

1 **A Discussion of Limitations and Future Work**

2 We have presented a novel framework that integrates Compiled Neural Networks (CoNNs) with
3 existing language models to bolster their rule understanding abilities. Although our approach has
4 shown promising performance improvements on symbolic and arithmetic reasoning tasks, there are
5 several limitations and potential avenues for future research that warrant further exploration.

6 A significant limitation of our current framework lies in the more efficient and natural incorporation
7 of CoNNs into language models. Currently, our method employs a sparse neural network that treats
8 the pre-trained language model and CoNNs as separate modules. A more desirable solution is to
9 leverage a dense neural network, simultaneously utilizing the benefits of both components. A potential
10 direction to achieve this is initializing part of the language model’s parameters with CoNNs before
11 pre-training and locking these parameters during training, thereby promoting seamless integration of
12 CoNNs and reducing the associated computational overhead.

13 Another promising area of research is the inclusion of explicit knowledge into CoNNs. While the
14 current implementation centers on encoding rules into CoNNs, future work could exploit techniques
15 from knowledge graphs to compile explicit knowledge into language models. This may significantly
16 enhance language models’ interpretability and knowledge representation capabilities, potentially
17 resulting in improved performance on an even broader range of tasks.

18 Furthermore, examining the large-scale applicability of CoNNs is a beneficial endeavor. Although
19 our experiments have been conducted on a relatively small scale (up to five stacked CoNNs), the
20 advancements and abilities language models may gain from larger-scale combinations of CoNNs
21 remain unclear. Exploring the scalability of our method and the performance advantages of deploying
22 more complex CoNN architectures in language models could provide valuable insights into their
23 potential.

24 In addition, investigating the adaptability of our framework for various rules and knowledge sources
25 is crucial. While our current implementation concentrates on rule-based reasoning, extending the
26 approach to encompass other types of knowledge, such as facts, heuristics, or more intricate logical
27 structures, is a possibility. Such an investigation could enhance our understanding of CoNNs’
28 capabilities and limitations while representing diverse forms of knowledge.

29 Finally, we acknowledge the limited scope of tasks and benchmarks used in our evaluation. To fully
30 comprehend the potential and limitations of incorporating CoNNs into language models, it is vital
31 to test our method on a more extensive array of tasks and domains. This could involve reasoning
32 tasks requiring the integration of multiple knowledge types, as well as real-world scenarios with more
33 demanding problem-solving contexts.

34 In conclusion, our work on enhancing language models’ rule understanding capabilities through
35 CoNN integration has yielded promising results, albeit with some limitations and remaining chal-
36 lenges. By addressing these areas and extending our approach, we believe that it can ultimately lead
37 to the development of more powerful, interpretable, and knowledge-rich language models.

38 **B Compiled Neural Networks**

39 In this section, we discuss the concept, implementation, and potential of Compiled Neural Networks
40 (CoNN), a type of neural network inspired from previous works on transformers. CoNNs can
41 perform diverse tasks such as computer arithmetic and linear algebra, demonstrating a wide range of
42 applications in language model and beyond.

43 **B.1 Introduction**

44 Transformers have garnered significant attention due to their ability to capture high-order relations
45 and manage long-term dependencies across tokens through attention mechanisms. This enables
46 transformers to model contextual information effectively. Pre-trained language models, such as
47 GPT-3 [Brown et al., 2020], exploit contextual learning to invoke various modules for different tasks,
48 like performing arithmetic upon receiving arithmetic prompts. To further enhance rule comprehension
49 in such models, CoNN-based modules are introduced as a part of Neural Comprehension.

Example 1 The Parity CoNN

If we want the transformer to perform the Parity task full accurately:

Input: 1 0

1. **Select**(Indices,Indices,**True**) = $\begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$

2. **Aggregate** $\left(\begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, [1 \ 0]\right) = [1 \ 1]$

3. **Zipmap**([1 1], **Lambda** $x : 0 \text{ if } x \% 2 == 0 \text{ else } 1$) = [1 1]

Output: [1 1]



Figure 1: Demonstration of the principles of Parity CoNN.

50 Distinct from common models like BERT [Devlin et al., 2018], CoNNs leverage a transformer
51 structure and derive their weights from specialized design rather than pre-training. Each Attention
52 layer and Multilayer Perceptron (MLP) layer in a CoNN represents a specific sequence transformation,
53 leading to a neural network module embodying explicit and interpretable operations.

54 RASP [Weiss et al., 2021] is a Restricted Access Sequence Processing Language that abstracts
55 the computational model of Transformer-encoder by mapping its essential components, such as
56 attention and feed-forward computation, into simple primitives like select, aggregate, and zipmap.
57 This language enables RASP programs to perform various tasks like creating histograms, sorting, and
58 even logical inference, as demonstrated by Clark et al. [2020].

59 Tracr [Lindner et al., 2023] serves as a compiler that converts human-readable RASP code into
60 weights for a GPT-like transformer architecture with only a decoder module. The Tracr framework
61 uses JAX to transform RASP-defined code into neural network weights. Our neural reasoning
62 framework employs weights generated by Tracr, which are then converted into PyTorch weights to be
63 compatible with the pre-trained language model.

64 Looped Transformers as Programmable Computers [Giannou et al., 2023] introduces a novel trans-
65 former framework that simulates basic computing blocks, such as edit operations on input sequences,
66 non-linear functions, function calls, program counters, and conditional branches. This is achieved
67 by reverse engineering attention and hardcoding unique weights into the model, creating a looped
68 structure. The resulting CoNN can emulate a general-purpose computer with just 13 layers of trans-
69 formers, and even implement backpropagation-based context learning algorithms, showcasing the
70 approach’s vast application prospects.

71 Overall, the potential applications of CoNNs are extensive, given their capacity to perform a wide
72 array of tasks beyond natural language processing. CoNNs offer increased interpretability and
73 transparency through explicitly defined operations, which is vital in fields such as medical diagnosis
74 and legal decision-making. Additionally, CoNNs can lead to more efficient and effective neural
75 network architectures by reducing pre-training requirements and facilitating improved optimization
76 of network parameters.

77 B.2 Example

78 In this subsection, we briefly describe how computational processes can be represented using trans-
79 former code and demonstrate how new CoNN weights can be obtained with the aid of the Tracr
80 compiler.

81 B.2.1 Parity CoNN

82 In the introduction, we tried to introduce how to perform parity checking on a sequence containing [0
83 1 1] using a compiled neural network. Whenever we need to check the sequence, this compiled neural
84 network can output the completely correct answer.

