

## Highlights

### **Learning neuro-symbolic convergent term rewriting systems**

Flavio Petruzzellis, Alberto Testolin, Alessandro Sperduti

- A neuro-symbolic system inspired by rewriting algorithms can learn convergent term rewriting systems from training data.
- The system exhibits systematic generalization beyond the training distribution, outperforming relevant neural baselines.
- The system can learn several convergent term rewriting systems at the same time when trained in a multi-domain setting.

# Learning neuro-symbolic convergent term rewriting systems

Flavio Petruzzellis<sup>a</sup>, Alberto Testolin<sup>a,b</sup>, Alessandro Sperduti<sup>a,c</sup>

<sup>a</sup>*Department of Mathematics, University of Padova, Via Trieste 63, Padova, 35121, Italy*

<sup>b</sup>*Department of General Psychology, University of Padova, Via Venezia 12, Padova, 35131, Italy*

<sup>c</sup>*Augmented Intelligence Center, Fondazione Bruno Kessler, Via Sommarive, Povo, 38123, Italy*

---

## Abstract

Building neural systems that can learn to execute symbolic algorithms is a challenging open problem in artificial intelligence, especially when aiming for strong generalization and out-of-distribution performance. In this work, we introduce a general framework for learning convergent term rewriting systems using a neuro-symbolic architecture inspired by the rewriting algorithm itself. We present two modular implementations of such architecture: the Neural Rewriting System (NRS) and the Fast Neural Rewriting System (FastNRS). As a result of algorithmic-inspired design and key architectural elements, both models can generalize to out-of-distribution instances, with FastNRS offering significant improvements in terms of memory efficiency, training speed, and inference time. We evaluate both architectures on four tasks involving the simplification of mathematical formulas and further demonstrate their versatility in a multi-domain learning scenario, where a single model is trained to solve multiple types of problems simultaneously. The proposed system significantly outperforms two strong neural baselines: the Neural Data Router, a recent transformer variant specifically designed to solve algorithmic problems, and GPT-4o, one of the most powerful general-purpose large-language models. Moreover, our system matches or outperforms the latest o1-preview model from OpenAI that excels in reasoning benchmarks.

*Keywords:* algorithmic learning, mathematical reasoning, out-of-distribution generalization, neuro-symbolic AI, transformer

---

## 1. Introduction

Deep learning systems have spread dramatically in the last decade thanks to their ability to automatize tasks typically carried out by humans, from basic pattern recognition (He et al., 2016), to natural language processing (Brown et al., 2020) and the generation of synthetic but stunningly realistic media content (Chang et al., 2023). However, despite these advances, neural network models struggle with tasks that require iterative and reflective reasoning, which in humans require conscious deliberation and understanding beyond pattern recognition (Kahneman, 2011). As a result, deep learning often fails in problems where the ability to reason systematically through compositional concepts is essential, such as learning to simulate symbolic algorithms or solve advanced mathematical formulas (Testolin, 2024; Davis, 2024).

On the other hand, computer programs stemming from classical artificial intelligence techniques have traditionally been very successful in the domain of mathematics and formal reasoning (Newell and Simon, 1956). In such a framework, computer scientists model real-world phenomena using two equally important and complementary formalization tools: algorithms and data structures. The latter can be viewed as a formal description of the entities involved in the modeled phenomena, while the former are descriptions of the processes that can bring the modeled entities to a desired state. Under the assumption that input data are stored in specific data structures that the algorithm was designed to process, the result of the algorithm’s execution will be predictable for the end-user. The power of algorithms can therefore be reduced to the use of *abstractions* that formally describe real-world entities and processes (Cormen and Leiserson, 2022).

Classical algorithms in computer science – such as sorting or graph traversal algorithms – can map inputs to outputs independently of the data distribution from which the input was drawn, with well-studied time and space scaling laws. On the other hand, from a statistical perspective, it is almost always assumed that the inputs to deep learning models belong to the same data distribution of training samples, making it challenging to design learning architectures that can handle out-of-distribution (OOD) test data (Hendrycks et al., 2021; Ye et al., 2021). At the same time, however, the design and implementation of classical algorithms requires a significant amount of human labor: the programmer needs to formalize each problem class using specific data structures and engineer explicit algorithms that can solve the

problem at hand. Given the immense wealth of digital data that is now available to organizations and the value they can bring if processed by computer programs, new lines of research propose to reduce the need for humans in the automation loop exploiting machine learning. In this context, information is encoded using distributed representations and is processed by manipulating numerical vectors rather than with classical algorithms and data structures (Rumelhart et al., 1986). For example, graph neural networks have recently been used to learn graph algorithms, combining the strong real-world data handling capabilities of deep learning with the theoretical guarantees of classical algorithms (Velickovic et al., 2020; Velickovic and Blundell, 2021).

In this work, we consider a class of problems from the tradition of artificial intelligence and computer science that can be formalized as convergent term rewriting systems (Baader and Nipkow, 1998). Generally speaking, rewriting systems are composed of a set of elements and a set of rules that describe how to transform those elements. Elements can be several different mathematical objects, including strings, graphs, or terms in a formula. When combined with an appropriate algorithm, rewriting systems become programs that can execute the transformation of a sequence of objects into another one by the subsequent application of the given rules. We consider here term rewriting systems, in which the elements are mathematical expressions represented as sequences of symbols, and the rules define their semantically equivalent forms. Specifically, in a *convergent* term rewriting system, rewrite rules applied sequentially always transform the input into the same final output, independently of the order of application, and sequences of rewrite rules never form loops.

We present two related neuro-symbolic architectures designed to learn convergent term rewriting systems: the Neural Rewriting System (NRS) and the Fast Neural Rewriting System (FastNRS). Both models are built upon a shared architectural blueprint inspired by rewriting algorithms. They exhibit strong generalization capabilities, approaching those of traditional term rewriting systems, but their processing dynamics emerges through learning from data rather than manual design. Both models can be trained on a limited subset of formulas and effectively generalize to more complex ones, eliminating the need for exhaustive training on all possible formulas. Such capability for out-of-distribution generalization is enabled by a modular approach informed by the rewrite mechanism and by architectural modifications to the transformer block.

We show that these models can function in a multi-domain scenario, a

setting where a single model is trained on multiple datasets simultaneously without task-specific architectural adjustments. This results in a “multipotent” system capable of solving a variety of problem instances within the considered class. Unlike the more conventional multi-task setting in machine learning (Caruana, 1997), where a shared backbone architecture is typically combined with task-specific outputs, our architecture’s algorithmic-inspired design allows the same components to be effectively applied to learn multiple term rewriting systems, enabling robust generalization without task-specific modifications.

While the NRS has been described in recent work (Petruzzellis et al., 2024c), the FastNRS is presented here for the first time, together with analysis of the performance of both systems in a multi-domain scenario. The FastNRS system improves efficiency in terms of memory usage, training, and inference time, without significantly sacrificing performance. In this article, we detail the architectural elements of both the NRS and the FastNRS and provide a comparative analysis of their performance and efficiency on the four different domains of logic, lists, integer arithmetic, and simple algebra, describing the benefits and trade-offs associated with each system.

As additional baselines that could exhibit a systematic behavior similar to the execution of convergent term rewriting systems, we consider three architectures from two independent but related streams of research: the Neural Data Router (Csordás et al., 2022), a recently proposed variant of transformer designed to achieve strong systematic generalization capabilities, which can be considered as a representative of small-scale neural architectures specialized on a single domain; OpenAI’s GPT-4o (OpenAI, 2023), one of the best performing general-purpose LLMs currently available, with strong reasoning capabilities that can further improve through prompting methods like Chain-of-Thought (Wei et al., 2022); and OpenAI’s o1-preview (OpenAI, 2024), a recently proposed LLM based on GPT-4 that has been optimized to excel in systematic reasoning tasks thanks to the production of long reasoning chains.

The remainder of this paper is organized as follows. Section 2 provides background on systematic generalization and algorithmic learning with neural networks, establishing the theoretical foundation for our work. In Section 3, we define the specific class of problems addressed in this study — formula simplification problems. In Section 4 we present the architectural blueprint of our models designed to solve these problems, and we describe each model in more detail. Section 5 details the experimental setup, including datasets composition and baselines, and Section 6 contains a presentation and anal-

ysis of the experimental results. Finally, Section 8 concludes the paper. Additional methodological details and supplementary results are provided in Appendix B and Appendix D.

## 2. Background and related works

Connectionist systems designed to process symbolic data have been proposed since the 1990s (Hinton, 1990). In the deep learning era, this line of research has seen a significant increase in interest and development, thanks to the introduction of novel neural architectures that could effectively process symbolic sequences, such as those based on external memory like the Neural Turing Machine (Graves et al., 2014) and its successor, the Differentiable Neural Computer (Graves et al., 2016). These models are designed to learn algorithmic tasks by leveraging mechanisms that mimic classical computational processes, with memory-based architectures incorporating external memory to handle complex data structures, and attention-based models focusing on selectively attending to parts of the input sequence to perform tasks like sorting or routing (Vinyals et al., 2015). Similar goals motivated parallel research efforts on the possibility of simulating the execution of classical algorithms in computer science with Graph Neural Networks (Velickovic et al., 2020). These initial contributions laid the groundwork for the framework of Neural Algorithmic Reasoning (Velickovic and Blundell, 2021), which focuses on training neural networks to perform classical algorithms on graph-based problems bridging the gap between deep learning and traditional algorithmic theory.

At the same time, inspired by the rich debate in cognitive science and linguistics about the role of rules in language acquisition (Pinker and Prince, 1988), other researchers started to investigate the capability of popular sequence-to-sequence architectures to extrapolate simple compositional rules from the training distribution and apply them to out-of-distribution test samples (Lake and Baroni, 2018). Among these approaches, one model introduced an *ad hoc* trainable component named the copy-decoder, specifically designed to assist in learning to copy parts of the input to the output (Ruiz et al., 2021); another one proposed the Neural Data Router (NDR), a variant of transformer encoder tailored to compositional generalization problems with sequential solution procedures, which we consider as a baseline in our work (Csordás et al., 2022). Other recent work explored the effectiveness of several architectural elements on the compositional generalization capabilities of transformers,

such as recursive decoding (Setzler et al., 2022), positional encodings and early stopping (Csordás et al., 2021). Both initial contributions and subsequent research efforts demonstrated that recurrent and transformer-based models can achieve varying degrees of success, yet they still struggle to learn the underlying rules systematically and reliably from training data (Hupkes et al., 2020; Csordás et al., 2021).

A research problem closely related to the principles of productivity, systematicity, and substitutivity as described in Hupkes et al. (2020) is solving mathematical problems with neural networks, where learning to apply these principles is crucial to achieve true compositionality. In this area, significant progress has been made using transformers to tackle a range of mathematical tasks, including arithmetic (Cognolato and Testolin, 2022), derivation and integration (Lample and Charton, 2019) and polynomial simplification (Agarwal et al., 2021). However, research has shown that while transformers can learn to solve generic mathematical problems, their ability to generalize and apply learned rules systematically remains limited (Saxton et al., 2019; Testolin, 2024; Davis, 2024).

As an alternative approach more aligned with the principles of compositionality identified by Hupkes et al. (2020), some researchers have explored the implementation of neural architectures inspired by symbolic rewriting systems. In early work, network weights were used to represent tokens to be rewritten in unsupervised learning settings (Komendantskaya., 2009). Other approaches included using custom feature engineering and feed-forward networks for algebraic problems (Cai et al., 2018). Yet other developments have introduced reinforcement learning-based systems that learn a general rewriting mechanism, selecting regions of a formula to simplify and applying appropriate rewriting rules, thereby aiming to achieve a more systematic and compositional handling of symbolic expressions (Chen and Tian, 2019).

