Can Test-time Computation Mitigate Reproduction Bias in Neural Symbolic Regression?

Anonymous authors
Paper under double-blind review

Abstract

Mathematical expressions play a central role in scientific discovery. Symbolic regression aims to automatically discover such expressions from given numerical data. Recently, Neural symbolic regression (NSR) methods that involve Transformers pre-trained on synthetic datasets have gained attention for their fast inference, but they often perform poorly, especially with many input variables. In this study, we analyze NSR from both theoretical and empirical perspectives and show that (1) ordinary token-by-token generation is ill-suited for NSR, as Transformers cannot compositionally generate tokens while validating numerical consistency, and (2) the search space of NSR methods is greatly restricted due to reproduction bias, where the majority of generated expressions are merely copied from the training data. We further examine whether tailored test-time strategies can reduce reproduction bias and show that providing additional information at test time effectively mitigates it. These findings contribute to a deeper understanding of the limitation of NSR approaches and provide guidance for designing more robust and generalizable methods.

1 Introduction

Discovering underlying equations from collected experimental data is a crucial process in many fields of scientific research. Symbolic regression is a branch of regression analysis that seeks to automatically identify the underlying mathematical expressions. In contrast to methods that model data without explicit mathematical expressions, symbolic regression offers advantages in terms of interpretability and generalizability. This is because the outputs of symbolic regression are usually compact, human-readable equations, making them less susceptible to overfitting. However, symbolic regression is a challenging task due to its vast search space; the number of possible mathematical expressions grows exponentially with expression length or the number of input variables. Applications for symbolic regression span various fields of scientific research such as physics (Tenachi et al., 2023), materials science (Wang et al., 2019), and weather forecasting (Abdellaoui & Mehrkanoon, 2021).

Various methods for symbolic regression have been proposed in recent years. Traditionally, approaches based on genetic programming (GP) (Koza, 1994) have been used to solve symbolic regression. These methods tend to be computationally expensive because they generate each expression entirely from scratch. To mitigate this inefficiency, a research direction called neural symbolic regression (NSR) has emerged. NSR methods leverage encoder-decoder Transformer architectures Vaswani et al. (2017) pre-trained on large-scale synthetic datasets Biggio et al. (2021); Valipour et al. (2021). NSR methods generate expressions similar to natural language processing tasks, where expressions are generated token-by-token in an auto-regressive manner. Since a single forward pass through the Transformer suffices to output a mathematical token (e.g., $x_1, \sin, +$), NSR models can generate solutions far more quickly than GP-based approaches. However, NSR methods often fall short in terms of numerical accuracy, with particularly poor performance when the number of input variables is large (Kamienny et al., 2022; Bendinelli et al., 2023). This study aims to uncover the underlying cause of this drawback and explore methods to alleviate it.

Our analysis begins by examining the mechanisms a Transformer relies on to select the next token during expression generation. Specifically, we theoretically analyze the limitations Transformers face when generating

mathematical expressions. A natural and ideal token-generation strategy would be to select, in a compositional manner at each step, the token that is most appropriate in light of the numerical data. However, by using circuit complexity theory, we show that Transformers fail to generate expressions in such ways; they cannot compositionally generate mathematical expressions while taking numerical data into account. In other words, while Transformers generate tokens sequentially, they are not able to carry out meaningful processing at every step of token generation, suggesting that ordinary token-by-token generation is not particularly suitable for NSR. To illustrate this limitation, consider a situation where a Transformer has generated an expression up to $x_1^2 + \sin(x_2)+$. Our analysis implies that Transformers are unable to internally compute which leaf token (e.g., $x_1, x_2, x_3, \ldots$) would be the most appropriate given the input numerical data. The result indicates that in practice, Transformers generate expressions by some alternative, more coarse-grained mechanism instead of generating tokens in a compositional manner.

We next investigated how NSR methods actually generate expressions under empirical conditions. We hypothesize that, in NSR methods, naively using a Transformer for inference leads to **reproduction bias**, meaning that models struggle to generate novel expressions not seen during training and instead tend to generate expressions copied from the training data. Given that the expressions in the training data typically represent only a small subset of the full space of possible expressions, our hypothesis implies that standard NSR methods operate within a significantly constrained search space. We investigated this hypothesis in NSR methods such as NeSymReS (Biggio et al., 2021), a pioneering work in NSR models. We found that the majority of expressions generated by Transformers are expressions that were included in the training dataset, which supports our hypothesis of reproduction bias. Prior work has highlighted NSR methods' limited generalizability with respect to the range of numerical data—e.g., models trained on data whose input variable x lies in the interval $[-1, 1]$ often fail when evaluated on inputs from the wider interval $[-2, 2]$ (Li et al., 2024; Shojaee et al., 2023). However, the reproduction bias that we identify is orthogonal to this phenomenon and represents an even more fundamental limitation: NSR models often fail to generalize even within their training domain. This work is the first to show that standard NSR models primarily copy training expressions instead of composing familiar components into genuinely novel formulas.

Towards the end of this paper, we explore methodologies to mitigate the reproduction bias of standard NSR models and improve numerical accuracy. Since the theoretical analysis suggested that normal auto-regressive generation may be inappropriate for NSR, we focus particularly on test-time strategies and investigate how they affect reproduction bias and numerical accuracy. We compare three strategies: decoding with a large beam size, decoding using MCTS, and providing verification feedback at the subtree level. The last strategy is a new method that we propose, which we refer to as neural symbolic regression guided by verified subtrees (NSR-gvs). We found that providing new information to the model during test-time leads to generating expressions beyond the training dataset. However, we also identify cases where reproduction bias was mitigated but numerical accuracy decreased, as well as cases where numerical performance improved despite little alleviation in reproduction bias.

The contributions of our work are summarized as follows:

- We formally show that Transformers lack the ability to compositionally generate expressions while accounting for numerical data.
- We empirically demonstrate, under various settings, that naively applying a Transformer to symbolic regression leads to reproduction bias.
- We compare varying test-time computing strategies and analyze how such strategies affect reproduction bias and numerical accuracy.

2 Related Work

Several approaches to symbolic regression exist, such as GP, brute-force algorithms, reinforcement learning, and NSR. Since our study focuses on analyzing and improving NSR methods, we mainly describe NSR in detail in this section, and explain other symbolic regression methods in Appendix E.

Table 1: Comparison between our work and other major NSR studies

NSR Methods	Automatic Data Generation	Direct Constant Prediction	Information Added During Test-time	Assessing Reproduction Bias
Biggio et al. (2021)	✓	-	-	-
Kamienny et al. (2022)	✓	✓	-	-
Shojaee et al. (2023)	✓	✓	MCTS Feedback	-
Li et al. (2024)	-	-	Historical Context	-
Bendinelli et al. (2023)	✓	-	Prior Knowledge	-
Ours (NSR-gvs)	✓	-	Verified Subtrees	✓

Traditional symbolic regression methods such as GP generate each equation from scratch, resulting in long inference times, with equation generation potentially taking hours. In order to achieve a shorter inference time, studies such as NeSymReS (Biggio et al., 2021) and SymbolicGPT (Valipour et al., 2021) carried out large scale pre-training of Transformers. In these studies, an artificial dataset consisting of millions of randomly generated equations was used for training. These methods, often categorized as NSR, can generate an expression in just a few seconds, significantly reducing inference time compared with other approaches.

In recent years, several studies, summarized in Table 1, have focused on improving NSR methods. Studies such as (Kamienny et al., 2022) and (Vastl et al., 2024) proposed an end-to-end approach using a Transformer model to directly predict full mathematical expressions including constants, whereas previous methods followed a two-step procedure where constant fitting had to be done separately. Lalande et al. (2023) analyzed several different architectures to find a suitable encoder architecture for NSR. Shojaee et al. (2023) focused on improving the decoding strategy for NSR, incorporating the MCTS algorithm during the generation of expressions. In their study, Li et al. (2024) trained a Transformer model to imitate the process of improving mathematical formulas, as performed in the reinforcement learning-based approach proposed by Mundhenk et al. (2021). Bendinelli et al. (2023) proposed a model called NSRwH that enables incorporating prior knowledge, which is often available during application in scientific research. For example, scientists may anticipate symmetries between variables or expect certain partial expressions to appear in the mathematical laws governing the data. NSRwH adds a dedicated encoder that processes such prior knowledge, allowing the model to generate expressions that are consistent with both the numerical data and the prior knowledge provided.

More recently, there has been growing research on methods that iteratively refine mathematical expressions, most of which rely on large language models (LLMs) rather than pre-trained Transformer models. These methods are similar to NSR-gvs, one of the test-time computation approaches considered in this paper, in that they iteratively improve their outputs by incorporating feedback from the generated expressions. In (Merler et al., 2024) and (Sharlin & Josephson, 2024), the authors introduce approaches in which a base equation structure is generated using LLMs, and the equation is subsequently improved iteratively by receiving feedback from external numerical solvers. While Shojaee et al. (2024) follow a similar methodology, it incorporates supplementary descriptions regarding the variables in the prompt, facilitating more effective use of the LLM's scientific knowledge. Grayeli et al. (2024) proposed a method that uses an LLM to identify "concepts" representing features of high-performing expressions and leverages them to further evolve a set of equations. Zhang et al. (2025) introduces an iterative algorithm that replaces features of suboptimal expressions at each step while incorporating relevant expressions as needed. Pre-training-free methods described above may have the potential to address some of the limitations of conventional NSR approaches. On the other hand, using a pre-trained Transformer, as opposed to an LLM, offers certain advantages similar to those of small language models (SLMs), such as keeping the model size manageable and enhancing domain specificity through careful design of the training data. For these reasons, we believe that conducting a deeper analysis of pre-trained Transformer-based NSR and exploring ways to improve it remains a valuable research direction.

3 Problem Formulation

We formalize NSR as the problem of learning a *parameterized symbolic regressor* $S_{\boldsymbol{\theta}}$ that maps a numerical dataset $\mathcal{D}$ to a symbolic expression $\hat{e} = S_{\boldsymbol{\theta}}(\mathcal{D})$. The learning algorithm is formulated as minimizing a loss that measures how well $\hat{e}$ matches the ground-truth expression e^* underlying $\mathcal{D}$. In this section, we specify how synthetic training pairs $(e^*, \mathcal{D})$ are generated in NeSymReS (Biggio et al., 2021), since it is the foundational work underlying our research.

3.1 Synthetic Expression Distribution

We first sample a random binary–unary tree whose internal nodes are operators and whose leaves are variables.

Let $\mathcal{V} = \{x_1, \ldots, x_d\}$ be a finite set of variables, $\mathcal{O}_{\text{bin}}$ the binary operators (e.g., $\{+, -, \times, \div\}$), and $\mathcal{O}_{\text{un}}$ the unary operators (e.g., $\{\sin, \cos, \log, \exp\}$). Denote by $\mathcal{C} = [c_{\min}, c_{\max}] \subset \mathbb{R}$ the interval from which numeric constants are drawn. The complete vocabulary for the expression is $\Sigma = \mathcal{V} \cup \mathcal{O}_{\text{bin}} \cup \mathcal{O}_{\text{un}} \cup \mathcal{C}$.

Let $\mathcal{E}$ be an expression space and $p_{\mathcal{E}}$ be the generator of symbolic expressions employed in NeSymReS (Biggio et al., 2021). We also denote by p_{Tree} the generator of unary-binary trees introduced by Lample & Charton (2019). We write $e^* \sim p_{\mathcal{E}}$ for the following procedure.

1. Draw a random binary-unary tree $T \sim p_{\text{Tree}}$.
2. Assign internal nodes independently and uniformly from $\mathcal{O}_{\text{bin}} \cup \mathcal{O}_{\text{un}}$, and leaves uniformly from the variable set $\mathcal{V}$, resulting in a *template* expression e_{templ}.
3. For each unary operator u, sample a constant c_{mul} from distribution $\mathcal{D}_{\text{mul}}$ and replace u with $c_{\text{mul}}u$; otherwise keep the unary operator as is.
4. For each variable x, sample a constant c_{mul} from distribution $\mathcal{D}_{\text{mul}}$ and a constant c_{add} from distribution $\mathcal{D}_{\text{add}}$ and replace x with $c_{\text{mul}}x + c_{\text{add}}$; otherwise keep the variable as is.
5. The resulting expression is the final $e^* \in \mathcal{E}$.

3.2 Synthetic Dataset Generation

Given an expression e^*, we construct the dataset

$$\mathcal{D} \;=\; \{\,(\mathbf{x}_i,\, y_i)\,\}_{i=1}^{n}, \quad x_{ij} \;\sim\; \mathcal{U}([x_{\min,j}, x_{\max,j}])$$
$$\text{for} \quad j = 1, 2, \ldots, d, \quad y_i \;=\; e^*(\mathbf{x}_i).$$

Where $\{[x_{min,j}, x_{max,j}]\}_{j=1}^{d}$ denotes the intervals for each independent variable. The joint distribution of training pairs is therefore $(e^*, \mathcal{D}) \sim p_{\mathcal{E}} \times \mathcal{G}$, where $\mathcal{G}$ denotes the above stochastic data generation process.

