The Thirty-Eighth AAAI Conference on Artificial Intelligence (AAAI-24)

Frugal LMs Trained to Invoke Symbolic Solvers Achieve Parameter-Efficient Arithmetic Reasoning

Subhabrata Dutta[*1], **Ishan Pandey**[*2], **Joykirat Singh**[*2],
Sunny Manchanda[3], **Soumen Chakrabarti**[4], **Tanmoy Chakraborty**[1]
[1]IIT Delhi, India
[2]IIIT Delhi, India
[3]DYSL-AI, India
[4]IIT Bombay, India
subha0009@gmail.com, joykirat19166@iiitd.ac.in, ishan20304@iiitd.ac.in, manchanda.sunny@gmail.com,
soumen@cse.iitb.ac.in, tanchak@ee.iitd.ac.in

Abstract

Large Language Models (LLM) exhibit zero-shot mathematical reasoning capacity as a behavior emergent with scale, commonly manifesting as chain-of-thoughts (CoT) reasoning. However, multiple empirical findings suggest that this prowess is exclusive to LLMs with exorbitant sizes (beyond 50 billion parameters). Meanwhile, educational neuroscientists suggest that symbolic algebraic manipulation be introduced around the same time as arithmetic word problems to modularize language-to-formulation, symbolic manipulation of the formulation, and endgame arithmetic. In this paper, we start with the hypothesis that much smaller LMs, which are weak at multi-step reasoning, can achieve reasonable arithmetic reasoning if arithmetic word problems are posed as a formalize-then-solve task. In our architecture, which we call SyReLM, the LM serves the role of a translator to map natural language arithmetic questions into a formal language (FL) description. A symbolic solver then evaluates the FL expression to obtain the answer. A small frozen LM, equipped with an efficient low-rank adapter, is capable of generating FL expressions that incorporate natural language descriptions of the arithmetic problem (e.g., variable names and their purposes, formal expressions combining variables, etc.). We adopt policy-gradient reinforcement learning to train the adapted LM, informed by the non-differentiable symbolic solver. This marks a sharp departure from the recent development in tool-augmented LLMs, in which the external tools (e.g., calculator, Web search, etc.) are essentially detached from the learning phase of the LM. SyReLM shows massive improvements (e.g., $+30.65$ absolute point improvement in accuracy on the SVAMP dataset using GPT-J 6B model) over base LMs, while keeping our testbed easy to diagnose, interpret and within reach of most researchers.

1 Introduction

Large Language Models (LLMs) trained on giant corpora of text, code, reasoning chains, and dialogues have recently taken centerstage in not only most NLP tasks but also arithmetic and logical reasoning (Brown et al. 2020; Kojima et al. 2022). Despite major recent strides, LLMs, on their own, are resource-intensive, inefficient and unreliable devices for many rigorous tasks such as arithmetic, logic, calculus, or geometry. Indeed, one should not expect word sequence inputs and outputs to capture such domains and tasks. Moreover, folding such 'skills' into an opaque, massive LM resists diagnosis and interpretability. Therefore, LLMs are also being trained to invoke *tools* that can perform such tasks as subroutines (Paranjape et al. 2023; Schick et al. 2023; Wolfram 2023), with the LLM acting as a 'glue' between user utterances and these tools.

Here, we focus on the interplay between symbolic and arithmetic reasoning tasks mediated by an LM. Educational neuroscientists have long detected a stronger correlation between dysfunctional word problem-solving capability and symbolic algebraic reasoning than between the former and arithmetic calculation abilities (Nathan, Kintsch, and Young 1992; Powell and Fuchs 2014). In light of fMRI evidence, this may not be surprising that different areas of the brain (Mahowald et al. 2023, §2.2.1, §2.2.2) are responsible for linguistic, analytical and logical processing. A common pedagogic trick employed by grade school teachers is to repeat arithmetic word problems with diverse numerals to see if a pupil understands the problem at the level of symbolic reasoning, as against numeric shortcuts. Beyond grade school, the correctness of SQL queries is checked by weeding out incorrect *mutant* queries using adversarially engineered database instances (Chandra et al. 2015).

The above discussion naturally suggests that an LM that can call an arithmetic calculator tool to solve word problems can perform better (perhaps even with a smaller model size) if it also learns to translate the word problem to a logical form in a *formal language* (FL) and a *symbolic solver* to manipulate the FL specification. After all, the recipe often given to school children to solve word problems goes roughly like this (paraphrased for the current audience):

1. Read the word problem to allocate variables to quantities mentioned therein.
2. Parse the text to bind some subset of variables to constants grounded in the problem statement.
3. Further parse the text to extract arithmetic relationships

[*]These authors contributed equally.

and constraints between variables.
4. Identify the target variable(s) whose ground value(s) will answer the question.
5. Invoke a symbolic solver to express the target variable(s) in terms of grounded variables.
6. Invoke an arithmetic calculator/interpreter to obtain values for target unknown variables.

Figure 1 shows an extended example of the above steps in action. Dysfunction in solving word problems (such as sometimes evident in even powerful LLMs, as well as grade-school students) is often traced to a failure in one or more of the above steps. This naturally suggests that we attempt to apply grade-school pedagogy tricks to tool-using LMs.

