ARC Challenge

Chandrahas Aroori Abdelazim Lokma

November 13, 2024

Abstract

Our approach to solving the Abstraction and Reasoning Corpus (ARC) Challenge was to ensemble two methods with the Solvers built atop IceCubers solution. First, we used CodeIT which was a Reinforcement based LLM based solution. Post which we used a finetuned Instruct Llama 8B, which was trained on the Re-ARC.

1 Introduction

The Abstraction and Reasoning Corpus (ARC) presents a unique challenge in advancing artificial general intelligence (AGI) by requiring systems to generalize human-like reasoning across a diverse set of tasks. These tasks are open-ended, often ambiguous, and demand high levels of abstraction, making them a benchmark for testing the limits of computational intelligence.

Our solution to building on ARC benchmark is to use to ensemble improvements atop the Solvers solution[Kaz20] which already scored 26.

Given the shortcomings of current large language models (LLMs) on the ARC benchmark, we initially adopted Code Iteration (CodeIt)[BMW+24] as our approach to tackle this problem. CodeIt is a neuro-symbolic method that combines program synthesis with iterative self-improvement through hindsight relabeling and prioritized experience replay. This approach was promising because it allowed the model to learn from sparse rewards and adapt dynamically to new tasks by relabeling outputs to align with observed results. With pretraining and data augmentation, CodeIt achieved state-of-the-art performance on ARC, solving 15% of evaluation tasks—a significant improvement over existing neural and symbolic baselines, while also being light enough to run locally in the environment provided by Kaggle.

However we did not see a good result with CodeIt on the Kaggle runtime which lead us to shift to using a Llama model which we believed incorporating the Natural Language descriptions should provide us with a large accuracy. We believed by finetuning a Llama 8B model on multiple tasks, we should get a higher accuracy. To achieve this we used the Re-ARC[Hod24b] dataset which had multiple train data pairs based off of the original 420 training pairs provided by ARC.

2 Methods

2.1 CodeIt

Our first attempt was to use CodeIt[BMW+24] which leveraged an LLM with hindsight experience replay. We were originally inspired by the effectiveness of the winning solution from the previous ARC Challenge, which was submitted by IceCuber. Ice Cuber's approach was to use a Domain Specific Language, generating primitives that can apply basic transformations to the input grid, some examples include color manipulation and object detection. These primitives, while simple, are general enough that they can be chained together to solve a variety of different types of ARC tasks, allowing for the modularity that is necessary to potentially solve new and unseen tasks. However, we believed that the search component of IceCuber's solution required improvement.

2.1.1 Explaining CodeIT

Rather than mimicking Ice Cuber's fixed rule-based system and heuristic-driven search, the CodeIt approach relies on an L.L.M as a means of exploring the search space of possible DSL solvers, iteratively refining its ability to solve ARC tasks through reinforcement learning inspired techniques.

The CodeIt paper uses Hodel's DSL repository[Hod24a], which provides primitives, as well as solvers that, are designed to solve the 400 training tasks provided by the ARC Challenge team. However, the CodeIt research team first augmented this training set through program mutation. This mutation process involves randomly altering function calls, modifying parameters, and substituting functions in existing programs, in order to diversify the training set in a manner that still generates syntactically valid solvers. These mutated programs are then executed on the input grids of the original tasks to generate new, potentially different output grids. The mutated programs, along with the original input grids and the newly generated output grids, form a new "artificially generated" ARC task.

After the data augmentation stage, CodeIt researches use the CodeT5+ LLM, which is designed for code understanding and generation tasks, as the policy network to generate new DSL program solvers. In order to train the model, CodeIt researchers have defined two key stages, which together form what they refer to as a meta-iteration. This meta-iteration process allows the model to iteratively explore, evaluate, and learn from its generated programs, progressively enhancing its ability to solve ARC tasks. The two stages of a meta-iteration are the sampling stage, and the learning stage. Each meta-iteration allows the model to improve its ability to generate correct programs for solving ARC tasks.

