Slots, Transitions, Loops: Learning Composable World Models for ARC

Anonymous ACL submission

Abstract

ARC tests in-context rule induction: given a few input-output demonstrations, a model must infer the hidden rule and apply it to a new query. While many approaches express ARC rules through language, code, or symbolic programs, ARC itself is visual-symbolic: rules appear as grid transitions over objects, colors, shapes, and spatial relations. We introduce Loop-OWM, an object-centric world-modeling architecture that learns these rules as composable transitions over structured states. It combines color-prototype slots, demonstration-conditioned task summaries, and a looped transition model with dense propagation and slot-conditioned correction. On both ARC-1 and ARC-2, Loop-OWM outperforms non-looped and looped baselines with comparable or fewer parameters. These results suggest that ARC rules can be learned not only as language descriptions or searched programs, but also as transitions over visual-symbolic world states.

1 Introduction

The Abstraction and Reasoning Corpus (ARC) was introduced as a benchmark for studying abstract reasoning under limited supervision: each task provides only a few input-output demonstrations, and the model must infer the hidden rule and apply it to a new query input. This setting has naturally attracted language-centric and program-centric approaches, where the demonstrations are interpreted as prompts for rule induction, search, or program synthesis. However, the task interface of ARC is not linguistic. The rules are expressed through transitions of discrete 2D grids: objects are moved, copied, recolored, completed, counted, or related through spatial configurations. ARC therefore sits at an interesting boundary between language-like reasoning and visual-symbolic state transition.

Recent visual approaches show that this visual-symbolic interface can be modeled directly by representing ARC examples as grid images or visual canvases. However, they still largely treat ARC as direct grid-to-grid prediction: the model is asked to produce the target grid, while the intermediate structure of the rule – what objects are present, how they change, and how the observed transition should apply to a new state – remains implicit. This is limiting because many ARC tasks are naturally object-indexed. We therefore ask: can ARC rules be learned as composable object-aware transitions over structured grid states, and rolled out through iterative updates?

In this paper, we study ARC from the perspective of *object-centric world modeling*. In this view, each grid is a visual-symbolic world state, each demonstration pair provides evidence of a state transition, and each task specifies a state transition rule that should generalize from given pairs to news query examples. Solving an ARC task then requires more than predicting an output grid: the model must interpret the current state, identify reusable object-level structure, infer how the state changes from demonstrations, and execute the inferred transition on a new input. This formulation connects ARC to in-context rule induction: the demonstration pairs act as a prompt, but the induced rule is grounded in structured visual states rather than expressed purely in natural language or code.

We propose **Loop-OWM**, a composable architecture built around this formulation. The model consists of three conceptual components. First, an *object-centric grid interpreter* converts ARC grids into structured visual-symbolic states, exposing candidate entities through slots. Second, a *task encoder* reads the demonstration inputs, outputs, and transition-related features to infer a task-specific rule representation. Third, a *looped transition model* applies the inferred rule to the query state through iterative updates. The loop is not used merely as additional computation depth; rather, it implements a rollout mechanism in which the model repeatedly updates an internal state under a

task-conditioned transition rule.

This design allows us to separate high-level modeling principles from lower-level architectural choices. Loop-OWM is not a single monolithic predictor, but a structured design space for studying how neural models can learn visual-symbolic transition rules. In our final instantiation, each recurrent update is implemented through two complementary branches: one branch models *dense grid-level* state propagation, while the other injects *object-aware corrections* through slot-conditioned representations. We evaluate design choices through controlled comparisons with non-looped and looped ViT baselines, together with ablation studies on object-aware updates, task-summary tokens, transition supervision, and recurrent rollout. Beyond improving ARC prediction, we hope this study contributes to a broader view of abstract reasoning: ARC rules need not be represented only as language descriptions or searched programs, but can also be learned as composable transitions over structured world states.

2 Related Work

2.1 ARC as Rule Induction

ARC was introduced to measure skill-acquisition efficiency and abstract reasoning from limited experience (Chollet, 2019). Each task provides only a few input-output demonstrations, requiring the solver to infer a hidden transformation rule and apply it to a new query. This has motivated symbolic, program-centric, and more recently LLM-based approaches that describe, search, or synthesize candidate programs for ARC tasks (Wind, 2020; Ouellette, 2024; Singhal and Shroff, 2025). Such methods exploit the discrete and compositional structure of ARC, but often rely on explicit program spaces, hand-designed primitives, execution-guided search, or language-mediated reasoning. Recent analyses also suggest that LLMs can identify high-level concepts but still struggle with compositional execution of spatial transformations (Lee et al., 2024). We study a complementary direction: learning ARC rules as latent transitions over visual-symbolic grid states, rather than expressing them explicitly as language or code.

2.2 Neural Visual Models for ARC

Recent work shows that ARC's visual interface can be modeled directly. Vision-centric ARC methods represent tasks as spatial canvases and train neural architectures for grid-to-grid translation (Hu et al., 2025). Looped visual transformers further introduce recurrent computation, repeatedly applying shared blocks to increase effective reasoning depth without increasing parameter count (Shu et al., 2026; Saunshi et al., 2025; Geiping et al., 2025; Schwarzschild et al., 2021). These approaches demonstrate that visual modeling and iterative computation are useful inductive biases for ARC. However, they primarily treat the task as output-grid prediction. In contrast, Loop-OWM formulates ARC as demonstration-conditioned state transition learning: the model must infer what objects are present, how they change, and how the observed transformation should be applied compositionally to a new state.

2.3 Object-Centric and World Models

Many ARC rules are naturally object-indexed: transformations often operate on shapes, colors, connected components, and spatial relations rather than independent grid cells (Chollet, 2019; Xu et al., 2024). This makes object-centric representation learning relevant to ARC. Slot-based methods provide a mechanism for exposing an entity-like structure from dense visual features (Burgess et al., 2019; Locatello et al., 2020; Kirilenko et al., 2024), while object-centric video and world models show that such representations can support dynamics and relational reasoning (Zadaianchuk et al., 2023; Wu et al., 2023; Nam et al., 2026). Our setting differs from video or control domains: ARC provides no natural temporal trajectories, so the transition rule must be inferred from a few input-output demonstrations. Loop-OWM therefore treats the demonstration pairs themselves as evidence for a task-specific dynamics model.

3 Problem Formulation

An ARC task consists of a small set of input-output demonstrations

$$\mathcal{D} = \{(x^{(i)}, y^{(i)})\}_{i=1}^{N}, \tag{1}$$

and a query input $x \in \{0, \ldots, C-1\}^{H \times W}$ where each grid is a discrete visual-symbolic state, C counts the valid ARC colors. The goal is to predict the corresponding output grid y by inferring the task-specific rule expressed by the demonstrations. A direct prediction model treats this problem as

$$\hat{y} = f_\theta(x, \mathcal{D}), \tag{2}$$

where the demonstrations act as conditioning information for grid-to-grid prediction.

