ETRM: Encoder-based Tiny Recursive Models for Few-Shot Abstract Reasoning

Barış Deniz Sağlam
1558410

January 18, 2026

1 Introduction

1.1 Problem context

The Abstraction and Reasoning Corpus (ARC-AGI) benchmark Chollet [2019] was designed to test abstract reasoning and few-shot learning capabilities. Each task presents 2–5 demonstration pairs showing an input–output transformation, followed by test inputs where the model must infer and apply the same transformation. This format explicitly tests whether systems can extract transformation rules from examples and generalize to new inputs.

Recent work has achieved surprising success on this benchmark. The Tiny Recursive Model (TRM) Jolicoeur-Martineau [2025] achieves 45% accuracy on ARC-AGI-1 with only 7 million parameters, outperforming much larger models including GPT-4. TRM's key innovation is recursive reasoning: it iteratively refines predictions through nested loops with deep supervision at each step.

1.2 The hidden problem

Analysis by the ARC Prize Foundation ARC Prize Foundation [2025] and subsequent work revealed a critical limitation in TRM's approach. TRM uses *puzzle_id* embeddings: each task is assigned a unique learned vector that conditions the model's predictions. During inference the model receives only the input grid and this puzzle_id—it never actually processes the demonstration pairs to extract transformation rules.

Consequently, transformation rules are effectively memorized in embedding weights during training rather than inferred from demonstrations at test time. The evidence is compelling: when researchers trained a related model only on the 400 evaluation tasks, performance dropped from 41% to 31% ARC Prize Foundation [2025]—still remarkably high given the restricted training set. Furthermore, replacing the puzzle_id with a random token results in 0% accuracy Roye-Azar et al. [2025], showing the model cannot function without task-specific embeddings it has already learned.

1.3 Research question

This project asks: **Can we replace task-specific embeddings with an encoder that extracts transformation rules directly from demonstration pairs at test time?** Such an approach would enable true few-shot generalization to novel tasks never seen during training—the original intent of the ARC benchmark. The architectural comparison between TRM's embedding-based approach and our encoder-based approach is illustrated in Section 3.

1.4 Contributions

We present ETRM (Encoder-based TRM), an architecture that replaces TRM's puzzle_id lookup with a neural encoder that processes demonstration pairs.[1] Our main contributions are:

1. **Encoder-based architecture.** We design and implement ETRM, which computes task representations from demonstrations rather than retrieving memorized embeddings.
2. **Systematic evaluation of encoder designs.** We evaluate three encoder paradigms—deterministic transformers, variational encoders, and iterative encoding with adaptive computation—to understand which architectural choices matter.
3. **Strict train/eval separation.** We implement a protocol where evaluation puzzles and their demonstrations are never seen during training, enabling measurement of true generalization.
4. **Analysis of failure modes.** We identify and analyze why encoder-based approaches struggle, including gradient starvation during training and the fundamental asymmetry between learning embeddings over many gradient updates versus extracting rules in a single forward pass.

1.5 Key findings

Our results are primarily negative but informative:

- All ETRM variants achieve moderate-to-high training accuracy (40–79%) but near-zero test accuracy ($< 1\%$) on held-out puzzles.
- The deterministic encoder shows low cross-sample variance (0.15–0.36), suggesting the decoder learns to ignore uninformative encoder outputs and instead memorizes training patterns.
- Variational encoding increases representation diversity (approximately $10\times$ higher variance) but does not improve generalization—variance alone is insufficient.
- Iterative encoding (ETRM-TRM) also fails, indicating the problem is not simply insufficient computation but the absence of a feedback signal during refinement.

These results highlight a fundamental asymmetry: TRM refines each puzzle embedding over hundreds of thousands of gradient updates during training, while we ask the encoder to extract equivalent information in a single forward pass with no task-specific supervision. The most promising path forward appears to be adding test-time optimization with self-supervised signals, as demonstrated by Latent Program Networks Bonnet and Macfarlane [2024].

2 Background and Related Work

2.1 The ARC-AGI benchmark

The Abstraction and Reasoning Corpus (ARC) was introduced by Chollet Chollet [2019] as a benchmark designed to measure abstract reasoning and skill-acquisition efficiency in AI systems. Unlike most machine learning benchmarks that test interpolation within a training distribution, ARC explicitly requires out-of-distribution generalization: the hidden test set contains tasks that follow different underlying rules than those seen during training.

[1] Code available at https://github.com/bdsaglam/etrm

Each ARC task consists of 2–5 demonstration pairs showing an input grid and its corresponding output grid, followed by 1–2 test inputs for which the system must predict the output. Grids can be up to 30×30 cells, with each cell containing one of 10 possible values (typically rendered as colors). The demonstration pairs implicitly specify a transformation rule, and the challenge is to infer this rule and apply it correctly to novel test inputs. Figure 1 shows an example task where the rule involves extracting and recoloring a shape.

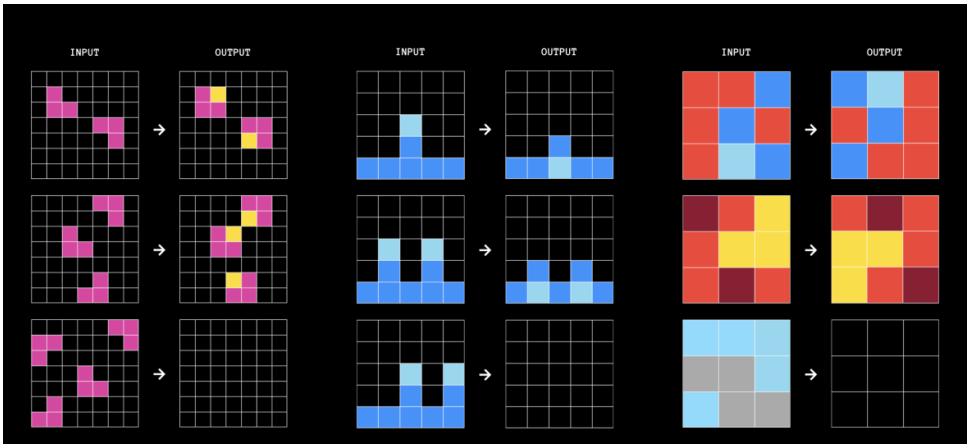

Figure 1: Example ARC puzzles. Each task shows 2–5 demonstration pairs and test inputs requiring the model to infer and apply the transformation rule.

ARC is challenging for two reasons. First, every task follows a different underlying logic—there is no single algorithm that solves all tasks. Second, compute efficiency is an explicit goal: competition submissions must operate within fixed hardware constraints and capped time budgets, preventing brute-force scaling as a solution strategy. Chollet argues that solving ARC requires "core knowledge priors" such as objectness, elementary physics (cohesion, persistence), goal-directedness, basic numerics, and elementary geometry and topology Chollet [2019].

2.2 Framing ARC as program synthesis

A productive way to understand ARC is through the lens of program synthesis: given demonstration input–output pairs, find the shortest program P such that $P(\text{input}) = \text{output}$ for all demonstrations Li et al. [2024]. This framing unifies disparate approaches and clarifies trade-offs.

