The Surprising Effectiveness of Test-Time Training for Abstract Reasoning

Ekin Akyürek Mehul Damani Linlu Qiu Han Guo Yoon Kim Jacob Andreas

Massachusetts Institute of Technology

Abstract

Language models have shown impressive performance on tasks within their training distribution, but often struggle with novel problems requiring complex reasoning. We investigate the effectiveness of test-time training (TTT)—updating model parameters temporarily during inference using a loss derived from input data—as a mechanism for improving models' reasoning capabilities, using the Abstraction and Reasoning Corpus (ARC) as a benchmark. Through systematic experimentation, we identify three crucial components for successful TTT: (1) initial finetuning on similar tasks (2) auxiliary task format and augmentations (3) per-instance training. TTT significantly improves performance on ARC tasks, achieving up to 6× improvement in accuracy compared to base fine-tuned models; applying TTT to a 8B-parameter language model, we achieve 53% accuracy on the ARC's public validation set, improving the state-of-the-art by nearly 25% for public and purely neural approaches. By ensembling our method with recent program generation approaches, we get SoTA public validation accuracy of 61.9%, matching the average human score. Our findings suggest that explicit symbolic search is not the only path to improved abstract reasoning in neural language models; additional test-time applied to continued training on few-shot examples can also be extremely effective.

1 Introduction

Large-scale neural language models (LMs) excel at performing tasks that occur in their training data, and often elementary variations or compositions of those tasks (Brown et al., 2020; Todd et al., 2024). Given natural language task specifications or a small number of examples, LMs often successfully infer the desired task and produce an appropriate output. But can LMs also solve new problems, involving non-trivial reasoning, planning, or string manipulation of a kind very different from their pre-training data? This question is central to understanding the novel skill acquisition capabilities of current AI systems, which has been proposed as a key measure of intelligence (Chollet, 2019).

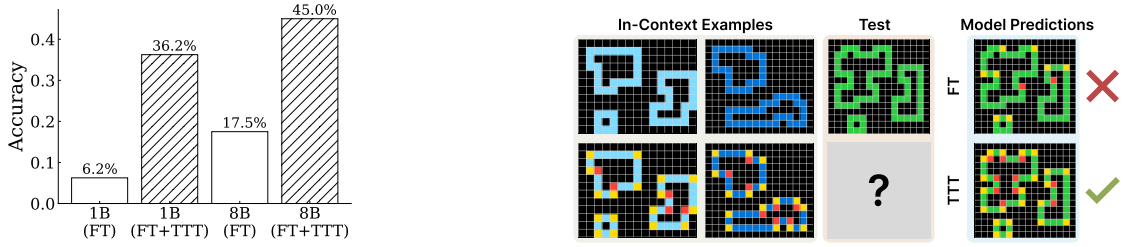

Figure 1: **(Left):** Pass@2 accuracy on a subset of 80 randomly selected ARC validation tasks. TTT boosts the performance of fine-tuned models (FT) by up to 6×, with consistent improvements across different model sizes. **(Right):** Example of a task that the model successfully solves only after applying TTT. Full dataset results in Section 6.

For complex and novel tasks, it is often difficult to obtain a correct answer simply by sampling from an LM (Wu et al., 2023). However, a significant finding in recent years has been that LM performance can be substantially improved by augmenting LM decoding with additional *test-time computation*. Methods in this category include chain-of-thought prompting (Wei et al., 2022), sampling with majority voting (self-consistency; Wang et al., 2022), code execution (Brown et al., 2024; Snell et al., 2024; Damani et al., 2024), and search (Yao et al., 2024).

One scaling strategy that has gained recent attention is **test-time training (TTT)**, in which models are updated through explicit gradient steps based on test-time inputs (Krause et al., 2018; 2019). This method differs from standard fine-tuning as it operates in an *extremely low-data regime*—typically via an unsupervised objective on a single input, or a supervised objective applied to one or two in-context labeled examples. Modern versions of this approach was proposed for vision models by Sun et al. (2020), and also applied to sequence models by Gandelsman et al. (2022). The design space for TTT approaches is large, and there is currently a limited understanding of which design choices are most effective for LMs (and specifically for novel-task learning). In this paper, we systematically study the impact of various TTT design choices, as well as its interaction with pre-training and sampling schemes.

We evaluate these methods in the **Abstraction and Reasoning Corpus** (ARC) (Chollet, 2019), a collection of extremely challenging few-shot visual reasoning problems. ARC is an ideal benchmark for testing the limits of LM generalization as it presents novel tasks, in a novel format, requiring nontrivial search and inference capabilities. Current language models perform poorly on ARC. Most successful approaches have relied on program synthesis techniques (Butt et al., 2024; Ainooson et al., 2023; Huang et al., 2023), though recently Cole et al. (2024) reported promising results using TTT on the benchmark.

We identify several crucial ingredients for effective application of TTT to few-shot learning: (1) **initial fine-tuning** on synthetic tasks similar to those encountered at test time, (2) an augmented, **leave-one-out** task generation strategy for constructing the test-time dataset, (3) **per-instance** adapter training and (4) a **self-consistency** (Wang et al., 2022) approach under invertible transformations. With careful choices of these components, TTT can significantly improve LM performance on ARC—increasing accuracy by up to a factor of six over a 1B model, and achieving state-of-the-art results for published, purely neural models on the ARC task with a 8B model. Indeed, our results show that when equipped with test-time training, ordinary LMs can match or exceed the performance of many neuro-symbolic approaches on ARC.

Our main contributions[1] are:

1. We identify and systematically analyze the key components needed for test-time training on ARC tasks, with a a novel test time training data generation and self-consistency component.
2. We achieve state-of-the-art results among published neural approaches on the ARC validation set:
    - 53% accuracy on the public validation set with a 8B parameter model.
    - 61.9% accuracy when ensembled with program synthesis approaches, matching average human performance on the dataset.
3. We demonstrate that tasks that could only be solved by program synthesis previously can be solved with fully neural approaches equipped with our TTT framework.

These results challenge the assumption that symbolic components are strictly necessary for solving such complex tasks. Instead, they suggest that the critical factor in solving novel reasoning problems may be the allocation of proper computational resources during test time, perhaps independently of whether these resources are deployed through symbolic or neural mechanisms.

2 Preliminaries

In this section, we first formally describe the ARC challenge. Next, we give an overview of in-context learning and test-time training, which form the foundation of our investigation. Finally, we detail our default experimental setup.

[1]Our implementation can be found at this link.

2.1 ARC Challenge

The Abstraction and Reasoning Corpus (ARC) aims to evaluate the abstract reasoning capabilities of language models through their ability to solve visual puzzles. Each puzzle, henceforth referred to as *task*, is comprised of input-output pairs of 2-D grids (up to 30×30 in size) that contain shapes or patterns made with up to 10 different colors, as displayed in Fig. 1(b). The output of each pair is obtained by applying an *intuitive* and *shared* transformation rule or function $y = f(x)$. In practice, these transformations are highly diverse and composite, ranging from simple concepts such as reflection and counting, to more complex ones such as application of gravity and path finding.

Each task in ARC is composed of a training and test split, with:

- Training examples denoted $(x_k^{\text{train}}, y_k^{\text{train}})_{k=1}^{K}$ (typically K ranges from 2 to 7).
- Test examples denoted $(x_m^{\text{test}}, y_m^{\text{test}})_{m=1}^{M}$ (typically M ranges from 1 to 3).

Given the set of training examples, the *goal* is to predict the test output y^{test} for test test input x^{test} by reasoning about the underlying transformation.

We denote a task as $d = (\mathbf{x}^{\text{train}}, \mathbf{y}^{\text{train}}, \mathbf{x}^{\text{test}}, \mathbf{y}^{\text{test}})$ where $d \in \mathcal{D}_{\text{ARC}}$, the collection of such ARC tasks. The original training and validation sets of ARC dataset, respectively $\mathcal{D}_{\text{ARC}}^{\text{train}}$ and $\mathcal{D}_{\text{ARC}}^{\text{val}}$, consists of 400 tasks each. Success criteria requires to produce exact match for all test outputs (if not partial points are given). Please refer to Johnson et al. (2021) for a taxonomy and analysis of these tasks.

Most approaches to ARC can be categorized into two main categories: *program synthesis* and *fully neural*. Program synthesis approaches (Butt et al., 2024; Wang et al., 2024; Li et al., 2024; Greenblatt, 2024) try to first find the transformation function f, and later apply it to the test example. On the other hand, fully neural approaches (Thoms et al., 2023; Bober-Irizar and Banerjee, 2024) try to directly predict the output y^{test}, only implicitly reasoning about the underlying transformation. In this work, we use a fully neural approach, using a LM to predict the test outputs.

We start with an LM pre-trained on text data (without a vision encoder). To provide ARC examples as input to these models, we thus require a formatting function (denoted `str`) that converts 2D grids into their textual representations as shown in Appendix A.3. Previous work has presented examples as lists of numbers (Wang et al., 2024) or color words, or lists of connected components labeled with shapes and locations (Greenblatt, 2024). Given any such string representation of a task, we may present it to an LM and perform predictions with few-short prompting, as explained in the next section.

2.2 In-context Learning

At a certain scale, many LMs exhibit the ability to adapt to new tasks without updating their parameters by simply conditioning on input examples or instructions provided. Given a sequence of input-output pairs $(x_1, y_1), ..., (x_n, y_n)$ and a new input x_{n+1}, a LM can be used to generate the output $\hat{y}_{n+1}$ by sampling from:

$$\hat{y}_{n+1} \sim \text{LM}(\cdot \mid x_1, y_1, \ldots x_n, y_n, x_{n+1}) \tag{1}$$

In-context learning does not resemble any standard machine learning algorithm (Zhao et al., 2024; Min et al., 2022), and it does not work out-of-the box for novel tasks — e.g. small language models (few billion parameters) performs poorly on ARC (Opielka et al., 2024; Bober-Irizar and Banerjee, 2024).