In contrast, we formulate ARC as demonstration-

conditioned state transition learning. Each demonstration pair $(x^{(i)}, y^{(i)})$ provides evidence of how a state changes under the hidden task rule. The model must infer a latent transition rule $z_{\mathcal{D}}$ from the demonstrations and apply it to the query state:

$$z_{\mathcal{D}} = \psi_\theta^{\text{task}}(\mathcal{D}), \quad \hat{y} = \text{Rollout}(x, z_{\mathcal{D}}). \tag{3}$$

This separates rule induction from rule execution: the task encoder $\psi_\theta^{\text{task}}(\cdot)$ infers what transition should be applied, while the rollout model applies the inferred rule to a new state. We use the term ***world model*** in this ***restricted sense***: a model of structured ARC states and the task-conditioned transitions between them, rather than a model of physical dynamics or embodied control.

Object-aware state abstraction. We introduce an object-aware state abstraction. Given a grid x, we first map each cell color to an embedding using a shared color embed table $E \in \mathbb{R}^{C \times d}$. The resulting feature map is patchified with a $\mathrm{p} \times \mathrm{p}$ convolution (Dosovitskiy et al., 2021) and augmented with $2D$ positional embeddings, yielding dense grid-state tokens

$$H = \phi^{\text{grid}}(x) \in \mathbb{R}^{L \times d}, \quad L = (H/\mathrm{p})(W/\mathrm{p}). \tag{4}$$

These dense tokens preserve fine-grained spatial information and serve as the grid-level state representation used by the transition model.

From the dense grid-state tokens, we extract a set of K object slots

$$S = \{s_k\}_{k=1}^{K}, \quad s_k \in \mathbb{R}^d, \tag{5}$$

using a *Slot Attention* (Locatello et al., 2020) module $\mathrm{S}_\theta(\cdot)$: the slots attend competitively to the dense tokens and are iteratively updated through attention, recurrent refinement, and a residual MLP.

The slots are not required to be explicit symbolic objects or perfect segmentation masks. Instead, they serve as a *structured interface* through which the model can expose candidate entity-level information to the task encoder and transition model.

Looped rule execution. Given the inferred task rule $z_{\mathcal{D}}$, the query grid is initialized as a latent state

$$H^0 = \phi^{\text{grid}}(x). \tag{6}$$

The model then applies a shared Transformer-based (Vaswani et al., 2017) transition module for L recurrent steps:

$$H^{t+1} = F_\theta(H^t, z_{\mathcal{D}}), \quad t = 0, \ldots, T-1. \tag{7}$$

The final state is decoded into logits $\hat{Y}$ over the ARC color vocabulary,

$$\hat{Y} = \text{Dec}_\theta(H^T), \tag{8}$$

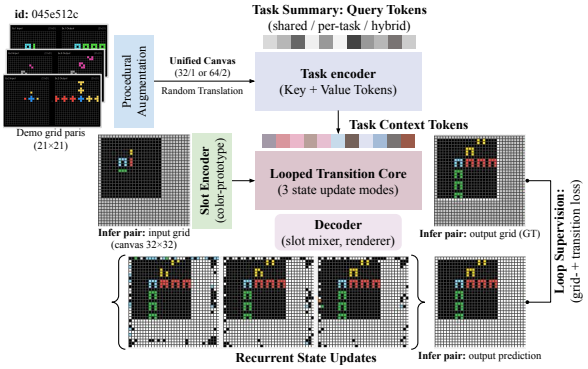


Figure 1: **Loop-OWM overview.** Demonstration pairs are augmented and encoded into task context tokens, which condition an object-aware transition model to recurrently update the query state and decode the final prediction under grid and transition supervision.

which defines a categorical distribution over colors.

$$p_\theta(c \mid u, v) = \text{softmax}\left[\hat{Y}_{(u,v)}\right]_c. \tag{9}$$

Given the ground-truth output grid y, we define a standard cross-entropy loss over valid cells:

$$\mathcal{L}_{\text{grid}} = -\frac{1}{|\mathcal{V}|} \sum_{(u,v)\in\mathcal{V}} \log p_\theta(y_{(u,v)} \mid u, v), \tag{10}$$

where $\mathcal{V}$ denotes the set of valid non-padding cells.

The loop should not be understood merely as extra depth. It provides a rollout mechanism: the same inferred rule can be repeatedly applied to an evolving internal state, allowing ARC prediction to be framed as iterative state transition rather than one-shot image translation. For simplicity, we use θ to collectively denote all learnable parameters.

4 Method

Loop-OWM instantiates the formulation above with four learnable modules: an object-centric interpreter $\mathrm{S}_\theta(\cdot)$, a task encoder $\psi_\theta^{\text{task}}(\cdot)$, a looped transition model $F_\theta(\cdot)$ with decoding module $\text{Dec}_\theta(\cdot)$. An overview of Loop-OWM is shown in Figure 1.

In this section, we focus on the ***design choices*** that instantiate the formulation: how grids are interpreted as object-aware states, how demonstrations are summarized into task representations, how the looped transition module updates the query state, and how different transition branches and supervision signals are combined in the final model.

4.1 Color-Prototype Slot Initialization

A key design choice is how slots are initialized. Instead of sampling free Gaussian slots per image, we initialize each slot from a *color prototype*. Specifically, the initial seed of slot i is given by

$$s_i^0 = E_i + b_i \in \mathbb{R}^d \tag{11}$$

where E_i is a row of the embedding table E and b_i is a learnable slot-specific bias, $b_i = \mu + \sigma \cdot \epsilon_i$ with $\mu, \sigma \in \mathbb{R}^d$ shared learnable bias and log-variance, and $\epsilon_i \sim \mathcal{N}(0, I)$ resampled at every forward pass. Special symbols such as padding or ignored tokens are not used as slot prototypes. When $K = C$, slot i is seeded by E_i directly; if K is a positive integer multiple of C, we tile the table and each color anchors K/C slots that share the same seed.

This color-prototype initialization binds slot identity to the shared ARC color palette. As a result, the same slot index has a consistent color anchor across demonstration pairs and query grids. This is useful for ARC because color is not merely a visual attribute; it often functions as a symbolic identifier for objects, regions, or roles within a task. The prototype slots therefore provide a stable object-aware interface while still allowing the Slot Attention updates to adapt each slot to the spatial context of a particular grid.