Following Li et al. and Hemens Li et al. [2024], Hemens [2025], we organize approaches along three orthogonal axes inspired by neurosymbolic programming Chaudhuri et al. [2021]:

Inference mode – Induction vs. Transduction: Inductive approaches first infer an explicit program or rule from demonstrations, then execute it on test inputs (interpretable but requires program search). Transductive approaches directly predict test outputs from demonstrations and test inputs without constructing an intermediate program (implicit rule in activations). Li et al. report a "skill split": induction excels at counting and long-horizon logic, while transduction handles noisy inputs and gestalt/topological reasoning better Li et al. [2024].

Program representation – Discrete vs. Continuous: Discrete representations (DSLs, code) offer precision and compositionality but are non-differentiable and suffer combinatorial search. Continuous representations (neural weights, latent vectors, learned embeddings) enable gradient-based optimization but sacrifice formal guarantees Chaudhuri et al. [2021].

Program search – Heuristic vs. Learned: Hand-crafted heuristics (enumeration, MDL) are efficient when engineered; gradient-based search and learned search procedures (recursive refinement, thinking models) leverage differentiability and data-driven discovery but can get stuck in local optima Hemens [2025].

ETRM (this work) sits in the transductive, continuous, learned-search region: encoder-derived latent vectors decoded by a recursive refinement decoder (Section 3).

2.3 Overview of approaches to ARC

Table 1 summarizes major ARC approaches by program representation and search strategy.

Table 1: Taxonomy of ARC approaches by program representation and search strategy.

Approach	Program representation	Program search
DSL search (Icecuber)	Discrete (custom DSL)	Hand-crafted heuristics
LLM program synthesis	Discrete (Python)	LLM sampling + refinement
LLM + TTT (ARChitects, NVARC)	Neural (weights + prompt)	Gradient-based (SGD)
Latent Program Networks	Neural (latent vector)	Gradient-based (test-time)
TRM/HRM	Neural (puzzle embedding)	Learned (recursive decoder)
Thinking models	Neural (weights + thinking tokens)	Learned (implicit search)

Recent ARC Prize analyses Chollet et al. [2024], Knoop [2025] highlighted test-time adaptation and refinement loops as central themes: static-inference transduction solutions rarely exceed 11% accuracy, while test-time training and adaptation push scores far higher. Ensembling inductive and transductive strategies and aggressive data augmentation with voting are common practical tactics.

As of January 2026, frontier LLMs with extended thinking achieve 45–54% on ARC-AGI-2, while specialized solutions combining test-time training and TRM components (e.g., NVARC) demonstrated strong cost/accuracy trade-offs Knoop [2025].

2.4 Hierarchical and recursive reasoning models

2.4.1 HRM: Hierarchical Reasoning Model

Wang et al.'s HRM Wang et al. [2025] introduced a brain-inspired architecture with two coupled recurrent modules operating at different timescales. A high-level module reasons about abstract structure and strategy, while a low-level module executes pixel-level transformations. Hierarchical convergence—fast inner-loop convergence reset by slow high-level updates—yields effective computational depth while maintaining stability. HRM reported strong performance with modest parameter counts.

2.4.2 TRM: Tiny Recursive Model

Jolicoeur-Martineau's TRM Jolicoeur-Martineau [2025] simplified HRM into a compact 2-layer recursive model (7M parameters) that achieves competitive ARC performance by iteratively refining a predicted solution using latent reasoning states. TRM conditions on tasks via a puzzle_id embedding table, which we analyze as a source of memorization (Section 1).

2.4.3 TRM as a refinement loop

Refinement loops—iteratively transforming candidate programs or outputs using feedback—emerged as a dominant paradigm in ARC progress Knoop [2025]. TRM's recursive updates with deep supervision are a clear example of this pattern.

2.4.4 The memorization problem

Analyses revealed a critical limitation: when puzzle_id embeddings are removed or randomized, TRM/HRM accuracy collapses to near-zero Roye-Azar et al. [2025], Knoop [2025]. Empirical evidence shows substantial memorization of task-specific mappings rather than extraction of general transformation rules. Scaling the embedding table to very large synthetic datasets incurs massive memory costs (e.g., NVARC reports an infeasible 51B parameter requirement for naive scaling) Sorokin and Puget [2025].

2.5 Latent Program Networks

Latent Program Networks (LPN) Bonnet and Macfarlane [2024] learn a continuous latent space of implicit programs via an encoder–decoder architecture. The encoder maps demonstration pairs to a latent program vector; the decoder executes that latent program on test inputs. Crucially, LPN applies test-time gradient search in latent space (using leave-one-out loss on demos) to refine the latent representation, substantially improving out-of-distribution performance at the cost of expensive test-time optimization.

2.6 Positioning ETRM

TRM, LPN, and ETRM share a continuous-vector program representation but differ in how that representation is obtained and searched. TRM relies on learned lookup embeddings, LPN uses an encoder plus gradient-based test-time search, and ETRM computes encoder-derived representations decoded by a learned recursive decoder without test-time gradient updates (Section 3). Table 2 compares these approaches.

Table 2: Comparison of TRM, LPN, and ETRM along taxonomy axes.

Aspect	TRM	LPN	ETRM (Ours)
Program representation	Continuous (embedding)	Continuous (latent vector)	Continuous (latent vector)
How obtained	Learned lookup	Encoder	Encoder
Program search	Learned (recursive decoder)	Gradient-based	Learned (recursive decoder)
Processes demos at test time	No	Yes	Yes
Test-time optimization	No	Yes (gradient ascent)	No
Can generalize to unseen tasks	No	Yes	Yes (goal)

ETRM aims to combine LPN's ability to compute task representations from demonstrations with TRM's efficient learned search, replacing puzzle_id lookup with an encoder that enables true few-shot generalization while avoiding costly test-time optimization.

3 Method

3.1 Overview: From memorization to generalization

The Tiny Recursive Model (TRM) Jolicoeur-Martineau [2025] achieves strong performance on ARC-AGI through recursive reasoning with deep supervision. However, TRM relies on a learned embedding matrix that maps puzzle identifiers to task-specific representations. Critically, this embedding matrix includes entries for *both* training and evaluation puzzles—their embeddings receive gradient updates during training even though their test queries are held out. This design choice means TRM cannot solve puzzles without corresponding embeddings, fundamentally limiting it to interpolation rather than true generalization.

We propose **Encoder-based TRM (ETRM)**, which replaces the embedding lookup with a neural encoder that computes task representations from demonstration input–output pairs at inference time. This modification transforms TRM from a memorization-based system into one capable of few-shot learning:

$$\text{TRM:} \quad \mathbf{c} = \text{EmbeddingMatrix}[\text{puzzle\_id}] \tag{1}$$

$$\text{ETRM:} \quad \mathbf{c} = \text{Encoder}(\{(\mathbf{x}_i^{\text{in}}, \mathbf{x}_i^{\text{out}})\}_{i=1}^{K}) \tag{2}$$

where $\mathbf{c} \in \mathbb{R}^{T \times D}$ is the task context ($T = 16$ tokens, $D = 512$ dimensions), and K is the number of demonstration pairs. The encoder learns to extract transformation rules from demonstrations, enabling generalization to puzzles not seen during training. We preserve TRM's decoder architecture unchanged Jolicoeur-Martineau [2025]; only the task-context source is modified.

