# T$^5$-ARC: Test-Time Training for Transductive Transformer Models in ARC-AGI Challenge

**Shaoting Zhu**[*]
2024311565
IIIS, Tsinghua University
zhust24@mails.tsinghua.edu.cn

**Shuangyue Geng**[*]
2024311543
IIIS, Tsinghua University
gengsy24@mails.tsinghua.edu.cn

**Un Lok Chen**[*]
2024270027
IIIS, Tsinghua University
yuanle-c24@mails.tsinghua.edu.cn

## Abstract

The Abstraction and Reasoning Corpus (ARC-AGI) benchmark has emerged as a key challenge in evaluating machine intelligence by emphasizing skill generalization on novel tasks. Recent advancements primarily utilize Test-Time Training (TTT) to enhance reasoning ability of Large Language Models (LLMs). In this project, we focus on TTT for transductive models (end-to-end transformer models that directly generate the output grid) and develop our pipeline following the SOTA methods, which consists of three steps: Base Model Training, TTT and Active Inference. We have conducted detailed experiments to investigate on the effects of different base models, TTT strategies, data and model sizes and the use of negative training samples. Notably, a small transformer model trained from scratch equipped with TTT achieves performance comparable to SOTA LLMs, revealing the potentials of specialized models for certain reasoning tasks.

## 1 Introduction

The concept of Artificial General Intelligence (AGI) has diffused rapidly to the public after the success of ChatGPT [1], but the quest for *what is intelligence?* has a much longer history. In 2019, François Chollet published a paper called "On the Measure of Intelligence," in which he provided an in-depth discussion of how we should define and measure intelligence in terms of the efficiency of *skill acquisition*, rather than the level of skills [2]. Along with the paper, a new benchmark called the Abstraction and Reasoning Corpus (ARC) dataset was released.

### 1.1 The ARC-AGI challenge

Through the lens of knowledge priors, experience (an agent/system's exposure to the new task), and generalization difficulty (of a task), Chollet carefully designed a series of pattern-finding geometric tasks in the ARC dataset [2]. Each task is defined as a split of training examples and test input:

- Training examples: $T_e = \{(x_1, y_1), (x_2, y_2), \cdots, (x_n, y_n)\}$ (typically $n \in [2, 10]$).
- Test input: $T_i = \{x_{test1}\}$ or $\{x_{test1}, \cdots, x_{testk}\}$ (maybe more than 1 test input).

For each task, the goal is to infer the output grid $y_{testi}$ of each test input $x_{testi}$ after observing the set of training examples $T_e$, as illustrated in Fig. 1.

[*]Equal contribution (authors listed in alphabetical order).

Preprint. Course project.

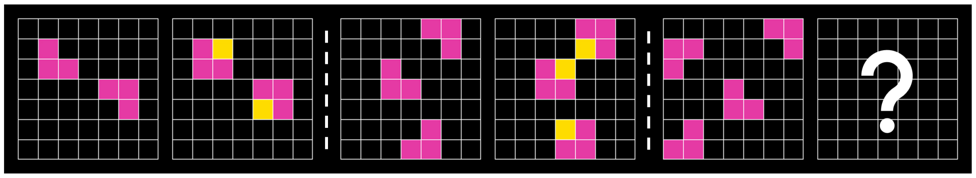

Figure 1: An example of the ARC geometric task.

### 1.2 Challenges and Impacts

The average human performance on the ARC public test is above 60% accuracy [3]. On the contrary, the most competent models can only achieve accuracy below 56% leveraging SOTA LLMs [4]. The gap between current AI and humans is more distinct when considering the vast amount of pre-training data. The investigation of solutions to the ARC competition could bring us significant insights into modeling both the intuition and reasoning process in the human mind, promoting the construction of novel AI paradigms. Meanwhile, "[a]t minimum, solving ARC-AGI would result in a new programming paradigm [5]," enabling program synthesis for people who do not have experience in coding by simply showing a few input-output examples.

In this project, we employed a three-step approach consisting of Base Model Training, Test-Time Training (TTT), and Active Inference to achieve promising results on the ARC task. We aim to analyze the performance of Test-Time Training (TTT) on both pre-trained LLMs and custom Transformer [6] models that performs *transduction* on the ARC-AGI challenge.

## 2 Related work

### 2.1 Induction vs. Transduction

There are different attempts to represent the logic chain in the problem solving process. Omni-ARC [7] explores multiple ARC-related sub-tasks, including the generation of code from examples, the generation of output from code and test input, etc. The variety of approaches are further generalized into two main categories in BARC [8]: *induction* and *transduction*. Induction refers to the process that learns to find an explicit explanation for the examples first and applies such intermediate functions (e.g. program code) on the testing input, while transduction directly generates the output from the given inputs. In our project, we focus on transductive models since we believe that without confining the learning process in an intermediate bottleneck, transduction allows for more flexibility in searching solutions in the latent space and less error in the extra translation process.

### 2.2 SOTA methods in ARC-AGI challenge

In the 2024 technical report [9], both program synthesis and transductive methods are more widely explored with pretrained LLMs. Notably, Test-time Training (TTT) has emerged as a major technique for enhancing the inference ability of fine-tuned LLMs [10, 11], as verifed by the 1st and 2nd place winner of the 2024 challenge (the Architects and Akyürek et al. respectively) [4]. The main idea of TTT is to temporarily update the model parameters in inference time to better adapt to novel tasks.

On the other hand, despite the widespread success of LLMs, the emergence of larger ARC-related training datasets such as Re-ARC [12] and Concept-ARC [13] and the popularization of TTT have encouraged the exploration of smaller task-specific Transformer models trained from scratch. Among them is the notable work LPN [14], which probe into the possibility of discarding large-scale pre-training and synthetic data. One evident advantage of custom models compared with LLMs is the freedom to design architectural components specialized for visual reasoning, exemplified by the 2D nGPT model [15] which employs a small-sized Transformer with 2D attention.

