ARC-AGI Solver: Fine-Tuning Qwen with LoRA and Augmented Inference

Minseo Kim* [1] **Yeonjae Kim*** [1] **Jeonghun Park*** [1] **Hyunwoo Lee*** [1]

Abstract

We present a pipeline that fine-tunes a large pretrained language model (Qwen-4B) on ARC-AGI tasks using efficient LoRA adapters and data augmentation. The input-output grid pairs in each task are serialized as tokens and concatenated into a few-shot prompt, enabling the model to learn grid transformations from examples. At inference time, we construct a prompt from the three given examples of a task (few-shot prompting), optionally perform test-time tuning on these examples, and generate multiple output candidates via the model. The final output is obtained by aggregating these candidates using a simple grid-wise majority vote. This approach leverages pretrained reasoning capabilities, LoRA fine-tuning, and augmentation to solve the ARC-AGI tasks. Our repositories are publicly available at https://github.com/minseo25/arc-agi-solver.

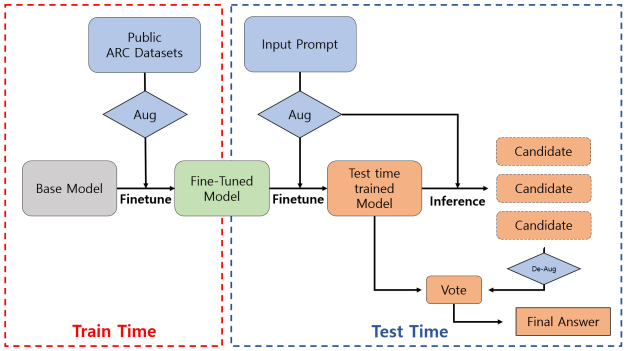

Figure 1. Overview of our full pipeline, including training and inference stages.

1. Introduction

The Abstraction and Reasoning Corpus (ARC) is a benchmark of visual reasoning tasks. Each task consists of several input–output pairs of colored grids, and the goal is to infer the transformation rule to apply to a new input grid. In our setting, a synthetic ARC-AGI dataset of 380 tasks is provided, with 300 tasks available for training and the rest reserved for evaluation. For each test task, the system is given three example input–output grid pairs and one target input; the objective is to predict the correct output grid following the pattern illustrated by the examples.

Recent ARC-AGI approaches have largely abandoned exhaustive *DSL-based* program synthesis (Ouellette, 2024) in favor of direct *transduction* using pretrained models. These methods typically apply a brief test-time training step on the few available examples, combine multiple outputs via simple ensembling (e.g. majority voting or confidence-based tie-breaking), and some even incorporate spatially aware architectures to natively model 2D grid structure.

In this work, we adopt a pretrained Qwen-4B fine-tuned via LoRA on serialized grid examples. At inference, we optionally perform a brief test-time adaptation and then ensemble multiple augmented predictions with grid-wise majority voting. This approach unites efficient parameter adaptation, few-shot prompting, and robust ensembling to generalize effectively to unseen ARC tasks.

2. Method

Our approach consists of a parameter-efficient fine-tuning phase and an inference strategy incorporating task-specific adaptation and ensembling. Below, we outline the training setup and inference procedure in detail.

2.1. Training

We fine-tune the Qwen-4B model—a 4-billion-parameter language model from Qwen—on the ARC-style training tasks using Low-Rank Adaptation (LoRA). This allows us to adapt a large model efficiently with significantly fewer trainable parameters.

*Equal contribution [1]Department of Computer Science and Engineering, Seoul National University.

