\mathcal{Z}}\big(A(E_\theta(x_i)), E_\theta(y_i)\big), \tag{19}$$

where $\ell_{\mathcal{Z}} : \mathcal{Z} \times \mathcal{Z} \to \mathbb{R}_{\geq 0}$. The fitting rule selects a member of $\arg\min_{A\in\mathcal{A}} \widehat{R}_C(A;\theta)$. Equation 7 then uses target pairs to train the representation through that selection. Both the fit and the target embeddings depend on θ.

For a general operator family, fitting may require approximate optimization, and existence, uniqueness, and differentiability of a minimizer are not guaranteed. With orthogonal transformations and squared error, compactness and continuity ensure existence, and SVD provides an analytical minimizer subject to the rank and numerical qualifications below.

B.2 From empirical risk to the Procrustes solution

Using the column-vector convention of Section 3.2, stack the context embeddings into $X_C, Y_C \in \mathbb{R}^{d\times m}$. Empirical risk minimization becomes

$$\widehat{Q}_C \in \operatorname*{arg\,min}_{Q^\top Q = I_d} \frac{1}{md}\|QX_C - Y_C\|_F^2. \tag{20}$$

The constant normalization does not change the minimizers. Orthogonality preserves $\|QX_C\|_F$, giving

$$\|QX_C - Y_C\|_F^2 = \|X_C\|_F^2 + \|Y_C\|_F^2 - 2\,\mathrm{tr}(Q^\top Y_C X_C^\top). \tag{21}$$

For $Y_C X_C^\top = U\Sigma V^\top$, set $W = U^\top Q V$. Since W is orthogonal and singular values are nonnegative,

$$\mathrm{tr}(Q^\top Y_C X_C^\top) = \sum_{k=1}^{d} \sigma_k W_{kk} \leq \sum_{k=1}^{d} \sigma_k. \tag{22}$$

Choosing $W = I_d$ attains this bound, yielding $\widehat{Q}_C = UV^\top$, the classical Procrustes solution (Schönemann, 1966). We fit the embeddings without centering and allow reflections as well as rotations.

The implementation stores embeddings as rows. For row-stacked matrices $X_{\mathrm{row}} = X_C^\top$ and $Y_{\mathrm{row}} = Y_C^\top$, it computes the SVD of $X_{\mathrm{row}}^\top Y_{\mathrm{row}}$ and applies the resulting orthogonal matrix on the right. This right-acting matrix is the transpose of a minimizer under the column-vector convention.

Rank and differentiation. A full-rank $Y_C X_C^\top$ has a unique orthogonal polar factor. Rank deficiency can leave the action on unconstrained directions undetermined. The condition $m \geq d$ is necessary for full rank but not sufficient, particularly when inputs repeat. Differentiation through individual SVD factors can be unstable near degenerate singular values. Forming the cross-product costs $O(md^2)$, its dense SVD costs $O(d^3)$, and applying the fitted matrix to n targets costs $O(nd^2)$, excluding encoding and decoding.

B.3 Regularization and decoder training

Applying SIGReg. We adopt SIGReg from LeJEPA (Balestriero & LeCun, 2025). For each of the B tasks in a batch, concatenate context pairs followed by target pairs and interleave their input

and output embeddings. Write this ordered collection as $(z_{b,r})_{r=1}^{q}$, where $q = 2(m+n)$. At each position r, SIGReg receives the B embeddings at that position across tasks. The batch regularizer is

$$\mathcal{R}_{\text{batch}} = \frac{1}{q}\sum_{r=1}^{q} \text{SIGReg}\big((z_{b,r})_{b=1}^{B}; \Omega\big), \tag{23}$$

where Ω denotes random projection directions shared across positions within an update and resampled at the next update. Repeated observations are retained. Each statistic uses B samples and retains its sample-count multiplier B; the qB embeddings are not pooled into a single sample collection.