## 3 Method

Our representation of the ARC task can be formulated as follows:

$$y_{\text{test}} = F_{\text{model}}(x_1, y_1, \cdots, x_n, y_n, x_{\text{test}}), \tag{1}$$

where $F_{model}$ is a transductive transformer model to autoregressively generate one cell at a time (assuming only one test output). We summarize our method into three processes: **Base Model Training**, **Test-Time Training**, and **Active Inference**. The overall pipeline is shown in Fig. 2. The details of each process are described in the following sub-sections.

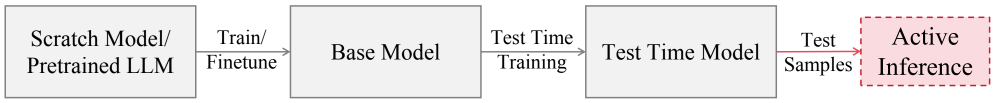

Figure 2: The basic pipeline of our method includes 3 consecutive components: Base Model Training, Test-Time Training, and Active Inference.

### 3.1 Base Model

#### 3.1.1 Base Model Training Datasets

**Data Generation** The original ARC Prize competition provides three datasets: public training set, public evaluation set, and private evaluation set. Both the public training and public evaluation sets include 400 task files, while the private evaluation set consists of 100 task files. Each task has 2 to 10 pairs (typically 3) of training examples and 1 to 3 pairs (typically 1) of tests [2, 16]. This is insufficient for base model training. To address this, several previous efforts have focused on expanding the ARC dataset, and we incorporate their work for training:

1. BARC [8] extracts problem seeds and uses GPT-4/GPT-4o-mini to synthetically generate variations of different ARC problems. It also collects Python programs that solve ARC training tasks to train neural models, resulting in a dataset of 400k samples.
2. Re-ARC [12] employs a DSL (domain-specific language) method to represent each task as a combination of the pre-defined primitive functions. For each task, a procedural generator is hand-crafted, and an unlimited number of samples can be generated by randomizing the parameters of the primitives. Following BARC, we utilize 100k data samples from Re-ARC.

**Data Augmentation** In the training of the custom Transformer model, we further perform data augmentation to enrich the dataset. Specifically, the augmentation process involves two main operations: shuffle and transform. Fig. 3 is an example of data augmentation.

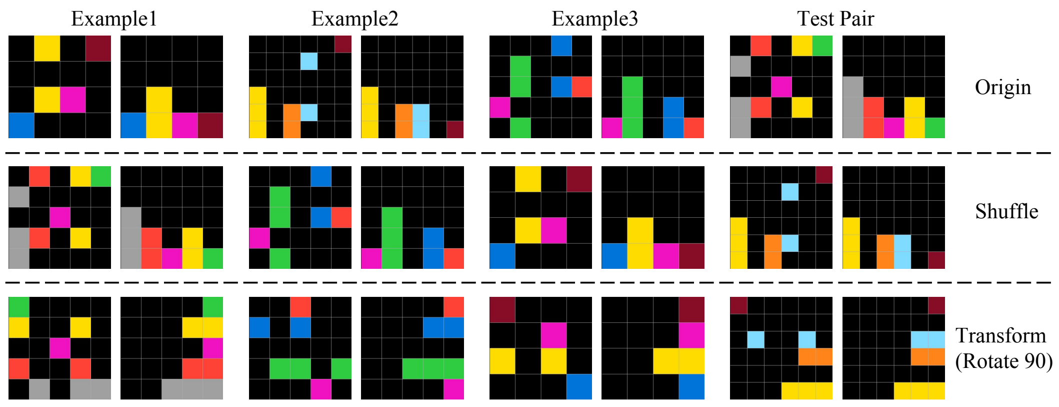

Figure 3: An example of the training data augmentation.

1. **Shuffle**: First, a new test pair $(x^*_{test}, y^*_{test})$ is randomly selected from $(x_i, y_i), i = 1, 2, \cdots, n, test$. Next, the order of the remaining pairs is shuffled, and the modified sample is composed as: $D^* = \{(x^*_1, y^*_1), (x^*_2, y^*_2), \cdots, (x^*_n, y^*_n), (x^*_{test}, y^*_{test})\}$.
2. **Transform**: We apply three transformation strategies: rotate, flip, and color permute. 1) Rotate: Rotate all grids in $D$ randomly by 0, 90, 180, or 270 degrees. 2) Flip: Flip all grids in $D$ either horizontally or vertically. 3) Color Permute: Randomly define a color displacement map and transform the colors of all grids in $D$ based on this map.

### 3.1.2 Base Model Architecture

In our project, all the base models we use are the decoder-only transformer with causal masks. We mainly use two categories of model: open-sourced pre-trained LLMs and custom transformer trained from scratch.

**Open-sourced Pre-trained LLMs.** In our project, we use Llama-3.1-8B-Instruct [17] as the foundation LLM. This model is pre-trained on a dataset containing over 15 trillion tokens derived from publicly available sources, explicitly excluding any meta user data. The instruction-tuned variant is further refined using curated datasets comprising millions of human-annotated examples.

In order to make better use of the knowledge of the pre-trained model, we convert the ARC number grids into text prompt. Specifically, we represent each grid cell with its corresponding color word and then prepend and append prompt words. A concrete example is shown in Appendix B (copied from BARC [8]).

