Heidelberg University
Institute for Computer Science
Computer Vision and Learning Lab Heidelberg

Master's Thesis

⚡ANSR: Fast Amortized Neural Symbolic Regression

Name: Paul Stefan Saegert
Matriculation number: 4065845
Supervisor: Prof. Dr. rer. nat. Ullrich Köthe
Date of submission: 21st of April 2025

Declaration

I hereby certify that I have written the work myself and that I have not used any sources or aids other than those specified and that I have marked what has been taken over from other people's works, either verbatim or in terms of content, as foreign. I also certify that the electronic version of my thesis transmitted completely corresponds in content and wording to the printed version. I agree that this electronic version is being checked for plagiarism at the university using plagiarism software.

_____

Abstract

Symbolic Regression has been approached with many different methods and paradigms. The overwhelming success of transformer-based language models in recent years has since motivated researchers to solve Symbolic Regression with large-scale pre-training of data-conditioned "equation generators" at competitive levels. However, as most traditional methods, the majority of these Amortized Neural Symbolic Regression methods rely on SymPy to simplify and compile randomly generated training equations, a choice that inevitably brings tradeoffs and requires workarounds to efficiently work at scale. I show that replacing SymPy with a novel token-based simplification algorithm with hand-crafted transformation rules enables training on *fully-procedurally* generated and *higher-quality* synthetic data, and thus develop ⚡ANSR. On various test sets, my method perfectly recovers +80% more equations numerically than the NeSymReS baseline while being 84 times faster natively, and yields comparable recovery rates to PySR in a quarter of its time. I provide an in-depth performance analysis of my method on stricter and more meaningful metrics than previous work. ⚡ANSR is open-source and available on GitHub and Huggingface, and allows for straight-forward replicability on consumer-grade hardware.

Abstract

Die symbolische Regression wurde mit vielen verschiedenen Methoden und Paradigmen angegangen. Der überwältigende Erfolg von Transformer-basierten Sprachmodellen in den letzten Jahren hat Forscher dazu motiviert, die symbolische Regression mit umfangreichem Vortraining von datengesteuerten "Gleichungsgeneratoren" auf konkurrenzfähigem Niveau zu lösen. Wie die meisten traditionellen Methoden stützt sich jedoch die Mehrheit dieser Amortisierten Neuronalen Symbolischen Regressionssmethoden auf SymPy, um zufällig generierte Trainingsgleichungen zu vereinfachen und zu kompilieren. Diese Wahl bringt unweigerlich Kompromisse mit sich und erfordert Workarounds, um effizient in großem Maßstab zu arbeiten. Ich zeige, dass der Ersatz von SymPy durch einen neuartigen Token-basierten Vereinfachungsalgorithmus mit handgefertigten Transformationsregeln das Training mit *vollständig prozedural* generierten und *hochwertigeren* synthetischen Daten ermöglicht und entwickle so ⚡ANSR. Auf verschiedenen Testdatensätzen stellt meine Methode +80% mehr Gleichungen numerisch perfekt wieder her als die NeSymReS-Baseline, ist dabei nativ 84-mal schneller, und erzielt vergleichbare Wiederherstellungsraten wie PySR in einem Viertel der Zeit. Ich liefere eine detaillierte Leistungsanalyse meiner Methode anhand strengerer und aussagekräftigerer Metriken als bisherige Arbeiten. ⚡ANSR ist Open-Source und auf GitHub und Huggingface verfügbar und ermöglicht unkomplizierte Replizierbarkeit auf Consumer-Hardware.

Contents

1 Introduction

Symbolic Regression (SR) is the task of finding expressive, interpretable symbolic models in the form of mathematical expressions that accurately describe relations between variables in a given dataset, and has been proven to be NP-hard (Virgolin and Pissis 2022). Still, various state-of-the-art methods provide sufficient accuracy and usability for uses in scientific applications such as model discovery (Cranmer 2024) or the identification of effective surrogate models for simulation (Champion et al. 2019).

Early methods follow an instance-based approach where an SR model is fit to new data from scratch via Evolutionary Algorithms (Burlacu et al. 2020; Cranmer 2023), Reinforcement Learning (Landajuela et al. 2022; 2021), or Ensemble Learning (Udrescu and Tegmark 2020) among others. Recently, however, methods have been proposed which leverage knowledge gained by large-scale pre-training on a diverse set of data and associated equations. These "Amortized Neural Symbolic Regression" (ANSR) methods are intended to reduce user costs by making the prediction of the equation merely a problem of informed inference rather than training from scratch for each new dataset.

In this work, I improve on the previous state-of-the-art baseline *"Neural Symbolic Regression That Scales"* (NeSymReS) (Biggio et al. 2021) and thus develop ⚡ANSR (pronounced: "Flash-Answer"). I it compare to the original work NeSymReS (Biggio et al. 2021) and the state-of-the-art PySR (Cranmer 2023) method on a variety of test sets and metrics.

2 Related Work

Recent works explore a number of different paradigms to solve Symbolic Regression.