All experiments use 64 projection directions obtained by normalizing independent standard Gaussian vectors. For each direction, the implementation compares the empirical characteristic function to $\exp(-t^2/2)$ on 17 equally spaced frequencies from 0 to 3. It weights squared complex discrepancies by $\exp(-t^2/2)$ and uses the symmetric trapezoidal integral: with spacing $\Delta t = 3/16$, endpoint weights are Δt and interior weights are $2\Delta t$. The resulting statistic is multiplied by B and averaged over directions and positions.

The encoder objective averages target MSE across tasks and adds this regularizer:

$$\mathcal{L}_{\text{enc}} = (1-\lambda)\frac{1}{B}\sum_{b=1}^{B} \mathcal{L}_{\text{pred}}(C_b, T_b) + \lambda \mathcal{R}_{\text{batch}}. \tag{24}$$

Gradients propagate through the context fit and all encoder branches, including the target-output embeddings. Only the auxiliary decoder objective stops gradients at the encoder.

The reference distribution $z \sim \mathcal{N}(0, I_d)$ has $\mathbb{E}\|z\|_2^2 = d$ and $\text{Var}(\|z\|_2^2) = 2d$, so the relative standard deviation of the squared norm is $\sqrt{2/d}$. As dimension increases, norm fluctuations become small relative to the radius $\sqrt{d}$. This concentration supports the spherical-shell intuition in Section 3.2; the regularizer does not constrain embeddings to a sphere of fixed radius.

Decoder training. Let S_b concatenate the context and target pairs of training task b, retaining repetitions. An auxiliary decoder minimizes

$$\mathcal{L}_{\text{dec}}(\phi) = \frac{1}{B}\sum_{b=1}^{B} \frac{1}{|S_b|} \sum_{(x,y)\in S_b} \ell_{\text{dec}}(D_\phi(\text{sg}[E_\theta(y)]), y)\,, \tag{25}$$

where sg stops gradients and ℓ_{dec} is distinct from the evaluation loss ℓ_{out}. The encoder update precedes the decoder update, which uses its output embeddings and a separate optimizer. Modular-addition decoding uses a linear classifier with cross-entropy over digits; cube decoding predicts categorical facelet colors. At evaluation, predicted embeddings replace the true output embeddings used for decoder training. Reconstruction accuracy from true output embeddings is evaluated separately to assess decoder error within the complete prediction procedure.

Latent-error diagnostics. To account for representation scale when monitoring modular-addition training, we divide latent MSE by the mean coordinate variance of the ten digit embeddings, with a denominator floor of 10^{-12}. We also record this variance, mean squared embedding norm, decoder reconstruction accuracy, and the ten digit embeddings themselves. These quantities are diagnostics and do not modify the encoder objective.

C Experimental details

C.1 Models and training

All JEPIA experiments use the orthogonal latent program family described in Section 3.2. A shared encoder embeds each input and output independently. For each task, an orthogonal transformation is fitted to the context embeddings by the Procrustes solution in Equation 11, then applied to the target-input embeddings. An auxiliary decoder converts the resulting embeddings into output predictions. The model receives input–output examples; the generating program is used to construct the task and is not supplied to the model.

The encoders are multilayer perceptrons with one GELU hidden layer and a linear output layer. Modular addition uses five-dimensional embeddings and a linear decoder with ten output logits. Cube states use 32-dimensional embeddings and a decoder with one GELU hidden layer, followed by six color logits for each of the 24 facelets. Predictions select the highest-scoring digit or facelet color. Table 1 gives the layer widths and training settings.

The encoder minimizes target prediction error in the latent space together with SIGReg, as specified in Equation 24. Gradients pass through the context fit. The decoder is trained with cross-entropy on the observed output embeddings from both context and target pairs, without updating the encoder. Each batch produces one encoder update followed by one decoder update. Both optimizers are Adam with constant learning rates and no weight decay. SIGReg uses 64 random projections and 17 frequency points up to 3, with the batch aggregation defined in Equation 23.