Stationary Color Anchors. Since the same color embedding table $E \in \mathbb{R}^{C \times d}$ is used both to encode grid cells and to initialize slot prototypes, unconstrained updates to E would move the color anchors during training. This can blur the intended correspondence between slot identity and ARC color identity. To keep the prototype anchors stable, we initialize E as an *orthogonal* row frame and *freeze* it throughout training:

$$EE^\top \approx d \cdot \mathbf{I}_C, \tag{12}$$

up to the subset of rows used for valid ARC colors. The scaling by d corresponds to using a gain of $\sqrt{d}$, so that each color embedding has norm $\sqrt{d}$. This keeps the input magnitude comparable to a standard embedding layer and avoids changing the scale seen by downstream patch and positional layers. Empirically, we find that freezing E stabilizes the training of the looped transition model $F_\theta(\cdot)$.

4.2 Complementary Transition Module

The module $F_\theta(\cdot)$ serves as the transition operator of the ARC world model. Given the current state $H^t \in \mathbb{R}^{L \times d}$ and the inferred task context $z_{\mathcal{D}}$, it predicts both how information should be transported across grid positions and what slot-conditioned content should be written back into the state.

The update is decomposed into two complementary branches. The first branch predicts a dense *patch-to-patch transition* operator

$$P^t \in \mathbb{R}^{L \times L}, \tag{13}$$

which routes information over grid positions. The second branch writes *slot-conditioned* content back to the patch level, producing an update

$$R^t \in \mathbb{R}^{L \times d}. \tag{14}$$

Together, these branches allow the model to represent both spatial propagation and object-aware content modification. In practice, they are implemented with a shared task-conditioned cross-attention backbone and separate prediction heads..

Slot-Conditioned Update. To map object-level information back to the grid state, inspired by VideoSAUR (Zadaianchuk et al., 2023), we introduce a slot-to-grid mixer $\mathcal{M}_\theta^{\text{s2g}}(\cdot)$ as part of the decoding module $\text{Dec}_\theta(\cdot)$. At each loop step, the mixer maintains a set of learnable positional patch queries $\mathbf{Q} = \{q_j\}_{j=1}^L$, where $q_j \in \mathbb{R}^d$ corresponds to one grid patch and is t-independent. Each patch query q_j attends to the current slots $S^t = \{s_i^t\}_{i=1}^K$ and reconstructs a slot-conditioned patch feature:

$$\tilde{h}_j^t = \sum_{i=1}^{K} \text{softmax}_i \left(\frac{\langle W_{\mathbf{Q}} q_j, W_{\mathbf{K}} s_i^t \rangle}{\sqrt{d}} \right) W_{\mathbf{V}} s_i^t. \tag{15}$$

We denote the reconstructed patch features as

$$\tilde{\mathbf{h}}^t := \{\tilde{h}_j^t\}_{j=1}^L := \mathcal{M}_\theta^{\text{s2g}}(\mathbf{Q}, S^t) \in \mathbb{R}^{L \times d} \tag{16}$$

and pass them through a lightweight MLP renderer to produce the slot-conditioned update

$$R^t := \text{MLP}_\theta(\tilde{\mathbf{h}}^t). \tag{17}$$

This branch serves as the channel through which object-level information can write new or corrected content into the grid state.

State update modes. We study three ways of composing P^t and R^t into the next state:

(1) Slot-only update. The next state is fully generated by the slot-to-grid mixer:

$$H^{t+1} = R^t. \tag{18}$$

This mode is closest to the formulation used in object-centric video models (Zadaianchuk et al., 2023; Nam et al., 2026), where P^t is only trained through auxiliary transition supervision and does not affect the forward prediction.

(2) Transport-only update. The next state is produced only by dense transport:

$$H^{t+1} = P^t H^t. \tag{19}$$

This forces the model to express the transition purely as routing over existing patch features. While suitable for permutation-like transformations, this mode is too restrictive for ARC rules that require writing new content, such as recoloring, filling, or completing shapes.

(3) Complementary update. The last tests their complementarity:

$$H^{t+1} = P^t H^t + \alpha^t \cdot R^t. \tag{20}$$

where α^t is a learnable gate parameter. The two paths therefore play complementary roles: P^t keeps the dense spatial propagation in the forward computation, while R^t provides the object-aware update to express content changes that cannot be captured by transport alone. We ablate different updating modes in Section 5.3.

4.3 Task Encoder with Summary Tokens

We use the task encoder $\psi_\theta(\cdot)$ to summarize the demonstration set $\mathcal{D}$ into a small set of context tokens that condition the looped transition model $F_\theta(\cdot)$ at every step [1].

Given the encoded demo-pair features, we construct a sequence of *demo-pair tokens* that represent the input/output states and transition-related features for each example. Depending on the update mode (introduced in Section 4.2), these pair tokens may be defined over slot-level representations or patch-level grid features. A learnable *task-summary query bank* then cross-attends to these demo pair tokens, producing the task embedding $z_\mathcal{D}$. More implementation details for the task encoder are provided in A.4.

The main design choice is how to parameterize the task-summary queries. Consider three variants:
(1) Shared-only Summary. The summary queries are shared across all tasks. Conditioning on a particular task therefore comes entirely from cross-attention to that task's demonstration tokens. This encourages amortized rule induction: the same query bank must learn a general procedure for reading demonstrations and extracting a task rule.
(2) Per-task Summary. The summary queries can instead be retrieved from a task-specific lookup table. For a training task m, the encoder uses a learned table entry E_m^{task} as the initial summary queries. This gives the model task-specific memory and may help when the same training tasks are revisited many times. However, it also ties part of the representation to the training task-id space, so unseen tasks require either newly initialized task entries and test-time adaptation.
(3) Hybrid Summary. The hybrid variant combines shared global summary tokens with task-specific summary tokens. This design separates two roles: shared tokens provide a task-agnostic interface for reading demonstrations, while task-specific tokens provide additional capacity for storing task-dependent biases. During training, task-token dropout can be applied so that the model cannot rely exclusively on task identifiers and must still use the demonstration readout.

These variants instantiate different points in the trade-off between amortized rule induction and task-specific memory. The shared variant is most compatible with unseen task generalization and cross-benchmark transfer; the per-task variant provides stronger task-specific capacity; and the hybrid variant attempts to combine both. We therefore treat the choice of task-summary parameterization as an empirical design question and compare the variants in the ablation study in Section 5.3.