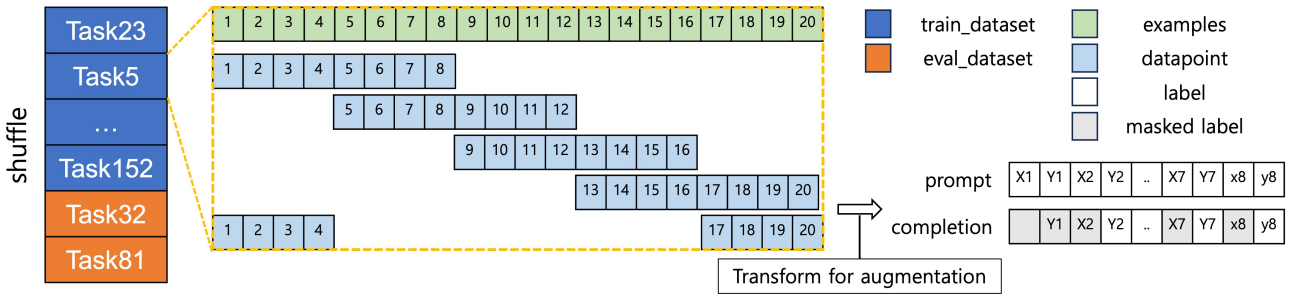

Figure 2. Dataset Construction Pipeline

2.1.1. Model and Configuration

We use the `Qwen/Qwen3-4B` Hugging Face model as our base. LoRA adapters with rank $r = 16$ and scaling factor $\alpha = 16$ are inserted into all attention projection modules and MLP layers, specifically targeting `q_proj`, `k_proj`, `v_proj`, `o_proj`, `gate_proj`, `up_proj`, `down_proj`.

To further reduce the vocabulary size and enhance output accuracy, we replace the `input embedding` and `LM head` with smaller, optimized weights. We shrink these weights by restricting the vocabulary to digits (0–9), line delimiters (`\n`), and a selected set of special formatting tokens. Additionally, to fine-tune exclusively the LM head without altering the embedding weights, we clone the existing weights of `input embedding` specifically for the `LM head` and fine-tune them with a learning rate lower than that used for LoRA adapters. All other model parameters remain frozen during fine-tuning.

2.1.2. Optimization and Hyperparameters

We train the model using the 8-bit optimizer with a learning rate of 5×10^{-5} for LoRA and 1×10^{-5} for the LM head. Training is performed on a single 12GB GPU with gradient accumulation (batch size 4), cosine scheduling, FP16 mixed precision, and early stopping. Both adapter and LM head weights are saved at regular checkpoints.

2.1.3. Training Objective and Data

Input and Label Construction The model is trained on a prompt-completion objective. Each training instance consists of some examples serving as context and one target query whose output the model predicts. Only tokens corresponding to the target output contribute to the loss computation, while input tokens and context example outputs are masked out.

Dataset Preparation and Augmentation We use 300 official ARC training tasks, randomly partitioned into training and evaluation sets. For each task, multiple datapoints are generated using a sliding-window strategy with overlapping context examples (as depicted in Figure 2). Each datapoint in Figure 2 comprises training pairs and one test pair. To enhance data diversity during training, the following augmentations are applied:

- **Color shuffling:** Non-zero color tokens are randomly permuted during training to prevent overfitting to specific color-ID mappings.
- **Geometric transforms:** Random horizontal flips and rotations by multiples of 90 degrees are applied to the grids.
- **Prompt shuffling:** The order of the three context examples is randomized to mitigate positional biases.

2.1.4. Training with Joint Loss

In the setup described above, each task is defined by multiple examples plus a single test input and its corresponding test output. Conventionally, one uses teacher forcing(Lamb et al., 2016) and computes the loss only on the test output. Here, however, we treat the examples themselves as additional "test-like" data with limited context. Accordingly, we train with a **Joint loss** that supervises not only the test output but also the outputs of all preceding examples. Because the very first example has no prior context, we apply a mask to its output so that it does not contribute to the loss. This strategy enables the model to learn from multiple contextually diverse data points within each task, and by increasing the number of examples, we can cover more data in fewer training steps.

2.2. Inference

At test time, the trained model is used to solve new ARC tasks using few-shot prompting and optional task-specific tuning.

2.2.1. Prompt Construction

For each test task, we construct a prompt by serializing the three provided input-output examples, followed by the test input grid and an empty "Output:" field. The format

strictly adheres to the template used during training, with grid delimiters and special tokens to guide the model. The prompt includes a fixed preamble string and formatting tokens to help the model identify each part of the example. This prompt is then tokenized and fed into the model for autoregressive decoding.

