CS 957, System-2 AI
Program Synthesis

Mahdi Samiei

Mar 2025

Sharif University of Technology

Recap: Symbolic Regression

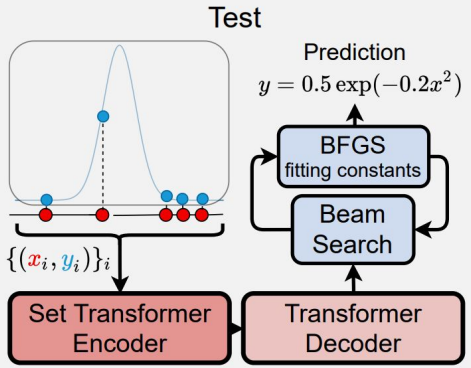
📐 Traditional regression fits parameters to a predefined model structure, while symbolic regression searches for both the structure and parameters simultaneously.

💡 Interpretable formula

🚧 Extrapolation to Out-of-Distribution Data

🏗️ Generalization to more complex formulas

🖼️ Regression is a metaphor, get the gist



Recap: Symbolic Regression

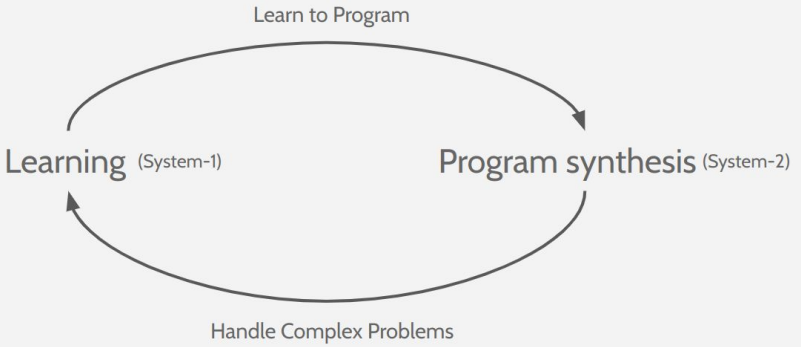
➡️ Consider Inference Phase: Given $\{(x_i, y_i)\}$, We should discover f that $y_i = f(x_i)$

➡️ Indeed each task is a sample for symbolic regression.

🤔 Remember Definition of Generalization?

📊 The performance of our symbolic regression method can be measured either on OoD data or OoD equations.

➡️ Program Synthesis: learning an input-output mapping from very few examples



Today Question:
Program Synthesis beyond Regression

DreamCoder: Growing generalizable, interpretable knowledge with wake-sleep Bayesian program learning

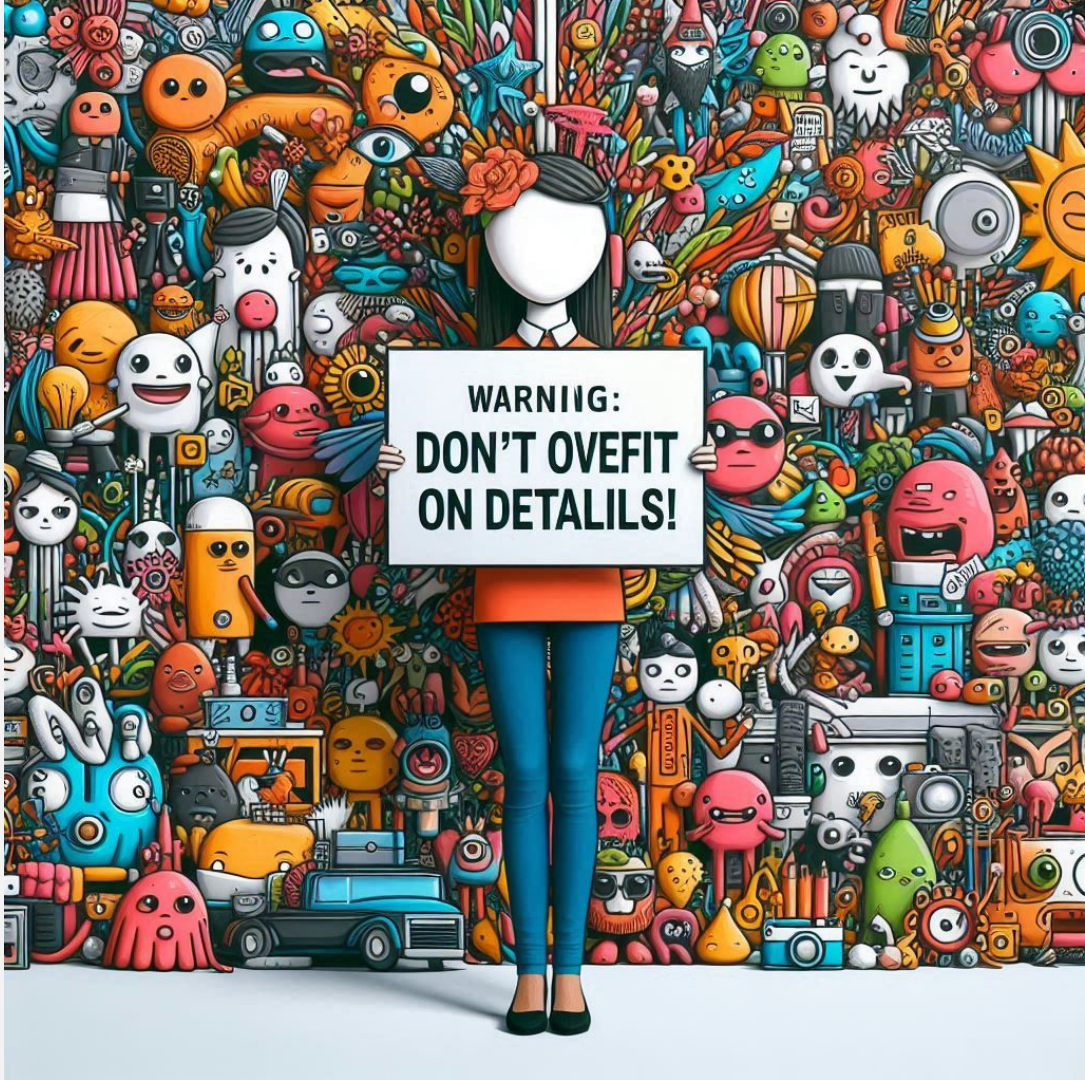
Kevin Ellis, Catherine Wong, Maxwell Nye, Mathias Sable-Meyer, Luc Cary, Lucas Morales, Luke Hewitt, Armando Solar-Lezama, Joshua B. Tenenbaum

June 2020

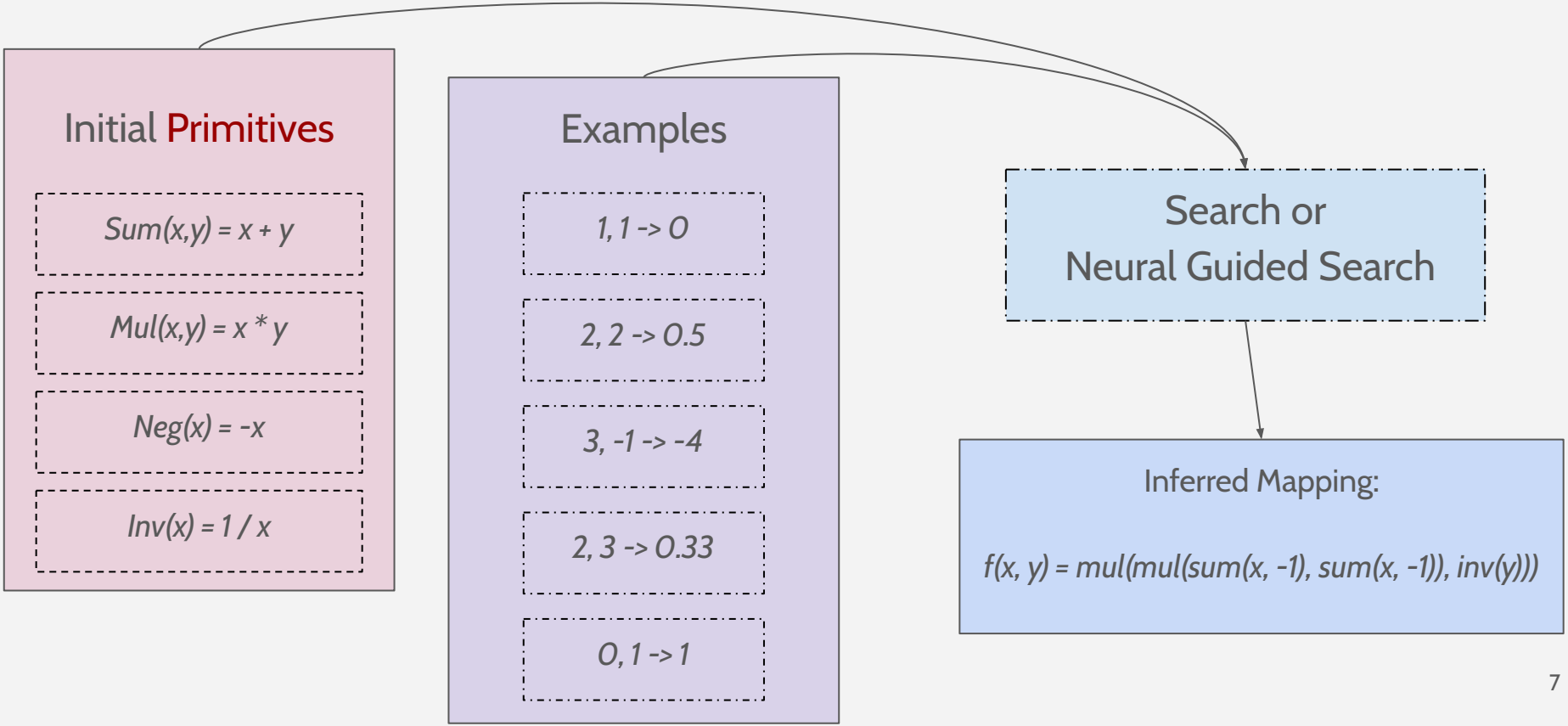
Warning: Don’t overfit on details! Get the point!

Dreamcoder is complex so that can’t easily run it!

Pay attention to common concepts, pain points, and suggested solutions.

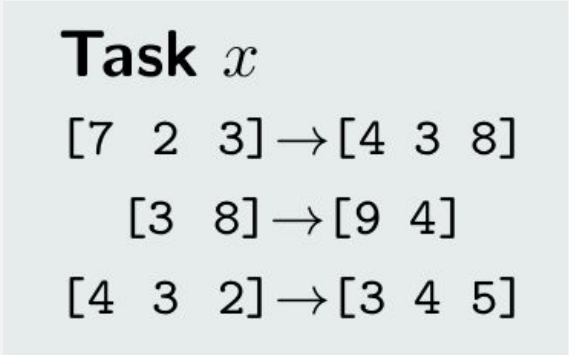


Recap: Symbolic Regression Setting



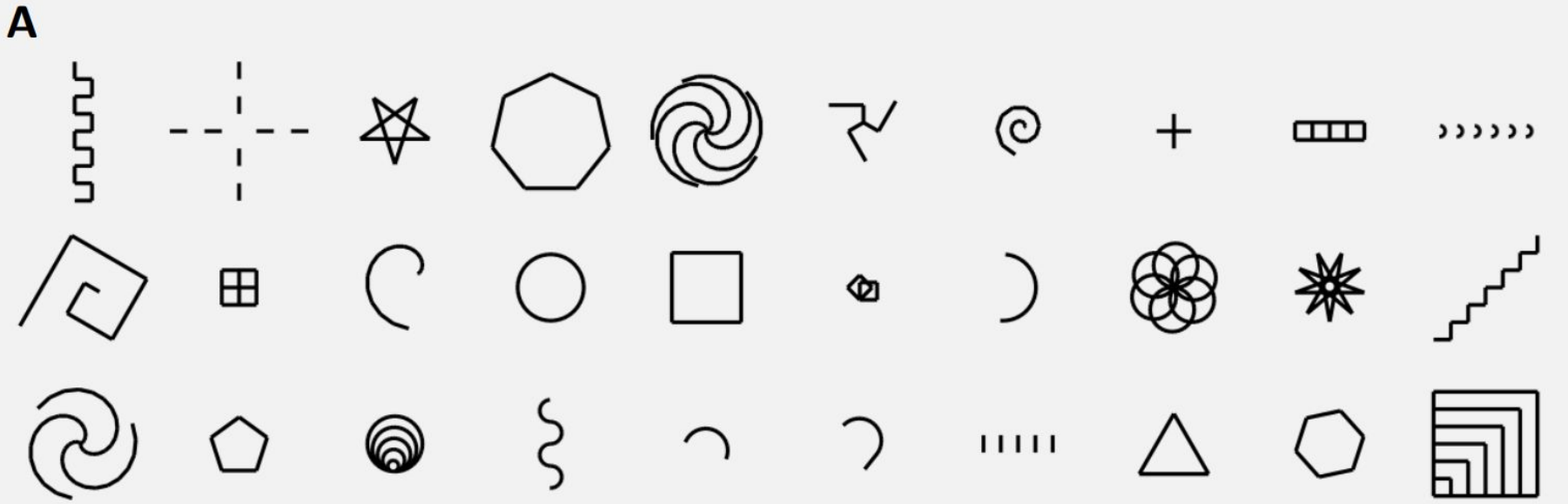
Task: List Processing

Primitives: *fold*, *unfold*, *if*, *map*, *length*, *index*, *=*, *+*, *-*, *0*, *1*, *cons*, *car*, *cdr*, *nil*, and *is-nil*