Training tasks are sampled once and reused in shuffled order throughout training. Each task samples a program from the training program space, then samples its context and target pairs independently conditional on that program, with replacement. At evaluation, the encoder and decoder remain fixed, and a new latent program is fitted for each task. Evaluation tasks are held fixed across checkpoints. Thus, learning curves compare predictions on the same examples as the representation changes.

Table 1: JEPIA model and training settings. Layer widths include input and output dimensions. The number of training tasks counts distinct sampled tasks, which are reused across updates. The program/context study varies the training program space and evaluation context size while retaining the settings in its column.

Setting	Modular addition	Program/context study	Cube
Encoder widths	$10, 16, 5$	$10, 16, 5$	$144, 256, 32$
Decoder widths	$5, 10$	$5, 10$	$32, 256, 144$
Training context / targets	$8/1$	$8/1$	$64/8$
Training tasks	1,000	1,000	4,096
Batch size	16	5	16
Updates	1,000	2,000	800
Encoder learning rate	10^{-3}	5×10^{-3}	2×10^{-3}
Decoder learning rate	10^{-2}	10^{-2}	2×10^{-3}
SIGReg weight λ	0.03	0.1	0.1

C.2 Modular addition

Domain and task construction. The data space is $\mathcal{X} = \{0, \dots, 9\}$ and the program space is $\mathcal{P} = \{p_a : a \in \mathcal{X}\}$, where $p_a(x) = (x + a) \bmod 10$. A task samples an addend uniformly from the relevant program space and samples each input uniformly from the ten digits. Inputs and outputs are represented as ten-dimensional one-hot vectors. These representations identify the digits without encoding their numerical order.

For the experiment in Figure 3, each training program space contains nine additions and the remaining addition is held out. All ten digits can occur in both training and evaluation; the split withholds an input–output relationship rather than a data value. We evaluate every choice of held-out addend, with three training runs per partition, for 30 runs in total. Each run uses 1,000 training tasks and 1,000 independently sampled held-out tasks, each with eight context pairs and one target pair. Repeated runs vary initialization and task sampling.

Accuracy and uncertainty. Accuracy is the fraction of target digits predicted exactly, with chance accuracy $1/10$. Seen-program accuracy is measured on the training tasks and held-out accuracy on the separate evaluation tasks. Figure 3 averages equally over the 30 runs. Pointwise 95% intervals are estimated from 2,000 hierarchical bootstrap draws, resampling the ten program partitions and then the three runs within each selected partition. Accuracy is measured throughout the 1,000 training updates.

Representation visualization. The PCA sequence in Figure 3 shows the learned embeddings from one training run with $+3$ held out, at steps 1, 100, 250, 500, and 1,000. At each checkpoint, the ten digit embeddings are centered and projected onto their first two principal components. The pro-

jections are orthogonally aligned to the final snapshot and displayed on a common scale without rescaling, so changes in orientation do not obscure changes in the arrangement. This sequence illustrates the geometry of one learned representation; the accuracy curves summarize all runs. Latent fitting and prediction use the full five-dimensional embeddings.

C.3 CUBE TRANSFORMATIONS

Domain and program partitions. The data are states of a $2 \times 2 \times 2$ Rubik's cube in a fixed orientation. A program is a transformation generated by quarter turns from $\{R, R^{-1}, U, U^{-1}, F, F^{-1}\}$. Programs are identified by their permutations of the 24 facelet positions, so sequences with the same effect define the same program. The generating sequence length need not be the shortest length that produces that transformation.

All two-move sequences generate 27 distinct nonidentity transformations. A random partition assigns 19 to training, four to validation, and four to held-out evaluation. Evaluation at length one uses all six primitive turns. For each generating length from three through twenty, we sample 16 distinct transformations from sequences without consecutive turns of the same face. We exclude all one- and two-move transformations and those already selected at shorter evaluation lengths. The resulting evaluation program sets are mutually disjoint and contain no training programs.

Tasks and evaluation. Input states are generated by 30-move scrambles from the solved state, allowing quarter turns in both directions on all six faces and excluding consecutive turns of the same face. Each state is represented by 24 six-way one-hot vectors, giving 144 input coordinates. A task samples one program uniformly from its program set and applies it to independently sampled input states, providing 64 context pairs and eight target pairs. Training uses 4,096 tasks. The length comparison uses 32 independently sampled evaluation tasks per program set, held fixed across checkpoints.