**Custom Transformer from Scratch.** We follow the approach of Nikola Hu [18] to design a decoder-only Transformer trained from scratch, with two notable design features [18]:

1. **Tokenizer**: The tokenizer contains only 19 tokens in total. Tokens 0–9 correspond to different colors, while tokens 11–19 represent special symbols, such as [start], [end] and [padding]. This design significantly reduces the number of prediction classes for the classification head, improving training efficiency.
2. **Grid Positional Embedding**: A specialized grid encoding method is introduced, outperforming traditional NLP positional encodings. This encoding is better suited for the grid-based inputs of ARC tasks.

The detail of the tokenizer and embedding can be found in Appendix C. There is also a data example for the custom transformer training (following Nikola Hu [18]) in the appendix.

### 3.1.3 Training Methods

**Supervised Fine-Tuning (SFT).** We use cross-entropy loss to predict the next token:

$$\mathcal{L}_{\text{SFT}} = -\frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T_i}\log P_\theta(y_t^i \mid y_{<t}^i, x^i), \tag{2}$$

where $N$ is the number of training samples, $T_i$ is the sequence length of the $i$-th sample, $x^i$ is the input sequence, $y_t^i$ is the ground truth token at time step $t$, and $P_\theta$ represents the model's predicted probability for the next token given the previous context.

**Direct Preference Optimization (DPO).** We also experiment Direct Preference Optimization (DPO) [19] on model after SFT trainig. In ARC tasks, we create incorrect test outputs (rejected samples) by randomly changing 10% the color of the ground truth test output grid (with at least one cell modified). The DPO optimization method can then be applied using the following loss function:

$$\mathcal{L}_{\text{DPO}} = -\frac{1}{N}\sum_{i=1}^{N}\log\sigma\left(r_\theta(y_{\text{true},i}) - r_\theta(y_{\text{rejected},i})\right). \tag{3}$$

where $N$ is the number of preference pairs, $\sigma(x)$ is the sigmoid function, $r_\theta$ is the reward model parameterized by $\theta$, $y_{\text{true},i}$ represents the true output for the $i$-th example, and $y_{\text{rejected},i}$ represents the corresponding perturbed (rejected) output. However, in our experiments, we find that the incorporation of DPO does not provide extra benefits, more detailed results can be referred to Appendix A.

## 3.2 Test-Time Training

Traditional LLM inference methods such as in-context learning (ICL) quickly adapt to the sample data with frozen model weights. In contrast, we employ a test-time training (TTT) [11] approach to fine-tune the model parameters via back-propagation before performing inference.

### 3.2.1 Test-Time Training Data

We obtain the test-time training dataset solely from the evaluation set. Recall that each task in the evaluation set is in the format: $D = \{(x_1, y_1), (x_2, y_2), \cdots, (x_n, y_n), x_{\text{test}}\}$. We obtain the augmented training dataset following [11]. As illustrated in Fig. 4, it includes two steps:

1. **Select pseudo-test pair**: We randomly select a test pair $(x^*_{\text{test}}, y^*_{\text{test}})$ from $\{(x_1, y_1), (x_2, y_2), \cdots, (x_n, y_n)\}$. Then, together with the remaining pairs as examples, we can construct a pseudo-problem.
2. **Random data augmentation**: After constructing the pseudo-problem, we apply the same data augmentation strategies as in the base model training stage (shuffle and transform).

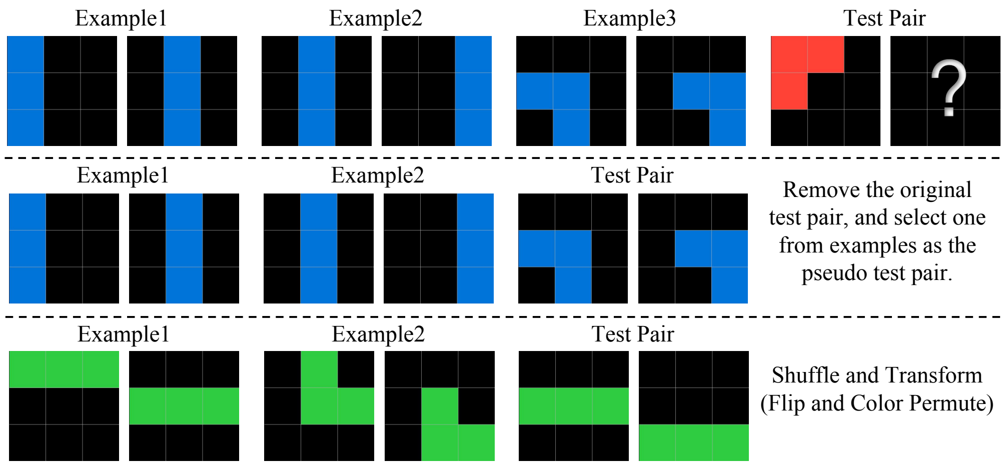
Example1
Example2
Example3
Test Pair
?
Example1
Example2
Test Pair
Remove the original test pair, and select one from examples as the pseudo test pair.
Example1
Example2
Test Pair
Shuffle and Transform (Flip and Color Permute)

Figure 4: An example of TTT training data.

### 3.2.2 Training Methods

We use the same training methods as in the base model training step: SFT and DPO. DPO is applied after the SFT step. We primarily consider two categories of test-time training: *single-TTT* and *multi-TTT*. Specifically, single-TTT means separately training one model for each evaluation task, using only the dataset from that task. In contrast, multi-TTT involves using a mixed dataset from all evaluation tasks to train one model for all evaluation tasks.

