Natural Language Enhanced Neural Program Synthesis for Abstract Reasoning Task

Stanford CS224N Custom Project

Yijia Wang
Department of Computer Science
Stanford University
yijawang@stanford.edu

Abstract

Abstract Reasoning Corpus (ARC) is a benchmark that measures general intelligence, through the task of inferring the shared pattern of a list of 2D grid pairs and predicting the output grid for novel inputs. The pretrained Large Language Models (LLMs) have shown impressive inductive reasoning capabilities through in-context learning on tasks such as commonsense, arithmetic, and symbolic reasoning, but show a large performance gap compared with humans on more complex reasoning tasks like ARC. In this project, we explore fine-tuning LLMs to solve the ARC tasks, and further enhancing the models by incorporating natural language guidance into the problem-solving process. Our experimental results demonstrate the effectiveness of natural language guidance in enhancing the language model's abstract reasoning capabilities, for both program-synthesis-based approaches and direct-prediction-based approaches. The program-synthesis-based approaches further show advantages in terms of higher interpretability, transferability, and scalability. Their predictive power increase as the number of generated programs scales up, and the generated programs are verifiable without needing to access test time target outputs.

1 Key Information to include

- Course mentor: Yuhui Zhang <yuhuiz@stanford.edu>
- External mentor: Ruocheng Wang <rcwang@stanford.edu>

2 Introduction

Abstract Reasoning Corpus (ARC) (Chollet, 2019) is a challenging visual inductive reasoning benchmark. Each ARC task comprises a list of input-output examples and one or more new inputs for which the outputs are to be predicted. The data for inputs and outputs are structured in arrays, with each numerical value corresponding to a specific color. An example of the ARC task is visualized in Figure 1. As we can see from the 5 demo input-output pairs, a common pattern is to fill the enclosed regions with yellow color, and the task is to apply this pattern to the novel input. The ARC tasks pose many challenges to the prevailing methods especially for neural models. Compared with traditional supervised learning settings with rich in-distribution label information, the ARC tasks are of the visual grid format with complex spacial patterns, which provides only limited demonstration information, and requires exact match for correctness on out-of-distribution tasks.

Among the current attempts to solve the ARC tasks, some approaches cover a subcategory of the ARC task with specific attributes. For example, Kolev et al. (2020) tackles the task with grid sizes 10×10 or smaller, Alford et al. (2022) focuses on symmetry tasks, and Xu et al. (2023) targets object-based tasks. These approaches are confined to a subset of the problem space. Among program-synthesis-based approaches, some works involve brute force search over customized Domain Specific Language (DSL) (ICECUBER, 2020; Ainooson et al., 2023), which requires intensive knowledge and work from human domain experts. Another avenue leverages the in-context learning capability of pretrained language models (Mirchandani et al., 2023; Wang et al., 2023a) and devises algorithms to

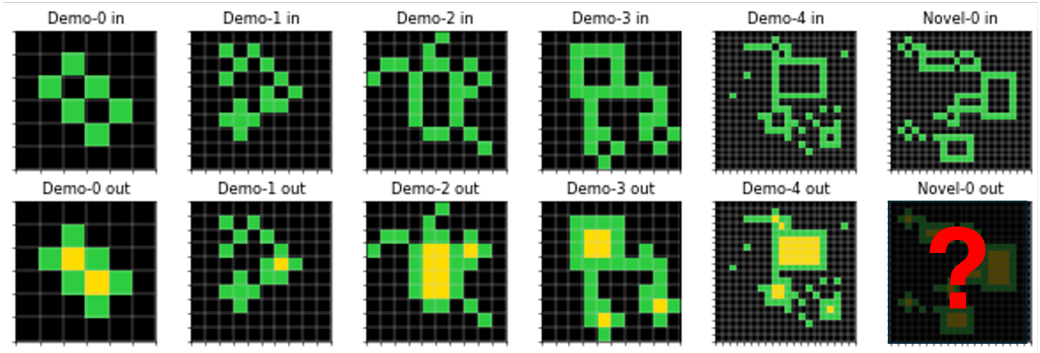


Figure 1: An example of ARC task: given a list of demo input-output pairs, the goal is to predict the output for one or more noval inputs.

solve ARC tasks in a tune-free manner. While these approaches reveal the inherent potential of LLM and benefit from obviating the need for additional training, they rely on highly intelligent language models which are often offered with high monetary costs, and with limited adaptability.

Inductive reasoning is a core part of human intelligence, and it is the main capability needed for solving ARC tasks. The human inductive reasoning process can be modeled as learning Bayesian posterior distribution over hypotheses from examples (Tenenbaum et al., 2006; Goodman et al., 2008). During this process, people often use more abstract hypothesis representation to guide the search for more specific ones (Goodman et al., 2011). Inspired by human inductive reasoning, we explore enhancing the capabilities of LLM by introducing high-level natural language guidance during the problem-solving process. At the same time, targeting a more cost-effective and task-adaptable solution, we fine-tune the open-source LLMs that offer greater accessibility and flexibility.