Figure 5 shows one training run at initialization and after 50 and 800 updates; evaluation is performed every ten updates. At length two, the plotted tasks use the four held-out transformations. At other lengths, they use the evaluation sets described above. Each value averages facelet accuracy over 32 tasks with eight target states each. Facelet accuracy measures the fraction of facelet colors predicted correctly, with chance accuracy $1/6$. It gives partial credit for an imperfect state and therefore differs from exact whole-state accuracy. These measurements characterize transfer on sampled programs and inputs, rather than exhaustive performance over a program space.

Size of the program spaces. The upper panel of Figure 5 counts distinct nonidentity transformations expressible by unrestricted sequences of exactly L moves from the six program moves. These moves fix one corner and generate a group of $7!3^6 = 3{,}674{,}160$ transformations. Let h_k be the number whose shortest quarter-turn sequence has length k, from the exhaustive enumeration of Bryan (1993). Each quarter turn changes corner-permutation parity, while insertion of an inverse-move pair increases length by two without changing the transformation. Consequently,

$$N_L = \sum_{\substack{0 \le k \le L \\ k \equiv L \pmod 2}} h_k - \mathbf{1}\{L \text{ is even}\}. \tag{26}$$

The subtraction excludes the identity. This gives $N_1 = 6$, $N_2 = 27$, $N_5 = 2{,}382$, $N_{10} = 1{,}054{,}267$, and $N_{20} = 1{,}837{,}079$. These counts describe the full program spaces; the accuracy measurements use the restricted samples specified above.

C.4 VARYING TRAINING PROGRAMS AND CONTEXT SIZE

This experiment varies two sources of information separately: the additions available during representation learning and the examples available when fitting a held-out program. We use the same modular-addition domain and model architecture as above, with the training settings in the program/context column of Table 1.

We draw five random orderings of the ten additions. Each ordering defines nested training program spaces containing the first one, three, five, seven, or nine additions, with the remaining additions held out. Three training runs per program space give 75 models in total. Every model receives 1,000

training tasks with eight context pairs and one target pair, and is trained for 2,000 updates. Varying the training program count therefore changes the available relationships while holding the number of tasks and updates fixed.

Each trained model is evaluated on 1,000 tasks sampled from its held-out program space. A task contains a pool of 32 context pairs and ten target pairs. We fit latent programs using nested subsets of 5, 8, 12, 20, or 32 context pairs, keeping the model and target pairs fixed. All pairs are sampled independently with replacement conditional on the task's program. This comparison measures the benefit of additional context without further representation learning.

Figure 7 reports exact target accuracy as mean ± sample standard deviation across the five program partitions and three training runs per partition. Each cell therefore summarizes 15 trained models. Figure 6 presents the same means as curves against the number of held-out programs, with the standard-deviation band shown for five context pairs. The held-out program space changes with the training program space; the results therefore summarize transfer across several choices of training and held-out additions.