2.3 Test-Time Training

Test-time training (TTT) enables parametric models to adapt during inference through dynamic parameter updates, an approach that remains relatively unexplored in the era of large language models. This technique is a form of transductive learning, where models leverages the test data structure to improve its predictions. The general TTT process works as follows: Starting with initial model parameters $\boldsymbol{\theta}_0$, for each test input (or batch of inputs), we first generate training data $\mathcal{D}_{\text{TTT}}(d_{\text{input}})$ from the test inputs. We then optimize these parameters to minimize a loss function $\mathcal{L}(\mathcal{D}_{\text{TTT}}; \boldsymbol{\theta})$, producing temporarily updated parameters $\boldsymbol{\theta}_d$ for prediction. After generating predictions, the model is restored to the original parameters $\boldsymbol{\theta}_0$ for the next instance or batch. Thus, TTT trains a specialized prediction model for each test input, obtained by fine-tuning a base model on a test-time dataset generated from that test input.

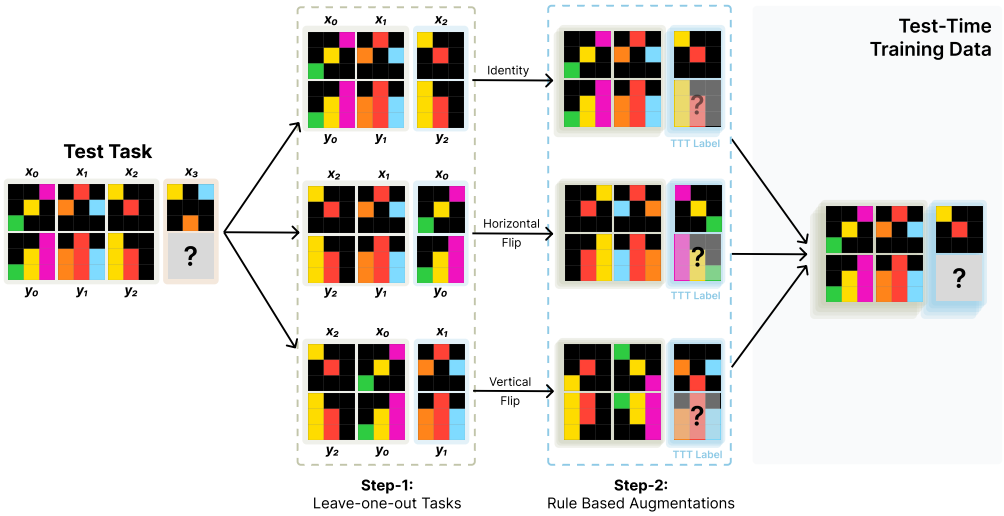

Figure 2: **TTT dataset generation for a test task (Section 3.1):** We start by creating leave-one-out tasks from the given training examples of the task. These tasks are then augmented through rule-based transformations to obtain the full TTT dataset. Finally, we train task-specific LoRA adapters on top of the base FT model.

In past work (e.g. Sun et al., 2020), $\mathcal{D}_{\text{TTT}}$ is typically constructed by applying an unsupervised objective (e.g. masked autoencoding) to the input $\mathbf{x}$ alone. However, the in-context learning setting we consider provides richer context in the form of demonstration pairs $(x_1, y_1), \ldots, (x_K, y_K)$. Here, applying test-time tuning involves first constructing an initial language model LM, mapping each test input x to an input-specific dataset $\mathcal{D}_{\text{TTT}}$, fine-tuning the LM to optimize some loss function $\mathcal{L}$ over the dataset according to: $\sum_{d \in \mathcal{D}_{\text{TTT}}} \mathcal{L}(\text{LM}(d))$, and finally sampling from the updated model to obtain a final prediction. Our experiments in this paper characterize each component of this pipeline, describing:

1. How to construct the augmented TTT dataset $\mathcal{D}_{\text{TTT}}$ from the test input (Section 3).
2. An augmented inference strategy based on self-consistency over transformations (Section 4).
3. A base model with parameters $\boldsymbol{\theta}_0$ that is fine-tuned on a dataset $\mathcal{D}_{\text{FT}}$ of similar tasks (Section 5).

2.4 Experimental Setup

To investigate the impact of each TTT component, we conduct experiments by varying one component while holding the others constant at their optimal values (described in their respective sections). Our default configuration in the experiments uses the following settings:

Model Architecture & Optimization We use an 8B parameter language model from the Llama-3 models, and 1B, 3B from Llama-3.2 models (Dubey et al., 2024). We use Low-Rank Adaptation (LoRA) (Hu et al., 2021) for parameter-efficient test-time training. For each task d, we initialize a separate set of LoRA parameters that are trained on the dataset $\mathcal{D}_{\text{TTT}}$. The LoRA rank is set to 128, and adaptations are applied to MLP, attention, and output layers. We train models with AdamW optimizer (Loshchilov and Hutter, 2019) with 2 epochs with batch sizes of 2.

Data & Formatting For efficient evaluation purposes, we randomly pick 80 balanced ARC tasks from ARC validation set, includes 20 easy, 20 medium, 20 hard, 20 expert tasks according to the classification in LeGris et al. (2024a) (see Appendix A.2 for this task list). We will use this subset of ARC tasks throughout the paper, except our final results given in for the full validation set (Section 6). We limit $\mathcal{D}_{\text{TTT}}$ to have maximum of 250 examples per task for efficiency reasons. With that, the whole TTT and inference process takes approximately 12 hours for 100 randomly sampled validation tasks when using an NVIDIA-A100 GPU. Appendix B.2 provides additional details on the hyper-parameters. Input grids are converted to text using `numpy`'s default array printing format as shown in Fig. 8.

In the following sections, we investigate the key factors that contribute to successful abstract reasoning with language models. Our analysis covers the impact of fine-tuning data $\mathcal{D}_{\text{FT}}$ data, TTT data $\mathcal{D}_{\text{TTT}}$, training objectives, inference procedures, and model size, providing insights into effective strategy for deploying test-time training.

3 What Dataset and Loss During TTT?

3.1 Data Generation

Given a task, we take the set of training input-output pairs $(x_k^{\text{train}}, y_k^{\text{train}})_{k=1}^K$ and turn them into an augmented set of test-time-training tasks $\mathcal{D}_{\text{TTT}}$. We obtain $\mathcal{D}_{\text{TTT}}$ using a two-step process: First, we create a set of leave-one-out in-context learning tasks from the given training input-output pairs. Second, we use invertible rule-based transformations on this set to obtain an augmented dataset. This process is summarized in Fig. 2.

Step 1 - Leave-one-out Tasks: By excluding the j^{th} example pair from the training examples, we can create the following synthetic task:

$$d_j^{\text{ICL}} = (\underbrace{(x_k, y_k)_{k\in\{1,\ldots,K\}\setminus\{j\}}}_{\text{synthetic training examples}}, \underbrace{x_j, \quad y_j}_{\text{synthetic test example}}) \quad \text{where } j \in [1, K] \tag{2}$$

where d_j synthetic training task with the j-th example pair treated as the test case. We can generate n different tasks, each containing $n-1$ example pairs. We further include two randomly permuted version of d_j where we permute the order of the training examples.

Step 2 - Rule-based transformations: Consider an invertible transformation t such that $t^{-1}(t(x)) = x$. For every task obtained in step 1, we can use t to generate a new augmented task $t(d_j^{\text{ICL}})$, where t is applied to each individual grid in the task.

We choose simple transformations that preserve the fundamental relationships while introducing controlled variations such as rotation, flips, color permutation, example permutaton, size scaling, etc. The list and the description of these transformations are provided in Appendix B.1. Finally, we obtain

$$\mathcal{D}_{\text{TTT-ICL}} = \{t(d_j^{\text{ICL}})\} \quad \text{for all } t, j \text{ pairs.} \tag{3}$$

Baseline: End-to-End Learning Tasks For comparison to the "test-time in-context learning" approach described above, we also evaluate an "test-time end-to-end learning" approach. We create a supervised dataset directly from the example demonstrations by treating each input-output pair as an independent training instance. Unlike the in-context learning setup, no context is used for prediction:

$$d_j^{\text{E2E}} = (x_j, y_j) \quad \text{where } j \in [1, K] \tag{4}$$

Note that this would be equivalent to leave-$(n-1)$-out task set in ICL setting as no training examples are provided as context. Similar to ICL case, we can apply rule-based transformations to augment the dataset:

$$\mathcal{D}_{\text{TTT-E2E}} = \{t(d_j^{\text{E2E}})\} \quad \text{for all } t, j \text{ pairs.} \tag{5}$$

This approach is computationally more efficient as it directly learns the input-output mapping without the overhead of managing demonstration context i.e. the few-shot prompt.

3.2 Optimization Objective

During test-time training, we optimize a set of task-specific parameters using LoRA (low-rank adaptation; Hu et al. (2021)) while keeping most of the base model frozen. This approach allows computationally efficient adaptation while maintaining the model's general capabilities.