Our research indicates that incorporating natural language guidance enhances language models' abstract reasoning capabilities, for both direct-prediction and program-synthesis based methodologies. The latter, in particular, demonstrates higher interpretability, adaptability, and scalability, with its effectiveness increasing with the number of synthesized programs. These programs can be validated independently, obviating the need for test time target outputs.

The experimental findings demonstrate that task-specific fine-tuning significantly enhances LLM's capability to solve ARC tasks. In addition, the results underscore the vital influence of a rich and diverse training dataset in improving LLM's training efficiency and model capacity.

In subsequent research, we aim to create synthesized datasets with greater diversity by modifying multiple program lines. Additionally, we're interested in exploring prompting or tuning LLMs to generate natural language guidance along the problem-solving process. Moreover, we intend to investigate methodologies that maintain the prior knowledge of the pretrained model while effectively adapting to fine-tuning scenarios.

3 Related Work

3.1 Natural language enhanced abstract reasoning

Various works have applied natural language to the abstract reasoning process. Wang et al. (2023a) proposes to enhance the inductive reasoning capabilities of LLM by disentangling the inductive reasoning process into two parts: explicitly generating accurate natural language hypotheses, and formalizing the hypothesis as programs. Wang et al. (2024) proposes the symbol-to-language (S2L) method, which first converts the symbols to language-based representations, and then integrates these language-based representations into the original problem. Acquaviva et al. (2022) studies the relation and distinction between computer and natural programs and contributes a DSL-open and Language-complete ARC dataset, aiming at facilitating the development of more intelligent program synthesizers that bridge the gap between human-human and human-machine communications.

3.2 Program synthesis

The above works by Wang et al. (2023a) and Acquaviva et al. (2022) also draw a natural connection with program synthesis. Program synthesis shows advantages such as interpretability, generalizability, and efficiency, hence have been consistently explored to solve reasoning tasks. Butt et al. (2024) approaches ARC as a programming-by-examples problem, and introduces an iterative self-improvement method that iterates between program sampling and prioritized hindsight replay. Romera-Paredes et al. (2024) introduces an evolutionary algorithm to search for programs in the function space, and applies the technique to mathematical discoveries.

3.3 LLM for Code and parameter-efficient instruction fine-tuning

Many open-source Large Language Models for code such as CodeT5 (Wang et al., 2021), CodeT5+ (Wang et al., 2023b), and CodeLlama (Roziere et al., 2023) are pretrained on both general-purpose text and source code data. This dual-focus pre-training endows them with the potential to excel in both natural language understanding and code generation, making them uniquely powerful tools for tasks that bridge these two domains. Instruct tuning has been shown to effectively improve LLM's performance and generalizability, and there are many works exploring scaling up the model size and number of training tasks, such as Chung et al. (2022). However, our work focuses on fine-tuning medium-sized LLM, which are approachable by individuals or groups with limited computing resources. Multiple techniques are available to enable efficient model adapation, such as LoRA (Hu et al., 2021), Prefix-Tuning (Li and Liang, 2021), and Adaptor (Houlsby et al., 2019). In our work, we apply LoRA to reduce the number of trainable parameters, and enable conducting the fine-tuning on a single GPU machine.

4 Approach

We fine-tune Code Llama which is built on Llama 2 and further trained on programming tasks. Specifically, we select the 7B - Instruct variant in the Code Llama family, which is fine-tuned to better follow instructions in code-related tasks.

In our experiments, we employ the Domain Specific Language (DSL) developed by Hodel (2023) to solve the ARC tasks. Different from the direct-prediction-based approach where the LLM generates the output grid directly, the LLM will generate series of atomic operations under the program-synthesis-based settings. Some examples of DSL atomic transformations include: flip, repeat, colorfilter, etc.. By applying the DSL program on the task inputs, we will obtain the output grid. An illustration of example DSL programs and corresponding visualizations is shown in Figure 4.

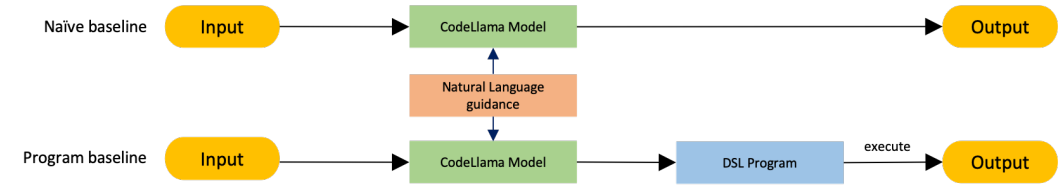
Baseline approaches: We implement 2 baseline approaches, and enhance each of them by adding natural language guidance to the problem-solving process. We will refer to our natural language enhanced methods as nl-naive and nl-program respectively. The two baseline approaches are described below:

- Naive baseline: predict output directly based on original inputs.
- Program baseline: predict testable hypothesis in DSL, and verify correctness by executing the program in Python.