3.2 Background: TRM architecture

3.2.1 Task context via puzzle embedding lookup

TRM obtains task context through a learned embedding matrix indexed by puzzle identifier. Pseudocode:

```
# TRM: Task context from embedding lookup
def trm_get_context(puzzle_id):
    # puzzle_emb: learned matrix of shape (num_puzzles, T, D)
    # Includes entries for ALL puzzles (training AND evaluation)
    return puzzle_emb[puzzle_id]  # Returns (T, D) context
```

The embedding matrix must include an entry for every puzzle the model will encounter. Since evaluation puzzle IDs receive gradient updates during training, TRM cannot generalize to truly unseen puzzles—it effectively memorizes task-specific mappings.

3.2.2 Dual-state recursive reasoning

Given task context, TRM maintains two latent states that are iteratively refined:

- $\mathbf{y}$: current predicted solution (embedded)
- $\mathbf{z}$: latent reasoning state

Pseudocode for the forward pass:

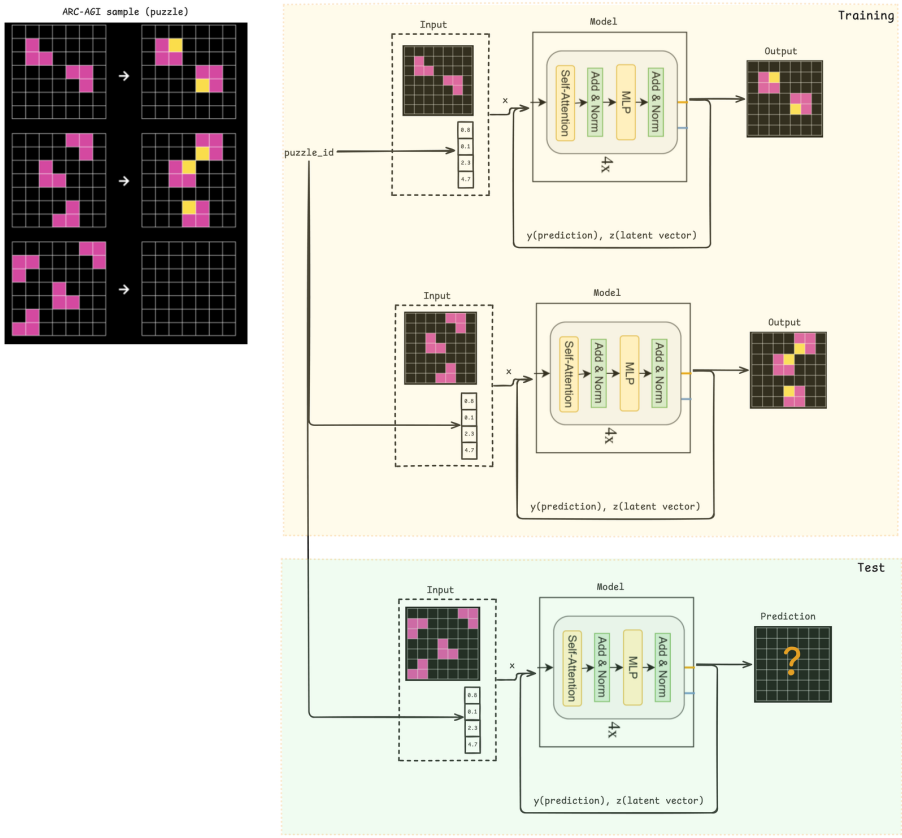

Figure 2: TRM architecture: task context is obtained via embedding matrix lookup indexed by puzzle ID.

```
def trm_forward(x, puzzle_id, y, z, H_cycles=3, L_cycles=6):
    # Get task context via embedding lookup (not from demos!)
    c = puzzle_emb[puzzle_id]          # <-- The limitation

    input_emb = embed(x) + c           # Add task context to input

    for h in range(H_cycles):
        for l in range(L_cycles):
            z = f_theta(z, y + input_emb)  # Refine reasoning
        y = f_theta(y, z)                   # Update solution

    return y, z
```

The network f_θ is a small 2-layer transformer Vaswani et al. [2017]. Recursive application creates an effective depth of $H \times (L+1) \times 2 = 42$ layers.

3.2.3 Deep supervision with adaptive computation

TRM uses deep supervision: the carry state $(\mathbf{y}, \mathbf{z})$ persists across training steps, with supervision applied at each recursion. A Q-head implements Adaptive Computation Time (ACT) Graves [2016] to learn halting; during training an exploration probability p_{explore} encourages continued computation sometimes.

3.3 Encoder architectures

We study three encoder paradigms, each reflecting a different hypothesis about effective task representation.

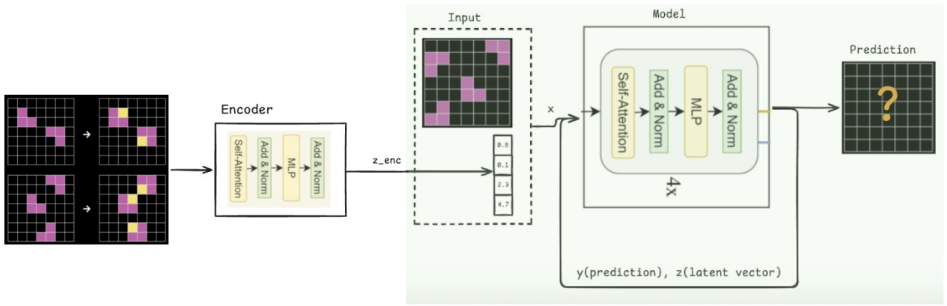

Figure 3: ETRM architecture: task context is computed from demonstration pairs via a neural encoder, replacing the embedding lookup.

3.3.1 Feedforward deterministic encoder

Hypothesis: a single forward pass through a feedforward encoder can extract sufficient information from demos to characterize the transformation rule.

The encoder has two stages:

Stage 1 – Per-demo grid encoding. Each demo pair $(\mathbf{x}_i^{\text{in}}, \mathbf{x}_i^{\text{out}})$ is encoded independently by concatenating input and output grids and processing with a transformer:

$$\mathbf{h}_i = \text{TransformerEnc}([\mathbf{x}_i^{\text{in}}; \mathbf{x}_i^{\text{out}}]) \in \mathbb{R}^{2S \times D}$$

with $S = 900$ (flattened 30×30 grid). Pool to a single vector:

$$\mathbf{e}_i = \text{MeanPool}(\mathbf{h}_i \odot \mathbf{m}_i) \in \mathbb{R}^D$$

where $\mathbf{m}_i$ masks padding tokens.