### 3.2.3 Meta Learning in Base Model Training Step

We also explore meta-learning methods [20, 21] to further enhance performance. First, we collect a test-training dataset from the entire evaluation set. During the base model training, we alternately train one step on the original training data and one step on the test-training data. After completing the base model training, we apply the single-TTT method for test-time training. Meta-learning enables the model to observe an appropriate number of TTT examples during the base model training phase, theoretically facilitating better convergence of TTT. In our experiments, however, we find that employing such Meta Learning algorithm does not lead to more advantages as expected (more results and analysis can be seen in Appendix A).

## 3.3 Active Inference (AI)

In the inference stage, we apply a two-stage pipeline following the design of Nikola Hu [18], which simply extends the data augmentation strategy in the previous training stages. As illustrated in Fig. 5, we do the following:

1. Augment each testing data to $n$ samples (we use $n = 19$).
2. Perform inference on the $n$ augmented samples individually.
3. Perform reverse transformation on each inference output.
4. Conduct majority vote on the $n$ reversely transformed output (select the top two submissions with the most appearances).

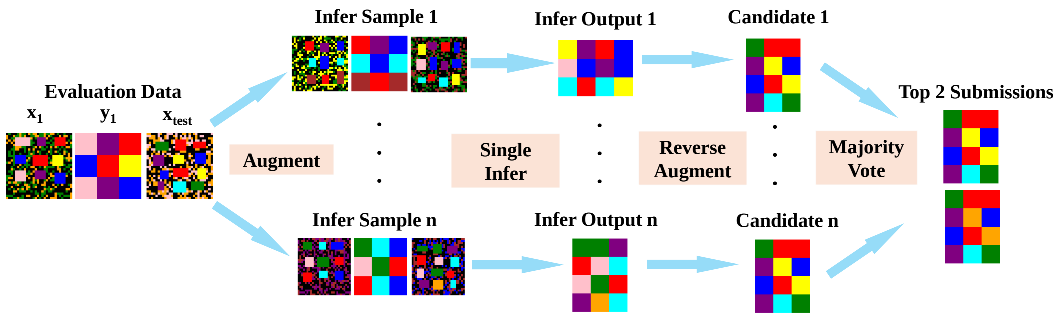

Figure 5: The active inference pipeline. Infer Sample 1 to n are randomly augmented using method mentioned in data generation.

## 4 Experiment

In order to save experimental time, we select 80 tasks from the evaluation set following [11]. This subset includes 20 easy, 20 medium, 20 hard and 20 expert level tasks from the difficulty classification of H-ARC [22]. The detail of task id we selected can be found in Appendix D.

For evaluation metrics, we use correct rate — whether or not the inferred output exactly matches the test output in terms of shape, color, and position. If a task has $n > 2$ test inputs, each test input is counted as $\frac{1}{n}$ questions.

### 4.1 Comparison with Former Works

We compare our model with two recent works, focusing only on the **transductive setting**. Overall, our model achieved a best score of 38/80 using the fine-tuned Llama-3.1-8B-Instruct [17] with single TTT and Active Inference. This performance slightly exceeds the 36/80 reported by the Surprising TTT method [11]. Furthermore, compared to the original BARC method [8], which achieved a score of 31/80 using multi-TTT, our combination of single TTT and Active Inference demonstrates a significant improvement in performance.

Table 1: Transductive results comparison over baselines

| **Ours Best** | **Surprising TTT [11]** | **BARC [8]** |
|---|---|---|
| **38/80** | 36/80 | 31/80 |

### 4.2 Custom vs Pretrained Base Model

We compare the performance of different base models in Table 2. Without TTT and active inference, the small-sized Transformer model trained from scratch exhibits much poorer performance compared with fine-tuned LLM. However, coupled with the same single-task TTT and active inference (AI) strategy, the dedicated Transformer model can reach comparable results as the fine-tuned LLM (37/80 vs 38/80). This remarkable improvement reveals the effectiveness of small and specialized models coupled with TTT.

Table 2: Performance comparison of custom transformer vs pretrained LLM

| Test-time components | Custom Transformer | Finetuned LLM |
|---|---|---|
| w/o TTT, w/o AI | 8/80 | 22.5/80 |
| single TTT, w/ AI | 37/80 | **38/80** |

## 4.3 Test-Time Training and Active Inference

We further investigate the impacts of different TTT methods and the active inference strategy on the fine-tuned LLM, as summarized in Fig. 6.

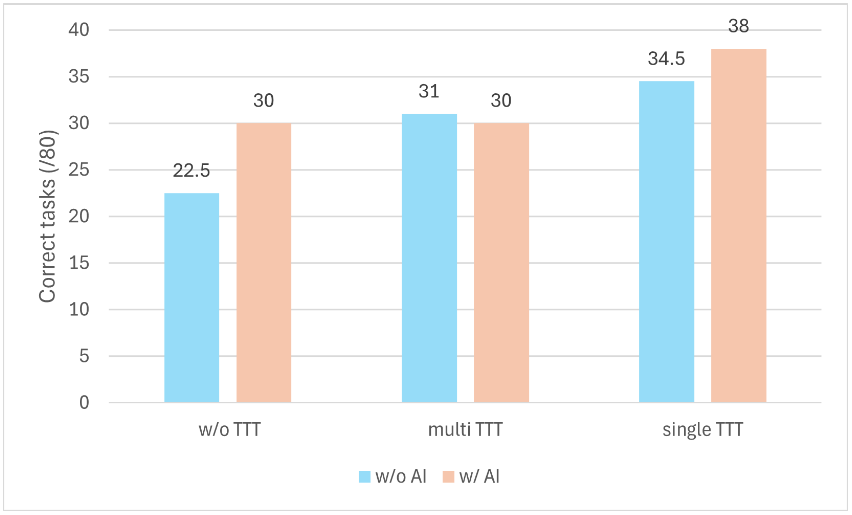

Figure 6: LLM performance across different TTT methods and inference setups.