Evolutionary Algorithms

PySR (Cranmer 2023) and Operon (Burlacu et al. 2020) iteratively refine expressions through evolution: The best expressions in a randomly subsampled population are selected to derive new expressions via random mutation, combinations of two expressions ("crossover") or simplification.

Reinforcement Learning

Deep Symbolic Regression (DSR) (Petersen et al. 2021) formulates the search for expressions as learning a policy that, when converged, would autoregressively generate the expression by following a series of next-symbol-selection actions. Deep Symbolic Optimization (DSO) (Landajuela et al. 2021) combines DSR with Genetic Programming (Fortin et al. 2012).

Amortized Neural Symbolic Regression

As one of the fundamental works in the field, NeSymReS (Biggio et al. 2021) employs a Set-Transformer (Li et al. 2022) to encode data represented by a set of points, and a

Transformer decoder (Vaswani et al. 2023) to generate symbolic expressions in the prefix form. The expressions used for pre-training are generated with an algorithm proposed by Lample and Charton (2019) and simplified with SymPy (Meurer et al. 2017), a trend that persists throughout most other works.

Another similar method called SymbolicGPT (Valipour et al. 2021) uses a T-net (Qi et al. 2017) encoder and the GPT architecture (Brown et al. 2020; Radford et al. 2018; 2019) for decoding. The ODEFormer (d'Ascoli et al. 2023) applies large-scale pre-training on ordinary differential equations and simulated trajectories. They also encode and predict the values of constants through combinations of special sign, mantissa, and exponent tokens to obtain coarse initial guesses for further numerical optimization. Symformer (Vastl et al. 2024) shares the same architecture with NeSymReS (Biggio et al. 2021). Their main contribution is a more expressive encoding and prediction of constants than is provided by the encoding scheme proposed in the ODEFormer (d'Ascoli et al. 2023). Neural Symbolic Regression with Hypothesis (NSRwH) (Bendinelli et al. 2023) introduces a method that allows for the control over properties like complexity of predicted expressions or required or disallowed terms at test time. A recent work by Li et al. (2024) uses DSO to generate training data for pre-training.

Miscellaneous

Landajuela et al. (2022) also propose uDSR, a "unified" approach to Symbolic Regression realized with an ensemble of state-of-the-art methods. Another ensemble method is AI Feynman (Udrescu and Tegmark 2020; Udrescu et al. 2020) which recursively exploits gradient properties such as symmetry and separability among others to reconstruct expressions.

Learning, pruning, and symbolically identifying activation functions as in the recently proposed Kolmogorov-Arnold-Network (KAN) (Liu et al. 2024a; 2024b) also appears to facilitate Symbolic Regression.

Symbolic Regression has also been demonstrated to work in scenarios where even the optimal coordinates of an equation are unknown and learned with the SINDy-Autoencoder (Brunton et al. 2016; Champion et al. 2019; Saegert 2022).

3 Methodology

In the following sections, I explain the underlying model, its training data and procedure, inference and evaluation, and how they differ from NeSymReS (Biggio et al. 2021).

3.1 Model

I mostly adopt the architecture proposed by Biggio et al. (2021), featuring a Set-Transformer Encoder (Lee et al. 2019) and a Transformer decoder (Vaswani et al. 2023) (Figure 1).