Given the recent prominence of large language models (LLMs), there is also a growing interest in understanding the symbolic reasoning abilities of foundation models trained on huge corpora of text and/or code (Chen et al., 2021; Petruzzellis et al., 2024a,b). These models could be considered the pinnacle of scientific and engineering advancements in neural technology, and their proficiency in language manipulation makes them particularly relevant for investigating systematic generalization from a linguistic and cognitive science perspective. Reasoning is one of the key abilities that is believed to emerge in very large models (Wei et al., 2022), yet it remains an area of active research, with ongoing efforts aimed at achieving further improve-

ments. Symbolic reasoning tasks, which require the model to follow structured logical rules to arrive at a conclusion, can serve as a testbed for these abilities. These symbolic reasoning tasks are similar to the problems addressed in this work, as they involve synthetically generated instances that can be solved by applying simple, algorithmic rules. Examples of such tasks include coin flip prediction, last letter concatenation (Wei et al., 2022), and boolean variable assignment (Anil et al., 2022), all of which require a form of systematic rule application similar to compositional generalization benchmarks. Interestingly, research on prompt engineering techniques has shown that appropriate prompting can enhance the reasoning capabilities of LLMs also on symbolic reasoning tasks (Wei et al., 2022; Wang et al., 2023). For instance, Chain-of-Thought prompting leverages the auto-regressive nature of these models to break down complex problems into multi-step reasoning chains, enabling more effective processing of contextual information. Leveraging reasoning steps in context has also been adopted as the core strategy to improve reasoning capabilities in the new OpenAI o1 family of language models specialized for complex reasoning tasks (OpenAI, 2024), an exemplar of which we consider in this work.

### 3. Formula simplification problems

Convergent term rewriting systems are typically used to simplify mathematical formulas. Here, rules describe how expressions involving two or more operands can be rewritten into atomic elements that are semantically equivalent and represent their values. For example, in arithmetic formulas, the expression  $(15 + 5)$  can be transformed into the equivalent atomic element 20. We call these problems “formula simplification problems.” We will now formally characterize these problems and then use this formalization to describe how convergent term rewriting systems for this class of problems can be formed, i.e., how rewriting rules can be written and what algorithm is implemented by such a rewriting system.

In formula simplification problems, we can see formulas  $f \in F$  as entities that are composed of two semantically distinct elements: operators  $o \in O$  and arguments  $a \in A$ . Arguments can be either atomic elements  $e \in E$ , such as integers, which are also the final values of any formula, or other formulas. Furthermore, since these problems can be solved by convergent term rewriting systems and thus always have exactly one final value, the following equality holds for any formula:  $f = o(a_1, \dots, a_n) = e$ , where

$o \in O$ ,  $e \in E$  and  $a_j \in F \cup E$ ,  $\forall j \in [1, n]$ . Finally, we can define leaf formulas  $F^L \subset F$  as the subset of formulas whose arguments are all atomic elements:  $f^L = o(a_1, \dots, a_n)$  s.t.  $a_k \in E$ ,  $\forall k \in [1, n]$ .

For any problem, we can define a set of rewriting rules  $r \in R$  which map leaf formulas to their values:  $r : F^L \rightarrow E$ ,  $\forall r \in R$ . This set defines a convergent term rewriting system for any formula simplification problem. Indeed, iteratively applying the rewriting rules described above in any order, the initial formula can be simplified to an atomic final element  $e$ . The algorithm implemented by the rewriting system thus consists of the execution of four steps, which are repeated until the formula becomes an atomic value: **1.** pick any valid rewriting rule from the set;<sup>1</sup> **2.** find in the input formula all the elements that match the left-hand side of the rewriting rule; **3.** apply the rewriting rule to the elements found and compute the substitution; **4.** replace the elements with the computed values. This algorithm serves as a blueprint for the design of our architectures, providing strong guarantees on the reliability of the models, regardless of how they are implemented.

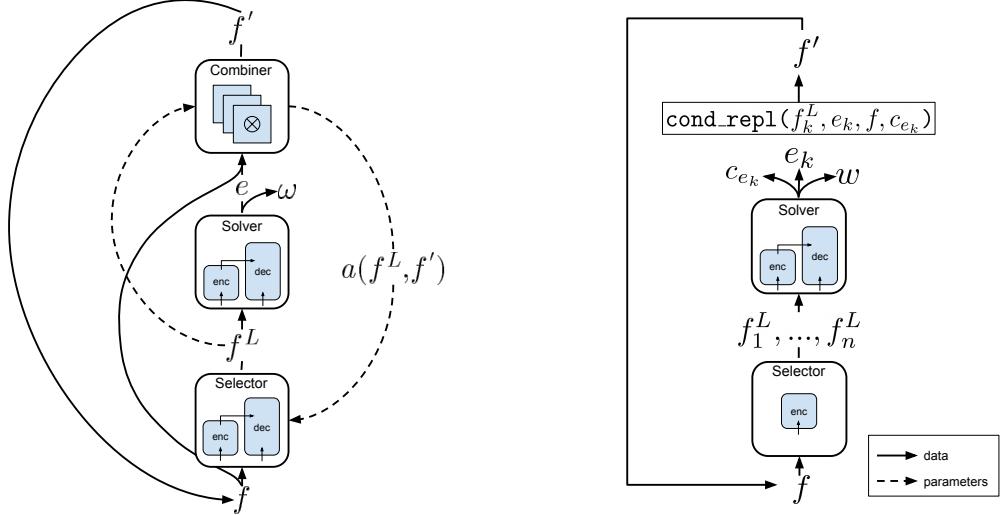
#### 4. Neuro-Symbolic architectures to learn rewriting systems

Traditional term rewriting systems rely heavily on manually crafted rules and algorithms tailored to specific problems, which require significant human expertise and effort. In contrast, our approach aims to *learn* both the rewriting rules and the contexts in which these rules should be applied. By mirroring the algorithm’s steps in the architecture design itself, we adopt a structured approach similar to that of classical algorithms. At the same time, combining this with the flexibility of learning-based methods, we obtain a versatile system that can be applied to different problems within the class we consider. This adaptability is particularly advantageous in multi-domain scenarios, where the same architecture can be effectively employed for diverse problems, as we demonstrate with experimental results.

Compared to other neural systems, such as general-purpose Large Language Models (LLMs) and specialized neural architectures for compositional reasoning, our system balances generality and reliability. Indeed, while LLMs can tackle a wide range of tasks, they lack the specialized focus needed to provide strong guarantees in solving structured problems like those that can be

---

<sup>1</sup>In step **1.**, a rewriting rule that can be applied to the current formula should be chosen, i.e., one whose left-hand side appears in the formula.



(a) NRS architecture. The three modules in the NRS operate sequentially, but the Selector and the Combiner also interact via the agreement score  $a(f^L, f')$  to produce the Selector output, as described in Paragraph 4.1.1.

(b) FastNRS architecture. The Selector module in the FastNRS operates by selecting  $n$  leaf formulas in parallel with a text-segmentation mechanism, as described in Paragraph 4.2.1. Each leaf formula is then processed independently by the Solver.

Figure 1: Schematic representations of the NRS and FastNRS architectures. Both architectures implement the three modules of the algorithmic blueprint described in Section 4. The models process input formulas  $f$ , selecting one or more leaf formulas  $f^L$ . Solver modules simplify leaf formulas to atomic values  $e$  and produce a special end-of-computation token  $w$ . Combiner modules produce simplified formulas  $f'$ .

addressed by term rewriting. On the other hand, small-scale transformer-based architectures designed for compositional reasoning tend to provide stronger generalization guarantees, but the spectrum of tasks they can handle is limited, if defined at all (Zhou et al., 2024). Our neuro-symbolic system is more specialized than LLMs but more versatile than narrowly specialized architectures, offering a structured and more reliable solution for solving formula simplification problems across different domains.

Differently from other neuro-symbolic architectures proposed in the literature, our system does not include any symbolic AI component (e.g., a symbolic solver). We describe our system as neuro-symbolic since it is composed of neural modules whose interaction schema is informed by the rewriting algorithm: In the taxonomy of neuro-symbolic AI proposed by Kautz (2022), our system could be grouped in the class of neural architectures whose structure is obtained using symbolic rules as a template, called Neuro-{Symbolic}

systems by the author.

The algorithm for resolving mathematical formulas we described in Section 3 can be implemented in a neuro-symbolic architecture in various ways, depending on design choices and priorities. In this work, we present two neuro-symbolic implementations of this algorithmic blueprint: the Neural Rewriting System (NRS) and the Fast Neural Rewriting System (FastNRS). Both these architectures are composed of three modules, the Selector, the Solver and the Combiner, which handle steps **1.** and **2.**, **3.** and **4.** of the algorithm, respectively. Note that in a neural implementation, picking a valid rule can coincide with identifying a part of the input that can be simplified, as will be clear from our implementation of the Solver. These designs, represented schematically in Figures 1a and 1b, differ mainly in how the Selector module is implemented, corresponding to steps **1.** and **2.** of the algorithm—the identification of matching elements for rule application. In the Neural Rewriting System, only one element matching the left-hand side of a rewriting rule is selected at a time. On the other hand, the Fast Neural Rewriting System selects and replaces multiple elements in parallel by framing the problem as a text segmentation task, allowing for a more efficient implementation at the expense of some accuracy.

---

**Algorithm 1** Pseudo-code of the NRS execution. The model executes the Selector, Solver and Combiner in a pipeline. Algorithm 2 describes the execution of the Selector.

---

```

1: function NRS( $f$ )
2:   while True do
3:      $f^L, c(f^L), a(f^L, f) \leftarrow \text{SELECTOR}(f)$ 
4:     if  $a(f^L, f) \neq 1$  then
5:       return  $\emptyset$ 
6:     end if
7:      $e \leftarrow \text{SOLVER}(f^L)$ 
8:     if  $e = \omega$  then
9:       return  $f$ 
10:    end if
11:     $f \leftarrow \text{COMBINER}(f, f^L, e)$ 
12:   end while
13: end function
```

---

#### 4.1. The Neural Rewriting System (NRS)

The general functioning of the architecture can be described as the iterative execution of the Selector, Solver and Combiner modules in a pipeline. A formal description in pseudo-code of the NRS execution is given in Algorithms 1 and 2.

---

**Algorithm 2** Pseudo-code of the NRS Selector execution. The TRFM function represents the transformer. The model conditionally applies Dynamic Windowing (lines 9-13) depending on the input length.

---

**Require:** parameters  $M, T$

```

1: function SELECTOR( $f$ )
2:    $L \leftarrow []$ 
3:   for  $i = 1 \rightarrow M$  do
4:     if  $|f| < T$  then
5:        $\hat{f}^L, c(\hat{f}^L) \leftarrow \text{TRFM}(f)$ 
6:        $a(\hat{f}^L, f) \leftarrow \frac{\max CNN_{\hat{f}^L}(f)}{|\hat{f}^L|}$ 
7:       Append  $\langle \hat{f}^L, c(\hat{f}^L), a(\hat{f}^L, f) \rangle$  to  $L$ 
8:     else
9:        $k \leftarrow \text{floor}(|f| \cdot \frac{i \bmod 20}{20})$ 
10:       $f_w \leftarrow w(f, k)$ 
11:       $\hat{f}^L, c(\hat{f}^L) \leftarrow \text{TRFM}(f_w)$ 
12:       $a(\hat{f}^L, f_w) \leftarrow \frac{\max CNN_{\hat{f}^L}(f_w)}{|\hat{f}^L|}$ 
13:      Append  $\langle \hat{f}^L, c(\hat{f}^L), a(\hat{f}^L, f_w) \rangle$  to  $L$ 
14:    end if
15:  end for
16:  Sort  $L$  by  $a(\hat{f}^L, f)$  and  $c(\hat{f}^L)$ 
17:  return  $L[0]$ 
18: end function

```

---

##### 4.1.1. The Selector module

The Selector module in the Neural Rewriting System is responsible for identifying an element in the input formula that matches the left-hand side of a rewriting rule. As mentioned earlier, it is designed to solve a sequence-to-sequence task. Formally, it implements the  $sel : F \rightarrow F^L$  function, i.e. it is trained to map a formula, which we assume to always be syntactically

correct, to a leaf formula appearing therein. In analogy to what happens in humans when they deploy object-based attention to locate algebraic sub-expressions that can be simplified (Marghetis et al., 2016), the Selector is trained to identify the last leaf formula occurring in the input formula on which a rewriting rule can be applied. For example, given the arithmetic expression  $(12+(3-(4+5)))$  the Selector’s task is to correctly identify the solvable leaf formula  $(4+5)$ .