We find complex relationships between test-time training (TTT) and active inference (AI). In terms of TTT applied individually, both multi-TTT and single-TTT largely boost performance, with the latter one exhibiting a superior enhancement (an increase of 12 correct tasks). This observation is consistent with the major findings by Ekin et al. [11]. On the other hand, active inference also brings moderate increase in inference results when no TTT or single-TTT is applied (an increase in 7.5 and 3.5 number of correct tasks respectively). However, the combined use of multi-TTT and active inference does not show composite positive influence as expected (31/80 vs 30/80). The reason behind such phenomenon is still unknown and is much worth investigating.

## 4.4 Data Size and Model Size Scaling

**Data Size.** We compare model performance with different data sizes. The model used in this section is the same custom model trained with SFT and single-TTT.

Table 3: Performance comparison across data sizes

| Data Size | No TTT (Score/80) | With TTT (Score/80) |
|---|---|---|
| 100k | 8/80 | 33.5/80 |
| 500k | 8/80 | **37/80** |

We notice that with TTT, increasing the data size from 100k to 500k leads to a modest improvement in performance (from 33.5/80 to 37/80). This demonstrates that scaling up the data size can enhance performance on the ARC task.

**Model Size.** We adjust the model size by removing some transformer layers. The model used in this section is a custom model trained with SFT, utilizing all 500k training data, and single-TTT.

Table 4: Performance comparison across model sizes

| Layer Number | No TTT (Score/80) | With TTT (Score/80) |
|---|---|---|
| 6 Layers | 5/80 | 24.5/80 |
| 12 Layers | 12/80 | 34.5/80 |
| 18 Layers | 8/80 | **37/80** |

We observe the following: 1) Increasing the number of layers improves performance, as seen in scores for 6 layers (24.5/80), 12 layers (34.5/80), and 18 layers (37/80) with TTT. This clearly demonstrates that scaling up the model size can lead to better performance. 2) However, for the 18-layer model, performance without TTT is worse than the 12-layer model (8/80 vs. 12/80), likely due to optimization challenges or overfitting.

### 4.5 Grid Encoding in Scratch Model

We compare the model performance across grid encoding in the specially-designed transformer model from scratch. The model used in this section is a custom model trained with Supervised Fine-Tuning (SFT), utilizing all 500k training data, and Single-TTT. The following table summarizes the results.

Table 5: Performance comparison across grid encoding

| Method | No TTT (Score/80) | With TTT (Score/80) |
|---|---|---|
| Grid Positional Encoding | 8/80 | **37/80** |
| Sinusoidal Positional Encoding | 2/80 | 14/80 |

We observe that the grid encoding method significantly outperforms the traditional sinusoidal encoding method. Without TTT, grid encoding achieves a score of 8/80 compared to only 2/80 for sinusoidal encoding. The performance improves further with the application of TTT, where grid encoding achieves 37/80, while the correct rate of sinusoidal encoding is only 14/80. This result demonstrates the potential of custom models that can be tailored to better align with the specific reasoning task.

## 5 Conclusion

In this project, we employ a three-step approach consisting of Base Model Training, Test-Time Training (TTT), and Active Inference to achieve promising results on the ARC task. Our results highlight the significant role of TTT in improving the reasoning performance, with single-TTT outperforming multi-TTT. More importantly, we reveal that a small custom model that allows for specialized architectural designs can achieve comparable results with pre-trained LLMs when utilizing single-TTT and active inference.

However, due to time limitations, we have not tested some concurrent works, such as those released in December that have achieved top scores on the ARC competition 2024 [10, 7]. Besides, some methods, such as Meta learning and DPO, don't lead to performance improvements in some cases. Their underlying mechanisms are not fully understood yet.

Future work will aim to address these limitations and further refine the model configurations. Additionally, the inference phase could benefit from incorporating Chain of Thought (CoT) reasoning [23], which may help identify relationships and contradictions among the outputs, potentially correcting errors and improving accuracy.

## References

[1] Meredith Ringel Morris, Jascha Sohl-dickstein, Noah Fiedel, Tris Warkentin, Allan Dafoe, Aleksandra Faust, Clement Farabet, and Shane Legg. Levels of agi for operationalizing progress on the path to agi. *arXiv preprint arXiv:2311.02462*, 2024.

[2] François Chollet. On the measure of intelligence. *arXiv preprint arXiv:1911.01547*, 2019.

[3] 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*, 2024.

[4] ARC Prize. Arc-agi 2024 high scores, 2024. `https://arcprize.org/leaderboard`.

[5] ARC Prize. Arc-agi, 2024. `https://arcprize.org/arc`.

[6] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[7] Guillermo Barbadillo. Solution summary for arc-24, 2024. `https://ironbar.github.io/arc24/05_Solution_Summary/`.

[8] Wen-Ding Li, Keya Hu, Carter Larsen, Yuqing Wu, Simon Alford, Caleb Woo, Spencer M Dunn, Hao Tang, Michelangelo Naim, Dat Nguyen, et al. Combining induction and transduction for abstract reasoning. *arXiv preprint arXiv:2411.02272*, 2024.

[9] François Chollet, Mike Knoop, Gregory Kamradt, and Bryan Landers. Arc prize 2024: Technical report, December 2024. `https://arcprize.org/media/arc-prize-2024-technical-report.pdf`.

[10] Daniel Franzen, Jan Disselhoff, and David Hartmann. The llm architect: Solving arc-agi is a matter of perspective, December 2024. Accessed: 2024-12-18.

[11] Ekin Akyürek, Mehul Damani, Linlu Qiu, Han Guo, Yoon Kim, and Jacob Andreas. The surprising effectiveness of test-time training for abstract reasoning. *arXiv preprint arXiv:2411.07279*, 2024.

