Solving Visual Puzzels using Large Language Models

Markus Goetz
M1361018
m1361018@cgu.edu.tw

Jye Sawtell-Rickson
M1361019
m1361019@cgu.edu.tw

Debonnet Orion
M1361022
m1361022@cgu.edu.tw

Chang Gung University

Abstract

This study explores the potential of Large Language Models (LLMs) in tackling the Abstract Reasoning Corpus (ARC) Challenge, designed to assess AI's ability to perform human-like abstract reasoning. Traditional ARC solutions rely on extensive image manipulations, which, while effective, lack the cognitive flexibility required for Artificial General Intelligence (AGI). We propose a hybrid approach that combines LLMs with autoencoders (AEs) to bridge this gap. By converting visual inputs into text-based formats, we enable LLMs to reason abstractly, and we further enhance performance through encoding strategies, fine-tuning, and test-time training mechanisms.Our experiments reveal that while LLMs alone struggle with complex transformations, their integration with AEs consistently improves accuracy and reduces errors. Furthermore, reasoning-centric models like o1-mini outperform larger models on tasks requiring logical inference. Recent advancements, such as OpenAI's o3 reasoning model, demonstrate remarkable progress, achieving human-like accuracy on ARC benchmarks through dynamic test-time adaptations. However, high computational costs remain a limitation. These findings highlight the potential of hybrid architectures and reasoning-centric approaches in advancing machine reasoning capabilities, contributing to the broader quest for AGI.

1. Introduction

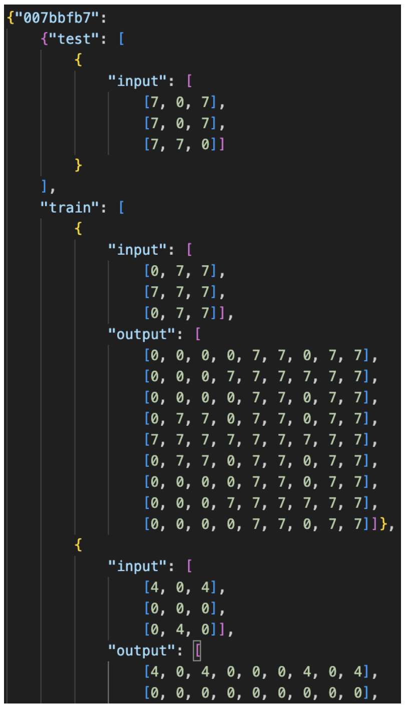
Artificial General Intelligence (AGI) represents a significant milestone in artificial intelligence, where machines achieve cognitive flexibility and reasoning abilities similar to those of humans. Unlike today's AI, which excels at narrow, specific tasks, AGI aims to handle a wide range of cognitive challenges without requiring extensive retraining. Achieving AGI involves developing systems that can reason, adapt, and learn in ways that go beyond pre-programmed knowledge. One approach to advancing these capabilities is the Abstract Reasoning Corpus (ARC) Challenge on Kaggle, a competition that challenges AI systems to solve problems requiring human-like abstract reasoning. ARC presents tasks that mirror the cognitive flexibility necessary for AGI by requiring models to interpret visual patterns, sequences, and transformations with minimal prior information—abilities that current AI systems find challenging. By examining AGI through the lens of ARC, we can better understand both the progress and the limitations in machine reasoning, as well as the broader implications for AGI development.
In recent years of this challenge, some solutions with high success rates have been developed, but none seem to bring AI closer to AGI, as they "brute force" the problem by applying extensive image transformations rather than enabling the AI to "think" through the solution. This insight motivated us to experiment with Large Language Models (LLMs) to solve ARC puzzles. LLMs, such as GPT-3 and ChatGPT, have demonstrated impressive capabilities in natural language processing and understanding, making them promising candidates for solving abstract reasoning tasks. By leveraging LLMs, we aim to develop a more human-like approach to solving ARC puzzles, moving beyond image manipulation to reasoning and understanding.

2. The ARC Dataset

The ARC challenge contains a variety of tasks. Each task includes input/output pairs that demonstrate reasoning pattern to be applied to the "test" input for each task. These are used for training models. For testing the model, the test output is not given, and it must be inferred based on the example tasks. A snippet of the data prvided by the dataset for one of the tasks is shown in Figure 1.

2.1. Expansion of the Dataset

Expanding the dataset enhances the models ability to learn generalizable patterns rather than memorizing specific ex-



```
{"007bbfb7":
    {"test": [
        {
            "input": [
                [7, 0, 7],
                [7, 0, 7],
                [7, 7, 0]]
        }
    ],
    "train": [
        {
            "input": [
                [0, 7, 7],
                [7, 7, 7],
                [0, 7, 7]],
            "output": [
                [0, 0, 0, 0, 7, 7, 0, 7, 7],
                [0, 0, 0, 7, 7, 7, 7, 7, 7],
                [0, 0, 0, 0, 7, 7, 0, 7, 7],
                [0, 7, 7, 0, 7, 7, 0, 7, 7],
                [7, 7, 7, 7, 7, 7, 7, 7, 7],
                [0, 7, 7, 0, 7, 7, 0, 7, 7],
                [0, 0, 0, 0, 7, 7, 0, 7, 7],
                [0, 0, 0, 7, 7, 7, 7, 7, 7],
                [0, 0, 0, 0, 7, 7, 0, 7, 7]]},
        {
            "input": [
                [4, 0, 4],
                [0, 0, 0],
                [0, 4, 0]],
            "output": [
                [4, 0, 4, 0, 0, 0, 4, 0, 4],
                [0, 0, 0, 0, 0, 0, 0, 0, 0],
```