We use a variant of the transformer encoder-decoder (Vaswani et al., 2017) to implement the core of the NRS Selector. In order to achieve strong length generalization capabilities, we make two modifications to the vanilla transformer. First, we follow recent evidence showing that length generalization in transformers can be influenced by the choice of positional encodings, especially when, at test time, these fall out of the range observed during training (Csordás et al., 2021; Ruoss et al., 2023; Kazemnejad et al., 2023). We thus use Label-based Positional Encodings (Li and McClelland, 2022) to enable the Selector to identify leaf formulas in very long sequences. The positional information of an input sequence of  $L$  tokens is thus encoded in the following way: given a sequence of  $N$  sinusoidal positional encodings, where  $N$  is a large number that represents the maximum expected length of an input,  $L$  integers are sampled in the interval  $[0, N - 1]$  and then sorted. The encodings found in the positions corresponding to the sampled integers are then summed to the embeddings of the tokens in the input sequence before the forward pass. Notice that the sampling and sorting mechanisms are applied internally in the transformer, similar to the pooling operations in convolutional networks. Furthermore, this type of positional encoding operates as a sort of data augmentation mechanism and thus is not learned and is not involved in the backpropagation of the errors.

As a second modification, we constrain the receptive field of the self-attention layer of the encoder. This choice was motivated by both intuition and extensive experimentation. Indeed, the Selector more likely learns a function with good length generalization properties in a smaller search space that contains functions with minimal dependencies on parts of the sequence that do not correspond to leaf formulas. Furthermore, identifying leaf formulas is a local problem in any part of the input sequence, i.e., it can be solved without integrating information carried by tokens located in distant parts of the input sequence. Therefore, we mask all entries in the self-attention matrix of the encoder but the ones around the main diagonal (i.e., we make them  $-\inf$ ). The active values in the self-attention matrix are thus located

in a diagonal window that is  $2k + 1$  tokens wide, where  $k$  is a hyperparameter. Preliminary experiments showed that models with vanilla self-attention achieved worse out-of-distribution generalization, and hyperparameter selection demonstrated that the strongest generalization can be achieved with a narrow diagonal window.

Other than these architectural modifications, the NRS Selector includes two specialized mechanisms – the multi-output generation and the dynamic windowing – that enhance accuracy and reliability, ensuring higher resilience to noise and errors in the neural network’s outputs. The multi-output generation mechanism was introduced after we observed experimentally that it can be useful to repeat the auto-regressive generation of transformer outputs. After sampling several output sequences from the probability distribution derived from the decoder’s outputs, we choose the best one considering both a measure of confidence of the Selector and a measure of input-output agreement computed by the Combiner.

Given the specialized purpose of the Selector, we can see each output of the module as a candidate leaf formula  $\hat{f}^L$ . We generate any token  $\hat{f}_i^L$  in an output sequence  $\hat{f}^L$  by sampling from the probability distribution obtained applying the softmax function to the logits produced by the final fully-connected layer of the decoder. We do not use any temperature parameter when sampling the output tokens. For any input formula  $f$ , we repeat the stochastic generation process  $M$  times, thus generating a sequence of candidate leaf formulas  $\hat{F}^L = \langle \hat{f}^{L,1}, \dots, \hat{f}^{L,M} \rangle$ . We define the confidence of the Selector on any  $\hat{f}^{L,j}, 1 \leq j \leq M$  as the joint probability of sampling its tokens:  $c(\hat{f}^{L,j}) = \prod_{i=1}^N p_i^j$ , where  $N$  is the number of tokens in  $\hat{f}^{L,j}$ , and  $p_i^j$  is the probability to sample token  $\hat{f}_i^L$  in  $\hat{f}^{L,j}$ . We also define an agreement score  $a(\hat{f}^{L,j}, f) \in [0, 1]$  which gives information on the fraction of  $\hat{f}^{L,j}$  that is exactly present in the input formula  $f$ . This measure is computed by the Combiner and thus it is formally defined in Section 4.1.3. We then select the final output  $f^L$  of the Selector as the one with the highest confidence which has an agreement score equal to 1 — that is, it matches the input sequence exactly. More formally,  $f^L = \hat{f}^{L,j} \in \hat{F}^L$  s.t.  $c(\hat{f}^{L,j}) \geq c(\hat{f}^{L,k}) \forall j, k \in [1, M] \wedge a(\hat{f}^{L,j}, f) = 1$ .

We also implement a dynamic windowing mechanism on longer input sequences that allows us to increase the model’s generalization capacity on complex problem instances. The core idea behind this mechanism is to repeat the process of selecting a leaf formula several times, changing each time the

*window* of the input formula that the Selector observes, and then relying on the confidence  $c(\hat{f}^L)$  to pick the best output. We apply this mechanism on top of multi-output generation by modifying its behavior for sequences longer than a given threshold  $T$ . Given an input formula  $f$ , if  $|f| < T$  the computation is executed as described before. Otherwise, we generate  $M$  copies of the input  $\langle f^{(1)}, \dots, f^{(M)} \rangle$ , whose lengths will be reduced by applying a window function  $w$ . Considering any input  $f$  as a sequence  $f_1, \dots, f_N$  of  $N$  tokens, we define the window function  $w(f, k) = f_{k+1}, \dots, f_N$  which reduces the length of the input by giving as output its last  $k$  tokens. Since the Selector is trained to output the last leaf formula appearing in the input, the window function reduces the input length starting from the first tokens. We divide the sequence of copies of the input  $\langle f^{(1)}, \dots, f^{(M)} \rangle$  into 20 groups  $F^{(1)}, \dots, F^{(20)}$  of equal size. Intuitively, in each group the length of the input is reduced by a different percentage of tokens. More formally, the window function will be parameterized by  $k = \text{floor}(|f^{(i)}| \cdot \frac{j}{20}) \forall f^{(i)} \in F^{(j)}, \forall j \in [1, 20]$ . We then pick the final leaf formula using the confidence and agreement scores, as described in the previous paragraph. This ensures that the model can observe the whole input sequence and select a leaf expression in the part of the input where it can identify one with more confidence.

#### 4.1.2. The Solver module

The Solver a central component in our system. Indeed, as we described in Section 1, classic rewriting systems are composed of a set of elements and a set of rules, which are then used in the algorithm to transform sequences. In our neuro-symbolic architecture, both elements and rewriting rules are represented sub-symbolically in the Solver, which rewrites relevant parts of the input.

Given a leaf formula  $f^L$  by the Selector, the Solver is trained to produce the equivalent reduction  $e$  according to the corresponding rewriting rule. Therefore, valid elements and rewriting rules are implicitly stored in the network weights through optimization. For example, given the leaf formula  $(4+5)$ , the Selector produces its value,  $9$ . The Solver also learns to recognize the termination state of the computation, signaling when such a state is reached. Given atomic elements representing the final value of a formula, such as the number  $9$  for an arithmetic formula, it is trained to map them to the special symbol  $\omega$ , indicating the end of computation. During training, the Solver only observes well-formed leaf formulas and atomic values.

We frame the Solver task as a sequence-to-sequence problem. We imple-

ment it as a transformer encoder-decoder without any modification since it learns input-output mappings corresponding to the rules.

#### 4.1.3. The Combiner module

The last module in the architecture is the Combiner, a neural implementation of the function  $\text{com} : F \times F^L \times E \rightarrow F$ . Its purpose is thus to produce a simplified version of the original formula, given the formula itself  $f$ , the leaf formula  $f^L$  identified by the Selector, and its reduction  $e$  computed by the Solver.

In order to carry out its task, the first operation that the Combiner must perform is finding the position in  $f$  where the leaf formula  $f^L$  appears. We notice that the convolution is a suitable operation to detect which portion of an input sequence has the highest match with another sequence used as a filter, so we implement this operation using a 2D Convolutional Neural Network (CNN) whose filters are set dynamically at execution time using the output of the Selector, rather than being learned with backpropagation. For example, if we have the arithmetic expression  $(12 + (3 - (4 + 5)))$  as input, using the leaf formula  $(4 + 5)$  as the filter of a 2D CNN we can obtain a signal of correspondence between the leaf formula and the input expression, and therefore identify if the leaf formula is present in the input and where it is located.

More precisely, we represent both the input sequence  $f$  and the leaf formula  $f^L$  as sequences of one-hot vectors over the same vocabulary. Since the leaf formulas found for different sequences in a batch can have different lengths, we pad each one with zeros to prevent the padding to match in the input. Then, we set the filter of the 2D CNN to the 1-hot representation of  $f^L$ . We refer to the CNN parameterized in this way as  $\text{CNN}_{f^L}$ . Doing so allows us to obtain from the output of the convolution both information on the location of the best match of  $f^L$  in  $f$  and on the number of tokens in  $f^L$  that match  $f$  exactly in some point. Indeed, we can compute the location of the best match as  $\text{pos}(f^L, f) = \text{argmax } \text{CNN}_{f^L}(f)$ . Furthermore, we can calculate the agreement score  $a(f^L, f) = \frac{\max \text{CNN}_{f^L}(f)}{|f^L|}$ , where  $|f^L|$  is the number of tokens in  $f^L$ . Dividing by  $|f^L|$  makes the score normalized, which allows us to compare the agreement scores of leaf formulas with different lengths. Indeed, as described in Section 4.1.1, the Selector uses this score for multi-output generation to discard the outputs that do not have an exact match in the input formula. Notice that in this case the CNN is parameterized using

candidate leaf formulas  $\hat{f}_j^L$  whose accuracy scores with  $f$  are compared. If there is no Selector output such that  $a(\hat{f}_j^L, f) = 1$ , the computation on the input sequence  $f$  is stopped, and this is considered a failure of the model.

After finding the position of the leaf expression in  $f$ , the Combiner replaces  $f^L$  with  $e$  in  $f$ , to compute the simplified version of the formula  $f'$ . We implement this operation as a deterministic operator with input  $f$ ,  $f^L$ ,  $e$ , and  $\text{pos}(f^L, f)$ .

---

**Algorithm 3** Pseudo-code of the FastNRS execution. The model iterates through the leaf formulas extracted by the Selector and conditionally replaces them in the main formula.

---

```

1: function FASTNRS( $f$ )
2:   while True do
3:     mask  $\leftarrow$  SELECTOR( $f$ )
4:      $\langle f_1^L, \dots, f_n^L \rangle \leftarrow$  EXTRACT(mask,  $f$ )
5:     replaced  $\leftarrow$  False
6:     for  $f_k^L$  in  $\langle f_1^L, \dots, f_n^L \rangle$  do
7:        $e_k, c_{e_k} \leftarrow$  SOLVER( $f_k^L$ )
8:       if  $e_k = \omega$  then
9:         return  $f$ 
10:      end if
11:       $f, \text{replaced} \leftarrow \text{cond\_rep1}(f_k^L, e_k, f, c_{e_k})$ 
12:    end for
13:    if replaced = False then
14:      return  $\emptyset$ 
15:    end if
16:  end while
17: end function
```

---

#### 4.2. The Fast Neural Rewriting System (FastNRS)

The general functioning of the architecture can be described as the iterative execution of the Selector, the Solver and a deterministic `cond_repl` function in a pipeline. A pseudo-code description of the FastNRS execution is shown in Algorithm 3.

##### 4.2.1. The Selector module

Unlike the NRS, the FastNRS Selector is implemented using only a transformer encoder. This module shares the same core architecture as the trans-

former encoder used in the Neural Rewriting System. Specifically, we use Label-based Positional Encodings to enable the Selector to identify leaf formulas within very long sequences, and we constrain the receptive field of the self-attention layer to obtain localized attention on the close neighbors of each token. In FastNRS, the Selector is designed to solve a text-segmentation task. Formally, it implements the function  $\text{multisel} : F \rightarrow F^{L^n}$  where  $F^{L^n}$  represents the  $n$ -ary cartesian product of the set of leaf formulas  $F^L$ . The function maps a formula to one or more leaf formulas within it, corresponding to the left-hand sides of applicable rewrite rules.

In this implementation, given a sequence of tokens, the transformer performs a binary classification task on each token independently. A positive label indicates that a token is part of a leaf formula and will be selected for rewriting, while a negative label marks tokens that will not be selected. Given an input formula  $f$ , the Selector produces a mask over the input in which all parts of the formula that cannot be rewritten are masked. Using these masks, leaf formulas are extracted from the input, and a sequence of strings is obtained. Each string should correspond to the left-hand sides of some rewriting rule, and thus, it is given as input to the Solver, which computes the substitution according to the corresponding rule.