Training Objective: Given a task's test-time training dataset $\mathcal{D}_{\text{TTT}}^i$, we minimize the standard language modeling loss on both the demonstrations and test outputs:

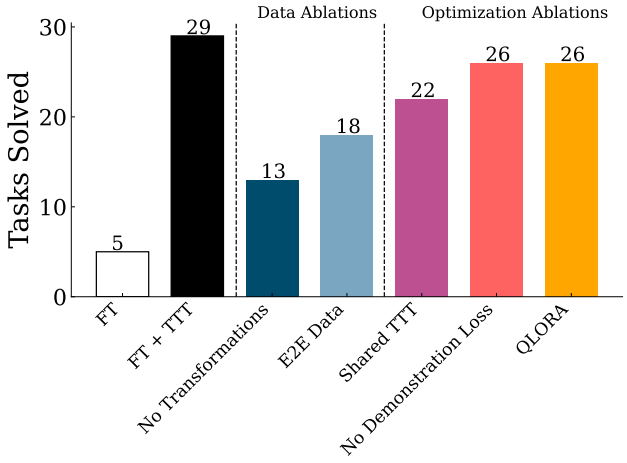

Figure 3: **Accuracy of different data and optimization ablations in TTT:** Our data ablation studies reveal that the ICL data format is crucial for effective TTT, and that applying transformations to augment the TTT dataset notably enhances performance. In optimization ablations, learning task-specific adapters significantly outperforms using a single adapter. Additionally, taking a loss on the in-context demonstrations provides a minor performance boost, while using quantized LoRA results in only a slight performance decrease drop. Full discussion in Section 3.3.

$$\mathcal{L}_i(\mathcal{D}^i_{\text{TTT}}; \boldsymbol{\theta}_{\text{i}}) = \sum_{d \in \mathcal{D}^i_{\text{TTT}}} \left(\sum_{n=2}^{K} \mathcal{L}_{\text{LM}}(y_n | x_1, y_1, ..., x_n; \boldsymbol{\theta}_{\text{i}}) + \mathcal{L}_{\text{LM}}(y_{\text{test}} | x_1, y_1, ..., x_K, y_K, x_{\text{test}}; \boldsymbol{\theta}_{\text{i}}) \right) \tag{6}$$

where $\mathcal{L}_{\text{LM}}$ is the standard cross-entropy loss for language modeling. Note that we include loss terms for demonstrations starting from the second example ($n = 2$). By doing so, we encourage the model to start reasoning about the transformation pattern from the second demonstration pair itself.

Task-Specific Parameters: Instead of learning a single LoRA adapter for all tasks in the test set, we learn an individual task-specific LoRA adapter for each task. That is, we obtain N different LoRA adapters, where N is the number of test tasks.

3.3 Results

We compare the main implementation of our method to the following ablations:

1. **FT (No TTT):** The vanilla baseline where TTT is ablated and the fine-tuned model is used instead.
2. **No Transformations:** No transformation-based data augmentation. That is, data from step 2 of the data generation pipeline described in Section 3.1 is not included in the test-time training dataset.
3. **End-to-End (E2E) Data:** Instead of the standard in-context task setup, we use the end-to-end task formulation, as described in Section 3.1.
4. **Shared TTT:** In contrast to learning a task-specific LoRA adapter, a single LoRA adapter is learned using an aggregated dataset of all tasks.
5. **No Demonstration Loss:** No loss is taken on the demonstrations in the training outputs of the data. That is, the TTT loss is simply:

$$\mathcal{L}_i(\mathcal{D}^i_{\text{TTT}}; \boldsymbol{\theta}_{\text{i}}) = \sum_{d \in \mathcal{D}^i_{\text{TTT}}} \left(\mathcal{L}_{\text{LM}} \left(y_{\text{test}} | x_1, y_1, ..., x_K, y_K, x_{\text{test}}; \boldsymbol{\theta}_{\text{i}} \right) \right) \tag{7}$$

6. **QLoRA:** Rather than full-precision base model updates, quantized LoRA adapters (Dettmers et al., 2024) are learned for each task, which is the alternative for LoRA considered for memory efficiency.

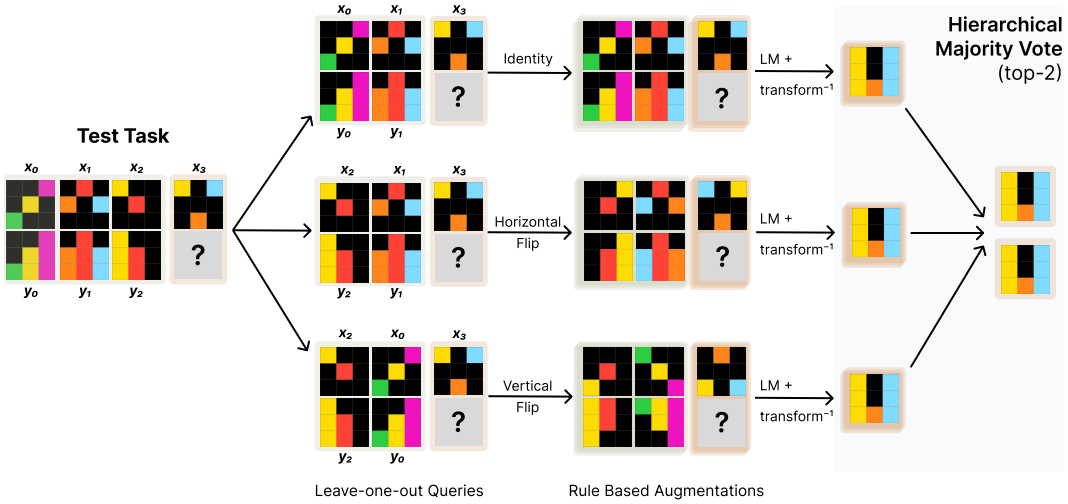

Figure 4: **Augmented inference and hierarchical voting (Section 4):** We use leave-one-out tasks and invertible geometric transformations to obtain multiple equivalent versions of the task for augmented inference. Predictions from these versions are aggregated with a hierarchical voting strategy: first, voting is performed within each transformation, and then the top candidates from each transformation undergo global voting to yield the top two predictions.

Results are presented in Fig. 3. Our TTT method is effective, improving fine-tuned model accuracy approximately **6**× (**5** → **29**). The structure of the auxilary task significantly impact TTT effectiveness. Using in-context learning tasks substantially outperforms using end-to-end tasks, showing a **11** (**38**% **decrease**) tasks relative performance drop under identical conditions. This may be simply due to training less parameters. Dropping transformations applied to augment data hurts by **16 tasks** (**55**% **decrease**).

Next, we ablate multiple components of TTT optimization to analyze their contribution to the performance. Learning a single LoRA adapter across all tasks reduces performance on **7 tasks (24**% **decrease)**. This is expected as learning a dedicated adapter allows more parameters to train per task. Second, the decision that we made by taking loss on the output demonstrations marginally improves the performance (26 → 29), as we believe that this forces the model to reason about the transformation while processing the demonstrations. Finally, we observe that using *quantized* LoRA (QLoRA) only leads to a marginal drop in performance (29 → 26) — in memory-bottlenecked scenarios using QLoRA may be viable.

4 What Inference Strategy After TTT?

4.1 Augmented Inference

Recent work has shown that scaling test-time compute can significantly improve the performance of LMs. One of the most common techniques to do this is by sampling multiple responses, and then selecting the best response using a ranker. However, while sampling is very effective in domains with multiple possible solutions (programs in code) or multiple possible paths to the final answer (math), it can be detrimental when generating answers directly, as there is no way to directly enforce diversity *across* samples while ensuring coherence *within* samples. As an alternative inference-time scaling, we use an *augmented inference* strategy that generates multiple prediction candidates by using geometric transformations, combined with a greedy decoding scheme.

For a given task with training examples $(x_k, y_k)_{k=1}^{K}$ and test input x_{test}, we use invertible geometric transformations to produce equivalent transformed versions of the task, as shown in Fig. 3. Let $\mathcal{T}$ be some set set of invertible geometric transformations (e.g., rotations and reflections). For each transformation $t \in \mathcal{T}$, we apply t to all training demonstrations and the test input and run our model with these transformed inputs.

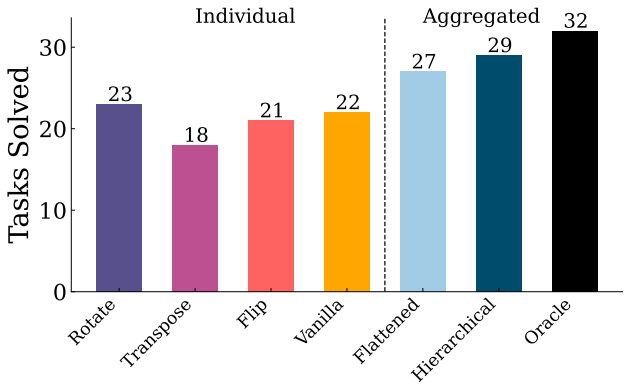

Figure 5: **Accuracy of different invertible transformations and voting schema:** Our analysis shows that while individual transformations generally perform at a modest level and are comparable to one another, aggregating across them through voting yields substantial improvements. Notably, a hierarchical voting strategy with two voting stages surpasses a flat voting approach. Our hierarchical method approaches oracle-level performance, demonstrating its effectiveness in accurately selecting the correct answer when present. Full discussion in Section 4.3.

We then apply the inverse transformation to obtain the final prediction for that transformation.

$$\tilde{y} \sim \text{LM}(t(\mathbf{d}_{\text{input}})) := [t(x_1), t(y_1), \ldots, t(x_{\text{test}})] \tag{8}$$

$$y_t = t^{-1}(\tilde{y}) \tag{9}$$

We further augment our predictions by permuting the order of training examples. For each transformation g, we sample $n = 2$ different permutations of the demonstration sequence, resulting in $n \cdot |\mathcal{T}|$ total predictions per task. This is to mitigate any bias in the model's processing of the demonstration sequence. Bober-Irizar and Banerjee (2024) also find transpose and rotation is helpful to produce extra prediction candidates.