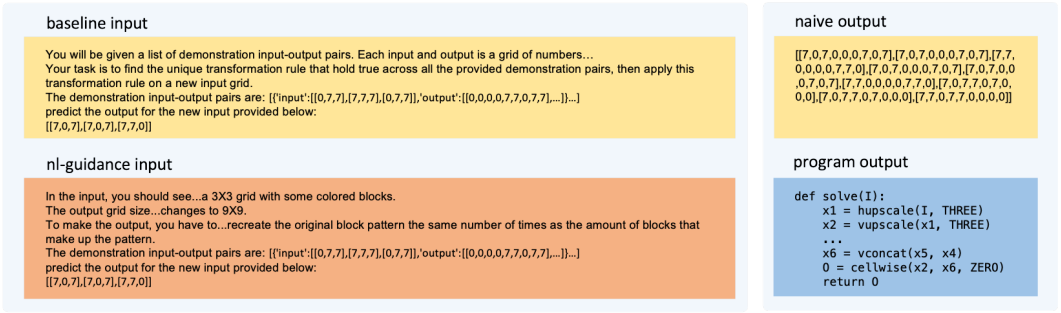
Model inputs and outputs: In the training stage, we construct prompts by concatenating the following 4 components as the model input: (1) the general ARC task description proposed by Wang et al. (2023a) (see Appendix A) (2) demonstration input-output pairs (3) the new test input to be predicted on (4) the target output. The fine-tuning tasks are defined as predicting the next token of the input sequence. In the evaluation mode, we exclude the target output from the input sequence. When generating sequence, the training employs teacher-forcing (Williams and Zipser, 1989), while the post-training evaluation does not.

Proposed approaches: All 4 approaches use the same training pipeline by only varying the inputs and outputs. Specifically, the baseline approach uses the original output array in text format as output, and the program baseline uses the hypothesis defined in DSL developed by Hodel (2023) as output. In the natural language guidance enhanced approaches, we replace the original general ARC task description in the original model prompt inputs with the task-specific natural language description. We compare the model performances among the baseline approaches and the enhanced approaches, to assess the impacts of natural language guidance.

The pipelines of the baseline methods and the natural-language-guidance-enhanced approaches are illustrated in Figure 2.



(a)



(b)

Figure 2: (a) The pipeline of the naive baseline, program baseline, and the natural-language-guidance-enhanced approaches. (b) The example input and output formats. We experiment with 4 setups: (1) naive baseline setup takes inputs in the baseline input format and generates predictions in the naive output format; (2) the program baseline setup takes inputs in the baseline input format and generates predictions in the program output format; (3) the nl-naive setup takes inputs in the nl-guidance input format and generates predictions in the naive output format; (4) the nl-program setup takes inputs in the nl-guidance inputs format and generates predictions in the program output format.

5 Experiments

In this section we present the details of our experiments, including data, training framework and hyperparameters, evaluation metric and method, and experimental results.

5.1 Data

Datasets. We employ ARC (Chollet, 2019) data to train and evaluate the proposed methods. The ARC dataset contains 400 training tasks, 400 evaluation tasks, 100 public test tasks, and 100 private test tasks. For natural language guidance augmentations, we utilize the LARC dataset introduced by Acquaviva et al. (2022). This dataset enhances the original ARC tasks with language-complete instructions that enable an unambiguous understanding of the desired outcomes of instructions without needing to refer to extra contexts or the original input-output examples. The LARC dataset covers the 400 training tasks of the original ARC dataset.

Data processing. We select the set of tasks that are common to ARC and LARC, and modify them to accommodate the GPU memory restriction. Specifically, if the model input sequence for a task contains more than 4096 tokens, we iteratively remove one demonstration pair, until it fits into the maximum token length or has less than 2 demonstration pairs remaining. We remove the tasks that can't fit into the sequence length limit with at least 2 demonstration pairs. If a task contains multiple new test cases, we divide them into separate tasks. We divide the dataset into 2 splits due to limited data, with training and test splits by a 3:1 ratio. With the above preprocessing, we obtain 384 tasks, with 285 tasks for training and 99 tasks for testing.