Figure 1. A snippet of the ARC dataset structure

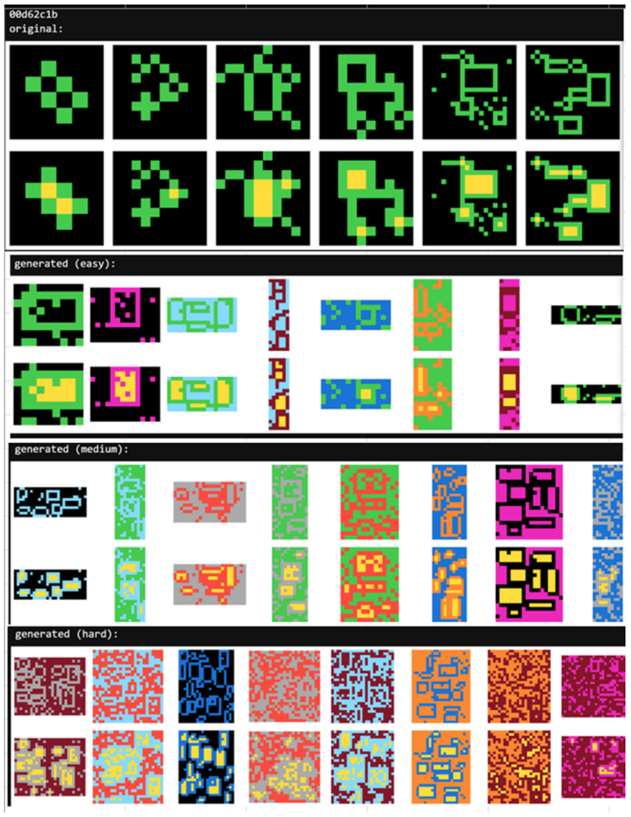


Figure 2. The examples from RE-ARC with different complexitys

amples, which reduces the risk of overfitting. Overfitting occurs when a model performs well on training data but fails to generalize to unseen data. A larger dataset exposes the model to a broader range of variations and outliers, thereby increasing its robustness in handling real-world data. To expand our data pool, we leverage code from the reverse-engineered ARC (RE-ARC) project on GitHub (https://github.com/michaelhodel/re-arc). The project provides a generator for each of the 400 ARC tasks. By generating 1,000 verified examples per task, we obtain an additional 400,000 cases for training and testing. Individual examples from the generated dataset can be accessed by parsing it with a JSON library. An example of both the original and generated datasets is shown in Figure 2.

2.2. Difficulty Ranking

To identify tasks that are more accessible for LLMs to understand and solve efficiently, we categorized the 400 ARC tasks based on difficulty. Since the original dataset does not include difficulty levels, we ranked tasks using two criteria:

1. **Solver Function Line Count**: The ARC-Domain Specific Language (ARC-DSL) is designed to handle tasks within the ARC framework, utilizing principles such as functional design, abstracted functions, and generic operations. ARC-DSL supports a range of operations, including transformations, attribute extraction, and utility functions. We assume that tasks requiring functions with more lines are inherently more complex. (For further details on ARC-DSL, see Section 5.)
2. **PSO Difficulty**: This difficulty measure is based on the assumption that more complex examples typically contain a higher number of pixels, greater color variety, and a higher object density, leading to a higher PSO-Difficulty score.

We constructed two tables using the pandas library: one ranking tasks by ARC-DSL complexity and the other by PSO Difficulty. The tables were then merged and sorted in ascending order, first by ARC-DSL complexity and then by PSO Difficulty. Figure 3 presents examples of the easiest and most challenging tasks according to this ranking.

3. Validating Existing and Basic Solutions

In this section, we assess both established and foundational methodologies applied to the Abstract Reasoning Challenge (ARC). We explore both the autoencoder-based Convolutional Neural Network (CNN) solution and Large Language Models (LLMs) to determine their effectiveness in handling abstract reasoning tasks. Our experiments highlight the strengths and limitations of these methods, particularly in recognizing patterns, applying transformations, and generating solutions. By analyzing these initial attempts, we aim to identify areas where these models succeed and where

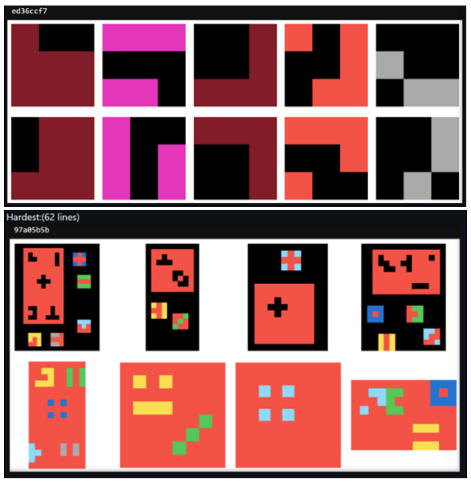


Figure 3. Examples of different difficulty levels in ARC, easy problems (top) and hard problems (bottom).

more sophisticated approaches or iterative improvements are required.

3.1. Autoencoder-Based CNN Solver Replication

We set out to replicate the autoencoder and CNN methodology presented by Kirill Khoruzhii on Kaggle, using a pre-trained model. This approach demonstrates genuine ARC problem-solving potential, achieving 25% accuracy for fully correct images and 93% accuracy for correct pixels. The main goal of the notebook is to establish a foundation for a CNN-based model architecture and illustrate the capabilities of CNN generation from the latent space. While we do not yet have an improvement plan for this implementation, understanding the strengths and weaknesses of this approach is valuable. CNNs are highly effective in processing local pattern recognition tasks and can also build non-local structures to understand connected areas. They excel in local tasks, such as denoising and shape construction, but are less effective in handling symmetry understanding (see Figure 4).

3.2. Basic Solutions Using LLMs

After categorizing the puzzles by difficulty, we began by attempting to solve simpler puzzles using only LLM-based prompts. Initially, we asked LLMs to directly generate solutions for the puzzles. However, due to the inherent limitations of LLMs in image generation, the output quality was poor.

To improve the results, we provided an example image (Figure 5) to ChatGPT-4, explaining that it was a puzzle requiring pattern recognition and transformation. The initial response lacked accuracy, so we intervened by giving further guidance.