4.4 Composed Transition across Loops

How the transition matrix P^t is computed and supervised across loops is an orthogonal design choice. We separate *two axes*: (i) what target the predicted transition is compared against, and (ii) which transition quantity is supervised. For the first axis, we compare raw patch-feature similarity with a slot-mediated similarity target. For the second axis, we compare supervising an individual transition matrix with supervising the composed transition induced by the full looped rollout.
(1) Dense vs. Slot-mediated Transition. Following VideoSAUR (Zadaianchuk et al., 2023), a natural target for dense patch-wise transport is the softmax-normalized cosine similarity between destination ($\cdot^{\text{dst}}$) and source ($\cdot^{\text{src}}$) patches q and p :

$$\mathcal{P}^{\text{raw}}_{q,p} = \text{softmax}_p \left(\frac{\langle \hat{f}^{\text{dst}}_q, \hat{f}^{\text{src}}_p \rangle}{\tau} \right), \tag{21}$$

where $\hat{f}$ denotes a ℓ_2-normalized patch feature and τ is a temperature.

Empirically, we found this target can be *biased* towards *identity transport*: patch features include position-dependent information, source and destination patches at the same coordinate can become similar simply because they share the same position, even when the task rule requires nontrivial content movement.

To reduce this positional identity bias, we use a *slot-mediated* transition target. We pass both the source and destination grids through the same slot encoder $\text{S}_\theta(\cdot)$ with shared color-prototype slot seeds, obtaining slot-membership masks

$$M^{\text{src}}, M^{\text{dst}} \in \mathbb{R}^{L \times K}, \tag{22}$$

[1] In implementation, at each loop step, a loop-index embedding is concatenated with $z_\mathcal{D}$ along the sequence dimension before being used to condition the transition model $F_\theta(\cdot)$.

System	#Params	**ARC-1**	**ARC-2**
ViT			
VARC (Hu et al., 2025)	18M	54.5	8.3
VARC (ensemble)	73M	60.4	11.1
recurrent models			
HRM (Wang et al., 2025)	27M	40.3	5.0
TRM (Jolicoeur-Martineau, 2025)	7M	44.6	7.8
recurrent ViT			
LoopViT-S (Shu et al., 2026)	3.8M	60.1	10.0
LoopViT-M	11.2M	63.8	11.5
LoopViT-L	18M	65.8	14.2
Loop-OWM 32/1 (Ours)	10.42M	**67.3**	**20.2**
Loop-OWM 64/2 (Ours)	10.63M	**68.5**	**22.5**

System	Type	Cost/task($)	ARC-1	ARC-2
large language models (LLMs)				
Deepseek V3.2	Base LLM	0.080	57.0	4.0
o3-Pro (High)	CoT+Syn.	7.55	59.3	4.9
o3 (Preview, Low)	CoT+Syn.	200.00	75.7	4.0
Gemini 3.1 Pro (Pre.)	CoT	0.962	98.0	77.1
GPT-5.5 (xHigh)	CoT	1.87	95.0	85.0
human results				
Human Panel	-	-	98.0	100.0
Avg. Human	-	-	60.2	-

Note. System type describes the inference strategy:
Base LLM represents single-shot inference from standard language models;
CoTs represents chain-of-thought reasoning;
CoT+Syn. denotes CoT reasoning with additional synthesized programs or candidate solutions.

Table 1: **System-level Comparison** on ARC-1 and ARC-2. **(Left)** Compact neural ARC systems. Loop-OWM uses two-stage training: offline procedural augmentation followed by task-specific test-time training with task-variant augmentation. ARC-2 results use ARC-1-to-2 transfer initialization from the checkpoint trained on ARC-1 for 200 offline epochs. We report *pass@2 mean accuracy* over 5 runs, using 30 random test-time views per task. Loop-OWM performs 4 recurrent steps; parameter counts exclude per-task summary tables. **(Right)** LLM-based systems and human references (Chollet et al., 2025; LeGris et al., 2024). LLM results are from the ARC-AGI leaderboard and use different inference strategies, including base LLM inference, chain-of-thought reasoning, and synthesized candidate solutions.

where $M^*_{j,k}$ is the membership of a src/dst patch j to slot k. We compute the slot-mediated similarity by marginalizing over the shared slot index:

$$\mathrm{sim}^{\mathrm{slot}}_{q,p} = \sum_{k=1}^{K} M^{\mathrm{dst}}_{k,q} M^{\mathrm{src}}_{k,p}, \tag{23}$$

and define the transition target by normalizing over source patches:

$$\mathcal{P}^{\mathrm{slot}}_{q,p} = \mathrm{softmax}_p \left(\frac{\mathrm{sim}^{\mathrm{slot}}_{q,p}}{\tau} \right). \tag{24}$$

This target asks whether a destination patch and a source patch are assigned to the same color-prototype slot, rather than whether their raw patch embeddings are similar. It therefore reduces positional identity bias and provides a transport target aligned with the object-centric abstraction used by the model.

Note that the color-prototype initialization provides a *shared slot coordinate system* across source and destination grids: unlike randomly initialized slots, whose ordering is permutation-symmetric, prototype slots are anchored to the ARC color palette and are therefore more consistently aligned across grids. In practice, we use this *slot-mediated target* in the main model.

(2) Final-loop vs. Composed Transition. The second design choice concerns which transition quantity should be supervised. A straightforward option is to attach the transition loss only to the final-step transition matrix

$$\hat{P}^{\mathrm{final}} = \hat{P}^{T-1}. \tag{25}$$

This supervises the last routing operation before decoding, but does not directly constrain the effective transport induced by the full looped rollout.

When applying the transition module recurrently, the overall transport from the initial query state to the final state is given by the composition of all per-step transition matrices. With the update convention $H^{t+1} = P^t H^t \ (+R^t)$, the effective transition after T steps is

$$\hat{P}^{\mathrm{total}} = \hat{P}^{T-1} \hat{P}^{T-2} \cdots \hat{P}^{0}. \tag{26}$$

We supervise $\hat{P}^*$ using a transition target computed between the query input grid and the ground-truth output grid:

$$\mathcal{L}_{\mathrm{trans}} = -\frac{1}{L} \sum_{q=1}^{L} \sum_{p=1}^{L} \mathcal{P}^*_{q,p} \log \left(\hat{P}^*_{q,p} + \epsilon \right), \tag{27}$$

where $\mathcal{P}^*_{q,p}$ is the transition target chosen (either raw or slot), and $\hat{P}^*_{q,p}$ is the supervised prediction.

Supervising $\hat{P}^{\mathrm{total}}$ aligns the transport objective with the actual multi-step routing experienced by the query state. In contrast to supervising only $\hat{P}^{\mathrm{final}}$, the composed objective discourages any single step from collapsing to a trivial identity solution unless the remaining steps compensate for the full input-to-output transport. In practice, we supervise the *composed total transition* in our main setting.

Overall objective. The full training loss is:

$$\mathcal{L} = \mathcal{L}_{\mathrm{grid}} + \lambda \, \mathcal{L}_{\mathrm{trans}}, \tag{28}$$

where λ controls the transition loss weight.