We now denote by $\Gamma = \mathcal{V} \cup \mathcal{O}_{\text{bin}} \cup \mathcal{O}_{\text{un}} \cup \{C, \textsc{End}\}$ the vocabulary for token sequences, where C is a placeholder token to represent constants, and $\textsc{End}$ denotes the explicit end-of-sequence marker. The vocabulary Γ is slightly different from Σ since continuous numeric constants cannot be represented with a finite number of tokens.

Let $\text{seq} : \mathcal{E} \longrightarrow \Gamma^*$ be a *serialization map* that converts any symbolic expression into its unique prefix token representation. For a ground–truth expression $e^* \in \mathcal{E}$ we set

$$\mathbf{s}^* \;=\; \text{seq}\big(e^*\big) \;=\; (s_1^*, \ldots, s_L^*), \qquad L := |\mathbf{s}^*|.$$

The predictive distribution $q_{\boldsymbol{\theta}}(\,\cdot \mid \mathbf{s}_{<j}, \mathcal{D})$ is realized by an *encoder–decoder Transformer* parametrized by $\boldsymbol{\theta}$. Conditioned on the dataset $\mathcal{D}$ (encoded by the encoder) and the previously emitted prefix $\mathbf{s}_{<j}$, the decoder outputs a probability over the next token $s_j \in \Gamma$.

The *token-level loss* for a single training pair $(e^*, \mathcal{D})$ is then

$$L_{\text{tok}}(e^*; \boldsymbol{\theta}) = -\sum_{j=1}^{L} \log q_{\boldsymbol{\theta}}(s_j^* \mid \mathbf{s}_{<j}^*, \mathcal{D}). \tag{1}$$

Note that, in practice, the training dataset is the collection of e_{templ}, and both e^* and $\mathcal{D}$ are generated dynamically during training. Further details concerning the work of NeSymReS (Biggio et al., 2021) are described in Appendix A.

4 Theoretical Analysis on Expression Generation Ability of Transformers

Transformer-based symbolic regression tends to suffer from low performance, particularly when the number of input variables is large. In this section, we explore the theoretical basis of this limitation. Ideally, Transformers should be able to generate tokens in a compositional manner, while maximizing the probability for the final expression to fit the numerical data. However, we show that Transformers inherently lack the capacity to generate expressions compositionally while accounting for their numerical characteristics. We introduce a simplified version of the symbolic regression task and show that Transformers are not expressive enough to solve the task.

We define the *last-token prediction problem* as the task of predicting the most suitable final token in an otherwise complete mathematical expression. Although predicting the entire optimal expression is NP-hard Virgolin & Pissis (2022), this task is much easier since the search space is limited to several leaf tokens. We present a formal definition of this task in the following.

We first introduce $\text{expr} : \Gamma^* \times \mathbb{R}^{(d+1)*n} \longrightarrow \mathcal{E}$, a function that maps a token sequence $\mathbf{s}$ to the most appropriate expression $e_{\mathbf{s}}$ that can be represented by $\mathbf{s}$, taking into account the numerical data $\mathcal{D}$. Since the token sequence may contain the placeholder token C representing constants, the mapping is tasked with identifying the optimal values for these constants and transforming the sequence into a corresponding expression tree.

Definition 4.1 (Last-token prediction problem)**.** Given numerical data $\mathcal{D}$ of n features-value pairs $(\mathbf{x}_i, y_i) \in \mathbb{R}^d \times \mathbb{R}$, a metric $\mathcal{L} : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}$, and an incomplete token sequence $\widetilde{\mathbf{s}}$ that forms a prefix representation of an expression when terminated with a leaf token, the last-token prediction problem asks to find a leaf token u^* such that:

$$u^* = \underset{u \in \Gamma}{\operatorname{argmin}}\ \mathcal{L}(\mathbf{y}, e_{(\widetilde{\mathbf{s}}, u)}(\mathbf{x})),$$

where $e_{(\widetilde{\mathbf{s}}, u)} = \text{expr}((\widetilde{\mathbf{s}}, u), \mathcal{D})$ with $(\widetilde{\mathbf{s}}, u)$ representing the concatenation of sequence $\widetilde{\mathbf{s}}$ and token u. When the length of $\widetilde{\mathbf{s}} = m$, we denote this problem as $\text{LastTokenPrediction}(m)$.

For the analysis, we assume a bounded-depth log-precision Transformer as in (Feng et al., 2023; Merrill & Sabharwal, 2023b;a; Strobl, 2023), a realistic setting where the intermediate computation values of the Transformer are limited to $O(\log k)$ bit precision, with k denoting the maximal length of the input sequence. We now present the theoretical result stating that Transformers with bounded size cannot solve the last-token prediction problem.

Theorem 4.2. *Assume $\mathsf{TC}^0 \neq \mathsf{NC}^1$. For any integer D and any polynomial Q, there exists a problem size m such that no log-precision Transformer defined in Section G.1 with depth D and hidden dimension $d \leq Q(m)$ can solve* $\text{LastTokenPrediction}(m)$.

We show the above theorem by leveraging circuit complexity theory. Specifically, TC^0 and NC^1 are types of circuit complexity classes, and it is generally conjectured that $\mathsf{TC}^0 \subsetneq \mathsf{NC}^1$. Prior work (Merrill & Sabharwal, 2023b) shows that log-precision Transformers can be simulated with TC^0 circuits. We provide a proof for the above theorem by showing that the complexity of the last-token prediction problem is lower bounded by NC^1. Detailed specifications of the problem setting and proof of the theorem are provided in Appendix G.

Although the final token of a mathematical expression is arguably the easiest to predict among its components, the above theorem shows that even this seemingly simple task presents substantial difficulties for

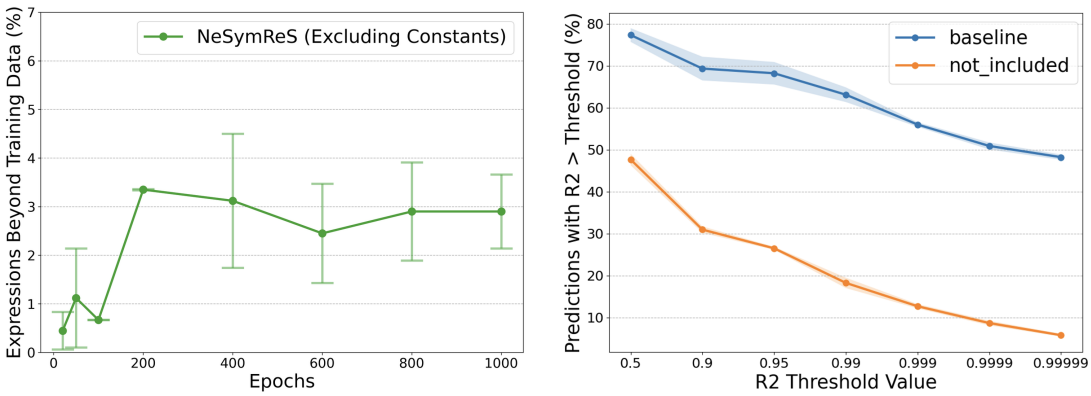

Figure 1: (Left) Percentage of expressions beyond the training dataset generated by NeSymReS on the `not_included` dataset. Throughout the training procedure, NeSymReS can hardly generate expressions that are not included in the training data, indicating strong reproduction bias. (Right) NeSymReS exhibits strong fitting performance on the `baseline` dataset but performs poorly on expressions from the `not_included` dataset, whose tree structures are absent from the training data. The result indicates the severe effect of reproduction bias on numerical accuracy.

Transformer models. In general, we find that when Transformers generate tokens via a single forward pass, they lack the capacity to account for numerical consistency, making it questionable whether auto-regressive generation is an appropriate paradigm for symbolic expression generation.

5 Exploring Reproduction Bias in NSR

When generating expressions in an auto-regressive manner, a seemingly appropriate strategy would be to compositionally produce the next token that maximizes accuracy, conditioned on both the previously generated partial expression and the numerical data. However, our theoretical analysis from the last section showed that Transformers lack the ability to do so, meaning that they rely on more coarse-grained strategies. This brings us to the following question: *How, in practice, does a transformer generate expressions during inference?* In this section, we empirically analyze how expressions are actually generated by NSR models. We demonstrate that NSR models primarily rely on reproduction—that is, they tend to generate expressions by directly copying those seen in the training data.

5.1 Reproduction Bias in Simplified Setting

We first tested how expressions are generated in NeSymReS, which is the method that we mainly focus on in this study. We examined whether the expressions generated by NeSymReS are merely copies from the training data or newly constructed formulas generated compositionally by the model.

Definition of reproduction bias. We first describe how the novelty of expressions are measured in this section. We introduce $\mathrm{seq}^{-1} : \Gamma^* \longrightarrow \mathcal{E}$, which is the inverse function of seq defined in Section 3.2, and $\mathrm{strip} : \mathcal{E} \longrightarrow \mathcal{E}$, which is a function that removes all constants from the input expression. We also define E_{templ} as the set of all e_{templ} contained in the training dataset.

Definition 5.1 (Reproduction bias)**.** Given an output token sequence $\mathbf{s}$, let $e = \mathrm{seq}^{-1}(\mathbf{s})$ be the original expression that is represented by $\mathbf{s}$. If $\mathrm{strip}(e) \in E_{\mathrm{templ}}$, we say $\mathbf{s}$ is a *reproduction* of expressions seen during training.

Remark 5.2. We have defined and measured reproduction bias based on whether the training dataset contains an expression that is structurally equivalent to the generated one. However, one may argue that we should define and measure reproduction based on functional equivalence; there are many expressions that

are structurally different but functionally equivalent (e.g., $x_1(x_1 + x_2)$ and $x_1^2 + x_1x_2$), and that such expressions should also be considered as equivalent expressions. To account for both structural and functional equivalence, we choose the operators as described below, so that any pair of structurally different expressions is also functionally different. For a more detailed discussion of the definition of reproduction bias, please refer to Appendix D.

Method. We constructed a simplified training dataset consisting of 100K equations. The allowed operator tokens were `add`, `sub`, `sin`, `cos`, `tan`, and `exp`, with up to 5 independent variables per equation. We then trained a NeSymReS model on this dataset for $1,000$ epochs. The variation of operators was limited due to two reasons. Firstly, the complicated training procedure of NeSymReS, where expressions with operators such as `mul` or `pow` are dynamically transformed and presented in different forms across epochs, makes it difficult to judge whether the model's generated expressions are novel or memorized from training. Therefore, such operators were excluded. Secondly, as explained in Remark 5.2, the choice of operators were restricted in order to account for both structural and functional equivalence when measuring reproduction bias. The dataset size was also kept relatively small due to computational cost and to balance the size of the training data against the size of the search space.

For evaluation, we constructed two test datasets: `not_included` and `baseline`, each containing 150 expressions. For the `not_included` set, we removed all e_{templ} appearing in the training data. In contrast, the `baseline` set was sampled directly from the generator $p_{\mathcal{E}}$ without any filtering. We associated each expression with 100 data points, generated in the same way as during training. We set the beam size to 5 for this experiment.

To evaluate fitting performance, we used the R^2 score, defined as follows. Given a test equation, a set of n data points $\{\mathbf{x}_i, y_i\}_{i=1}^n$, and the corresponding model predictions $\{\hat{y}_i\}_{i=1}^n$, the R^2 score is computed as:

$$R^2 = 1 - \frac{\sum_{i=1}^n (y_i - \hat{y}_i)^2}{\sum_{i=1}^n (y_i - \bar{y})^2} \quad \text{where} \quad \bar{y} = \frac{1}{n}\sum_{i=1}^n y_i.$$

Note that these n evaluation points are distinct from the inputs provided to the model at the test time. By definition, $R^2 \leq 1$, and values closer to 1 indicate that the predicted outputs closely match the true equation. In our experiments, we counted the number of predictions whose R^2 exceeds the thresholds of $0.5, 0.9, 0.95, 0.99, 0.999, 0.9999$, and 0.9999, respectively. This allows us to assess the model's ability to fit the data under both moderate and stringent accuracy requirements.

Result. Figure 1 shows the results for NeSymReS under the simplified setting. The left figure demonstrates NeSymReS's ability to generate novel expressions using the `not_included` dataset. As this dataset comprises instances of e_{templ} unseen in the training data, the model is expected to produce previously unseen tree structures. However, the result indicates that NeSymReS struggles to generate expression trees beyond the training data across varying epochs. After 1000 epochs of training, over 97% of the generated expression trees were direct copies from the training data, highlighting strong reproduction bias and revealing that the search space of NeSymReS is severely restricted. The right figure demonstrates how this reproduction bias negatively affects numerical accuracy, where NeSymReS's fitting performance on the `not_included` and `baseline` datasets is compared. The results indicate a substantial drop in performance for expressions whose tree structures are not present in the training data, compared with those sampled randomly. This suggests that for expressions not seen during training, the model's reproduction bias directly leads to poor numerical accuracy. This result also helps explain why NSR methods often fail to achieve high performance on expressions with many input variables; an increase in the number of input variables leads to an expanded search space, thereby increasing the likelihood that a given expression is absent from the training set.