THE TRACR CODE OF PARITY CONN

```
def parity(sop) -> rasp.SOp:
    """Multiply the length of each token."""
    sop = rasp.SequenceMap(lambda x,y: x * y, sop, length).named('map_length')

    """Add each bit."""
    out = rasp.numerical(rasp.Aggregate(rasp.Select(rasp.indices, rasp.indices, rasp.
        Comparison.TRUE).named('Select'), rasp.numerical(rasp.Map(lambda x: x, sop).named('
        map_length')), default=0).named('Aggregate'))

    """Calculate whether the remainder of dividing it by 2 is odd or even."""
    out = rasp.Map(lambda x: 0 if x % 2 == 0 else 1, out).named('Zipmap')

    return out
```

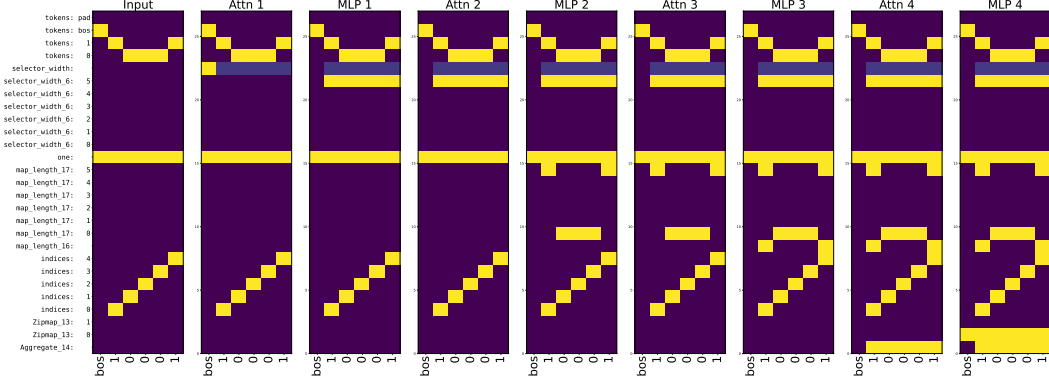


Figure 2: Input the $[1, 0, 0, 0, 1]$ (target output = 0) for Parity CoNN.

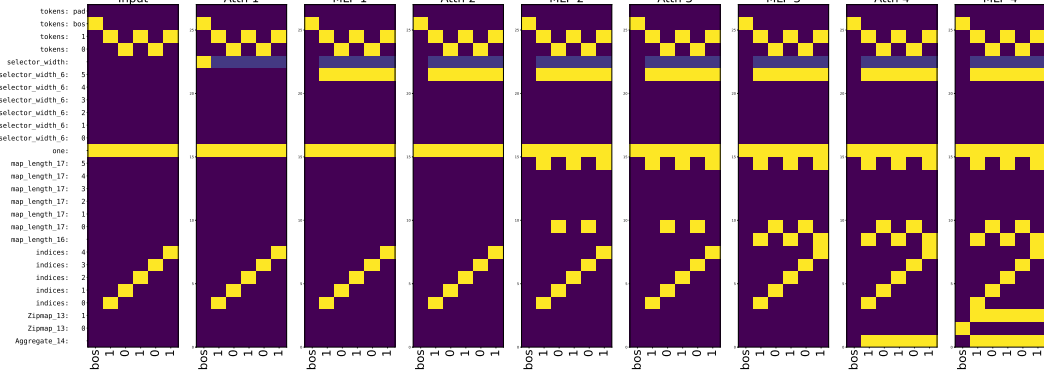


Figure 3: Input the $[1, 0, 1, 0, 1]$ (target output = 1) for Parity CoNN.

85 The first step is to obtain a matrix of all ones to that of the sequence using the 'Select' operation. In
 86 the second step, the 'Aggregate' operation is used to combine the matrix obtained in the previous step
 87 with the input sequence (with the aim of calculating the total number of 0's and 1's in the sequence).
 88 The third step involves determining whether the total count is odd or even by "Zipmap".

89 Figures 2 and 3 present two distinct input sequences, and illustrate the corresponding hidden state
 90 and final output obtained after passing through the internal layers of the Parity CoNN architecture.

THE TRACR CODE OF REVERSE CONN

```
def reverse(sop) -> rasp.SOp:
    """Get the indices from back to front."""
    opp_idx = (length - rasp.indices).named("opp_idx")

    """opp_idx - 1, so that the first digit of indices = 0."""
    opp_idx = (opp_idx - 1).named("opp_idx-1")

    """Use opp_idx to query indices, get the Select."""
    reverse_selector = rasp.Select(rasp.indices, opp_idx, rasp.Comparison.EQ).named("
        reverse_selector")

    """Aggregate the reverse_selector and sop"""
    return rasp.Aggregate(reverse_selector, sop).named("reverse")
```

91 B.2.2 Reverse CoNN

92 Figures 4 and 5 show the hidden state and output of Reverse CoNN when inputting text. The
 93 embedding of CoNN can be customized, so tokens can be either words like 'hello' or individual
 94 letters.

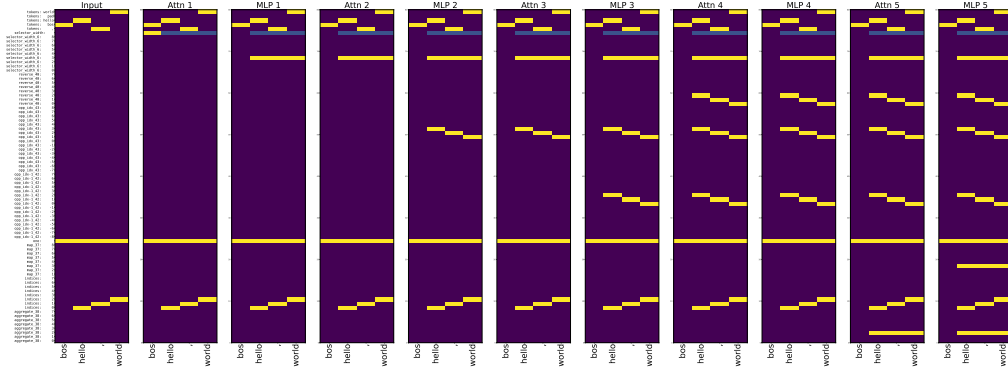


Figure 4: Input the $['hello', ',', 'world']$ for Reverse CoNN.

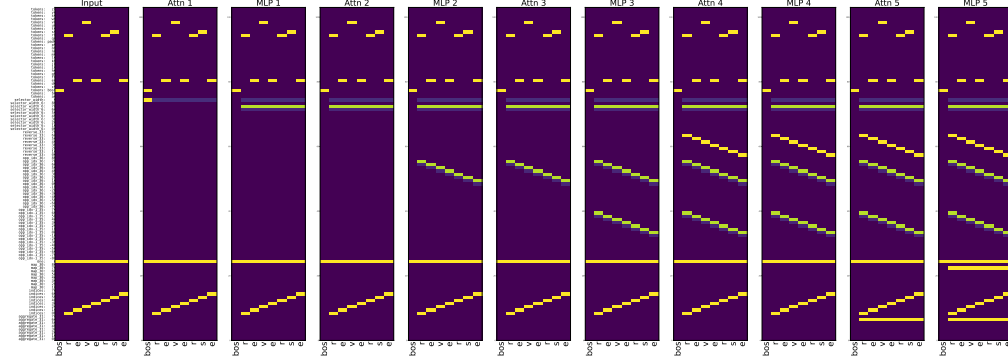


Figure 5: Input the $['r', 'e', 'v', 'e', 'r', 's', 'e']$ for Reverse CoNN.