2.2.2. Test-Time Training (TTT)

Test-time training (TTT)(Sun et al., 2020) is a technique that adapts the model on the given task at test time. To avoid overfitting on the same support examples—and thus preserve the ability to predict the test input's output—we generate multiple variants of the original task rather than training on it repeatedly. Section 2.1.3 describes three such augmentation methods: random geometry, random color, and random shuffling. We observe that different combinations of these augmentations lead to varying TTT performance. During TTT, we freeze the language-model head and update only the LoRA adapters. Once inference on the current task is complete, we discard the task-specific LoRA parameters and restore the original adapters. Finally, to respect GPU limitations and overall time constraints, we limit the number of augmented tasks per TTT session and terminate TTT after a preset duration before proceeding directly to inference.

2.2.3. Candidate Grid Generation and Selection

In order to improve inference robustness, we first apply random geometry transformations to the given task, generating multiple augmented versions of the prompt. For each augmented prompt, we run the model to produce not only the output grid but also the per-token logits and the overall sequence probability. This yields a set of candidate grids, each paired with its score information, which we then use to make our final prediction.

The available grid-selection policies are as follows:

- **Naive Voting:** Perform a simple majority vote across the full candidate grids.
- **Grid-wise Selection:** Choose the candidate whose total log-probability (sum of its token logits) is highest.
- **Cell-wise Argmax:** For each cell position, select the value coming from the candidate with the highest logit at that cell.
- **Voted Grid-wise (ours):** First identify the most common grid(s) via voting, then break any ties by comparing total sequence probabilities.

Among these, we focus on the voted grid-wise policy, which achieved the best performance in our evaluation. A more detailed explanation is given below in Algorithm 1.

Algorithm 1 Voted Grid-wise with Augmentation and Tie-Breaking

Input: Test prompt P, model M, augmentation functions $\{A_k\}_{k=1}^{K}$
Output: Final prediction grid $\hat{G}$
for $k = 1$ **to** K **do**
  $P_k \leftarrow A_k(P)$ // Apply k-th augmentation
  $G_k, \log p_k \leftarrow M(P_k)$ // Inference + log-probability
  $G_k^{\text{orig}} \leftarrow A_k^{-1}(G_k)$ // Reverse augmentation
end for
$\mathcal{C} \leftarrow$ list of G_k^{orig} with counts
$\mathcal{T} \leftarrow$ top grids with highest count in $\mathcal{C}$
if $|\mathcal{T}| = 1$ **then**
  $\hat{G} \leftarrow \mathcal{T}[1]$
else
  $\hat{G} \leftarrow \arg\max_{G_i \in \mathcal{T}} \log p_i$
end if
return $\hat{G}$

By combining geometry-based augmentation with sequence-level scoring, we reduce the variance introduced by random decoding noise and make our final predictions more stable and accurate.

3. Evaluation

We first constructed the training dataset following the procedure described in Section 2 and used it to train our model. During training, we applied early stopping based on the validation loss. Once training was complete, we evaluated the final model's accuracy on the training dataset. To boost our performance metrics, we then applied three different techniques.

3.1. Eval Joint Loss Model

In our initial setup, we trained the model with teacher forcing using three support examples and a test input **X** to predict the test output **Y**, computing the loss on only one example per step. Under this configuration, our local evaluation scores fell in the 30–40 point range. Next, by switching to a joint loss over all support examples during training, we saw performance improve to roughly 50–60 points. Keeping the same number of epochs but increasing the support set size to seven examples further boosted scores into the 60–70 point range. Finally, we extended training for many more epochs in this fixed environment, saving a checkpoint at every step.

3.2. Eval with TTT

Before finalizing the model, we performed local evaluation by applying test-time training (TTT) using parameters from intermediate checkpoints. We fixed the total number of

tasks at 12 but generated them in four different ways. The results are shown in Table 1. Based on these experiments, we selected the combination of random geometry with ring order for our final TTT setup.

Table 1. Local evaluation scores under different TTT strategies with 12 fixed tasks. Δ indicates the change relative to the no-TTT baseline.