Our Contributions. Early LM-based word problem solvers (Wei et al. 2022; Chowdhery et al. 2022) employed LM to manage *all* the above steps; however, recent LLM-based systems (Paranjape et al. 2023; Schick et al. 2023) are taught to invoke tools. Instead of burdening a gigantic LLM with language understanding, symbolic processing, and proper invocation of an arithmetic calculator tool, we propose SYRELM, a system where the LM serves the role of a translator from natural language arithmetic questions into a formal language description, and then invokes a symbolic solver to evaluate the formal language expression to obtain the answer. In contrast to gigantic LLMs that are beyond many researchers' capability to train or finetune, we use a frozen LM of modest size with a low-rank adapter (Hu et al. 2021). We adopt policy-gradient reinforcement learning to train the adapted LM, along with the non-differentiable symbolic solver. This marks a sharp departure from the recent development in tool-augmented LLMs, in which the external tools are essentially detached from the learning phase of the LM. SYRELM shows strong improvements over base LMs, while keeping our testbed easy to diagnose, interpret and within the reach of most researchers. While still far short of LLMs trained with huge corpora and human feedback (OpenAI 2023; Zheng et al. 2023), SYRELM demonstrates strong performance boosts and establishes the validity of middle-school math pedagogy for AI pupils. On the SVAMP arithmetic reasoning dataset, SYRELM improves upon the base k-shot accuracy of GPT-J 6 billion (Wang and Komatsuzaki 2021) (Vicuna 13 billion[1]) by 31.6 (27.12) absolute points, outshining recent methods like Toolformer (Schick et al. 2023) and ART (Paranjape et al. 2023) by large margins. On SVAMP and GSM8K datasets, SYRELM makes Vicuna 13B perform comparable to GPT-3.5 (standard 8-shot prompting). We release the code and data for reproducibility [2].

2 Related Work

Given their original purpose of modelling word correlations, early LMs like BERT (Devlin et al. 2019) were found astonishingly capable of arithmetic but within restricted ranges (Wallace et al. 2019). Middle-school teachers expect students to solve word problems no matter whether John gave Jane 12 pencils or 13,625,018. However, LLMs like T5 and GPT-3 were getting so good at covering common benchmarks like GSM8K (Cobbe et al. 2021) — and even performing addition (Zhou et al. 2022) and multiplication (Narayanan 2022) 'properly' — that it took a few months before arithmetic calculator tools were harnessed to LLMs. Prominent early systems to do so are Toolformer (Schick et al. 2023), ART (Paranjape et al. 2023), Program-of-Thoughts Prompting (Chen et al. 2022) and Program-aided Language Models (Gao et al. 2022), which take somewhat different paths toward similar goals.

Toolformer uses GPT-J (Wang 2021) as the base LM with 6.7 billion parameters, which is amenable to fine-tuning using modest hardware. It outperforms the much larger GPT-3 model at various tasks using the following tools: a calculator, a calendar, a question-answering (QA) tool, a traditional BM25-based Wikipedia search tool and a machine translation tool. For our purposes, the only relevant tool is the calculator, and the only relevant datasets are SVAMP (Patel, Bhattamishra, and Goyal 2021), MAWPS [3] and ASDiv (Miao, Liang, and Su 2020a). We will compare SYRELM against some of these tasks (§6). Toolformer does not use a symbolic solver in conjunction with a numeric calculator like SYRELM. ART uses GPT-3 with 175 billion parameters, which means it already has access to substantial reasoning ability packaged into the massive but opaque LM.

Program-of-Thoughts Prompting (Chen et al. 2022) and Program-aided Language Models (Gao et al. 2022) use LLMs as symbolic solvers to read natural language problems and generate programs as intermediate steps, offloading computation steps to a Python interpreter. PoT and PAL depend heavily on prompt engineering with fixed LLMs and not efficient LM adaptation. Qiao et al. (2023) used RLHF (Yuan et al. 2023) framework to teach LLMs how to selectively use tools. First, the model is taught how to invoke tools, and then reinforcement learning is used in selective tool usage.

The latest development in this direction has been the integration of Wolfram Alpha 'plugins' into ChatGPT (Wolfram 2023). This collaboration incorporated diverse tools from various domains that led to prompting ChatGPT to compose queries suited to Wolfram Alpha, collect the latter's output, and then either relay or transcribe or compose it with further operations. While Wolfram+ChatGPT wide-ranging tools will have a great impact, our focus here is on the specific synergy between symbolic interpretation and arithmetic problem-solving skills.

3 Arithmetic Reasoning: Natural to Formal Language

As we discussed earlier, expressing the reasoning steps required to solve a given arithmetic problem in some FL (parsable pseudocode, python, lambda calculus, etc.) shifts the burden of deterministic evaluation from the LM itself to some dedicated solver. However, the choice of FL is tricky due to multiple factors. First, the underlying LM should be