4.2 Ensembling Predictions (Voting Strategy)

We employ a hierarchical voting strategy to determine the final prediction from the set of candidates $\{y\}_{i=1}^{n\cdot|\mathcal{T}|}$. This approach involves two stages of voting to progressively narrow down the best candidates: first, by selecting the most frequent predictions within each transformation, and then by conducting an overall vote across transformation-specific candidates to identify the top-2 most frequent predictions. The details of each stage are as follows:

1. **Intra Transformation Voting:** We group predictions by their corresponding transformation t and select the top-3 most frequent predictions within each group. If fewer than 3 unique predictions exist within a group, we supplement the candidates by computing additional predictions through:
   - **Row-based majority**: For each row in the predicted output grid, we take the most frequent row values across all predictions in the transformation group.
   - **Column-based majority**: Similarly, for each column in the predicted output grid, we take the most frequent column values across all predictions in the transformation group.
2. **Global Voting:** Using the selected transformation-specific candidates obtained from (1), we conduct an overall vote to select the top-2 most frequent predictions for submission. In case of a tie, predictions with the identity transformation are given priority.

4.3 Results

To analyze the impact of augmented inference and voting, we run the following ablations:

1. **Vanilla:** This baseline follows a standard inference approach without any augmented inference or voting. It generates 2 predictions from the model for 2 permutations of the task. This setup serves as a reference point to assess the benefits of our augmented inference and voting strategy.

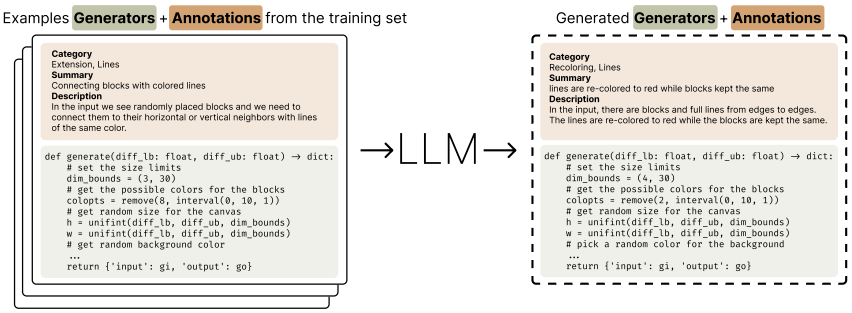

Figure 6: **LLM based synthetic tasks generation:** Given some seed task descriptions and task generator functions in Python, we generate more generator functions to produce novel tasks. We use three different approaches: (1) few-shot prompting with only generators, (2) few-shot prompting with generators and task descriptions, (3) two-stage approach: first generate free form descriptions, then condition on them to generate more generators (shown in Fig. 9).

2. **Transformed Inference (Rotate/Transpose/Flip)**: Measures performance when predictions are generated solely from a specific transformed version of the task shown in Fig. 5. This assesses the individual effectiveness of each transformation applied in isolation. Note that **Vanilla** can also be considered a part of this category, with the transformation being the *identity* function .
3. **Hierarchical Voting:** Our full pipeline, which includes both augmented inference and voting.
4. **Flattened Voting:** Instead of using a hierarchical voting strategy, we perform a single voting round on the full set of $n \cdot |\mathcal{T}|$ predictions to identify the 2 most frequent predictions.
5. **Oracle:** The oracle selects the correct answer if it exists in the set of $n \cdot |\mathcal{T}|$ predictions. The oracle provides an upper-bound on the best performance possible if the voting procedure was perfect.

The results are summarized in Figure 5. As shown in the figure, the individual performance of specific transformed versions is generally poor, with the transpose transformation yielding the worst accuracy. However, aggregating across these transformations through voting procedures leads to significant improvements. This suggests that some tasks may be easier to solve in their transformed versions, and that using self-consistency (voting) for aggregation is generally beneficial, a finding also observed in prior work. Additionally, while the flattened voting procedure improves accuracy, our hierarchical voting procedure outperforms it. In fact, our hierarchical procedure is comparable to the oracle, indicating that hierarchical aggregation effectively selects the correct answer (when it exists) with high accuracy.

5 What Fine-Tuning Before TTT?

While test-time training facilitates task-specific adaptation, the base model's capabilities impacts the final performance. We developed several approaches for generating synthetic training data to enhance the base model's abstract reasoning capabilities through fine-tuning, exploring both automated and semi-automated methods for task generation. In this section, we detail our fine-tuning data generation strategies and analyze the impact of different data sources and model sizes on final performance.

5.1 Preparing Fine-tuning Data

Hodel (2024) provides domain-specific language (DSL), REARC, as well as the transformation f_i that solves the task-i, and the data generation function g_i that are implemented in this DSL for each training task in the $\mathcal{D}_{\text{ARC}}^{\text{train}}$ dataset. These functions enable sampling of new input-output pairs that maintains the same underlying transformation principle:

$$d = (x, y) \sim \text{eval}(g_i) \tag{10}$$

where d represents a newly generated input-output pair that can be solved using the same transformation function f_i as the original task-i[2].

[2]We can verify the generated examples by asserting $f_i(x) = y$.

(a) Using Existing Generators The generator functions gs in REARC already provide an effective data augmentation tool by producing different instantiations of same tasks. We generate extra samples from these training tasks by running these codes many times and randomly splitting these new examples ($d \sim \text{eval}(g_i)$) to set of train and test examples. These augmented examples are already provided with their DSL release.

(b) Few-shot Prompting an LLM Additionally, we used several approaches to generate *novel* tasks using an LM (in our case an ensemble of GPT4 and GPT4-o).

The simplest approach generates new task generators using few-shot examples:

$$g' \sim \text{LM}(g_1, g_2, \ldots, g_m) \tag{11}$$

where g' is a new generator function and $g_1, \ldots, g_m$ are existing generator functions (shown in Fig. 6)s. We sample different m examples by uniformly from existing training set. We repeat this process multiple times to get a good amount of tasks.

We augment the generator functions with task descriptions and jointly generate both descriptions and generators:

$$(s', g') \sim \text{LM}(s_1, g_1, s_2, g_2, \ldots s_m, g_m) \tag{12}$$

where s_i represents the description of task i.

To get the task descriptions, we manually created seed descriptions for 10 training tasks. These seed descriptions were then used to generate descriptions for the training and validation tasks through few-shot prompting. To increase diversity of tasks we use task descriptions with hierarchical fields (category, summary, and description). The process of getting these descriptions provided in the Appendix D.1.

Instead of jointly generating task descriptions and function generations, we additionally deployed a two-stage approach described as following:

$$s' \sim \text{LM}(s_1, s_2, \ldots s_m) \tag{13}$$

$$g' \sim \text{LM}(s_1, g_1, s_2, g_2, \ldots, s_m, g_m, s') \tag{14}$$

This approach first generates a task description s' and then conditions the generator creation on both existing task pairs and the new description. In total we collected 6426 generators with these LLM based approaches. We provide qualitative samples from these LM generated tasks in Fig. 11

(c) Geometric Transformations Finally, our synthetic tasks are enhanced through various geometric transformations, such as basic transformations (rotations, reflections, random shift and size scaling), pattern operations (random patching, tiling, and repetition), color permutations, and composite transformations involving sequential application of multiple basic transformations. These transformations are applied in three ways:

- Input grids only: $(x, y) \to (t(x), y)$
- Output grids only: $(x, y) \to (x, t(y))$
- Both input and output: $(x, y) \to (t(x), t(y))$

The complete specification of transformations and their application details are provided in Appendix B.1. These transformations are applied randomly to variants of tasks with 30% of the time.

5.2 Results

We perform full fine-tuning 1B, 3B Llama 3.2 instruction-tuned, and 8B Llama 3 instruction-tuned using augmented data. The format and training objective is same as the ones described for TTT in Section 2.4. Hyper-parameter details are given in Appendix B.2. We do the following ablations for augmented data:

1. **No FT:** The original Llama 3 instruction-tuned model without any fine-tuning.
2. **All:** We use all methods described in Section 5.1, including REARC, rule-based augmentation, and LM generation.
3. **No-Geom:** We remove geometric transformations from all tasks.
4. **No-LM:** We only use REARC and rule-based augmentation, excluding tasks generated by the LM.

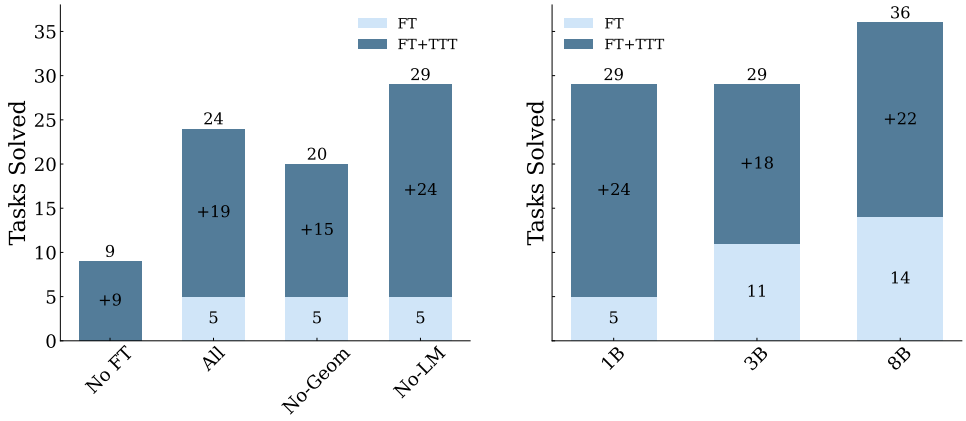

Figure 7: **Left: Accuracy when fine-tuning with different data sources.** While all fine-tuned models perform similarly, their performance after TTT shows considerable variance. As expected, removing geometric transformations from the fine-tuning data reduces performance compared to the model trained on the full dataset. Surprisingly, excluding LM-generated data from fine-tuning actually outperforms the model trained on all data. **Right: Performance results across different model sizes.** As expected, performance of the base fine-tuned model improves with increasing model size, aligning with current scaling law trends. However, the scaling behavior after TTT is less clear. For instance, the final performance of the 1B and 3B models is identical after TTT. Full discussion in Section 5.2.