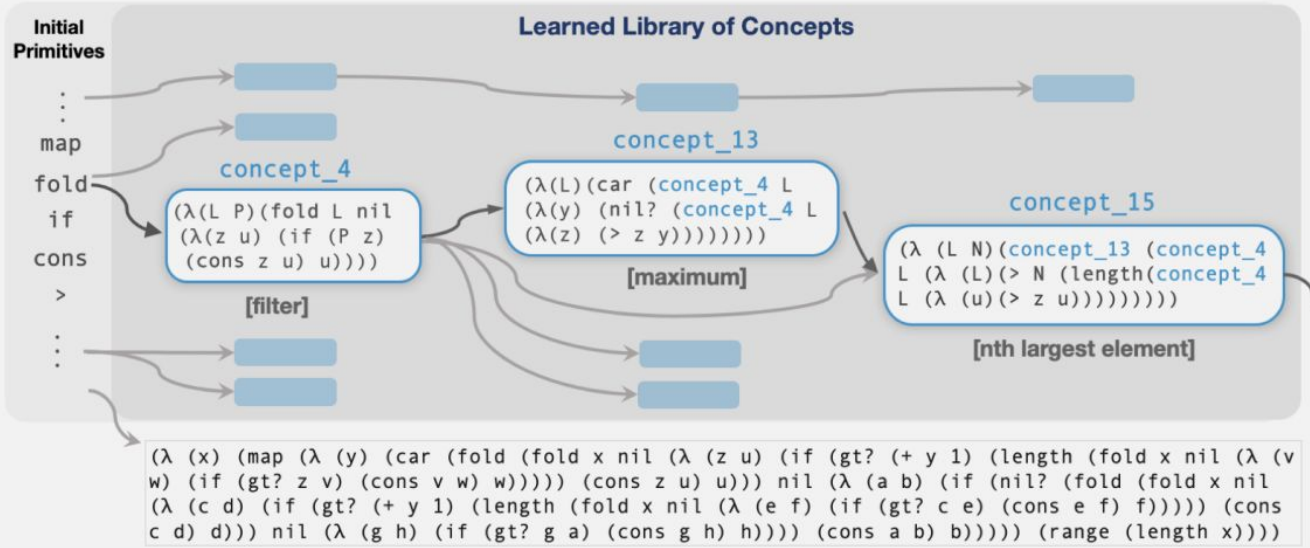


Task: LOGO Graphics

Primitives: *for*, *get/set*, *move*, *pen-up*



how DreamCoder learns to solve problems in one domain



Sample Problem: `Sort List`

[9 2 7 1] → [1 2 7 9]
[3 8 9 4 2] → [2 3 4 8 9]
[6 2 2 3 8 5] → [2 2 3 5 6 8]
...

Solution to `Sort List` discovered in learned language:

```
(map  (λ (n)
  (concept_15 L (+ 1 n)))
  (range (length L)))
```

Solution to `sort list` if expressed in initial primitives

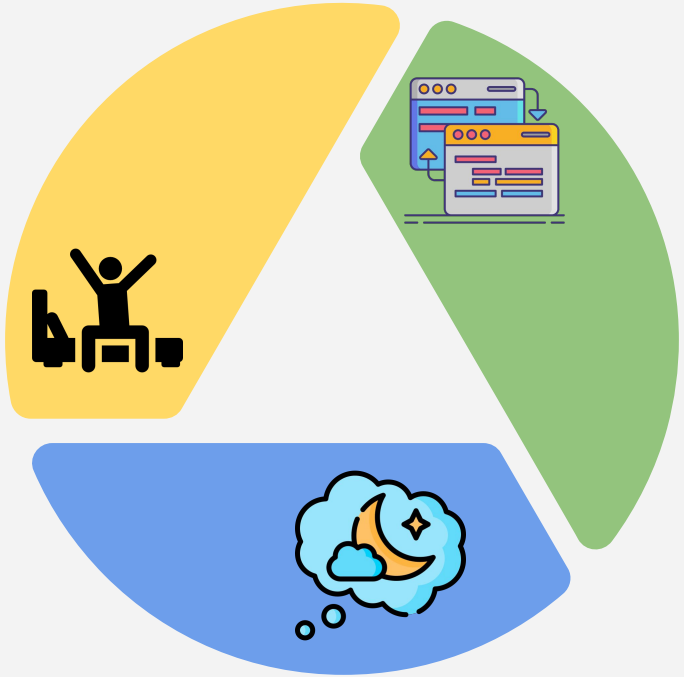
Wake/Sleep Program Learning

Wake

For each task x in X, find best program ρx solving x under current library L

Dreaming

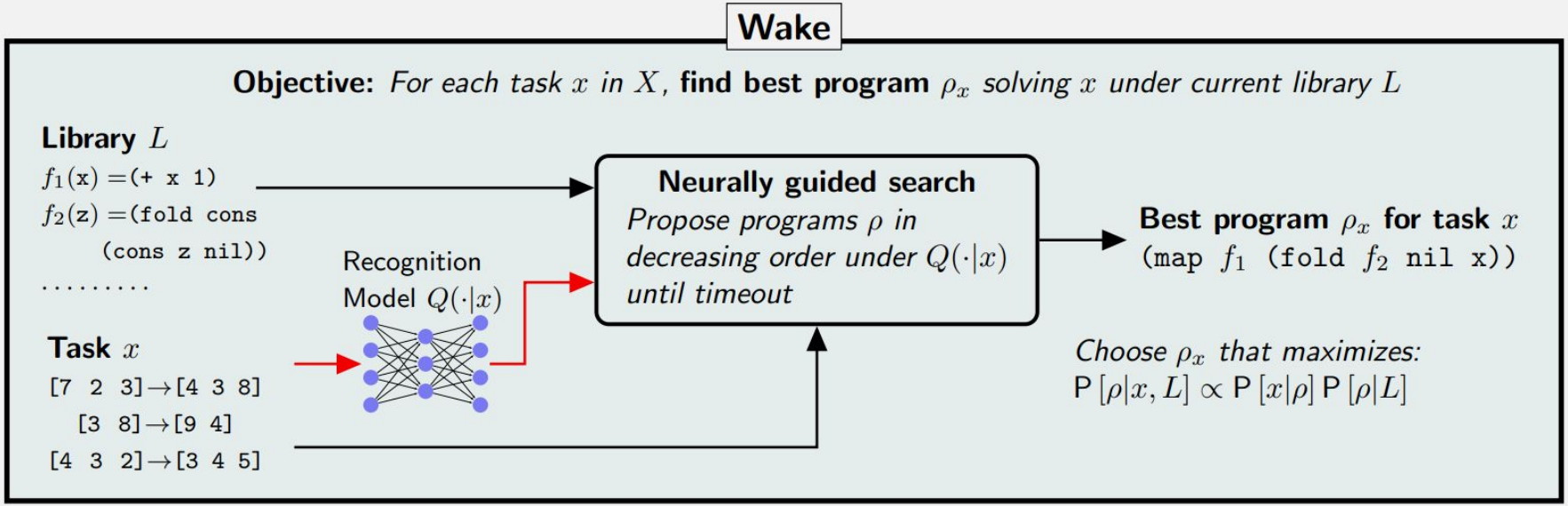
Train recognition model Q(ρ|x) to predict best programs ρx for typical tasks x and current library L



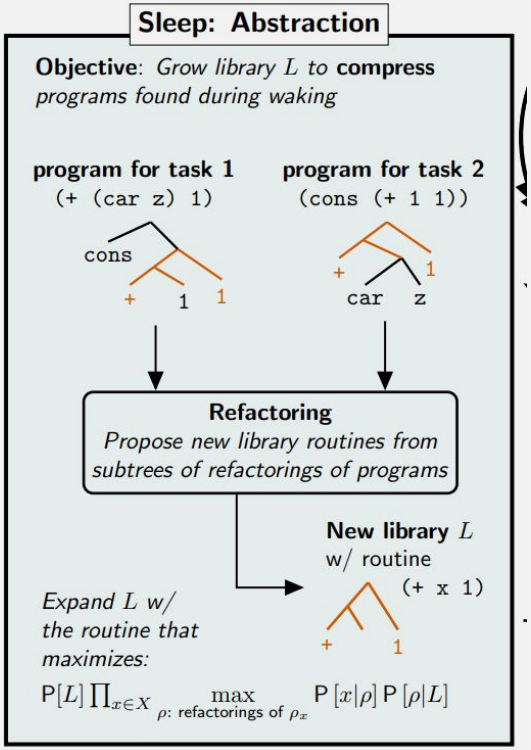
Abstraction

Grow library L to compress programs found during waking

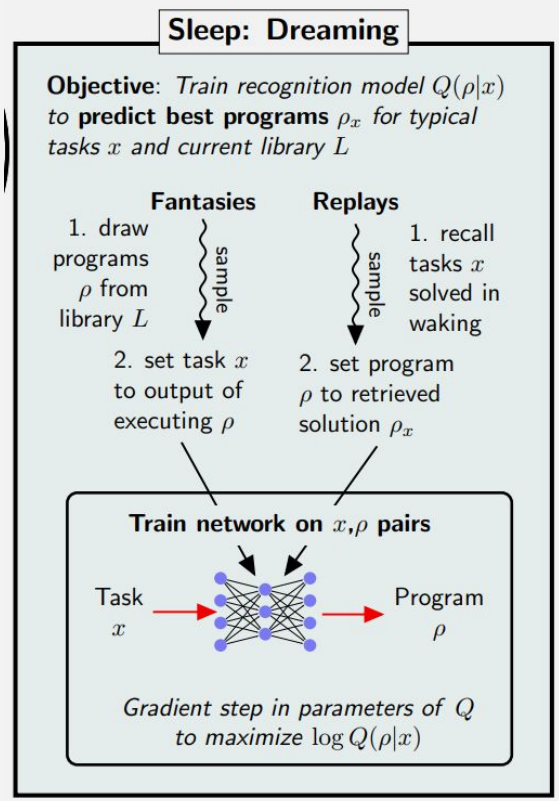
Wake Phase


Wake
Objective: For each task x in X, find best program ρ_x solving x under current library L
Library L
f_1(x) =(+ x 1)
f_2(z) =(fold cons
(cons z nil))
.....
Task x
[7 2 3]→[4 3 8]
[3 8]→[9 4]
[4 3 2]→[3 4 5]
Recognition
Model $Q(\cdot|x)$
Neurally guided search
Propose programs ρ in
decreasing order under $Q(\cdot|x)$
until timeout
Best program ρ_x for task x
(map f_1 (fold f_2 nil x))
Choose ρ_x that maximizes:
$\mathrm{P}[\rho|x, L] \propto \mathrm{P}[x|\rho]\,\mathrm{P}[\rho|L]$