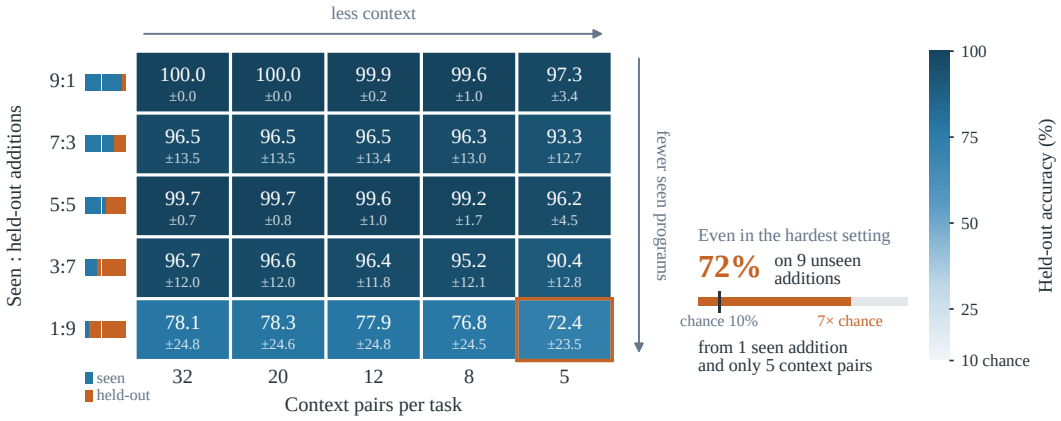


Figure 7: **Full results for varying training programs and context size in modular addition.** Each cell gives accuracy on held-out additions as mean ± SD over five program partitions and three seeds. Rows set how many of the ten additions are seen in training (blue) and held out for evaluation (orange). Columns set the number of context pairs per test task, with ten target pairs throughout. The context comparison keeps each trained model and its target pairs fixed and varies only the context. The training-program comparison keeps the number of training tasks and updates fixed and evaluates each model on the additions it did not see. Colors run from chance (10%) to 100%.

D Comparing with the Latent Program Network

D.1 Latent Program Networks

LPN (Macfarlane & Bonnet, 2025) performs program induction by search in a learned continuous latent space. An encoder maps the context pairs to a distribution over latent programs; a decoder predicts the output for a given input conditioned on a latent program; and at inference the latent is refined by gradient ascent on the decoder likelihood of the context pairs. Gradient steps can also be taken during training so that the encoder and decoder are adapted to this refinement. The original work reports that accuracy on synthetic pattern tasks and on ARC-AGI (Chollet, 2019) improves with the number of gradient steps. Unlike JEPIA, LPN places no constraint on the geometry of the latent space or on the form of a program's action on data. The experiments below ask whether this generality suffices for programs withheld from training in the modular-addition setting of Section 4.1.

D.2 Setup

We evaluate LPN on the modular-addition tasks using the authors' implementation[1]. The training loss is the cross-entropy of the predicted output grid. We match the program partitions (1, 3, 5, 7, or 9 addends seen) and two context sizes (5 and 12 pairs, with 10 targets), and vary the number of gradient-ascent steps during training (0, 1, 5) and inference (0, 5, 20, 100). Other settings follow the original work and are listed in Table 2; note that LPN receives substantially more updates and a larger batch than JEPIA. Results are reported as mean $\pm$ SD over three replicates, each with an independent training seed and an independently sampled program partition.

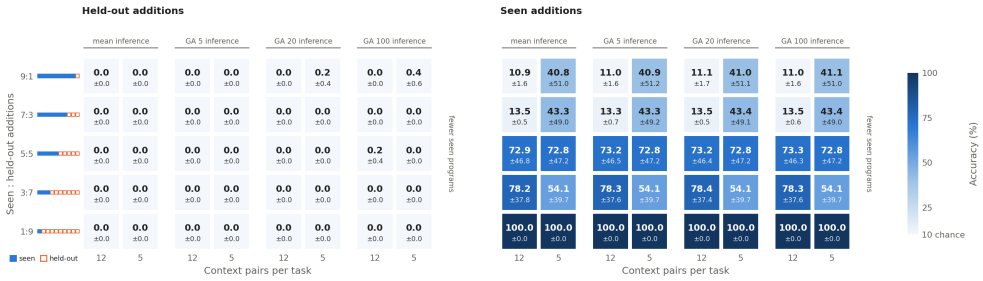
Table 2: LPN settings for modular addition

Task	
Modulus	10
Input encoding	one-hot (1×10)
Addends seen / held out	9/1, 7/3, 5/5, 3/7, 1/9
Context & Target	$(5, 10), (12, 10)$
Replicates (seed and partition)	3
Architecture	
Encoder / decoder layers	$2/2$
Attention heads $\times$ dim per head	6×12
MLP dim factor	4
Latent dim	8
Training	
Batch size	128
Steps	100,000
Learning rate	4×10^{-4}
Prior KL weight	10^{-3}
Latent GA steps / lr (train)	$0, 1, 5/0.1$
Latent optimizer	Adam ($\beta_2 = 0.9$)
Evaluation	
Latent GA steps / lr (test)	$0, 5, 20, 100/0.1$
Test tasks (seen / unseen)	$128/64$