5.2 Reproduction Bias in Practical Setting

Due to the complicated training procedure of NeSymReS, the above analysis was carried out in a simplified setting. To examine whether reproduction bias is a general phenomenon, we conducted an additional analysis in a more practical setting using transformer4sr (Lalande et al., 2023), a method similar to NeSymReS

but with a simpler training process. In transformer4sr, no dynamic transformations of expressions are applied during training, which makes it much easier than in NeSymReS to verify whether the generated expressions are included in the training data. We were also able to analyze the novelty of the expressions by explicitly taking into account the positions of the constant placeholder tokens. Consequently, the definition of reproduction bias slightly differs from that described in Section 5.1, and we therefore provide additional clarification in Appendix D.

We followed the model architecture, number of epochs, operator selection, and inference strategies described in Lalande et al. (2023). We constructed training datasets with varying size, with the largest one consisting of 1.5M expressions based on the practical setting used in the original paper. We used the full set of operator tokens, which are `add`, `mul`, `cos`, `log`, `exp`, `neg`, `inv`, `sqrt`, `sq` (squared), `cb` (cubed), and the number of independent variables was 6. We constructed a test set similar to `not_included` in the previous analysis, which consists of 300 expressions that were not included in the training data.

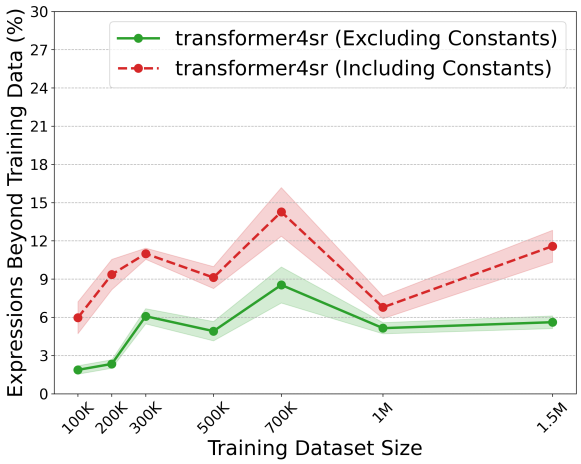

Figure 2: Reproduction bias in transformer4sr with varying training dataset sizes. While small training dataset sizes (100K, 200K) exhibit stronger reproduction bias, scaling the training dataset size does not necessarily mitigate reproduction bias after a certain limit.

Figure 2 shows the result for transformer4sr's ability to generate expressions beyond the training data. Even in a practical setting with 1.5M training expressions, less than 12% of the expressions generated by transformer4sr were novel expressions (taking into account the position of the constant placeholder tokens) beyond the training data, and less than 6% of the expressions had novel tree structures (excluding constants). The result also demonstrates that the effect of scaling training data on mitigating reproduction bias tends to saturate, suggesting that reproduction bias is likely to persist even when the training data size is further increased.

6 Can Test-time Strategies Mitigate Reproduction Bias?

The results from the previous section indicate that the search space of NeSymReS is mostly confined to expressions seen during training due to reproduction bias. Since our theoretical analysis indicates that naively performing next-token prediction makes it difficult to generate novel expressions in a compositional manner, we investigated the possibility of devising inference-time computational techniques to reduce reproduction bias in this section. Our hypothesis is that providing the model with hints about which tokens are appropriate could help steer the model to generate expressions that were not included in the training data. We begin by briefly introducing the three test-time strategies employed in our experiments. The detailed explanation for the strategies is presented in Appendix B.

6.1 Test-time Strategies

Decoding with large beam size. The beam search serves as the default decoding strategy employed by NeSymReS. During decoding, given a beam size of b, the decoding process generates b candidate sequences via beam search. The constant placeholders of each candidate are subsequently optimized using the Broyden–Fletcher–Goldfarb–Shanno (BFGS) algorithm (Fletcher, 2000). The expression exhibiting the highest numerical accuracy on the test data is then selected as the model's output. While the experiments in Section 5.1 used a beam size of $b = 5$, in this section we conducted experiments with a larger beam size of $b = 150$. Since increasing the beam size does not provide the model with any additional information, our hypothesis is that simply adopting a decoding strategy with a larger beam size will not alleviate reproduction bias.

Incorporating MCTS. TPSR (Shojaee et al., 2023) is a method that leverages MCTS during decoding time. In TPSR, the process starts by preparing a pre-trained NSR model (e.g., the NeSymReS model). Instead of relying on standard decoding methods like beam search, the method generates tokens using MCTS, where both the expansion and evaluation stages of MCTS leverage the pre-trained NSR model. In the expansion phase, to avoid unnecessary exploration, the set of expandable tokens is restricted to the top-$k_{\max}$ candidates based on the logits from the NSR model. During the evaluation phase, the NSR model first completes the remainder of the expression following the expanded token. The completed expression is then evaluated primarily based on its fitting accuracy, with additional consideration given to its complexity. In the experiments presented in this section, we used the default hyperparameter settings of TPSR as specified in the original paper; we set the number of rollouts to $r = 3$, the number of expandable tokens to $k_{\max} = 3$, and the beam size for expression completion to $b = 1$.

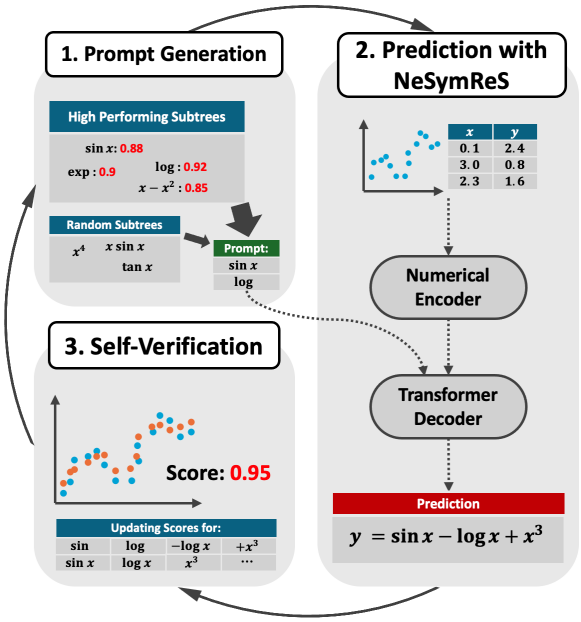

Figure 3: Overview of NSR-gvs's inference procedure. We first sampled subtrees from the candidate pool, then supplied them to the model together with numerical data. Then, the generated prediction is numerically verified and the self-verification feedback is used to update the candidate pool. This procedure is performed repeatedly to generate better predictions over time.

NSR-gvs. TPSR provides feedback to the model by assigning a reward to each token, reflecting the quality or appropriateness of that token. In contrast, we hypothesized that incorporating feedback at the subtree level as well may have a positive effect on the model. To this end, we propose NSR-gvs, a method grounded in the following intuition: expressions that fit the same numerical data well are likely to share common substructures.

We first trained a slightly modified version of the NeSymReS model, where the model takes subtrees as prompts and generates expressions that incorporate them. We achieved this by extracting subtrees from the ground-truth expressions and feeding them to the model together with numerical data during training. Figure 3 illustrates the inference procedure of NSR-gvs. We generated multiple predictions iteratively by augmenting the model with varying prompts. For each iteration, we first sampled subtrees from a pool of candidate subtrees, which were extracted from high-performing expressions in previous predictions. To maintain output diversity, we also occasionally sampled subtrees from a random distribution. We then provided the sampled subtrees to the pre-trained model as prompts, along with the numerical data. After the model generates a prediction, it is automatically verified according to the fitting accuracy on the test data. Finally, the pool of candidate subtrees are updated based on the results of self-verification. This method can be formulated within the framework of reinforcement learning, and we provide a more detailed explanation in Appendix B.

We conducted experiments in this section using 30 iteration loops per expression, with the beam size $b = 5$ for generating each prediction. In addition, we experimented with a method that combines NSR-gvs with TPSR; in this approach, each prediction is produced via MCTS-based decoding instead of simple beam search.

6.2 Results

We evaluated the impact of each test-time strategy on reproduction bias and numerical accuracy in an experimental setting. The experimental setup closely follows that described in Section 5.1. We trained a

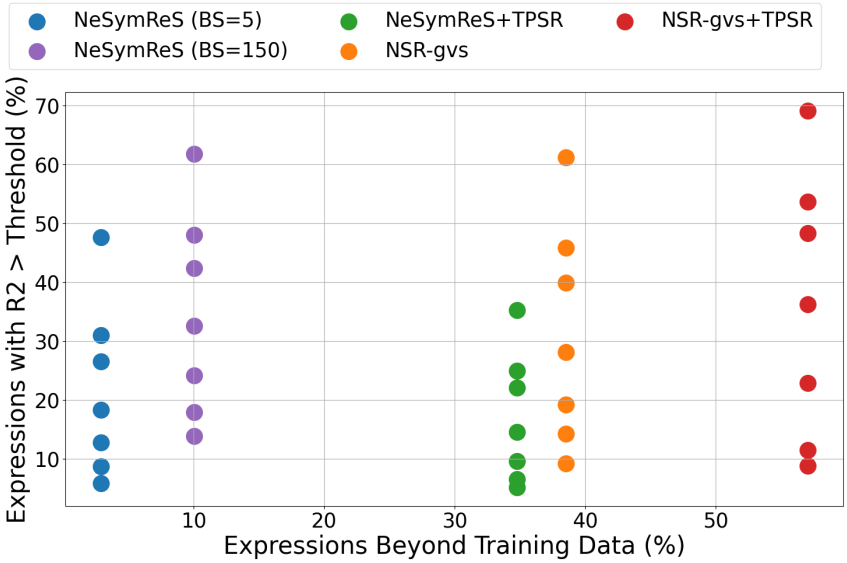

Figure 4: Evaluation of test-time strategies on the `not_included` dataset. The x-axis represents the percentage of expressions generated that were not included in the training data. The y-axis shows the proportion of expressions that exceeded the R^2 thresholds of 0.5, 0.9, 0.95, 0.99, 0.999, 0.9999 and 0.99999, respectively.

NeSymReS model and a prompt-augmented model for NSR-gvs with the same training dataset for the same number of epochs. We evaluated the strategies using the `not_included` dataset, where we used the R^2 metric to evaluate numerical accuracy, and the number of novel expressions to evaluate reproduction bias.

Figure 4 shows how the test-time strategies perform under the simplified setting. In terms of the ability to generate novel expressions, TPSR, NSR-gvs, and their combination demonstrate strong performance. These results imply that strategies involving the provision of additional information during inference (TPSR and NSR-gvs) are more effective in reducing reproduction bias. However, the result shows that high novelty in generated expressions does not necessarily imply high numerical accuracy. In some cases, acquiring the ability to generate novel expressions leads to a decrease in numerical accuracy (TPSR), whereas some strategies can improve numerical accuracy despite high reproduction bias (large beam size). This suggests that, in some cases, Transformers struggle to leverage the additional information effectively. We therefore argue that providing additional information at test time in a way that is easy for the Transformer to leverage is important for developing a truly generalizable NSR approach. Viewed in this way, the use of subtrees at inference, as in the proposed method NSR-gvs, can be seen as a potentially valuable approach, since it contributes to mitigating reproduction bias and improving numerical accuracy.

7 Conclusion

In this work, we identified a major drawback of standard NSR models both theoretically and empirically. Our theoretical analysis shows that Transformers are incapable of generating expressions in a compositional way, while taking numerical data into account. We then examined the strategies that Transformers actually employ to generate expressions, and the results suggest that they mostly generate expressions copied from the training data, highly limiting the search space. Finally, we demonstrate that incorporating additional information to the model during test-time can reduce reproduction bias. In future work, we aim to build on the findings of this study to design symbolic regression methods that further improve generalizability.

References

Ismail Alaoui Abdellaoui and Siamak Mehrkanoon. Symbolic regression for scientific discovery: an application to wind speed forecasting. In *2021 IEEE Symposium Series on Computational Intelligence (SSCI)*, pp. 01–08. IEEE, 2021.

Sanjeev Arora and Boaz Barak. *Computational complexity: a modern approach*. Cambridge University Press, 2009.

Tommaso Bendinelli, Luca Biggio, and Pierre-Alexandre Kamienny. Controllable neural symbolic regression. *arXiv preprint arXiv:2304.10336*, 2023.

Luca Biggio, Tommaso Bendinelli, Alexander Neitz, Aurelien Lucchi, and Giambattista Parascandolo. Neural symbolic regression that scales. In *International Conference on Machine Learning*, pp. 936–945. PMLR, 2021.

Bogdan Burlacu, Gabriel Kronberger, and Michael Kommenda. Operon c++ an efficient genetic programming framework for symbolic regression. In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion*, pp. 1562–1570, 2020.

Samuel R Buss. The boolean formula value problem is in alogtime. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pp. 123–131, 1987.

Miles Cranmer. Interpretable machine learning for science with pysr and symbolicregression. jl. *arXiv preprint arXiv:2305.01582*, 2023.

Guhao Feng, Bohang Zhang, Yuntian Gu, Haotian Ye, Di He, and Liwei Wang. Towards revealing the mystery behind chain of thought: a theoretical perspective. *Advances in Neural Information Processing Systems*, 36:70757–70798, 2023.

Roger Fletcher. *Practical methods of optimization*. John Wiley & Sons, 2000.

Arya Grayeli, Atharva Sehgal, Omar Costilla Reyes, Miles Cranmer, and Swarat Chaudhuri. Symbolic regression with a learned concept library. *Advances in Neural Information Processing Systems*, 37:44678–44709, 2024.