95 B.2.3 Addition CoNN

96 Due to the high complexity of the model, we decided to omit the hidden state transformation for
 97 the Addition CoNN. However, we have provided code later in the text that will allow for easy
 98 implementation of this CoNN. The code includes `add_in_the_same_position` and `add_carry`
 99 functions, which are used to calculate the addition and carry of pairs in the CoNN respectively. We
 100 divide the entire operation into two models. For the `add_carry` model, we refer to the approach

THE TRACR CODE OF ADDITION CONN

```
def split(sop, token, index):
    """Match the position of target token"""
    target_position = rasp.Aggregate(rasp.Select(sop, rasp.Map(lambda x: token, sop),
        rasp.Comparison.EQ), rasp.indices)

    """If need to match the front position."""
    if index == 0:
        out = rasp.Aggregate(rasp.Select(rasp.indices, rasp.indices - (length -
            target_position), rasp.Comparison.EQ),
            sop) # Move the sop on the left side of the token to the far
                right.
        return rasp.SequenceMap(lambda x, i: x if i == 2 else "_", out, rasp.categorical(
            rasp.SequenceMap(lambda x, i: 2 if x >= i else 0, rasp.indices, length -
                target_position))) # Use "_" to fill the empty position on the left.

    """If need to match the finally number."""
    else:
        return rasp.SequenceMap(lambda x, i: x if i else "_", sop,
            rasp.SequenceMap(lambda x, i: 1 if x > i else 0, rasp.
                indices, target_position)).named(
                f"shift") # Use "_" to fill the empty position on the left.

def atoi(sop):
    """Converts all text to number, and uses 0 for strings of types other than numbers,
    It may be mixed with 'str' or 'int'."""
    """
    return rasp.SequenceMap(lambda x, i: int(x) if x.isdigit() else 0, sop, rasp.indices)
        .named(
            "atoi")

def shift(sop):
    """Get the target indices."""
    idx = (rasp.indices - 1).named("idx-1")

    """Use opp_idx to query indices, get the Select."""
    selector = rasp.Select(idx, rasp.indices,
        rasp.Comparison.EQ).named("shift_selector")

    """Aggregates the sops and selectors (converted from indexes)."""
    shift = rasp.Aggregate(selector, sop).named("shift")
    return shift

def add_in_the_same_position(sop):
    x = atoi(split(sop, '+', 0)) + atoi(split(sop, '+', 1))
    return x

def carry(sop):
    weight = shift(rasp.Map(lambda n:1 if n>9 else 0, sop))

    weight = rasp.Aggregate(rasp.Select(rasp.indices, rasp.indices, lambda key, query: key ==
        query), weight, default=0)
    x = rasp.Map(lambda n: n-10 if n>9 else n, sop)
    return x + weight
```

101 of ALBERT. After the output of the add_in_the_same_position model, we cyclically use the
 102 add_carry model L times, where L is the length of the text, to ensure that all digits can carry. It
 103 is important to note that this particular Addition CoNN is only capable of performing addition
 104 operations on natural numbers.

105 B.2.4 Subtraction CoNN

106 The subtraction CoNN is similar to the addition CoNN. First, each digit is subtracted from its
 107 corresponding digit, and then it is determined whether to carry over. For ease of experimentation, this
 108 subtraction CoNN only supports subtraction of natural numbers where the minuend is greater than
 109 the subtrahend.

THE TRACR CODE OF SUBTRACTION CoNN

```
def split(sop, token, index):...

def atoi(sop):...

def shift(sop):...

def sub_in_the_same_position(sop):
    x = atoi(split(sop, '-', 0)) - atoi(split(sop, '-', 1))
    return x

def carry(sop):
    weight = shift(rasp.Map(lambda n:1 if n<0 else 0,sop))
    weight = rasp.Aggregate(rasp.Select(rasp.indices, rasp.indices, lambda key, query:key ==
        query), weight, default=0)
    x = rasp.Map(lambda n:n+10 if n<0 else n,sop)
    return x - weight
```

Model	Layers	Heads	Vocabulary Size	Window Size	Hidden Size	MLP Hidden Size	# Parameters	Compared to GPT-3
Pariity	4	1	4	40	132	1959	2.2M	≈ 1/100,000
Reverse	4	1	28	40	297	1640	4.3M	≈ 1/50,000
Last Letter	3	1	28	16	103	32	62.6K	≈ 1/3,000,000
Copy	1	1	28	16	69	26	8.8K	≈ 1/20,000,000
Add_in_the_same_position	7	1	13	40	535	6422	51.8M	≈ 1/3000
Add_Carry	3	1	122	40	130	52	117K	≈ 1/1,500,000
Sub_in_the_same_position	7	1	13	40	535	6422	51.8M	≈ 1/3000
Sub_Carry	3	1	122	40	130	52	117K	≈ 1/1,500,000

Table 1: We reported on a CoNN with a single function, including its actual parameter size and comparison with the parameters of GPT-3.

B.3 CoNN model parameters

The parameter sizes of all CoNN models used in this work are listed in Table 1. It is noteworthy that even for GPT-3, which has parameters that are orders of magnitude larger, it remains challenging to solve symbolic problems. However, with the use of compiled neural networks, only a small number of parameters are needed to achieve Neural Comprehension.

B.4 Environmental and Human-centric Benefits of Compiled Neural Networks

Compiled Neural Networks (CoNNs) address concerns related to the environmental impact of training large models and the need for human-centric computing. CoNN models can reduce energy consumption and carbon emissions by minimizing extensive pre-training and decreasing parameter size, as seen in Table 1. This reduction in computational power and energy requirements makes both the training and inference processes more environmentally friendly. Additionally, Neural Comprehension offers a more interpretable and transparent alternative to conventional deep learning models. CoNN’s explicit operation definitions and specialized architecture enable users to comprehend the reasoning behind model decisions, fostering trust and facilitating human-AI collaboration. Increased interpretability also allows for scrutiny of model behavior, promoting the development of fair, accountable, and transparent systems aligned with ethical considerations and human values.