Once the LLM had a clearer understanding, we sent a subsequent image with instructions to apply the derived transformation (see Figure 6).

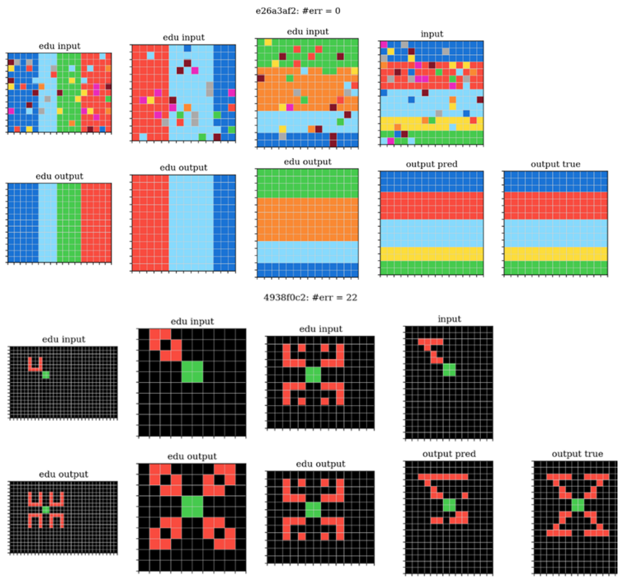


Figure 4. The good and bad performance from AE-based CNN

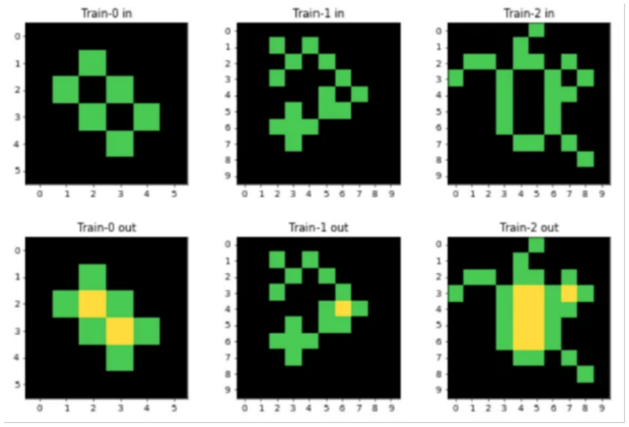


Figure 5. Original prompt photo

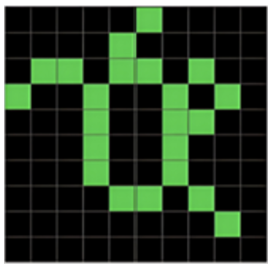


Figure 6. Photo sent after the LLM understood the transformation

The response of the LLM, shown in Figure 7, contained the following description:

> "Here is the transformed image following the same logic as before, focusing on emphasizing key areas of the turtle-like pattern using yellow highlights. The transformation reflects central or significant parts of the structure, similar to the process applied to previous grids."

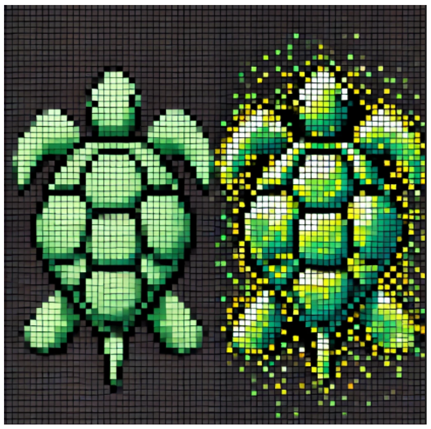


Figure 7. Attempted transformation by ChatGPT-4

However, two main issues were identified:

- The LLM seemed to misunderstand the transformation requirements, as reflected in the response text.
- The generated image was imprecise, failing to replicate even the input puzzle with consistency.

3.3. Transitioning to Matrix Inputs and Code Generation

Given the limitations with image recognition and generation, we transitioned to using matrix representations of puzzles as inputs, requesting Python code as output instead. This approach yielded better results. The conversation with ChatGPT which can be found here, shows the first prompt where ChatGPT understood the puzzle's logic; however, the initial function output did not produce the correct transformation. It only adjusted the image size without applying the required symmetry operation (see Figure 8). After pointing out this issue to the LLM, we received a refined function that performed the task correctly.

These experiments have revealed several insights:

- Matrix inputs are necessary for LLMs to understand and solve puzzles effectively, as the output must be generated in text or Python function form.
- Even with simple puzzles, it is unlikely that an LLM will produce a working function on the first attempt. Feedback is essential, but relying on human intervention is inefficient. Therefore, an efficient, automated feedback mechanism is required.

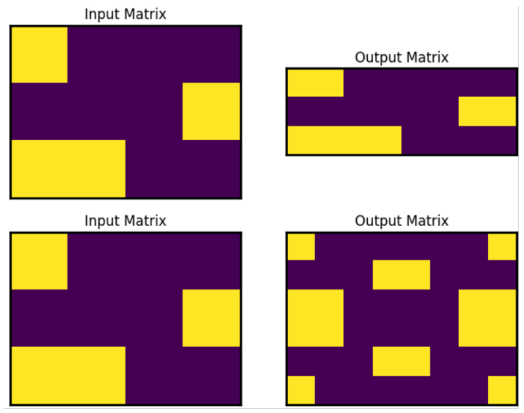


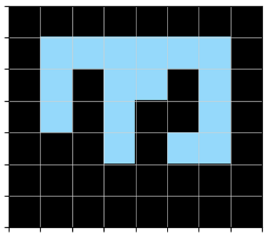
Figure 8. Comparison between the output of the LLM function (top) and the desired output (bottom)

4. Encoding for LLMs

To query LLMs, ARC problems must first be encoded as text. There are several methods to encode the data, including:

- Pixel Encoder
- Object Encoder

The **pixel encoder** converts an ARC problem into a 2D array representation, as shown in Figure 9.



```
[[0 0 0 0 0 0 0 0]
 [0 8 8 8 8 8 8 0]
 [0 8 0 8 8 0 8 0]
 [0 8 0 8 0 0 8 0]
 [0 0 0 8 0 8 8 0]
 [0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0]]
```

Figure 9. Example showing the original training example (left) and its pixel encoding (right), where each color is mapped to an integer in a two-dimensional array.