Pierre-Alexandre Kamienny, Stéphane d'Ascoli, Guillaume Lample, and François Charton. End-to-end symbolic regression with transformers. *Advances in Neural Information Processing Systems*, 35:10269–10281, 2022.

John R Koza. Genetic programming as a means for programming computers by natural selection. *Statistics and computing*, 4:87–112, 1994.

William La Cava, Patryk Orzechowski, Bogdan Burlacu, Fabrício Olivetti de França, Marco Virgolin, Ying Jin, Michael Kommenda, and Jason H Moore. Contemporary symbolic regression methods and their relative performance. *arXiv preprint arXiv:2107.14351*, 2021.

Florian Lalande, Yoshitomo Matsubara, Naoya Chiba, Tatsunori Taniai, Ryo Igarashi, and Yoshitaka Ushiku. A transformer model for symbolic regression towards scientific discovery. *arXiv preprint arXiv:2312.04070*, 2023.

Guillaume Lample and François Charton. Deep learning for symbolic mathematics. *arXiv preprint arXiv:1912.01412*, 2019.

Mikel Landajuela, Chak Shing Lee, Jiachen Yang, Ruben Glatt, Claudio P Santiago, Ignacio Aravena, Terrell Mundhenk, Garrett Mulcahy, and Brenden K Petersen. A unified framework for deep symbolic regression. *Advances in Neural Information Processing Systems*, 35:33985–33998, 2022.

Juho Lee, Yoonho Lee, Jungtaek Kim, Adam Kosiorek, Seungjin Choi, and Yee Whye Teh. Set transformer: A framework for attention-based permutation-invariant neural networks. In *International conference on machine learning*, pp. 3744–3753. PMLR, 2019.

Yanjie Li, Weijun Li, Lina Yu, Min Wu, Jingyi Liu, Wenqiang Li, Meilan Hao, Shu Wei, and Yusong Deng. Generative pre-trained transformer for symbolic regression base in-context reinforcement learning. *arXiv preprint arXiv:2404.06330*, 2024.

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

Matteo Merler, Katsiaryna Haitsiukevich, Nicola Dainese, and Pekka Marttinen. In-context symbolic regression: Leveraging large language models for function discovery. *arXiv preprint arXiv:2404.19094*, 2024.

William Merrill and Ashish Sabharwal. A logic for expressing log-precision transformers. *Advances in neural information processing systems*, 36:52453–52463, 2023a.

William Merrill and Ashish Sabharwal. The parallelism tradeoff: Limitations of log-precision transformers. *Transactions of the Association for Computational Linguistics*, 11:531–545, 2023b.

Aaron Meurer, Christopher P Smith, Mateusz Paprocki, Ondřej Čertík, Sergey B Kirpichev, Matthew Rocklin, AMiT Kumar, Sergiu Ivanov, Jason K Moore, Sartaj Singh, et al. Sympy: symbolic computing in python. *PeerJ Computer Science*, 3:e103, 2017.

T Nathan Mundhenk, Mikel Landajuela, Ruben Glatt, Claudio P Santiago, Daniel M Faissol, and Brenden K Petersen. Symbolic regression via neural-guided genetic programming population seeding. *arXiv preprint arXiv:2111.00053*, 2021.

Brenden K Petersen, Mikel Landajuela, T Nathan Mundhenk, Claudio P Santiago, Soo K Kim, and Joanne T Kim. Deep symbolic regression: Recovering mathematical expressions from data via risk-seeking policy gradients. *arXiv preprint arXiv:1912.04871*, 2019.

Michael Schmidt and Hod Lipson. Distilling free-form natural laws from experimental data. *science*, 324 (5923):81–85, 2009.

Samiha Sharlin and Tyler R Josephson. In context learning and reasoning for symbolic regression with large language models. *arXiv preprint arXiv:2410.17448*, 2024.

Parshin Shojaee, Kazem Meidani, Amir Barati Farimani, and Chandan Reddy. Transformer-based planning for symbolic regression. *Advances in Neural Information Processing Systems*, 36:45907–45919, 2023.

Parshin Shojaee, Kazem Meidani, Shashank Gupta, Amir Barati Farimani, and Chandan K Reddy. Llm-sr: Scientific equation discovery via programming with large language models. *arXiv preprint arXiv:2404.18400*, 2024.

David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144, 2018.

Lena Strobl. Average-hard attention transformers are constant-depth uniform threshold circuits. *arXiv preprint arXiv:2308.03212*, 2023.

Fangzheng Sun, Yang Liu, Jian-Xun Wang, and Hao Sun. Symbolic physics learner: Discovering governing equations via monte carlo tree search. *arXiv preprint arXiv:2205.13134*, 2022.

Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. *Advances in neural information processing systems*, 27, 2014.

Wassim Tenachi, Rodrigo Ibata, and Foivos I Diakogiannis. Deep symbolic regression for physics guided by units constraints: toward the automated discovery of physical laws. *arXiv preprint arXiv:2303.03192*, 2023.

Silviu-Marian Udrescu and Max Tegmark. Ai feynman: A physics-inspired method for symbolic regression. *Science Advances*, 6(16):eaay2631, 2020.

Silviu-Marian Udrescu, Andrew Tan, Jiahai Feng, Orisvaldo Neto, Tailin Wu, and Max Tegmark. Ai feynman 2.0: Pareto-optimal symbolic regression exploiting graph modularity. *Advances in Neural Information Processing Systems*, 33:4860–4871, 2020.

Mojtaba Valipour, Bowen You, Maysum Panju, and Ali Ghodsi. Symbolicgpt: A generative transformer model for symbolic regression. *arXiv preprint arXiv:2106.14131*, 2021.

Martin Vastl, Jonáš Kulhánek, Jiří Kubalík, Erik Derner, and Robert Babuška. Symformer: End-to-end symbolic regression using transformer-based architecture. *IEEE Access*, 2024.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

Marco Virgolin and Solon P Pissis. Symbolic regression is np-hard. *arXiv preprint arXiv:2207.01018*, 2022.

Marco Virgolin, Tanja Alderliesten, and Peter AN Bosman. Linear scaling with and within semantic backpropagation-based genetic programming for symbolic regression. In *Proceedings of the genetic and evolutionary computation conference*, pp. 1084–1092, 2019.

Yiqun Wang, Nicholas Wagner, and James M Rondinelli. Symbolic regression in materials science. *MRS Communications*, 9(3):793–805, 2019.

Yilong Xu, Yang Liu, and Hao Sun. Reinforcement symbolic regression machine. In *The Twelfth International Conference on Learning Representations*, 2024.

Hengzhe Zhang, Qi Chen, Wolfgang Banzhaf, Mengjie Zhang, et al. Rag-sr: Retrieval-augmented generation for neural symbolic regression. In *The Thirteenth International Conference on Learning Representations*, 2025.

Table 2: Operators used in NeSymReS

Arity	Operators

Unary	`pow2, pow3, pow4, pow5`
	`sqrt, log, exp`
	`sin, cos, asin`

Binary	`add, sub, mul, div`

Table 3: Hyperparameters in NeSymReS's dataset generation

Name	Explanation	Value
d	Dimension for input variables	5
n	Number of input points	Sampled from $\mathcal{U}(1, 1000)$
$\mathcal{D}_{\text{mul}}$	Distribution over multiplicative constants	Sampled from $\mathcal{LU}(0.05, 10)$
$\mathcal{D}_{\text{add}}$	Distribution over additive constants	Sampled from $\mathcal{U}(-10, 10)$
$\{x_{\min,j}\}_{j=1}^{d}$	Lower bound for sampling input variable	Sampled from $\mathcal{U}(-10, 9)$
$\{x_{\max,j}\}_{j=1}^{d}$	Upper bound for sampling input variable	Sampled from $\mathcal{U}(x_{\min,j} + 1, 10)$

A Details for NeSymReS

In this section, we present a detailed explanation for the study of NeSymReS that could not be fully explained in Section 3. We discuss the details of the dataset generation process, the model architecture, and the training procedure.

Generating the dataset. In the first step for generating the expression e^*, the unary-binary tree structure T is generated randomly within the limits of a maximum depth of 6. In the third step, the total number of constants added to the expression is also limited to a maximum of 6. The binary and unary operators $\mathcal{O}_{\text{bin}} \cup \mathcal{O}_{\text{un}}$ are shown in Table 2. Other hyparparameters are specified in Table 3, where $\mathcal{LU}$ denotes the log-uniform distribution. After the expression is sampled, the symbolic manipulation library Sympy (Meurer et al., 2017) is used to simplify any redundancy in the expression.

Model architecture. The NeSymReS model consists of two architectural components: the numerical encoder enc_{num} and a decoder dec. The numerical encoder processes the numerical data $\mathcal{D}$, represented as a tensor of shape (b, n, d), where b denotes the batch size, n the number of input points, and d the sum of dependent and independent variables. First, an embedding layer converts the numerical data into a higher dimensional tensor $\mathcal{D}'$ of shape (b, n, h). This tensor is then processed by a 5-layer set-transformer (Lee et al., 2019) encoder that outputs a new tensor Z_{num} of shape (b, s, h), where s denotes the number of embedding vectors produced by the encoder. The resulting tensor Z_{num} is subsequently passed to the decoder dec, a five-layer standard Transformer decoder that auto-regressively generates the corresponding expression token by token. We set $b = 200$, $h = 512$, and $s = 32$ for our experiments.

Details for training. During training, cross-entropy loss is used as the objective function, and teacher forcing (Sutskever et al., 2014) is applied during next-token prediction. The AdamW (Loshchilov & Hutter, 2017) optimizer is employed with an initial learning rate of 10^{-4}. After 4000 steps, the learning rate is adjusted proportionally to the inverse square root of the number of steps taken.

B Details for Test-time Strategies

This section is devoted to supplementing the details that were not fully covered in Section 6. We first supplement our explanation of TPSR, followed by a detailed formulation of NSR-gvs.

B.1 TPSR

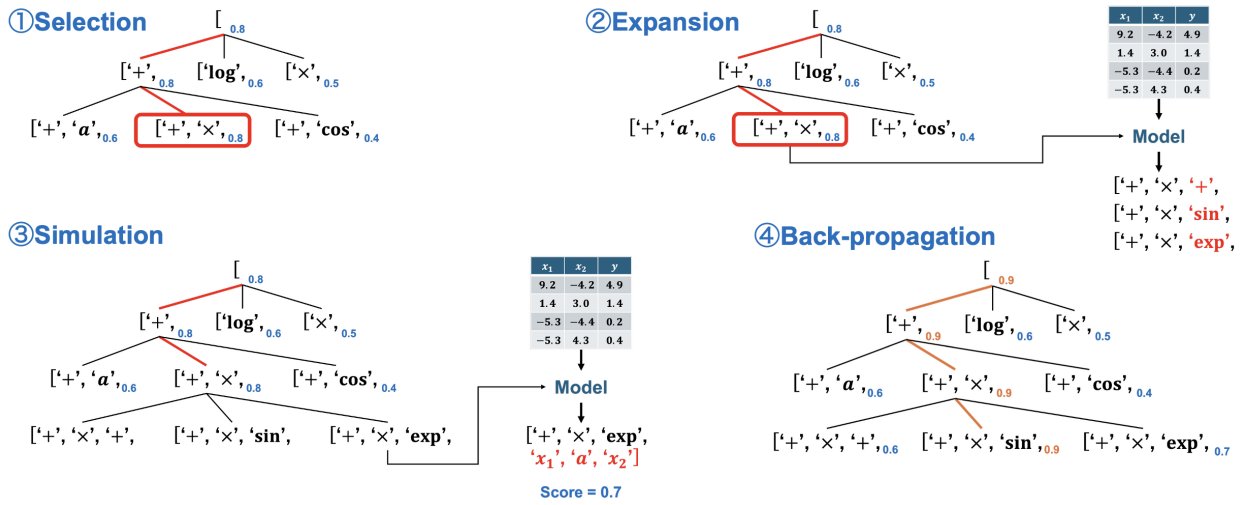

Figure 5: Overview of TPSR. Expressions are generated by MCTS with guidance from a NSR model. During the expansion phase, the next token to expand is decided based on the NSR model's logits, and during the simulation phase, the NSR model simulates the hypotheses instead of random simulation.

TPSR is a method where expressions are generated by MCTS. MCTS consists of 4 key steps, and here we supplement details for the steps that were not covered in the main text. We also provide an overview in Figure 5.

First, during the selection phase, the policy-guided upper confidence bound (P-UCB) heuristic (Silver et al., 2018) is used, where the NSR model is used as the policy.

Then, as described in Section 6, the NSR model decides the token to expand during the expansion phase and simulates the hypotheses during the simulation phase, playing a central role in the generation process. After the simulation phase, the reward is mainly calculated based on the generated expression's numerical accuracy, with additional consideration given to its complexity. In TPSR, a hyperparameter λ controls the balance between fitting accuracy and complexity. Given a set of n data points $\{\mathbf{x}_i, y_i\}_{i=1}^{n}$, and a candidate prediction $\tilde{f}$, the reward $r(\tilde{f}(\cdot) \mid \mathbf{x}, \mathbf{y})$ is calculated as follows:

$$r(\tilde{f}(\cdot) \mid \mathbf{x}, \mathbf{y}) = \frac{1}{1 + \mathrm{NMSE}(\mathbf{y}, \tilde{f}(\mathbf{x}))} + \lambda \exp\left(-\frac{|\mathrm{seq}(\tilde{f})|}{L}\right),$$

where seq is the serialization mapping introduced in Section 3, L denotes the the model's maximum sequence length, and NMSE represents the normalized mean square loss. In our work, we always set λ to 0.01, which is the default value in the original study of TPSR.