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

[13] Arseny Moskvichev, Victor Vikram Odouard, and Melanie Mitchell. The conceptarc benchmark: Evaluating understanding and generalization in the arc domain. *arXiv preprint arXiv:2305.07141*, 2023.

[14] Clément Bonnet and Matthew V Macfarlane. Searching latent program spaces, 2024.

[15] Jean-Francois Puget. A 2d ngpt model for arc prize, November 2024. Accessed: 2024-12-18.

[16] ARC Prize. Arc prize guide, 2024. `https://arcprize.org/guide#data-structure`.

[17] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

[18] Nikola Hu, 2024. `https://www.kaggle.com/competitions/arc-prize-2024/discussion/546302`.

[19] Rafael Rafailov, Archit Sharma, Eric Mitchell, Christopher D Manning, Stefano Ermon, and Chelsea Finn. Direct preference optimization: Your language model is secretly a reward model. *Advances in Neural Information Processing Systems*, 36, 2024.

[20] Sachin Ravi and Hugo Larochelle. Optimization as a model for few-shot learning. In *International conference on learning representations*, 2017.

[21] Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-agnostic meta-learning for fast adaptation of deep networks. In *International conference on machine learning*, pages 1126–1135. PMLR, 2017.

[22] 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*, 2024.

[23] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

## A Additional Experiment Results

Here we present performance results of DPO and Meta-learning methods, both of which do not produce positive influences as expected.

### A.1 SFT vs DPO

We compare the model performance across DPO. Here, we finetune the pre-trained LLM Llama-3.1-8B-Instruct [17] with multi-TTT.

Table 6: Performance comparison across DPO

| Method | With TTT (Score/80) |
|---|---|
| SFT | **31/80** |
| SFT+DPO | 28/80 |

As presented in Tab. 6, the DPO method is less effective (28/80) than SFT (31/80), suggesting that while DPO might have theoretical advantages, it does not provide a significant improvement in this task. This could be due to the complexity of DPO potentially leading to overfitting.

### A.2 Meta Base Model Learning

We compare the model performance across meta learning. The model used in this section is a custom model trained with SFT, utilizing all 500k training data, and single-TTT.

Table 7: Performance comparison across meta learning

| Method | No TTT (Score/80) | With TTT (Score/80) |
|---|---|---|
| Meta Learning | **13/80** | 26/80 |
| Non-Meta Learning | 8/80 | **37/80** |

The results reveal an interesting trade-off between the two methods: 1) Non-Meta Learning achieves a higher success rate (37/80) compared to Meta Learning (26/80) with test-time training. This suggests that Non-Meta Learning benefits significantly from task-specific training. In contrast, the performance of Meta Learning is significantly worse in this scenario probably due to overfitting. 2) Meta Learning performs better without test-time training. This indicates that Meta Learning provides more robust performance across tasks, especially when test-time training is limited or not feasible.

## B Prompt Example for Finetune Open-sourced Pretrained LLMs

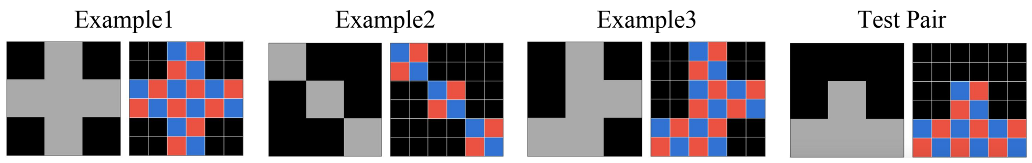

Figure 7: An example of the dataset for training Lama-3.1-8B-Instruct.

**Prompt Example**

**Role: system**
You are a world-class puzzle solver with exceptional pattern recognition skills. Your task is to analyze puzzles, spot patterns, and provide direct solutions.

**Role: user**
Given input-output grid pairs as reference examples, carefully observe the patterns to predict the output grid for new test input. Each pair follows the same transformation rule. Grids are 2D arrays represented as strings, with cells (colors) separated by spaces and rows by newlines.
Here are the input and output grids for the reference examples:

**Example 1**
**Input:**

```
Black Gray Black
Gray Gray Gray
Black Gray Black
```

**Output:**

```
Black Black Blue Red Black Black
Black Black Red Blue Black Black
Blue Red Blue Red Blue Red
Red Blue Red Blue Red Blue
Black Black Blue Red Black Black
Black Black Red Blue Black Black
```

**Example 2**
**Input:**

```
Gray Black Black
Black Gray Black
Black Black Gray
```

**Output:**

```
Blue Red Black Black Black Black
Red Blue Black Black Black Black
Black Black Blue Red Black Black
Black Black Red Blue Black Black
Black Black Black Black Blue Red
Black Black Black Black Red Blue
```

**Example 3**
**Input:**

```
Black Gray Black
Black Gray Gray
Gray Gray Black
```

**Output:**

```
Black Black Blue Red Black Black
Black Black Red Blue Black Black
Black Black Blue Red Blue Red
Black Black Red Blue Red Blue
Blue Red Blue Red Black Black
Red Blue Red Blue Black Black
```

**Test Input:**

```
Black Black Black
Black Gray Black
Gray Gray Gray
```

**Role: assistant**
The output grid for the test input grid is:

```
Black Black Black Black Black Black
Black Black Black Black Black Black
Black Black Blue Red Black Black
Black Black Red Blue Black Black
Blue Red Blue Red Blue Red
Red Blue Red Blue Red Blue
```

## C Details of Training Specially Designed Transformer from Scratch

The following codes are copied from Nikola Hu's repo [18].