[1] https://github.com/clement-bonnet/lpn

D.3 RESULTS

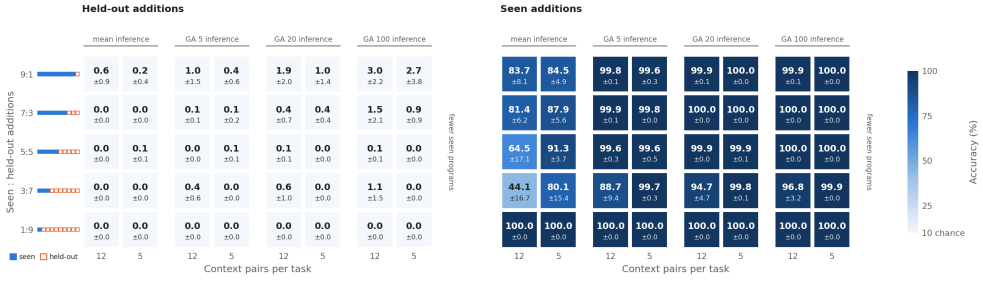
Figure 8 reports the full grid. On seen addends LPN reaches 100% in most settings, with lower accuracy only when no gradient steps are taken during training and many addends are seen. On held-out addends the highest mean accuracy is $3.0 \pm 2.2\%$, and most cells are at 0%, below the 10% chance level. Accuracy below chance indicates that the predictions are systematically wrong rather than random: on a held-out task the model produces the outputs of a training addend (Figure 10a). Neither more training gradient steps nor more inference steps change this outcome, and the larger context does not help.



(a) Training gradient-ascent steps: 0



(b) Training gradient-ascent steps: 1

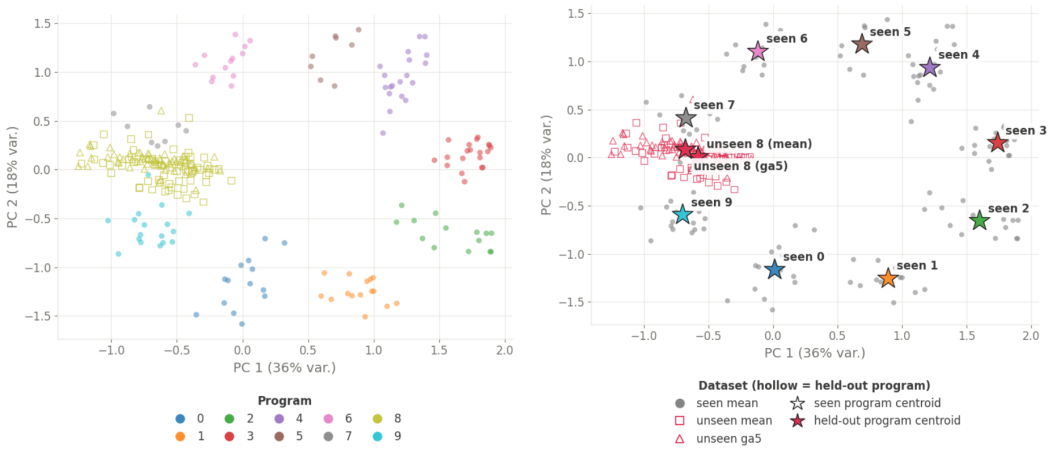


(c) Training gradient-ascent steps: 5

Figure 8: **Full LPN results on modular addition.** Accuracy on held-out (left) and seen (right) addends as mean ± SD over three replicates (seed and partition), for training gradient-ascent steps of 0, 1, and 5 (rows of panels), inference steps of 0, 5, 20, and 100 (columns), context sizes of 12 and 5, and 1 to 9 addends seen in training. Seen accuracy is high in most settings; held-out accuracy stays at or below chance (10%) throughout.