[1] https://lmsys.org/blog/2023-03-30-vicuna/

[2] https://github.com/joykirat18/SYRELM

[3] https://github.com/sroy9/mawps

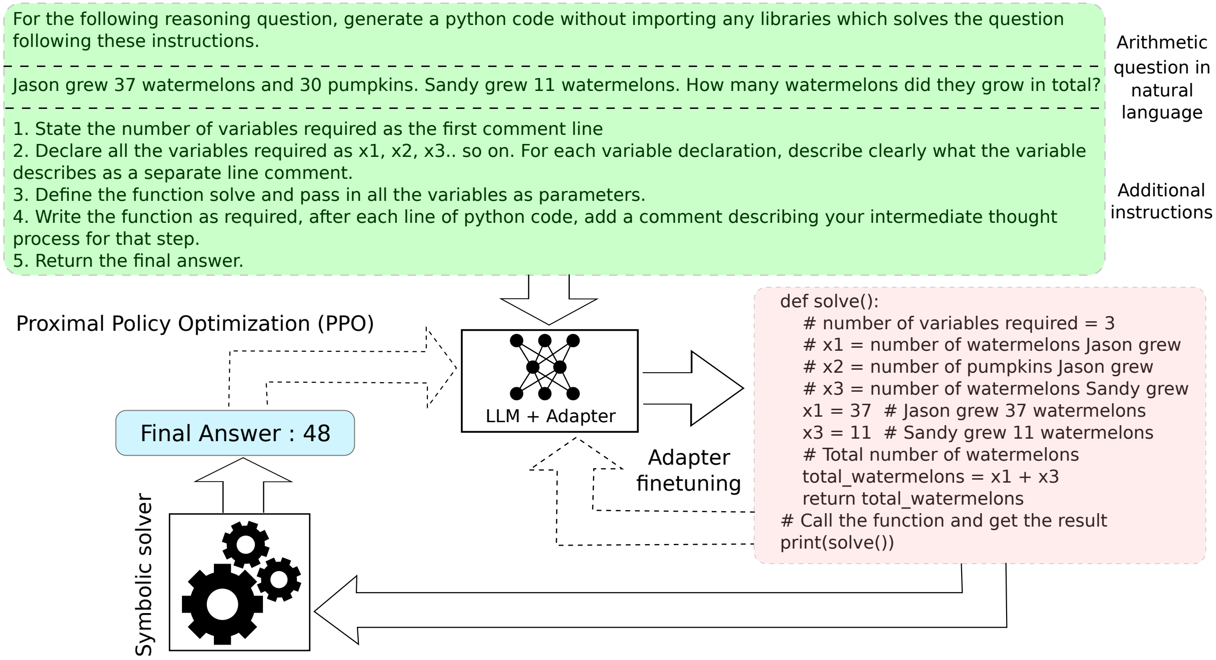

Figure 1: SyReLM architecture through an example. Given an arithmetic word problem in natural language, we first introduce a set of instructions to generate a formal language expression (Python, in this example). Next, a symbolic solver (a Python interpreter, in this example) executes the formal language expression to obtain the final answer. In the first stage of optimization, the adapter module is finetuned using a language modeling objective. Subsequently, we use Proximal Policy Optimization (PPO) to finetune the adapter, where the final answer and the coverage of variables needed are used as rewards. The base LM is kept frozen throughout.

exposed to generate expressions in the chosen FL in its pretraining stage. For example, Codex[4] (Chen et al. 2021) outperforms GPT-3 (Brown et al. 2020) by huge margins while using Program-of-Thoughts (PoT) prompting (Chen et al. 2022). Second, there is a trade-off between the complexity of the FL vs the size of the generated expression; an FL with a restricted vocabulary would require more steps to express the problem. The LM responsible for generating the FL expression might fail with longer reasoning chains. On the other hand, a large vocabulary creates a harder decision choice at each step due to increased possible options at each step.

Previous studies have shown the effectiveness of Python as a choice of FL for solving arithmetic problems with LMs (Gao et al. 2022; Chen et al. 2022). In designing SyReLM, we follow the same strategy with instruction-tuned models capable of generating Python codes. However, we adopt a few additional instructions to facilitate reasoning-specific rewards in downstream policy-gradient optimization. As shown in Figure 1, we instruct the LM in SyReLM to list all the necessary variables, along with their contextual meaning in natural language comments. We also add a CoT-like flavor by asking the model to generate interpretive natural language comments explaining the reasoning behind the Python statements. This aligns the two modes of reasoning: CoT vs. PoT. While pure CoT provides step-by-step and explainable reasoning to solve a problem, one cannot separate the computation from language generation. On the other hand, PoT provides a detachable computation graph at the expense of reduced reasoning explanation. By blending these two, we enforce that the LM actually generalizes across two different modes of reasoning. In the later section, we describe the usage of the additional instructions (e.g., variable names, values, etc.) while discussing the policy-gradient optimization step.

[4] https://platform.openai.com/docs/guides/code

Parsable Pseudocodes. Not all LMs are optimized for Python code generation. Given the rich syntax of Python, finetuning such models with additional Python scripts is beyond the computational capacity accessible to most researchers. Instead, in SyReLM, we formulate a simple FL prescription for such models.

Figure 2 shows an example of such a pseudocode. From the perspective of the language model, we instruct the LM tokenizer not to break the tokens corresponding to special symbols like **[find]**, **[add]**, etc., into subwords. We use a simple parser that reads these statements top to bottom, collects and stores the variables (**var1**, **var2**, etc.) along with their values (commented numerical values after each **[find]** statement), and evaluates expressions containing arithmetic operations (**[add]**, **[subtract]**, etc.). The LM acts in synergy with the parser; after generating a statement containing an arithmetic expression (e.g., **var4 = [subtract](var1,var3)** in Figure 2), the language model halts, and the parser is called

Jerry had 7 action figures on a shelf in his room. Later he added some more action figures to the shelf and removed 10 of the old ones. If there are 8 action figures on his shelf now How many action figures did he add to the shelf?

↓

```
var1 = [find](figures on shelf) # 7,
var2 = [find](figures added) # ?,
var3 = [find](figures removed) # 10,
var4 = [subtract](var1, var3) # 7 - 10 = -3,
var5 = [find](figures now) # 8,
var6 = [subtract](var5, var4) # 8-(-3) = 11,
 [return] (var6) # 11
```

Figure 2: Example of parsable pseudocode generation from LMs. Given the natural language arithmetic question (green), it first defines the required variables via [find] statements, followed by the computation statements using arithmetic operations like [add], [subtract], etc. The final answer is declared via the [return] statement.

for evaluation. The output of the parser is then appended as a comment (e.g., $\# 7 - 10 = -3$), and the LM resumes generation from there. This *halt-compute-resume* regime of incorporating an external tool with the LM is somewhat similar to Toolformer (Schick et al. 2023). However, a key distinction emerges at later stages with incorporating policy-gradient optimization.

We formally introduce the general problem as follows. Given a natural language arithmetic problem as a token sequence T_Q and a set of instructions as a token sequence T_{Inst}, the LM maps $[T_Q, T_{Inst}]$ to an FL token sequence T_{FL} that is then deterministically evaluated using a symbolic solver $\mathbf{S}$.

4 Training with Adapters

There exists a dramatic gap in expressiveness between models like GPT-3, and PaLM vs their smaller counterparts in the 1B-50B parameter scale, especially when the task is further away from simple language generation. Evidently, we need base LMs to *adapt* to the FL generation task via some form of finetuning. As shown in Figure 1, the training procedure is two-staged. First, the LM augmented with LoRA (Hu et al. 2021) adapters is finetuned to translate the natural language arithmetic problems to FL expressions. We further optimize the model along with the symbolic solver using Proximal Policy Optimization.