For the tokenizer design:

**Tokenizer Design**

```
class SpecialToken(Enum):
    CELL_TOKEN_SIZE = 10
    PAD = auto()                # 11
    START = auto()              # 12
    START_INPUT = auto()        # 13
    END_INPUT = auto()          # 14
    START_OUTPUT = auto()       # 15
    END_OUTPUT = auto()         # 16
    END = auto()                # 17
    ROW_SEPARATOR = auto()      # 18
    COUNT_OF_TOKENS = auto()    # 19
VOCAB_SIZE = SpecialToken.COUNT_OF_TOKENS.value
```

For the grid positional encoding:

**Grid Positional Encoding**

```
def combine_encoding(x: torch.Tensor, batch_size: int,
   use_grid_encoder: bool,
                    seq_length: int, max_grid_size: int,
                       grid_scale: torch.Tensor,
                    grid_encoding: torch.Tensor,
                       positional_encoding: torch.Tensor):
    if use_grid_encoder:
        # Validate input ranges
        assert torch.all(x[:, :, 1:] >= -1), "Indices out of
            range"
        assert torch.all(x[:, :, 1:] < max_grid_size), "Indices
            out of range"

        # Extract positional encodings
        row_encodings = grid_encoding[:, x[:, :, 1], :]
        col_encodings = grid_encoding[:, x[:, :, 2], :]
        top_encodings = grid_encoding[:, x[:, :, 3], :]    #
            Topology/chart information
        sample_encoding = grid_encoding[:, x[:, :, 4], :]  #
            Grid index

        # Combine encodings with scaling
        grid_encodings = torch.cat([
            row_encodings * grid_scale[0],
            col_encodings * grid_scale[1],
            top_encodings * grid_scale[2],
            sample_encoding
        ], dim=-1)  # Shape: (N, seq_length, embed_size // 2)

        return grid_encodings.squeeze(0)

    return positional_encoding[:, :seq_length, :]
```

Then, we provide an example using this special tokenizer:

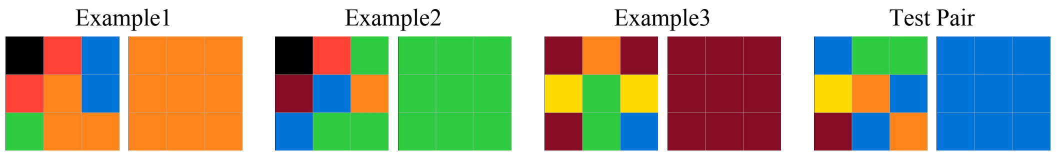

Figure 8: An example of the dataset for training specially designed transformer from scratch.

**Tokenize Example**

```
[[12, -1, -1, -1,  0],  # Begin sequence
 [13, -1, -1, -1,  1],  # Begin input
 [ 0,  0,  0,  3,  1],
 [ 2,  0,  1,  1,  1],
 [ 1,  0,  2,  2,  1],
 [18,  0,  3, -1,  1],  # Row seperate symbol
 [ 2,  1,  0,  1,  1],
 [ 7,  1,  1,  0,  1],
 [ 1,  1,  2,  2,  1],
 [18,  1,  3, -1,  1],
 [ 3,  2,  0,  4,  1],
 [ 7,  2,  1,  0,  1],
 [ 7,  2,  2,  0,  1],
 [18,  2,  3, -1,  1],
 [14, -1, -1, -1,  1],  # End input
 [15, -1, -1, -1,  2],  # Begin output
 [ 7,  0,  0,  0,  2],
 [ 7,  0,  1,  0,  2],
 [ 7,  0,  2,  0,  2],
 [18,  0,  3, -1,  2],
 [ 7,  1,  0,  0,  2],
 [ 7,  1,  1,  0,  2],
 [ 7,  1,  2,  0,  2],
 [18,  1,  3, -1,  2],
 [ 7,  2,  0,  0,  2],
 [ 7,  2,  1,  0,  2],
 [ 7,  2,  2,  0,  2],
 [18,  2,  3, -1,  2],
 [16, -1, -1, -1,  2],  # End output
 [13, -1, -1, -1,  3],
 [ 0,  0,  0,  2,  3],
 [ 2,  0,  1,  3,  3],
 [ 3,  0,  2,  4,  3],
 [18,  0,  3, -1,  3],
 [ 9,  1,  0,  5,  3],
 [ 1,  1,  1,  0,  3],
 [ 7,  1,  2,  6,  3],
 [18,  1,  3, -1,  3],
 [ 1,  2,  0,  0,  3],
 [ 3,  2,  1,  1,  3],
 [ 3,  2,  2,  1,  3],
 [18,  2,  3, -1,  3],
 [14, -1, -1, -1,  3],
 [15, -1, -1, -1,  4],
 [ 3,  0,  0,  0,  4],
 [ 3,  0,  1,  0,  4],
 [ 3,  0,  2,  0,  4],
 [18,  0,  3, -1,  4],
 [ 3,  1,  0,  0,  4],
 [ 3,  1,  1,  0,  4],
 [ 3,  1,  2,  0,  4],
 [18,  1,  3, -1,  4],
 [ 3,  2,  0,  0,  4],
 [ 3,  2,  1,  0,  4],
 [ 3,  2,  2,  0,  4],
 [18,  2,  3, -1,  4],
 [16, -1, -1, -1,  4],
 [13, -1, -1, -1,  5],
 [ 9,  0,  0,  1,  5],
 [ 7,  0,  1,  2,  5],
 [ 9,  0,  2,  3,  5],
 [18,  0,  3, -1,  5],
 [ 4,  1,  0,  4,  5],
 [ 3,  1,  1,  0,  5],
 [ 4,  1,  2,  5,  5],
 [18,  1,  3, -1,  5],
 [ 9,  2,  0,  6,  5],
 [ 3,  2,  1,  0,  5],
 [ 1,  2,  2,  7,  5],
 [18,  2,  3, -1,  5],
 [14, -1, -1, -1,  5],
 [15, -1, -1, -1,  6],
 [ 9,  0,  0,  0,  6],
 [ 9,  0,  1,  0,  6],
 [ 9,  0,  2,  0,  6],
 [18,  0,  3, -1,  6],
 [ 9,  1,  0,  0,  6],
 [ 9,  1,  1,  0,  6],
 [ 9,  1,  2,  0,  6],
 [18,  1,  3, -1,  6],
 [ 9,  2,  0,  0,  6],
 [ 9,  2,  1,  0,  6],
 [ 9,  2,  2,  0,  6],
 [18,  2,  3, -1,  6],
 [16, -1, -1, -1,  6],
 [13, -1, -1, -1,  7],
 [ 1,  0,  0,  3,  7],
 [ 3,  0,  1,  0,  7],
 [ 3,  0,  2,  0,  7],
 [18,  0,  3, -1,  7],
 [ 4,  1,  0,  4,  7],
 [ 7,  1,  1,  1,  7],
 [ 1,  1,  2,  2,  7],
 [18,  1,  3, -1,  7],
 [ 9,  2,  0,  5,  7],
 [ 1,  2,  1,  2,  7],
 [ 7,  2,  2,  1,  7],
 [18,  2,  3, -1,  7],
 [14, -1, -1, -1,  7],
 [15, -1, -1, -1, -1],  # Test output
 [ 1,  0,  0, -1, -1],
 [ 1,  0,  1, -1, -1],
 [ 1,  0,  2, -1, -1],
 [18,  0,  3, -1, -1],
 [ 1,  1,  0, -1, -1],
 [ 1,  1,  1, -1, -1],
 [ 1,  1,  2, -1, -1],
 [18,  1,  3, -1, -1],
 [ 1,  2,  0, -1, -1],
 [ 1,  2,  1, -1, -1],
 [ 1,  2,  2, -1, -1],
 [18,  2,  3, -1, -1],
 [16, -1, -1, -1, -1],
 [17, -1, -1, -1, -1]])
```