C AutoCoNN

We identify a critical limitation in pre-trained language models, namely, their lack of deep conceptual comprehension of symbolic operations like addition, which we refer to as Neural Comprehension. For example, when ChatGPT is directly asked, ‘What are the correct steps to perform addition?’, it can explain the correct addition steps. However, it cannot apply this knowledge to solve more complex addition problems, such as ‘What is the answer to $103541849645165+656184535476748421$?’. To address this gap, we propose a novel method called AutoCoNN (Automatic Compiler of Neural Networks) for the fast and efficient compilation of neural networks by utilizing the code writing, context learning, and language understanding capabilities of large language models.

AutoCoNN consists of a three-stage process: Observation, Induction, and Comprehension. The purpose of this framework is to automate the construction of Compiler Neural Networks (CoNN)

for various tasks and domains, thereby enhancing the learning and problem-solving abilities of language models within the Neural Comprehension framework in a scalable and efficient manner while minimizing the need for human intervention.

C.1 Method

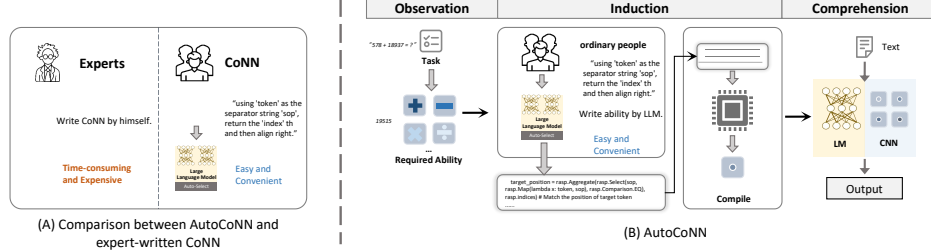


Figure 6: Introduction and Comparison of AutoCoNN

The specific process of AutoCoNN is illustrated in Figure 6. During the Observation stage, we identify the abilities required to solve a given task, such as addition, subtraction, multiplication, and division for arithmetic reasoning tasks. This stage aims to determine the rules and symbolic comprehension that pre-trained language models lack.

In the Induction stage, we leverage the code writing capabilities of large language models to generate CoNNs representing the desired rules and concepts. We achieve this by providing the model with a small set of examples that include CoNN code and accompanying comments. As a result, the model generates code that compiles a neural network customized for the specific problem at hand. However, as the model’s comprehension of CoNNs is limited, it may encounter run-time errors or produce unintended outputs. To mitigate this issue, we employ a trial-and-error mechanism throughout the decoding phase by implementing sample decoding, which adjusts the temperature parameter, allowing for random decodings and generating diverse text outputs. These outputs are compiled into new CoNNs, and the accuracy of the networks is evaluated using a small test set. By selecting the best-performing CoNN, we obtain a neural network that effectively captures the desired mathematical concepts and rules. The sample encoding formula for temperature-based decoding is as follows, where $p(w_i)$ is the probability of word w_i , z_i is the logits for word w_i , and T is the temperature parameter:

$$p(w_i) = \frac{e^{\frac{z_i}{T}}}{\sum_j e^{\frac{z_j}{T}}} \quad (1)$$

We evaluate the accuracy of these compiled networks using a small test set, requiring only one or two samples. By selecting the best-performing CoNN, we obtain a more accurate neural network that effectively captures the desired mathematical concepts and rules.

During the Comprehension stage, the original large language model is integrated with the compiled neural network using our Neural Comprehension framework, enhancing its rule comprehension capabilities.

Table 2 presents the effectiveness of AutoCoNN in obtaining CoNN models suitable for Neural Comprehension. The results indicate that for simple CoNN models, AutoCoNN can reduce manpower costs. However, when building more complex CoNN models, the method may be constrained by the code capabilities of large language models, posing challenges in achieving its goal.

CoNN Model	Expert's Working Time	Success by AutoCoNN	Can AutoCoNN solve
Parity Model	1 hours	8/20	✓✓
Reverse Model	0.5 hour	15/20	✓✓
Last Letter Model	0.5 hour	13/20	✓✓
Copy Model	0.2 hour	17/20	✓✓
Addition Model	48 hours	0/20	✗
Subtraction Model	48 hours	0/20	✗

Table 2: Comparison between AutoCoNN and Expert Built CoNN. The column 'Expert's Working Time' refers to the time required for a trained engineer to write the CoNN code; 'Success by AutoCoNN' refers to the accuracy of 20 results generated by using GPT-3.5 for diverse decoding; 'Can AutoCoNN solve' refers to whether AutoCoNN can identify suitable CoNN code from the 20 results through validation.

177 D Supplementary Experiment

178 D.1 The effect of training data scale on length generalization of gradient-based models

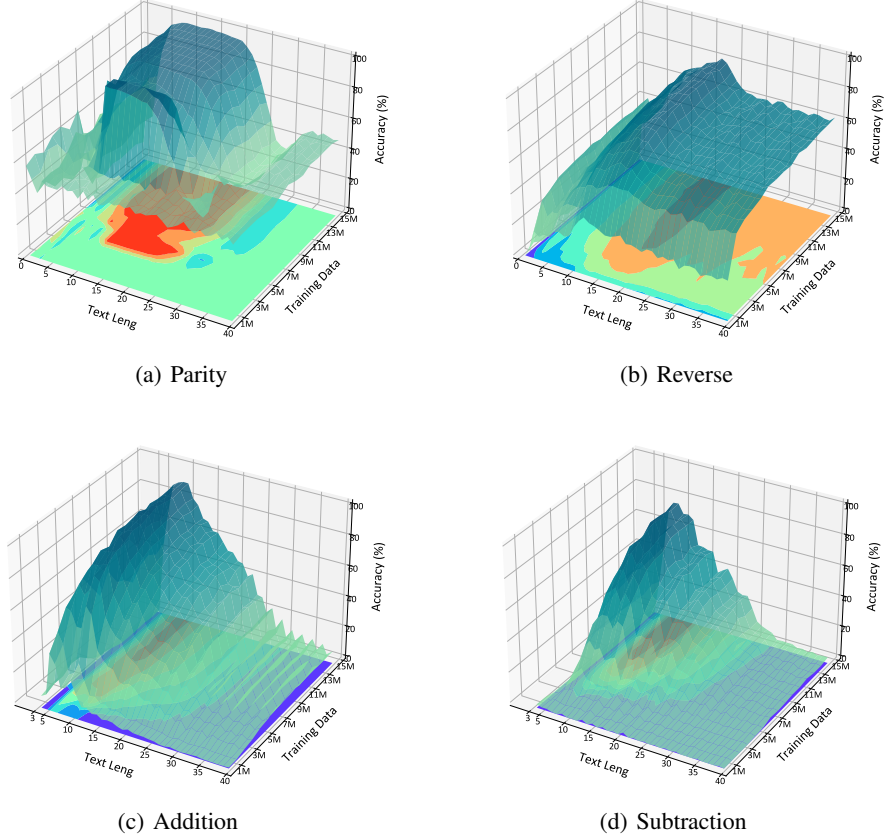


Figure 7: Length Generalization Performance of Language Models with Different Dataset Sizes.