Finally, during the back-propagation phase, the scores for each node are updated by taking the maximum between the current score and the newly obtained reward value.

B.2 NSR-gvs

As described in Section 6, NSR-gvs is a method that iteratively improves its predictions by providing expression subtrees as prompts to the model and receiving feedback through verification. In this section, we formulate the training and inference procedures of NSR-gvs within the framework of reinforcement learning.

B.2.1 Training

We first introduce a *prompt-conditioned symbolic regressor* $S'_{\boldsymbol{\theta}}$ defined by parameters $\boldsymbol{\theta}$, that maps a numerical dataset $\mathcal{D}$ and an auxiliary prompt sequence $\mathbf{p}$ to a symbolic expression $\hat{e} = S'_{\boldsymbol{\theta}}(\mathcal{D}, \mathbf{p})$. Learning aims to align $\hat{e}$ with the ground-truth expression e^* underlying $\mathcal{D}$. Among the elements of the synthetic training tuple $(e^*, \mathcal{D}, \mathbf{p})$, the generation of e^* and $\mathcal{D}$ is the same as explained in Section 3. Here we specify how prompt sequences are constructed.

We first define extract : $\mathcal{E} \longrightarrow \mathcal{P}(\mathcal{E})$ as a stochastic mapping, which assigns to each symbolic expression $e \in \mathcal{E}$ a probability distribution over the subtrees of e. The space $\mathcal{P}(\mathcal{E})$ denotes the power set of expressions.

Using this stochastic mapping, we first obtain N subtrees $\{e'_i \mid e'_i \sim \text{extract}(e^*), i = 1, 2, \ldots, N\}$ from the ground-truth expression e^*. Then, each of the subtrees are converted to token sequences $\{\mathbf{t}_i \mid \mathbf{t}_i = \text{seq}(e'_i), i = 1, 2, \ldots, N\}$using the serialization map seq. Given an expression e^*, we construct the prompt:

$$\mathbf{p} \;=\; (\tau_{\text{start}}, \mathbf{t}_1, \tau_{\text{end}}, \tau_{\text{start}}, \mathbf{t}_2, \tau_{\text{end}}, \ldots, \tau_{\text{start}}, \mathbf{t}_N, \tau_{\text{end}}),$$

where tokens τ_{start} and τ_{end} are partition tokens representing the beginning and end of each subtree representation.

Similar to the formulation in Section 3, the predictive distribution $q'_{\boldsymbol{\theta}}(\cdot \mid (\mathbf{p}, \mathbf{s}_{<j}), \mathcal{D})$ is realized by an encoder-decoder Transformer parametrized by $\boldsymbol{\theta}$. In NSR-gvs, however, the decoder is conditioned on $(\mathbf{p}, \mathbf{s}_{<j})$, which is the concatenation of the prompt $\mathbf{p}$ and previously emitted prefix $\mathbf{s}_{<j}$.

B.2.2 Inference

During inference, we guide the symbolic regressor $S'_{\boldsymbol{\theta}}$ by prompting it with expression subtrees, which are obtained by a self-verification process. We formalize the inference-time mechanism of NSR-gvs within the framework of a Markov Decision Process (MDP). The core components of the MDP are defined as follows:

State space $\mathcal{S}$ and action space $\mathcal{A}$. The state at time t is denoted by $s_t \in \mathcal{S}$. The state is defined as $s_t = \{(e'_i, z_i, c_i) \mid i = 1, 2, \ldots, n_t\}$, which is a n_t-sized set comprising tuples of subtrees $e'_i \in \mathcal{E}$, its corresponding *verification scores* $z_i \in \mathbb{R}$, and its appearance count $c_i \in \mathbb{N}$. Therefore, the state space can be represented as $\mathcal{S} = \mathcal{P}(\mathcal{E} \times \mathbb{R} \times \mathbb{N})$. The action $a_t \in \mathcal{A}$ is a prompt sequence described in the previous subsection. The action space is represented as $\mathcal{A} = (\Gamma \cup \{\tau_{\text{start}}, \tau_{\text{end}}\})^*$.

Policy $\pi(a_t \mid s_t)$. We define a stochastic policy to sample an action a_t from the current state s_t. An action is sampled following the procedure below.

First, we deterministically select a set of subtrees E'_{topk}, consisting of the top k subtrees with the highest verification scores in state s_t, as follows:

$$E'_{\text{topk}} \;=\; \{e'_i \mid (e'_i, z_i, c_i) \in s_{\text{topk}},\, i = 1, 2, \ldots, k\}, \quad \text{where} \quad s_{\text{topk}} = \underset{s \subseteq s_t,\, |s| = k}{\text{argmax}} \sum_{(e', z, c) \in s} z.$$

Subsequently, we filter out subtrees whose corresponding score z is smaller than a threshold value z_{thres}. The purpose of this operation is to prioritize exploration over exploitation when the quality of obtained subtrees are poor.

Next, we construct a set E'_{rand} by extracting k_{rand} subtrees from expressions sampled from the expression generator $p_{\mathcal{E}}$:

$$E'_{\text{rand}} \;=\; \{e'_i \mid e'_i \sim \text{extract}(e),\, e \sim p_{\mathcal{E}},\, i = 1, 2, \ldots, k_{\text{rand}}\}.$$

Finally, we uniformly sample a set of subtrees from the merged set $E'_{\text{topk}} \cup E'_{\text{rand}}$ and convert them to tokens in the same way as during training time, resulting in a prompt sequence a_t. During sampling, we filter out subtrees whose token representation is longer than l_{max}, and we sample subtrees until the total length of the subtrees' token representation exceeds the limit l_t.

By sampling from both the self-verification-based set E'_{topk} and the randomly obtained set E'_{rand}, the policy enables both exploration and exploitation. The hyperparameters k, k_{rand}, z_{thres}, l_{max}, and l_t characterize the policy.

Table 4: Hyperparameters in NSR-gvs

Name	Explanation	Value
k	Size of high-scored subtree set E'_{topk}	39
k_{rand}	Size of randomly sampled subtree set E'_{rand}	9
z_{thres}	Threshold value for high-scored subtrees	0.213
$l_{\max}$	Maximum length of a subtree's representation	9
l_t	Total length of the subtrees' representation	Sampled from $\mathcal{U}(0, \lfloor 15.58 + 0.42t \rfloor)$

Reward function $R(a_t, s_t)$ and transition probability $T(s_{t+1} \mid a_t, s_t)$. After an action a_t is sampled, it is provided to the prompt-conditioned symbolic regressor $S'_{\boldsymbol{\theta}}$ together with numerical data $\mathcal{D}$. We compute the reward based on the numerical accuracy of the prediction $\hat{e} = S'_{\boldsymbol{\theta}}(\mathcal{D}, a_t)$.

Let $\mathcal{L} : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}$ be a metric to evaluate the difference between two vectors (in practice, we use the R^2 value described in Section 5). When $\mathcal{D} = \{(\mathbf{x}_i,\, y_i)\}_{i=1}^n$, the reward is computed as:

$$R(a_t, s_t) = \mathcal{L}(\mathbf{y}, \hat{e}(\mathbf{x})).$$

Finally, we define the transition probability $T(s_{t+1} \mid a_t, s_t)$, determined by the following process. We denote by $\hat{E}'$ the set comprising all subtree expressions of $\hat{e}$. For each subtree $\hat{e}'$ in $\hat{E}'$, we update s_t so that the verification score of each subtree matches the average reward of all expressions that included the subtree, as described below.

1. If $\forall (e', z, c) \in s_t, \hat{e}' \neq e'$ holds, add the tuple $(\hat{e}', R(a_t, s_t), 1)$ to s_t.
2. If $\exists (e', z, c) \in s_t, \hat{e}' = e'$ holds, replace the tuple $(\hat{e}', z, c)$ with $(\hat{e}', \dfrac{cz + R(a_t, s_t)}{c+1}, c+1)$.

The updated state serves as the state s_{t+1} at the next timestep $t+1$.

The overall algorithm during inference-time is detailed in 1. For the hyperparameters that characterize the policy, we use the values shown in Table 4, which were tuned via Bayesian optimization on 5 randomly generated expressions. The function $\lfloor \cdot \rfloor$ indicates the floor function, which rounds down the input to its nearest integer.

Algorithm 1 Inference-time Algorithm

```
function VERIFY(e, D)
    (X, y) ← D
    ŷ ← e(X)
    Compute R^2 score between y and ŷ
    return R^2
end function

function UPDATE(e, s_t, R^2)
    E'_p ← Partial expressions extracted from e
    s_{t+1} ← [ ]
    for (e_p, z, c) in s_t do
        if e_p ∈ E'_p then
            z ← (cz + R^2) / (c + 1)
            c ← c + 1
        end if
        Append (e_p, z, c) to s_{t+1}
    end for
    return s_{t+1}
end function

function MAIN(D)
    s_1 ← [ ]
    e_best ← None
    R^2_best ← −∞
    for t ← 1 to T do
        if t = 1 then
            a_t ← [ ]
        else
            E'_topk ← Top k expressions in s_t with high score
            Filter out expressions in E'_topk whose corresponding score z < z_thres
            E'_rand ← Randomly sampled partial expressions
            E'_merged ← E'_topk ∪ E'_rand
            a_t ← Uniformly sampled subset from E'_merged, converted to tokens
        end if
        s ← Transformer(D, a_t)
        Convert sequence s to expression e
        R^2 ← Verify(e, D)
        s_{t+1} ← Update(e, s_t, R^2)
        if R^2 > R^2_best then
            e_best ← e
            R^2_best ← R^2
        end if
    end for
    return e_best
end function
```

C Details for Experiments, Implementation, and use of LLMs

In this section, we describe the details for the experiments conducted in Section 5, 6, and F. We also provide details regarding our implementation and the computational resources used in our experiments.

We provide the model with 100 data points in all experiments. We selected the range of the data support as follows: for the AI Feynman dataset, we used the support defined by the dataset itself. For all other datasets, we sampled the support range using the same procedure as used when generating the training data. For error bars, we report the standard deviation across three different random seeds. For the method combining NSR-gvs and TPSR, however, we conducted experiments with only a single seed due to the long inference time. Our implementation for data generation, model training, and related components is based on the original implementation of NSRwH [1]. For the transformer4sr and TPSR experiments, we used the official implementation provided by the authors [2], [3]. For both implementations, we used the version of the implementation that was available on May 15. 2025. We trained and tested the model on a single NVIDIA A100 GPU. Training requires approximately 24 hours either for 1000 epochs on a dataset with $100,000$ expressions or for 10 epochs on a dataset with 10 million expressions. The time required to generate a single expression at test time is less than one minute when using only NeSymReS or NSRwH, approximately 3 to 10 minutes with TPSR or NSR-gvs, and around 2 to 5 hours when combining TPSR with NSR-gvs.

We used large language models (LLMs) to aid writing and coding, where we mainly used Gemini 2.5 Flash and GPT-5 to generate code and check on errors in writing.

D Discussion Concerning the Definition of Reproduction Bias

We first restate the definition of reproduction bias introduced in Section 5.1. This definition was introduced to discuss the novelty of expressions in the NeSymReS setting without considering constants.

Definition D.1 (Reproduction bias)**.** Given an output token sequence $\mathbf{s}$, let $e = \mathrm{seq}^{-1}(\mathbf{s})$ be the original expression that is represented by $\mathbf{s}$. If $\mathrm{strip}(e) \in E_{\mathrm{templ}}$, we say $\mathbf{s}$ is a *reproduction* of expressions seen during training.

Here, we define reproduction bias in the transformer4sr setting. NeSymReS and transformer4sr are different in the way expressions are stored in the training data; in NeSymReS, constant-free expressions e_{templ} are collected to form a set of training expressions E_{templ}, whereas in transformer4sr, full expressions including constants, denoted by e^*, are stored, and the training dataset consists of the set E^* of such expressions.

For transformer4sr, we measured expression novelty both with and without considering constants. If we consider the set

$$E^*_{\mathrm{templ}} = \{e^*_{\mathrm{templ}} \mid e^*_{\mathrm{templ}} = \mathrm{strip}(\mathrm{e}^*)\} \quad \text{for all} \quad e^* \in E^*$$

the definition of reproduction bias without considering constants would be the same as Definition D.1. When considering constants, the definition of reproduction bias would be slightly changed as follows:

Definition D.2 (Reproduction bias considering constants)**.** Given an output token sequence $\mathbf{s}$, let $e = \mathrm{seq}^{-1}(\mathbf{s})$ be the original expression that is represented by $\mathbf{s}$. If $e \in E^*$, we say $\mathbf{s}$ is a *reproduction* of expressions seen during training.