#### 4.2.2. The Solver module

The Solver module in FastNRS shares the same architecture as the Solver module in the NRS and is designed to solve the same problem: applying the appropriate rewriting rule to compute the required substitutions. Also in this case, if the Solver outputs the  $\omega$  symbol signaling the end of computation, the algorithm stops. Additionally, in FastNRS, we measure the confidence of the Solver’s outputs, which plays a critical role in guiding the execution of the FastNRS Combiner module. This confidence measure helps ensure that only high-confidence outputs are used in the subsequent steps, enhancing the reliability of the overall system. We define the confidence of the Solver on any output  $e$  as the joint log-probability of sampling the output tokens from the distribution obtained by applying the softmax function to the logits produced by the final fully-connected layer of the decoder. Formally,  $c_e = \sum_{i=1}^N \log(p_i)$ , where  $N$  is the number of tokens in  $e$ , and  $p_i$  is the probability of sampling token  $e_i$  in  $e$ .

#### 4.2.3. The Combiner module

In contrast to the Neural Rewriting System (NRS), the Combiner module in FastNRS is not implemented using a neural architecture. Specifically, we do not use a convolutional neural network (CNN) to extract the position signal of the leaf formula identified by the Selector. Thanks to the text-segmentation implementation of the Selector, we can directly trace back the position of the identified leaf formula(s) within the input.

As a result, the Combiner module is implemented as a deterministic function, `cond_rep1`, which takes the original formula  $f$ , the identified leaf formula  $f^L$ , its replacement  $\text{sol}(f^L)$  computed by the Solver, and the measure of the Solver’s confidence  $c_{e_k}$  as inputs. This confidence measure handles cases where the Solver output may contain errors. Indeed, despite the strong length generalization properties guaranteed by the modifications made to the Selector, minor defects in the segmentation of out-of-distribution sequences can still occur. Such defects could cause the extraction of corrupt parts of the input formula that do not correspond to the left-hand side of any rewriting rule, thus leading to meaningless substitutions computed by the Solver. Any  $f^L$  will thus be replaced with  $\text{sol}(f^L)$  in  $f$  only if the corresponding measure of confidence of the Solver output  $c_{e_k}$  is sufficiently high. Intuitively, the measure  $c_{e_k}$  reflects the distance of the input from the training distribution of true left-hand sides of rewriting rules, and thus allows the identification of parts of the input that are not valid left-hand sides with a sufficient degree of accuracy. In our experiments, we set the confidence score threshold depending on the distribution of this quantity on the training samples, as detailed in Section 5.2.

If the input mask is drastically corrupted, and no  $e_k$  has a sufficiently high confidence score  $c_{e_k}$ , the computation is interrupted and this is considered an error of the model.

## 5. Experimental Setup

### 5.1. Datasets

We benchmark the proposed architecture on four formula simplification problems from different domains: logic, operations on lists, arithmetic, and algebra. For all problems, formulas are generated automatically, and their difficulty is determined by specifying the desired nesting level of the formula. Any formula is nested at each level in two points: exactly two arguments

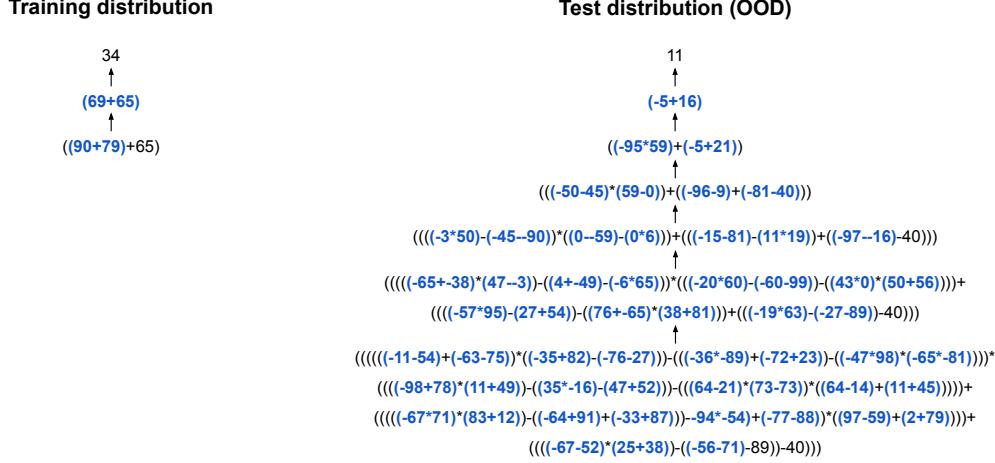


Figure 2: Visual representation of the simplification process of samples from the training set and the out-of-distribution (OOD) test set. The input parts that are simplified at each step are highlighted in blue.

in the formulas on that level will be other formulas. We now describe the formulas for each domain in more detail.

### 5.1.1. Logic

We build a dataset of nested logical formulas where the logical operators `AND`, `OR` and `NOT` are applied to non-grounded literal variables, represented by lowercase letters in  $\{a, \dots, z\}$  or grounded logical variables `True` and `False`. Formulas are generated automatically specifying the desired number of nesting levels. Unlike the other three datasets, logical formulas are nested up to 12 times, thus requiring more steps to be solved. Each logical formula can be reduced either to a non-grounded literal variable or to a logical value between `True` and `False`. For example, the logical formula  $((z \text{ OR } (z \text{ OR } (b \text{ AND } \text{False}))) \text{ OR } z) \text{ AND } (((j \text{ OR } \text{False}) \text{ AND } \text{True}) \text{ AND } \text{False}) \text{ OR } \text{True})$  is nested 5 times, contains the literal variables `b`, `j` and `z`, the logical values `True` and `False`, and evaluates to `z`.

### 5.1.2. ListOps

The ListOps dataset (Nangia and Bowman, 2018) was designed to assess neural networks’ ability to construct parse trees for nested formulas. Initially, the dataset featured formulas with operations on integer lists, such as minimum, maximum, median, and sum modulo 10. We adapted the ListOps

dataset to ensure that each nesting level had exactly two nesting points and allowed for specifying the number of arguments at any level. Focusing on the system’s ability to generalize on deeply nested formulas rather than mastering specific operations, we limited the operations to minimum, maximum, and sum modulo 10.

#### 5.1.3. Arithmetic

We created formulas using sum, subtraction, and multiplication operations between two integers sampled from the interval  $[-99, 99]$ . Since this study does not explore the ability to generalize to numbers with more digits than those encountered during training, we applied modulo 100 to the intermediate results obtained throughout the solution process.

#### 5.1.4. Algebra

We focus on a subset of algebraic formulas that can always be deterministically simplified to a minimal form. These formulas consist of sums and subtractions between two monomials, with the final value always being a monomial. The numerical coefficients of the monomials were sampled from the interval  $[-99, 99]$ , and each monomial could include up to four literal variables chosen from  $\{a, b, x, y\}$ . All monomials in a given formula shared the same literal variables. Similarly to the Arithmetic problem, all intermediate numerical values were computed modulo 100 when determining the formula’s final value.

## 5.2. Models

### 5.2.1. Neural Rewriting System

We describe here how we built the training and validation sets for the Selector and Solver modules for both the NRS and the FastNRS. A visual representation of the solution process of samples from the training and test distributions is shown in Figure 2. Statistics on all development splits used to train the models are provided in Appendix A. Further methodological details can be found in Appendix B.1.

In the training set of the Selector module, we included formulas with nesting levels of 1, 2, and 3 for all problems, along with atomic elements representing the final value of the initial formula. Simplifying formulas iteratively by reducing leaf formulas generates several intermediate simplifications

of the original formula. To demonstrate the full solution process to the Selector, we also included these intermediate formulas as steps in the training set.

We created a separate in-distribution validation set with samples mirroring the structural characteristics of those in the training set. Unlike typical machine learning tasks, where models are tested on the same data distribution they were trained on, we aim for the Selector to demonstrate out-of-distribution (OOD) generalization abilities, identifying leaf formulas even in longer inputs than those encountered during training. Therefore, we also developed a distinct OOD validation set featuring formulas with higher structural complexity, using this set for model selection. For all problems, we included in this set samples with nesting levels of 4, 5, and 6. To choose the most capable model throughout the iterative resolution process, we also added formulas representing examples of intermediate resolution steps. To manage the structural complexity of the formulas, we balanced the OOD validation sets across the nesting levels of the leaf expressions.

The training and validation sets for the Solver contained two types of samples: leaf formulas, which were mapped to their equivalent atomic elements, and atomic elements, which were mapped to the end-of-computation special symbol  $\omega$ . To prevent bias toward solving leaf formulas, we generated training batches that included both types of samples with equal probability.

Both the NRS and the FastNRS behavior can be modulated by choosing the value of some hyperparameters at inference time. The values of threshold  $T$  that regulates the Dynamic Windowing mechanism in the NRS Selector was chosen by examining the average Selector confidence score for inputs of the same length. We define these thresholds as 150 for ListOps and algebraic formulas, and 125 for arithmetic formulas, while we do not employ the mechanism on formulas in the Logic domain. We provide a representation of the average confidence score values in Appendix C. The values of the threshold on Solver confidence that regulates the `cond_rep1` function were chosen based on the distribution of these scores on training samples, as mentioned in Section 4.2. The thresholds used were -6 for ListOps, -2 for Arithmetic, -3 for Algebra, and -0.005 for Logic. The distributions of Solver confidence scores on training samples, are provided in Appendix D.

### 5.2.2. Neural Data Router

The Neural Data Router (NDR) is a modified transformer encoder designed to tackle algorithmic problems with robust out-of-distribution general-

ization capabilities. Previously, this model has been tested on relatively simple algorithmic benchmarks, such as solving formulas in the original ListOps dataset and handling basic arithmetic formulas with single-digit integers, which closely resemble the problems we address. However, the key difference in our Arithmetic and Algebra problems is the increased complexity of the operands. Additionally, we employ significantly fewer and less complex samples during training, as the NDR was initially trained on arithmetic and ListOps formulas containing up to 5 nested operations.

We made a minor modification to the architecture to adapt the model to our specific problems. In the original work, the problems always resulted in a single-digit integer, which the model was trained to output as the first token in the sequence generated by the encoder. Since this is not generally applicable to our problems, we read the final answer from the first  $k$  positions of the sequence produced by the encoder, where  $k$  is the maximum length of a problem’s targets.

We constructed all development sets for the NDR using the same top-level formulas included in the analogous sets for the Selector. Following the original experimental protocol, we ensured that the training set was balanced across nesting levels. Similar to the Selector module training, we created both in-distribution and out-of-distribution validation sets, using the latter to optimize hyperparameters through a Bayesian search using the Weights & Biases platform in the same hyperparameters intervals described in the original work. The final hyperparameter values for each task are detailed in Appendix B.3.

### 5.2.3. OpenAI GPT-4

In our experiments, we evaluate the performance of OpenAI’s GPT-4 on the nested formulas in the four domains. Using specialized prompts is currently considered the most effective method to improve the reasoning capabilities in large language models by researchers and practitioners. Specifically, Chain-of-Thought (CoT) prompting has been found to enhance the performance of large language models on reasoning tasks by facilitating step-by-step solution procedures. We opted to prompt GPT-4 using a combination of self-consistency prompting (Wang et al., 2023) and zero-shot Chain-of-Thought (CoT) (Kojima et al., 2022). Zero-shot CoT is a simpler alternative to traditional CoT prompting, achieving comparable performance on reasoning benchmarks without the need to engineer exemplars for few-shot reasoning. This is done by simply initiating the model’s response with the

sentence: “Let’s think step-by-step.” Once the model generates a response, it is prompted again to produce a well-formatted output. Self-consistency prompting leverages the idea that reasoning problems can have multiple valid paths leading to the same conclusion. Thus, we generate 10 outputs for each input and select the most consistently produced one. This approach enhances confidence in the model’s output and significantly improves accuracy, leading to a marked improvement in performance.

The zero-shot CoT prompt was designed by providing the model with a brief description of the problem and then asking it to solve it. For instance, the zero-shot CoT prompt for the ListOps input sample [MIN[SM54] [MIN39]] is: “*MIN, MAX, and SM are operators on lists of single-digit integers, representing minimum, maximum, and sum modulo 10, respectively. Solve the following expression using these operators: [MIN [SM 5 4] [MIN 3 9]].*” Following this initial prompt, the model was subsequently prompted a second time to provide a well-formatted final answer.

#### 5.2.4. OpenAI o1-preview

OpenAI has recently released a new family of highly capable models designed to excel in complex reasoning tasks. These models build on previous versions (like GPT-4) but focus on spending more time “thinking” before responding, making them particularly suitable for domains such as mathematics, coding, and logical reasoning. The o1 series introduces several new features, including “reasoning tokens”, which are assumed to represent the model’s internal thought process. There are currently two versions available: o1-mini, which has been designed for efficiency, and o1-preview, which is larger, slower, and more expensive, but also more accurate. In this work, we therefore only focus on the latter, more powerful model.