Stage 2 – Set aggregation via cross-attention. Aggregate the K demo encodings $\{\mathbf{e}_i\}_{i=1}^K$ using learnable query tokens Lee et al. [2019]:

$$\mathbf{c} = \text{CrossAttn}(\mathbf{Q}, \mathbf{E}, \mathbf{E}) \in \mathbb{R}^{T \times D}$$

where $\mathbf{Q} \in \mathbb{R}^{T \times D}$ are $T = 16$ learnable queries and $\mathbf{E} \in \mathbb{R}^{K \times D}$ stacks demo encodings.

Pseudocode:

```
def deterministic_encoder(demo_inputs, demo_outputs, demo_mask):
    # Stage 1: Encode each demo pair
    B, K, S = demo_inputs.shape
    encodings = []
    for i in range(K):
        h = transformer_enc(concat(demo_inputs[:,i], demo_outputs[:,i]))
        e = mean_pool(h, mask=demo_mask[:,i])
        encodings.append(e)
    E = stack(encodings, dim=1)  # (B, K, D)

    # Stage 2: Cross-attention aggregation
    Q = learnable_queries.expand(B, -1, -1)  # (B, T, D)
    context = cross_attention(Q, E, E, mask=demo_mask)
    return context  # (B, T, D)
```

3.3.2 Feedforward variational encoder

Hypothesis: A variational bottleneck promotes a structured latent space of transformation rules that may improve generalization.

We study two variants.

Cross-attention VAE. Apply variational bottleneck *after* aggregating demos via cross-attention. This allows the model to capture cross-demo patterns before compressing to a latent code.

```
def crossattn_vae_encoder(demo_inputs, demo_outputs, demo_mask):
    # Stage 1: Deep per-demo encoding (8 layers, CLS pooling)
    encodings = []
    for i in range(K):
        h = deep_transformer(concat(demo_inputs[:,i], demo_outputs[:,i]))
        e = cls_pooling(h)  # (B, D)
        encodings.append(e)
    E = stack(encodings, dim=1)  # (B, K, D)

    # Stage 2: Cross-attention aggregation
    Q = learnable_queries.expand(B, -1, -1)  # (B, T, D)
    context = cross_attention(Q, E, E, mask=demo_mask)  # (B, T, D)

    # Stage 3: Variational bottleneck
    z_pre = mean_pool(context)  # (B, D)
    z_pre = rms_norm(z_pre)     # Stabilize before projection
```

```
    mu, logvar = linear_proj(z_pre), linear_proj(z_pre)
    logvar = clamp(logvar, -10, 10)

    # Reparameterization (training only)
    z = mu + exp(0.5 * logvar) * randn_like(mu) if training else mu

    # Decode to output shape
    context = reshape(linear(z), (B, T, D))
    return context
```

Per-demo VAE. Inspired by LPN Bonnet and Macfarlane [2024], apply variational inference *per demo* before aggregation. Each demo gets its own latent code, which are then averaged.

```
def perdemo_vae_encoder(demo_inputs, demo_outputs, demo_mask):
    # Stage 1: Shallow per-demo encoding (2 layers, 128 hidden)
    encodings = []
    for i in range(K):
        h = shallow_transformer(concat(demo_inputs[:,i], demo_outputs[:,i]))
        e = cls_pooling(h)  # (B, D_internal=128)
        encodings.append(e)
    E = stack(encodings, dim=1)  # (B, K, 128)

    # Stage 2: Per-demo variational projection
    mu = linear_proj(E)      # (B, K, latent_dim=32)
    logvar = linear_proj(E)  # (B, K, 32)
    logvar = clamp(logvar, -10, 10)

    # Reparameterize each demo independently
    z_demos = mu + exp(0.5 * logvar) * randn_like(mu) if training else mu

    # Stage 3: Mean aggregation (after sampling)
    z_demos = z_demos * demo_mask.unsqueeze(-1)  # Mask invalid
    z_pooled = sum(z_demos, dim=1) / num_valid_demos  # (B, 32)

    # Project to output shape
    context = reshape(linear(z_pooled), (B, T, D))
    return context
```

Cross-attention VAE captures cross-demo patterns before bottlenecking; per-demo VAE is more compact (∼1M params vs. ∼16M) but may lose demo interactions. We adopt LPN's encoder design for comparison but *do not* use test-time gradient search; our encoders produce fixed representations.

3.3.3 Iterative encoder (joint encoder–decoder refinement)

Hypothesis: Recursive refinement that benefits the decoder might also benefit the encoder. The iterative encoder mirrors TRM's decoder structure.

Encoder maintains dual latent states:

- $\mathbf{z}_e^H$: high-level context (serves as decoder input)
- $\mathbf{z}_e^L$: low-level reasoning state

Hierarchical refinement (ACT-style) pseudocode:

```
def iterative_encoder_step(z_e_H, z_e_L, demo_input, H_cycles, L_cycles):
    # demo_input: aggregated demo representation

    for h in range(H_cycles):
        for l in range(L_cycles):
            z_e_L = L_level(z_e_L, z_e_H + demo_input)
        z_e_H = L_level(z_e_H, z_e_L)

    return z_e_H, z_e_L  # z_e_H is context for decoder
```

Unlike feedforward encoders, the iterative encoder's carry state persists across ACT steps. The encoder refines its understanding as the decoder refines its prediction—a form of joint reasoning.

3.4 Integration with TRM decoder

For all encoder types, integration follows:

1. **Context injection:** encoder output $\mathbf{c} \in \mathbb{R}^{T \times D}$ replaces puzzle embedding.
2. **Position concatenation:** context is prepended to input token embeddings.
3. **Forward pass:** combined representation flows through TRM's recursive reasoning.

Decoder architecture remains as in TRM Jolicoeur-Martineau [2025]: dual-state design $(\mathbf{y}, \mathbf{z})$, H/L cycles, ACT halting, and deep supervision.

3.5 Training protocol

3.5.1 Strict train/evaluation separation

We enforce full data separation:

Split	Puzzles	Purpose
Training	∼560 groups (ARC-AGI training + concept)	Model training
Evaluation	∼400 groups (ARC-AGI evaluation)	Testing generalization

Evaluation puzzles' demonstrations are never seen during training; the encoder must extract rules at test time.

3.5.2 Data augmentation

Following TRM, we apply ≈1000 augmented versions per puzzle:

- Color permutation: random shuffle of colors 1–9 (black=0 fixed)
- Dihedral transforms: 8 geometric transforms (4 rotations × 2 reflections)
- Translation: random positioning within 30×30 grid (training only)

The same augmentation is applied to all components of a puzzle (demos and tests) to preserve the transformation structure.

3.5.3 Pretrained decoder initialization

We initialize the decoder from TRM's pretrained weights, providing:

1. Faster convergence: decoder already knows recursive refinement for ARC tasks.
2. Encoder focus: encoder can concentrate on learning representations.

We explore two modes:

- Frozen decoder: only encoder updated (faster, ensures encoder learns).
- Joint fine-tuning: both encoder and decoder updated (potentially better final performance).