1. The Sampling Stage:

In the sampling stage, new programs are obtained using the policy Q_θ (the CodeT5+ L.L.M, where θ represents the model parameters). If the search set is the set of tasks for which we want to find a corresponding program, then for each task in the search set, we convert the example inputs I (the initial grid configurations given for the task) and target outputs O^* (the desired grid transformations we aim to achieve) from grid format to text. These text representations of I and O^* are encoded using the L.L.M (Q_θ), which then decodes a program:

$$\rho \sim Q_\theta(\rho|I, O^*)$$

This decoded program ρ is then executed on the input grids I to generate an output grid O (the resulting grid transformation produced by ρ). If the output O is incorrect syntactically or runtime is too high, it is discarded. Otherwise the outputs $O = \rho(I)$ are obtained, and the following is added to the replay buffer: the program ρ, the inputs I and the realized outputs O, which may or may not match the target outputs.

The replay buffer is a memory bank that stores these generated experiences to improve the model's learning in the subsequent stage. In each sampling phase we repeat this process n_p times per task. Even if the realized outputs do not match the target outputs, the syntactically valid examples allow for the model to get a better grasp of the DSL syntax, or behavior of the problems.

2. The Learning Stage:

During this stage the policy Q_θ is trained on experiences sampled from the buffer, training set, and augmented set. These experiences consist of input grids I, output grids O and the program ρ. The training process uses a log loss function to maximize the likelihood of the program ρ given the input I and the output O.

$$\mathcal{L}(\rho, I, O) = -logQ_\theta(p|I, O)$$

Hindsight Experience Replay [AWR+17] is a technique that allows the policy to learn from failed experiences by converting them into valuable training data. H.E.R is applied to experiences in the replay buffer where the generated output O did not match the correct output O^* (but the output O was still syntactically valid). Since generating the correct output O^* is extremely unlikely, this results in an environment where rewards are sparse, giving few opportunities for the model to actually learn. This is where H.E.R plays a crucial role, as it effectively changes the reward structure. H.E.R treats each valid attempt as a partially successful experience, allowing the model to learn even when it doesn't reach the original target. Each new relabeled goal provides some experience on how to navigate the search space, as it provides information of the effects of certain actions (transformations, in the case of ARC tasks). This relabeling process also leverages prioritized sampling in order to give more weight to experiences that are more useful for training, the importance of experiences is measured by how closely the program output O matches the correct demonstration output $O*$.

2.1.2 Issues with the solution

Our intent was to use the CodeIt approach as one of the solution generators, and ensemble it together with the original solvers provided by the winning solution. However, we failed to do so due to the following reasons:

1. Despite using a light L.L.M model (CodeT5), the number of RL iterations required to find solutions were too high. Preventing us from running CodeIt in the Kaggle environment.
2. Aside from inability to get CodeIt running in Kaggle, the primitives provided by Hodel's DSL repository may not have been adequate enough to allow the CodeIt L.L.M to effectively search the possible solution space, had we added more primitives to the existing code base, we may have been able to improve the performance of CodeIt.

2.2 Llama 3 - 8B Fine-Tuned

2.2.1 ReARC dataset

The repository presents code to procedurally generate examples for the ARC training tasks. For each of the 400 tasks, an example generator is provided. This dataset is highly useful for fine-tuning a models because it is structured to maximize both variety and quality of examples.

Specifically, we used this dataset to produce 10 new pairs per example, resulting in a significantly enriched dataset. This augmentation enhances the diversity of training data, offering more varied patterns and scenarios for the model to learn from.

2.2.2 Methodology

We used LoRA to fine-tune our model. The model was trained using the ReARC data in batches for 100 epochs.

A. Ensembling Methodology

We modified the ensembling methodology provided by '2020 Winning Solution'[Kaz20], prioritizing the most frequently occurring solution as the first choice. Next, we gave precedence to IceCuber, followed by Solvers, and finally, LLaMA.

Order	Methodology Priority
1	Most frequently occurring solution
2	IceCuber
3	Solvers
4	LLaMA

Table 1: Modified Ensembling Methodology

B. LoRA Configuration

To adapt the pre-trained model to our specific task efficiently, we leveraged Low-Rank Adaptation (LoRA). LoRA reduces the computational overhead and memory requirements by introducing low-rank updates to the model's pre-trained weights. This allows for task-specific adaptation without requiring fine-tuning of all model parameters.