Data augmentation. To enable the model to better learn DSL knowledge, we further augment the training dataset by randomly mutating one line from the provided reference program, as proposed by Butt et al. (2024), and use the generated new program to synthesize new outputs. Specifically, we mutate the line in 3 possible ways: (1) replace the input arguments; (2) replace the function call with another function with the same input variables; (3) replace the function call with another function with similar output type. We further conduct a series of post-processing to (1) remove duplicated contents; (2) remove unused lines after mutation; and (3) remove invalid functions that are non-executable or yield results of invalid shapes or types. We construct an augmented dataset by sampling 10,000 synthesized tasks, ensuring that each original task contributes an approximately equal number of new tasks to maintain balance and diversity in the dataset.

We denote the 285 tasks sampled from the original training dataset as dataset-orig, and the 10,000 synthesized tasks as dataset-aug in the below sections to facilitate representations.

We train 4 program-synthesis-based models using the above training datasets:

1. program-baseline: fine-tuned on dataset-orig, with general ARC task description as input, initialized with the original CodeLlama-7B-Instruct model weights.
2. nl-program: fine-tuned on dataset-orig, with task-specific natural language guidance as input, initialized with the original CodeLlama-7B-Instruct model weights.
3. program-aug: program baseline model initialized with the original CodeLlama-7B-Instruct model weights and further trained on dataset-aug. In our representation, we denote this procedure as pretraining, compared with the downstream fine-tuning on the smaller dataset data-orig.
4. nl-program-aug: fine-tuned on data-orig, with task-specific natural language guidance as input, initialized with the model weights obtained from model program-aug.

5.2 Evaluation metric

We use the prediction accuracy on the test dataset to measure the models' performances. Specifically, the prediction of a task is regarded as correct only if it exactly matches the ground truth. For the direct-prediction-based approach, we compare the predicted array with the answer, and for the program-synthesis-based approach, we execute the program and compare the obtained array with the answer. By the evaluation protocol of ARC, we select 3 predictions for each task and regard a task as successfully solved if any of the predictions match the answer. For the program-synthesis-based approaches, we generate 3 or more programs and follow the heuristic rules proposed by Butt et al. (2024), to select 3 predictions by prioritizing the programs that predict demonstration pair with higher accuracy, if given equal accuracy we prioritize the programs with less number of lines. Since there are 99 test tasks in total, without losing precision, we round the accuracy percent value to the nearest integer to simplify the notation.

5.3 Experimental details

Framework and hyperparameters. We use the deepspeed framework to accelerate training, and apply int8 training techniques to reduce memory consumption. We access these techniques through the integrated functionalities in the transformer library. We apply LoRA (Hu et al., 2021) using the peft library to reduce the trainable parameters to around 25% of the full parameter size. We use lora_alpha 16, lora_dropout 0.05 and r 16. We use AdamW optimizer (Loshchilov and Hutter, 2017) with WarmupDecayLR scheduler, with warm up steps 30 for pretraining task and 10 warmup steps for fine-tuning tasks. Some other key hyperparameters include: learning rate 3e-4, batch size 1, gradient accumulation steps 16. In sequence generation we employ top-p sampling strategy (Holtzman et al., 2019) with top probability 0.95 and temperature 0.8. Given the limitation in GPU memory capacity, we set the maximum sequence length as 4096 in training and 3000 in the evaluation, with truncation but no padding. We further restrict the maximum new tokens to be generated as 1024. Among all the 99 test tasks, 15 tasks contain demonstration pairs beyond the maximum sequence length limit.

Evaluation. When evaluating naive-based approaches, we generate 3 predicted arrays in text format. When evaluating program-synthesis-based approaches, we select checkpoint by generating 3 programs and calculate the prediction accuracy on the test dataset, then further evaluate the capacity of the selected best checkpoint by generating 72 programs. We calculate the prediction accuracy on the test tasks based on 3 heuristically selected programs. Constrained by GPU memory capacity, the 72 programs are generated by 12 separate runs, with 6 generated programs per run.

For the naive baseline, nl-naive, program baseline and nl-program methods, we train for 100 epochs, and evaluate the checkpoints obtained at the end of epochs 2, 5, 10, 20, 30, 40, 50, 70, 100. The best epochs for the above 4 setups are respectively 5, 70, 40, 50. For the program-aug method we pretrain the model on the dataset-aug for 7 epochs, and we further fine-tune on the data-orig for another 100 epochs. We follow the same procedure as above to evaluate models for finetune setup, and evaluate the models at the end of each epoch for pretrain setup. We select the best checkpoint based on the prediction accuracy on the test dataset with 3 generations. The best epochs for these 2 setups are respectively 5 and 2.