3.5.4 Variational training

For VAE encoders, the training loss is:

$$\mathcal{L} = \mathcal{L}_{\text{CE}} + \beta \cdot \mathcal{L}_{\text{KL}}$$

Practical considerations:

- KL weight: we use a fixed $\beta = 0.0001$.
- Numerical stability: clamp $\log \sigma^2 \in [-10, 10]$.
- Evaluation: use mean $\boldsymbol{\mu}$ without sampling for deterministic evaluation.

3.6 Design space summary

Table 3: Encoder architecture design space

Encoder type	Aggregation	Variational	Parameters	Key property
Feedforward deterministic	Cross-attention	No	∼15M	Simple, stable baseline
Cross-attention VAE	Cross-attention	Post-aggregation	∼16M	Regularized latent space
Per-demo VAE	Mean	Per-demo	∼1M	LPN-inspired, compact
Iterative (TRM-style)	Mean	No	∼8M	Joint refinement with decoder

Each architecture embodies hypotheses: cross-attention vs. mean pooling, variational vs. deterministic, iterative vs. feedforward. We evaluate these empirically in Section **??**.

4 Experiments

We evaluate ETRM's ability to generalize to puzzles whose demonstrations were never seen during training. Our experiments reveal a striking negative result: all encoder architectures get 0% accuracy on held-out puzzles despite reasonable training performance, a phenomenon we attribute to encoder collapse.

4.1 Experimental setup

4.1.1 Dataset

We train and evaluate on ARC-AGI-1, which contains 400 training puzzles and 400 evaluation puzzles Chollet [2019]. Following TRM Jolicoeur-Martineau [2025], we augment the training set with approximately 160 additional "concept" puzzles that target specific transformation primitives.

Data augmentation. Each puzzle is augmented roughly 1000× using:

- Color permutation: random shuffle of colors 1–9 (black remains fixed).
- Dihedral transforms: 8 geometric transformations (rotations and reflections).
- Translation: random positioning within the 30×30 grid (training only).

Augmentation is applied consistently to all components of a puzzle (demonstrations and test queries), preserving the transformation rule's structure.

True few-shot evaluation. A key distinction between ETRM and TRM evaluation is data separation. In TRM, evaluation puzzle identifiers exist in the embedding matrix and receive gradient updates during training—the model has effectively "seen" those puzzles ARC Prize Foundation [2025]. In ETRM we enforce strict separation: evaluation puzzle demonstrations are *never* seen during training. The encoder must extract transformation rules from demonstrations encountered for the first time at test time; this is true few-shot evaluation.

Table 4: Dataset split summary

Split	Puzzle Groups	Augmentation Factor	Total Samples
Training	∼560 (training + concept)	∼1000×	∼560,000
Evaluation	∼400 (evaluation)	∼1000×	∼400,000

4.1.2 Evaluation protocol

Metrics. We report Pass@k accuracy, where a puzzle is considered solved if the correct answer appears among the top-k most common predictions. We report Pass@1 (primary), Pass@2 and Pass@5.

Voting mechanism. Following TRM Jolicoeur-Martineau [2025], we aggregate predictions across all augmented versions of each puzzle (∼1000 per puzzle). Each augmented version produces a prediction which is inverse-transformed back to the original coordinate space; final prediction is determined by majority voting.

Subset evaluation. Due to computational constraints, we evaluate on a subset of 32 puzzle groups (about 8% of the full evaluation set). Evaluating the full set requires ∼24 hours per model since voting aggregates ∼1000 augmentations per puzzle. Although subset evaluation reduces statistical power, the contrast between 0% and ∼37%+ Pass@1 is large and informative.

4.1.3 Training configuration

Decoder initialization. The decoder is initialized from pretrained TRM weights, providing a capable recursive reasoning module. The decoder is *not* frozen—gradients flow through all decoder parameters during training—so failures to generalize cannot be attributed to a frozen decoder.

Hyperparameters. Batch size is 256 for deterministic and iterative encoders, reduced to 128 for the Cross-Attention VAE due to memory constraints. ACT maximum steps is 16 with exploration probability 0.5. We re-encode the full batch at every ACT step to ensure adequate gradient flow to the encoder.

Table 5: Training hyperparameters

Parameter	Deterministic	Variational	Iterative
Batch size	256	128	256
Learning rate	1e-4	1e-4	1e-4
ACT max steps	16	16	16
Grad clip norm	1.0	1.0	1.0

Computational resources. Experiments ran on a server with 4 NVIDIA A100 80GB GPUs using PyTorch distributed data-parallel (torchrun). Each ETRM variant required ∼12–24 hours to reach 175k training steps; the TRM baseline converged in ∼48 hours (518k steps).

4.2 TRM baseline

We reproduce TRM training to establish a baseline. Results at two checkpoints are shown in Table 6.

Table 6: TRM baseline results

Model	Params	Pass@1	Pass@2	Pass@5	Train Acc
TRM (155k steps)	7M	37.38%	41.25%	47.12%	92.50%
TRM (converged)	7M	41.75%	48.75%	52.25%	98.44%

4.3 ETRM results

We evaluate three encoder architectures (Section 3.3): deterministic feedforward, Cross-Attention VAE, and iterative TRM-style encoder. Results appear in Table 7.

Table 7: ETRM results

Model	Encoder Type	Params	Pass@1	Pass@2	Train Acc
ETRM-Deterministic	Feedforward Deterministic	22M	0.00%	0.50%	78.91%
ETRM-Variational	Cross-Attn VAE	23M	0.00%	0.00%	40.62%
ETRM-Iterative	Iterative TRM-style	15M	0.00%	0.25%	51.17%

All three encoder architectures get 0% Pass@1 on held-out puzzles despite substantial training accuracy, indicating a complete generalization failure.

4.4 Analysis

4.4.1 Training dynamics

Figure 4: Training accuracy over time for TRM and ETRM variants.

TRM reaches ∼98% training accuracy; ETRM variants plateau at substantially lower accuracies (Feedforward: 79%, Iterative: 51%, VAE: 41%).

4.4.2 Encoder collapse

We measure cross-sample variance of encoder outputs to quantify how different outputs are across puzzles. Low variance indicates collapse to near-constant representations.

Table 8: Encoder collapse analysis

Model	Cross-Sample Variance	Interpretation
ETRM-Deterministic	0.36	Low — collapsed
ETRM-Variational	3.33	Higher (KL prevents full collapse)
ETRM-Iterative	0.15	Very low — severely collapsed

With collapsed encoder outputs the decoder receives essentially the same task representation for every puzzle, preventing puzzle-specific behavior and explaining the 0% test accuracy despite nontrivial training accuracy.

4.4.3 Qualitative examples

ETRM produces structured but incorrect transformations on held-out puzzles; TRM (with embeddings) produces correct predictions, indicating the decoder is capable given appropriate task context.