5 Experiments

Datasets. We evaluate Loop-OWM on ARC-1 and ARC-2 (Chollet, 2019; Chollet et al., 2025). Since

the original ARC files provide only a few demonstrations per task, training directly on the raw files gives very sparse supervision for learning neural transition models. We therefore use procedural augmentation to generate additional input-output pairs for the training tasks, while keeping the official evaluation tasks out. More implementation and configurations details are provided in Appendix A.1.

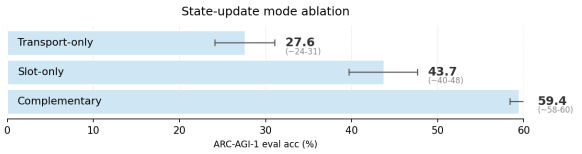
5.1 Two-stage Training

We adopt a two-stage training pipeline consisting of offline pretraining and test-time adaptation. This protocol is natural for ARC because each test task may instantiate a rule that is not directly covered by the offline training distribution. The offline stage trains general visual-symbolic machinery for grid interpretation, object-centric abstraction, and transition modeling, while the test-time stage specializes this machinery to the rule specified by the demonstrations of the current task. This separates general mechanism learning from task-specific rule specialization, matching the few-shot nature of ARC.

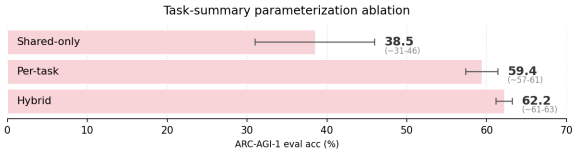
Offline procedural augmentation. For ARC-1, we augment the 400 training tasks with RE-ARC (Hodel, 2024), a procedural reimplementation of the ARC-1 training tasks. For each original training task, RE-ARC can generate fresh input-output pairs by sampling from the underlying task program. Following (Hu et al., 2025), we generate approximately 1000 pairs per task, allowing the model to train with dense task-level supervision rather than only the few examples provided in the raw ARC files. This augmentation is applied only to the training split; the evaluation split is held out and never used for generation or training.

For ARC-2, we use ARC-GEN (Moffitt, 2025), a mimetic procedural generator for ARC. Among the 1000 ARC-2 training tasks, 891 task ids are covered by ARC-GEN: 500 correspond to ARC-2-specific task instances, while the remaining 391 are overlapping with ARC-1. We generate up to 1000 pairs per covered task. Tasks whose rejection sampler exceeds a per-call time budget are dropped; this affects 16 tasks, all from the ARC-2-only subset. None of the 120 ARC-2 evaluation tasks have a corresponding generator, so the offline augmentation is leakage-free.

Test-time augmentation. We follow the the *task-variant augmentation* strategy introduced in VARC (Hu et al., 2025): for each original test task, we create 50 augmented task variants via rotations, flips, or color permutations. Resuming the offline-trained checkpoint, we further train the model on the demonstrations of the current test task and their augmented variants. For inference, we evaluate the adapted model on *multi-augmented views*, invert the transformations on the predicted grids, and consolidate the resulting candidates by *majority voting*. We use 30 random views per task and report the mean accuracy over 5 independent runs.



(a) State-update composition.



(b) Task-summary parameterization.

Figure 2: Ablation results on the ARC-1 evaluation set using the $32/1$ configuration. All variants use 100 offline training epochs and 100 test-time training epochs, and are evaluated with a reduced multi-view budget of 10 random views over 2 independent runs.

ARC-1 to ARC-2 transfer. For ARC-2, we additionally study cross-benchmark initialization from ARC-1. Specifically, we first train Loop-OWM on ARC-1 for 200 offline epochs, and then use this checkpoint to initialize offline training on ARC-2. Only task-independent parameters are transferred; task-id-dependent parameters, such as per-task summary tables, are re-initialized for ARC-2. This protocol tests whether the model can transfer general visual-symbolic machinery across ARC benchmarks. Detailed training curves and analysis are provided in Appendix A.5.

5.2 System-Level Comparison

We compare Loop-OWM with prior neural systems, LLMs, recurrent methods, and human-level references on ARC-1 and ARC-2. As shown in Table 1 **(Left)**, Loop-OWM achieves the strongest results among compact neural systems. Compared with recurrent baselines (Wang et al., 2025; Jolicoeur-Martineau, 2025; Shu et al., 2026), Loop-OWM achieves substantially higher performance while remaining in a compact parameter regime.

Table 1 **(Right)** also highlights a different trade-off from LLM-based systems. Recent LLMs can obtain strong performance on ARC, especially when using chain-of-thought (CoT) reasoning with optional synthesized candidate solutions. However,

this performance often comes with substantially higher inference cost. In contrast, Loop-OWM is a compact ViT-based model trained to operate directly on grid-structured inputs, without relying on external language reasoning, program synthesis, or LLM-scale candidate search. This suggests that explicitly modeling object-centric recurrent transitions can provide an effective inductive bias for ARC-style visual reasoning.

Finally, the gap between ARC-1 and ARC-2 remains significant for all neural systems, indicating that ARC-2 continues to pose a substantially harder generalization challenge. Nevertheless, Loop-OWM narrows this gap relative to prior compact neural models, suggesting that recurrent object-world modeling is a promising direction for scaling vision-based ARC solvers.

5.3 Ablation Study

We ablate the two design choices most directly tied to our modeling hypothesis: how the looped transition module composes dense and object-aware updates, and how the task encoder parameterizes the summary tokens used for rule induction. All variants are conducted on ARC-1 using a 32×32 canvas with patch size $p = 1$) and the same two-stage training protocol: 100-ep offline training followed by 100-ep test-time training. We use 10 random views per task over 2 independent runs and report pass@2 mean accuracy on ARC-1 evaluation set. More choices are analyzed in the appendix.

(a) State update composition. As introduced in Section 4.2, we first compare three update variants: *slot-only*, *transport-only*, and *complementary update*. To isolate this factor, all variants use only the per-task summary parameterization. Figure 2a shows a clear ordering; the complementary update performs best. This supports our hypothesis that ARC world-state transitions require both spatial transport and object-aware content updates, rather than pure routing alone.

(b) Task-summary parameterization. We next compare three variants introduced in Section 4.3: *shared-only*, *per-task*, and *hybrid*. To isolate this factor, all variants use the same complementary update mode. Figure 2b highlights an empirical trade-off. Shared-only summary tokens provide the cleanest amortized formulation, but offer limited task-specific capacity during test-time training. Adding per-task summary entries substantially improves performance, suggesting that these entries act as adaptable placeholders for task-specific rule information. The hybrid is slightly better than per-task-only, indicating that shared summary queries still provide a useful amortized rule-induction bias when combined with task-specific capacity.