## 6. Results

In this section, we present the evaluation of the Neural Rewriting System and the Fast Neural Rewriting System, focusing on performance and efficiency, both in single-domain and multi-domain scenarios. Since the final target of any formula corresponds to an atomic value, we measure the performance of the models on all tasks using Sequence Accuracy, i.e. the exact match between model output and target sequence. The test sets on which all models are evaluated are composed of 100 formulas per nesting level. We

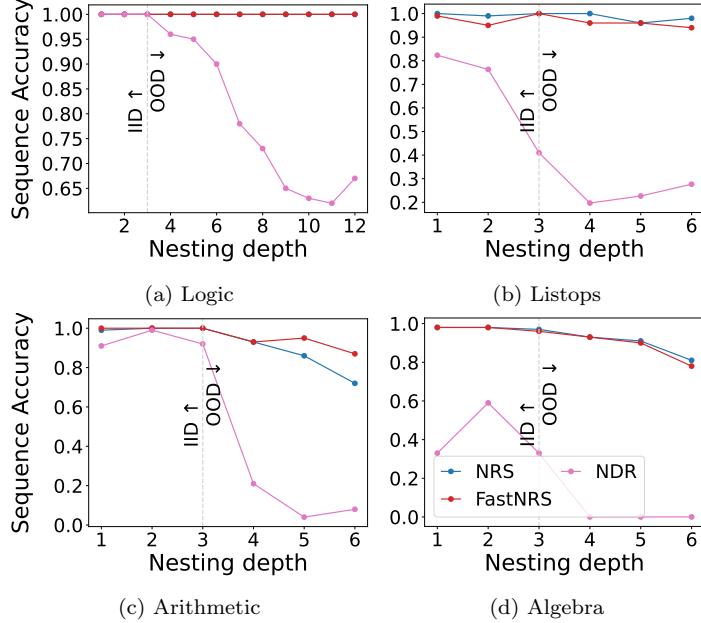


Figure 3: Performance of FastNRS, NRS, and NDR on each domain. Sequence accuracy is measured on data splits of 100 samples.

observed non-significative variance across runs, which we therefore do not report.

### 6.1. Learning domain-specific convergent term rewriting systems

In this section, we evaluate the performance of both models across all four datasets — Logic, ListOps, Arithmetic, and Algebra — in a single-domain scenario. We compare the NRS and the FastNRS to the Neural Data Router (NDR), which constitutes a neural baseline that has also been separately trained on individual tasks.

The performance of the models on all domains is represented in Figure 3. Across all datasets, the models show similar performance on in-distribution samples, with NDR generally performing the worst. However, on out-of-distribution samples, the baseline models exhibit a much sharper decline in performance compared to both the NRS and the FastNRS.

Interestingly, the FastNRS is more accurate than the NRS on deeply nested arithmetic formulas. Therefore, in this case, the design choices in the FastNRS yield a significant improvement in terms of performance other than efficiency. By examining the type of errors committed by both systems on

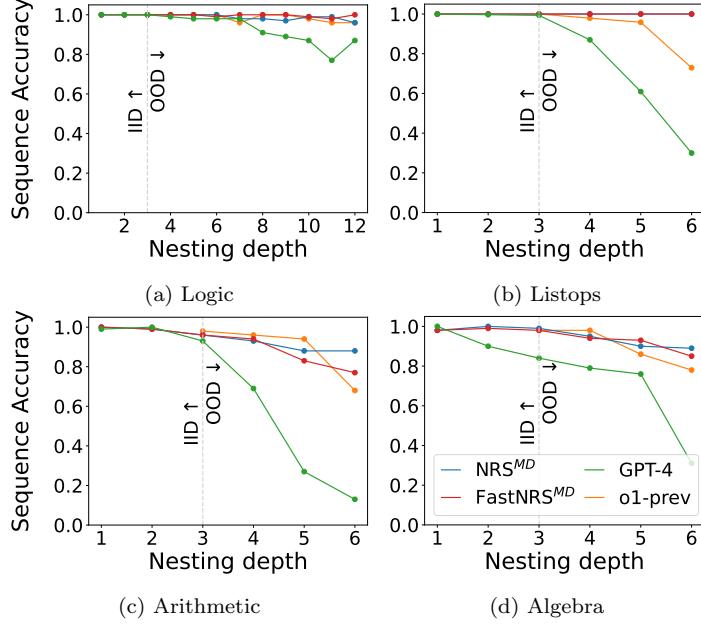


Figure 4: Performance of multi-domain models: GPT-4, o1-preview, FastNRS and NRS on each domain. Sequence accuracy is measured on data splits of 100 samples.

arithmetic formulas in Section 6.4, we will clearly see how the superior performance of the FastNRS depends on the greater robustness in the identification of leaf formulas, guaranteed by the text segmentation-based implementation of the Selector module.

### 6.2. Learning multi-domain convergent term rewriting systems

As detailed in Section 4, the architectures and execution dynamics of both the NRS and the FastNRS are specifically designed to support learning algorithms within the class of convergent term rewriting systems. This capability is primarily attributed to two key aspects: the algorithmic-inspired modular design of the architectures and the strong out-of-distribution generalization capability of the Selector module (see Section 6.5). As we mentioned, this generalization capability is particularly useful in a multi-domain scenario.

In this case, we choose to benchmark the models against OpenAI’s GPT-4 and o1-preview, whose training regimen is multi-domain by definition. The o1-preview model was benchmarked only on out-of-distribution data splits, as simpler formulas can be considered trivial for this type of model. Performance metrics for the four models are illustrated in Figure 4. GPT-4 is the

Problem	# Param.	Inf. time
Multi-domain	18,651,574	51h 48m 7s
Logic	3,047,365	16m 57s
ListOps	3,842,209	8h 24m 30s
Arithmetic	10,890,364	7h 59m 30s
Algebra	9,904,856	14h 16m 48s

Table 1: Space and time efficiency statistics for the NRS.

weakest performer across the tasks, particularly struggling with the ListOps, Arithmetic, and Algebra benchmarks. While it still surpasses the previously proposed Neural Deductive Reasoner (NDR), it shows significant limitations when faced with deeply nested formulas with complex operands. On the other hand, o1-preview demonstrates a substantial leap in performance over GPT-4. This improvement can be attributed to the more advanced CoT reasoning process implemented in the model, which enables it to process and simplify formulas more effectively, even within the complex tasks under consideration. These results could provide evidence of the importance of producing explicit reasoning steps for tasks that demand compositional reasoning.

In the multi-domain training scenario, both the Neural Rewriting Systems demonstrate their capacity to generalize to out-of-distribution samples. Specifically, the NRS has a slightly better performance than the FastNRS, especially on out-of-distribution samples, demonstrating the effectiveness of the specialized architectural elements introduced for this purpose. In the Logic and ListOps domains, the FastNRS maintains nearly the same accuracy as the NRS, with only a slight accuracy decrease of few percentage points in some cases. Notice that in the Logic domain, we evaluated the models on out-of-distribution test formulas with up to 12 nesting levels, where both models achieve consistently high accuracy. In these domains, the models show superior or similar performance to o1-preview on both in-distribution and out-of-distribution data splits. In the more complex Arithmetic and Algebra domains, there is a slightly larger drop in accuracy on certain out-of-distribution formulas. On these two tasks, the o1-preview model shows superior performance on some of the out-of-distribution splits. However, the NRS still outperforms o1-preview on the most complex formulas, demonstrating its effectiveness in learning convergent term rewriting systems with significant generalization capabilities.

Problem	# Param.	Inf. time
Multi-domain	15,061,616	3m 42s
Logic	2,501,795	38s
ListOps	4,095,633	32s
Arithmetic	8,728,338	50s
Algebra	8,752,920	3m 02s

Table 2: Space and time efficiency statistics for the FastNRS.

### 6.3. Analysis of efficiency

The design of the FastNRS mainly leads to performance improvements with respect to the twin implementation of the framework we propose. The number of parameters and inference time statistics for the NRS and the FastNRS in both training scenarios are reported in Table 1 and Table 2<sup>2</sup>. The FastNRS has, in most cases, fewer parameters than the NRS, and it achieves several orders of magnitude speed-up in inference time, both in the case of single- and multi-domain models (in the case of the multi-domain setting, cumulative inference time on all test samples in the four tasks is reported for both models). This efficiency gain demonstrates the possibility of implementing an efficient learning mechanism for the solution process of symbolic formulas across different domains within a single, unified neural circuitry. The modifications introduced in the FastNRS thus not only improve computational efficiency but also preserve the model’s generalization capabilities, and, as we have seen, can occasionally improve performance.

For comparison, we report analogous statistics for the baseline models in Appendix E. The LLMs considered in this work have a much larger number of parameters compared to our neural architectures (several order of magnitudes, according to independent estimates), and their inference times are much higher compared to our efficient FastNRS model. The Neural Data Router, instead, has a slightly lower complexity both in terms of number of parameters and inference time.

---

<sup>2</sup>All runs were executed on a single NVIDIA A100 GPU. Statistics reported for the NRS refer to the simplest hyperparameters configuration that achieves the best performance in each test scenario.

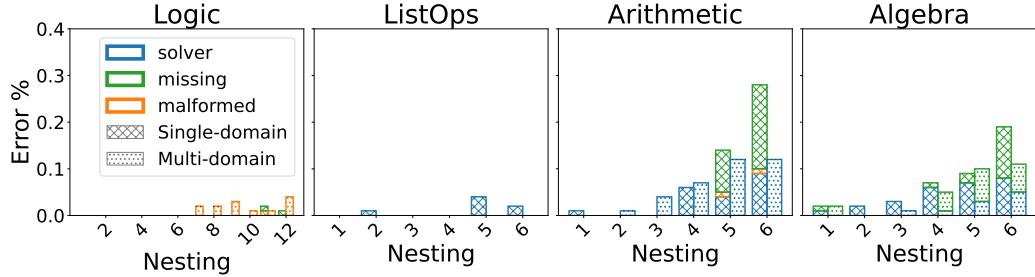


Figure 5: Breakdown of NRS errors by type in single- and multi-domain settings.

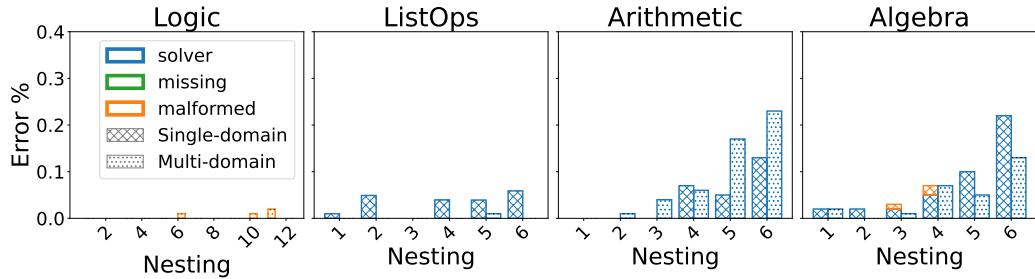


Figure 6: Breakdown of FastNRS errors by type in single- and multi-domain settings.

#### 6.4. Analysis of errors

We analyze the errors committed by the Neural Rewriting Systems when simplifying mathematical formulas in the four domains. We consider both the single-domain and multi-domain training settings. We breakdown errors for each domain by error type, and visualize the analysis in stacked barplots. We consider three error cases: the one in which leaf formulas identified by the Selector are not present in the input formula (**missing**), the case in which they are present but they are not valid formulas (**malformed**), and the case in which they are solved incorrectly (**solver**).

We start by observing that there are no errors in the **missing** class in the case of FastNRS, which is expected given that we use a segmentation-based approach to the Selector task. On the other hand, this type of error represents the majority of those committed by the Selector in the NRS, while errors in the **malformed** class are quite rare for both the NRS and the FastNRS. Therefore, we can conclude that when Selector modules in both architectures identify leaf formulas that are present in the input, these tend to be well-formed.

The multi-domain setting reveals interesting positive effects, but also in-

troduces challenges for both the NRS and FastNRS. In the case of NRS, multi-domain training seems to mostly have a neutral or positive effect across all tasks, significantly improving performance on algebraic and arithmetic formulas and reducing the amount of `missing` errors on the latter (see Figure 5). It seems that by training the Selector on multiple tasks, the NRS becomes more adept at identifying valid leaf formulas, reducing the number of this type of error.