LoRA Finetuning with LM Objective. This first stage of optimization aligns the model to generate FL expressions T_{FL} from the instruction augmented arithmetic question $[T_Q, T_{Inst}]$. The objective can be expressed as

$$\max_{\theta_{LoRA}} \sum_{[T_Q, T_{Inst}], T_{FL}} \sum_{y \in T_{FL}} \log(P(y|[T_Q, T_{Inst}], \theta, \theta_{LoRA}, \mathbf{S})) \quad (1)$$

where θ and θ_{LoRA} are the LM and LoRA parameters, respectively. $\mathbf{S}$ denotes the (optional) symbolic solver that only comes into play while generating pseudocode sequences, as described in Section 3.

Proximal Policy Optimization. Typically, a PPO setup contains three constituent modules: a policy model π_θ, a reference model π_{ref}, a reward function r_t, and a value function $V(s_t)$ describing the reward at step t and value at state s_t. π_θ and π_{ref} define probability distributions over actions a_t given the state s_t. We define an advantage estimator $\hat{A}_t$ at each timestep t over a trajectory of timesteps 0 to T as follows:

$$\hat{A}_t = \sum_{i=0}^{T-t+1} (\gamma\lambda)^i (r_{t+i} + \gamma V(s_{t+i+1}) - V(s_{t+i})) \quad (2)$$

where $\gamma, \lambda \in (0, 1]$ are hyperparameters controling the bias-variance trade-off. The PPO objective can be described as,

$$\max_{\theta} \left(E_t \left[\frac{\pi_\theta(a_t|s_t)}{\pi_{ref}(a_t|s_t)} \hat{A}_t \right] - \beta E_t \left[KL[\pi_{ref}(\cdot|s_t), \pi_\theta(\cdot|s_t)] \right] \right) \quad (3)$$

where $KL(\cdot, \cdot)$ signifies the KL-divergence, and β is an adaptive hyperparameter controlling the KL-penalty. Intuitively, this penalty term drastically restricts the policy from diverging from the reference model. In our setting, we initialize π_θ and π_{ref} with the adapter-augmented LM finetuned using LM objective presented in Eq. 1. Timesteps of trajectories are defined per token. The value function $V(s_t)$ is initialized as an MLP head on top of the policy model.

Reasoning-specific Reward for PPO. The reward function r_t requires some novel approach to fit with our final goal of enhancing the reasoning ability of the LM. Typically, in the Reinforcement Learning from Human Feedback (RLHF) setting (where the PPO algorithm has found its most promising usage in conjunction with LMs), a separate LM is used to encode the human preferences from the feedback data. In our case, however, such an approach is not possible. On the other hand, we seek to enhance the LM's ability to use the symbolic solver faithfully. Therefore, the reward function can be straightforwardly defined using the correctness of the result generated by the symbolic solver. This, however, provides very little direct information about the quality of the generated FL. Intuitively, learning with such sparse information would require a vast amount of high-quality data and computing time. To tackle this, we incorporate rewards based on the generated FL description as follows:

1. Successful compilation of generated Python program; $R_1 = R_{max}$ if successful, 0 otherwise.
2. The absolute difference between the number of variables passed to the solve function (v_{gen}) in the generated program vs. in the gold program (v_{gold}), a larger difference signifies smaller reward; $R_2 = R_{max}\left(1 - \frac{v_{gen} - v_{gold}}{v_{gold}}\right)$.
3. R_3: Number of matching arithmetic operators $(+, -, *, /)$ between generated and gold program; $+R_{max}$ for each matching operator, $-R_{max}$ for each missing operator, and $-0.5R_{max}$ for unnecessary operators generated.
4. Absolute difference between the gold answer (y_{gold}) and the generated answer (y_{gen}) scaled by the gold answer; $R_4 = R_{max}\left(1 - \frac{y_{gen} - y_{gold}}{y_{gold}}\right)$.

Here, R_{max} is the maximum possible reward. We set it to 1 upon experimenting. Total reward is then $r = R_1 + R_2 + R_3 + R_4$. Such a diverse set of rewards provides a richer set of learning feedback for the policy model. One can intuitively map the reward ordering (R_1 to R_4) as stages to learn to reason – first, the policy model learns to generate compilable programs, then to identify the correct set of variables to use, then to select the correct set of arithmetic operations, and finally, to generate the full program that would lead to the correct answer. Compared to rewarding only the correctness of the final answer, our proposed reward function enables demarcating the capability of the policy model even while generating wrong answers.

5 Experimental Setup

Training Dataset. Given the novel setting of the FL expressions and instructions needed to train the adapters, we developed our own dataset. We built upon three existing mathword problem datasets: ASDiv (Miao, Liang, and Su 2020b), MAWPS (Miao, Liang, and Su 2020b), and Math23k (Shen et al. 2021). Since the original Math23K dataset is in Mandarin, we used an English-translated version. To reduce manual curation effort, we employed InstructGPT (Ouyang et al. 2022) to generate a noisy version of the pseudocode and Python programs to solve the selected problems. A manually curated 8-shot prompt was used with InstructGPT 175B version. Out of the initial 12,876 examples, we manually discarded instances with incorrect answers or flawed FL expressions. This process yielded 7,920 problems with pseudocode and 6,665 with Python programs, forming the training data for our models and constructed k-shot prompts for the baselines used in different experiments. Further details of the training dataset can be found in Appendix A.

LMs Used. In Section 3, we mentioned that different LMs possess different capacities for generating FL expressions given their pretraining data/objectives. We employ two LM classes for experimentation: Vicuna 13B (instruction-tuned version of the Llama 13B model (Touvron et al. 2023)) to generate Python, and GPT-J 6B (Wang and Komatsuzaki 2021) to generate pseudocode as intermediate FL expressions. We employ SymPy as the symbolic solver on top of Vicuna. These two LMs are fine-tuned using their LM objective and PPO. We build our own symbolic parser following the pseudocode structure discussed in Section 3.

We test SyReLM with three different arithmetic problem datasets: SVAMP, GSM8K, and, MultiArith. Details of the hyperparameters used in SyReLM are described in Appendix B.