D.4 WHERE THE FAILURE ARISES

Three components could account for the held-out failure: the encoder could place held-out tasks at uninformative latents, the test-time search could fail to reach a latent that implements the program, or the decoder could implement the program at no latent. Figure 9 rules out the first. The training-program latents form clusters ordered cyclically by addend, and the latents of held-out +8 tasks, at the encoder mean and after five refinement steps, lie between the +7 and +9 clusters.

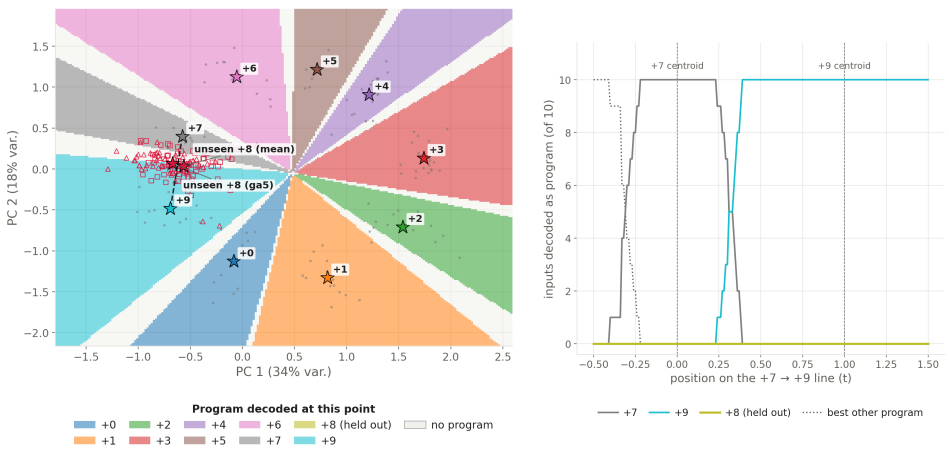


(a) PCA of latent — colored by program (b) PCA of latent — seen vs held out programs

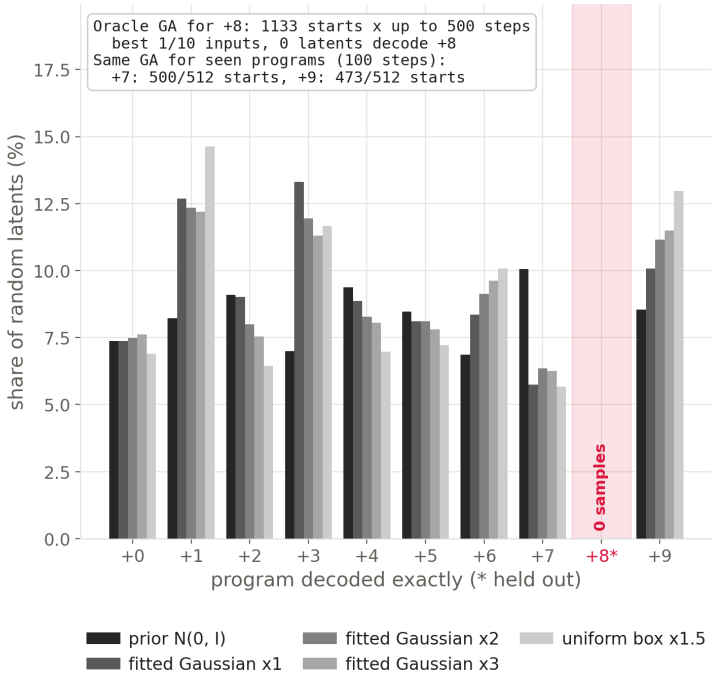
Figure 9: **LPN's encoder places a held-out program between its neighbors.** Two-dimensional PCA of the 8-dimensional latent programs from one run trained with one gradient-ascent step, five context pairs, ten targets, and $+8$ held out. Points are encoder means and their five-step refinements at inference. **(a)** Colored by addend: each training addend forms a cluster, and the clusters are arranged cyclically in numerical order. **(b)** Training-program centroids (stars) and held-out $+8$ latents (hollow), which fall between the $+7$ and $+9$ clusters.