5.4 Results

In this section, we report the main experimental results and findings.

	naive	program	program-augment
baseline	6%	5%	8%
language-guidance	8% (↑ 2%)	3% (↓ 2%)	9% (↑ 1%)

Table 1: Comparison of the prediction accuracy on the test dataset for each method. The values in the table are percentages, i.e. n represents the pass rate of $n\%$. The initial CodeLlama-7B-Instruct without task-specific fine-tuning solves 2% tasks under the naive setup, and solves 0% tasks under the program-based setup.

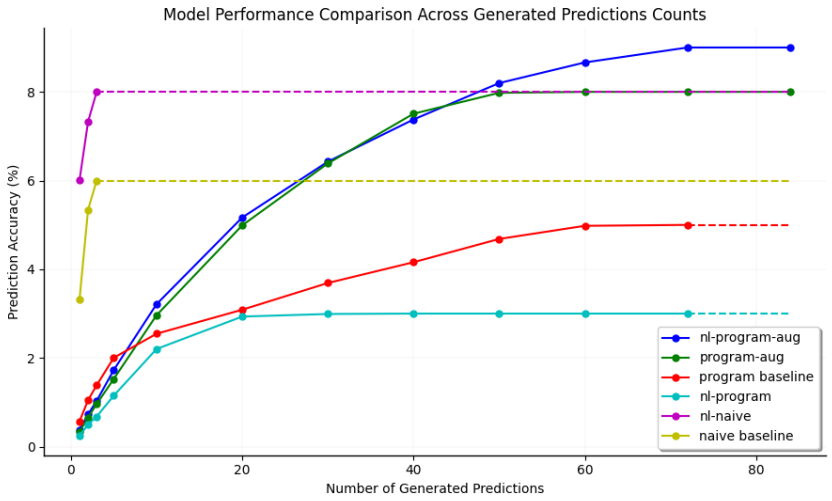


Figure 3: The relation between the prediction accuracy and the number of generated predictions. We use dash lines to depict the cases beyond the actual total number of generated predictions. By ARC evaluation protocol, only a maximum of 3 attempts on each test task are allowed. Hence the direct-prediction-based approaches, i.e. naive baseline and nl-naive, are allowed to only generate at most 3 predictions. Since program-based approaches can verify generated programs on demo pairs and heuristically select 3 programs for test submission, they are allowed to generate more than 3 programs.

Performances comparison among different methods. In Table 1, we compare the prediction accuracies of each method. The naive column refers to the setup to directly predict output array; the program column refers to the setup to predict DSL program, trained on dataset-orig; the program-aug column refers to the setup to predict DSL program, trained on the dataset-aug. The baseline row refers to the setup where it takes the general ARC task description as input; the nl-guidance row refers to the setup where it takes task-specific natural language guidance as input. We report the prediction accuracies of the best checkpoint of each method. Specifically, we select checkpoints by prediction accuracy on the test dataset with 3 generations, and further evaluate program-based approaches with 72 generations. Since the program-aug and nl-program-aug show equal accuracy in our experiment which is 8%, we increase the number of generations for these 2 models by 12, to a total of 84 to better compare their capacity. From the table we can see that, the fine-tuning improves the model's task-solving capability, specifically, compared with before fine-tuning, after fine-tuning the number of correct predictions under the naive-based setup is increased from 2 to 6, and under the program-based setup from 0 to 5. The nl-naive method outperforms the naive baseline by 2, the nl-program-aug method outperforms the program-aug method by 1, but the nl-program method underperforms the program baseline by 1. A detailed summary of the successfully solved test task indexes by each method is shown in Appendix B.

Relation between prediction accuracy and the number of generated predictionss. In Figure 3 we show the relationship between the prediction accuracy on the test set and the number of generated predictions. We interpolate the numbers of generated predictions that are less than the total number of generated predictions, by sampling a subset of predictions with the given size and calculating the accuracy. We repeat the sampling 1000 times and report the average prediction accuracy for

each specified number of generated predictions. From the plot, we can see that more generated predictions tend to solve more tasks correctly. However, the marginal gain decays as the number of generated predictions increases, and performance eventually converges. Without pretraining on a larger synthesized dataset, the natural language guidance enhanced method underperforms the baseline method. However, after pretraining on the larger dataset, the natural language guidance enhanced method eventually surpasses the corresponding baseline method. Furthermore, the pretraining brings significant improvements in performances, for both the natural language guidance enhanced approach and the baseline approach, increasing the prediction accuracies from 3% to 9% and 5% to 8% respectively.

6 Analysis

In this section, we discuss the experimental results reported in Section 5.4 and summarize the findings.