Abstraction Phase



Dreaming Phase



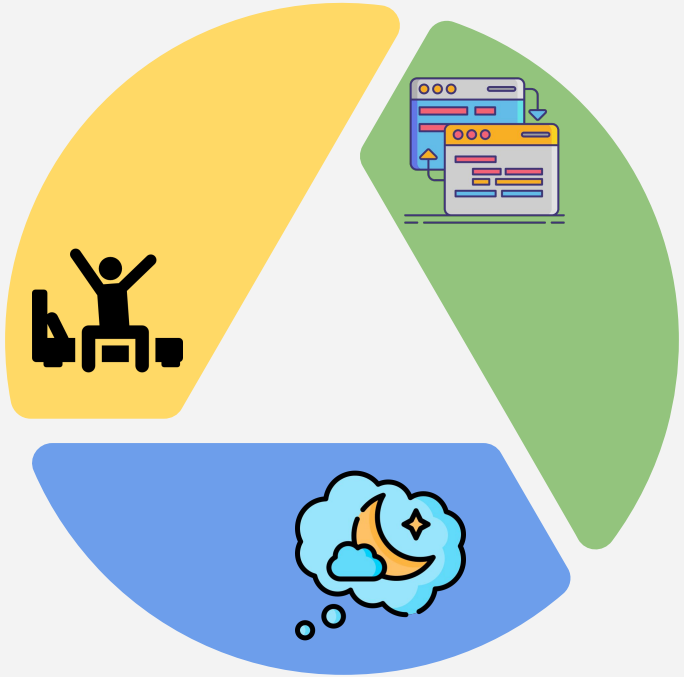
Wake/Sleep Program Learning

Wake

For each task x in X, find best program ρx solving x under current library L

Dreaming

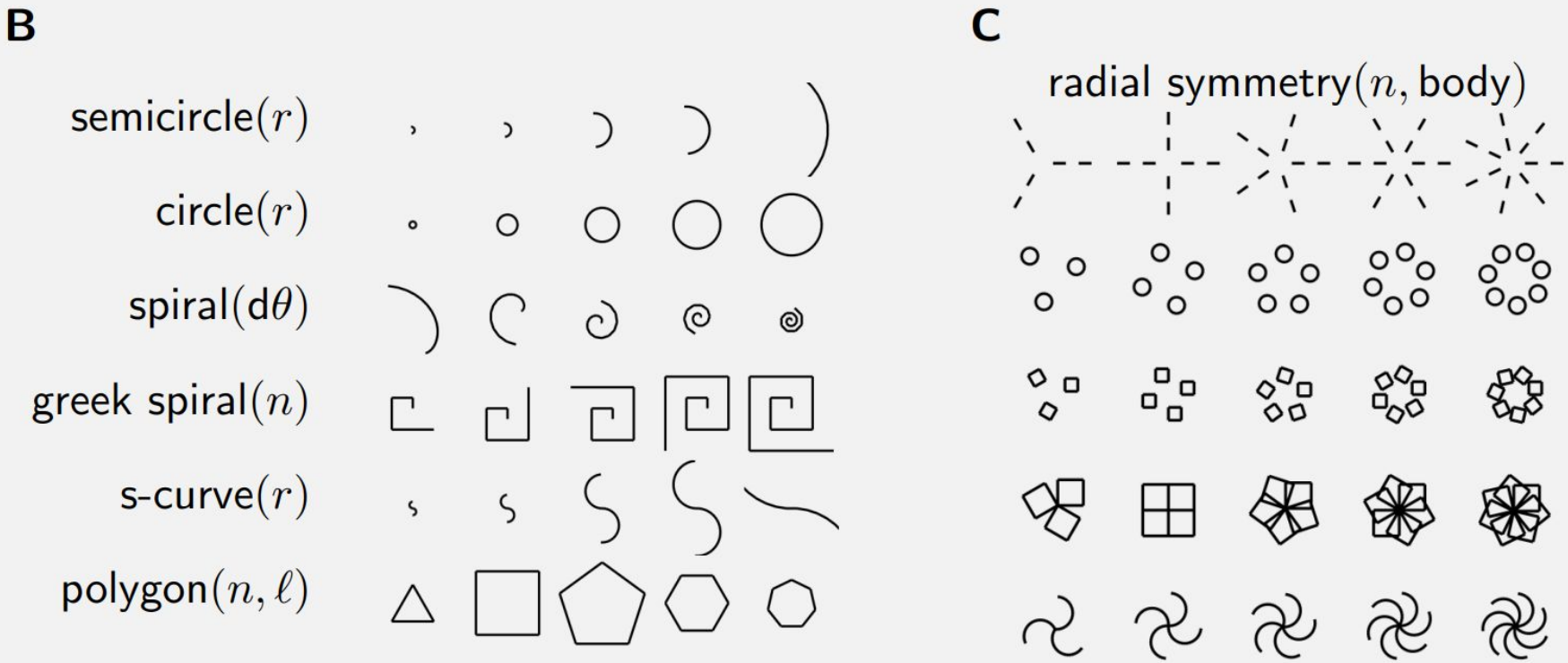
Train recognition model Q(ρ|x) to predict best programs ρx for typical tasks x and current library L



Abstraction

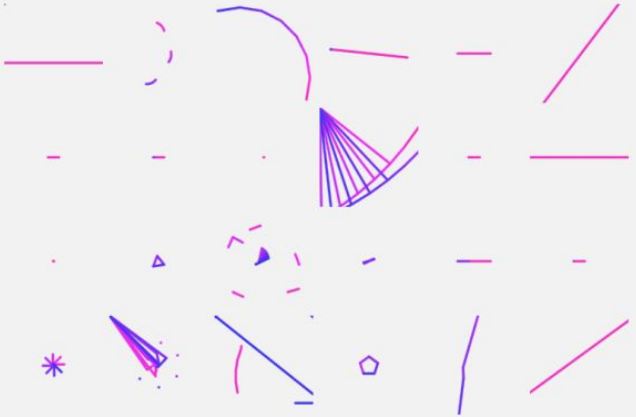
Grow library L to compress programs found during waking