This encoding is the simplest to implement. However, due to the 1D sequential nature of transformers powering LLMs, it may not be the most optimal encoding method. The **object encoder** first processes an ARC problem to identify objects according to a defined algorithm. Objects are defined as regions of the same color with contiguous connections. While this definition is not perfect, it successfully extracts objects in many cases. For instance, in Figure 10, the encoder extracts both the blue and red regions as separate objects.

In addition to the above encodings, transformations can also be applied to the output. Multiple transformations can be appended to the input to the LLM along with the untransformed encoding. By providing multiple "views" of the

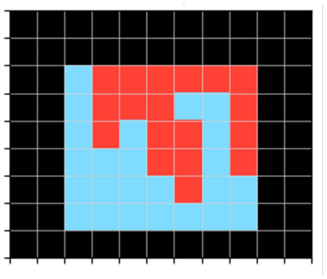


Figure 10. Example of an image with two 'objects' which must be separately extracted.

problem to the LLM, we hypothesize that it may improve problem-solving performance, especially for transformers, which are naturally 1D sequential models. The transformations considered include:

- **Color Swap**: The solution should be invariant to swapping colors (e.g., red with blue), so we can alter the integers to get different encodings.
- **Flips**: Flipping the problem along either the horizontal or vertical axis.
- **Rotations**: Performing 90-degree rotations, similar to flips.

Future work may include:

- Encoding representations with color names (e.g., "cyan") instead of integers (e.g., 8). We hypothesize this may enhance contextual understanding for the LLM.
- Experimentally validating the effectiveness of different encoding methods.

5. Fine Tuning

One of the approaches for solving the ARC Challenge was exploring fine-tuning of Large Language Models (LLMs). The goal was to enhance the general world knowledge of the LLM with the specific domain knowledge of solving complex visual puzzles using Python code. The first step in this approach was to create a dataset that mimics the expected behavior for solving these puzzles.

5.1. Creating the Dataset

Before dataset creation, our first objective was to understand the requirements for fine-tuning the model to solve ARC puzzles. ARC puzzles require models to identify abstract transformations between input and output patterns. This necessitated creating a dataset that would not only present visual puzzles as input but also include correct solutions and transformations for the model to learn.

For this task, the dataset had to simulate a puzzle-solving interaction, where each entry included a visual puzzle, an expected solution, and the transformation rules. The dataset was structured as follows:

- **System Message:** A prompt explaining the puzzles, the input data, and the expected output the LLM should generate.
- **Input:** Multiple puzzle images of the same type in array format, as well as the corresponding solutions.
- **Output:** A transformation function mapping the initial pattern to the desired solution, serving as the model's objective.

Thus, the input data for each entry represented a specific puzzle case (as user input), while the output was the transformation function (as assistant output). The goal was for the model to learn these mappings and independently infer transformation rules.

To define transformation functions efficiently, we relied on a pre-existing repository, the ARC-DSL (Domain-Specific Language) created by Michael Hodel, available at GitHub. This repository contains hardcoded Python solutions for each ARC puzzle, encapsulated as functions. By parsing these Python functions, we could systematically extract transformation rules and incorporate them into our dataset. Each transformation function in ARC-DSL directly solved a puzzle, allowing us to map each puzzle input to a known solution function, which became the target output for fine-tuning.

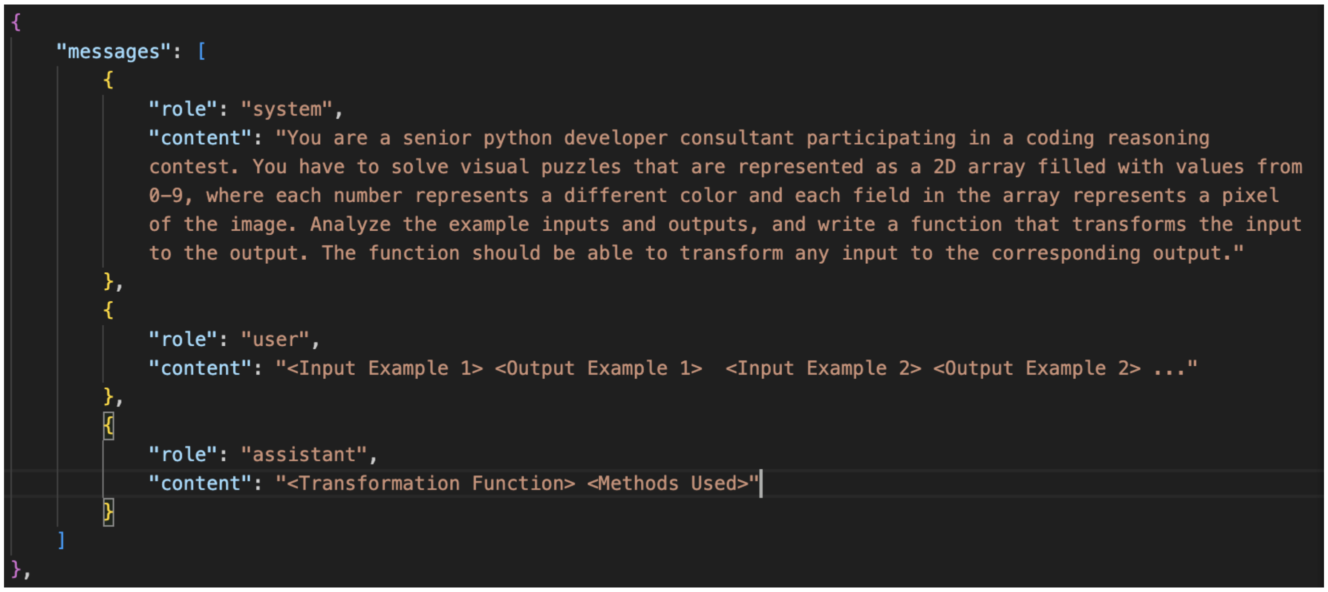
The dataset was formatted according to the fine-tuning structure used by OpenAIs models, with minor modifications to better suit our task. Each training instance in our dataset adhered to the following format represented in figure 11:

- **<Input Example X>:** Contains the structured representation of one visual puzzle.
- **<Output Example X>:** Holds the representation of the solution of that puzzle.
- **<transformation_function>:** Represents the Python function for solving that specific puzzle.
- **<Methods Used>:** Lists the code for all the methods that are used in the transformation function.

5.2. Fine Tuning Process

The fine-tuning was executed using a combination of tools combining efficient fine-tuning frameworks with capable hardware for the high demands of fine-tuning Large Language Models. The following tools were utilized:

- **Llama 3.1 8B:** An open-source 8-billion-parameter Large Language Model published by Meta. It was chosen for fine-tuning due to its balance of performance and manageable computational requirements. Llama 3.1 8B provided a robust architecture for the task, offering sufficient capacity to potentially learn transformation functions for ARC puzzles.
- **Unsloth:** A fine-tuning framework that streamlined the process of training our LLM with the customized ARC dataset. It provided a structured and efficient approach



```
{
    "messages": [
        {
            "role": "system",
            "content": "You are a senior python developer consultant participating in a coding reasoning
            contest. You have to solve visual puzzles that are represented as a 2D array filled with values from
            0-9, where each number represents a different color and each field in the array represents a pixel
            of the image. Analyze the example inputs and outputs, and write a function that transforms the input
            to the output. The function should be able to transform any input to the corresponding output."
        },
        {
            "role": "user",
            "content": "<Input Example 1> <Output Example 1>  <Input Example 2> <Output Example 2> ..."
        },
        {
            "role": "assistant",
            "content": "<Transformation Function> <Methods Used>"
        }
    ]
},
```

Figure 11. Example structure of the fine-tuning dataset in OpenAI format

to setting up the fine-tuning environment, allowing us to control batch sizes, learning rates, and other parameters essential for effective model training on this specific task.

- **Kubeflow:** The university-hosted Kubernetes platform equipped with GPU capabilities was initially chosen as the primary computational environment for the fine-tuning process. However, we soon encountered limitations with Kubeflow due to GPU capability constraints, so it was not possible to use Kubeflow for fine-tuning.
- **Google Colab:** Due to issues with Kubeflow, we resorted to using Google Colab as an alternative environment. The Google Colab free tier provides a Nvidia T4 GPU with 16GB of VRAM, enabling the fine-tuning process even though some limitations, such as a usage time limit, remained. With this, the fine-tuning process of the Llama model took two hours to complete.

Despite completing the fine-tuning process, the models performance was suboptimal. The trained model demonstrated limited success in generating usable Python code for solving ARC puzzles. Most generated code was incomplete with missing functions and general syntax errors. an example output of the Model can be seen in Fine-tuning of the Llama Model suggests that the current dataset and fine-tuning approach are insufficient for the desired level of problem-solving. The models inability to generate correct, runnable code indicates that the created dataset lacks the quality for sufficient fine-tuning. Moving forward, further refinement of the dataset, a different structure, or even a different type of data might be necessary to improve model performance and enhance its capabilities for solving the ARC puzzles.

6. Automated Experimentation with LLMs

To validate various components of our solution, such as encoding and prompt engineering, we required a scalable approach to evaluate performance. For this purpose, we developed an automated pipeline for testing, which operates as follows:

- Define a subset of tasks (typically ordered by difficulty).
- Encode each task as a prompt.
- Test each prompt (or variations of it) multiple times for each task.
- Report results as success rates.

Given the substantial resource costs of LLMs, either in monetary or computational terms, the experiments are conducted using smaller LLMs. Specifically, we run the experiments on the CGU system, configured with 4 CPU cores, 16GB of RAM, and a single GPU. We use Ollama to create an LLM server that can be queried via Python.

We explored various LLMs as the backend, ranging from models with 3 billion to 16 billion parameters. These include:

- `deepseek-coder-v2`
- `mistral`
- `llama3.2`
- `mistral-nemo`
- `qwen2.5-coder`
- `codegemma`
- `llava:13b`

7. Combining Autoencoders and LLMs

Autoencoders (AE) demonstrate the ability to perform certain tasks autonomously; however, their accuracy is often limited. Large Language Models (LLMs), on the other hand, excel at addressing tasks but frequently rely on iterative feedback for optimization. To mitigate these limitations, we propose a novel methodology that synergistically combines Autoencoders with LLMs. This integrated approach aims to leverage the respective strengths of both paradigms, yielding improved results compared to either method utilized in isolation. The proposed methodology employs a two-step process:

1. **Solution Generation by Autoencoder:** The Autoencoder (AE) first generates a preliminary solution for the given task.
2. **Solution Refinement by LLM:** The initial solution from the AE is formatted into a prompt for the LLM. This prompt includes:
   - Examples illustrating the task.
   - An explicit explanation of how the LLM should refine or correct the AE's output.

   The LLM then evaluates the generated solution and determines if further modifications are necessary.

This iterative mechanism capitalizes on the contextual reasoning capabilities of LLMs and the generative capabilities of AEs, thereby enhancing the overall task performance.