6 Experimental Results

We compare the performance of the LMs optimized using SyReLM against multiple existing methods using LMs for solving arithmetic problems along with ablated versions of SyReLM. In the following list of competitive methods, *Model-name* stands for Vicuna 13B and GPT-J:

- *Model-name* (**ART**) implements ART (Paranjape et al. 2023) with *Model-name* as the base LM.

Models	SVAMP	MultiArith	GSM8K
Vicuna 13B (SyReLM)	**56.65**	**59**	**35.2**
Vicuna 13B (PAL)	53.7	49.4	27.5
Vicuna 13B (ART)	49.83	9.55	5.76
Vicuna 13B (TRICE)	-	18.83	-
Vicuna 13B (LMFT)	42.5	48.3	29.8
Vicuna 13B (4-shot)	37.5	29	7.62
Vicuna 13B (1-shot)	27	15.51	6.89
GPT-J (SyReLM)	**40.1**	**53.16**	3.50
GPT-J (PAL)	22.33	6.79	**8.25**
GPT-J (LMFT)	31.6	42.16	2.2
GPT-J (4-shot)	9.45	2	0.61
GPT-J (1-shot)	3	0.12	0
Toolformer	29.4	-	-
GPT-J (ART)	3.10	1.428	3.25
GPT-3.5 (standard 8-shot)	64.8	34.0	15.1

Table 1: Accuracy (%) of SyReLM-optimized models against baselines. The performance of Toolformer on the SVAMP dataset is taken from their original paper (Schick et al. 2023); we could not compare SyReLM against Toolformer on other datasets since the model weights/codebase are not publicly available. GPT-3.5 results are taken from (Zheng et al. 2023).

- *Model-name* (**LMFT**) is the LM finetuned version of the base model optimized using Eq. (1).
- *Model-name* (**k-shot**) is the base LM prompted with examples of FL generation samples and the symbolic solver; this provides the baseline ability of the LM without additional adapter-tuning or PPO;
- Toolformer, as proposed by Schick et al. (2023);
- *Model-name* (**PAL**) implements the PAL module (Gao et al. 2022) with *Model-name* as base LM.
- *Model-name* (**TRICE**) implements the TRICE module (Qiao et al. 2023) with *Model-name* as base LM.

Table 1 summarizes the performance of SyReLM along with the baseline and ablation variants. Vicuna 13B and GPT-J show strong improvements with SyReLM optimization over their base versions with k-shot prompting. The performance gain is more staggering with GPT-J compared to Vicuna (in a zero-shot setting, GPT-J is not able to generate any pseudocode or Python code whatsoever). This aligns with the fact that the adapter finetuning with FL expression generation (Eq. 1) essentially adds an instruction-following capability to the vanilla LM. We observe an interesting trend in SyReLM vs LMFT across different datasets. With increasing order of difficulty (based on the performance of the base LM, MultiArith $\rightarrow$ SVAMP $\rightarrow$ GSM8K), the performance gain mostly increases. Intuitively, one can argue that the reasoning-specific rewards indeed help the model generalize better across harder tasks.

Reward ablation. To investigate the role of the reward functions in the performance of SyReLM, we remove each reward from R_1 to R_4 and run the PPO step. Table 2 summarizes the results of reward ablation. In case of Vicuna, the adverse effect of removing any of the reward function is evident; the order of importance can be deduced as $R_4 > R_3 > R_2 > R_1$. Given that Vicuna is an optimized

Case I. Wrong code syntax

```
Question: There are 11 different books and 17 different movies in the ' crazy silly school ' series. If you read 7 of the books and watched 21 of the movies How many more movies than books have you read?
def solve():
  #number of movies = 21
  number of books = 7
  python
  # To find the difference between movies and books,
  # we can subtract the number of books from the number of movies
  difference = number_of_movies - number_of_books
  # Printing the difference
  return difference
print(solve())
```

Case II. Irrelevant answer

```
Question: There were 12 people on the bus. At the next stop 4 more people got on the bus. Each bus can not have more than 36 people. How many people are there on the bus now?
def solve():
  #number of variables required = 2
  x1 = 12  # initial number of people on the bus
  x2 = 4  # number of people who got on the bus
  # Checking if the total number of people on the bus exceeds the maximum capacity
  if x1 + x2 > 36:
     return "The bus is overcrowded."
print(solve())
```

Case III. Reasoning mistake

```
Question: Debby bought some water bottles when they were on sale. She drank 109 bottles a day. If the bottles lasted for 74 days How many bottles had she bought?
def solve():
  #number of variables required = 2
  x1 = 109  # number of bottles Debby drank per day
  x2 = 74  # number of days the bottles lasted
  # calculating the total number of bottles Debby drank
  total_bottles = x1 * x2

  # finding the number of bottles Debby bought
bottles_bought = total_bottles - 109*74
return bottles_bought
print(solve())
```

Figure 3: Case-wise analysis of erroneous prediction by Vicuna 13B with SYRELM from examples in SVAMP. Questions and generated Python codes are presented in black and blue, respectively, with the sources of error shown in red.

code generation model, it is intuitive that the reward towards getting the correct answer (R_4) will be more influential than the reward to generate an executable code (R_1). Irrespective of the relative importance, these rewards play a crucial role in aligning the model towards better arithmetic reasoning. With GPT-J, however, the importance ordering is not that straightforward. On SVAMP, we observe a trend similar to Vicuna. On MultiArith, it looks like the absence of R_1 or R_2 is beneficial, while results on GSM8K would suggest that removal of R_4 actually helps. However, it should be noted that GPT-J in general performs in a near-random manner on GSM8K and any trend in such a low performance setup does not tell much of a story.

SYRELM vs ART. We discussed in Section 2 that ART presupposes a strong reasoning capability encoded within the LM itself (their reported results are primarily on 175B models). This assumption drastically fails with models having orders of magnitude smaller parameters like GPT-J. With GPT-J, ART exhibits near-random performance across all datasets. Appendix C displays GPT-J's inability to grasp reasoning strategies or tool applications. With Vicuna 13B, the model performs well on SVAMP but fails to generalise on the other two datasets (MultiAirth and GSM8K). SYRELM, on the other hand, is well suited for such small LMs as it provides a smooth interface between the LM and the symbolic solver via LoRA adapters. SYRELM is also able to handle datasets with varying difficulty.