## D Subset of Evaluation Set Used in this Project

**Task Levels**

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

## E Hyper-parameters in Pre-trained LLMs

### E.1 Base Model Training

#### E.1.1 SFT

Table 9: SFT training configuration in base model training (full finetune)

| Parameter Name | Value |
|---|---|
| `torch_dtype` | `bfloat16` |
| `learning_rate` | $1 \times 10^{-5}$ |
| `max_seq_length` | 8192 |
| `num_train_epochs` | 3 |
| `gradient_accumulation_steps` | 2 |
| `per_device_train_batch_size` | 8 |
| `weight_decay` | 0.01 |
| `lr_scheduler_type` | `cosine` |
| `warmup_ratio` | 0.1 |

### E.2 Test-Time Training

#### E.2.1 SFT

Table 10: SFT training configuration in test-time training (full finetune)

| Parameter Name | Value |
|---|---|
| torch_dtype | bfloat16 |
| gradient_accumulation_steps | 2 |
| learning_rate | $1.0 \times 10^{-5}$ |
| lr_scheduler_type | cosine |
| max_seq_length | 8192 |
| num_train_epochs | 3 |
| per_device_train_batch_size | 8 |
| warmup_ratio | 0.1 |
| use_liger_kernel | true |
| weight_decay | 0.01 |

#### E.2.2 DPO

In our experiments, we use DPO after SFT only in test-time training step.

Table 11: DPO training configuration in test-time training (LoRA finetune)

| Parameter Name | Value |
|---|---|
| torch_dtype | bfloat16 |
| lora_r | 64 |
| lora_alpha | 64 |
| lora_dropout | 0.00 |
| lora_target_modules | [q_proj, k_proj, v_proj, o_proj, gate_proj, up_proj, down_proj] |
| bf16 | true |
| gradient_accumulation_steps | 2 |
| learning_rate | $1.0 \times 10^{-6}$ |
| lr_scheduler_type | cosine |
| max_length | 8192 |
| num_train_epochs | 1 |
| warmup_ratio | 0.1 |
| weight_decay | 0.01 |

## F Hyper-parameters in Specially Designed Transformer from Scratch

### F.1 Base Model Training

#### F.1.1 SFT

Table 12: SFT training configuration in base model training (full training)

| Parameter Name | Value |
|---|---|
| `epochs` | 50000 |
| `max_seq_length` | 6000 |
| `samples_per_epoch` | 80 |
| `seed` | 0 |
| `learning_rate` | $1.0 \times 10^{-4}$ |
| `warmup_epochs` | 50 |
| `heads` | 4 |
| `base_lr` | $1.0 \times 10^{-8}$ |
| `num_layers` | 18 |
| `batch_size` | 4 |
| `schedular` | `cosine` |
| `embed_size` | 888 |
| `accumulation_steps` | 20 |
| `progressive_head` | 1 |

The hyper-parameters in meta-learning is the same.

### F.2 Test-Time Training

#### F.2.1 SFT

Table 13: SFT training configuration in test-time training (full finetune)

| Parameter Name | Value |
|---|---|
| `epochs` | 16 |
| `batch_multiplier` | 15 |
| `max_seq_length` | 6000 |
| `samples_per_epoch` | 80 |
| `seed` | 0 |
| `learning_rate` | $1.0 \times 10^{-5}$ |
| `base_lr` | $1.0 \times 10^{-8}$ |
| `num_layers` | 18 |
| `batch_size` | 4 |
| `schedular` | `cosine` |
| `embed_size` | 888 |
| `accumulation_steps` | 20 |
| `progressive_head` | 1 |