6 Limitations and Future Work

Although Loop-OWM uses recurrence as a rollout mechanism, we do not observe stable, human-interpretable semantics at individual loop steps, as shown in Figure 3. Ideally, different loops might correspond to reusable reasoning stages such as localization, transport, recoloring, or boundary refinement. In practice, however, computation is distributed across steps without a consistent cross-task pattern. This is expected to some extent: standard ARC provides only input-output supervision and *does not supervise intermediate states*; the internal rollout is therefore underdetermined as long as the final grid is correct.

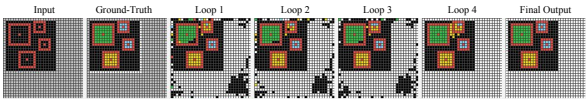


Figure 3: Example rollout without stable semantics.

Addressing this limitation requires trajectory-level supervision. As a first step, we are building an ARC-2-derived dataset with principled intermediate-state annotations, currently covering approximately 500 tasks. By supervising sub-transformations rather than only final outputs, such data could make recurrent updates more semantically grounded and move looped ARC models toward more inspectable world models.

7 Conclusion

We frame ARC as rule induction over visual-symbolic state transitions. Under this view, demonstrations define how a structured grid state should change, and solving a new query requires executing the inferred transition rather than merely predicting pixels. Our proposed model instantiates this idea with object-aware grid interpretation, task-summary tokens, and recurrent transition updates that combine dense transport with slot-conditioned writing. Results on ARC-1 and ARC-2, together with the targeted ablations, show that this formulation provides an effective inductive bias for compact neural ARC solvers. More broadly, the study suggests that ARC can serve not only as a benchmark for language- or program-based reasoning, but also as a controlled setting for learning composable world-state dynamics.

References

Christopher P. Burgess, Loic Matthey, Nicholas Watters, Rishabh Kabra, Irina Higgins, Matt Botvinick, and Alexander Lerchner. 2019. Monet: Unsupervised scene decomposition and representation. *arXiv preprint arXiv:1901.11390*.

François Chollet. 2019. On the measure of intelligence. *arXiv preprint arXiv:1911.01547*.

François Chollet, Mike Knoop, Gregory Kamradt, Bryan Landers, and Henry Pinkard. 2025. ARC-AGI-2: A new challenge for frontier AI reasoning systems. *arXiv preprint arXiv:2505.11831*.

Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. 2021. An image is worth 16x16 words: Transformers for image recognition at scale. In *Proc. of the International Conf. on Learning Representations (ICLR)*.

Jonas Geiping, Sean McLeish, Neel Jain, John Kirchenbauer, Siddharth Singh, Brian R. Bartoldson, Bhavya Kailkhura, Abhinav Bhatele, and Tom Goldstein. 2025. Scaling up test-time compute with latent reasoning: A recurrent depth approach. In *Advances in Neural Information Processing Systems (NeurIPS)*.

Kaiming He, Xinlei Chen, Saining Xie, Yanghao Li, Piotr Dollár, and Ross B. Girshick. 2022. Masked autoencoders are scalable vision learners. In *Proc. IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*, pages 15979–15988.

Michael Hodel. 2024. Addressing the abstraction and reasoning corpus via procedural example generation. *arXiv preprint arXiv:2404.07353*.

Keya Hu, Ali Cy, Linlu Qiu, Xiaoman Delores Ding, Runqian Wang, Yeyin Eva Zhu, Jacob Andreas, and Kaiming He. 2025. Arc is a vision problem! *arXiv preprint arXiv:2511.14761*.

Alexia Jolicoeur-Martineau. 2025. Less is more: Recursive reasoning with tiny networks. *arXiv preprint arXiv:2510.04871*.

Jordan, Keller, Jin, Yuchen, Boza, Vlado, Jiacheng, You, Cecista, Franz, Newhouse, Laker, Bernstein, and Jeremy. 2024. Muon: An optimizer for hidden layers in neural networks.

Daniil Kirilenko, Vitaliy Vorobyov, Alexey Kovalev, and Aleksandr Panov. 2024. Object-centric learning with slot mixture module. In *Proc. of the International Conf. on Learning Representations (ICLR)*.

Seungpil Lee, Woochang Sim, Donghyeon Shin, Wongyu Seo, Jiwon Park, Seokki Lee, Sanha Hwang, Sejin Kim, and Sundong Kim. 2024. Reasoning abilities of large language models: In-depth analysis on the abstraction and reasoning corpus. *arXiv preprint arXiv:2403.11793*.

Solim LeGris, Wai Keen Vong, Brenden M. Lake, and Todd M. Gureckis. 2024. H-ARC: A robust estimate of human performance on the abstraction and reasoning corpus benchmark. *arXiv preprint arXiv:2409.01374*.

Francesco Locatello, Dirk Weissenborn, Thomas Unterthiner, Aravindh Mahendran, Georg Heigold, Jakob Uszkoreit, Alexey Dosovitskiy, and Thomas Kipf. 2020. Object-centric learning with slot attention. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 33, pages 11525–11538.

Michael D. Moffitt. 2025. ARC-GEN: A mimetic procedural benchmark generator for the abstraction and reasoning corpus. *Preprint*, arXiv:2511.00162.

Heejeong Nam, Quentin Le Lidec, Lucas Maes, Yann LeCun, and Randall Balestriero. 2026. Causal-jepa: Learning world models through object-level latent interventions. *arXiv preprint arXiv:2602.11389*.

Simon Ouellette. 2024. Towards efficient neurally-guided program induction for ARC-AGI. *arXiv preprint arXiv:2411.17708*.

Nikunj Saunshi, Nishanth Dikkala, Zhiyuan Li, Sanjiv Kumar, and Sashank J. Reddi. 2025. Reasoning with latent thoughts: On the power of looped transformers. In *Proc. of the International Conf. on Learning Representations (ICLR)*.

Avi Schwarzschild, Eitan Borgnia, Arjun Gupta, Furong Huang, Uzi Vishkin, Micah Goldblum, and Tom Goldstein. 2021. Can you learn an algorithm? generalizing from easy to hard problems with recurrent networks. In *Advances in Neural Information Processing Systems (NeurIPS)*.

Wen-Jie Shu, Xuerui Qiu, Rui-Jie Zhu, Harold Haodong Chen, Yexin Liu, and Harry Yang. 2026. Loopvit: Scaling visual arc with looped transformers. *arXiv preprint arXiv:2602.02156*.

Kartik Singhal and Gautam Shroff. 2025. Conceptsearch: Towards efficient program search using llms for abstraction and reasoning corpus (arc). In *Proc. of the Conf. on Artificial Intelligence (AAAI)*.