Method	Model	SVAMP	M-Arith	GSM
SYRELM	Vicuna 13B	56.65	59	35.2
	GPT-J	40.1	53.16	3.50
SYRELM$\backslash R_1$	Vicuna 13B	51.0	52.3	34.95
	GPT-J	39.09	57.66	3.203
SYRELM$\backslash R_2$	Vicuna 13B	49.25	52.0	25.75
	GPT-J	39.78	54.16	3.43
SYRELM$\backslash R_3$	Vicuna 13B	46.75	50.1	26.15
	GPT-J	35.69	47.5	3.02
SYRELM$\backslash R_4$	Vicuna 13B	31.08	45.8	22.2
	GPT-J	36.92	52	3.81

Table 2: Ablation study on SYRELM with the reward functions. SYRELM$\backslash R_i$ denotes that the reward function R_i (see Section 4) is ablated.

SYRELM vs Toolformer. Contrary to ART, Toolformer is expected to achieve performance most similar to SYRELM with GPT-J as the base model, given the fact that both methods use external tools while learning. The results on the SVAMP dataset verify the conjecture. However, even the simple pseudocode generation setup is much more powerful than learning to use a simple calculator like Toolformer. Furthermore, the PPO stage provides even stronger learning signals in SYRELM, thereby showing a large performance improvement ($+10.7$) on the SVAMP dataset compared to Toolformer.

Vicuna vs GPT-J. The performance of all the GPT-J-based models compared to Vicuna reflects the difference in generalizability between vanilla LMs vs instruction-tuned ones. The base performance (k-shot) is near random across all the datasets. With the introduction of SYRELM, we can observe a sharp rise in the performance on SVAMP (+30.65) and MultiArith (+51.16); however, that is not the case with GSM8K, where the performance gain, despite being positive, is meagre. Math-word problems in the latter dataset are qualitatively distinct compared to SVAMP and MultiArith. GSM8K is generally harder with more complex reasoning. The simplistic design of the pseudocode is not suited for such tasks as they tend to create a longer chain of statements that could have been expressed in Python more succinctly. Consequently, the required jump in generalized reasoning power is much higher for a pseudocode-generating GPT-J compared to a Python-generating Vicuna.

SYRELM vs PAL. The experiment closest to PAL is *Model-name* (k-shot), which achieves lower performance when compared to PAL, but with SYRELM (teaching how to use symbolic solver), it can surpass PAL performance. Therefore, PAL cannot learn how to use external tool usage to its full extent. GPT-J lacks the capability to generate Python code; simple pseudocode helps the small model in reasoning, thereby drastically increasing the performance compared to PAL. As mentioned above, GSM8K is harder; therefore, the pseudocode fails to express the complete reasoning steps.

SYRELM vs TRICE. Vicuna 13B (TRICE) was trained from scratch with official code (Qiao et al. 2023). SVAMP and GSM8K were used as their training datasets; therefore, only MultiArith was used for comparison. SYRELM shows a large improvement in the accuracy, pointing towards the advantage of converting problems to FL expression and then teaching the model to use the symbolic solver.

Quality of NL to FL translation. For GPT-J, the degree of syntactically incorrect pseudocodes generated are as follows: 2.3% on SVAMP, 2.5% on MultiArith, and 17.175% on GSM8K. Note that GPT-J without SYRELM is not able to generate a single, correct pseudocode even with few-shot example. With Vicuna, the base model produces 9% syntactically incorrect python codes; with SYRELM, this number becomes zero.

Error Cases with SYRELM. As we argued in Section 3, the generation template used in SYRELM provides a better explanation as it synthesizes the flavor of CoT with the underlying PoT format. We analyze the prediction errors encountered by SYRELM-optimized Vicuna 13B model. We randomly sampled 100 mispredictions from the SVAMP testbed and manually checked for possible sources of error. In Figure 3, three major error types are presented with one example each. In **Case I**, the model is struggling to generate correct syntax; it misses the number of variables needed as a comment after generating the start-of-comment token #. Instead, it directly initializes the variables; as a result, the generated code throws a compilation error with the usage of undefined variables (*number_of_movies* in this example). This is very much a sign of the LM overlooking details of the instructions and following its pretraining character. Next, we have **Case II**, with the model generating irrelevant answers to the question. In this example, the model generates the answer to whether or not the bus is overcrowded, while the question is to count the number of people in the bus. The model fell prey to the misdirect presented in the question stating the maximum capacity of the bus and ignored the later part. While this failure is somewhat blunt in the manifestation in this example, even huge LMs like ChatGPT are not robust against such adversarial misdirects while reasoning (Borji 2023). Finally, in **Case III**, we have straightforward reasoning errors. The model essentially lost the reasoning chain in this example; it correctly identified how to calculate the total number of bottles. Yet, it could not identify that the total number of bottles bought is the same as the total number of bottles drunk. Additional analyses with examples generated by GPT-J with SYRELM are presented in Appendix D. Similar error analyses on examples from GSM8K and MultiArith are provided in Appendix E.

Generalizability across Reasoning Steps. Figure 4 (top panel) shows the distribution of the reasoning steps required in the training data (count of non-comment lines in the generated Python script). Additionally, we use a subset of the SVAMP dataset to annotate the Python scripts using similar methods as described in Section 5 to estimate the number of reasoning steps required to solve the problems. Figure 4 (bottom panel) shows the error rate of Vicuna 13B with SYRELM on problems requiring different reasoning steps. While the model generally shows an increasing error rate with an increased number of reasoning requirements, it is still able to generalize to longer reasoning chains that are rarely represented in the training data.