The configuration for low-rank adaptation (LoRA) includes a scaling factor (`lora_alpha`) of 64 to balance original and adapted weights, a dropout rate (`lora_dropout`) of 0.05 to prevent overfitting, and a rank (`r`) of 4 to control the size of the low-rank matrices and the number of additional parameters. Specific modules, such as `q_proj` and `k_proj`, were targeted to enhance key aspects of the model architecture for causal language modeling.

C. Training Configuration

The model was trained with Mixed precision training with FP16 was enabled to reduce memory usage and speed up computations. A small per-device batch size of 1 was used, with gradient accumulation over 8 steps to effectively simulate a larger batch size while managing memory constraints.

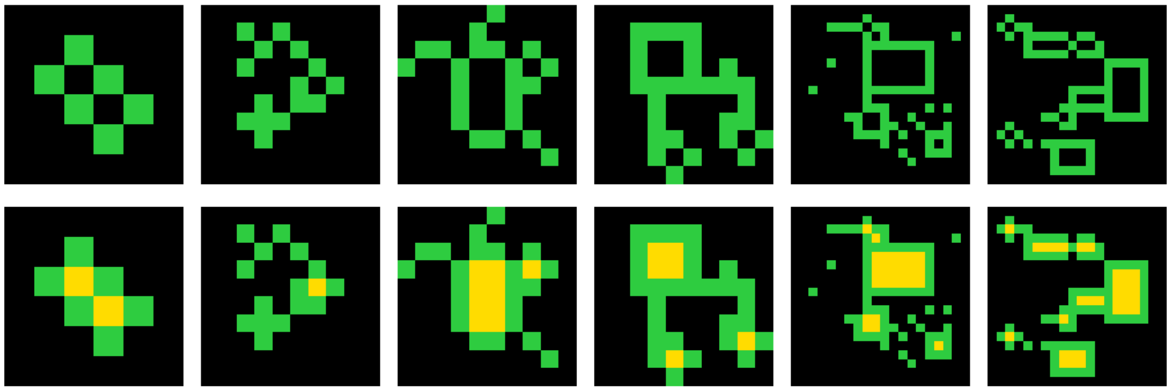

Figure 1: Original Problem

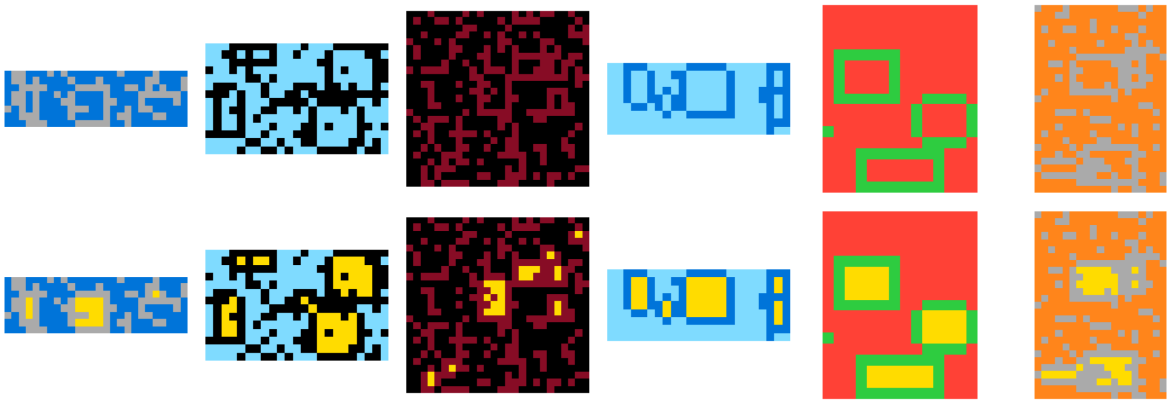

Figure 2: Generated Problems

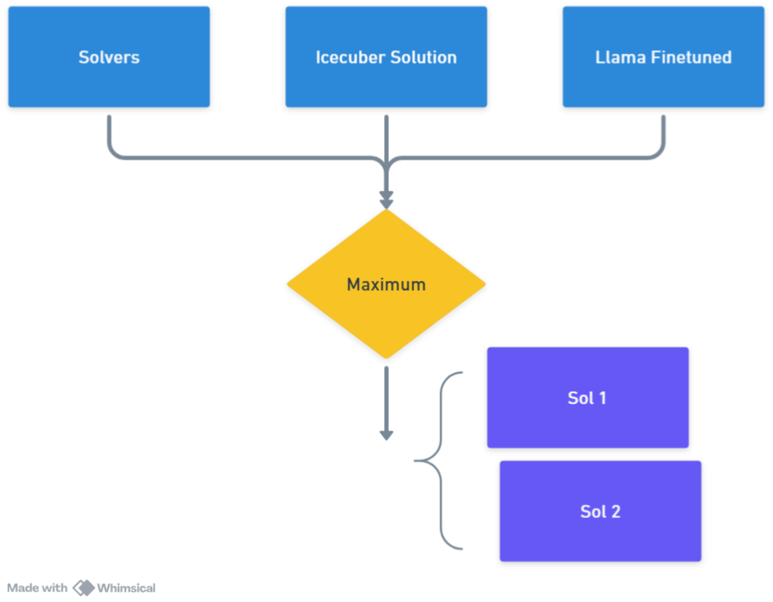

Figure 3: Algorithm used for ensembling

D. Inference Configuration

The fine-tuned model was optimized for inference by applying 4-bit quantization using the BitsAndBytesConfig configuration. This setup reduces memory usage and computational load while maintaining precision. To improve throughput, we also paired this with SDPA (scaled-dot product attention).

2.2.3 Prompts used

Finetuning & Inference

```
system_prompt = '''You are a puzzle solving wizard. You are given a
   puzzle from the abstraction and reasoning corpus'''

# User message template is a template for creating user prompts. It
   includes placeholders for training data and test input data,
   guiding the model to learn the rule and apply it to solve the given
    puzzle.
user_message_template = '''Here are the example input and output pairs
    from which you should learn the underlying rule to later predict
   the output for the given test input:
----------------------------------------
{training_data}
----------------------------------------
Now, solve the following puzzle based on its input grid by applying
   the rules you have learned from the training data.:
----------------------------------------
[{{'input': {input_test_data}, 'output': [[]]}}]
----------------------------------------
What is the output grid? Only provide the output grid in the form as
   in the example input and output pairs. Do not provide any
   additional information:'''
```

2.2.4 Issues with the solution