Jianlin Su, Murtadha Ahmed, Yu Lu, Shengfeng Pan, Wen Bo, and Yunfeng Liu. 2024. Roformer: Enhanced transformer with rotary position embedding. *Neurocomputing*, 568:127063.

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

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

Johan S. Wind. 2020. DSL solution to the ARC challenge. GitHub repository.

Ziyi Wu, Nikita Dvornik, Klaus Greff, Thomas Kipf, and Animesh Garg. 2023. Slotformer: Unsupervised visual dynamics simulation with object-centric models. In *Proc. of the International Conf. on Learning Representations (ICLR)*.

Yudong Xu, Wenhao Li, Pashootan Vaezipoor, Scott Sanner, and Elias B. Khalil. 2024. LLMs and the abstraction and reasoning corpus: Successes, failures, and the importance of object-based representations. *Journal of Machine Learning Research (JMLR)*.

Andrii Zadaianchuk, Maximilian Seitzer, and Georg Martius. 2023. Object-centric learning for real-world videos by predicting temporal feature similarities. In *Advances in Neural Information Processing Systems (NeurIPS)*.

A Appendix

A.1 Configurations and Model Size

We report the training configurations in Table 2. In addition, Table 3 shows the parameter breakdown of our $64/2$ model (64×64 canvas with patch size $p = 2$). The model has 10.63M parameters excluding the per-task summary table. This table is tied to the number of training task identifiers and therefore scales with the benchmark-specific task vocabulary rather than with the reusable model architecture. For ARC-2, the per-task summary table contains $1000 \times (N^q_{\text{task}} =)8 \times 256 \approx 2.048$M parameters, where N^q_{task} defines the capacity of the summary query bank (see A.4). Most parameters are concentrated in three components: the task encoder, the transition core, and the decoder. Together, these account for approximately 84.4% of the model. The slot module is shared between the task encoder and the looped predictor and contributes only 10.6% of the total parameter count. We use Muon (Jordan et al., 2024) optimizer with cosine lr-scheduler through training.

Number of recurrent steps. We ablate the number of recurrent rollout steps in the $32/1$ configuration, testing $T \in \{3, 4, 5, 6\}$. We ablate $T \in \{3, 4, 5, 6\}$ rollout steps and find that $T = 4$ gives the best validation performance, with no visible benefit from additional steps. We therefore use $T = 4$ throughout all experiments.

offline training	
epochs	200
warmup epochs	0
optimizer	Muon, $\beta = 0.95$
batch size (per GPU)	64
GPU type	H200 $\times$4
learning rate	1e$-$3
learning rate scheduler	cosine (min 0.05$\times$)
weight decay	0.01
dropout	0
task query bank N^q_{task}	8
use loop-index embedding	true
loop steps	4
$\mathcal{L}_{\text{trans}}$ weight	0.1
test-time training	
epochs	200
warmup epochs	0
optimizer	Muon, $\beta = 0.95$
batch size	8
learning rate	5e$-$4
learning rate scheduler	cosine (min 0.05$\times$)
weight decay	0.01
dropout	0
task query bank N^q_{task}	8
use loop-index embedding	true
loop steps	4
$\mathcal{L}_{\text{trans}}$ weight	0.1
views per test pair	30
independent runs	5

Table 2: **Training Configurations.**

A.2 Data Preprocessing

Following the visual-canvas setup in VARC (Hu et al., 2025), we place ARC grids, whose spatial size is at most 30×30, on an image-like canvas. We use a $2\times$ scaling ratio with a 64×64 canvas and patch size $\mathrm{p} = 2$, producing a 32×32 token grid with $L = 1024$ patch tokens. This keeps the transformer sequence length comparable to a 32×32 canvas with 1×1 patches, while allowing each 2×2 patch to contain richer local color patterns.

Since ARC input and output grids may have different raw shapes, we follow (Hu et al., 2025) and handle shape prediction on the fixed canvas. Canvas cells outside the placed grid receive an `[IGNORE]` token, while output grids additionally use a `[PAD]` token to mark the right column and bottom row immediately adjacent to the valid output region [2]. The grid cross-entropy is computed on all non-`[IGNORE]` cells; `[PAD]` tokens are supervised as meaningful boundary tokens, allowing the model to recover the output shape from its own predictions. At inference time, we locate the rightmost and bottommost predicted `[PAD]` tokens and crop the canvas accordingly. Following (Hu et al., 2025), we also apply a foreground mask in the predictor self-attention layers by masking keys corresponding to `[IGNORE]` cells; `[PAD]` keys are left unmasked because they encode valid output-boundary information.

[2] In our implementation, `[IGNORE]` and `[PAD]` correspond to `IGNORE_INDEX=10` and `PAD_INDEX=11`, respectively.

Component	#Params	Share	Type	Hidden dim	Description
Task encoder $\psi_\theta(\cdot)$	3.17M	29.8%	4 blocks	256	Cross- and Self-attn, MLP (ff= 512)
Transition $F_\theta(\cdot)$	3.16M	29.8%	6 blocks	256	Looped transformer core, 4 steps
Decoder $\mathrm{Dec}_\theta(\cdot)$	2.63M	24.8%	Cross-Attn + MLP	256	Slot mixer, renderer, output head (ff= 512)
Slot encoder $S_\theta(\cdot)$	1.12M	10.6%	Slot attention	256	Shared color-prototype, 24 slots, 3 iters
Patch embedding	0.26M	2.5%	Conv2d (1×1)	256	Convolutional patch embedding
2D pos embedding	0.26M	2.5%	sin-cosine	1024×256	Positional embedding table
Output head	0.01M	0.1%	Linear	$256 \to 12$	Color prediction head
Other small modules	< 0.01M	$< 0.1\%$	others	256	Color-, role-, step-embeddings, LayerNorm
Total	10.64M	100%			Excluding per-task summary bank

Table 3: Parameter breakdown of the Loop-OWM (64/2) configuration, excluding the per-task summary table. The total model size is 10.64M parameters. The task encoder, transition core, and decoder dominate the parameter count, while the shared slot tokenizer remains comparatively small.

During training, we apply VARC-style canvas augmentations, including random translation, scale changes, rotations, flips, and color permutations. Spatial augmentations are applied consistently to the input and output grids of each pair so that the underlying task rule is preserved.

A.3 Positional Embeddings

We use absolute 2D sine-cosine positional embeddings (He et al., 2022) at several sites in the model. At the input, each cell token is formed by concatenating a color embedding slice with a 2D positional embedding, rather than adding position to color. This keeps color and position in separate channels at initialization, which preserves the initial dot-product alignment between color-prototype slot seeds and same-color patches. Importantly, this does not force slots to ignore position: $S_\theta(\cdot)$ applies learnable projections to the full token dimension, so the model can still learn to use positional information when useful.