179 To investigate the impact of training data scale on out-of-distribution (OOD) performance, we
 180 conducted experiments using the T5-large model with varying amounts of in-distribution training
 181 data. The experimental setup closely followed that of **Main Figure 3**, utilizing numbers with 10 to
 182 20 digits as the training set but varying the number of training examples between 1 million and 15
 183 million. The peak validation set performance for each experiment is reported in Figure 7.

184 The results in Figure 7 show that increasing the scale of the In-Dist training data leads to only marginal
 185 improvements in OOD performance. This finding is discouraging, suggesting that gradient-based
 186 language models face challenges in capturing the true underlying meaning of symbols and their
 187 transformation rules based on the data distribution alone.

188 D.2 Real-world Arithmetic Reasoning Tasks

189 In Table 3, we compared Vanilla CoT with the Neural Comprehension framework for arithmetic
 190 reasoning tasks. We integrated the Addition and Subtraction CoNNs with LLMs and observed
 191 improved performance across several tasks. This suggests that the proposed Neural Comprehension
 192 framework can compensate for the difficulties faced by large-scale language models in computational
 193 tasks. Nevertheless, the performance improvement is not as significant due to the choice of specific
 194 CoNN models to ensure clarity in our experiments. Designing CoNN models to support more general
 195 arithmetic tasks could potentially yield more substantial improvements. In addition, since the Neural

Method	GSM8K	SingleEq	AddSub	MultiArith	SVAMP	Average
Previous SOTA (Fintune)	35 ^a /57 ^b	32.5 ^c	94.9 ^d	60.5 ^e	57.4 ^f	-
GPT-3 Standard	19.7	86.8	90.9	44.0	69.9	62.26
GPT-3 (175B)	13.84	62.02	57.22	45.85	38.42	43.47
code-davinci-001	13.95 _(+0.11)	62.83 _(+0.81)	60.25 _(+3.03)	45.85 _(+0.0)	38.62 _(+0.2)	44.30 _(+0.83)
Instruct-GPT (175B)	60.20	91.01	82.78	96.13	75.87	81.20
code-davinci-002	60.42 _(+0.22)	91.01 _(+0.0)	82.78 _(+0.0)	96.13 _(+0.0)	76.09 _(+0.22)	81.29 _(+0.09)

Table 3: Problem solve rate (%) on arithmetic reasoning datasets. The previous SoTA baselines are obtained from: (a) GPT-3 175B finetuned [Cobbe et al., 2021]; (b) GPT-3 175B finetuned plus an additional 175B verifier[Cobbe et al., 2021]; (c) Hu et al. [2019]; (d) Roy and Roth [2016]; (e) Roy and Roth [2016]; (f) Amini et al. [2019]; (f) Pi et al. [2022]

Comprehension framework improves the gap between the data distribution learned by the language model during training through gradient descent and the real rules, it can also be combined with some existing logical improvements to language models, including self-consistency [Wang et al., 2023], least-to-most [Zhou et al., 2022], self-improve [Huang et al., 2022], and self-verification [Weng et al., 2022]. It can also be combined with some zero-shot methods [Kojima et al., 2022, Zhang et al., 2023].

To further evaluate the effectiveness of the Neural Comprehension framework, Table 4 presents the results of fine-tuning T5 models with Addition and Subtraction CoNN on the GSM8K training dataset. The comparison of three different-sized models reveals that the framework can model deterministic rules defined by humans, thus avoiding the uncertainty associated with gradient descent learning from data distribution.

Method	T5-small	T5-base	T5-large
Origin	1.74	1.52	3.87
Neural Comprehension	1.82	1.59	4.02
Ours Improve	+0.08	+0.07	+0.15

Table 4: The test set problem-solving rate (%) of the T5 model on the GSM8K dataset.

D.3 The Efficiency of Neural Comprehension

Model	Params	Task	Vanilla		Neural Comprehension		δ_{Time}	
			GPU	CPU	GPU	CPU	GPU	CPU
T5-small	60M	Coin Flip	5.280s	5.720s	5.431s	5.872s	0.151s (2.86%)	0.152s (2.66%)
T5-base	220M	Coin Flip	7.865s	13.767s	8.010s	13.939s	0.145s (1.84%)	0.172s (1.25%)
T5-large	770M	Coin Flip	14.055s	32.953s	14.194s	33.120s	0.139s (0.99%)	0.167s (0.51%)
T5-small	60M	Last Letter Concatenation	16.233s	28.309s	16.744s	28.720s	0.511s (3.15%)	0.411s (1.45%)
T5-base	220M	Last Letter Concatenation	28.912s	55.660s	29.426s	56.087s	0.514s (1.78%)	0.427s (0.77%)
T5-large	770M	Last Letter Concatenation	49.584s	103.739s	50.066s	104.134s	0.482s (0.97%)	0.395s (0.38%)

Table 5: In Neural Comprehension framework, the inference latency comparison of the T5 model.

To evaluate the efficacy of Neural Comprehension, we conducted further experiments comparing the inference latency of both the Vanilla and Neural Comprehension frameworks on an equal number of sequences and equal sequence lengths using GPU and CPU configurations. We employed a batch size of 1 and assessed the inference latency of Neural Comprehension in conjunction with various T5 model sizes across two symbolic inference tasks to ascertain efficiency. The full results are detailed in Table 5.

Our findings reveal that implementing Neural Comprehension increases computational requirements, primarily attributed to the supplementary parameter volume and computational demands introduced by CoNNs. However, as the scale of pre-trained language models expands, the proportion of δ_{Time} within the Neural Comprehension framework progressively diminishes, particularly for larger language models.

Model	Model Creator	Modality	Version	# Parameters	Tokenizer	Window Size	Access
T5-small	Google	Text	T5.1.0	60M	T5	512	Open
T5-base	Google	Text	T5.1.0	220M	T5	512	Open
T5-large	Google	Text	T5.1.0	770M	T5	512	Open
GLM-130B	Tsinghua University	Text	GLM-130B	130B	ICE	2048	open
GPT-3	OpenAI	Text,Code	code-davinci-001	175B*	GPT-2	2048	limited
GP-3.5	OpenAI	Text,Code,Instruct	code-davinci-002	175B*	GPT-2	8096	limited
GPT-4	OpenAI	Text,Code,Instruct,...	gpt-4	175B*	GPT-2	8000	limited

Table 6: Models. Description of the models evaluated in this effort: provenance for this information is provided in models. * indicates that we believe the associated OpenAI models are this size, but this has not been explicitly confirmed to our knowledge.

E Implementation and Details

In this section, we provide a detailed description of the experimental setup from a model and dataset perspective, ensuring repeatability of our experiments.