4.4.4 Key findings

1. All ETRM variants obtain 0% Pass@1 on held-out puzzles despite 41–79% training accuracy.

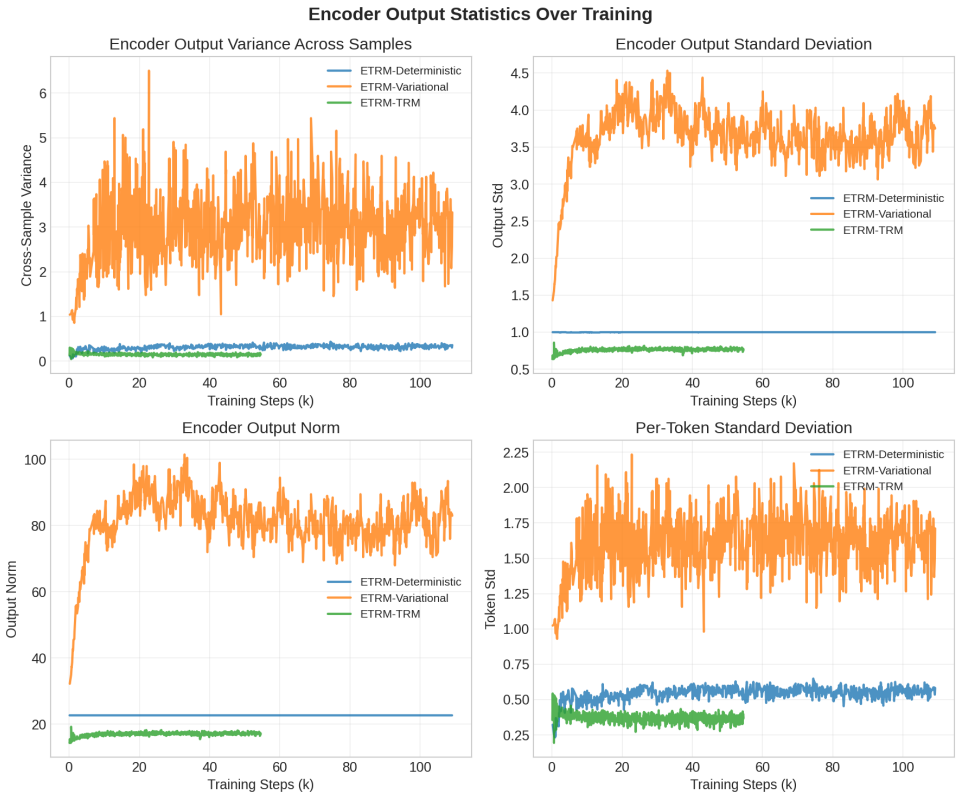

Figure 5: Encoder output statistics: cross-sample variance, within-sample variance, and distributions.

2. Encoder collapse: all encoders produce near-constant outputs across different puzzles (cross-sample variance 0.15–3.33).

3. Architecture-agnostic: feedforward, variational, and iterative encoders all exhibit collapse.

4. TRM achieves ∼37% Pass@1 at comparable training time; the decoder can solve puzzles when given appropriate task context.

5 Discussion

5.1 Analysis of Results

5.1.1 The Generalization Gap

The training–test disconnect observed in Section 4.4 admits two hypotheses:

(A) Task Difficulty. Extracting transformation rules from demonstrations in a single forward pass—without any feedback signal—is fundamentally harder than refining puzzle-specific embeddings over many gradient updates. The encoder must perform meta-learning: learning to learn from examples [Jolicoeur-Martineau, 2025].

(B) Implementation Issues. Our encoder designs, training procedures, or hyperparameters may have flaws.

Evidence favors hypothesis (A). Three fundamentally different architectures all failed identically. The ETRM-Iterative experiment is particularly informative: iteration analogous to

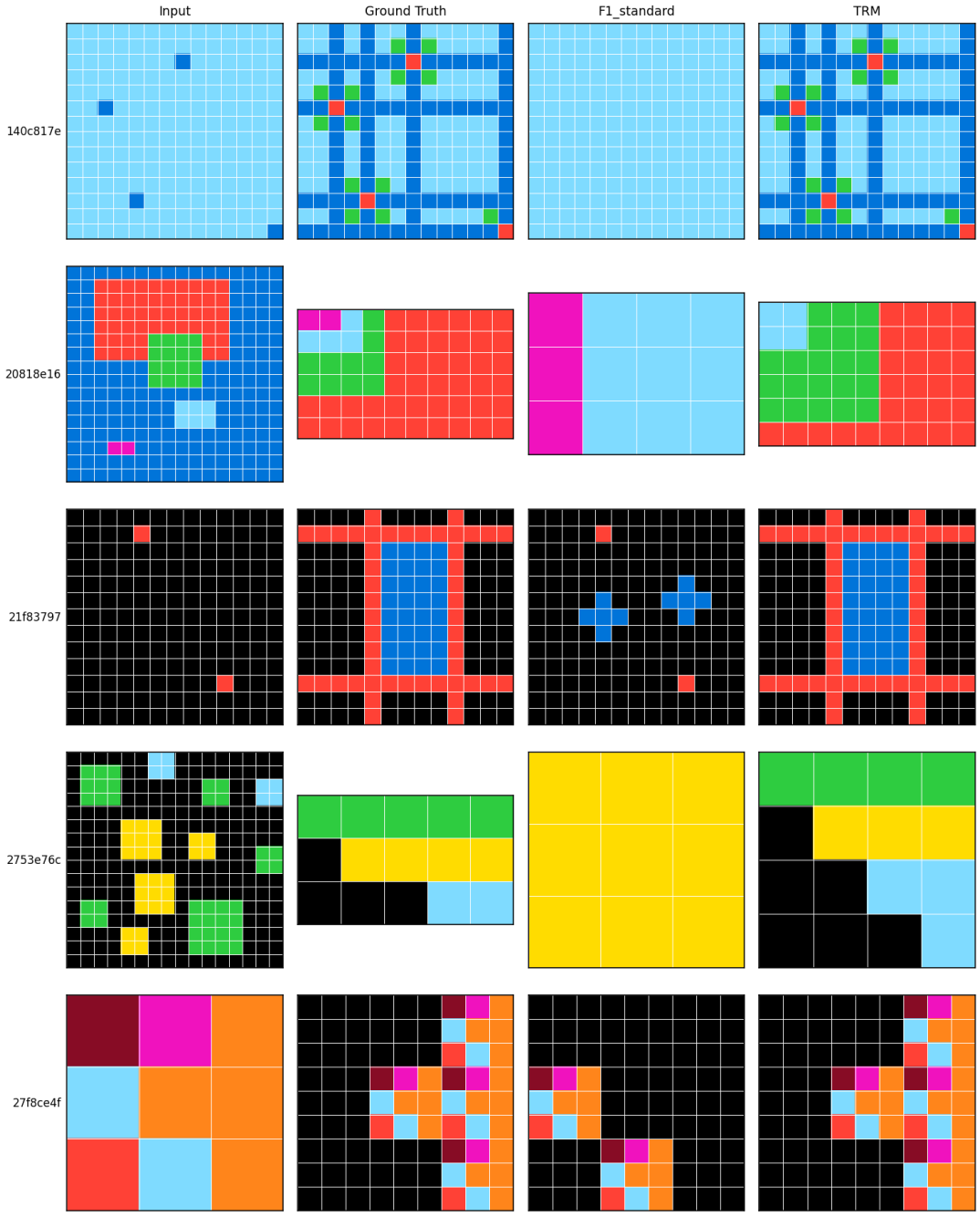

Figure 6: Predictions on held-out puzzles: Input, Ground Truth, ETRM-Deterministic prediction, and TRM prediction.