LOGO graphics: Emerge of Higher-Orders functions

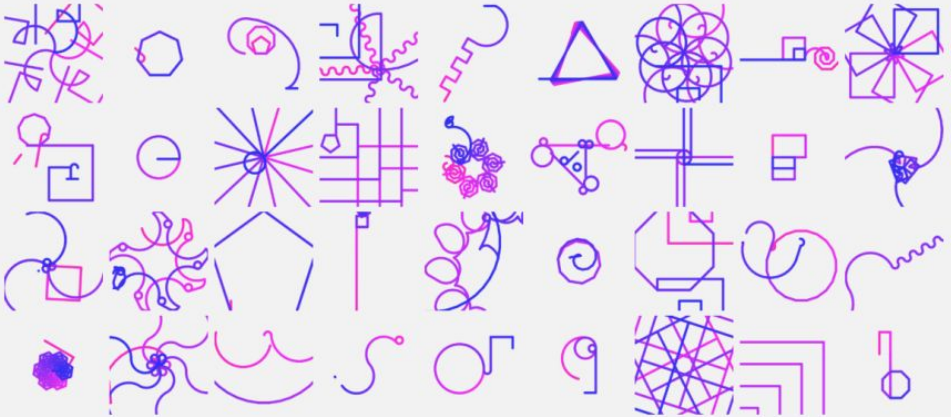


LOGO graphics: Dreaming

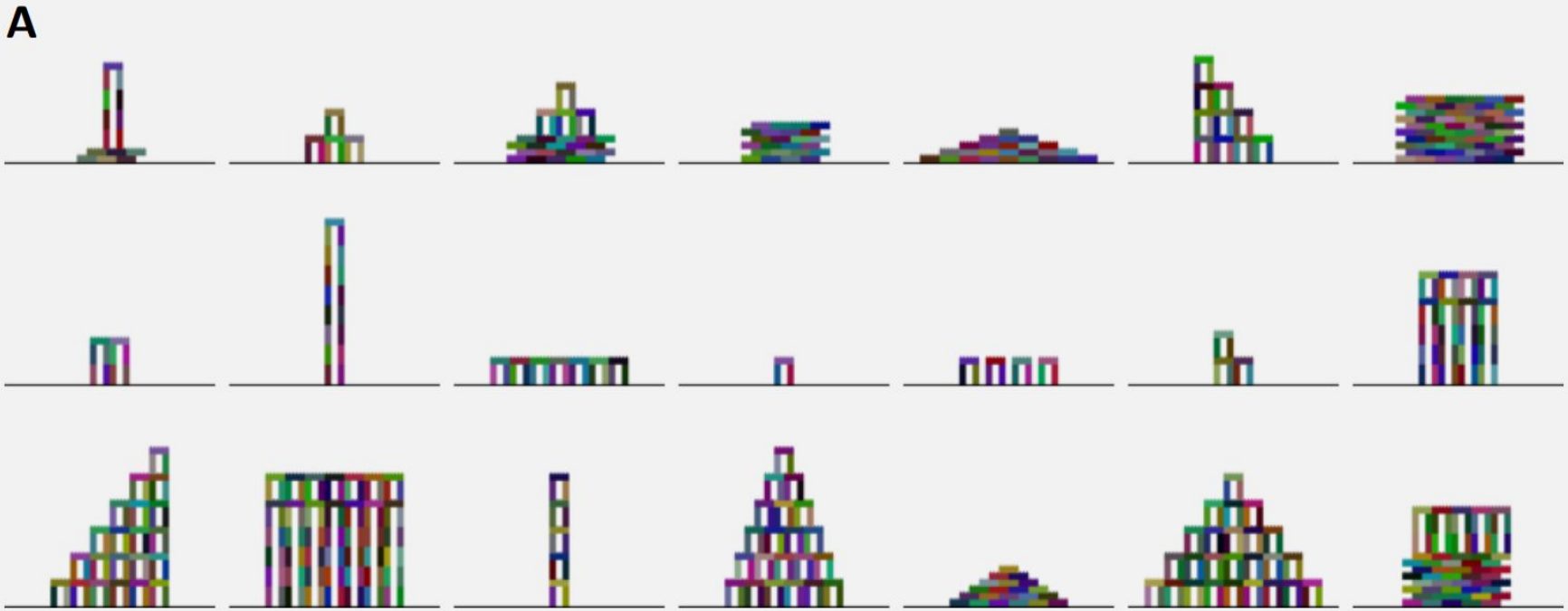
D



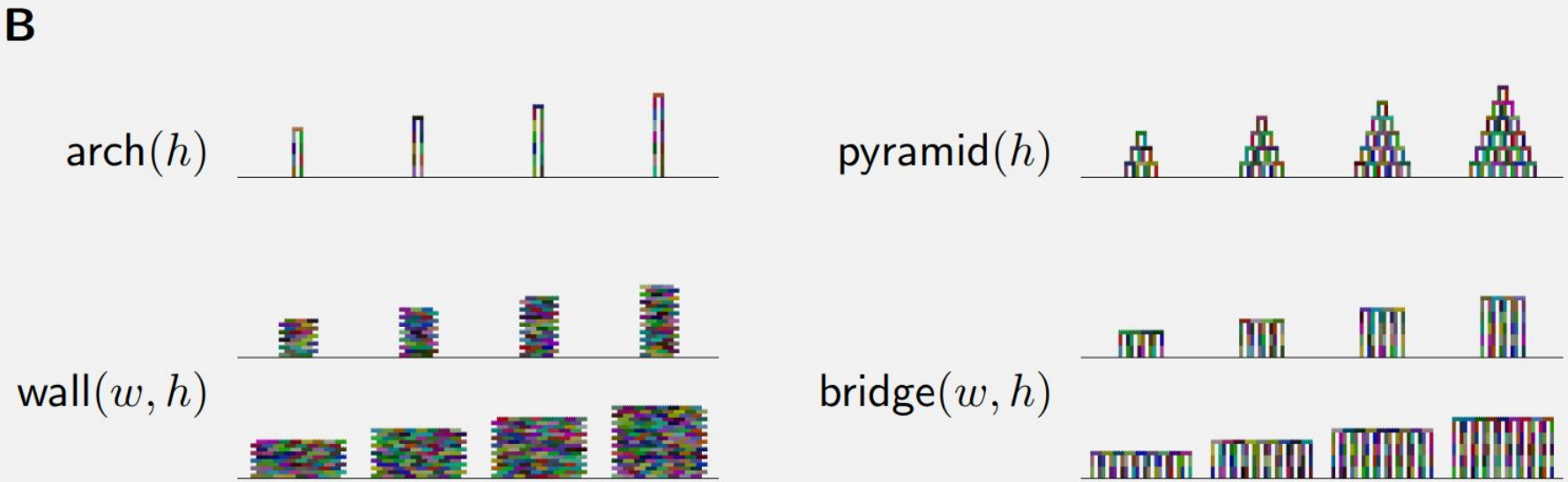
E



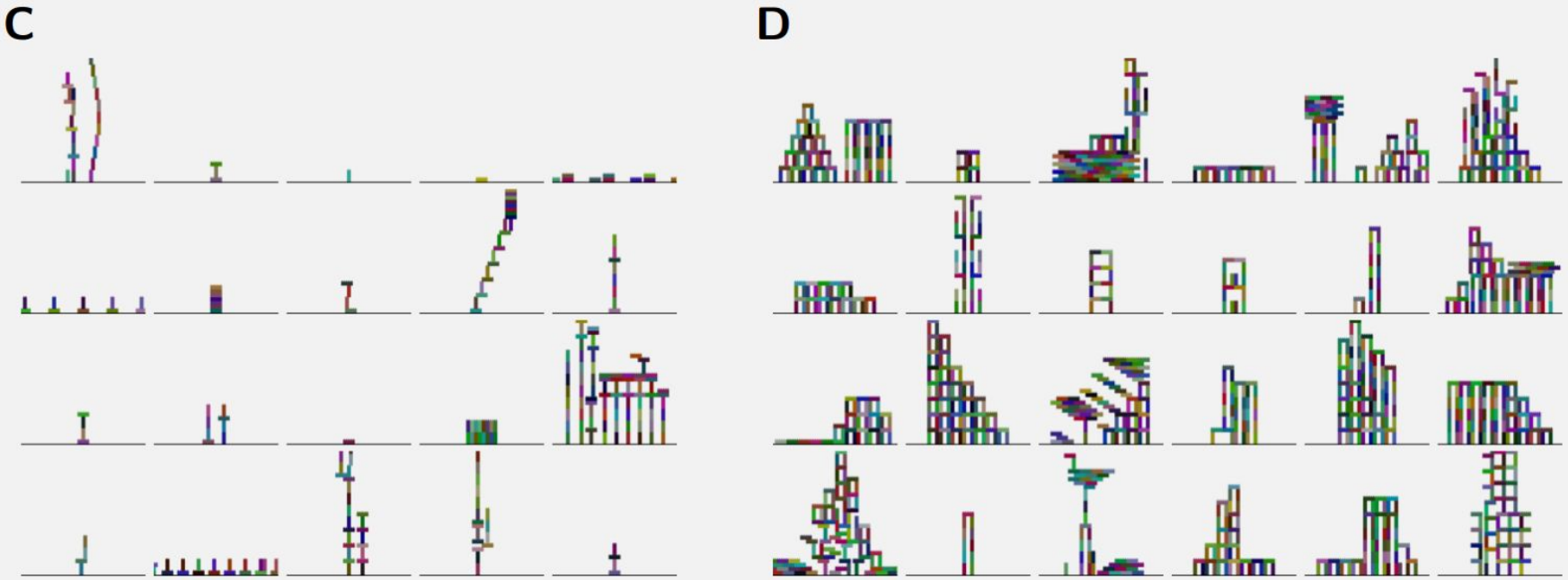
Tower Building Tasks



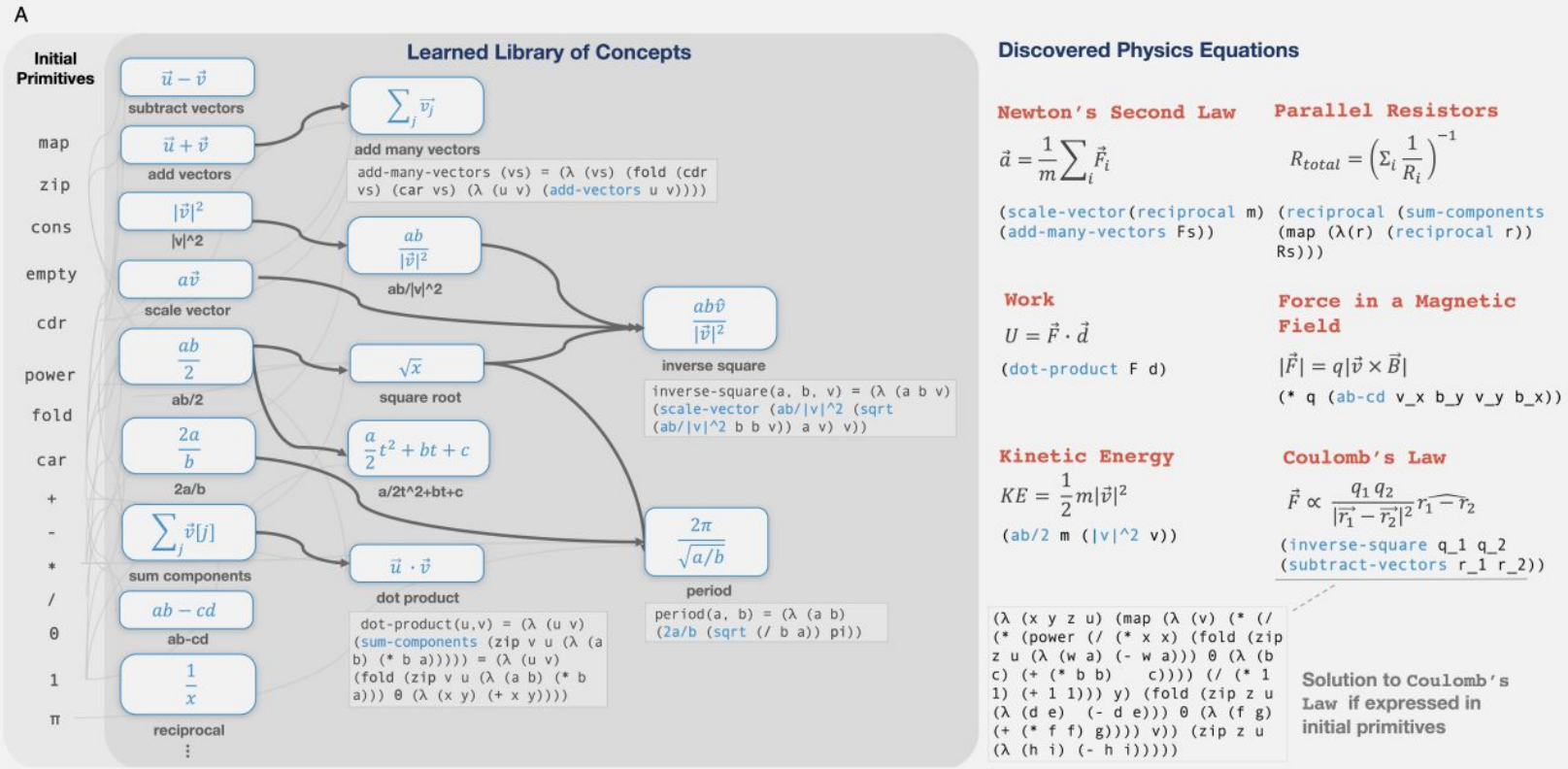
Tower Building: Emerge of Higher Orders



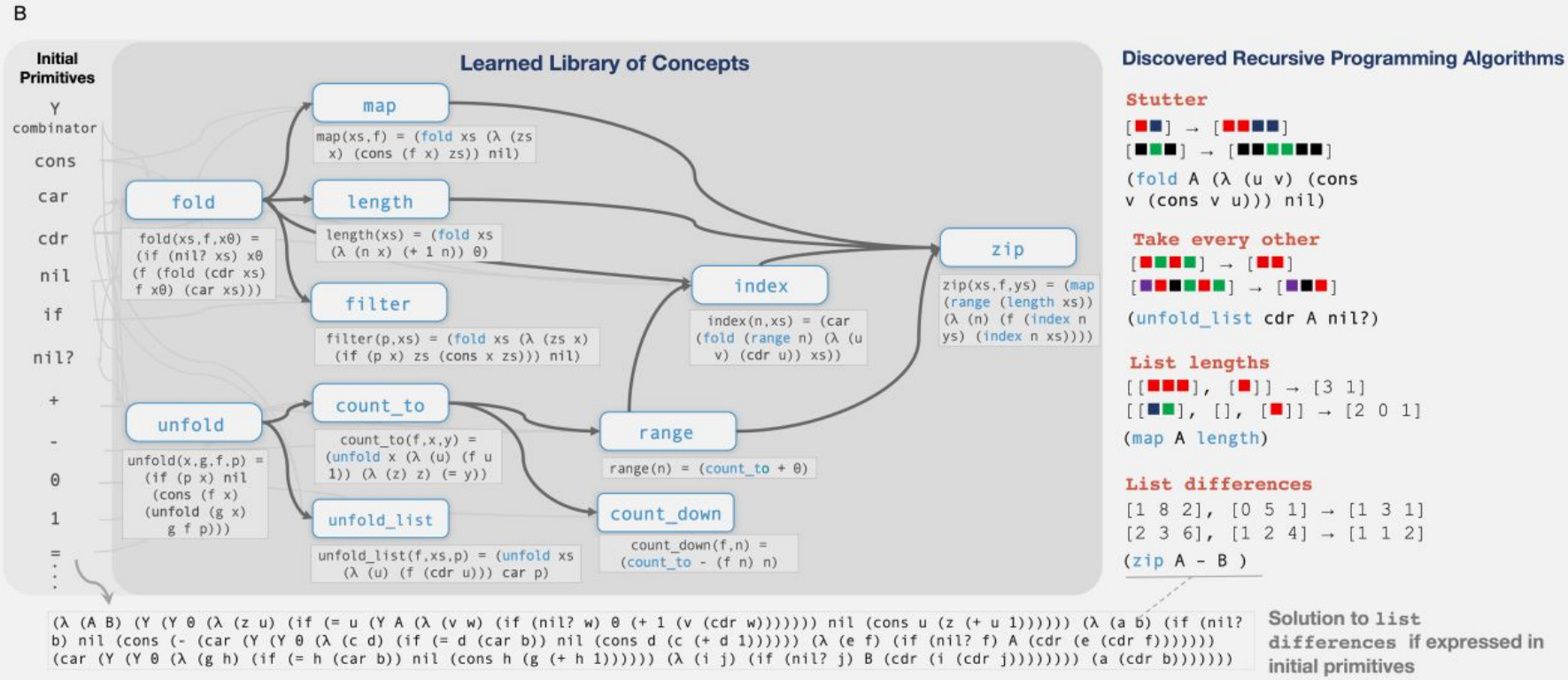
Tower Building: Dreaming



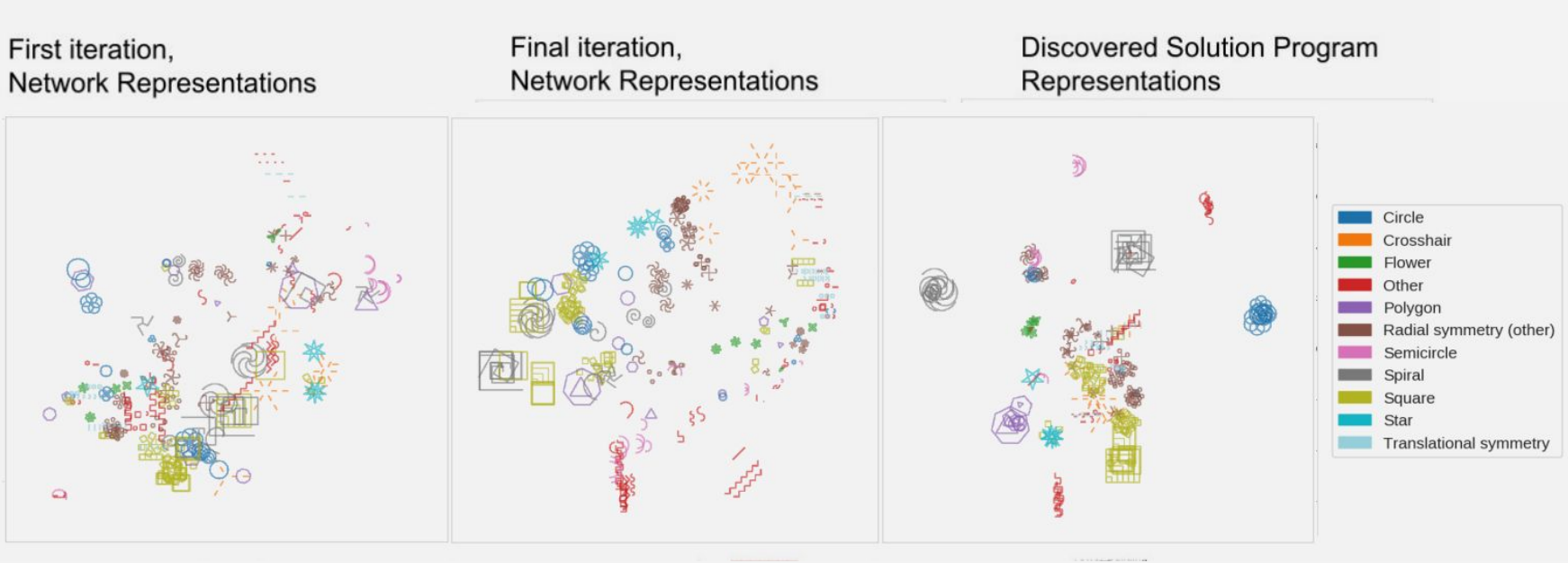
From learning libraries to learning languages



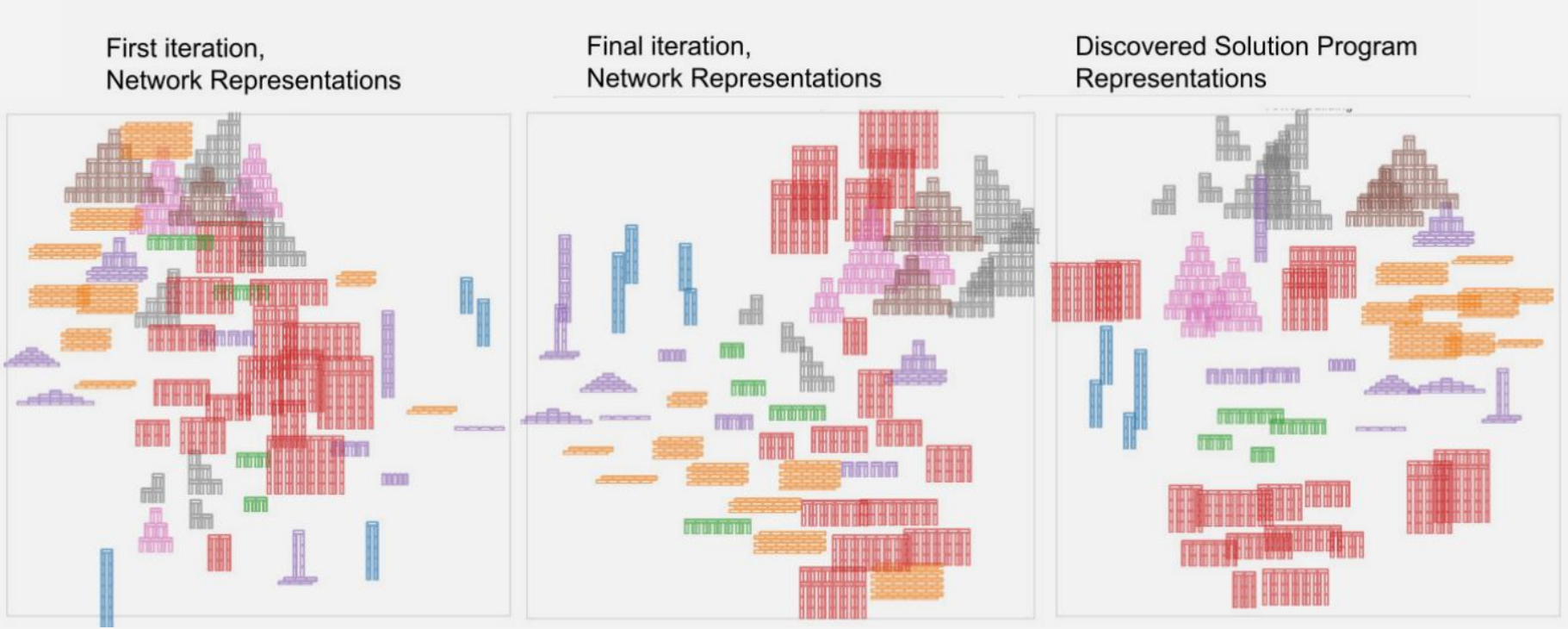
From learning libraries to learning languages