Our assumption was that fine-tuning the model would provide llama with the ability to correctly search for the solutions in the search space. However it seemed like Llama had issues solving harder questions.

1. Llama even with fine-tuning was only able to solve 4 questions which were already solved by '2020 Winning Solution'[Kaz20]

3 Results

We aimed to improve upon the previous winning solution for the ARC challenge by incorporating techniques from advanced implementations such as CodeIT and using a fine-tuned Llama model. These approaches showed great potential, and may still be worth using as inspiration for future ARC competitions, however, despite our efforts, we were unable to extract any additional performance gains when compared to the previous ARC Competition's winning solutions. Our final submission matched the baseline provided by the winning solution from the previous ARC competition but did not exceed it. Ultimately, we placed 608th on the leader board, with a score of 26 and 9 total entries overall. This outcome emphasizes the complexity of the ARC challenge and gives us insights on how we could approach solving the challenge in the future. In hindsight, we would have benefited greatly from focusing on solutions that were scalable and applicable in a competition scenario. Our use of Llama model could have been improved had we taken additional measures to improve its performance, such as applying active inference (as Francois Chollet, the creator of the ARC competition has suggested). Despite the challenges, this experience provided valuable insights into the strengths and limitations of current L.L.Ms and neuro-symbolic methods in abstract reasoning tasks.

References

[AWR+17] Marcin Andrychowicz, Filip Wolski, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, Pieter Abbeel, and Wojciech Zaremba. Hindsight experience replay, 2017.

[BMW+24] 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, 2024.

[Hod24a] Michael Hodel. arc-dsl: A domain-specific language for abstract reasoning. https://github.com/michaelhodel/arc-dsl, 2024. Accessed: 2024-11-13.

[Hod24b] Michael Hodel. re-arc: A flexible architecture for react applications. https://github.com/michaelhodel/re-arc, 2024. Accessed: 2024-11-13.

[Kaz20] Mehran Kazeminia. 3-arc24: Developed 2020 winning solutions. https://www.kaggle.com/code/mehrankazeminia/3-arc24-developed-2020-winning-solutions, 2020. Accessed: 2024-11-13.