TRM's recursive decoder still yielded 0% test accuracy, suggesting the problem is not insufficient computation but rather the absence of a guiding signal during refinement.

We cannot fully rule out (B), but the consistent failure pattern points toward task difficulty.

5.1.2 The Asymmetry Between TRM and ETRM

Understanding why TRM succeeds while ETRM fails requires examining how each learns transformation rules.

How TRM learns. TRM [Jolicoeur-Martineau, 2025] refines each puzzle embedding over hundreds of thousands of training steps. Each of the ∼876,000 puzzle-augmentation combinations receives approximately 500+ gradient updates during training. The embedding gradually captures task-specific patterns through this extended optimization—a significant advantage where the model has many opportunities to learn each transformation.

What we ask the encoder to do. ETRM's encoder must extract the transformation rule from just 2–5 demonstration pairs in a single forward pass, producing a representation that works for transformations never seen during training. This is meta-learning: the encoder must learn *how to learn* from examples.

The difficulty gap becomes clearer when we compare refinement mechanisms:

Table 9: Refinement mechanisms comparison

Approach	Refinement	Feedback Signal
TRM [Jolicoeur-Martineau, 2025]	Gradient descent on embedding	Ground-truth labels (supervised)
LPN [Bonnet and Macfarlane, 2024]	Gradient ascent in latent space	Demo consistency (self-supervised)
ETRM (feedforward)	Single forward pass	None
ETRM-Iterative	Multiple encoder iterations	None

Our ETRM-Iterative experiment tested whether iteration alone could help—it could not. The issue is not computation but feedback. TRM's refinement is guided by gradients from ground-truth labels. LPN's test-time search is guided by leave-one-out demo consistency. ETRM's encoder iterates without any signal indicating whether its representation is improving.

Takeaway: Effective latent-space refinement requires a feedback signal—either supervised (labels) or self-supervised (demo consistency). Unguided iteration is insufficient.

5.1.3 Implications of Encoder Collapse

The encoder collapse documented in Section 4.4.2 explains the training–test disconnect. With near-constant encoder outputs, the decoder receives essentially the same "task representation" for every puzzle, so it learns to ignore the uninformative encoder signal and instead memorizes input–output mappings for training examples directly [ARC Prize Foundation, 2025]. High training accuracy through memorization; zero generalization to new transformations.

5.1.4 Did Variational Encoding Help?

We hypothesized that variational encoders might encourage more diverse representations through KL regularization. The Cross-Attention VAE achieved an order-of-magnitude higher cross-sample variance (3.33 vs. 0.36) compared to the deterministic encoder—yet still 0% test accuracy.

Higher variance does not equal useful variance. The variational encoder produces diverse representations, but these representations are not discriminative for transformation rules. The KL penalty toward a standard normal prior may push representations away from task-relevant structure. Additionally, VAE regularization prevented decoder memorization (lower 40.62% training accuracy) without producing transformation-relevant features.

Takeaway: Variance alone is insufficient; representations must capture transformation-relevant information. Diversity without discriminability does not enable generalization.

5.1.5 Qualitative Failure Modes

Examination of ETRM predictions on held-out puzzles (Figure 6, Section 4.4.3) reveals several failure modes:

1. **Collapsed outputs.** Some predictions are nearly uniform (solid color), directly reflecting encoder collapse propagating to the decoder.
2. **Structured but wrong.** Some predictions show grid structure and color patterns, but the wrong transformation is applied—e.g., color filling where rotation was required.
3. **Partial correctness.** Occasionally, predictions capture some aspect of the transformation (correct colors, wrong arrangement), suggesting the decoder has learned general grid manipulation skills.

These patterns are consistent with a model that has learned *something* about ARC transformations from training but cannot select the correct transformation without a useful encoder signal. In contrast, TRM predictions (with puzzle embeddings) are often correct or close, demonstrating the decoder is capable when given appropriate task context.

5.1.6 Context for Comparison with TRM

Direct comparison between TRM and ETRM requires acknowledging a fundamental difference in what each model is asked to do:

Table 10: Context for TRM/ETRM comparison

Model	Pass@1	Train Acc	Task
TRM [Jolicoeur-Martineau, 2025] (155k steps)	37.38%	92.50%	Generalize to augmented versions of l
TRM [Jolicoeur-Martineau, 2025] (518k steps)	41.75%	98.44%	Same
ETRM-Deterministic (175k steps)	0.00%	78.91%	Generalize to entirely unseen transfor

TRM test puzzles have embeddings that receive gradient updates during training—it generalizes to different augmentations (color permutations, rotations) of puzzles it has "seen." ETRM must generalize to transformations it has never encountered, a fundamentally harder task.

Our encoder approach does not achieve few-shot generalization. However, the comparison is not entirely fair. A more appropriate comparison would be with LPN [Bonnet and Macfarlane, 2024], which achieves 15.5% on held-out puzzles—but only with test-time gradient optimization that we deliberately avoided.

5.2 Challenges Encountered

Several practical challenges emerged during development that required careful debugging.

5.2.1 Gradient Starvation

Our initial implementation cached encoder outputs across ACT steps for efficiency. This turned out to be a significant mistake. With TRM's dynamic halting mechanism, only samples that reset their carry state receive fresh encoder outputs—roughly 2% of the batch per step. The encoder therefore received gradients from only 2% of training data, severely stunting its learning. Training accuracy plateaued around 35–50% with the encoder producing essentially random outputs.

The fix was straightforward once diagnosed: re-encode the full batch at every ACT step rather than caching. This ensures 100% gradient coverage at the cost of additional computation. The lesson here is that gradient flow analysis is critical when modifying architectures with dynamic computation patterns—efficiency optimizations that seem harmless can silently break learning.

5.2.2 Training Stability

Early experiments exhibited sudden training collapse around step 1900. The encoder output distribution was shifting faster than the decoder could adapt, creating a feedback loop that destabilized both components. Gradient clipping (`grad_clip_norm`=1.0) resolved this issue and stabilized training throughout all subsequent experiments.

5.2.3 Monitoring Representation Quality

Standard training metrics (loss, accuracy) failed to reveal encoder collapse until evaluation. We introduced cross-sample variance as a diagnostic metric, tracking how differently the encoder responds to different puzzles within each batch. Low variance proved to be an early warning sign of representation collapse, often visible long before evaluation revealed the problem. This metric proved essential for debugging and would be valuable for future encoder-based approaches.

5.3 Limitations

5.3.1 Computational Constraints

Our experiments operated under significant resource constraints:

- **Training duration:** 25k–175k steps vs. TRM's 518k steps (∼4 days per run on 4 GPUs).
- **Evaluation scope:** 32 puzzle groups (8%) instead of full 400 (∼1 day on 4 GPUs for full evaluation).
- **Single seed:** No variance estimates across runs.