For slot-mediated update (Sec. 4.2), we also add 2D positional embeddings to the per-patch queries $\mathbf{Q} = \{q_j\}_{j=1}^{L}$ in the slot mixer $\mathcal{M}_\theta^{\mathrm{s2g}}(\cdot)$, enabling the allocator to route slot content to specific grid positions. In addition, the Transformer-based transition module $F_\theta(\cdot)$ applies 2D RoPE (Su et al., 2024) to the image-token attention block, providing a relative positional signal complementary to the absolute sine-cosine embeddings.

A.4 Task Encoder Implementation

The task encoder $\psi_\theta(\cdot)$ (introduced in Section 4.3) maps the demonstration set $\mathcal{D}$ to a sequence of context tokens $z_\mathcal{D}$ that condition the looped transition model $F_\theta(\cdot)$.

For each demonstration pair, let H_{in} and H_{out} denote the encoded input and output features. We first compute a demo-side transition T_{demo} as a softmax-normalized feature similarity from input units to output units. This provides a lightweight alignment between the two grids and is used only to construct *transition-aware features* for the task encoder. Using this alignment, we represent each demonstration pair with $r \in \{3, 4\}$ role streams: (1) the input features H_{in}, (2) the output features H_{out}, (3) the post-transport residual

$$\Delta_{\mathrm{demo}} = H_{\mathrm{out}} - T_{\mathrm{demo}} H_{\mathrm{in}},$$

and, (4) optionally, a position-flow stream obtained by transporting the per-unit positional embeddings through the same transition T_{demo}. The residual stream highlights content that cannot be explained by directly transporting the input features, while the optional position-flow stream exposes the spatial displacement implied by the demo-side transition.

Each role stream is shifted by a learnable role embedding $e_{\mathrm{role}} \in \mathbb{R}^d$, shared across demonstrations and tasks. The role-tagged streams are then concatenated to each demo-pair and projected to the model dimension. This produces the demo-pair tokens attended by the task-summary query bank.

The summary query bank consists of $N_{\mathrm{task}}^q (= 8)$ learnable queries. These queries cross-attend to the demo-pair tokens through D_{task} $(= 4)$ layers, producing the task context tokens $z_\mathcal{D}$. Depending on the summary-token variant introduced in Section 4.3, the initial queries may be shared across all tasks, retrieved from a per-task lookup table, or formed by combining shared and task-specific queries. The resulting context tokens are concatenated with a loop-index embedding at each rollout step before conditioning the transition model $F_\theta(\cdot)$.

Role embeddings. Inside the task encoder, each

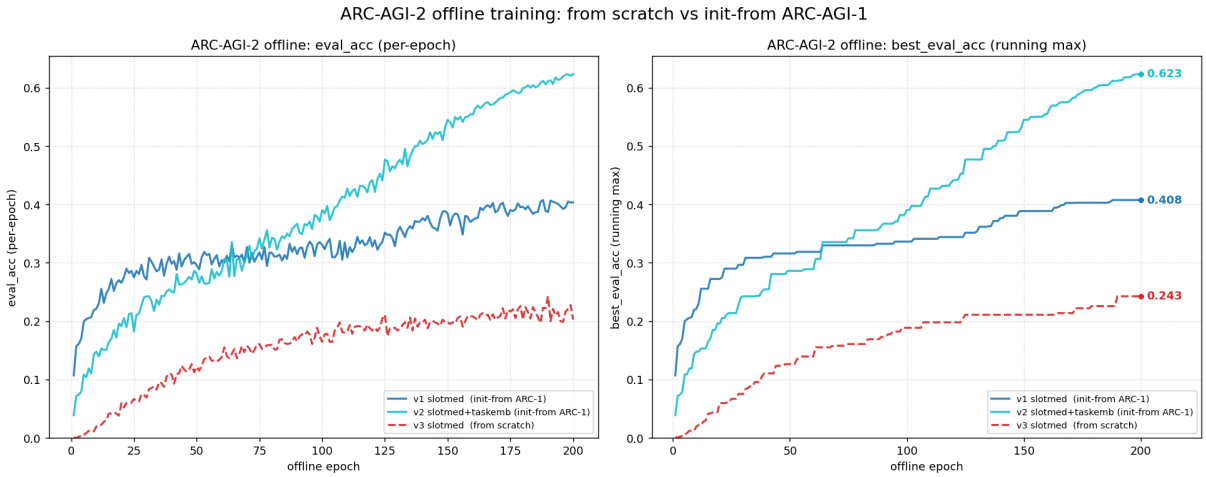


Figure 4: ARC-2 offline training from scratch versus initialization from an ARC-1 checkpoint trained for 200 offline epochs. Left: per-epoch validation accuracy. Right: running maximum validation accuracy.

demonstration pair is decomposed into $r \in \{3, 4\}$ role-specific streams at a common unit granularity: slots in slot-encoder mode and patches otherwise. The streams correspond to the demonstration input, demonstration output, post-transport residual, and optionally a position-flow stream obtained by transporting per-unit positional embeddings through the same demo-side transition. We add a learnable role embedding $e_r \in \mathbb{R}^d$ to each stream, shared across demonstrations and units. The role-tagged streams are then concatenated along the channel dimension and linearly projected to the model dimension, producing the demo-pair tokens attended by the task-summary queries. The additive role tag complements channel concatenation by giving each stream a learnable role anchor before projection.

A.5 Cross-benchmark Transfer Learning

ARC-1 and ARC-2 share the same grid and color interface, but ARC-2 defines a new and more challenging distribution of tasks. This creates a natural transfer setting for evaluating training efficiency: if the model has learned general visual-symbolic machinery on ARC-1, then adapting to ARC-2 should require less training than learning the same machinery from scratch.

We study 1-to-2 transfer by first training Loop-OWM on ARC-1 for 200 offline epochs, and then using the resulting checkpoint to initialize ARC-2 offline training. We copy all task-independent components; parameters tied to benchmark-specific task identities, such as per-task summary tables, are re-initialized for ARC–2. We reset the optimizer before training on ARC-2 and use cosine learning-rate scheduler for both runs.

Figure 4 compares ARC-2 offline validation accuracy between model training from scratch (v3, red curve) with model initialized from ARC-1 checkpoints (v1, deeper-blue curve). The transferred model converges faster and reaches substantially higher offline validation accuracy, indicating that ARC-1 pretraining provides a useful initialization for object-centric abstraction and task-conditioning mechanism. Since task-id-dependent parameters are not reused (the [+taskembed] curve (v2, lighter-blue) isolates the effect of adding a per-task summary, which is re-initialized for ARC-2), this improvement between v1 and v3 cannot be explained by carrying over task-specific lookup memory for repeated ARC-1 tasks.