Figure 10 tests the other two by probing the trained decoder directly, with neither encoder nor search involved. For a latent z we decode all ten inputs and say that z implements $+s$ if the decoded table equals $(x + s) \bmod 10$ on every input. Three probes give the same answer. Along the segment from the +7 to the +9 centroid, the decoded table passes from +7 through latents whose table mixes $+7$ and $+9$ outputs, and matches +8 on no input at any point; the held-out tasks' own latents decode to +9, +7, or no program, never to +8. Among 250,000 latents drawn from five distributions (the prior, Gaussians fitted to the seen latents at one to three times their scale, and a uniform box), every training addend is implemented by at least 2,658 latents and +8 by none. Finally, an oracle search runs the training-time gradient ascent on the full ten-pair table of +8, more information than the five context pairs available at test time, from 1,133 starting points for up to 500 steps; its best latent matches 1 of 10 inputs. The same search with 100 steps recovers +7 from 500/512 and +9 from 473/512 random starts, so the search procedure is adequate for programs the decoder can express. Four further runs, including a run with +4 held out and one trained with five gradient steps, show the same pattern, with at most 3 of 10 inputs matched by the oracle and no random latent implementing the held-out addend.

The decoder therefore partitions the latent space among the training addends and implements the held-out addend nowhere, so no search can recover it regardless of where the encoder initializes. This is a consequence of the objective: the decoder is trained only at the latents of training programs, and nothing constrains how intermediate points decode. The improvements with gradient steps reported for the original pattern tasks are consistent with this account, since there each training task draws a fresh pattern from a space of about 10^{16} and the training latents populate the space densely. Modular addition supplies at most nine training programs, which the objective is satisfied to place as separate clusters. JEPIA does not represent programs as points: a program is a transformation between data embeddings, and once the digit embeddings are arranged so that every addition is an orthogonal map, a held-out addend is expressible by the same fitting procedure.



(a) Decoder map over the latent plane (b) Decoding along the $+7 \to +9$ line



(c) Programs found by random sampling

Figure 10: **LPN's decoder implements the held-out program at no latent.** Decoder probe of the model of Figure 9, with the encoder and search removed: a latent implements $+s$ if decoding all ten inputs yields $(x + s) \bmod 10$. **(a)** A PCA plane of the latent programs, colored by the program each point implements when fed to the decoder (off-plane coordinates at their mean). Every training addend has a region; $+8$ has none, and the held-out tasks' latents (hollow markers) fall in the $+7$, $+9$, and no-program regions. **(b)** Number of inputs matching $+7$, $+9$, and $+8$ along the line through the $+7$ and $+9$ centroids, with the best-matching other program dotted: the table switches from $+7$ to $+9$ through latents matching neither on all inputs, and never matches $+8$. **(c)** Share of 250,000 random latents from five distributions implementing each addend exactly; $+8$ is implemented by none. Text box: gradient ascent on the full $+8$ table from 1,133 starts for up to 500 steps matches at most 1 of 10 inputs, whereas the same search recovers $+7$ and $+9$ from over 90% of random starts.

D.5 SCOPE OF THE COMPARISON

The comparison isolates the effect of representing programs as transformations rather than as points, under a domain in which the orthogonal family is well matched; it does not rank the two methods in general. We did not evaluate LPN on the cube tasks, which would require adapting its grid encoding to facelet states, nor include an in-context transformer trained on the same tasks, which would separate this design choice from other architectural differences. The settings are matched in program partitions and context sizes but not in training budget, input encoding, or number of replicates: LPN receives more updates and, through positional embeddings over the 1×10 grid, access to cell order that JEPIA's one-hot inputs lack, and is run with three seed-and-partition replicates against five partitions and three seeds for JEPIA. The first two differences favor LPN.