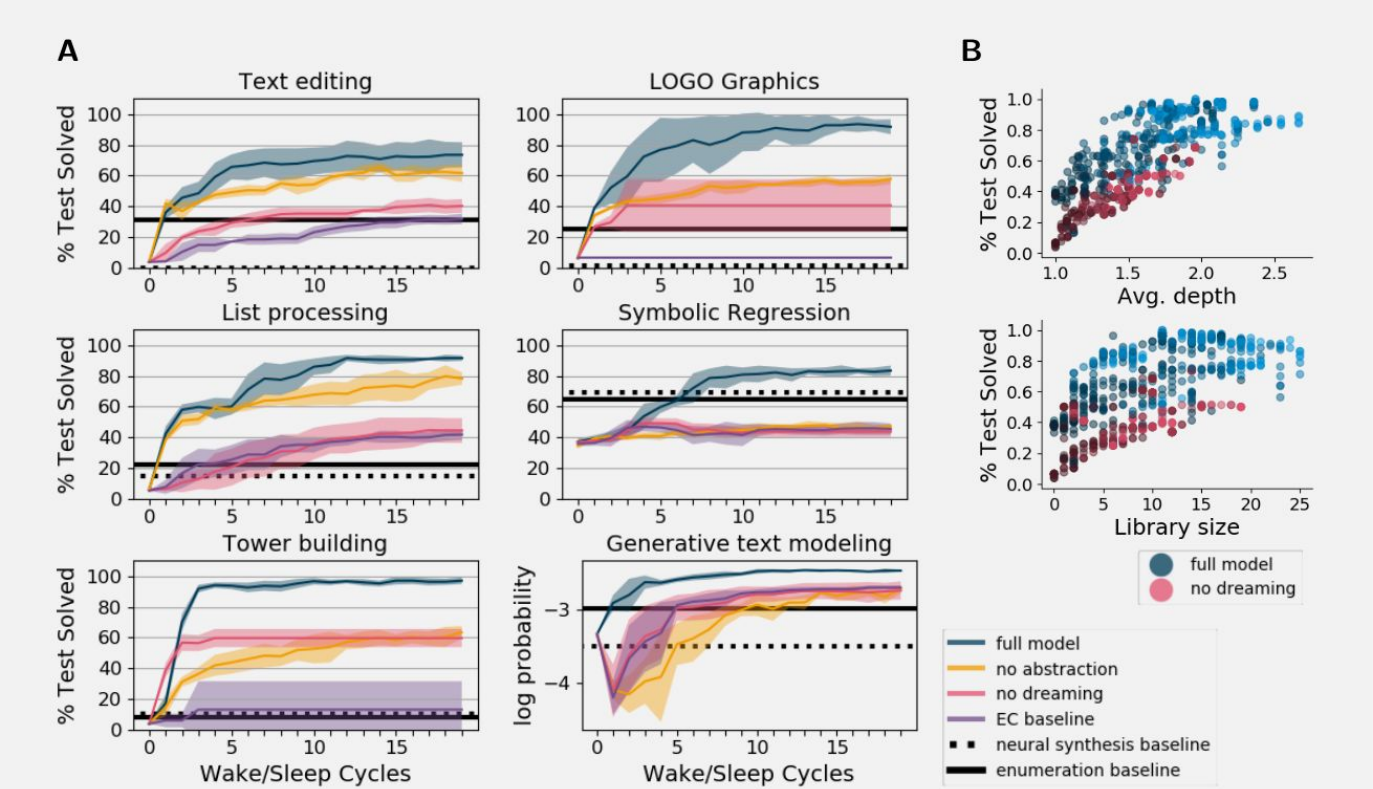
TSNE embedding



TSNE embedding



Role of Abstraction and Dreaming



Combining Induction and Transduction for Abstract Reasoning

Wen-Ding Li, Keya Hu, Carter Larsen, Yuqing Wu, Simon Alford, Caleb Woo, Spencer M. Dunn, Hao Tang, Michelangelo Naim, Dat Nguyen, Wei-Long Zheng, Zenna Tavares, Yewen Pu, Kevin Ellis

Dec 2024

Remember ARC?

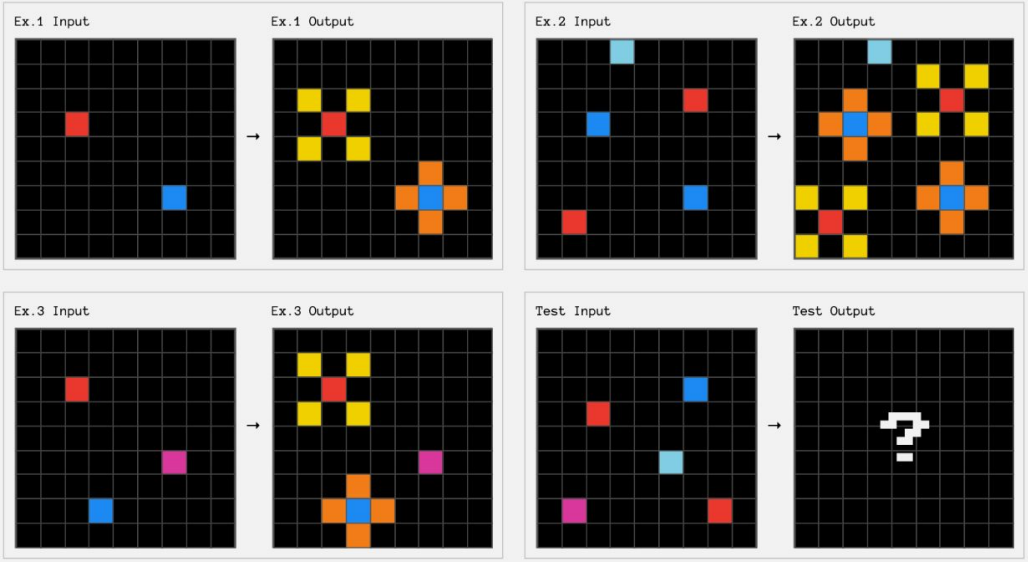
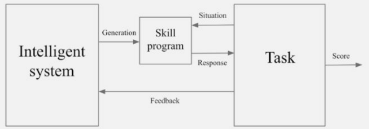
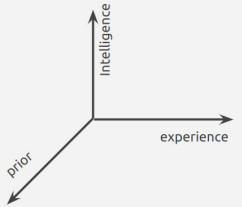


Figure 1: Example ARC-AGI task (0ca9ddb6).

Chollet Criticisms

- 👉 Intelligence is orthogonal to Prior and Experience.
- 👉 Measure intelligence? factor out priors and experience.
- 👉 Developer-aware generalization

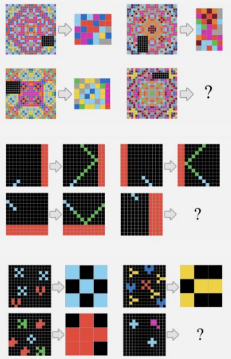




26

Chollet Criticisms: ARC Challenge

- 👉 Core Knowledge priors
- 👉 Focus on Reasoning and Abstraction
- 👉 Developer-Aware Generalization
- 👉 Suggestion for Using Domain Specific Language



33

Remember ARC?

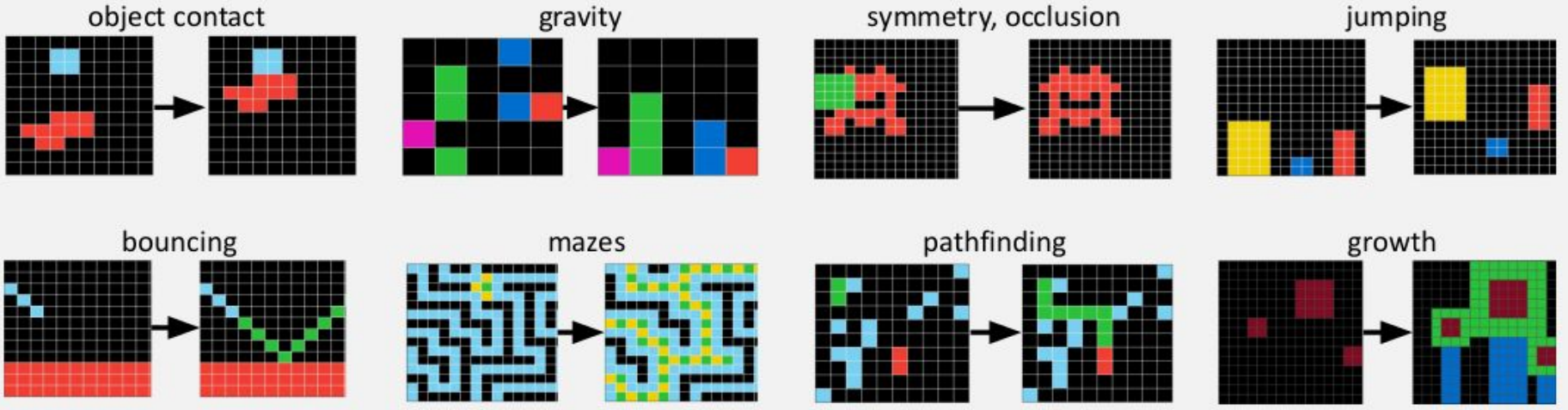
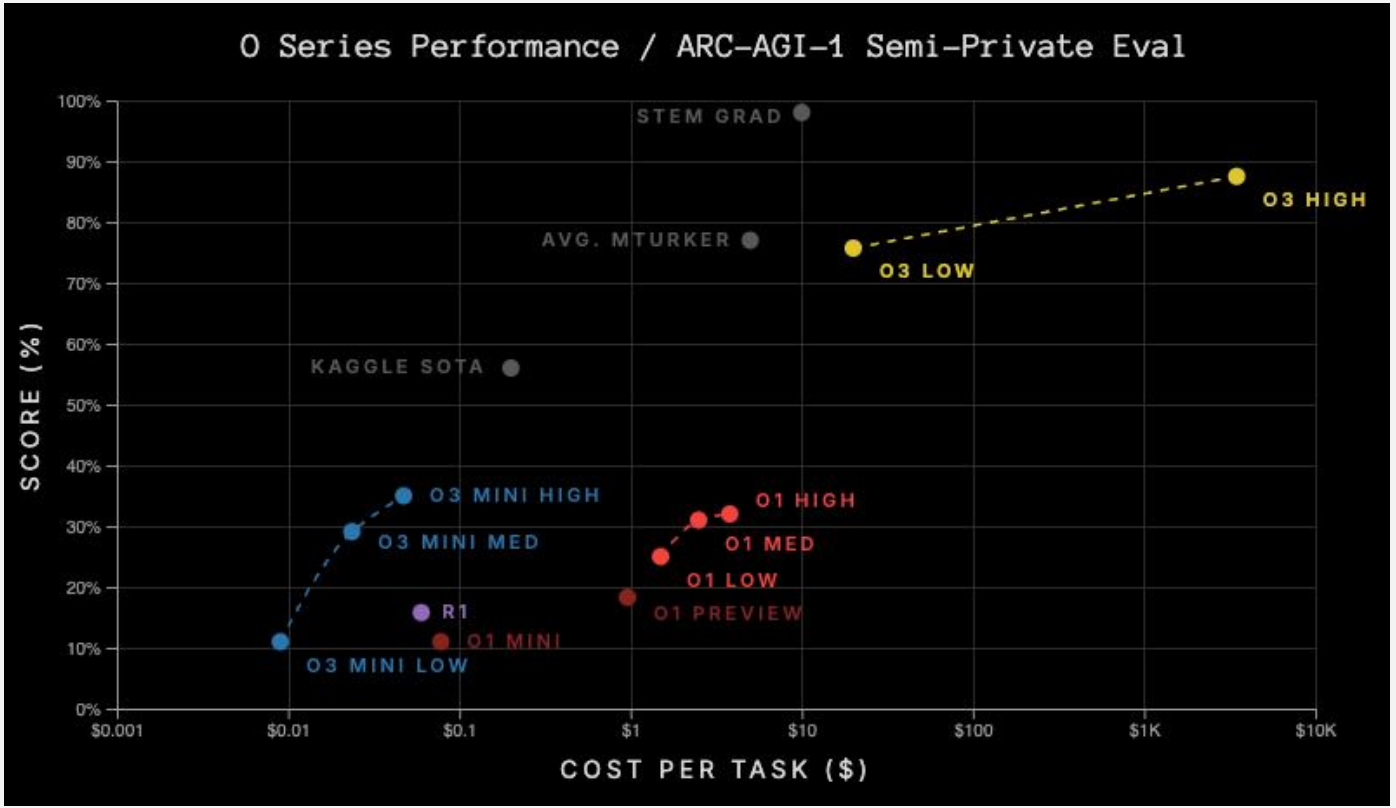


Figure 1: Few-shot learning tasks from the Abstraction and Reasoning Corpus (ARC). Each task typically has 2-5 input-output examples. Here we show just one input-output example per task.