The effects of multi-domain training on FastNRS are more heterogeneous and depend on the specific task. In the ListOps and algebra domains, the multi-domain model shows consistent improvements, with a marked reduction in Solver errors. Interestingly, FastNRS performance worsens in the multi-domain setting on arithmetic formulas, where Solver errors increase. This could reflect the fact that arithmetic formulas, involving operations between double-digit integers, are the hardest type of operation for the Solver.

Notably, in both single- and multi-domain scenarios, FastNRS never fails to find at least one valid leaf in any iteration, indicating robustness in the Selector module.

### 6.5. Out-of-distribution generalization in the FastNRS Selector

As we described in Section 4.2.1, the Selector module is a transformer encoder with two main architectural modifications: Label-based Positional Encodings and a strong limitation of the self-attention’s receptive field. Furthermore, we described how designing the Selector as a text segmentation module allowed us to simplify and improve the efficiency of the whole architecture. The plots in Figure 7 show how a Selector with the abovementioned architectural modifications, trained to segment input formulas, exhibits almost indistinguishable convergence trends on in- and out-of-distribution instances in all problems. Carefully tuned Selector modules can thus segment an input formula several tens of tokens longer than formulas observed during training, with very limited or zero error rate. Additional results about the impact of the width of the Selector’s self-attention window and the number of layers in the model are reported in Appendix B.1.1.

We also notice that the Selector is particularly sample efficient, as it requires only 5,000 iterations to converge to almost perfect accuracy, while the best NRS Selector modules, chosen after hyperparameters tuning, could be trained for up to 30,000 iterations, depending on the task (see Appendix B.2). We should also highlight that in all domains except ListOps, FastNRS

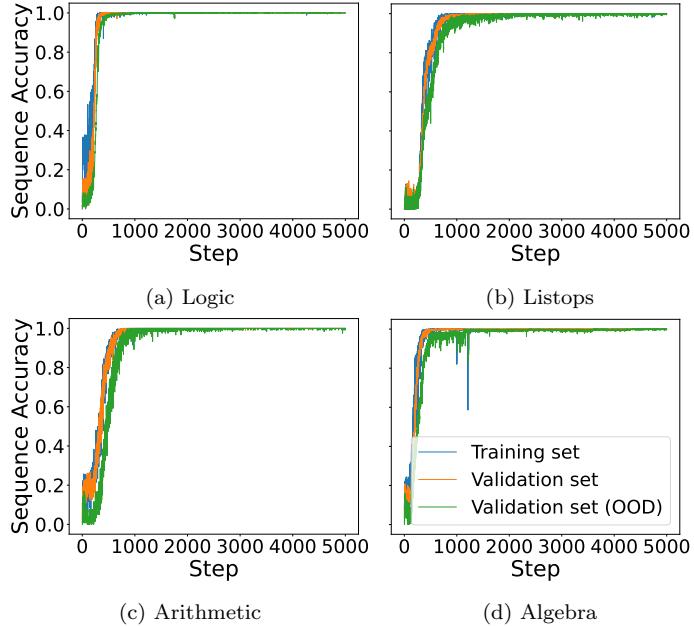


Figure 7: FastNRS Selector accuracy during training on the text segmentation task.

Selector modules are several thousands of parameters smaller than their NRS counterparts in the same domain.

## 7. Limitations

Despite the significant generalization capabilities demonstrated by the framework we propose, its scope and applicability are constrained by several limitations. First, the current implementation of NRS is restricted to tasks that can be framed as sequence-based rewriting problems. This assumption limits the range of tasks it can handle: for example, many real-world tasks involve hierarchical structures or visual reasoning, which cannot be addressed within the current sequence-only framework. Second, rewriting rules must currently operate on local substrings of the input sequence, even in the Fast-NRS which identifies multiple substrings in parallel. Finally, the NRS is built on an algorithmic structure where the steps of the rewriting process are predefined by the human designer. Although the system efficiently applies these predefined rules generalizing to more complex, unseen cases, it does not possess the capacity to learn or infer the rewriting algorithm from data.

Therefore, this design choice limits the possibility of applying the system to new problems beyond the algorithmic template initially provided.

## 8. Conclusions

In this work, we presented a general framework for learning convergent term rewriting systems using a neuro-symbolic architecture, inspired directly by the rewriting algorithm itself. Within this framework, we introduced two distinct implementations: the Neural Rewriting System (NRS) and the Fast Neural Rewriting System (FastNRS). Both architectures are designed to learn and generalize across the class of problems solvable with convergent term rewriting systems. The FastNRS, in particular, builds upon the NRS by incorporating key modifications that significantly improve memory efficiency, training time, and inference speed.

We evaluated both the NRS and FastNRS in both single-domain and multi-domain testing scenarios. In the multi-domain scenario, a single model is trained across multiple datasets or problem types simultaneously, resulting in a system that can solve various tasks within the same architecture. Using datasets such as Logic, ListOps, Arithmetic, and Algebra, we showed that both models consistently prove strong generalization capabilities across tasks, and that the FastNRS offers substantial reductions in computational costs.

We compared the Neural Rewriting Systems trained in a single-domain scenario with the Neural Data Router as a representative of small-scale neural architectures specialized to learn single reasoning tasks. Our systems clearly outperformed the baseline, especially on out-of-distribution samples.

We further compared both systems trained in a multi-domain scenario against two general-purpose Large Language Models: OpenAI’s GPT-4 and the recently presented o1-preview model designed to excel in complex reasoning tasks. Although the performance of our models on the most complex formulas consistently surpassed that of GPT-4 on the same problems, o1-preview showed surprising capabilities of solving even very complex formulas with a relatively high degree of accuracy. While o1-preview outperformed the Neural Rewriting Systems on some out-of-distribution formulas of intermediate complexity in some tasks, the models we propose consistently achieved equal or higher accuracy on the hardest formulas in all tasks, demonstrating significantly higher systematic generalization capabilities. The drop in performance of o1-preview on complex problems might suggest a fundamental lack of systematic reasoning capabilities and understanding of mathematical

concepts still persistent in this new class of models, as also noted in recent work by Mirzadeh et al. (2024).

The strengths of our architecture can be traced back to its modular design, which is informed by the rewrite algorithm, and to the architectural modifications to the transformer, which have proven effective in enabling strong out-of-distribution generalization. However, these design choices also limit the scope of applicability of our system to sequence-based problems solvable by convergent term rewriting systems with local substitution rules. Future work could be dedicated to expanding the system to handle rules that could act on patterns across different parts of the sequence. This would involve rethinking the Selector, where it would be necessary to design a neural circuit that is capable of generalizing on the selection of non-local patterns as consistently as the current circuit does with local ones. Furthermore, the substitution mechanism should be designed to be capable of reliably replacing the global patterns, potentially maintaining a level of flexibility to noise and errors in the Selector output. As we previously noticed, the algorithmic-informed design of our systems, which guarantees robustness, is defined *a priori* rather than being learned from data, and thus limits their scope of applicability. Designing an end-to-end learned system where the algorithmic blueprint of the problem at hand is inferred directly from data could prove to be a challenging and interesting venue for future research. While defining the algorithmic blueprint for a specific class of problems in advance imposes on the system a strong inductive bias that is aligned to the class itself, a general-purpose framework for algorithmic learning would involve designing a learning bias that allows the system to dynamically align to the specific class of problems under consideration.

## 9. Acknowledgements

The authors wish to thank OpenAI for granting free research access to the GPT-4 and o1-preview APIs. OpenAI had no involvement in the study design, collection, analysis and interpretation of data, writing of the report, or the decision to submit the article for publication.

## 10. Funding

This research did not receive any specific grant from funding agencies in the public, commercial, or not-for-profit sectors.

## Appendix A. Dataset statistics

As described in Section 5.2.1, the development sets for the Neural Rewriting System (NRS) across the four datasets—logic, listops, arithmetic, and algebra—were constructed to capture a diverse range of formula complexities. Each development set is composed of multiple subsets with varying nesting levels, alongside intermediate formulas generated during the resolution process of the main ones. By design, the number of unique formulas available in splits with simpler formulas is smaller than in splits with more complex ones, due to the combinatorial nature of the problem. Despite these differences in the number of unique formulas per split, during training, samples are drawn from each split with equal probability. This ensures balanced exposure across the splits, allowing the model to generalize across different formula complexities. Refer to Tables A.3 and A.4 for the exact number of samples in each development split for the four tasks. We report in Table A.5 the number of (unique) samples in the test sets of the four tasks used across all experiments.

Task	Training set	ID validation set	OOD validation set
Logic	238436	710	900
ListOps	840209	2332	900
Arithmetic	399218	180	60
Algebra	323787	900	300

Table A.3: No. of unique samples in the NRS and FastNRS development sets. ID and OOD indicate in-distribution and out-of-distribution sets, respectively.

Task	Training set	ID validation set
Logic	155	45
ListOps	22158	5541
Arithmetic	30510	7628
Algebra	18992	4749

Table A.4: Number of unique samples in the NRS and FastNRS Solver development sets. ID indicates the in-distribution set.

Task	Num. samples
Logic	1200
ListOps	600
Arithmetc	600
Algebra	600

Table A.5: Number of unique samples in the test sets

	Logic	ListOps	Arithm.	Algebra	MD
Embedding size	256	256	256	256	256
Num. Enc. Layers	3	4	4	6	4
Width self-attn window	1	1	1	1	1
Learning rate	3.55e-05	3.65e-05	2.66e-05	4.49e-05	1.69e-05

Table B.6: FastNRS Selector tuned hyperparameters values for each test scenario. MD indicates the multi-domain scenario.

## Appendix B. Training details

### Appendix B.1. Fast Neural Rewriting System

**Selector** For all problems, we adopted a problem-dependent tokenizer whose vocabulary contains atomic values, operators and parentheses. For example, the vocabulary for the arithmetic problem contains one token for each single- or double-digit integer, tokens for the sum, subtraction and multiplication operators and tokens for open and closed parentheses. In preliminary experiments, we also tried using a character-level tokenizer but observed worse out-of-distribution generalization capabilities of the Selector in some domains.

In all models, we used four attention heads and a hidden state in the feed-forward layers that was four times larger than the embedding size. We trained the models using the Adam optimizer with default parameters, a batch size of 512, a dropout probability of 10% and a cosine annealing schedule of the learning rate with 1000 linear warm-up iterations. We tuned the embedding size, the number of encoder layers, the width of the diagonal window applied to the self-attention matrix and the learning rate using a random search. For all tasks, we searched hyperparameters values in the following ranges: {128, 256, 512} for the embedding size, [1, 9] for the number of encoder layers and [1e-6, 6e-6] for the learning rate. All models were trained using the Adam optimizer for 5,000 iterations, apart from the multi-domain model

	Logic	ListOps	Arithm.	Algebra	MD
Embedding size	64	128	256	256	320
Num. Enc. Layers	1	2	3	2	4
Num. Dec. Layers	1	2	3	2	4
Dropout	0.18	0.18	0.1	0.33	0.13
Learning rate	9.23e-05	9.59e-05	9e-05	8e-05	6.19e-05
Warm-up it.	1282	1910	1500	1500	1714

Table B.7: FastNRS Solver tuned hyperparameters values for each test scenario. MD indicates the multi-domain scenario.

which was trained for 7,000 iterations. The final values chosen after tuning each hyperparameter are reported in Table B.6.

**Solver** We used a simple character-level tokenizer for all problems. We tuned the hyperparameters of the Solver using a random search on the embedding size, the number of encoder and decoder layers, the dropout rate, and the learning rate. In all models, we used four attention heads and a hidden state in the feed-forward layers that was four times larger than the embedding size. We trained the models using the Adam optimizer with default parameters, a batch size of 512 and a cosine annealing schedule of the learning rate. The models were trained for 10,000 iterations in the case of Logic and ListOps tasks and for 40,000 and 100,000 iterations in the case of Algebra and Arithmetic tasks, respectively. For all tasks, we searched hyperparameters values in the following ranges: {64, 128, 256} for the embedding size, [1, 4] for the number of encoder and decoder layers, [0.1, 0.4] for the dropout probability, [1e-5, 1e-4] for the learning rate and [1000, 2000] for the number of warmup iterations. The final values chosen after tuning each hyperparameter are reported in Table B.7.