How does FT data affect TTT? We compare models using different fine-tuning data in Fig. 7. We find that the model trained on REARC with rule-based augmentation achieves the strongest performance. Surprisingly, including **LM-generated tasks hurts performance by 5%**, indicating that current LM-based task generation methods may need more sophisticated filtering mechanisms as used in Li et al. (2024) (see their results in Section 6). Finally, we find that FT performance shows little correlation with TTT performance.

Model Size and Scaling in TTT We show results using different model sizes in Fig. 7. Increasing the model size consistently improves FT performance, with the 8B model achieving the highest accuracy of 36%. We also observe that TTT effectively closes the performance gap for smaller models, with the 1B and 3B models achieving similar accuracy after TTT.

6 ARC Benchmark and Comparison to Other Systems

Following our development experiments on 80 tasks, we present comprehensive results on the full ARC public evaluation set, comparing our system against existing approaches. Our analysis focuses on three key aspects: the impact of our TTT methodology, the benefits of combining our approach with existing methods and the differences between fully neural and program synthesis methods.

Impact of Test Time Training We applied to our TTT and inference procedure (explained in Section 3 and Section 4) to our base fine-tuned models (fine-tuned 8B model without any LM data in Section 5). TTT improves accuracy from 39.3% to 47.1%, surpassing existing end-to-end neural model results.

Integration with Existing Methods A concurrent work by Li et al. (2024) introduced BARC, achieving 54.4%accuracy by combining neural and program synthesis approaches—previously the highest publicly available result. While their fully neural approach shares similarities with our system, our TTT and inference pipeline has several additional components that boost performance. In particular, our test-time-training includes per-task LoRA and a larger set of augmentations, while our prediction pipeline includes an augmented inference under invertible transformations and a hierarchical self-consistency voting scheme. To validate our improvements, we applied our TTT pipeline to BARC's fully neural model, achieving 53% accuracy—a 35% improvement over their original TTT method.

Table 1: **Scores of different systems on the ARC validation set:** Our TTT pipeline improves base models consistently. We achieve 47.1% accuracy when applied to our fine-tuned model, 53% when applied to BARC model from Li et al. (2024), achieving state-of-the-art on pure LM based approaches. We ensemble our method with program synthesis based models, where we achieve (61.9%) state-of-the-art performance comparable to average human performance (60.2%).

Program Synthesizer	Fine-tuned LM	TTT Method	Score (pass@2)
X	Ours	X	18.3%
X	Ours	Ours	47.1%
X	BARC	Ours	53.0%
BARC	Ours	Ours	58.5%
BARC	BARC	Ours	**61.9%**
		Avg. Human	60.2%
		Best Human	**97.8%**
		BARC (ensemble)	54.4%
		BARC (no synthesizer)	39.3%
		Claude - Few-shot prompting	21.0%
		GPT-4.0 - Few-shot prompting	9.0%

Building on these results, we explored various combinations of our approach with BARC's components:

- Combining our TTT pipeline and neural model with BARC's synthesizer raised accuracy to 58.5%.
- Combining our TTT pipeline with BARC's neural model and synthesizer raised accuracy to 61.9%.

This final configuration establishes a new state-of-the-art on the ARC public evaluation set, comparable to the average human performance of 60.2%. While this represents significant progress, there remains a substantial gap to the best human performance of 97.8%, indicating room for further improvements.

Comparing Program Generation and End-to-End Modeling Li et al. (2024) found that program synthesis and fully neural predictors for ARC are highly complementary, even when trained on the same tasks. Their end-to-end neural model can only solve 42.2% of the tasks solved by the program synthesis model. However, we find that when equipped with our TTT pipeline, BARC's fine-tuned fully neural model solves 73.5% of the tasks that are solved by the program synthesis model. This suggests that our TTT pipeline significantly improves the neural model's ability to learn systematic reasoning patterns similar to those captured by program synthesis models.

7 Conclusion

In this work, we conduct an investigation of test-time training and demonstrate that it can significantly improve LM performance on the popular ARC dataset. We find that learning task-specific LoRA adapters and generating augmented test-time datasets using geometric transformations are crucial. We also develop an augmented inference pipeline that uses invertible transformations to generate multiple predictions and then uses self-consistency to select the best candidates. Our overall pipeline applies multiple test-time computation methods, with each component contributing positively. This suggests that not only can test-time compute improve LM performance, but different test-time methods can also complement one another. Our TTT pipeline, combined with an existing method (BARC), achieves state-of-the-art results on the ARC public set and performs comparably to an average human. Our findings suggest that test-time methods could play a pivotal role in advancing the next generation of LMs.

Limitations

Evaluation Framework The ARC challenge maintains separate public and private leaderboards, with the private evaluation conducted on 100 novel tasks. While our TTT pipeline demonstrates promising results on the public benchmark, hardware constraints (12-hour runtime on an A100 GPU for Llama 8B) currently

precludes submission to the official leaderboard, which requires completion within 12 hours on P100 or 2xT4 NVIDIA GPUs. Our development process utilized 80 tasks for validation, and while our method is designed to be task-agnostic, we acknowledge potential sources of optimization bias. The fixed rule-based augmentations detailed in Appendix B.1 were selected during the TTT phase. Standard hyper-parameters (learning rate, batch size, epochs) were optimized using our development set with 80 validation tasks.

Experimental Reproducibility Given the computational requirements of our experiments, this preprint reports results without comprehensive standard error analysis. Our preliminary observations indicate minimal variance across runs, and we plan to include detailed statistical analysis in the final version.

Data Leakage Even though the base Llama-3 perform extremely poorly on the public validation set, the public availability of the dataset on various platforms (GitHub, Kaggle) introduces the possibility that these models may have encountered these examples during pre-training.

Acknowledgments

We thank Aniruddha Nrusimha for helpful discussions on parameter efficient training, and Jyo Pari for feedback on early drafts of this paper.

References

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners, 2020. URL https://arxiv.org/abs/2005.14165.

Eric Todd, Millicent L. Li, Arnab Sen Sharma, Aaron Mueller, Byron C. Wallace, and David Bau. Function vectors in large language models. In *Proceedings of the 2024 International Conference on Learning Representations*, 2024.

François Chollet. On the measure of intelligence, 2019.

Zhaofeng Wu, Linlu Qiu, Alexis Ross, Ekin Akyürek, Boyuan Chen, Bailin Wang, Najoung Kim, Jacob Andreas, and Yoon Kim. Reasoning or reciting? exploring the capabilities and limitations of language models through counterfactual tasks. *arXiv preprint arXiv:2307.02477*, 2023.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems*, 35:24824–24837, 2022.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. *arXiv preprint arXiv:2203.11171*, 2022.

Bradley Brown, Jordan Juravsky, Ryan Ehrlich, Ronald Clark, Quoc V Le, Christopher Ré, and Azalia Mirhoseini. Large language monkeys: Scaling inference compute with repeated sampling. *arXiv preprint arXiv:2407.21787*, 2024.

Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. Scaling llm test-time compute optimally can be more effective than scaling model parameters. *arXiv preprint arXiv:2408.03314*, 2024.

Mehul Damani, Idan Shenfeld, Andi Peng, Andreea Bobu, and Jacob Andreas. Learning how hard to think: Input-adaptive allocation of lm computation. *arXiv preprint arXiv:2410.04707*, 2024.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. *Advances in Neural Information Processing Systems*, 36, 2024.

Ben Krause, Emmanuel Kahembwe, Iain Murray, and Steve Renals. Dynamic evaluation of neural sequence models. In *International Conference on Machine Learning*, pages 2766–2775. PMLR, 2018.

Ben Krause, Emmanuel Kahembwe, Iain Murray, and Steve Renals. Dynamic evaluation of transformer language models, 2019.

Yu Sun, Xiaolong Wang, Zhuang Liu, John Miller, Alexei Efros, and Moritz Hardt. Test-time training with self-supervision for generalization under distribution shifts. In *International conference on machine learning*, pages 9229–9248. PMLR, 2020.

Yossi Gandelsman, Yu Sun, Xinlei Chen, and Alexei Efros. Test-time training with masked autoencoders. *Advances in Neural Information Processing Systems*, 35:29374–29385, 2022.

Natasha Butt, Blazej Manczak, Auke Wiggers, Corrado Rainone, David W Zhang, Michaël Defferrard, and Taco Cohen. Codeit: Self-improving language models with prioritized hindsight replay. In *International Conference on Machine Learning*, 2024.

James Ainooson, Deepayan Sanyal, Joel P. Michelson, Yuan Yang, and Maithilee Kunda. A neurodiversity-inspired solver for the abstraction & reasoning corpus (arc) using visual imagery and program synthesis, 2023.

Di Huang, Ziyuan Nan, Xing Hu, Pengwei Jin, Shaohui Peng, Yuanbo Wen, Rui Zhang, Zidong Du, Qi Guo, Yewen Pu, and Yunji Chen. Anpl: Towards natural programming with interactive decomposition, 2023.

Jack Cole, Mohamed Osman, Michael Hodel, Keith Duggar, and Tim Scarfe. Machine learning street talk, June 2024.

Aysja Johnson, Wai Keen Vong, Brenden M Lake, and Todd M Gureckis. Fast and flexible: Human program induction in abstract reasoning tasks. *arXiv preprint arXiv:2103.05823*, 2021.

Ruocheng Wang, Eric Zelikman, Gabriel Poesia, Yewen Pu, Nick Haber, and Noah D Goodman. Hypothesis search: Inductive reasoning with language models. *ICLR*, 2024.