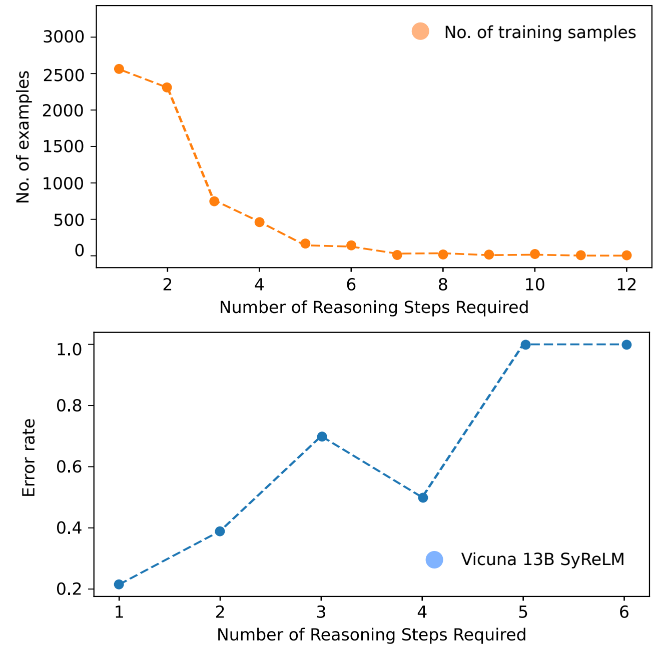

Figure 4: (Top) Distribution of training data examples across the number of reasoning steps required (number of lines in the Python code). (Bottom) The prediction error rate of Vicuna 13B with SYRELM vs the number of reasoning steps required to solve the problem.

7 Conclusion

Notwithstanding great recent enthusiasm about LLMs as all-purpose problem solvers, practitioners appreciate that LLMs work best when limited in their role to act as a glue between tools specialized to non-linguistic tasks like logic, arithmetic, or structured information retrieval. In response, LLMs are steadily getting better at invoking tools. Here, through the design of a new system, SYRELM, we explore a synergy between symbolic and numeric reasoning that has been established in middle-school pedagogy for a while, but not yet commonplace with LLMs. SYRELM is based on a frozen LM of modest size, coupled with a low-rank adapter for fine-tuning, keeping the setup within the computational capacity of most research groups. By prompting SYRELM to build a bridge between chain-of-thoughts and formal programs, we show that even frugal LMs can be effective at solving complex, multi-step arithmetic word problems, a capability thought to be emergent only in LLMs of massive sizes trained on enormous amounts of data.

Limitations. Our work primarily focuses on frugal LMs, which, as our experiments and existing literature suggest, have limited language generation and abstract reasoning capability. Despite improvement from SYRELM, some implicit shortcomings of these LMs manifest across the optimized versions as well. The LoRA finetuning process along with PPO requires additional GPU-hours compared to the fully zero-shot generation of the API-based large LMs.

Acknowledgments

The authors acknowledge the financial support of DYSL-AI.

References

Borji, A. 2023. A Categorical Archive of ChatGPT Failures. arXiv. arXiv:2302.03494.

Brown, T.; Mann, B.; Ryder, N.; Subbiah, M.; Kaplan, J. D.; Dhariwal, P.; Neelakantan, A.; Shyam, P.; Sastry, G.; Askell, A.; Agarwal, S.; Herbert-Voss, A.; Krueger, G.; Henighan, T.; Child, R.; Ramesh, A.; Ziegler, D.; Wu, J.; Winter, C.; Hesse, C.; Chen, M.; Sigler, E.; Litwin, M.; Gray, S.; Chess, B.; Clark, J.; Berner, C.; McCandlish, S.; Radford, A.; Sutskever, I.; and Amodei, D. 2020. Language Models are Few-Shot Learners. In Larochelle, H.; Ranzato, M.; Hadsell, R.; Balcan, M.; and Lin, H., eds., *NeurIPS*, volume 33, 1877–1901.

Chandra, B.; Chawda, B.; Kar, B.; Reddy, K. V.; Shah, S.; and Sudarshan, S. 2015. Data Generation for Testing and Grading SQL Queries. *The VLDB Journal*, 24(6): 731–755.

Chen, M.; Tworek, J.; Jun, H.; Yuan, Q.; Pinto, H. P. d. O.; Kaplan, J.; Edwards, H.; Burda, Y.; Joseph, N.; Brockman, G.; et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Chen, W.; Ma, X.; Wang, X.; and Cohen, W. W. 2022. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. *arXiv preprint arXiv:2211.12588*.

Chowdhery, A.; Narang, S.; Devlin, J.; Bosma, M.; Mishra, G.; Roberts, A.; Barham, P.; Chung, H. W.; Sutton, C.; Gehrmann, S.; Schuh, P.; Shi, K.; Tsvyashchenko, S.; Maynez, J.; Rao, A.; Barnes, P.; Tay, Y.; Shazeer, N.; Prabhakaran, V.; Reif, E.; Du, N.; Hutchinson, B.; Pope, R.; Bradbury, J.; Austin, J.; Isard, M.; Gur-Ari, G.; Yin, P.; Duke, T.; Levskaya, A.; Ghemawat, S.; Dev, S.; Michalewski, H.; Garcia, X.; Misra, V.; Robinson, K.; Fedus, L.; Zhou, D.; Ippolito, D.; Luan, D.; Lim, H.; Zoph, B.; Spiridonov, A.; Sepassi, R.; Dohan, D.; Agrawal, S.; Omernick, M.; Dai, A. M.; Pillai, T. S.; Pellat, M.; Lewkowycz, A.; Moreira, E.; Child, R.; Polozov, O.; Lee, K.; Zhou, Z.; Wang, X.; Saeta, B.; Diaz, M.; Firat, O.; Catasta, M.; Wei, J.; Meier-Hellstern, K.; Eck, D.; Dean, J.; Petrov, S.; and Fiedel, N. 2022. PaLM: Scaling Language Modeling with Pathways. arXiv:2204.02311.