E.1 Model

Our experiments primarily involve the T5, GPT, and GLM-130B families of models. Neural Comprehension framework supports seamless integration with language models having decoder structures, regardless of the scale of the language model. We fine-tune the T5 models, while the larger models with over 10 billion parameters are used for few-shot In-context learning. Table 6 presents a comprehensive list of all models used in our experiments¹.

E.1.1 Fine-tuning

Training Setting	Configuration
optimizer	Adafactor
base learning rate	1×10^{-4}
weight decay	2×10^{-5}
decay rate	-0.8
optimizer eps	$(1 \times 10^{-30}, 2 \times 10^{-3})$
batch size	64
training epochs	20
gradient clip	1.0
gradient accumulation	1.0
warmup epochs	1
warmup schedule	cosine

Table 7: Training Setting

for the T5 models during training, which is conducted on four NVIDIA A6000 GPUs with 48GB of memory each.

In Table 7, we list the hyperparameters used to train the T5 model. We carefully selected these parameters to ensure that the within-distribution validation accuracy roughly converged. We report all peak validation set results, and in every experiment we ran, we found that within-distribution validation accuracy monotonically increased during training iterations (until it approached 100%), and we never observed overfitting. This may be due to the regular nature of the tasks we considered in the paper. We followed the Anil et al. [2022]’s setup and did not use OOD performance in model selection, as this would constitute "peaking at the test conditions". Regarding the number of training iterations, we also tried training the T5 model with more iterations in the addition task, but this did not lead to substantial differences in OOD performance (i.e., it was still equally poor).

¹The specific configurations of the GPT series of models can be found at <https://platform.openai.com/docs/model-index-for-researchers/model-index-for-researchers>

PARITY DATASET

```
def generate_parity_data(n):
    data = []
    for _ in range(100000):
        input_str = ''.join(str(random.randint(0, 1)) for _ in range(n))
        label = sum(int(x) for x in input_str) % 2
        data.append({'input': input_str, 'label': label})
    return data

parity_data = {}
for n in range(1, 41):
    parity_data[n] = generate_parity_data(n)
```

E.1.2 Few-shot In-Context Learning

For the few-shot context learning on GPT and GLM-130B models, we employ an approach inspired by the recent work on GPT-3 [Brown et al., 2020]. In this methodology, the models are provided a context consisting of a few examples of the input-output pairs in natural language format. Following this, the models are expected to generalize and perform well on the task without any explicit fine-tuning. We carefully design the context to include diverse examples that represent the range of input types and required model reasoning. Importantly, we limit the context length to be within the maximum token limits of the models. For instance, GPT-3 has a token limit of 2048. Due to limited access to the GPT family of models, we utilize the official API for these experiments². For the GLM-130B, we employ the FasterTransformer framework to set up local inference with INT4 on eight NVIDIA GeForce RTX 3090 GPUs with 24GB of memory each.

E.2 Tasks and Dataset

In this paper, all data sets related to length generalization consist of independent data sets with the same number of digits but different lengths, and each digit in the test set is unique. Therefore, there may be slight fluctuations between data sets of different lengths, but the overall trend is generally clear. To further illustrate the differences between data sets of different lengths, the following examples are provided:

Parity:	$\underbrace{110}_{\text{Length} = 3} = 0$	$\underbrace{101100}_{\text{Length} = 6} = 1$	$\underbrace{010001110101}_{\text{Length} = 12} = 0$
Reverse:	$\underbrace{abc}_{\text{Length} = 3} = cba$	$\underbrace{\text{figure}}_{\text{Length} = 6} = \text{erugif}$	$\underbrace{\text{accomplished}}_{\text{Length} = 12} = \text{dehsilpmocca}$
Addition:	$\underbrace{1 + 2}_{\text{Length} = 3} = 3$	$\underbrace{18 + 245}_{\text{Length} = 6} = 263$	$\underbrace{48864 + 964315}_{\text{Length} = 12} = 1013179$
Arithmetic Reasoning:	Joan found $\underbrace{6546634574688499}_{\text{Length} = 16}$ seashells and Jessica found $\underbrace{3855196602063621}_{\text{Length} = 16}$ seashells on the beach. How many seashells did they find together ?		

E.2.1 Data Generation Details

Synthetic Parity Dataset: We sample instances of lengths 1-40 from a uniform Bernoulli distribution. We first uniformly sample the number of ones, and then randomly shuffle the positions of each one within a fixed-length bitstring. For the experiments in **Main Figure 3**, we train T5 on lengths 10-20, with 99000 training samples per bit. For all methods, we test each bit using 1000 samples.

Synthetic Reverse Dataset: We selected a dataset of instances with lengths ranging from 1 to 40. For the experiments in **Main Figure 3**, the training set consists of 99000 samples each for lengths 10-20. Each input is a randomly generated word string of specified length, where the letters are selected uniformly at random from a set of 26 letters using a Bernoulli distribution (without necessarily having

²OpenAI's API: <https://openai.com/api/>

REVERSE DATASET

```
reverse_data = {}  
for n in range(1, 41):  
    reverse_data[n] = []  
    for _ in range(100000):  
        word = ''.join(random.choice(string.ascii_lowercase) for _ in range(n))  
        reverse_data[n].append({'input': word, 'label': ''.join(list(reversed(word)))})
```

ADDITION DATASET

```
# Generate two random n-digit numbers and their sum  
def generate_additive_example(n):  
    data = []  
    k = (n - 1) % 2  
    n = (n - 1) // 2 if n > 2 else n  
  
    for _ in range(100000):  
        a = random.randint(10**(n+k-1), 10**(n+k) - 1)  
        b = random.randint(10**(n-1), 10**n - 1)  
        data.append({'input': str(a)+'+'+str(b), 'label': a + b})  
    return data  
  
# Generate an additive data set for digits ranging from 3 to 40  
additive_data = {}  
for n in range(3, 41):  
    additive_data[str(n)] = generate_additive_example(n)
```

286 any actual meaning), and all letters are converted to lowercase. The test set for lengths 1-40 contains
287 at least 1000 test samples.

288 **Synthetic Addition and subtraction Dataset:** Addition and subtraction are fundamental arithmetic
289 operations that are commonly taught in primary school. To generate the dataset, we takes as input the
290 number of digits n and returns a list of 100000 examples. The function first calculates the remainder
291 k when $(n - 1)$ is divided by 2, and then divides $(n - 1)$ by 2 if n is greater than 2, else it sets n
292 to itself. This ensures that the length of the first number is either equal to or one digit longer than
293 the length of the second number. The function then generates 100000 examples using randomly
294 generated numbers. Specifically, it generates two numbers a and b where a is a random integer
295 between $10^{(n+k-1)}$ and $10^{(n+k)} - 1$, and b is a random integer between $10^{(n-1)}$ and $10^n - 1$. It
296 then appends each example to the list $data$ in the form of a dictionary with the input as the string
297 " $a+b$ " and the label as the sum $a+b$.