Wen-Ding Li, Keya Hu, Carter Larsen, Yuqing Wu, Simon Alford, Caleb Woo, Spencer M. Dunn, Hao Tang, Michelangelo Naim, Dat Nguyen, Wei-Long Zheng, Zenna Tavares, Yewen Pu, and Kevin Ellis. Combining induction and transduction for abstract reasoning, 2024. URL https://arxiv.org/abs/2411.02272.

Ryan Greenblatt. Getting 50% (sota) on arc-agi with gpt-4o, 2024. URL https://redwoodresearch.substack.com/p/getting-50-sota-on-arc-agi-with-gpt. [Accessed 09-11-2024].

Luca H. Thoms, Karel A. Veldkamp, Hannes Rosenbusch, and Claire E. Stevenson. Solving arc visual analogies with neural embeddings and vector arithmetic: A generalized method. *ArXiv*, abs/2311.08083, 2023. URL https://api.semanticscholar.org/CorpusID:265158110.

Mikel Bober-Irizar and Soumya Banerjee. Neural networks for abstraction and reasoning: Towards broad generalization in machines. *arXiv preprint arXiv:2402.03507*, 2024.

Siyan Zhao, Tung Nguyen, and Aditya Grover. Probing the decision boundaries of in-context learning in large language models. *arXiv preprint arXiv:2406.11233*, 2024.

Sewon Min, Xinxi Lyu, Ari Holtzman, Mikel Artetxe, Mike Lewis, Hannaneh Hajishirzi, and Luke Zettlemoyer. Rethinking the role of demonstrations: What makes in-context learning work? *arXiv preprint arXiv:2202.12837*, 2022.

Gustaw Opielka, Hannes Rosenbusch, Veerle Vijverberg, and Claire E. Stevenson. Do large language models solve arc visual analogies like people do?, 2024. URL https://arxiv.org/abs/2403.09734.

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, Anirudh Goyal, Anthony Hartshorn, Aobo Yang, Archi Mitra, Archie Sravankumar, Artem Korenev, Arthur Hinsvark, Arun Rao, Aston Zhang, Aurelien Rodriguez, Austen Gregerson, Ava Spataru, Baptiste Roziere, Bethany Biron, Binh Tang, Bobbie Chern, Charlotte Caucheteux, Chaya Nayak, Chloe Bi, Chris Marra, Chris McConnell, Christian Keller, Christophe Touret, Chunyang Wu, Corinne Wong, Cristian Canton Ferrer, Cyrus Nikolaidis, Damien Allonsius, Daniel

Song, Danielle Pintz, Danny Livshits, David Esiobu, Dhruv Choudhary, Dhruv Mahajan, Diego Garcia-Olano, Diego Perino, Dieuwke Hupkes, Egor Lakomkin, Ehab AlBadawy, Elina Lobanova, Emily Dinan, Eric Michael Smith, Filip Radenovic, Frank Zhang, Gabriel Synnaeve, Gabrielle Lee, Georgia Lewis Anderson, Graeme Nail, Gregoire Mialon, Guan Pang, Guillem Cucurell, Hailey Nguyen, Hannah Korevaar, Hu Xu, Hugo Touvron, Iliyan Zarov, Imanol Arrieta Ibarra, Isabel Kloumann, Ishan Misra, Ivan Evtimov, Jade Copet, Jaewon Lee, Jan Geffert, Jana Vranes, Jason Park, Jay Mahadeokar, Jeet Shah, Jelmer van der Linde, Jennifer Billock, Jenny Hong, Jenya Lee, Jeremy Fu, Jianfeng Chi, Jianyu Huang, Jiawen Liu, Jie Wang, Jiecao Yu, Joanna Bitton, Joe Spisak, Jongsoo Park, Joseph Rocca, Joshua Johnstun, Joshua Saxe, Junteng Jia, Kalyan Vasuden Alwala, Kartikeya Upasani, Kate Plawiak, Ke Li, Kenneth Heafield, Kevin Stone, Khalid El-Arini, Krithika Iyer, Kshitiz Malik, Kuenley Chiu, Kunal Bhalla, Lauren Rantala-Yeary, Laurens van der Maaten, Lawrence Chen, Liang Tan, Liz Jenkins, Louis Martin, Lovish Madaan, Lubo Malo, Lukas Blecher, Lukas Landzaat, Luke de Oliveira, Madeline Muzzi, Mahesh Pasupuleti, Mannat Singh, Manohar Paluri, Marcin Kardas, Mathew Oldham, Mathieu Rita, Maya Pavlova, Melanie Kambadur, Mike Lewis, Min Si, Mitesh Kumar Singh, Mona Hassan, Naman Goyal, Narjes Torabi, Nikolay Bashlykov, Nikolay Bogoychev, Niladri Chatterji, Olivier Duchenne, Onur Çelebi, Patrick Alrassy, Pengchuan Zhang, Pengwei Li, Petar Vasic, Peter Weng, Prajjwal Bhargava, Pratik Dubal, Praveen Krishnan, Punit Singh Koura, Puxin Xu, Qing He, Qingxiao Dong, Ragavan Srinivasan, Raj Ganapathy, Ramon Calderer, Ricardo Silveira Cabral, Robert Stojnic, Roberta Raileanu, Rohit Girdhar, Rohit Patel, Romain Sauvestre, Ronnie Polidoro, Roshan Sumbaly, Ross Taylor, Ruan Silva, Rui Hou, Rui Wang, Saghar Hosseini, Sahana Chennabasappa, Sanjay Singh, Sean Bell, Seohyun Sonia Kim, Sergey Edunov, Shaoliang Nie, Sharan Narang, Sharath Raparthy, Sheng Shen, Shengye Wan, Shruti Bhosale, Shun Zhang, Simon Vandenhende, Soumya Batra, Spencer Whitman, Sten Sootla, Stephane Collot, Suchin Gururangan, Sydney Borodinsky, Tamar Herman, Tara Fowler, Tarek Sheasha, Thomas Georgiou, Thomas Scialom, Tobias Speckbacher, Todor Mihaylov, Tong Xiao, Ujjwal Karn, Vedanuj Goswami, Vibhor Gupta, Vignesh Ramanathan, Viktor Kerkez, Vincent Gonguet, Virginie Do, Vish Vogeti, Vladan Petrovic, Weiwei Chu, Wenhan Xiong, Wenyin Fu, Whitney Meers, Xavier Martinet, Xiaodong Wang, Xiaoqing Ellen Tan, Xinfeng Xie, Xuchao Jia, Xuewei Wang, Yaelle Goldschlag, Yashesh Gaur, Yasmine Babaei, Yi Wen, Yiwen Song, Yuchen Zhang, Yue Li, Yuning Mao, Zacharie Delpierre Coudert, Zheng Yan, Zhengxing Chen, Zoe Papakipos, Aaditya Singh, Aaron Grattafiori, Abha Jain, Adam Kelsey, Adam Shajnfeld, Adithya Gangidi, Adolfo Victoria, Ahuva Goldstand, Ajay Menon, Ajay Sharma, Alex Boesenberg, Alex Vaughan, Alexei Baevski, Allie Feinstein, Amanda Kallet, Amit Sangani, Anam Yunus, Andrei Lupu, Andres Alvarado, Andrew Caples, Andrew Gu, Andrew Ho, Andrew Poulton, Andrew Ryan, Ankit Ramchandani, Annie Franco, Aparajita Saraf, Arkabandhu Chowdhury, Ashley Gabriel, Ashwin Bharambe, Assaf Eisenman, Azadeh Yazdan, Beau James, Ben Maurer, Benjamin Leonhardi, Bernie Huang, Beth Loyd, Beto De Paola, Bhargavi Paranjape, Bing Liu, Bo Wu, Boyu Ni, Braden Hancock, Bram Wasti, Brandon Spence, Brani Stojkovic, Brian Gamido, Britt Montalvo, Carl Parker, Carly Burton, Catalina Mejia, Changhan Wang, Changkyu Kim, Chao Zhou, Chester Hu, Ching-Hsiang Chu, Chris Cai, Chris Tindal, Christoph Feichtenhofer, Damon Civin, Dana Beaty, Daniel Kreymer, Daniel Li, Danny Wyatt, David Adkins, David Xu, Davide Testuggine, Delia David, Devi Parikh, Diana Liskovich, Didem Foss, Dingkang Wang, Duc Le, Dustin Holland, Edward Dowling, Eissa Jamil, Elaine Montgomery, Eleonora Presani, Emily Hahn, Emily Wood, Erik Brinkman, Esteban Arcaute, Evan Dunbar, Evan Smothers, Fei Sun, Felix Kreuk, Feng Tian, Firat Ozgenel, Francesco Caggioni, Francisco Guzmán, Frank Kanayet, Frank Seide, Gabriela Medina Florez, Gabriella Schwarz, Gada Badeer, Georgia Swee, Gil Halpern, Govind Thattai, Grant Herman, Grigory Sizov, Guangyi, Zhang, Guna Lakshminarayanan, Hamid Shojanazeri, Han Zou, Hannah Wang, Hanwen Zha, Haroun Habeeb, Harrison Rudolph, Helen Suk, Henry Aspegren, Hunter Goldman, Ibrahim Damlaj, Igor Molybog, Igor Tufanov, Irina-Elena Veliche, Itai Gat, Jake Weissman, James Geboski, James Kohli, Japhet Asher, Jean-Baptiste Gaya, Jeff Marcus, Jeff Tang, Jennifer Chan, Jenny Zhen, Jeremy Reizenstein, Jeremy Teboul, Jessica Zhong, Jian Jin, Jingyi Yang, Joe Cummings, Jon Carvill, Jon Shepard, Jonathan McPhie, Jonathan Torres, Josh Ginsburg, Junjie Wang, Kai Wu, Kam Hou U, Karan Saxena, Karthik Prasad, Kartikay Khandelwal, Katayoun Zand, Kathy Matosich, Kaushik Veeraraghavan, Kelly Michelena, Keqian Li, Kun Huang, Kunal Chawla, Kushal Lakhotia, Kyle Huang, Lailin Chen, Lakshya Garg, Lavender A, Leandro Silva, Lee Bell, Lei Zhang, Liangpeng Guo, Licheng Yu, Liron Moshkovich, Luca Wehrstedt, Madian Khabsa, Manav Avalani, Manish Bhatt, Maria Tsimpoukelli, Martynas Mankus, Matan Hasson, Matthew Lennie, Matthias Reso, Maxim Groshev, Maxim Naumov, Maya Lathi, Meghan Keneally, Michael L. Seltzer, Michal Valko, Michelle Restrepo, Mihir Patel, Mik Vyatskov, Mikayel Samvelyan, Mike Clark, Mike Macey, Mike Wang, Miquel Jubert Hermoso, Mo Metanat, Mohammad Rastegari, Munish Bansal, Nandhini Santhanam, Natascha Parks, Natasha White, Navyata Bawa, Nayan Singhal, Nick Egebo, Nicolas Usunier, Nikolay Pavlovich Laptev, Ning Dong, Ning Zhang, Norman Cheng, Oleg Chernoguz, Olivia Hart, Omkar Salpekar, Ozlem Kalinli, Parkin Kent, Parth Parekh, Paul Saab, Pavan Balaji, Pedro Rittner, Philip Bontrager, Pierre Roux, Piotr Dollar, Polina