ARC Challenge



Some Perspectives on ARC

🛠️ Deep learning-guided program synthesis

🛠️ Test-time training (TTT) for transductive models (Fine-tuning an LLM)

🛠️ Combining program synthesis together with transductive models

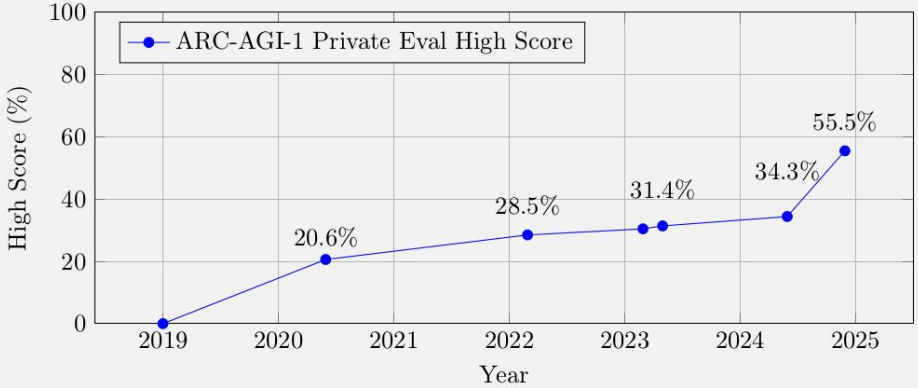


Figure 2: ARC-AGI-1 high scores over time

Some Perspectives on ARC

🛠️ Deep learning-guided program synthesis

- 👉 Brute-force search over a Domain Specific Language (DSL)
- 👉 LLM-powered program generation in open-ended languages
- 👉 LLM-guided discrete program search over a DSL
- 👉 LLM-powered iterative program debugging

Test-time training (TTT) for transductive models

Combining program synthesis together with transductive models

Some Perspectives on ARC

Deep learning-guided program synthesis

🛠️ Test-time training (TTT) for transductive models

- 👉 Data augmentation and alternative datasets
- 👉 Fine-tuning strategies
- 👉 Specialized 2D-aware architectures

Combining program synthesis together with transductive models

Some Perspectives on ARC

Deep learning-guided program synthesis

Test-time training (TTT) for transductive models

🛠️ Combining program synthesis together with transductive models

- 👉 Program synthesis, or “induction” (find mapping then apply it)
- 👉 Transduction (prompt llm to directly predict)

Induction vs Transduction

is it better to first infer a latent function that explains the examples, or is it better to directly predict new test outputs?

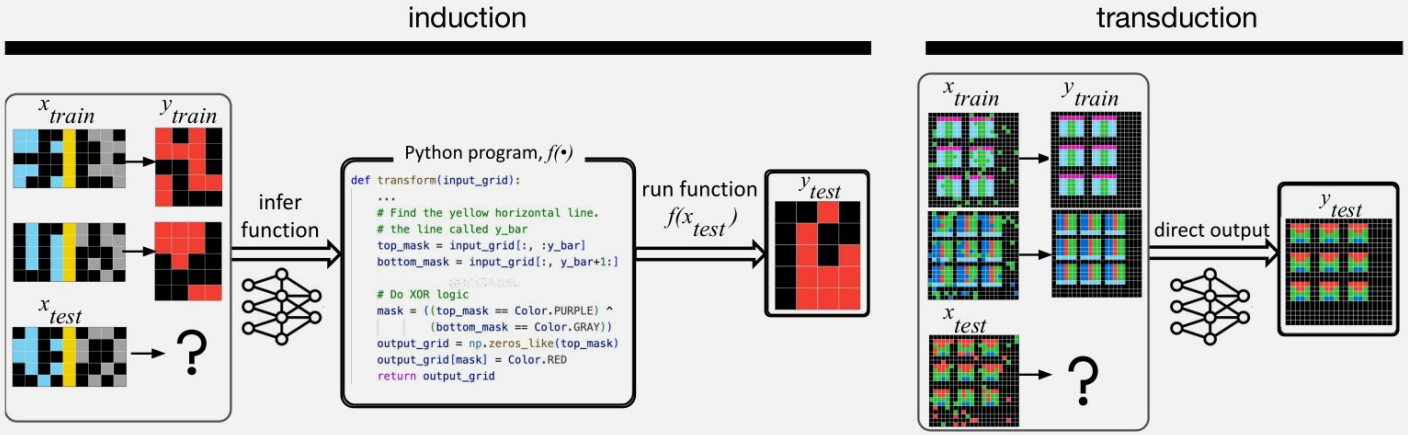


Figure 2: Induction generates an intermediate function f to explain training input-outputs. Transduction directly predicts the test output, for example using a neural network.

Induction vs Transduction

Training induction and transduction. Both types of models are trained via meta-learning. We assume a meta-learning dataset $\mathcal{D}$ of few-shot learning problems, each equipped with a ground-truth function f such that $f(x) = y$ for every x, y in $(\boldsymbol{x}_{\text{train}}, \boldsymbol{y}_{\text{train}})$ and $(x_{\text{test}}, y_{\text{test}})$. Inductive and transductive models are meta-trained to minimize the following losses:

$$\textsc{Transduction Loss} = \mathbb{E}_{(\boldsymbol{x}_{\text{train}}, \boldsymbol{y}_{\text{train}}, x_{\text{test}}, y_{\text{test}}, f) \sim \mathcal{D}} \left[-\log \mathtt{t}_\theta(y_{\text{test}} | \boldsymbol{x}_{\text{train}}, \boldsymbol{y}_{\text{train}}, x_{\text{test}}) \right] \quad (1)$$

$$\textsc{Induction Loss} = \mathbb{E}_{(\boldsymbol{x}_{\text{train}}, \boldsymbol{y}_{\text{train}}, x_{\text{test}}, y_{\text{test}}, f) \sim \mathcal{D}} \left[-\log \mathtt{i}_\theta(f | \boldsymbol{x}_{\text{train}}, \boldsymbol{y}_{\text{train}}, x_{\text{test}}) \right] \quad (2)$$

Testing induction and transduction. After meta-learning the models encounter a test-time few-shot learning task $(\boldsymbol{x}_{\text{train}}, \boldsymbol{y}_{\text{train}}, x_{\text{test}})$. Transductive models predict their most likely output for y_{test} (approximated via beam search). Inductive models sample a test-time budget of B functions $f_1 \cdots f_B$, which are filtered by $(\boldsymbol{x}_{\text{train}}, \boldsymbol{y}_{\text{train}})$, and finally used to predict $y_{\text{test}} = f(x_{\text{test}})$. Writing $\hat{y}_{\text{test}}$ for the predicted test output:

$$\textsc{Transduction}: \hat{y}_{\text{test}} = \arg\max_{y \in \mathcal{Y}} \mathtt{t}_\theta(y | \boldsymbol{x}_{\text{train}}, \boldsymbol{y}_{\text{train}}, x_{\text{test}}) \quad (3)$$

$$\textsc{Induction}: \hat{y}_{\text{test}} \sim \text{Uniform}(\mathcal{F}) \quad (4)$$

$$\text{where } \mathcal{F} = \{ f_b(x_{\text{test}}) \ : \ \text{for } 1 \leq b \leq B \text{ if } f_b(\boldsymbol{x}_{\text{train}}) = \boldsymbol{y}_{\text{train}} \}$$

$$f_b \sim \mathtt{i}_\theta(f | \boldsymbol{x}_{\text{train}}, \boldsymbol{y}_{\text{train}}, x_{\text{test}})$$

Combine Induction and Transduction

Combining induction and transduction. Induction allows checking candidate hypotheses against the training examples. Therefore, we know when induction has found a plausible solution–but sometimes it fails to find any solution. Transduction has the opposite property: We can't check if its predictions match the training examples, but it always offers a candidate answer. Therefore we ensemble by attempting induction first, then transduction if none of the candidate hypotheses explained the examples:

$$\begin{aligned} \text{ENSEMBLE: } \hat{y}_{\text{test}} &\sim \text{Uniform}\left(\mathcal{F}\right) \text{ if } \mathcal{F} \neq \varnothing \\ \hat{y}_{\text{test}} &= \arg\max_{y \in \mathcal{Y}} \mathtt{t}_\theta(y | \boldsymbol{x}_{\text{train}}, \boldsymbol{y}_{\text{train}}, x_{\text{test}}) \text{ if } \mathcal{F} = \varnothing \end{aligned} \tag{5}$$

Creating datasets for induction and transduction

At a high level, our dataset grows out of 100 manually-written Python programs, each of which both solves a given task (function f), and also randomly generates new input grids. We call these 100 manually-written programs **seeds.** Each seed is commented with natural language describing the core intuitions behind the problem. We then use a large language model to mutate and recombine the seeds, producing many thousands of programs (Figure 3).

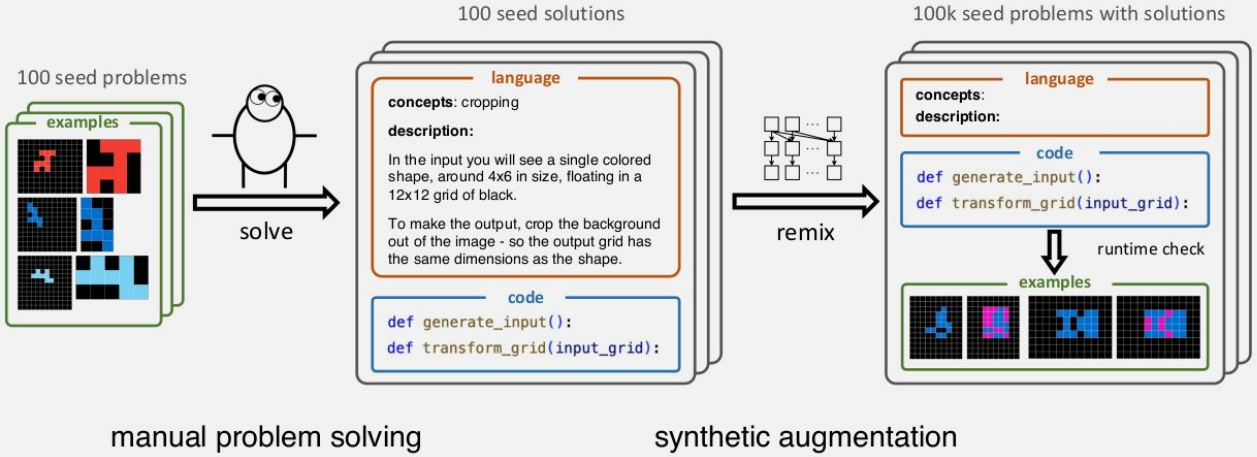


Figure 3: Synthetic data generation pipeline, starting with human-written programs (seeds).

Creating datasets for induction and transduction

Figure 4 illustrates example problems generated by our pipeline, with further examples **visualized at this link.** Unless otherwise mentioned, we create synthetic datasets with GPT4o-mini and ada-002.

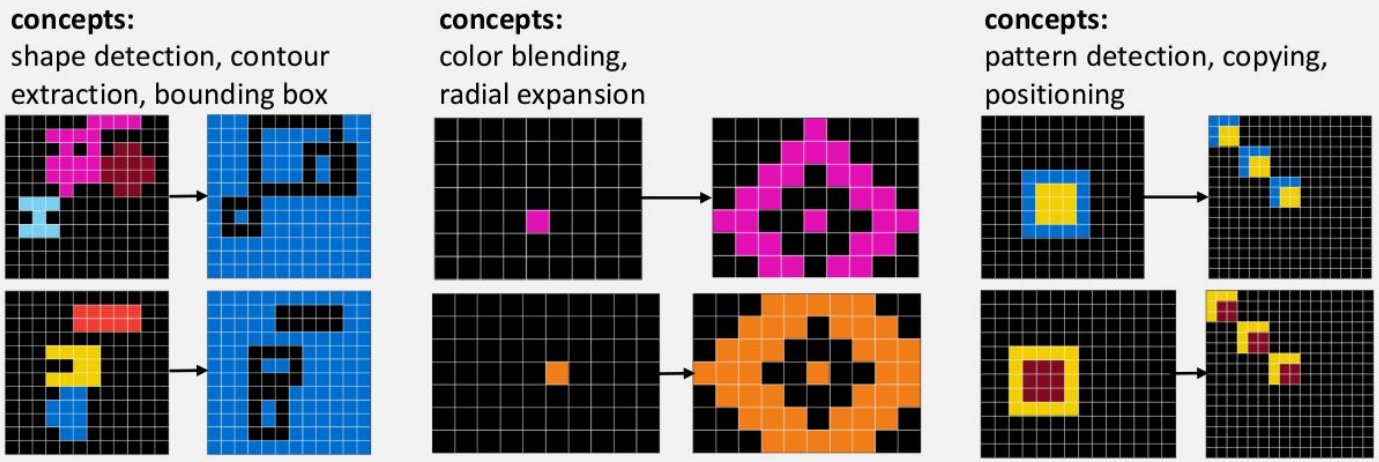


Figure 4: Example synthetic ARC problems generated by our pipeline. Concepts are generated in a comment near the top of the Python script as part of the natural language description of the seed.