298 For the experiments in Figure **Main Figure 1**, we provided 99000 training data examples for addition
299 with numbers ranging from 3 to 10 digits in length for the T5 model. For the GPT-3.5 and GPT-4
300 models, we provided 8 few-shot samples within 10 digits. We evaluated the performance of all three
301 models on numbers ranging from 3 to 30 digits in length, with 1000 test samples per digit. On the
302 other hand, for the experiments in Figure **Main Figure 3**, we provided 99000 training data examples
303 for addition with numbers ranging from 10 to 20 digits in length for the T5 model. For the GPT-3.5
304 and GPT-4 models, we provided 8 few-shot samples within the range of 10 to 20 digits. We evaluated
305 the performance of all three models on numbers ranging from 3 to 30 digits in length, with 1000 test
306 samples per digit.

307 For subtraction, we use a similar approach.

308 E.2.2 Symbolic Reasoning Dataset

309 **Coin Flip:** We followed Wei et al. [2022]’s setup and randomly concatenated first and last names
310 from the top 1000 names in the census data (<https://namecensus.com/>) to create the <NAME>
311 token. In our work, flipping a coin corresponds to 1 and not flipping a coin corresponds to 0. To make
312 the inputs as close to English as possible without using too many symbols, we used the sentence
313 models "<NAME> flips the coin." and "<NAME> does not flip the coin." to represent whether
314 the coin was flipped or not. This task is similar to the parity task, but requires further semantic
315 understanding. We constructed a training set of 1500 samples, with 500 samples for each of 2-4

SUBTRACTION DATASET

```
# Generate two random n-digit numbers and their minus
def generate_minus_example(n):
    data = []
    k = (n - 1) % 2
    n = (n - 1) // 2 if n > 2 else n
    for _ in range(100000):
        a = random.randint(10**(n+k-1), 10**(n+k) - 1)
        b = random.randint(10**(n-1), 10**n - 1)
        if a > b:
            data.append({'input': str(a)+'-'+str(b), 'label': a - b})
        else:
            data.append({'input': str(b)+'-'+str(a), 'label': b - a})
    return data

# Generate an subtraction data set for digits ranging from 3 to 40
minus_data = {}
for n in range(3, 41):
    minus_data[str(n)] = generate_minus_example(n)
```

COIN FLIP DATASET

```
dataset = []
for i in range(500):
    # randomly choose two names from the name_list
    for o in range(2,5):
        sentence = 'A coin is heads up.'
        label = []
        for time in range(o):
            name = random.sample(names, k=1)[0]

            # randomly choose whether to flip the coin or not
            flip = random.choice([True, False])

            # generate the statement and label based on whether the coin was flipped or not
            if flip:
                sentence += f" {name.capitalize()} flips the coin."
                label.append(1)
            else:
                sentence += f" {name.capitalize()} does not flip the coin."
                label.append(0)
        sentence += ' Is the coin still heads up?'

    dataset.append({'question': sentence, 'answer': {0: 'yes', 1: 'no'}[sum(label)%2]})
```

316 coin flips. For the test set, we selected 100 non-overlapping samples for each of 2-4 coin flips, and
 317 evaluated the model every 5 steps.

318 **Last Letter Concatenation:** We followed Wei et al. [2022]’s setup and randomly concatenated first
 319 and last names from the top 1000 names in the census data to create the <NAME> token. This task
 320 requires the model to connect the last letter of each word in a concatenated name. This task requires
 321 Neural Comprehension of rules in two aspects. First, it requires the model to correctly identify the
 322 last letter of each word. Second, it requires the model to concatenate all the last letters of the words
 323 together. We concatenated 2-5 first or last names, and constructed a training set of 1500 samples,
 324 with 500 samples for each name length of 2-4. For the test set, we selected 100 non-overlapping
 325 samples for each name length of 2-4, and evaluated the model every 5 steps.

326 E.2.3 Arithmetical Reasoning Dataset

Dataset	Number of samples	Average words	Answer Format	Lience
GSM8K	1319	46.9	Number	MIT License
SingleEq	508	27.4	Number	MIT License
AddSub	395	31.5	Number	Unspecified
MultiArith	600	31.8	Number	Unspecified
SVAMP	1000	31.8	Number	MIT License

Table 8: Arithmetical Reasoning Dataset Description.

In Table 8, we summarize the information of all arithmetic reasoning datasets used in this work. We provide the links to access these datasets:

- **GSM8K**: <https://github.com/openai/grade-school-math>
- **SingleEq**: <https://gitlab.cs.washington.edu/ALGES/TACL2015>
- **AddSub**: <https://www.cs.washington.edu/nlp/arithmetic>
- **MultiArith**: http://cogcomp.cs.illinois.edu/page/resource_view/98
- **SVAMP**: <https://github.com/arkilpatel/SVAMP>

F Some examples of Neural Comprehension

In this section, we will use gray font to represent the task input, yellow font to represent the neural network output during training, and blue background to represent the neural network output during generated.

F.1 Synthetic Symbolic

Q: 1011001010 A: 1
Q: 01111011000 A: 0
Q: 1010011001110 A: 1
Q: 10000001001001 A: 0
Q: 110100011110001 A: 0
Q: 1110011001010110 A: 1
Q: 1100000111011000101 A: 1
Q: 01100000110110010001 A: 0
——(LLM’s few-shot prompt)——
Q: 011110001010101101011 A: 0

Table 9: The example of Parity

Q: neofascism A: msicsafoen
Q: betaquinine A: eniniuqateb
Q: corediastasis A: sisatsaideroc
Q: ferroelectronic A: cinortceleorref
Q: cryoprecipitation A: noitatipicerpoyrc
Q: cryofibrinogenemia A: aimenegonirbifoyrc
Q: chemocarcinogenesis A: sisenegoniracomehc
Q: ponjpcdqjuuhiviojmy A: ybmjoivihuujqdcnpjnop
——(LLM’s few-shot prompt)——
Q: helloworldhellochina A: anihcollehdrlrowolleh

Table 10: The example of Reverse

Q: 82637+3058 A: 85695
Q: 58020+96632 A: 154652
Q: 717471+58704 A: 776175
Q: 298309+702858 A: 1001167
Q: 1061462+2623780 A: 3685242
Q: 58720970+61609034 A: 120330004
Q: 364920479+78861480 A: 443781959
Q: 6050330002+211065324 A: 6261395326
------(LLM's few-shot prompt)-----
Q: 20021012+20021004 A: 40042016

Table 11: The example of Addition

Q: 82637-3058 A: 79579
Q: 96632-58020 A: 38612
Q: 717471-58704 A: 658767
Q: 702858-298309 A: 404549
Q: 2623780-1061462 A: 1562318
Q: 68720970-61609034 A: 7111936
Q: 364920479-78861480 A: 286058999
Q: 6050330002-211065324 A: 393967676
------(LLM's few-shot prompt)-----
Q: 20021012-20021004 A: 8