#### Appendix B.1.1. Selector Depth and Width of the Self-Attention Window

Figures B.8 and B.9 illustrate the impact of both the width of the self-attention diagonal window and the number of layers in the FastNRS Selector model on sequence accuracy, measured on an out-of-distribution set of samples during training. We report the mean and standard deviation of a group of three runs with different random seeds. As shown in the plots, there is a clear inverse relationship between the width of the self-attention window and model performance, consistent across all the domains we consider. The width of the self-attention window is expressed in terms of the hyperparam-

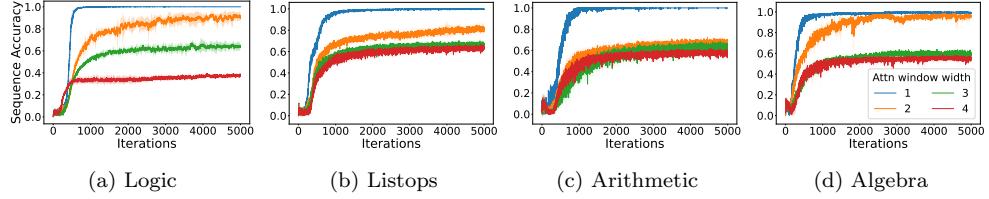


Figure B.8: Impact of self-attention window width on out-of-distribution sequence accuracy.

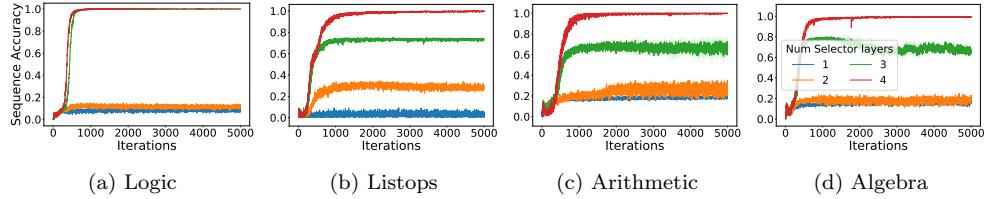


Figure B.9: Impact of number of layers on out-of-distribution sequence accuracy.

eter  $k$ . Specifically, as the window width increases, the model becomes progressively less capable of generalizing on out-of-distribution samples. Peak performance, indicated by the highest accuracy, is achieved when the hyperparameter  $k$  is set to 1, suggesting that a narrower focus in self-attention enhances the model’s ability to generalize to out-of-distribution data.

In contrast, there is a direct relationship between the number of layers and model performance: as the number of layers increases, the model’s generalization ability improves, leading to better accuracy on both in-distribution and out-of-distribution samples. We also notice that all the models with different window widths were equally able to fit the training set with no sign of overfitting on the in-distribution validation set. On the contrary, models with varying numbers of layers showed similar performance across the training, in-distribution, and out-of-distribution validation sets (not shown). Both analyses use model hyperparameters that were selected through hyperparameter tuning and are consistent with those applied in the rest of the experiments.

### Appendix B.2. Neural Rewriting System

**Selector** As done with the FastNRS, we employ a problem-dependent tokenizer in the NRS Selector. In all models we used four attention heads and a hidden state in the feed-forward layers that was four times larger than the embedding size. Selector models were trained for 20,000 iterations for

	Logic	ListOps	Arithm.	Algebra	MD
Embedding size	256	256	256	256	256
Width self-attn window	2	2	3	3	2
Num. Enc. Layers	1	1	3	4	5
Num. Dec. Layers	2	2	2	2	2
Dropout	0.29	0.37	0.17	0.20	0.10
Learning rate	2.7e-5	2.65e-5	2.35e-5	5.54e-5	7.86e-5
Warm-up it.	1600	1700	1900	2900	1500
MHA init. gain	0.97	0.71	1.69	0.75	1.00

Table B.8: NRS Selector tuned hyperparameters values for each test scenario. MD indicates the multi-domain scenario.

the Logic and ListOps tasks, and 30,000 iterations for the Arithmetic and Algebra tasks. We trained the models using the Adam optimizer with default parameters, a batch size of 512 (256 for Algebra) and a cosine annealing schedule of the learning rate with warm-up. We tuned the embedding size, the number of encoder and decoder layers, the width of the diagonal window applied to the self-attention matrix, the dropout rate, the learning rate, the number of warm-up iterations and the value of gain parameter for initialization of the self-attention layers using a random search. For all tasks, we searched hyperparameters values in the following ranges: {128, 256, 512} for the embedding size, [1, 3] for the width of the diagonal self-attention window, [2, 5] for the number of encoder and decoder layers, [0.1, 0.4] for the dropout probability, [1e-5, 6e-5] for the learning rate, [500, 3000] for the number of warm-up iterations, [0.5, 2.5] for the initialization gain parameter. The final values chosen after tuning each hyperparameter are reported in Table B.8.

**Solver** In our experiments we used the same Solver modules both in the FastNRS and in the NRS, thus the tokenization method and hyperparameters for the NRS Solver correspond to those detailed for the FastNRS Solver in Section Appendix B.1 and Table B.7.

### Appendix B.3. Neural Data Router

As done with the Neural Rewriting Systems, we employ a problem-dependent tokenizer at the atomic value level when training the Neural Data Router. Therefore, the size of the result window  $k$  equals 3 in the case of Algebra, 2 for Arithmetic and 1 for ListOps and Logic problems.

	Logic	ListOps	Arithmetic	Algebra
Num. Enc. Layers	19	5	11	17
Embedding size	256	512	512	512
Attention heads	8	16	8	16
FF size	1024	1024	2048	2048
Learning rate	2.33e-04	4.13e-04	7.64e-04	9.59e-04
Dropout	0.45	0.09	0.05	0.40
Attention dropout	0.38	0.49	0.06	0.18
Weight decay	0.02	0.09	0.08	0.03

Table B.9: NDR tuned hyperparameters values for each task.

Here we report the hyperparameters of the best Neural Data Router configuration we selected for each problem. We searched hyperparameters values in the same ranges used in the original paper. Models were trained using the AdamW optimizer for 5,000 iterations in the case of Logic, 30,000 iterations in the case of ListOps, and 100,000 iterations in the case of Arithmetic and Algebra. We used a batch size of 512 for all task except Algebra, for which it was 256. The final values chosen after tuning are reported in Table B.9.

### Appendix C. NRS Selector confidence scores

As described in Section 4.1, the Dynamic Windowing mechanism in the NRS Selector is regulated by a threshold  $T$ , which is used to determine on which formulas the mechanism should be applied. We select these thresholds by examining the average Selector confidence score for inputs of the same length, and choose the value corresponding to a decrease in average Selector confidence. We measure the average Selector confidence on several formulas of different lengths and nesting levels drawn from both the in-distribution and out-of-distribution validation sets. The distribution of these values is represented in Figures C.10 and C.11.

### Appendix D. Distribution of FastNRS Solver confidence scores

The conditional replacement of leaf formulas in the FastNRS operated by the `cond_repl` function is regulated by a threshold on Solver confidence. For both the single- and multi-domain versions of the FastNRS, we determined

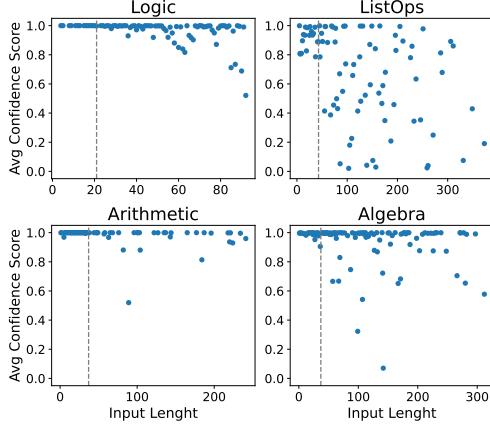


Figure C.10: Average single-domain NRS Selector confidence scores by input length. The vertical line represents the maximum length of training formulas.

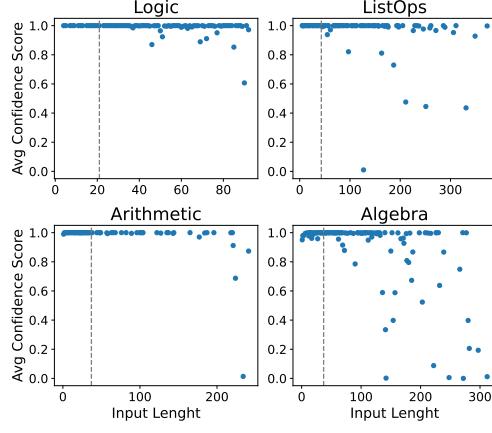


Figure C.11: Average multi-domain NRS Selector confidence scores by input length. The vertical line represents the maximum length of training formulas.

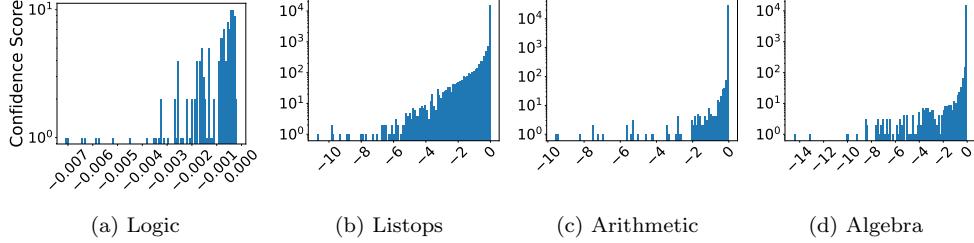


Figure D.12: Distribution of Solver confidence scores on training samples (y-axis in log scale).

these thresholds based on the distribution of these scores on training samples, as mentioned in Section 4.2. Specifically, the thresholds used were -6 for ListOps, -2 for Arithmetic, -3 for Algebra, and -0.005 for Logic. The distributions of Solver confidence scores on training samples, which informed these thresholds, are represented in Figure D.12. The plots can provide insight into how frequently high-confidence predictions occur.

## Appendix E. Baseline models statistics

In table E.10 we report the number of parameters and inference time statistics for the three baselines we consider in this study. For OpenAI o1-preview, statistics were computed only on out-of-distribution data splits, on

<b>Architecture</b>	<b>Problem</b>	<b># Param.</b>	<b>Inf. time</b>
o1-preview	Logic	~ 1.76 trillion	1h54m
	ListOps	~ 1.76 trillion	1h47m
	Arithmetics	~ 1.76 trillion	2h01m
	Algebra	~ 1.76 trillion	2h04m
gpt-4	Logic	~ 1.76 trillion	3h41m
	ListOps	~ 1.76 trillion	2h13m
	Arithmetics	~ 1.76 trillion	2h00m
	Algebra	~ 1.76 trillion	3h06m
NDR	Logic	1,007,667	24s
	ListOps	2,921,521	13s
	Arithmetics	4,055,676	25s
	Algebra	5,197,540	21s

Table E.10: Space and time efficiency statistics for the baseline models.<sup>3</sup>

which the model was tested (namely, formulas with three or more nesting levels). For the NDR, the runs were executed on a single NVIDIA A100 GPU, as done with our models.

## References

- Agarwal, V., Aditya, S., Goyal, N., 2021. Analyzing the nuances of transformers' polynomial simplification abilities. CoRR abs/2104.14095. URL: <https://arxiv.org/abs/2104.14095>, arXiv:2104.14095.
- Anil, C., Wu, Y., Andreassen, A., Lewkowycz, A., Misra, V., Ramasesh, V.V., Slone, A., Gur-Ari, G., Dyer, E., Neyshabur, B., 2022. Exploring length generalization in large language models, in: Koyejo, S., Mohamed, S., Agarwal, A., Belgrave, D., Cho, K., Oh, A. (Eds.), Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022.

---

<sup>3</sup>The number of parameters of GPT-4 is an independent estimate based on inference speed Schreiner (2023). The number of parameters of o1-preview is a ballpark estimate based on the assumption that the model is a fine-tuned version of GPT-4.