Next, we discuss the problem of structural equivalence and functional equivalence in the transformer4sr setting. While we have defined reproduction bias based on structural equivalence, one may argue that we should define and measure reproduction based on functional equivalence. For example, $x_1(x_1 + x_2)$ and $x_1^2 + x_1x_2$ are different expressions from a structural perspective, but they are functionally equivalent. We stated in Remark 5.2 that for the NeSymReS setting, structural equivalence and functional equivalence can be treated the same due to the operator choices (there is no possibility that, for example, $x_1(x_1 + x_2)$ is generated when $x_1^2 + x_1x_2$ is included in the training dataset). In the transformer4sr setting, this is not always the case, since we do not restrict the operators in this setting.

[1] `https://github.com/SymposiumOrganization/ControllableNeuralSymbolicRegression`
[2] `https://github.com/omron-sinicx/transformer4sr`
[3] `https://github.com/deep-symbolic-mathematics/TPSR`

However, even if we were to redefine reproduction bias from the perspective of functional reproduction bias, the claims of Section 5.2 would remain unaffected. This is because every expression regarded as novel under the definition of functional reproduction bias is already also regarded as novel under the definition of structural reproduction bias. Since the claim of 5.2 is that the proportion of expressions classified as novel is small for naive NSR methods, adopting the definition of functional reproduction bias would only further reduce that proportion, without altering the direction of the conclusions.

E Additional Related Work

In this section, we describe symbolic regression methods other than NSR. Specifically, we provide explanation for methods that use GP, brute-force algorithms, and reinforcement learning.

The GP framework is a traditional and widely used framework for solving symbolic regression. The GP algorithm is a method based on evolutionary computation; initially, several mathematical expressions are formed randomly, and subsequently the expressions are "evolved" by operations such as recombining two expressions, mutating an expression, and eliminating inappropriate expressions (Burlacu et al., 2020; Schmidt & Lipson, 2009; Virgolin et al., 2019; Cranmer, 2023).

An example of using brute-force algorithms for symbolic regression is AI Feynman (Udrescu & Tegmark, 2020; Udrescu et al., 2020). In AI Feynman, neural networks were used to identify properties such as symmetry and separability within given numerical data. These properties were then used to recursively simplify the problem, ultimately reducing it to a form amenable to brute-force solutions.

Petersen et al. (2019) proposed Deep Symbolic Regression (DSR), a method that uses reinforcement learning to tackle symbolic regression. In this approach, the authors used a recurrent neural network (RNN) to generate equations as token sequences, with the parameters that govern the selection of the token learned through reinforcement learning. Studies such as Symbolic Physics Learner (Sun et al., 2022) and Reinforcement Symbolic Regression Machine (Xu et al., 2024) also use reinforcement learning, where Monte Carlo Tree Search (MCTS) is applied to discover expressions.

Some studies combine several approaches for symbolic regression. For example, neural-guided genetic programming (Mundhenk et al., 2021) integrates DSR and genetic programming (GP), while the Unified DSR Framework (Landajuela et al., 2022) combines GP, AI Feynman, DSR, linear models, and NSR.

F Additional Experiments

F.1 Do Novel Expressions Contribute to Improvements in Numerical Accuracy?

In Section 6, we saw that providing additional information to the model during inference can lead to generation of novel expressions. However, we also demonstrated that mitigating reproduction bias does not necessarily lead to better numerical accuracy. In this section, we analyzed how much the novel expressions generated under each test-time strategy (including NSRwH) contribute to improvements in numerical accuracy, and present the corresponding results in Table 5. "Novel" indicates that the generated expression does not appear in the training data, while "Not Novel" means it does. The values indicate the percentage of expressions that satisfy each condition.

The results show that for test-time strategies that are capable of mitigating reproduction bias (strategies shown in bold), a large proportion of generated novel expressions do not perform well in terms of numerical accuracy. Especially for TPSR, hardly any of the novel expressions exhibit high numerical accuracy. This indicates the difficulty of generating appropriate expressions from an expanded search space. However, for strategies using NSR-gvs, novel expressions contribute to high accuracy to some extent, showing that additional information can be beneficial for both mitigating reproduction bias and improving numerical accuracy in some occasions.

Table 5: Breakdown of generated expressions by novelty and high accuracy ($R^2 > 0.99$) across test-time strategies

Test-time Strategy	Novel, $R^2 > 0.99$	Novel, $R^2 \leq 0.99$	Not Novel, $R^2 > 0.99$	Not Novel, $R^2 \leq 0.99$
NeSymReS (BS=1)	0.45	2.89	12.92	83.74
NeSymReS (BS=5)	0.67	2.23	17.59	79.52
NeSymReS (BS=50)	4.23	2.23	25.17	68.38
NeSymReS (BS=100)	4.45	2.45	26.72	66.37
NeSymReS (BS=150)	6.25	3.79	26.33	63.62
NeSymReS + TPSR (BS=1)	0.45	**27.17**	14.70	57.69
NeSymReS + TPSR (BS=3)	0.67	**29.62**	18.04	51.67
NeSymReS + TPSR (BS=5)	2.02	**34.31**	18.16	45.51
NSR-gvs (BS=1)	4.91	**38.18**	14.05	42.86
NSR-gvs (BS=3)	6.67	**36.00**	15.56	41.78
NSR-gvs (BS=5)	8.68	**30.51**	19.38	41.43
NSR-gvs + TPSR (BS=1)	12.75	**44.30**	23.46	19.46
NSRwH (Complexity, BS=5)	2.23	5.80	15.18	76.78
NSRwH (Symmetry, BS=5)	1.11	2.90	14.93	81.06
NSRwH (Positives, BS=5)	2.46	**24.83**	6.71	66.00
NSRwH (Negatives, BS=5)	0.89	4.68	13.81	80.62
NSRwH (All, BS=5)	4.90	**47.66**	4.23	43.21

F.2 Trade-off Between Performance and Computational Cost

The results in Section 6 show how the relationship between reproduction bias and numerical accuracy differ between various test-time strategies. However, test-time strategies also differ in terms of the computational cost required to generate an expression. In this section, we aim to better understand each test-time strategy by analyzing the trade-off between performance and the computational cost of expression generation. We also varied the beam size during decoding for NeSymReS, TPSR, and NSR-gvs for a more comprehensive analysis. We tested under the controlled setting described in Section 5, using the `not_included` dataset as the test dataset.

To measure the computational cost, we followed the approach of Shojaee et al. (2023) and used the number of candidate expressions generated by the model during the generation of a single equation. For example, this value corresponds to the beam size in NeSymReS, the number of total rollouts multiplied by beam size in TPSR, and the number of iteration loops multiplied by beam size in NSR-gvs.

We present the results in Figure 6. It can be observed that, unlike NeSymReS—where larger beam size yields only limited reduction in reproduction bias—TPSR and NSR-gvs achieve notable reductions in reproduction bias at comparable computational costs. However, in terms of numerical accuracy, simply increasing the beam size in NeSymReS yields better performance than using NSR-gvs or TPSR at a comparable computational cost. The results support the conclusion in Section 6 that the reduction of reproduction bias is only weakly correlated with numerical accuracy.

F.3 Can NSRwH Also Mitigate Reproduction Bias?

When researchers in fields of natural sciences or engineering model their experimental data, they often make use of prior knowledge. For example, scientists may anticipate a symmetry between variables or predict that a particular operator appears in the mathematical laws describing the data. NSRwH (Bendinelli et al., 2023) is a method that enables incorporating such prior knowledge into the NeSymReS model. The types of prior knowledge provided to the model include the following:

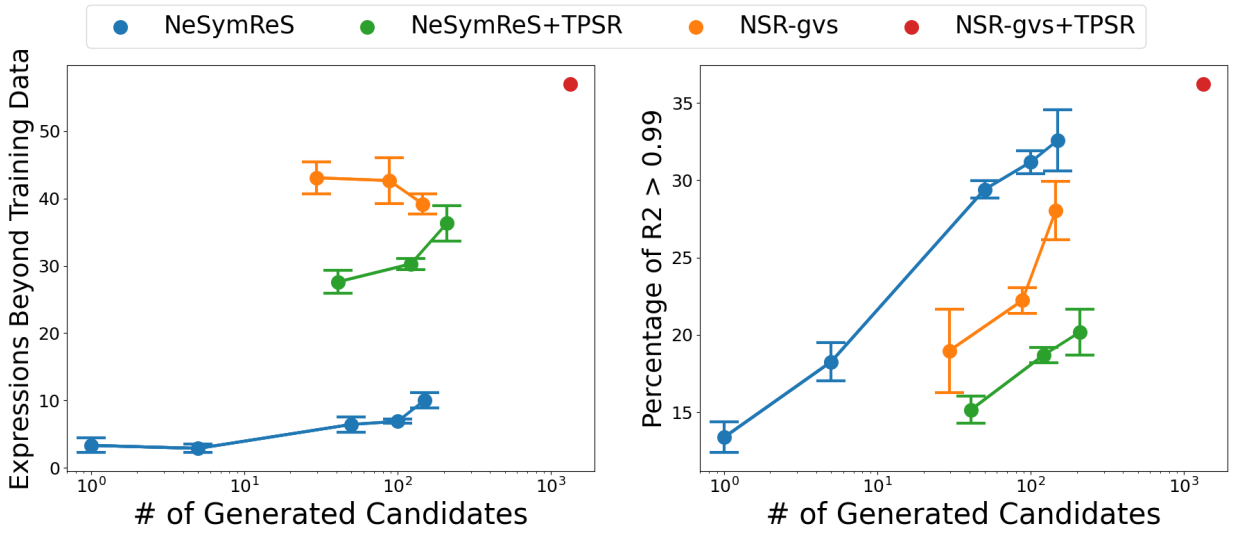

Figure 6: Trade-off between performance and computational cost for different test-time strategies. We varied the beam sizes for each model as follows: $\{1, 5, 50, 100, 150\}$ for NeSymReS, and $\{1, 3, 5\}$ for both NeSymReS+TPSR and NSR-gvs. For NSR-gvs+TPSR, we only experimented with beam size set to 1. The left figure shows the trade-off between the ability to generate expressions and computational cost, while the right figure shows the trade-off between numerical accuracy and computational cost.

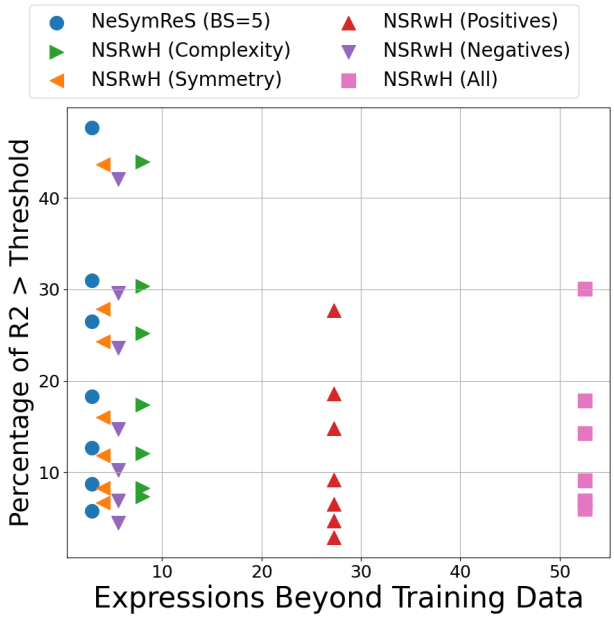

Figure 7: Evaluation of NSRwH on the `not_included` dataset. The x-axis represents the percentage of expressions generated that were not included in the training data. The y-axis shows the proportion of expressions that exceeded the R^2 thresholds of 0.5, 0.9, 0.95, 0.99, 0.999, 0.9999 and 0.99999, respectively.

- **Complexity.** The complexity of an expression is defined by the number of tokens used in the expression's token sequence. The model is provided with the complexity of the ground-truth expression.
- **Symmetry.** The presence or absence of symmetry among the input variables is provided to the model.
- **Positives.** Subtrees appearing in the ground-truth expression are provided to the model. Additionally, the value of constants appearing in the ground-truth expression may also be provided.
- **Negatives.** Subtrees that do not appear in the ground-truth expression are provided to the model.

In NSRwH, prior knowledge is encoded by an additional symbolical encoder enc_{sym}. The output of the symbolical encoder is summed together with the output of NeSymReS's numerical encoder and is fed to the decoder.

While prior knowledge is required beforehand to use NSRwH, it is a method that provides the model with additional information during inference, similar to TPSR and NSR-gvs. In this section, we test whether NSRwH can mitigate reproduction bias when prior knowledge is provided. We obtained a NSRwH model by finetuning the NeSymReS model that we trained in Section 6. We froze the numerical encoder of the NeSymReS model, attached a symbolical encoder, and finetuned the model for 250 epochs. We used the same training dataset as in Section 5 consisting of $100,000$ expressions; however, during fine-tuning, prior knowledge was extracted from the ground-truth expressions and fed into the symbolic encoder. At test time, we evaluated the NSRwH model under settings where each type of prior knowledge is provided individually, as well as under a setting where all types of prior knowledge are provided simultaneously. We follow the default settings of NSRwH to determine the amount of prior knowledge provided during test-time, and we used the `not_included` dataset as the test dataset. We set the beam size to 5 and compare the results with those of NeSymReS, which is also configured with a beam size of 5.