8. LLMs with Mechnisms from Test Time Training

In the paper "The Surprising Effectiveness of Test-Time Training for Abstract Reasoning" [arXiv], the authors propose a method to improve the performance of LLMs on abstract reasoning tasks. Test-Time Training (TTT) is an adaptive optimization technique that enables models to update their parameters during inference to better handle specific test cases. Unlike traditional approaches where models remain static after training, TTT allows for dynamic adaptation to each new input. The process involves generating a task-specific training dataset, temporarily updating model parameters through techniques like Low-Rank Adaptation (LoRA), making predictions, and then resetting parameters for the next input. To further enhance prediction quality, the authors implemented an Augmented Inference and Hierarchical Voting mechanism. For this the authors create subtasks form the existing data by leaving out one of the training examples. Therefore they can generate n subtasks for n training examples of a task. Each of the subtask applies inversible geometric transformations (such as rotations and reflections) to the input data. These transformed versions are processed independently, and their predictions are transformed back to the original orientation. The results are then aggregated through a hierarchical voting process to select the most likely answer. This mechanism is illustrated in Figure 12. For our test we adapted the augmentation mechanism. We create "leave-one-out" tasks for each of the training examples and apply the transformations to the input data. We then use the LLM to predict the answer for each of the transformed inputs and aggregate the results using a hierarchical voting mechanism. The final prediction is the answer with the highest number of votes.

9. Results

Our experiments produced several key insights regarding the integration of large language models (LLMs) with computer vision (CV) methods, the necessity of reasoning for the Abstraction and Reasoning Corpus (ARC), and performance differences across problem difficulties. Below, we present these findings in detail.

9.1. Performance Metrics

To evaluate model performance comprehensively, we defined a suite of metrics that capture different aspects of accuracy and model behavior. These metrics fall into two primary categories:

Accuracy Metrics:

- Overall Accuracy: Percentage of tasks correctly predicted.
- Pixel Accuracy: Percentage of pixels correctly predicted.
- Shape Accuracy: Percentage of shapes correctly identified and predicted.

LLM-Specific Metrics:

- Percentage Corrected: Percentage of tasks where the LLM modified autoencoder (AE) predictions to yield correct outputs.
- Percentage Incorrected: Percentage of tasks where the LLM modified AE predictions, resulting in incorrect outputs.
- Failure Rate: Percentage of tasks where the LLM failed to produce parseable output.

These metrics allow for a nuanced analysis of both traditional CV methods and their integration with LLMs, highlighting not only overall performance but also the specific contributions and limitations of LLMs in correcting and refining predictions.

With these metrics established, the following sections present detailed results and insights derived from our experiments.

9.2. Performance Enhancement with LLM Integration

Table 1 compares the performance of various configurations combining autoencoders (AE) with LLMs. Among these, the configuration of AE + LLM achieved the highest overall accuracy of 31%, outperforming AE alone (18%) and the standalone LLM (28%). Notably, while AE + LLM did not

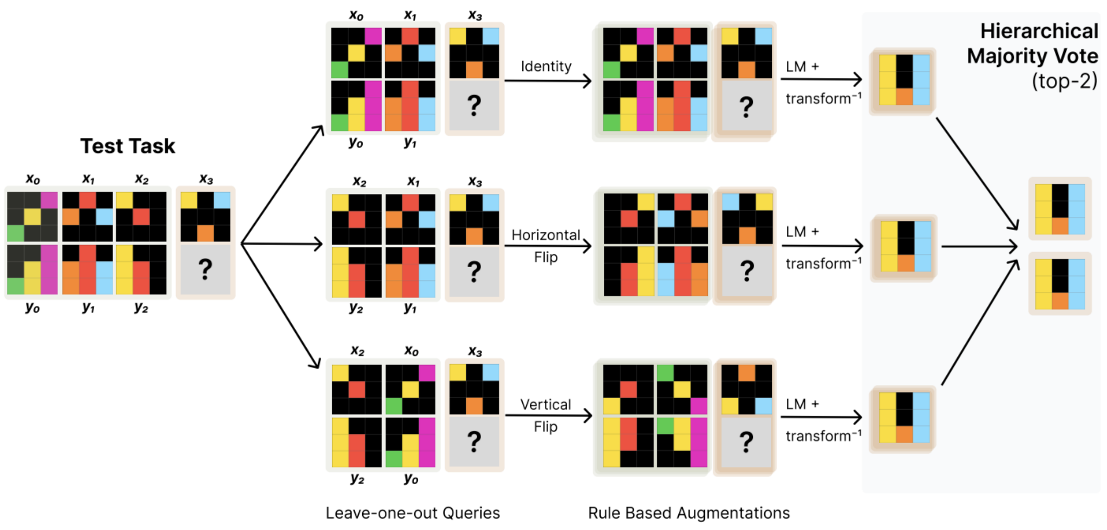


Figure 12. Augmented Inference and Hierarchical Voting mechanism (source).

lead to significant improvements in pixel accuracy or shape accuracy compared to other configurations, it demonstrated a substantial reduction in the percentage of incorrect predictions (5%) compared to standalone LLMs (13%).

These results highlight that integrating LLMs with traditional CV methods, such as autoencoders, effectively leverages the strengths of both modalities. The superior reduction in errors further underscores the complementary roles of AE and LLMs, particularly in tasks requiring nuanced contextual understanding and reasoning.

9.3. The Role of Reasoning in ARC Performance

Table 2 illustrates that the o1-mini model was the only model to achieve comparable performance to AE + LLM across metrics, with a similarly low failure rate (0%). This finding suggests that reasoning, a key capability of o1-mini, is critical for success in ARC tasks. Conversely, models like qwen2.5-coder and mistral, which lack advanced reasoning capabilities, exhibited higher failure rates (50% and 54%, respectively) despite comparable or larger model sizes.

This divergence in performance underscores that success in ARC is not merely a function of model size but rather of the model's ability to perform abstract reasoning. The o1-mini's design, emphasizing reasoning, allows it to outperform larger models in tasks requiring logical inference and pattern recognition.