Induction and Transduction are strongly complementary

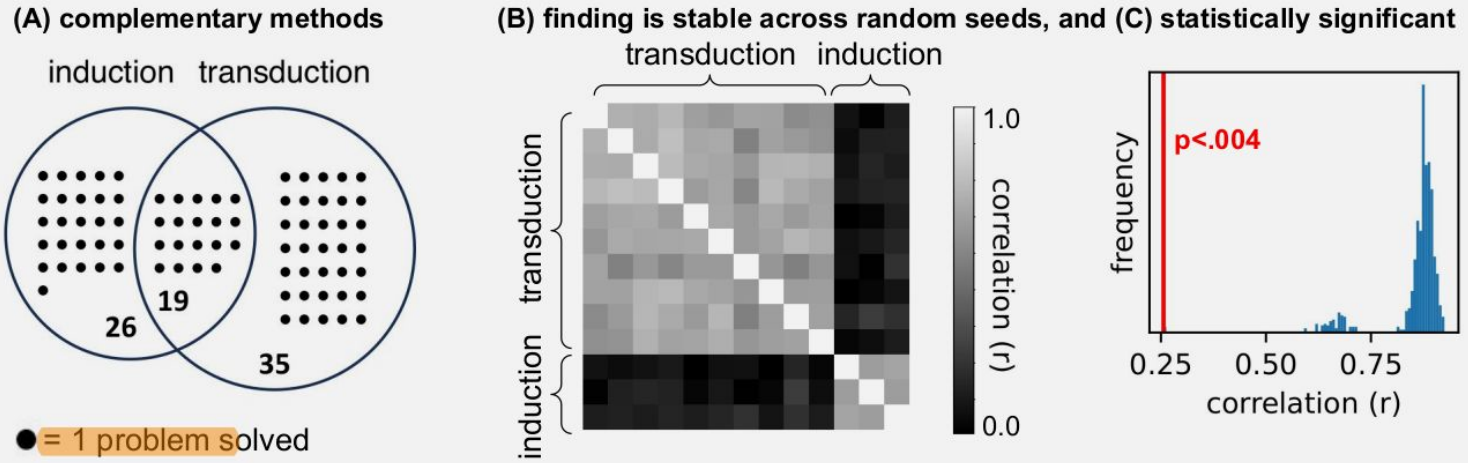
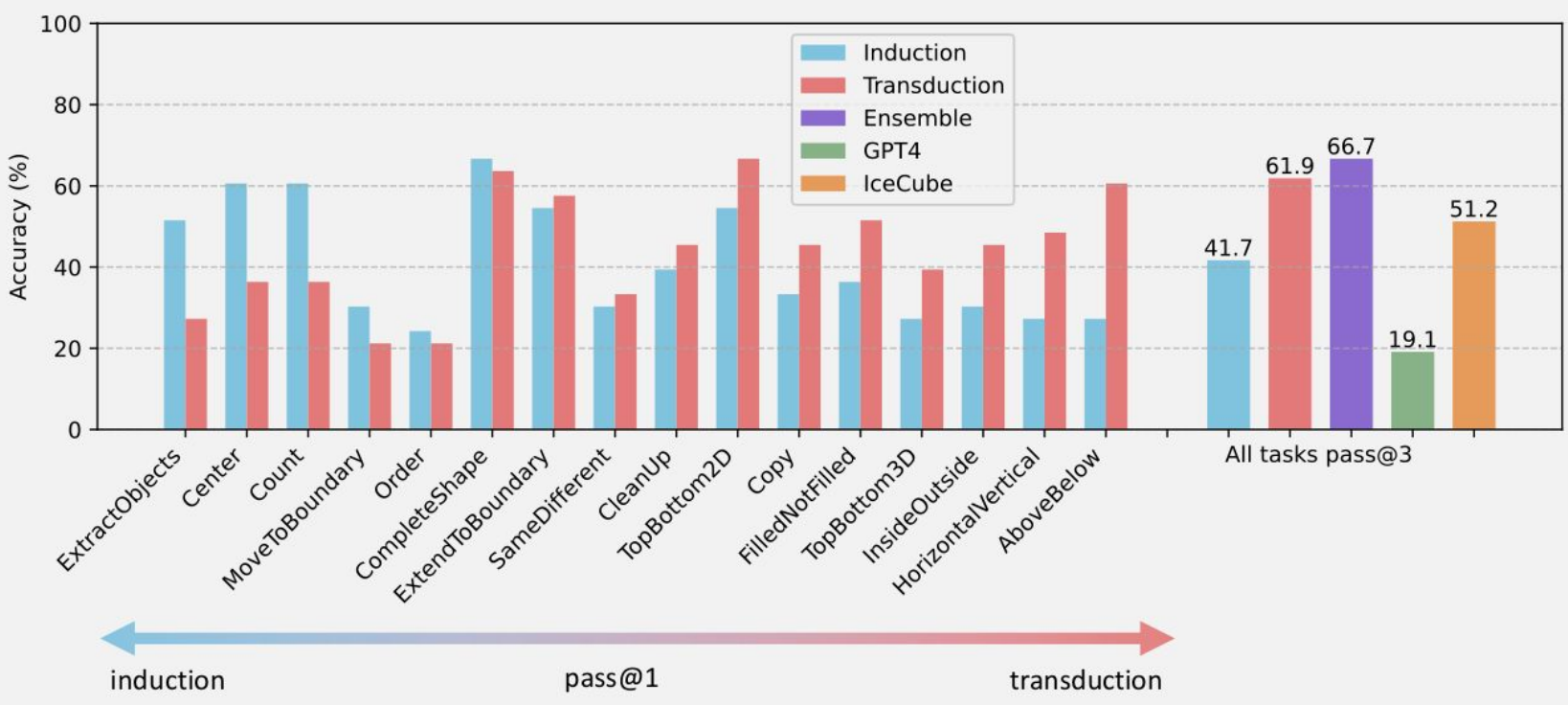
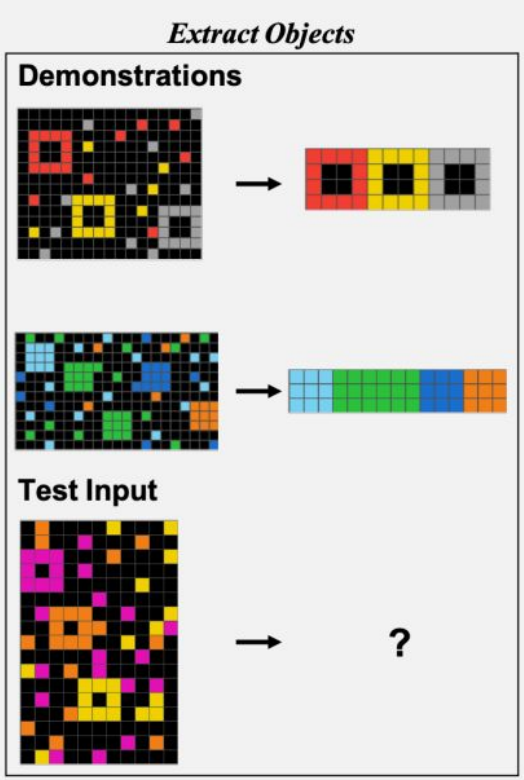


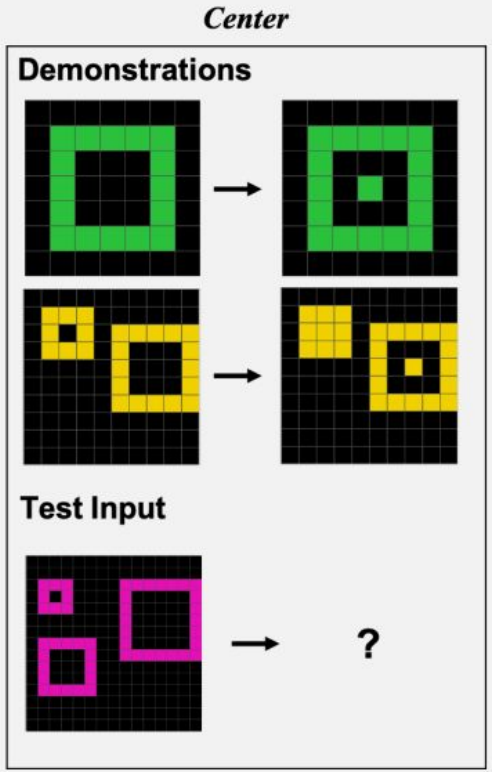
Figure 5: (A) Induction and transduction solve different problems, where solve means predicting the right output given 2 tries. Venn diagram for models trained on 100k synthetic problems generated using gpt4o-mini. (B) Training many models with different random seeds, and then measuring the correlation between solved tasks by different models. Solved tasks strongly correlates with other models of the same class but not the other class. (C) Statistical significance test evaluating the null hypothesis that correlation is independent of whether a model is inductive/transductive.

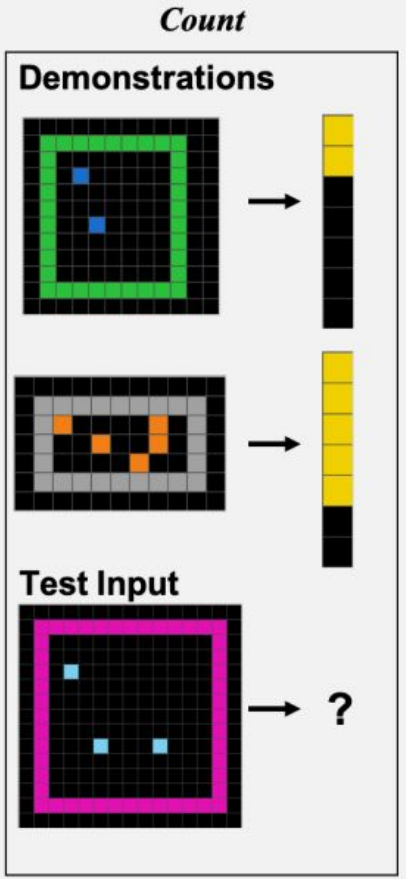
Which concepts are easier for the models?



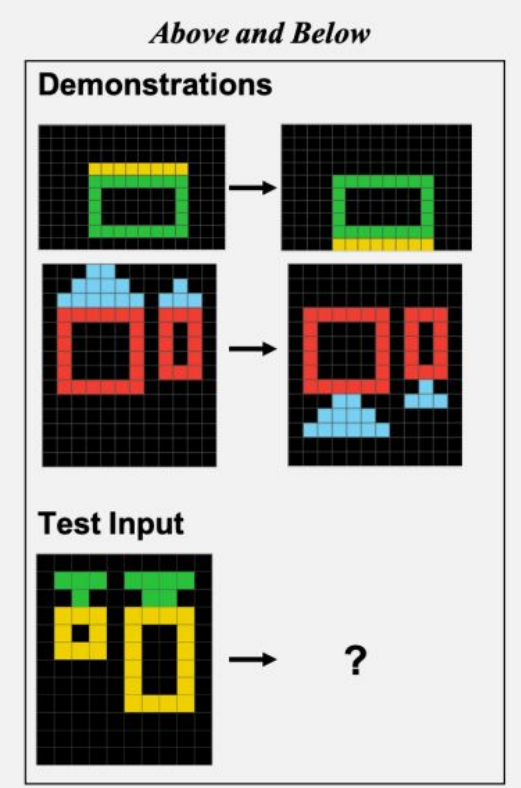
Induction

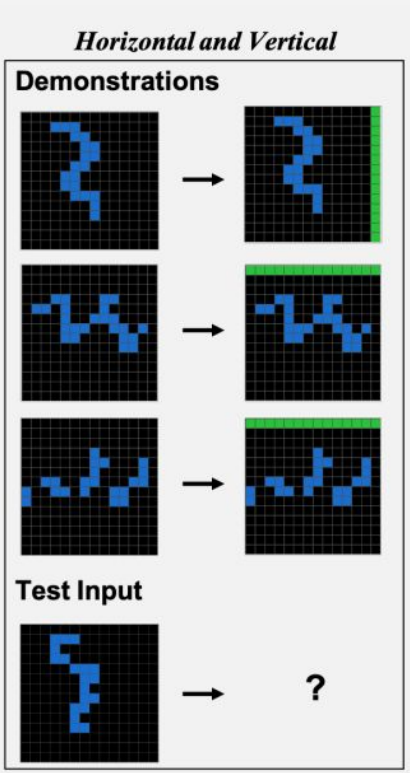

Extract Objects
Demonstrations
Test Input
?

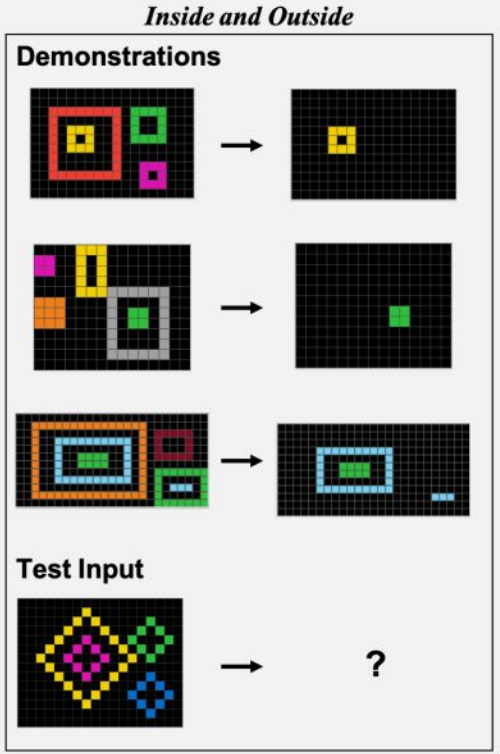

Center
Demonstrations
Test Input
?


Count
Demonstrations
Test Input
?

Transduction







Performance scales with dataset size, but quickly saturates with increasing number of seeds.