These results provide directional signal—0% vs. 37% accuracy is unambiguous—but absolute performance numbers might improve with additional training.

5.3.2 Architecture Exploration

We tested three encoder paradigms but did not explore alternatives that might succeed:

- Slot attention for object-centric encoding.
- Graph neural networks for relational reasoning.
- Contrastive objectives for discriminative representations.

A different architecture might succeed where ours failed.

5.3.3 Implementation Caveats

We cannot fully rule out implementation issues:

- Encoder architectures may be suboptimal.
- Training dynamics may have undetected problems.
- Hyperparameters may be poorly tuned.

One known issue: EMA weights from the pretrained TRM checkpoint were not properly loaded for the decoder. This is unlikely to explain the generalization failure because the decoder was trainable and reached 79% training accuracy—the problem is generalization, not learning capacity.

The consistent failure across three architectures suggests task difficulty rather than implementation bugs, but more exploration is warranted.

5.4 Future Directions

Our results suggest that computing task representations in a single forward pass is insufficient for extracting transformation rules. Drawing on the program-synthesis taxonomy from Section 2, we identify promising directions.

5.4.1 Self-Supervised Test-Time Search

Our ETRM-Iterative experiment showed that iteration alone is insufficient—the encoder refines its representation but has no signal indicating whether it is improving. Both TRM [Jolicoeur-Martineau, 2025] and LPN [Bonnet and Macfarlane, 2024] succeed because they have feedback signals guiding refinement: TRM uses label gradients during training, LPN uses demo consistency at test time.

LPN demonstrates that self-supervised gradient search in latent space can significantly improve generalization (7.75% to 15.5%). Combining ETRM with test-time optimization could provide the missing ingredient:

- **Leave-one-out loss:** Use held-out demo pairs as self-supervision for latent-space search—no labels required, only the demos themselves.
- **Hybrid search:** Encoder provides a warm start, gradient-based refinement provides the feedback loop.
- **Efficiency:** Starting from a learned encoder estimate may require fewer gradient steps than LPN's random initialization.

The common thread in successful approaches is *guided refinement*. Our failed ETRM-Iterative suggests unguided iteration is not enough—but guided iteration at test time may bridge the gap.

5.4.2 Contrastive Learning for Encoder

An alternative to test-time optimization is training a more discriminative encoder:

- Train encoder to produce similar representations for demos from the same puzzle, and different representations for demos from different puzzles.
- This could encourage discriminative representations without requiring test-time optimization.

5.4.3 Improved Decoder Initialization

Our decoder initialization did not include EMA weights from the pretrained TRM checkpoint. While unlikely to explain the generalization failure—the decoder reached 79% training accuracy—future experiments should use properly EMA-initialized weights to eliminate this confound.

5.5 Conclusions

Our experiments reveal that replacing TRM puzzle embeddings with a demonstration encoder is substantially harder than expected. Three encoder architectures—feedforward deterministic, variational, and iterative—all achieve reasonable training accuracy but complete failure on held-out puzzles. Analysis shows this stems from encoder collapse: the encoders produce near-constant outputs regardless of input demonstrations, forcing the decoder to memorize training patterns rather than learn generalizable rule extraction.

The key insight is the asymmetry between TRM and ETRM. TRM refines each puzzle embedding over hundreds of gradient updates during training. ETRM asks the encoder to extract equivalent information in a single forward pass with no task-specific feedback. Our ETRM-Iterative experiment shows that iteration alone does not solve this problem—the missing ingredient is a feedback signal guiding refinement.

The most promising path forward is combining ETRM's encoder with test-time optimization using self-supervised signals, as demonstrated by LPN [Bonnet and Macfarlane, 2024]. This would provide the guided refinement that successful approaches share while preserving the ability to generalize to truly novel puzzles.

References

ARC Prize Foundation. The hidden drivers of HRM's performance on ARC-AGI. ARC Prize Foundation Blog, 2025. URL https://arcprize.org/blog/hrm-analysis.

Clément Bonnet and Matthew V. Macfarlane. Searching latent program spaces. *arXiv preprint arXiv:2411.08706*, 2024. URL https://arxiv.org/abs/2411.08706. 3rd Place Paper Award, ARC Prize 2024.

Swarat Chaudhuri, Kevin Ellis, Oleksandr Polozov, Rishabh Singh, Armando Solar-Lezama, and Yisong Yue. Neurosymbolic programming. *Foundations and Trends in Programming Languages*, 7(1-2):1–151, 2021. URL https://www.nowpublishers.com/article/Details/PGL-031.

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

François Chollet, Mike Knoop, Gregory Kamradt, and Bryan Landers. ARC prize 2024: Technical report. *arXiv preprint arXiv:2412.04604*, 2024. URL https://arxiv.org/abs/2412.04604.

Alex Graves. Adaptive computation time for recurrent neural networks. *arXiv preprint arXiv:1603.08983*, 2016. URL https://arxiv.org/abs/1603.08983.

Lewis Hemens. How to beat ARC-AGI-2. lewish.io, 2025. URL https://lewish.io/posts/how-to-beat-arc-agi-2.

Alexia Jolicoeur-Martineau. Less is more: Recursive reasoning with tiny networks. *arXiv preprint arXiv:2510.04871*, 2025. URL https://arxiv.org/abs/2510.04871. 1st Place Paper Award, ARC Prize 2025.

Mike Knoop. ARC prize 2025: Results and analysis. ARC Prize Foundation Blog, 2025. URL https://arcprize.org/blog/arc-prize-2025-results-analysis.

Juho Lee, Yoonho Lee, Jungtaek Kim, Adam Kosiorek, Seungjin Choi, and Yee Whye Teh. Set transformer: A framework for attention-based permutation-invariant neural networks. In *International Conference on Machine Learning (ICML 2019)*, 2019. URL https://arxiv.org/abs/1810.00825.

Yuxuan Li et al. Combining induction and transduction for abstract reasoning. *arXiv preprint arXiv:2411.02272*, 2024. URL https://arxiv.org/abs/2411.02272. 1st Place Paper Award, ARC Prize 2024.

Antonio Roye-Azar, Santiago Vargas-Naranjo, Dhruv Ghai, Nithin Balamurugan, and Rayan Amir. Tiny recursive models on ARC-AGI-1: Inductive biases, identity conditioning, and test-time compute. *arXiv preprint arXiv:2512.11847*, 2025. URL https://arxiv.org/abs/2512.11847.

Dmitry Sorokin and Jean-François Puget. NVARC: ARC prize 2025 1st place solution. NVIDIA, 2025. URL https://www.kaggle.com/competitions/arc-prize-2025/discussion.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems 30 (NeurIPS 2017)*, 2017. URL https://arxiv.org/abs/1706.03762.

Guan Wang, Jin Li, Yuhao Sun, Xing Chen, Changling Liu, Yue Wu, Meng Lu, Sen Song, and Yasin Abbasi Yadkori. Hierarchical reasoning model. *arXiv preprint arXiv:2506.21734*, 2025. URL https://arxiv.org/abs/2506.21734.