Table 12: The example of Subtraction

339 F.2 Symbolic Reasoning

A coin is heads up. Devin flips the coin. Maxwell does not flip the coin.
James flips the coin. Kenneth flips the coin. Is the coin still heads up?
1 0 1 1 -> 1
A coin is heads up. Ira flips the coin. Danny does not flip the coin.
Horace flips the coin. Is the coin still heads up?
1 0 1 -> 0

Table 13: The example of Coin Flip

Take the last letters of the words in Elias Earnest Milton and concatenate them.
The last letter of Elias -> s The last letter of Earnest -> t
The last letter of Milton -> n The answer is stn
Take the last letters of the words in Randolph Weldon Olin Robbie and concatenate them.
The last letter of Randolph -> h The last letter of Weldon -> n
The last letter of Olin -> n The last letter of Robbie -> e The answer is hnne

Table 14: The example of Last Letter Concatenation

340 F.3 Arithmetical Reasoning

Joan found 65466345746884996 seashells and Jessica found 38551966020636213
seashells on the beach . How many seashells did they find together ?

Joan started with 65466345746884996 seashells. She gave some to Sam. So:
65466345746884996 - 38551966020636213 = 2691437972624878

The answer is 2691437972624878

Table 15: The example of Arithmetirc Reasoning

341 References

- 342 A. Amini, S. Gabriel, P. Lin, R. Koncel-Kedziorski, Y. Choi, and H. Hajishirzi. Mathqa: Towards
343 interpretable math word problem solving with operation-based formalisms. *north american chapter
344 of the association for computational linguistics*, 2019.
- 345 C. Anil, Y. Wu, A. J. Andreassen, A. Lewkowycz, V. Misra, V. V. Ramasesh, A. Slone, G. Gur-Ari,
346 E. Dyer, and B. Neyshabur. Exploring length generalization in large language models. In A. H.
347 Oh, A. Agarwal, D. Belgrave, and K. Cho, editors, *Advances in Neural Information Processing
348 Systems*, 2022. URL <https://openreview.net/forum?id=zSkYVeX7bC4>.
- 349 T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam,
350 G. Sastry, A. Askell, et al. Language models are few-shot learners. *Advances in neural information
351 processing systems*, 33:1877–1901, 2020.
- 352 P. Clark, O. Tafford, and K. Richardson. Transformers as soft reasoners over language. In C. Bessiere,
353 editor, *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence,
354 IJCAI-20*, pages 3882–3890. International Joint Conferences on Artificial Intelligence Organization,
355 7 2020. doi: 10.24963/ijcai.2020/537. URL <https://doi.org/10.24963/ijcai.2020/537>.
356 Main track.
- 357 K. Cobbe, V. Kosaraju, M. Bavarian, J. Hilton, R. Nakano, C. Hesse, and J. Schulman. Training
358 verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.
- 359 J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. Bert: Pre-training of deep bidirectional
360 transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.
- 361 A. Giannou, S. Rajput, J. yong Sohn, K. Lee, J. D. Lee, and D. Papailiopoulos. Looped transformers
362 as programmable computers, 2023.
- 363 M. Hu, Y. Peng, Z. Huang, and D. Li. A multi-type multi-span network for reading comprehension
364 that requires discrete reasoning. *empirical methods in natural language processing*, 2019.
- 365 J. Huang, S. S. Gu, L. Hou, Y. Wu, X. Wang, H. Yu, and J. Han. Large language models can
366 self-improve. *arXiv preprint arXiv:2210.11610*, 2022.
- 367 T. Kojima, S. S. Gu, M. Reid, Y. Matsuo, and Y. Iwasawa. Large language models are zero-shot rea-
368 soners. In A. H. Oh, A. Agarwal, D. Belgrave, and K. Cho, editors, *Advances in Neural Information
369 Processing Systems*, 2022. URL <https://openreview.net/forum?id=e2TBb5y0yFf>.
- 370 D. Lindner, J. Kramár, M. Rahtz, T. McGrath, and V. Mikulik. Tracr: Compiled transformers as a
371 laboratory for interpretability. *arXiv preprint arXiv:2301.05062*, 2023.
- 372 A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein,
373 L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy,
374 B. Steiner, L. Fang, J. Bai, and S. Chintala. Pytorch: An imperative style, high-performance
375 deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and
376 R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran As-
377 sociates, Inc., 2019. URL [https://proceedings.neurips.cc/paper_files/paper/2019/
378 file/bdbca288fee7f92f2bfa9f7012727740-Paper.pdf](https://proceedings.neurips.cc/paper_files/paper/2019/file/bdbca288fee7f92f2bfa9f7012727740-Paper.pdf).

379 X. Pi, Q. Liu, B. Chen, M. Ziyadi, Z. Lin, Y. Gao, Q. Fu, J.-G. Lou, and W. Chen. Reasoning like
380 program executors. 2022.

381 S. Roy and D. Roth. Solving general arithmetic word problems. *arXiv: Computation and Language*,
382 2016.

383 N. Shazeer and M. Stern. Adafactor: Adaptive learning rates with sublinear memory cost, 2018.

384 X. Wang, J. Wei, D. Schuurmans, Q. V. Le, E. H. Chi, S. Narang, A. Chowdhery, and D. Zhou. Self-
385 consistency improves chain of thought reasoning in language models. In *The Eleventh International*
386 *Conference on Learning Representations*, 2023. URL [https://openreview.net/forum?id=](https://openreview.net/forum?id=1PL1NIMMrw)
387 [1PL1NIMMrw](https://openreview.net/forum?id=1PL1NIMMrw).

388 J. Wei, X. Wang, D. Schuurmans, M. Bosma, E. Chi, Q. Le, and D. Zhou. Chain of thought prompting
389 elicits reasoning in large language models. *arXiv preprint arXiv:2201.11903*, 2022.

390 G. Weiss, Y. Goldberg, and E. Yahav. Thinking like transformers. In *International Conference on*
391 *Machine Learning*, pages 11080–11090. PMLR, 2021.

392 Y. Weng, M. Zhu, S. He, K. Liu, and J. Zhao. Large language models are reasoners with self-
393 verification. *arXiv preprint arXiv:2212.09561*, 2022.

394 Z. Zhang, A. Zhang, M. Li, and A. Smola. Automatic chain of thought prompting in large language
395 models. In *The Eleventh International Conference on Learning Representations*, 2023. URL
396 <https://openreview.net/forum?id=5NTt8GFjUHkr>.

397 D. Zhou, N. Schärli, L. Hou, J. Wei, N. Scales, X. Wang, D. Schuurmans, O. Bousquet, Q. Le, and
398 E. Chi. Least-to-most prompting enables complex reasoning in large language models. *arXiv*
399 *preprint arXiv:2205.10625*, 2022.