9.4. LLM Superiority on Difficult Problems

Table 3 examines model performance across tasks of varying difficulty. For easy problems, AE alone outperformed all LLM-based configurations, achieving an accuracy of 13%. However, as problem difficulty increased, smaller LLMs like qwen2.5-coder (14b) began to demonstrate superior performance compared to AE, achieving 13% accuracy on hard tasks versus 8% for AE.

This trend highlights that while smaller LLMs may struggle with simpler tasks, they excel in scenarios demanding deeper reasoning. These findings further corroborate the earlier insight that LLMs' reasoning capabilities provide a distinct advantage in solving complex and abstract problems.

9.5. Summary of Key Observations

LLMs Enhance CV Methods (1): The integration of LLMs with autoencoders consistently improved task accuracy and reduced the percentage of incorrect predictions, demonstrating the complementary strengths of these approaches.

Reasoning is Essential for ARC (2): Performance disparities across models indicate that reasoning is a fundamental requirement for success in ARC tasks, with o1-mini outperforming larger models lacking reasoning capabilities.

LLMs Excel at Hard Problems (3): Smaller LLMs outperform traditional CV methods as problem difficulty increases, suggesting that LLMs are particularly adept at tasks requiring abstract reasoning and deeper understanding.

These findings collectively underscore the potential of integrating reasoning-centric LLMs into traditional CV workflows and highlight the importance of designing models optimized for specific task complexities and reasoning requirements.

10. LLM Cost

According to OpenAI's latest pricing policy (OpenAI API Pricing), API calls cost \$2.50 per 1 million input tokens

Table 1. Performance Comparison Across Configurations

Test	Accuracy	Pixel Accuracy	Shape Accuracy	% Corrected	% Incorrected (↓)
AE + LLM + TTT	28%	87%	93%	19%	9%
AE + LLM	31%	83%	93%	19%	5%
LLM	28%	83%	88%	(23%)	(13%)
AE	18%	85%	90%	-	-

Table 2. Model Performance Metrics

Model	Model Size	Correct	Correct Pixels	Correct Shape	% Corrected	% Incorrected (↓)	Failure Rate (↓)
o1-mini	?	31%	83%	93%	19%	5%	0%
gpt-4o	1800b	16%	82%	88%	10%	11%	0%
qwen2.5-coder	14b	11%	82%	86%	1%	8%	50%
gpt-4o-mini	8b	3%	78%	86%	1%	16%	0%
mistral	7b	10%	81%	86%	0%	8%	54%
(none)	-	18%	85%	90%	-	-	-

and $10.00 per 1 million output tokens. Given an estimated 4,000 input tokens and 2,000 output tokens per request, the final cost estimate for 9,000 API calls is detailed in Figure 13.

After running various experiments, we can found the costs are as follows:

- Total cost: $30
- Main cost: running 80 o1-mini tasks.
- Cost per task for o1-mini: 40c

In the end, we did not run any tasks on the o1 model. It's estimated to be around 5x more expensive than the mini variant so would cost around $120 to run for the subset of 40 tasks with two measures.

11. New Advancements

During the final stages of writing this report, OpenAI announced a major breakthrough in artificial intelligence with the announcment of their o3 reasoning model. The o3 model achieves human-like accuracy on the ARC benchmark. It scored an unprecedented 75.7% on the ARC-AGI Semi-Private Evaluation set and 82.8% on the Public Evaluation set under high-efficiency settings. When operating under low-efficiency conditions with increased computational resources (172x), it achieved scores of 87.5% and 91.5% respectively. A summary of the model's performance is shown in Table 4.

The o3 model distinguishes itself from traditional GPT-family architectures by incorporating test-time program search and execution. This dynamic approach allows the model to adapt to novel reasoning tasks beyond static pre-training capabilities. Furthermore o3's prompts only use simple instructions, such as the following:

Find the common rule that maps an input grid to an output grid, given the examples below.

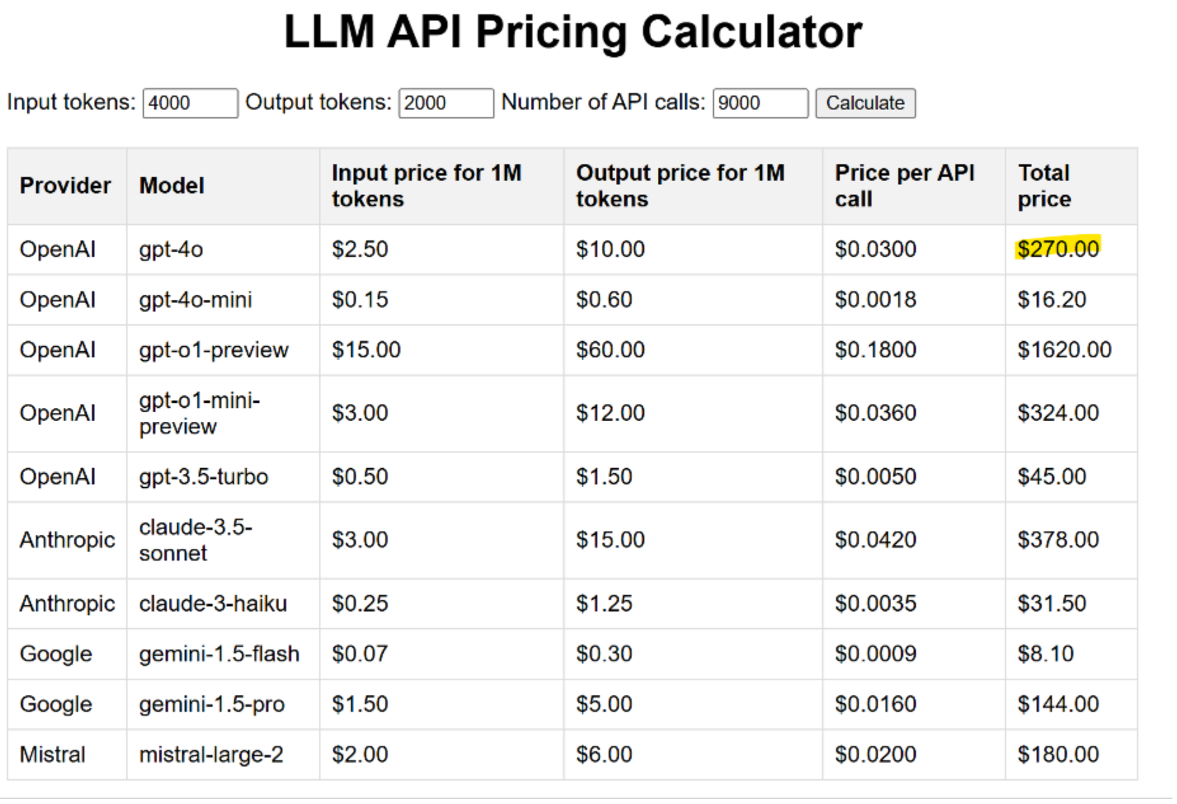
The prompt then provides the task examples of input and output grids, with the instruction being to generalize the rule and apply it to a test task. Looking ahead, François Chollet, the creator of the ARC benchmark, is now working on a more challenging ARC2 benchmark, where o3's scores drops significantly below 30% on this new benchmark. The ARC2 benchmark aims to set a new standard for AGI research by emphasizing even more complex reasoning and adaptability. While the o3 reasoning model has achieved groundbreaking results on the ARC benchmark, it is not without limitations. One notable challenge is the substantial computational resources required, even in the high-efficiency setting. At a cost of 17$ to 20$ per task, this mode achieves strong performance but may still be prohibitive for widespread or real-time applications. The low-efficiency mode, while demonstrating the model's upper bounds with a remarkable 91.5% score on the public dataset, requires billions of tokens per task. This comes at the cost of significantly higher computation times, reaching 13.8 minutes per task for the semi-private dataset and a 172x higher computational cost. Such resource demands emphasize the need for further optimization to make the model more cost-effective and scalable. Striking a balance between efficiency and performance will be crucial for enabling practical deployment across diverse use cases.