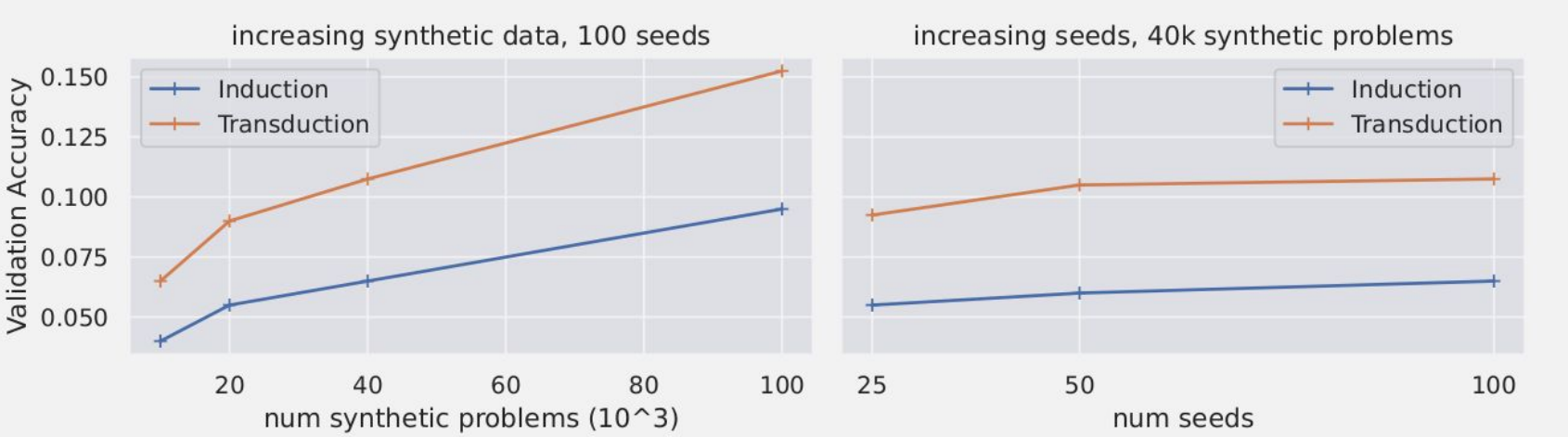


Figure 7: Increased manual human effort (# seeds) does not significantly increase performance, but increasing compute spent generating synthetic data increases performance of the final model.

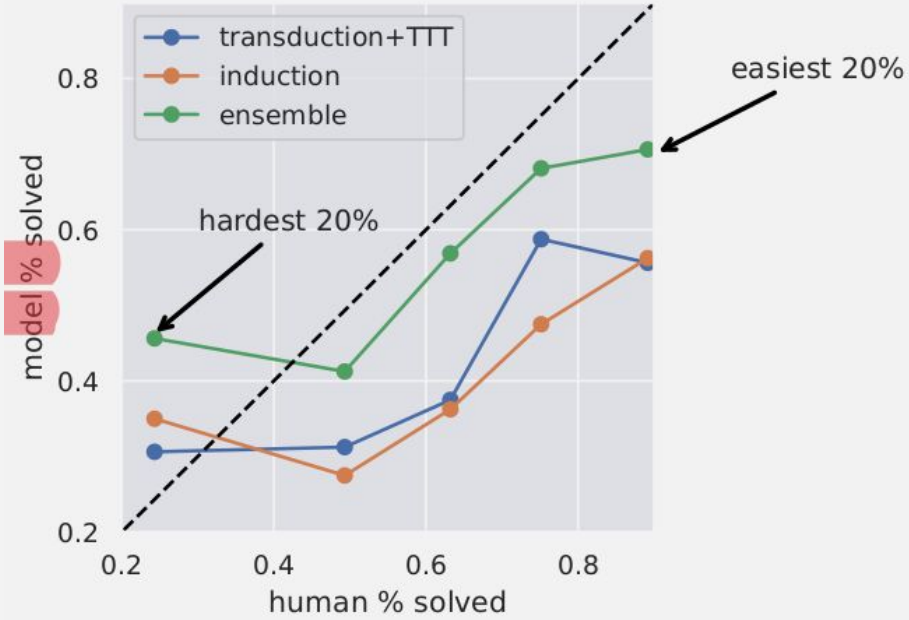
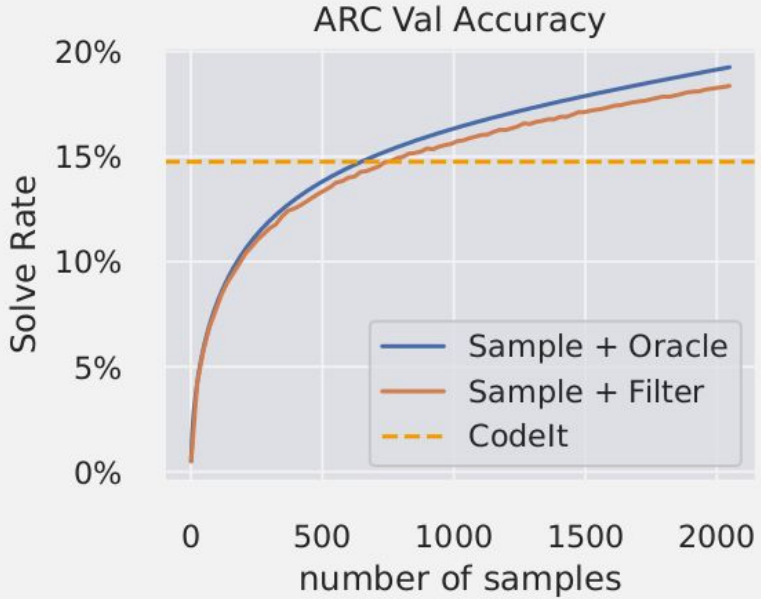


Figure 9: Human vs model performance across 5 difficulty levels. The easiest difficulty level contains problems in the top 20% of human accuracy, and the hardest difficulty level contains the 20% of problems with the lowest human accuracy.



Searching Latent Program Spaces

Clément Bonnet, Matthew V Macfarlane

Nov 2024

Program Synthesis Problem

🧩 Symbolic Approaches: suffer from a combinatorial search space

🧩 Neural Guided Program Synthesis: difficult to train models to perform program synthesis in one shot

🔍 Lets Test Time Search:

- 📤 Search on Output Program?
- 💤 Search on Latent Programs? (via backpropagation)

Latent Program Network

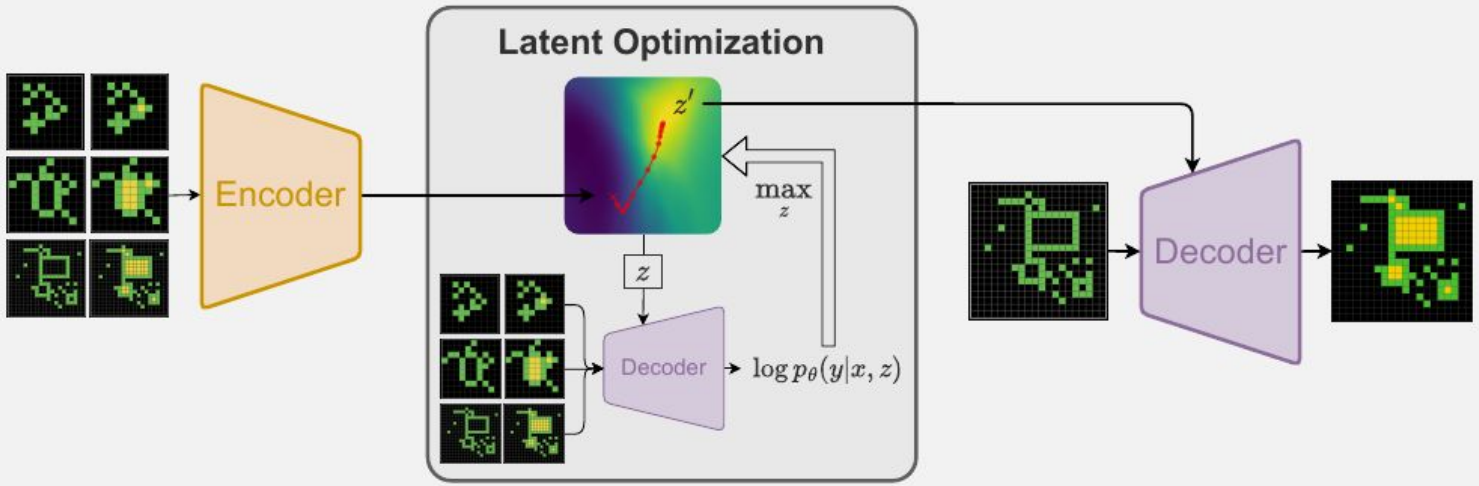


Figure 1: Inference of the Latent Program Network (LPN) model. (Left): the encoder maps I/O pairs to a latent space of encoded programs. (Middle): the latent program is refined during an optimization process to best explain the given I/O pairs. (Right): the decoder executes the latent program to generate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

Sth happens!

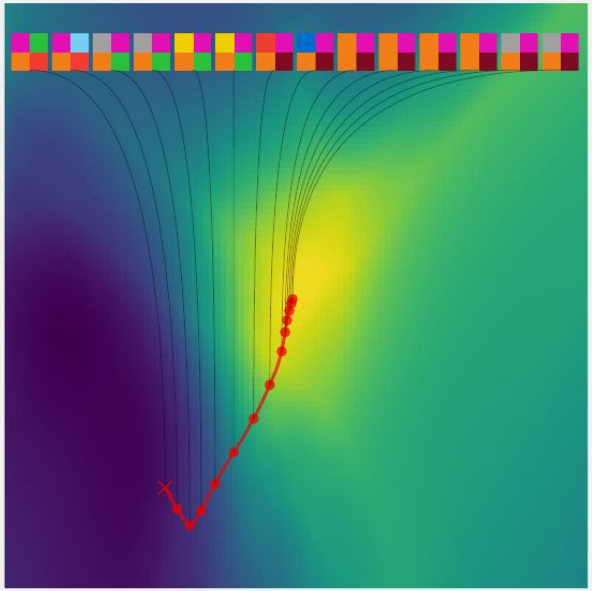


Figure 13: Gradient ascent trajectory as shown in figure 1.

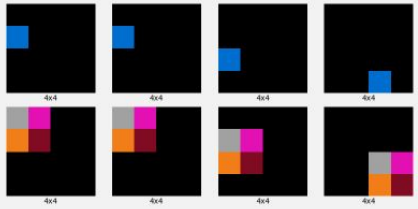


Figure 14: Small pattern task

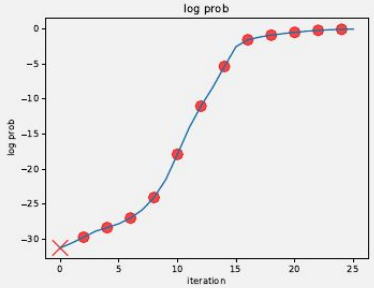


Figure 15: Log prob maximization during latent search

Figure 16: Gradient ascent trajectory detailed

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

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

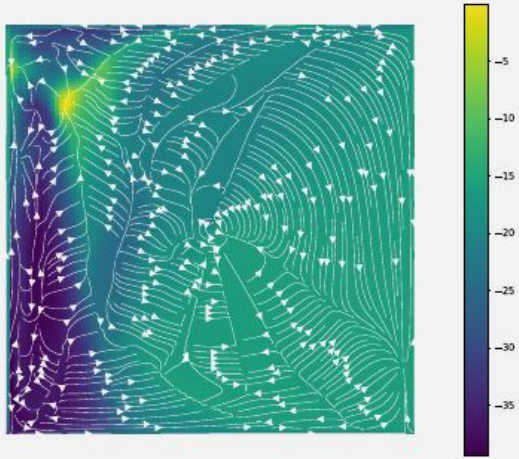


Figure 3: Gradient field of the decoder log-likelihood $f(z) = \log p_\theta(y|x, z)$

Algorithm 2 LPN Training with Gradient Ascent Latent Optimization

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

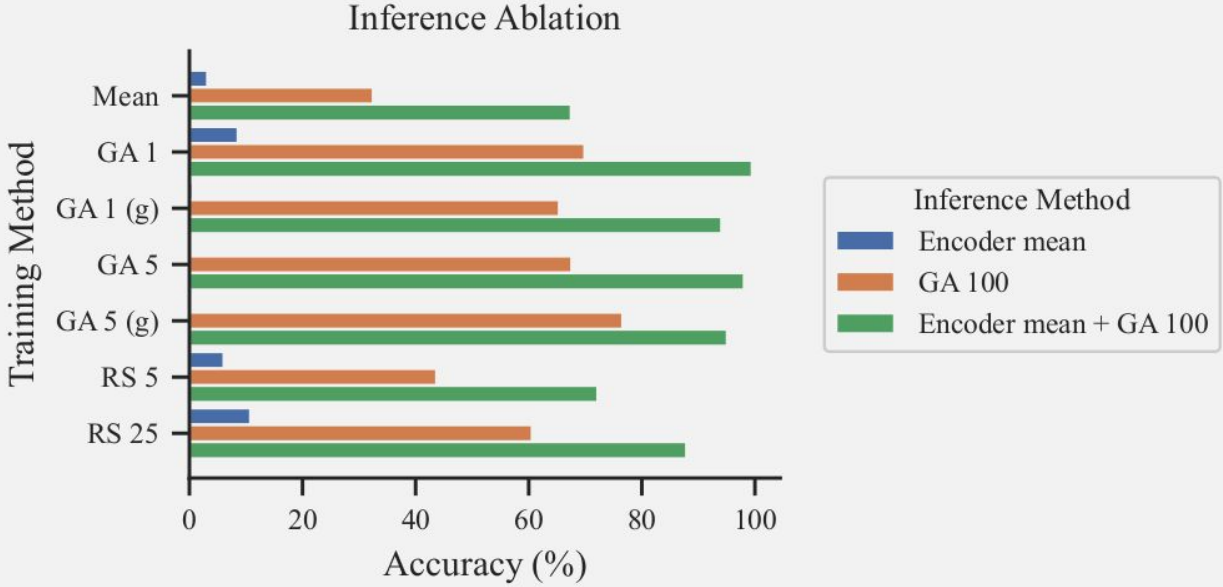


Figure 8: Ablation on the initialization of latent optimization and the role of the encoder. *Encoder mean* signifies that latent optimization is initialized using the encoder mean latents. GA 100 stands for 100 steps of gradient ascent during latent optimization. The results demonstrate the importance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