Zvyagina, Prashant Ratanchandani, Pritish Yuvraj, Qian Liang, Rachad Alao, Rachel Rodriguez, Rafi Ayub, Raghotham Murthy, Raghu Nayani, Rahul Mitra, Raymond Li, Rebekkah Hogan, Robin Battey, Rocky Wang, Rohan Maheswari, Russ Howes, Ruty Rinott, Sai Jayesh Bondu, Samyak Datta, Sara Chugh, Sara Hunt, Sargun Dhillon, Sasha Sidorov, Satadru Pan, Saurabh Verma, Seiji Yamamoto, Sharadh Ramaswamy, Shaun Lindsay, Shaun Lindsay, Sheng Feng, Shenghao Lin, Shengxin Cindy Zha, Shiva Shankar, Shuqiang Zhang, Shuqiang Zhang, Sinong Wang, Sneha Agarwal, Soji Sajuyigbe, Soumith Chintala, Stephanie Max, Stephen Chen, Steve Kehoe, Steve Satterfield, Sudarshan Govindaprasad, Sumit Gupta, Sungmin Cho, Sunny Virk, Suraj Subramanian, Sy Choudhury, Sydney Goldman, Tal Remez, Tamar Glaser, Tamara Best, Thilo Kohler, Thomas Robinson, Tianhe Li, Tianjun Zhang, Tim Matthews, Timothy Chou, Tzook Shaked, Varun Vontimitta, Victoria Ajayi, Victoria Montanez, Vijai Mohan, Vinay Satish Kumar, Vishal Mangla, Vítor Albiero, Vlad Ionescu, Vlad Poenaru, Vlad Tiberiu Mihailescu, Vladimir Ivanov, Wei Li, Wenchen Wang, Wenwen Jiang, Wes Bouaziz, Will Constable, Xiaocheng Tang, Xiaofang Wang, Xiaojian Wu, Xiaolan Wang, Xide Xia, Xilun Wu, Xinbo Gao, Yanjun Chen, Ye Hu, Ye Jia, Ye Qi, Yenda Li, Yilin Zhang, Ying Zhang, Yossi Adi, Youngjin Nam, Yu, Wang, Yuchen Hao, Yundi Qian, Yuzi He, Zach Rait, Zachary DeVito, Zef Rosnbrick, Zhaoduo Wen, Zhenyu Yang, and Zhiwei Zhao. The llama 3 herd of models, 2024. URL https://arxiv.org/abs/2407.21783.

Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*, 2021.

Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization, 2019. URL https://arxiv.org/abs/1711.05101.

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

Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. Qlora: Efficient finetuning of quantized llms. *Advances in Neural Information Processing Systems*, 36, 2024.

Michael Hodel. Addressing the abstraction and reasoning corpus via procedural example generation, 2024. URL https://arxiv.org/abs/2404.07353.

Solim LeGris, Wai Keen Vong, Brenden M. Lake, and Todd M. Gureckis. H-arc: A robust estimate of human performance on the abstraction and reasoning corpus benchmark, 2024b. URL https://arxiv.org/abs/2409.01374.

Ilya Loshchilov and Frank Hutter. Fixing weight decay regularization in adam, 2018. URL https://openreview.net/forum?id=rk6qdGgCZ.

torchtune maintainers and contributors. torchtune: PyTorch's finetuning library, April 2024. URL https://github.com/pytorch/torchtune.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*, 2023.

OpenAI. Gpt-4 technical report, 2023.

Aaron Hurst, Adam Lerer, Adam P Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, et al. Gpt-4o system card. *arXiv preprint arXiv:2410.21276*, 2024.

Sam Acquaviva, Yewen Pu, Marta Kryven, Theodoros Sechopoulos, Catherine Wong, Gabrielle Ecanow, Maxwell Nye, Michael Henry Tessler, and Joshua B. Tenenbaum. Communicating natural programs to humans and machines. In *Thirty-sixth Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2022. URL https://openreview.net/forum?id=OxFoLTKDcNm.

A ARC Dataset

We present the tasks in development set, format and evaliation for the ARC dataset (available in this https link.).

A.1 Data Format

We use numpy's array printing format for all the experiments as shown in Fig. 8.

A.2 List of 80 Tasks Used For Development

We use following (Table 2) tasks validation tasks for our development.

Table 2: Selected development tasks and their hardness level based on LeGris et al. (2024b).

ID	Level	ID	Level	ID	Level	ID	Level
0a1d4ef5	easy	762cd429	medium	e5c44e8f	hard	e99362f0	expert
692cd3b6	easy	e7639916	medium	604001fa	hard	1acc24af	expert
1da012fc	easy	e1d2900e	medium	4364c1c4	hard	f9a67cb5	expert
66e6c45b	easy	aee291af	medium	506d28a5	hard	ad7e01d0	expert
3194b014	easy	e95e3d8e	medium	2037f2c7	hard	ea9794b1	expert
963f59bc	easy	e0fb7511	medium	d5c634a2	hard	58e15b12	expert
d37a1ef5	easy	ae58858e	medium	ac605cbb	hard	891232d6	expert
358ba94e	easy	93c31fbe	medium	27f8ce4f	hard	5833af48	expert
f3cdc58f	easy	27a77e38	medium	66f2d22f	hard	4ff4c9da	expert
55059096	easy	9bebae7a	medium	3ed85e70	hard	5b692c0f	expert
c7d4e6ad	easy	9ddd00f0	medium	8b28cd80	hard	e2092e0c	expert
4b6b68e5	easy	fe9372f3	medium	d19f7514	hard	47996f11	expert
00576224	easy	69889d6e	medium	dc2aa30b	hard	34b99a2b	expert
a04b2602	easy	15663ba9	medium	f5c89df1	hard	1c56ad9f	expert
e9c9d9a1	easy	17b80ad2	medium	50f325b5	hard	e6de6e8f	expert
ef26cbf6	easy	16b78196	medium	08573cc6	hard	fea12743	expert
7ee1c6ea	easy	5b6cbef5	medium	3d31c5b3	hard	31d5ba1a	expert
e9ac8c9e	easy	40f6cd08	medium	94133066	hard	79fb03f4	expert
1a2e2828	easy	505fff84	medium	136b0064	hard	8719f442	expert
770cc55f	easy	d017b73f	medium	90347967	hard	a8610ef7	expert

A.3 Evaluation

We follow the competition rules and in any of the two pass@2 predictions of the system is correct, we consider that test as correct. In the reported task level accuracies, we did not give partial points if all tests are not solved, except the final table Section 6.

B TTT

We present transformation used in TTT and the training details.

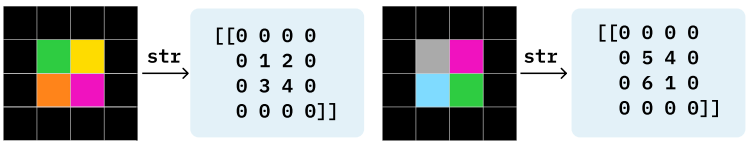

Figure 8: **Data Format:** We convert grids to strings by representing them as numpy arrays of digits from 0 to 10 where each digit corresponds to a different color.

Table 3: We provide the augmentations use in our TTT procedure with their function signature and description.