Natural language guidance helps. The natural language guidances are shown to improve the performance of the baseline methods in 2 out of 3 setups, which are the naive setup and program-aug setup, demonstrating the effectiveness of natural language guidance in enhancing LLM's reasoning capacities. However, under data-scarse setting, we observe that the small data size limits program-based methods more than naive-based methods, and limits the nl-based methods more than non-nl-based methods. We hypothesize this is potentially because, the models need a larger corpus and more sufficient training to fully grasp domain-specific language under the program-based setup than to understand the general natural language under naive-based setup; and establishing the connection between the demonstrations with task-specific natural language under nl-based settings, takes a more delicate learning process, than to build connections with general task descriptions under the non-nl-based settings.

The above hypothesis can be partially verified by the significant performance improvement introduced by pretraining on the larger synthesized dataset. Specifically, it improves both the program-based settings, i.e. program baseline from 5% to 8%, and nl-program approach from 3% to 9%. Furthermore, it bridges the negative gap between nl-program and program methods, and unleashes more potential for the nl-based approach.

However, to solidify the connection between task-specific natural language guidance and the DSL, we need more information from both these 2 spaces. The training dataset augmentation only introduces more knowledge from the DSL domain. Since the task-specific natural language generation takes nontrivial intelligence from human or language models, and can not be easily synthesized based on the resources that are accessible to us currently, the models still have only limited information from the task-specific natural language space. This might explain the marginal performance improvement from program-aug to nl-program-aug.

Program-based approaches are interpretable, transferable, and scalable. Compared with the direct-prediction-based approach, the program-based approach generates solutions that are interpretable by human experts, transferable to new problems, and verifiable without needing to access test time target outputs. In our case, the program solutions are often more succinct than the direct predictions, and the correct programs generated by our models are often more succinct than human-curated DSL solutions. Moreover, it shows increasing predictive power when the number of generated programs scales up. The marginal gains from the increasing number of programs decay and the realized capacity converge eventually. In our experiments, the performance overall saturates more slowly with respect to the number of programs for the models that are trained on more diverse datasets and embed a higher volume of information. Furthermore, the generation and evaluation of the programs are highly parallelizable and scalable, hence can be potentially largely accelerated by distributed systems.

Successful and failed cases. In Figure 4 we show an example of correct and incorrect prediction respectively. In the successful case, the model solves a non-trivial task that requires drawing vertical and horizontal lines between pairs of squares on a canvas with densely distributed noisy squares. Furthermore, the model solves the task with 2 different predicted programs. Besides a prediction that coincides with the reference solution, the model also discovers a new program that is more succinct than the human-curated reference solution. In the failed case, the model mistakenly swaps vertical (v) and horizontal (h) commands, where it applies vmirror and hconcat when it should use hmirror and vconcat, resulting in the shapes in the output displayed in a rotated direction.