Cobbe, K.; Kosaraju, V.; Bavarian, M.; Chen, M.; Jun, H.; Kaiser, L.; Plappert, M.; Tworek, J.; Hilton, J.; Nakano, R.; Hesse, C.; and Schulman, J. 2021. Training Verifiers to Solve Math Word Problems. *arXiv preprint arXiv:2110.14168*.

Devlin, J.; Chang, M.-W.; Lee, K.; and Toutanova, K. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *NAACL*, 4171–4186.

Gao, L.; Madaan, A.; Zhou, S.; Alon, U.; Liu, P.; Yang, Y.; Callan, J.; and Neubig, G. 2022. PAL: Program-aided Language Models. *arXiv preprint arXiv:2211.10435*.

Hu, E. J.; Shen, Y.; Wallis, P.; Allen-Zhu, Z.; Li, Y.; Wang, S.; and Chen, W. 2021. LoRA: Low-Rank Adaptation of Large Language Models. *CoRR*, abs/2106.09685.

Kojima, T.; Gu, S. S.; Reid, M.; Matsuo, Y.; and Iwasawa, Y. 2022. Large Language Models are Zero-Shot Reasoners. In *NeurIPS*.

Mahowald, K.; Ivanova, A. A.; Blank, I. A.; Kanwisher, N.; Tenenbaum, J. B.; and Fedorenko, E. 2023. Dissociating language and thought in large language models: a cognitive perspective. arXiv:2301.06627.

Miao, S.-y.; Liang, C.-C.; and Su, K.-Y. 2020a. A Diverse Corpus for Evaluating and Developing English Math Word Problem Solvers. In *ACL*, 975–984.

Miao, S.-y.; Liang, C.-C.; and Su, K.-Y. 2020b. A Diverse Corpus for Evaluating and Developing English Math Word Problem Solvers. In *ACL*, 975–984. Online.

Narayanan, A. 2022. Multiplication using ChatGPT. (Tweet).

Nathan, M. J.; Kintsch, W.; and Young, E. 1992. A Theory of Algebra-Word-Problem Comprehension and Its Implications for the Design of Learning Environments. *Cognition and Instruction*, 9(4): 329–389.

OpenAI. 2023. GPT-4 Technical Report. arXiv:2303.08774.

Ouyang, L.; Wu, J.; Jiang, X.; Almeida, D.; Wainwright, C.; Mishkin, P.; Zhang, C.; Agarwal, S.; Slama, K.; Ray, A.; Schulman, J.; Hilton, J.; Kelton, F.; Miller, L.; Simens, M.; Askell, A.; Welinder, P.; Christiano, P. F.; Leike, J.; and Lowe, R. 2022. Training language models to follow instructions with human feedback. In *NeurIPS*, volume 35, 27730–27744.

Paranjape, B.; Lundberg, S.; Singh, S.; Hajishirzi, H.; Zettlemoyer, L.; and Ribeiro, M. T. 2023. ART: Automatic multi-step reasoning and tool-use for large language models. *arXiv*.

Patel, A.; Bhattamishra, S.; and Goyal, N. 2021. Are NLP Models really able to Solve Simple Math Word Problems? In *NAACL*, 2080–2094. Online.

Powell, S. R.; and Fuchs, L. S. 2014. Does Early Algebraic Reasoning Differ as a Function of Students' Difficulty with Calculations versus Word Problems? *Learn. Disabil. Res. Pract.*, 29(3): 106–116.

Qiao, S.; Gui, H.; Chen, H.; and Zhang, N. 2023. Making Language Models Better Tool Learners with Execution Feedback. ArXiv:2305.13068 [cs].

Schick, T.; Dwivedi-Yu, J.; Dessì, R.; Raileanu, R.; Lomeli, M.; Zettlemoyer, L.; Cancedda, N.; and Scialom, T. 2023. Toolformer: Language models can teach themselves to use tools. *arXiv preprint arXiv:2302.04761*.

Shen, J.; Yin, Y.; Li, L.; Shang, L.; Jiang, X.; Zhang, M.; and Liu, Q. 2021. Generate & Rank: A Multi-task Framework for Math Word Problems. In *EMNLP Findings*, 2269–2279.

Touvron, H.; Lavril, T.; Izacard, G.; Martinet, X.; Lachaux, M.-A.; Lacroix, T.; Rozière, B.; Goyal, N.; Hambro, E.; Azhar, F.; Rodriguez, A.; Joulin, A.; Grave, E.; and Lample, G. 2023. LLaMA: Open and Efficient Foundation Language Models. arXiv:2302.13971.

Wallace, E.; Wang, Y.; Li, S.; Singh, S.; and Gardner, M. 2019. Do NLP Models Know Numbers? Probing Numeracy in Embeddings. In *EMNLP Conference*.

Wang, B. 2021. Mesh-Transformer-JAX: Model-Parallel Implementation of Transformer Language Model with JAX. https://github.com/kingoflolz/mesh-transformer-jax.

Wang, B.; and Komatsuzaki, A. 2021. GPT-J-6B: A 6 Billion Parameter Autoregressive Language Model. https://github.com/kingoflolz/mesh-transformer-jax.

Wei, J.; Wang, X.; Schuurmans, D.; Bosma, M.; Chi, E. H.; Le, Q.; and Zhou, D. 2022. Chain of Thought Prompting Elicits Reasoning in Large Language Models. *CoRR*, abs/2201.11903.

Wolfram, S. 2023. ChatGPT Gets Its "Wolfram Superpowers"! (blog).

Yuan, Z.; Yuan, H.; Tan, C.; Wang, W.; Huang, S.; and Huang, F. 2023. RRHF: Rank Responses to Align Language Models with Human Feedback without tears. arXiv:2304.05302.

Zheng, C.; Liu, Z.; Xie, E.; Li, Z.; and Li, Y. 2023. Progressive-Hint Prompting Improves Reasoning in Large Language Models. arXiv:2304.09797.

Zhou, H.; Nova, A.; Larochelle, H.; Courville, A.; Neyshabur, B.; and Sedghi, H. 2022. Teaching Algorithmic Reasoning via In-context Learning. arXiv:2211.09066.