Augmentation Name	Description
`Rotate(90)`	Rotates a grid 90 degrees.
`Rotate(270)`	Rotates a grid -90 degrees.
`Rotate(180)`	Rotates a grid 180 degrees.
`Flip(0)`	Flips a grid horizontally
`Flip(1)`	Flips a grid vertically
`Reflect(0, reverse=True)`	Flips a grid horizontally and prepend to the left of the original grid
`Reflect(1, reverse=True)`	Flips a grid vertically and prepend to the above of the original grid
`Reflect(0, reverse=False)`	Flips a grid horizontally and append to the right of the original grid
`Reflect(1, reverse=False)`	Flips a grid vertical and append to the left of the original grid
`RandomTranslateXY()`	Shifts a grid randomly both in horizontal and vertical directions. The maximum shift size is 4
`Transpose()`	Reflect a grid on diagonal
`IncreaseResolution(2)`	Upscale the grid by interleaving elements in both horizontal and vertical directions
`IncreaseHeight(2)`	Upscale the grid by interleaving elements in vertical direction
`IncreaseWidth(2)`	Upscale the grid by interleaving elements in horizontal direction
`Chain([Rotate(90),IncreaseResolution(2)])`	Sequential application of `Rotate(90)` and `IncreaseResolution(2)`
`Chain([Rotate(270),IncreaseResolution(2)])`	Sequential application of `Rotate(270)` and IncreaseResolution(2)
`Chain([Rotate(180),IncreaseResolution(2)])`	Sequential application of `Rotate(180)` and IncreaseResolution(2)
`Chain([Flip(0),IncreaseResolution(2)])`	Sequential application of `Rotate(180)` and IncreaseResolution(2)
`Chain([Flip(1),IncreaseResolution(2)])`	Sequential application of `Rotate(180)` and IncreaseResolution(2)
`Chain([Transpose(),IncreaseResolution(2)])`	Sequential application of `Rotate(180)` and IncreaseResolution(2)

B.1 Transformations

We provide the augmentations used in TTT in the Section 3, please refer to our code base for their implementations. After application of these augmentation, we additionaly shuffle colors and shuffle training examples. Note that these transformations are applied to all input and output grids.

B.2 Training Setup & Hyperparameters

We use `torchtune`(torchtune maintainers and contributors, 2024) library to train LoRA adapters on Llama-3 family of models. We apply LoRA training to query and value projection weights of the self-attention layer, to the MLP weights and to the output projection layer (was only available for Llama-3 8B in `torchtune`). We present hyper-parameters of this training in Table 4.

C Inference

We resort to `vLLM` (Kwon et al., 2023) library for prediction as it provides fast kernels and batched inference for our models and LoRA inference. We just use greed decoding as we did not see improvements with

temperature sampling in our early experiments. We use 90, 180 degree rotations, horizontal, vertical, and diagonal (transpose) flips as our invertible transformations.

D LM Data Generation

We described three approaches in Section 5 to use LM, we generated 6426 task generators by few-shot prompting GPT-4 and GPT-4o models (OpenAI, 2023; Hurst et al., 2024).

D.1 Getting Descriptions for Tasks

This procedure is shown in Fig. 10. We initially described 10 training tasks with the hierarchical-style shown in Fig. 6. Then, for other training tasks tasks, we obtained less quality crowd-worker annotations from LARC (Acquaviva et al., 2022) project. By using our high-quality seed annotations and their LARC version, we 10-shot prompt and LM to produce high quality annotations for the other training tasks.

You are an intelligent agent that can induce task descriptions from examples. For Category, please *do not* use generic terms like Transformation, Pattern Recognition.
————————-
Task: {stringified task inputs and outputs}
LARC Description: {description of the task-1 from LARC dataset}
Good Description: {hierarchical description}
————————-
[*truncated*]
————————-
Task: {stringified task inputs and outputs for task-K}
LARC Description: {description of the task-K from LARC dataset}
Good Description: {hierarchical description}
————————-
Task: {stringified task inputs and outputs for query task}
LARC Description: {description of the query task from LARC dataset}

D.2 Few-shot Prompting Details

We use the following simple prompting template with k-shot prompting for all data generation procedures, where numbers filled with examples sampled from seed set. In simple few-shot generation, we exclude examples. We use GPT-4 and GPT-4o to generate the new scripts.

Table 4: TTT hyper-parameters

Hyper parameter	**Search Space**
r lora rank	[128]
α lora alpha	16
learning rate	[5e-5, 1e-4]
epochs	2
batch size	[1, 2]
optimizer	AdamW (Loshchilov and Hutter, 2018)

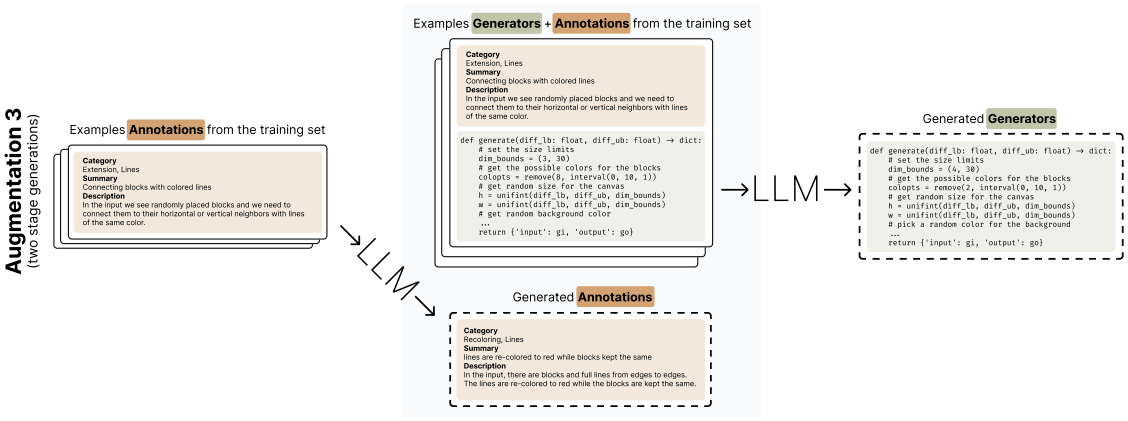

Figure 9: **Two-stage generation using an LLM**: First, we prompt the LLM to generate a task description using few-shot prompting. Then, we generate the new generator based on existing task pairs and the newly created description.

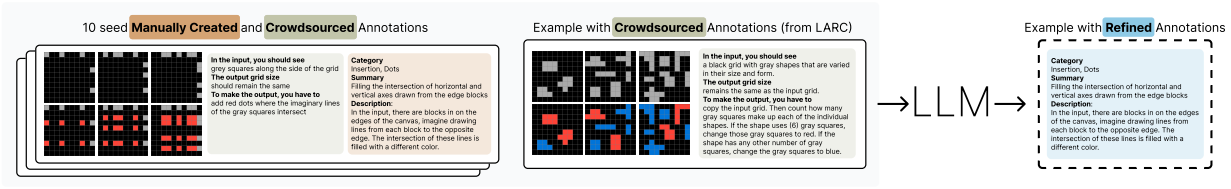

Figure 10: **Generating quality seed descriptions:** We use few-shot prompting to generate descriptions for a given task, using 10 manually created seed descriptions along with crowd-worker annotations from Acquaviva et al. (2022) as few-shot examples. For a given new task, we similarly provide the LM with examples and crowd-worker annotations (available only for training tasks).

> You are a problem generator on 2D grids of colors. Here are some examples of such transformations, please follow the format:
> ———————
> Example: {description of the generator function-1}
> Script: {generator function-1}
> ———————
> [*truncated*]
> ———————
> Example: {description of the generator function-K}
> Script: {generator function-K}
>
> Please generate more and make sure they are different:

E Fine-tuning

In each dataset described in Section 5, we generated approximately 600000 ARC tasks from the available task generator functions (e.g. training tasks, ReARC tasks, and LM generated tasks) by repeatedly picking 2-7 examples from the generated input outputs, and also applying geometric transformations if used.

E.1 Fine-tuning Transformations

We use all the transformations given in Appendix B.1, and some additional transformations given in Table 5. In the fine-tuning case, different from TTT, we apply augmentations to only inputs, only outputs or both.

Table 5: We provide the additional augmentations use in our data generation for fine-tuning with their function signature and description.

Augmentation Name	Description
`Repeat(direction, n)`	Rotates a grid in horizontal or vertical direction by n times.
`DropoutOutput`	Randomly deletes some patches of the output grids.
`DropoutInput`	Randomly deletes some patches of the input grids

Table 6: Fine-tuning hyper-parameters

Hyper parameter	Search Space
learning rate	2.5e-5
epochs	2
batch size	32
optimizer	AdamW (Loshchilov and Hutter, 2018)
scheduler	Cosine LR Schedule with 2k warmup

E.2 Fine-tuning Hyper-parameters

We perform full fine-tuning on LLama-3 family models by using `torchtune` library. We train each model up to 16000 steps. We use 2xNVIDIA A100 GPU for 1B models, 4xNVIDIA A100 GPU for 3B and 8B models. We present hyper-parameters in Table 6.

E.3 Qualitative Examples

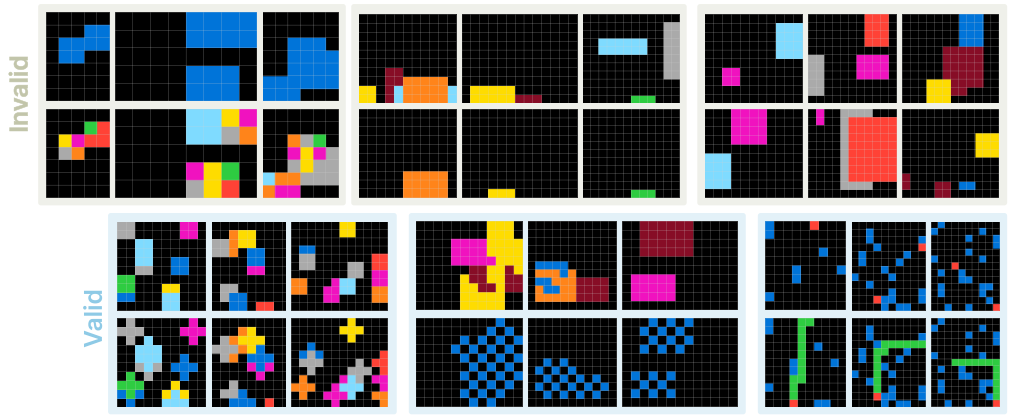

Figure 11: **Example tasks generated by LM data augmentation procedure:** We display three reasonable tasks that we can infer a simple transformation (valid), and three tasks that we could not infer a simple transformation (invalid).

We present some qualitative examples from our LLM data generation procedure in Fig. 11.