12. Conclusion

The exploration of solving the ARC Challenge using LLMs and their integration with autoencoders has highlighted the potential of hybrid approaches to enhance AI's ab-

Table 3. Accuracy Across Difficulty Levels

Model	Size	Easy	Medium	Hard
qwen2.5-coder	14b	9%	10%	13%
gpt-4o-mini	$\sim$8b	2%	4%	5%
mistral	7b	7%	10%	7%
(none)	-	13%	10%	8%



LLM API Pricing Calculator

Input tokens: 4000 Output tokens: 2000 Number of API calls: 9000 Calculate

Provider	Model	Input price for 1M tokens	Output price for 1M tokens	Price per API call	Total price
OpenAI	gpt-4o	$2.50	$10.00	$0.0300	$270.00
OpenAI	gpt-4o-mini	$0.15	$0.60	$0.0018	$16.20
OpenAI	gpt-o1-preview	$15.00	$60.00	$0.1800	$1620.00
OpenAI	gpt-o1-mini-preview	$3.00	$12.00	$0.0360	$324.00
OpenAI	gpt-3.5-turbo	$0.50	$1.50	$0.0050	$45.00
Anthropic	claude-3.5-sonnet	$3.00	$15.00	$0.0420	$378.00
Anthropic	claude-3-haiku	$0.25	$1.25	$0.0035	$31.50
Google	gemini-1.5-flash	$0.07	$0.30	$0.0009	$8.10
Google	gemini-1.5-pro	$1.50	$5.00	$0.0160	$144.00
Mistral	mistral-large-2	$2.00	$6.00	$0.0200	$180.00

Figure 13. Cost calculator estimating the expenses for API calls.

stract reasoning capabilities. Autoencoders excel at recognizing local patterns, while LLMs demonstrate the capacity for deeper reasoning, particularly in challenging tasks. Integrating these methods improved task accuracy and reduced error rates, leveraging their complementary strengths. Reasoning-centric models like o1-mini showed that reasoning capabilities, rather than model size, are pivotal for success in the ARC domain.

Recent advancements, particularly OpenAI's o3 reasoning model, represent a significant leap forward. The o3 model achieved unprecedented performance on the ARC benchmark, with scores exceeding 90% on public evaluation sets under low-efficiency settings. Its incorporation of test-time program search and execution allows it to dynamically adapt to novel reasoning tasks, moving beyond the static pretraining approaches of earlier models. Despite these groundbreaking results, o3's high computational costs and time requirements present barriers to widespread deployment. The model's resource demands emphasize the need for optimization, especially as it achieves reduced performance on the more complex ARC2 benchmark.

These findings underscore the critical role of reasoning in solving abstract tasks, as well as the challenges of scalability and efficiency in deploying advanced reasoning models. Future work should aim to further refine hybrid architectures, optimize reasoning-centric designs like o3, and explore innovative datasets that enhance learning efficiency. Bridging these gaps will be crucial for advancing machine reasoning capabilities and paving the way toward Artificial General Intelligence (AGI).

Set	Tasks	Efficiency	Score	Retail Cost	Tokens	Time/Task
Semi-Private	100	High	75.7%	\$2,012	33M	1.3 min
Semi-Private	100	Low	87.5%	-	5.7B	13.8 min
Public	400	High	82.8%	\$6,677	111M	N/A
Public	400	Low	91.5%	-	9.5B	N/A

Table 4. Performance of the o3 reasoning model across different evaluation settings.

13. Acknowledgments

The authors would like to thank Professor Chih-Yuan Yang for their valuable feedback throughout the project. This work was supported by Chang Gung University's College of Intelligent Computing, and we appreciate Tanya Lim's help in managing the request. We extend special thanks to one of our founding members Kenji Chen for his contributions, particularly in the initial testing and validation of the autoencoder methodology, which provided a foundational understanding for our hybrid approach.