Figure 7 shows the results for this experiment. While providing complexity, symmetry, or absent subtrees mitigates reproduction bias only to a limited extent, providing appearing subtrees or providing all properties significantly mitigates reproduction bias. However, we also observe that the numerical accuracy of NSRwH decreases when provided with appearing subtrees or with all properties. This indicates a limitation of NSRwH when dealing with data not included in the training set. The results also show that not all kinds of additional data are effective for mitigating reproduction bias.

F.4 Numerical Accuracy in Practical Settings

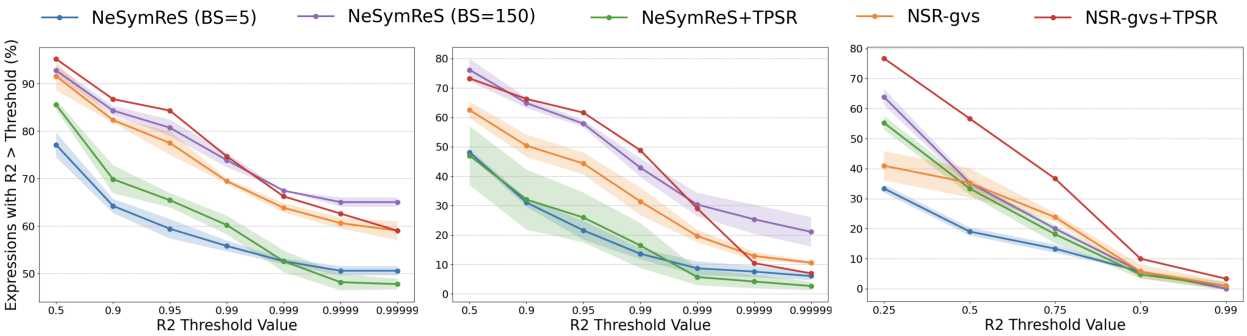

Figure 8: Comparison of test-time strategies on the `AI-Feynman` dataset, `only_five_variables_nc`, and `black-box` datasets from left to right. The general pattern remains consistent to the simplified setting, where large beam search and NSR-gvs+TPSR perform well, and NeSymReS+TPSR perform poorly.

Since the experiments in the main text were conducted under a relatively small training dataset with restricted operators, we additionally evaluated and compared the numerical performance of the test-time

strategies under conditions that better reflect practical applications. A total of 10 million expressions were used to construct the training dataset, employing all operators described in Section 3 without any restriction on operator types. We trained both a NeSymReS model and a prompt-augmented model on this dataset for 10 epochs. For the test datasets, we prepared the following three sets:

- `AI-Feynman.` This dataset consists of 91 equations with up to five independent variables, extracted from the AIFeynman database (Udrescu & Tegmark, 2020). It is commonly used in various studies to assess the performance of symbolic regression methods.
- `only_five_variables_nc.` This dataset consists of expressions containing exactly five independent variables, making it a challenging dataset. The "nc" designation indicates that the expressions do not include constants, which simplifies the problems slightly; however, it remains more difficult than the first dataset. The dataset was constructed by sampling expressions from $p_{\mathcal{E}}$, filtering for expressions that include exactly five variables, and finally deleting its constants. This dataset is derived from the study of NSRwH (Bendinelli et al., 2023), and we use the first 100 expressions for evaluation.
- `black-box`. We also evaluated on numerical data collected from the real world, whose ground-truth expressions do not exist. We extracted 35 expressions from the black-box dataset in SR-Bench (La Cava et al., 2021) whose number of independent variables are five or less. The data are often noisy and may be sampled from a range different from the numerical data that the models were trained on, making the task challenging for the test-time computation methods.

Figures 8 demonstrates how the different test-time strategies perform under `AI-Feynman`, `only_five_variables_nc`, and `black-box` datasets. TPSR relatively performs slightly better than in the controlled setting; however, the general pattern of numerical accuracy remains consistent. These results demonstrate that, NSR-gvs is able to improve performance in practical settings, including those with noisy data. However, it can also be seen that even in practical settings, test-time strategies that mitigate reproduction bias do not always result in better performance.

F.5 Varying the Number of Epochs in transformer4sr

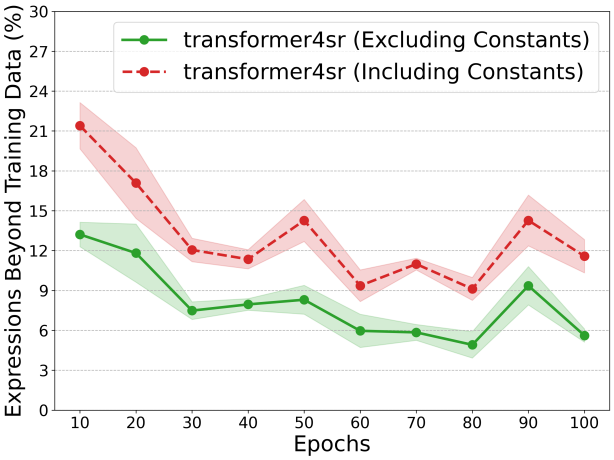

Figure 9: Reproduction bias in transformer4sr with varying epochs. Throughout the training process, the majority of both expression trees (both including and excluding constants) are copied directly from the training data.

In Section 5, we tested whether reproduction bias occurs in the practical setting of transformer4sr, while varying the training dataset size. Here, we present the results of an analysis of reproduction bias as a

function of the number of training epochs. In the experiments reported in Section 5, the number of training epochs was fixed to 100 following the original paper; therefore, we evaluate reproduction bias by varying the number of epochs from 10 to 100 in increments of 10. The results are shown in Figure 9. While reproduction bias is slightly lower at smaller numbers of epochs, it remains pronounced throughout the training process, indicating that reproduction bias cannot be attributed merely to underfitting or overfitting.

F.6 Further Results on the `baseline` dataset

As described in Section 5, the empirical results show that the `baseline` dataset is a much more easier dataset compared to the `not_included` dataset with naive inference. Since the majority of expressions in the `baseline` dataset are expressions that were included in the training dataset, by applying test-time strategies to this dataset, we can test whether the test-time strategies help in reproducing the expressions from the training data.

In Figure 10 and Figure 11, we present the results concerning the numerical accuracy for various test-time strategies. We also tested with NSRwH as well as the test-time strategies described in Section 6. The results show that only beam search with a large beam size significantly improves performance, aligning with the previous result in Section 6 that providing the model with additional information often prevents the model from reproducing expressions from the training data.

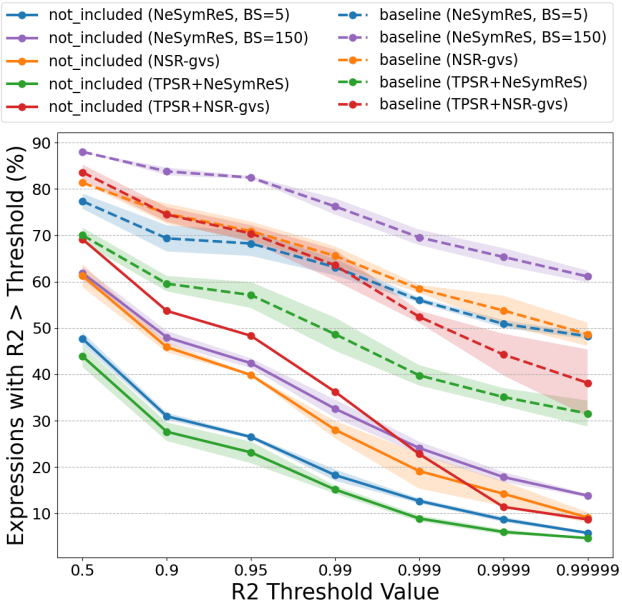

Figure 10: Numerical accuracy for various test-time strategies on the `baseline` (and `not_included`) dataset. The y-axis shows the proportion of expressions that exceeded the R^2 thresholds of 0.5, 0.9, 0.95, 0.99, 0.999, 0.9999 and 0.99999, respectively. Only beam search with a large beam size significantly improves performance for the `baseline` dataset.

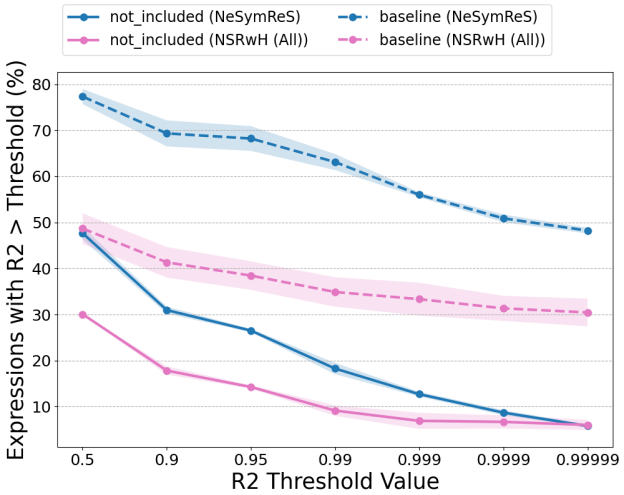

Figure 11: umerical accuracy for NSRwH on the `baseline` (and `not_included`) dataset. The y-axis shows the proportion of expressions that exceeded the R^2 thresholds of 0.5, 0.9, 0.95, 0.99, 0.999, 0.9999 and 0.99999, respectively. Adding additional information significantly curves the performance in the `baseline` dataset.

G Theoretical Background and Proof

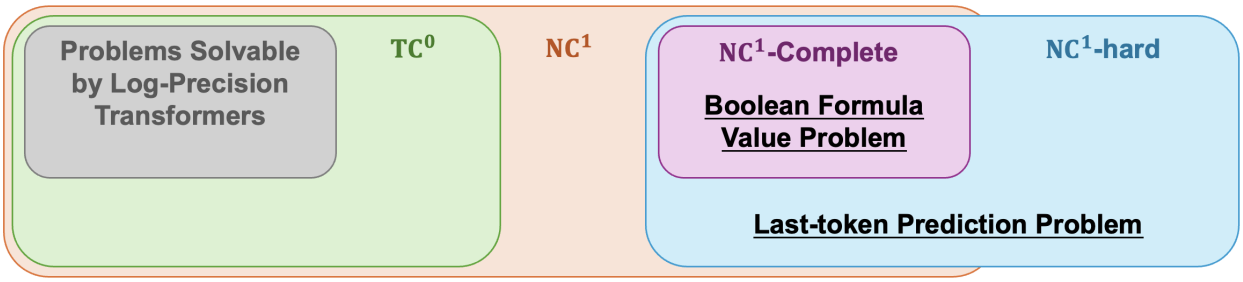

Figure 12: Overview of the theoretical analysis. When assuming $\mathsf{TC}^0 \neq \mathsf{NC}^1$, the last-token prediction problem is NC^1-hard (due to reduction from Boolean formula value problem) while Transformers can only solve problems in TC^0.

In this section, we provide background knowledge, detailed settings, and a complete proof for the theoretical result presented in Section 4.

G.1 Preliminary

We first provide a brief overview of relevant circuit complexity classes. We then define the class of log-precision Transformers and introduce its simulation guarantees. We also present a formal definition of the Boolean formula value problem, which we use in our proof.

G.1.1 Circuit Complexity Classes

We offer an explanation to several fundamental circuit complexity classes that are used in our theoretical analysis. Particularly, we discuss the complexity classes $\mathsf{AC}^0, \mathsf{TC}^0$ and , NC^1. The relationship between these

three classes can be summarized as follows:

$$\mathsf{AC}^0 \subsetneq \mathsf{TC}^0 \subset \mathsf{NC}^1.$$

Whether TC^0 is a proper subset of NC^1 is an open question, but it is widely believed that this is the case. For a more detailed and comprehensive introduction, we recommend reference to (Arora & Barak, 2009).

Circuit class AC^0. The class AC^0 consists of Boolean circuits of *constant depth* and *polynomial size* whose gates have unbounded fan-in and are restricted to the basis $\{\mathsf{AND}, \mathsf{OR}, \mathsf{NOT}\}$. Intuitively, AC^0 captures extremely shallow parallel computation.

Circuit class TC^0. The class TC^0 is an extension of AC^0, where a gate called the *majority gate* can be additionally used. A majority gate has unbounded fan-in and outputs false when half or more of the inputs are false, and true otherwise. Other definitions are the same as AC^0.

Circuit class NC^1. Circuits in NC^1 are polynomial sized with the depth *logarithmic* to the input size. They comprise of $\{\mathsf{AND}, \mathsf{OR}, \mathsf{NOT}\}$ gates with constant fan-in. The class NC^1 contains several well-known problems such as the parity check on a bit string.

G.1.2 Log-precision Transformers

We assume bounded-depth log-precision Transformers throughout the theoretical analysis. We first model the parametrized Transformer $\mathsf{TF}_{\boldsymbol{\theta}}$ as a next-token prediction function;

$$\mathsf{TF}_{\boldsymbol{\theta}} : \Gamma^m \times \mathbb{R}^{(d+1)\times n} \longrightarrow \Gamma, \tag{2}$$

i.e. the Transformer receives a length-m prefix along with a numerical dataset $\mathcal{D}$ and outputs a single token $u \in \Gamma$.

Definition G.1 ((D, d)-bounded log-precision Transformer)**.** Let k be the input length. A (D, d)-bounded log-precision Transformer is an encoder–decoder model that satisfies

1. constant depth $D = O(1)$,
2. hidden size $d \leq Q(k)$ for a fixed polynomial Q,
3. the values at all layers, as well as the outputs of all key intermediate operations in it (attention, activation, arithmetic operators, etc.), are represented using $O(\log k)$ bits.