- Baader, F., Nipkow, T., 1998. Term rewriting and all that. Cambridge University Press.
- Brown, T.B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., 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.M., 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., Amodei, D., 2020. Language models are few-shot learners, in: Larochelle, H., Ranzato, M., Hadsell, R., Balcan, M., Lin, H. (Eds.), Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual.
- Cai, C.H., et al., 2018. Learning of human-like algebraic reasoning using deep feedforward neural networks. *Biol. Inspired Cogn. Arch.* 25, 43–50.
- Caruana, R., 1997. Multitask learning. *Machine learning* 28, 41–75.
- Chang, H., Zhang, H., Barber, J., Maschinot, A., Lezama, J., Jiang, L., Yang, M.H., Murphy, K., Freeman, W.T., Rubinstein, M., Li, Y., Krishnan, D., 2023. Muse: Text-to-image generation via masked generative transformers. URL: <https://arxiv.org/abs/2301.00704>, arXiv:2301.00704.
- 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, X., Tian, Y., 2019. Learning to perform local rewriting for combinatorial optimization, in: Wallach, H.M., et al. (Eds.), Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada, pp. 6278–6289.
- Cognolato, S., Testolin, A., 2022. Transformers discover an elementary calculation system exploiting local attention and grid-like problem representation, in: 2022 International Joint Conference on Neural Networks (IJCNN), IEEE. pp. 1–8.
- Cormen, T.H., Leiserson, C.E., 2022. Introduction to Algorithms, fourth edition. MIT Press, London, England.

- Csordás, R., Irie, K., Schmidhuber, J., 2021. The devil is in the detail: Simple tricks improve systematic generalization of transformers, in: Moens, M., Huang, X., Specia, L., Yih, S.W. (Eds.), Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 7-11 November, 2021, Association for Computational Linguistics. pp. 619–634. doi:10.18653/V1/2021.EMNLP-MAIN.49.
- Csordás, R., Irie, K., Schmidhuber, J., 2022. The neural data router: Adaptive control flow in transformers improves systematic generalization, in: The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022, OpenReview.net. URL: [https://openreview.net/forum?id=KBQP4A\\_J1K](https://openreview.net/forum?id=KBQP4A_J1K).
- Davis, E., 2024. Mathematics, word problems, common sense, and artificial intelligence. *Bulletin of the American Mathematical Society* 61, 287–303.
- Graves, A., Wayne, G., Danihelka, I., 2014. Neural turing machines. CoRR abs/1410.5401. URL: <http://arxiv.org/abs/1410.5401>, arXiv:1410.5401.
- Graves, A., Wayne, G., Reynolds, M., Harley, T., Danihelka, I., Grabska-Barwinska, A., Colmenarejo, S.G., Grefenstette, E., Ramalho, T., Agapiou, J.P., Badia, A.P., Hermann, K.M., Zwols, Y., Ostrovski, G., Cain, A., King, H., Summerfield, C., Blunsom, P., Kavukcuoglu, K., Hassabis, D., 2016. Hybrid computing using a neural network with dynamic external memory. *Nat.* 538, 471–476. URL: <https://doi.org/10.1038/nature20101>, doi:10.1038/NATURE20101.
- He, K., Zhang, X., Ren, S., Sun, J., 2016. Deep residual learning for image recognition, in: 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp. 770–778. doi:10.1109/CVPR.2016.90.
- Hendrycks, D., Basart, S., Mu, N., Kadavath, S., Wang, F., Dorundo, E., Desai, R., Zhu, T., Parajuli, S., Guo, M., Song, D., Steinhardt, J., Gilmer, J., 2021. The many faces of robustness: A critical analysis of out-of-distribution generalization, in: 2021 IEEE/CVF International Conference on Computer Vision, ICCV 2021, Montreal, QC, Canada, October 10-17, 2021, IEEE. pp. 8320–8329. URL: <https://doi.org/10.1109/ICCV48922.2021.00823>, doi:10.1109/ICCV48922.2021.00823.

- Hinton, G.E., 1990. Connectionist symbol processing - preface. *Artif. Intell.* 46, 1–4. URL: [https://doi.org/10.1016/0004-3702\(90\)90002-H](https://doi.org/10.1016/0004-3702(90)90002-H), doi:10.1016/0004-3702(90)90002-H.
- Hupkes, D., Dankers, V., Mul, M., Bruni, E., 2020. Compositionality decomposed: How do neural networks generalise? *J. Artif. Intell. Res.* 67, 757–795. URL: <https://doi.org/10.1613/jair.1.11674>, doi:10.1613/JAIR.1.11674.
- Kahneman, D., 2011. Thinking, Fast and Slow. Farrar, Straus and Giroux.
- Kautz, H.A., 2022. The third AI summer: AAAI robert s. engelmore memorial lecture. *AI Mag.* 43, 105–125.
- Kazemnejad, A., Padhi, I., Ramamurthy, K.N., Das, P., Reddy, S., 2023. The impact of positional encoding on length generalization in transformers, in: Oh, A., Naumann, T., Globerson, A., Saenko, K., Hardt, M., Levine, S. (Eds.), *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023*, New Orleans, LA, USA, December 10 - 16, 2023.
- Kojima, T., Gu, S.S., Reid, M., Matsuo, Y., Iwasawa, Y., 2022. Large language models are zero-shot reasoners, in: Koyejo, S., Mohamed, S., Agarwal, A., Belgrave, D., Cho, K., Oh, A. (Eds.), *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022*, New Orleans, LA, USA, November 28 - December 9, 2022.
- Komendantskaya., E., 2009. Parallel rewriting in neural networks, in: *Proceedings of the International Joint Conference on Computational Intelligence (IJCCI 2009) - ICNC, INSTICC*. SciTePress. pp. 452–458. doi:10.5220/0002319704520458.
- Lake, B.M., Baroni, M., 2018. Generalization without systematicity: On the compositional skills of sequence-to-sequence recurrent networks, in: Dy, J.G., Krause, A. (Eds.), *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018, PMLR*. pp. 2879–2888. URL: <http://proceedings.mlr.press/v80/lake18a.html>.

- Lample, G., Charton, F., 2019. Deep learning for symbolic mathematics. ArXiv abs/1912.01412.
- Li, Y., McClelland, J.L., 2022. Systematic generalization and emergent structures in transformers trained on structured tasks, in: All Things Attention: Bridging Different Perspectives on Attention, Annual Conference on Neural Information Processing Systems.
- Marghetis, T., Landy, D., Goldstone, R.L., 2016. Mastering algebra retrains the visual system to perceive hierarchical structure in equations. Cognitive research: principles and implications 1, 1–10.
- Mirzadeh, S., Alizadeh, K., Shahrokhi, H., Tuzel, O., Bengio, S., Farajtabar, M., 2024. Gsm-symbolic: Understanding the limitations of mathematical reasoning in large language models. CoRR abs/2410.05229. URL: <https://doi.org/10.48550/arXiv.2410.05229>, doi:10.48550/ARXIV.2410.05229, arXiv:2410.05229.
- Nangia, N., Bowman, S.R., 2018. Listops: A diagnostic dataset for latent tree learning, in: Cordeiro, S.R., Oraby, S., Pavalanathan, U., Rim, K. (Eds.), Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics, NAACL-HLT 2018, New Orleans, Louisiana, USA, June 2-4, 2018, Student Research Workshop, Association for Computational Linguistics. pp. 92–99. doi:10.18653/V1/N18-4013.
- Newell, A., Simon, H., 1956. The logic theory machine—a complex information processing system. IRE Transactions on information theory 2, 61–79.
- OpenAI, 2023. GPT-4 technical report. arXiv:2303.08774.
- OpenAI, 2024. Learning to reason with LLMs. <https://openai.com/index/learning-to-reason-with-llms/>. Accessed: 2024-09-29.
- Petruzzellis, F., Testolin, A., Sperduti, A., 2024a. Assessing the emergent symbolic reasoning abilities of llama large language models. To appear in Proceedings of the 33rd International Conference on Artificial Neural Networks (ICANN24).
- Petruzzellis, F., Testolin, A., Sperduti, A., 2024b. Benchmarking GPT-4 on algorithmic problems: A systematic evaluation of prompting strategies.

Proceedings of the 2024 Joint International Conference on Computational Linguistics, Language Resources and Evaluation, LREC-COLING 2024, Turin (Italy), May, 20-25, 2024 .

Petruzzellis, F., Testolin, A., Sperduti, A., 2024c. A Neural Rewriting System to Solve Algorithmic Problems. To appear in Proceedings of the 27th European Conference on Artificial Intelligence.

Pinker, S., Prince, A., 1988. On language and connectionism: analysis of a parallel distributed processing model of language acquisition. *Cognition* 28, 73–193.

Ruiz, L., Ainslie, J., Ontañón, S., 2021. Iterative decoding for compositional generalization in transformers. CoRR abs/2110.04169. URL: <https://arxiv.org/abs/2110.04169>, arXiv:2110.04169.

Rumelhart, D.E., McClelland, J.L., AU, 1986. Parallel distributed processing. The MIT Press.

Ruoss, A., Delétang, G., Genewein, T., Grau-Moya, J., Csordás, R., Ben-nani, M., Legg, S., Veness, J., 2023. Randomized positional encodings boost length generalization of transformers, in: Rogers, A., Boyd-Graber, J.L., Okazaki, N. (Eds.), Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers), ACL 2023, Toronto, Canada, July 9-14, 2023, Association for Computational Linguistics. pp. 1889–1903. URL: <https://doi.org/10.18653/v1/2023.acl-short.161>, doi:10.18653/V1/2023.ACL-SHORT.161.

Saxton, D., Grefenstette, E., Hill, F., Kohli, P., 2019. Analysing mathematical reasoning abilities of neural models, in: 7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019, OpenReview.net. URL: <https://openreview.net/forum?id=H1gR5iR5FX>.

Schreiner, M., 2023. GPT-4 architecture, datasets, costs and more leaked. <https://web.archive.org/web/20230712123915/https://the-decoder.com/gpt-4-architecture-datasets-costs-and-more-leaked/>. Accessed: 2023-07-12.

- Setzler, M., Howland, S., Phillips, L.A., 2022. Recursive decoding: A situated cognition approach to compositional generation in grounded language understanding. CoRR abs/2201.11766. URL: <https://arxiv.org/abs/2201.11766>, arXiv:2201.11766.
- Testolin, A., 2024. Can neural networks do arithmetic? a survey on the elementary numerical skills of state-of-the-art deep learning models. *Applied Sciences* .
- Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, L., Polosukhin, I., 2017. Attention is all you need, in: Guyon, I., von Luxburg, U., Bengio, S., Wallach, H.M., Fergus, R., Vishwanathan, S.V.N., Garnett, R. (Eds.), *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017*, December 4-9, 2017, Long Beach, CA, USA, pp. 5998–6008.
- Velickovic, P., Blundell, C., 2021. Neural algorithmic reasoning. *Patterns* 2, 100273. URL: <https://doi.org/10.1016/j.patter.2021.100273>, doi:10.1016/J.PATTER.2021.100273.
- Velickovic, P., Ying, R., Padovano, M., Hadsell, R., Blundell, C., 2020. Neural execution of graph algorithms, in: *8th International Conference on Learning Representations, ICLR 2020*, Addis Ababa, Ethiopia, April 26-30, 2020, OpenReview.net. URL: <https://openreview.net/forum?id=SkgK00EtvS>.
- Vinyals, O., Fortunato, M., Jaitly, N., 2015. Pointer networks, in: Cortes, C., Lawrence, N.D., Lee, D.D., Sugiyama, M., Garnett, R. (Eds.), *Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015*, December 7-12, 2015, Montreal, Quebec, Canada, pp. 2692–2700.
- Wang, X., et al., 2023. Self-consistency improves chain of thought reasoning in language models, in: *The Eleventh International Conference on Learning Representations, ICLR 2023*, Kigali, Rwanda, May 1-5, 2023.
- Wei, J., Wang, X., Schuurmans, D., Bosma, M., Ichter, B., Xia, F., Chi, E.H., Le, Q.V., Zhou, D., 2022. Chain-of-thought prompting elicits reasoning in large language models, in: Koyejo, S., Mohamed, S., Agarwal, A., Belgrave, D., Cho, K., Oh, A. (Eds.), *Advances in Neural Information*

Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022.

Ye, H., Xie, C., Cai, T., Li, R., Li, Z., Wang, L., 2021. Towards a theoretical framework of out-of-distribution generalization, in: Ranzato, M., Beygelzimer, A., Dauphin, Y.N., Liang, P., Vaughan, J.W. (Eds.), Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual, pp. 23519–23531.

Zhou, H., Bradley, A., Littwin, E., Razin, N., Saremi, O., Susskind, J.M., Bengio, S., Nakkiran, P., 2024. What algorithms can transformers learn? A study in length generalization, in: The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024, OpenReview.net. URL: <https://openreview.net/forum?id=AssIuHnmHX>.