TTT Strategy	Score	Δ
No TTT	68	–
Random (geo, color, shuffling)	62	–6
Random (geo, color) + ring shuffling	64	–4
Random geo + ring shuffling	71	+3

3.3. Eval with Candidate Generation

To improve robustness during inference, we apply test-time augmentation—specifically geometric transformations such as horizontal flips and 90-degree rotations—to generate multiple transformed variants of each input. We then perform inference on each variant and reverse the transformations (de-augmentation) to obtain a set of candidate outputs. We evaluate several selection strategies: choosing the most frequent candidate (84 score), selecting the one with the highest log-probability (81 score), aggregating high-confidence cells across candidates (77 score), and majority voting with log-probability tie-breaking, which achieves the best performance (86 score). This approach overall enhances stability and accuracy by reducing variance across generations.

3.4. Performance Improvements Across Methods

We report performance improvements as we progressively incorporated our techniques. Starting from a baseline trained with teacher forcing and single-example loss, we introduced joint loss computation over support examples, which significantly improved training effectiveness. Increasing the support set size further boosted accuracy, and extended training epochs yielded additional gains. In the inference stage, applying test-time augmentation and robust candidate selection strategies (e.g., log-probability tie-breaking) led to further improvements. Figure 3 visualizes this trajectory, demonstrating how each method cumulatively contributed to a final score of 81. All reported scores are based on official leaderboard evaluations, reflecting performance on the held-out ARC test set.

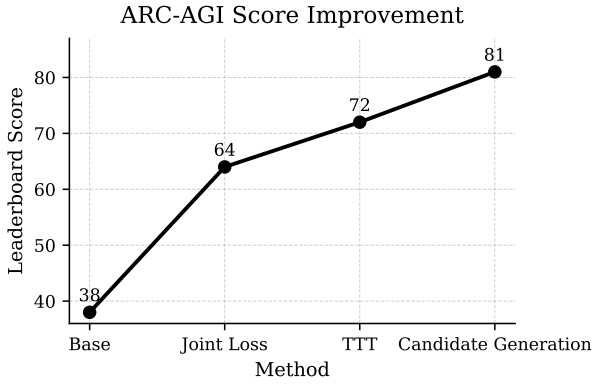

Figure 3. Progressive leaderboard score improvements across methods. Each step reflects a cumulative addition of training or inference strategies described in Section 3.

4. Conclusion

In this paper, we demonstrated that enriching our training regime with data augmentation and a joint-loss objective substantially boosts model learning on few-shot ARC tasks. By supervising not only the test output but also all support examples, our joint loss formulation enables the model to capture richer contextual dependencies and yields a marked improvement in training-time effectiveness.

At inference time, we further increase accuracy through two complementary strategies. First, test-time training (TTT) applies geometric transformations alongside ring-order permutations of the support examples—while keeping color fixed—to adapt more effectively to each new task. Second, our candidate-generation pipeline, combined with a voted grid-wise selection policy that breaks ties via total sequence log-probability, yields the strongest and most stable final predictions. Together, these methods achieve an overall leaderboard score of 81, setting a new benchmark for robust few-shot ARC performance.

5. Contribution

Minseo Organized the team's Notion page, conducted literature searches for ARC-AGI papers, implemented data augmentation pipelines, trained models and performed hyperparameter tests.

Yeonjae Built the initial model baseline, ported and integrated The Architects framework, performed TTT-related tests and experiments, and oversaw model training.

Jeonghun Integrated the user interfaces, refactored the codebase to adopt Hydra, applied data augmentation techniques, executed a range of tests and experiments, and trained the models.

Hyunwoo Managed the team repository and model zoo, implemented the TTT components, trained the models, and prepared the project report.

References

Lamb, A. M., Goyal, A., Zhang, Y., Zhang, S., Courville, A. C., and Bengio, Y. Professor forcing: A new algorithm for training recurrent networks. In *Advances in Neural Information Processing Systems 29 (NIPS 2016)*, pp. 4601–4609, 2016.

Ouellette, S. Towards efficient neurally-guided program induction for ARC-AGI. *arXiv preprint arXiv:2411.17708*, 2024. doi: 10.48550/arXiv.2411.17708.

Sun, Y., Wang, X., Liu, Z., Miller, J., Efros, A. A., and Hardt, M. Test-time training with self-supervision for generalization under distribution shifts. In *Proceedings of the 37th International Conference on Machine Learning (ICML)*, pp. 9229–9248, 2020.