For specific definitions of operations that enable approximation in $O(\log k)$ bits, please refer to Section 4 and Appendix A of Merrill & Sabharwal (2023b). We introduce the simulation guarantees for bounded-depth log-precision Transformers as follows.

Lemma G.2 (Circuit simulation (Merrill & Sabharwal, 2023b, Cor. 2.1))**.** *Any (D, d)-bounded log-precision Transformer can be simulated by a family of TC^0 circuits of size* $\mathrm{poly}(k)$ *and constant depth with respect to k.*

G.1.3 The Boolean Formula Value Problem

Following the definition by Buss (1987), we introduce the definition of the Boolean formula value problem as follows.

Definition G.3 (Boolean formula value problem)**.** Let $\Lambda = \{0, 1, \wedge, \vee, \neg, (,)\}$ be the alphabet. A Boolean formula is a string defined recursively as follows:

1. 0 and 1 are Boolean formulae;
2. If $\mathbf{t}_1$ and $\mathbf{t}_2$ are two Boolean formulae, then $(\neg \mathbf{t}_1), (\mathbf{t}_1 \wedge \mathbf{t}_2), (\mathbf{t}_1 \vee \mathbf{t}_2)$ are also Boolean formulae.

When given a boolean formula $\mathbf{t}$, the goal of the Boolean formula value problem is to compute whether the evaluation result $\mathrm{eval}(\mathbf{t})$ of a given Boolean formula is 0 or 1.

G.2 Main Theorem

Prior to proving the main theorem, we state the following Lemma from Feng et al. (2023). The detailed proof of this Lemma can be found in the same paper. The Lemma states that TC^0 circuits are capable of identifying the indexes of paired brackets in a string.

Lemma G.4 (Bracket parsing (Feng et al., 2023, Lem. D.3))**.** *Consider any string* $\mathbf{t} = t_1 t_2 \cdots t_n$ *of length* n *containing brackets '(', ')', and other characters, and all brackets in* $\mathbf{t}$ *are paired. Let* $\boldsymbol{g}$ *be a boolean function taking* $\mathbf{t}$ *as input and output* n *pairs of integers defined as follows:*

$$\boldsymbol{g}_i(\mathbf{t}) = \begin{cases} (-1, j) & \text{if } t_i \text{ is a left bracket and } t_i, t_j \text{ are paired.} \\ (j, -1) & \text{if } t_i \text{ is a right bracket and } t_i, t_j \text{ are paired.} \\ (j, k) & \text{if } t_i \text{ is not a bracket, and } t_j, t_k \text{ are the nearest paired brackets containing } t_i. \end{cases}$$

Then $\boldsymbol{g}$ *can be implemented by the* TC^0 *circuits.*

We now proceed to prove the main theorem of our theoretical analysis.

Theorem G.5 (Bounded log-precision Transformer lower bound)**.** *Assume* $\mathsf{TC}^0 \neq \mathsf{NC}^1$*. For any integer* D *and any polynomial* Q*, there exists a problem size* m *such that no* (D, d)*-bounded log-precision Transformer with* $d \leq Q(m)$ *can solve* LastTokenPrediction(m)*.*

Proof. Fix D and Q and suppose, for contradiction, that for some sufficiently large m there exists an (D, d)-bounded log-precision Transformer TF_θ with $d \leq Q(m)$ that solves the problem of LastTokenPrediction(m).

Step 1 (simulation). By Lemma G.2, TF_θ can be simulated by a TC^0 circuit family of size poly(m). Hence, under our assumption, LastTokenPrediction$(m) \in \mathsf{TC}^0$.

Step 2 (TC^0 construction). We show that there exists a TC^0 circuit that can translate any instance of a Boolean formula value problem to an instance of the last-token prediction problem.

Let $\mathbf{t}$ be a boolean formula. There exists a TC^0 circuit that performs:

1. *Translation of* $\mathbf{t}$ *to a* $\vee$*-free Boolean formula* $\mathbf{t}'$*.*
2. *Conversion of* $\mathbf{t}'$ *to its prefix notation* $\mathbf{t}'_{\text{pre}}$*.*
3. *Conversion of* $\mathbf{t}'_{\text{pre}}$ *to a token sequence* $\mathbf{s} \in \Gamma^*$ *by the following procedure:*
   (a) replace 0 with $\{\times, x_1, x_2\}$ and 1 with $\{-, x_1, x_2\}$;
   (b) replace $\wedge$ with $\times$;
   (c) replace $\neg$ with $\{-, -, x_1, x_2\}$.
4. *Local edits:*
   (a) prepend $+$ to $\mathbf{s}$ to form the incomplete token sequence $\widetilde{\mathbf{s}}$;
   (b) set $n = 2$, and attach the data points $(x_{1,1}, x_{2,1}, y_1) = (1, 0, 1)$ and $(x_{1,2}, x_{2,2}, y_2) = (0, -1, 0)$ to the input numerical data $\mathcal{D}$;
   (c) define the metric as the mean squared error: $\mathcal{L}(\mathbf{y}, \hat{\mathbf{y}}) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$

To perform the first step, for all $\vee$ in $\mathbf{t}$, we must replace the nearest left bracket containing $\vee$ with $\neg(\neg$ and also replace $\vee$ with $\wedge\neg$. By using the results of Lemma G.4, it follows that this operation can be performed by a circuit within TC^0 complexity. The second step can be implemented by AC^0 circuits, according to Buss (1987, Cor. 11). Since the third and fourth steps only involve replacing and extending obtained sequences, these steps can also be implemented by AC^0 circuits.

Step 3 (soundness of the reduction). When $\mathrm{eval}(\mathbf{t}) = 0$, the losses $\mathcal{L}(\mathbf{y}, e_{\widetilde{(\mathbf{s},u)}}(\mathbf{x}))$ for each leaf token $u \in \{x_1, x_2, C\}$ can be computed as follows:

$$\begin{cases} \mathcal{L}(\mathbf{y}, e_{\widetilde{(\mathbf{s},x_1)}}(\mathbf{x})) = \frac{1}{2}\sum_{i=1}^{2}(y_i - x_{1,i})^2 = \frac{1}{2}(0^2 + 0^2) = 0, \\ \mathcal{L}(\mathbf{y}, e_{\widetilde{(\mathbf{s},x_2)}}(\mathbf{x})) = \frac{1}{2}\sum_{i=1}^{2}(y_i - x_{2,i})^2 = \frac{1}{2}(1^2 + 1^2) = 1, \\ \mathcal{L}(\mathbf{y}, e_{\widetilde{(\mathbf{s},C)}}(\mathbf{x})) = \operatorname*{argmin}_{c\in\mathcal{C}} \frac{1}{2}\sum_{i=1}^{2}(y_i - c)^2 = \operatorname*{argmin}_{c\in\mathcal{C}} \frac{1}{2}((1-c)^2 + (-c)^2) \geq \frac{1}{4}, \end{cases}$$

where $\mathcal{C}$ is the interval from which numeric constants are drawn, and $e_{\mathbf{s}} = \mathrm{expr}(\mathbf{s}, \mathcal{D})$ is the mapping function defined in Section 4. When $\mathrm{eval}(\mathbf{t}) = 1$, the losses $\mathcal{L}(\mathbf{y}, e_{\widetilde{(\mathbf{s},u)}}(\mathbf{x}))$ can be computed as follows:

$$\begin{cases} \mathcal{L}(\mathbf{y}, e_{\widetilde{(\mathbf{s},x_1)}}(\mathbf{x})) = \frac{1}{2}\sum_{i=1}^{2}(y_i - (1 + x_{1,i}))^2 = \frac{1}{2}((-1)^2 + (-1)^2) = 1, \\ \mathcal{L}(\mathbf{y}, e_{\widetilde{(\mathbf{s},x_2)}}(\mathbf{x})) = \frac{1}{2}\sum_{i=1}^{2}(y_i - (1 + x_{2,i}))^2 = \frac{1}{2}(0^2 + 0^2) = 0, \\ \mathcal{L}(\mathbf{y}, e_{\widetilde{(\mathbf{s},C)}}(\mathbf{x})) = \operatorname*{argmin}_{c\in\mathcal{C}} \frac{1}{2}\sum_{i=1}^{2}(y_i - (1 + c))^2 = \operatorname*{argmin}_{c\in\mathcal{C}} \frac{1}{2}((-c)^2 + (-1-c)^2) \geq \frac{1}{4}, \end{cases}$$

Consequently, when $\mathrm{eval}(\mathbf{t}) = 0$, the result for the corresponding last-token prediction problem is $u^* = x_1$, while when $\mathrm{eval}(\mathbf{t}) = 1$, the result is $u^* = x_2$. Hence the mapping introduced in Step 2 is a valid TC^0 many-one reduction from the Boolean formula value problem to the last-token prediction problem.

Step 4 (contradiction). The Boolean formula value problem is NC^1-complete under AC^0 reductions (Buss, 1987, Thm. 9). Hence Step 2 and Step 3 indicate that $\mathrm{LastTokenPrediction}(m) \notin \mathsf{TC}^0$, contradicting Step 1 and the assumed strict inclusion $\mathsf{TC}^0 \subsetneq \mathsf{NC}^1$. Therefore, such a Transformer cannot exist. □

G.3 PAC approximation via iterated self-verification

We further present theoretical analysis regarding the performance of the proposed method, NSR-gvs.

Assumption G.6. We make the following assumptions.

1. **Hypothesis class**. Fix a maximum depth D_0 and a grid spaced in $\varepsilon/2$ on $[-1, 1]$.

$$\mathcal{U} := \{e : \mathrm{depth}(e) \leq D_0\}, \quad U := |\mathcal{U}| \leq \mathrm{poly}(n),$$

   where $n := |\mathcal{D}|$.

2. **Data**. $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^n$ with $y_i \in [-1, 1]$ and $e^\star = \arg\min_{e\in\mathcal{U}} \mathrm{MSE}(e; \mathcal{D})$.

3. **Transformer**. A depth-L log-precision Transformer T (L constant).

4. **Exact oracle**. A routine $\mathcal{M}$ returns $\mathrm{MSE}(e; \mathcal{D})$ for any e.

5. **Hit rate**. If every subtree of $e^\star$ is present in the prompt, T outputs $e^\star$ with probability at least $\beta \in (0, 1]$.

6. **Dictionary growth**. Each round appends at least one *uniformly random* unseen subtree to the prompt (chosen without replacement; if fewer than r remain, insert all).

Theorem G.7 (informal)**.** *Let the algorithm cycle long enough for its prompt to have seen every possible sub-expression; then keep running a few more rounds. With very high probability, it returns a formula whose error is no worse than an optimally chosen tree by more than a tiny tolerance, and it has queried the oracle only a moderate, logarithmically growing number of times.*

Theorem G.8 (PAC guarantee)**.** *Run the loop*

$$e_t \leftarrow T(\textit{prompt}); \quad R_t \leftarrow \mathcal{M}(e_t); \quad \textit{prompt} \mathrel{+}= \textit{sub-trees}(e_t)$$

for a burn-in $B = \left\lceil \frac{U}{r} \ln\!\big(2^{D_0}/(\delta/2)\big) \right\rceil$ *rounds, followed by* $R = \left\lceil \frac{\ln(2/\delta)}{\beta} \right\rceil$ *additional rounds, and return the best-so-far expression* e_{best}*.*

Then, Under Assumption G.6, for any $\varepsilon, \delta \in (0,1)$*,*

$$\Pr\Big[\mathrm{MSE}\big(e_{\text{best}}, \mathcal{D}\big) \le \mathrm{MSE}\big(e^\star, \mathcal{D}\big) + \varepsilon\Big] \;\ge\; 1-\delta, \qquad \#\textit{oracle calls} \;=\; \mathcal{O}\big(U \ln(1/\delta)\big).$$

Proof. **(i) Burn-in.** There are $K \le 2^{D_0}$ distinct sub-trees of $e^\star$. Drawing $r \ge 1$ uniform sub-trees per round, the probability a fixed sub-tree is never drawn in B rounds is $(1-\frac{r}{U})^B \le e^{-rB/U} \le \delta/(2K)$. A union bound over all K sub-trees implies that, after B rounds, the prompt contains *every* sub-tree of $e^\star$ with probability at least $1-\delta/2$.

(ii) Post burn-in success. Condition on the burn-in success event. By assumption (3) each subsequent round now hits $e^\star$ with probability at least β, regardless of possible prompt changes. Therefore

$$\Pr[\text{miss in all } R \text{ rounds}] \;\le\; (1-\beta)^R \;\le\; e^{-\beta R} \;\le\; \delta/2,$$

for $R = \lceil \ln(2/\delta)/\beta \rceil$.

(iii) Union bound. Total failure probability $\le \delta/2 + \delta/2 = \delta$.

(iv) Quality of e_{best}. Whenever $e^\star$ appears, the exact oracle certifies its MSE; the algorithm stores it permanently. Hence on the complement of failure the returned expression meets the stated error bound.

(v) Oracle calls. At most one full-expression evaluation per round, so the algorithm issues $B + R = \mathcal{O}(U \ln(1/\delta))$ oracle calls.

□