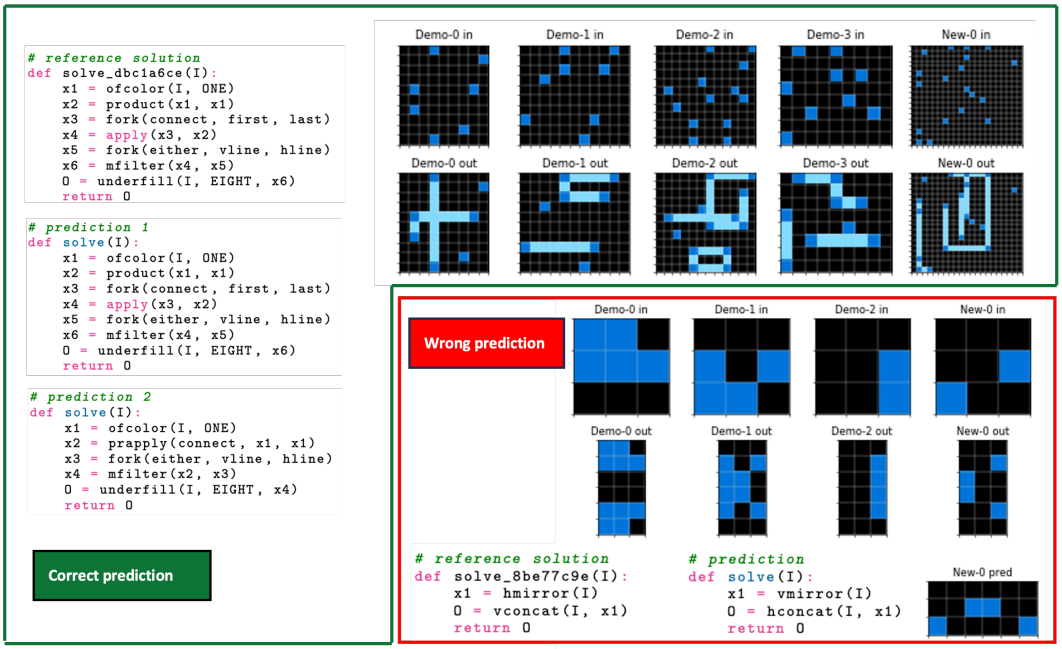


Figure 4: Examples of correct and wrong predictions. The content enclosed in the green border is an example of correct prediction. This is a non-trivial task, which requires line drawing operations between pairs of squares on a canvas with densely distributed squares. Furthermore, the new input and output pair is of large canvas size. The model generated 2 different programs that could both predict final outputs correctly. One of the generated programs is identical to the reference solution. Other than that, the model also discovers another new solution that is more succinct than the human-curated reference solution. The content enclosed in the red border is an example of a wrong prediction. Note that since the predicted program incorrectly used vmirror and hconcat when it should use hmirror and vconcat, the shapes in the generated output show in a rotated direction.

Error analysis. Our models typically produce correct solutions with relatively concise transformations, unlike incorrect ones that often become overly lengthy, sometimes unreasonably repeating certain transformations. Imposing a cap on the number of new tokens generated can effectively filter out likely incorrect predictions and conserve computational resources during generation time. Despite some successfully solved tasks having long demonstrations, the model struggles more with increasing sequence lengths, likely related to the maximum sequence length constraint during training and evaluation, imposed by GPU memory limits. However, the inherent complexity of longer tasks, demanding more sophisticated reasoning, might be the real challenge. We think that accessing to more powerful machines that could handle longer sequence lengths, coupled with more sufficient training on larger high-quality datasets would likely enhance the model's performance on these complex tasks.

7 Conclusion

In this project, we fine-tune CodeLlama-7B-Instruct model with the natural-language-guided program synthesis to solve ARC tasks. Our experimental results show that, fine-tuning on the ARC dataset effectively improves the model's ability to solve the ARC tasks, and natural language guidance further enhances the model's reasoning ability. The program-synthesis-based approaches demonstrate advantages such as interpretability, transferability, and scalability. A larger dataset with reasonable quality and diversity is demonstrated to significantly contribute to more efficient learning and higher model capacity. In future studies, we would like to explore curating synthesized datasets with even higher diversity by increasing the number of program lines to be mutated, and to explore prompting or tuning LLMs to generate natural language guidances along the problem-solving process. We would also like to explore techniques that efficiently adapt to fine-tuning tasks while better preserving the prior knowledge from pretraining. In terms of the ARC tasks, another interesting avenue is to combine the LLM with searching and optimization algorithms, and develop iterative approaches to achieve multi-layer and multi-step abstract reasoning.

References

Sam Acquaviva, Yewen Pu, Marta Kryven, Theodoros Sechopoulos, Catherine Wong, Gabrielle Ecanow, Maxwell Nye, Michael Tessler, and Josh Tenenbaum. 2022. Communicating natural programs to humans and machines. *Advances in Neural Information Processing Systems*, 35:3731–3743.

James Ainooson, Deepayan Sanyal, Joel P Michelson, Yuan Yang, and Maithilee Kunda. 2023. An approach for solving tasks on the abstract reasoning corpus. *arXiv e-prints*, pages arXiv–2302.

Simon Alford, Anshula Gandhi, Akshay Rangamani, Andrzej Banburski, Tony Wang, Sylee Dandekar, John Chin, Tomaso Poggio, and Peter Chin. 2022. Neural-guided, bidirectional program search for abstraction and reasoning. In *Complex Networks & Their Applications X: Volume 1, Proceedings of the Tenth International Conference on Complex Networks and Their Applications COMPLEX NETWORKS 2021 10*, pages 657–668. Springer.

Natasha Butt, Blazej Manczak, Auke Wiggers, Corrado Rainone, David Zhang, Michaël Defferrard, and Taco Cohen. 2024. Codeit: Self-improving language models with prioritized hindsight replay. *arXiv preprint arXiv:2402.04858*.

François Chollet. 2019. On the measure of intelligence. *arXiv preprint arXiv:1911.01547*.

Hyung Won Chung, Le Hou, Shayne Longpre, Barret Zoph, Yi Tay, William Fedus, Yunxuan Li, Xuezhi Wang, Mostafa Dehghani, Siddhartha Brahma, et al. 2022. Scaling instruction-finetuned language models. *arXiv preprint arXiv:2210.11416*.

Noah D Goodman, Joshua B Tenenbaum, Jacob Feldman, and Thomas L Griffiths. 2008. A rational analysis of rule-based concept learning. *Cognitive science*, 32(1):108–154.

Noah D Goodman, Tomer D Ullman, and Joshua B Tenenbaum. 2011. Learning a theory of causality. *Psychological review*, 118(1):110.

Michael Hodel. 2023. *Domain-Specific Language for the Abstraction and Reasoning Corpus*. Available at `https://github.com/michaelhodel/arc-dsl/tree/main`, version hash code 3071b3a7ac142affd5fa6d087391f23a8179e76e.

Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. 2019. The curious case of neural text degeneration. *arXiv preprint arXiv:1904.09751*.

Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin De Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. 2019. Parameter-efficient transfer learning for nlp. In *International conference on machine learning*, pages 2790–2799. PMLR.

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

ICECUBER. 2020. Arc kaggle competition winner solution - icecuber. Accessed on 03-13-2024.

Victor Kolev, Bogdan Georgiev, and Svetlin Penkov. 2020. Neural abstract reasoner. *arXiv preprint arXiv:2011.09860*.

Xiang Lisa Li and Percy Liang. 2021. Prefix-tuning: Optimizing continuous prompts for generation. *arXiv preprint arXiv:2101.00190*.

Ilya Loshchilov and Frank Hutter. 2017. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*.

Suvir Mirchandani, Fei Xia, Pete Florence, Brian Ichter, Danny Driess, Montserrat Gonzalez Arenas, Kanishka Rao, Dorsa Sadigh, and Andy Zeng. 2023. Large language models as general pattern machines. *arXiv preprint arXiv:2307.04721*.

Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M Pawan Kumar, Emilien Dupont, Francisco JR Ruiz, Jordan S Ellenberg, Pengming Wang, Omar Fawzi, et al. 2024. Mathematical discoveries from program search with large language models. *Nature*, 625(7995):468–475.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

Joshua B Tenenbaum, Thomas L Griffiths, and Charles Kemp. 2006. Theory-based bayesian models of inductive learning and reasoning. *Trends in cognitive sciences*, 10(7):309–318.

Ruocheng Wang, Eric Zelikman, Gabriel Poesia, Yewen Pu, Nick Haber, and Noah D Goodman. 2023a. Hypothesis search: Inductive reasoning with language models. *arXiv preprint arXiv:2309.05660*.

Yile Wang, Sijie Cheng, Zixin Sun, Peng Li, and Yang Liu. 2024. Speak it out: Solving symbol-related problems with symbol-to-language conversion for language models. *arXiv preprint arXiv:2401.11725*.

Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi DQ Bui, Junnan Li, and Steven CH Hoi. 2023b. Codet5+: Open code large language models for code understanding and generation. *arXiv preprint arXiv:2305.07922*.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859*.

Ronald J Williams and David Zipser. 1989. A learning algorithm for continually running fully recurrent neural networks. *Neural computation*, 1(2):270–280.

Yudong Xu, Elias B Khalil, and Scott Sanner. 2023. Graphs, constraints, and search for the abstraction and reasoning corpus. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 37, pages 4115–4122.

A Prompt template for general ARC task description

```
# reference prompt for naive-based approaches
# demo pairs at the end
# adapted from Appendix B.2, Wang, Ruocheng, et al. "Hypothesis
    search: Inductive reasoning with language models."

INSTRUCTION_NAIVE_END = """
You will be given a list of demonstration input-output pairs.
   Each input and output is a grid of numbers representing a
   visual grid. There is a SINGLE pattern that transforms each
    input grid to the corresponding output grid. Your task is
   to find the unique transformation rule that hold true
   across all the provided demonstration pairs, then apply
   this transformation rule on a new input grid, to predict
   the resulting output. The pattern may involve counting or
   sorting objects (e.g. sorting by size), comparing numbers (
   e.g. which shape or symbol appears the most? Which is the
   largest object? Which objects are the same size?), or
   repeating a pattern for a fixed number of time.
There are other concepts that may be relevant.
- Lines, rectangular shapes
- Symmetries rotations, translations.
- Shape upscaling or downscaling, elastic distortions.
- Containing / being contained / being inside or outside of a
   perimeter.
- Drawing lines, connecting points, orthogonal projections.
- Copying, repeating objects.
You should treat black cells as empty cells (backgrounds).
The number in the input grid can be mapped to the following
   colors: 0:black; 1:blue; 2:red; 3:green;
4:yellow; 5:grey; 6:fuschia; 7:orange; 8:teal; 9:blinen
The demonstration input-output pairs are:
{demo_pairs}
Given the unique and consistent transformation rule derived
   from the above demonstration pairs,
predict the output for the new input provided below:
{new_input}
"""
```

B The summary of correct predictions by each model

test_index	naive_baseline	nl_naive	program_baseline	nl_program	program_aug	nl_program_aug
0		√				
5					√	
8					√	√
15						√
16			√			
21	√					
22		√	√	√	√	√
26		√				
27		√			√	√
36	√					
37		√				
43	√	√	√		√	√
45						√
51	√					
57		√				
59	√					
67				√	√	√
68			√			
70			√	√	√	√
73					√	
83						√
97	√	√				