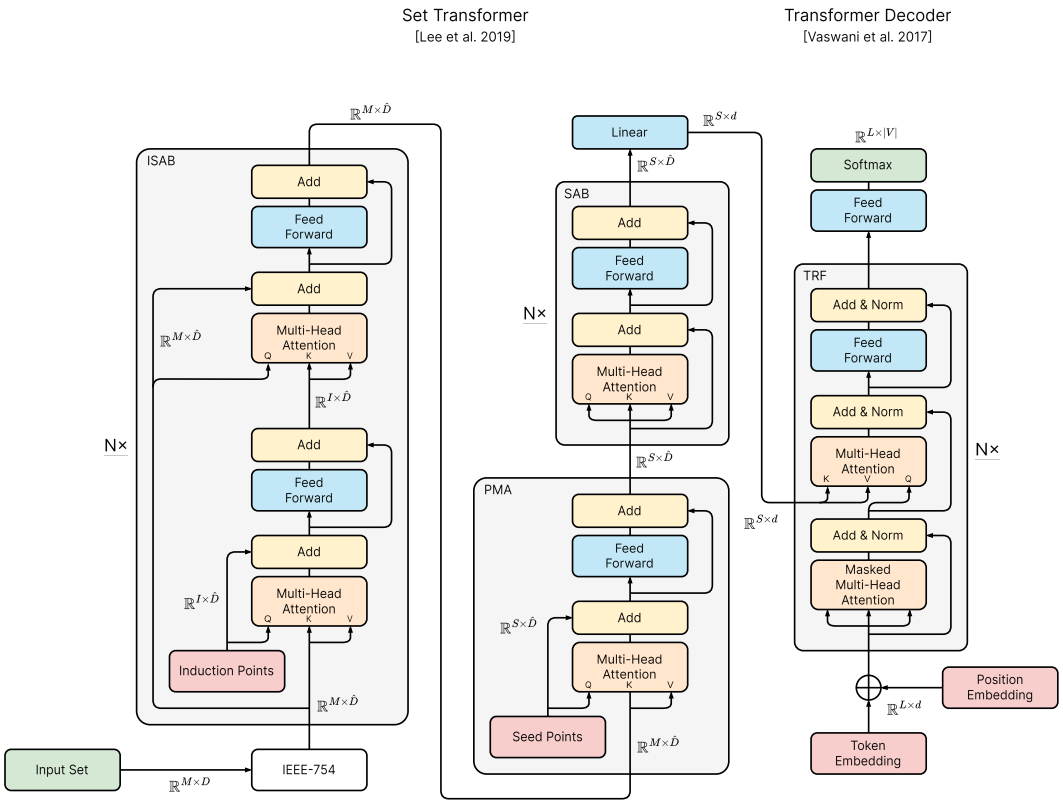

Figure 1: **NeSymReS (Biggio et al. 2021) Architecture used for ⚡ANSR.** An IEE-754 encoded input set is encoded with a Set-Transformer (Lee et al. 2019) (*left and middle column*). This encoding is then used to condition a Transformer decoder (Vaswani et al. 2023), which outputs next-token probabilities.

Biggio et al. motivate the use of the Set-Transformer referencing its comparatively low runtime complexity $\mathcal{O}(mn)$ with $m \ll n$ compared to $\mathcal{O}(n^2)$ for regular self-attention based Transformers, where n is the number of input elements and m a hyperparameter for the Induction Self Attention Blocks (ISAB)[1]. I apply three changes to the architecture:

- I disable the Layer-Norm (Ba et al. 2016) in the Multi-Head Attention Blocks of the Set-Transformer since it appeared to degrade performance in small-scale pilot experiments, which aligns with Zhang et al. (2022) who found that the Set-Transformer can indeed perform worse with Layer-Norm than without.
- The number of induction points I is increased from 50 to 64. The number of pooling seeds S is increased from 10 to 64, effectively providing more dimensions for encoding.
- In the decoder, I set the Dropout probability (Srivastava et al. 2014) to 0.1 for good measure whereas Biggio et al. (2021) do not use Dropout.

[1] $\mathcal{O}(IM)$ and $\mathcal{O}(M^2)$ in Figure 1

In total, this variation of the NeSymReS model has about 27M trainable parameters split into 13M encoder parameters (+2M from Biggio et al. (2021)), 13M decoder parameters, and additional token embedding and head parameters.

3.2 Vocabulary

NeSymReS (Biggio et al. 2021) uses a vocabulary of 15 operators, three variables, and various special and control tokens. I extend this vocabulary by dedicated additive and multiplicative inverse operators `<neg>` and `<inv>`, the absolute operator `<abs>`, and inverses matching the existing power and trigonometric operators. Additionally, I remove tokens representing specific constant values such as `-1`, `0` or `5`.

Index	Token	Description	Index	Token	Description
0	`<pad>`	Padding	17	`pow5`	x^5
1	`<bos>`	Beginning of Sentence	18	`pow1_2`	$\sqrt{x}$
2	`<eos>`	End of Sentence	19	`pow1_3`	$\sqrt[3]{x}$
3	`<unk>`	Unknown*	20	`pow1_4`	$\sqrt[4]{x}$
4	`<cls>`	Classification*	21	`pow1_5`	$\sqrt[5]{x}$
5	`<mask>`	Masking*	22	`sin`	$\sin(x)$
6	`<num>`	Constant	23	`cos`	$\cos(x)$
7	`+`	Addition	24	`tan`	$\tan(x)$
8	`-`	Subtraction	25	`asin`	$\arcsin(x)$
9	`neg`	$-x$	26	`acos`	$\arccos(x)$
10	`*`	Multiplication	27	`atan`	$\arctan(x)$
11	`/`	Division	28	`exp`	e^x
12	`abs`	$\lvert x \rvert$	29	`log`	$\ln(x)$
13	`inv`	$\frac{1}{x}$	30	`x1`	x_1
14	`pow2`	x^2	31	`x2`	x_2
15	`pow3`	x^3	32	`x3`	x_3
16	`pow4`	x^4			

Table 1: **Vocabulary and Token Semantics.** The reduced NeSymReS (Biggio et al. 2021) vocabulary (*gray tokens*) is extended by additional operators (*green tokens*). *Unused.

These changes allow for a more concise representation of certain equations: For example, the term $-x_1$ can thus be represented by merely two tokens `[ <neg> , x1 ]` instead of the three tokens `[ * , -1 , x1 ]`, saving compute and improving clarity. Furthermore, the added

inverse operators facilitate the creation of simplification rules (Section 3.3) and improve the theoretical identification capabilities of the method.

The vocabulary used in this work is presented in Table 1.

3.3 Data

Arguably one of the most important aspects of current Amortized Neural Symbolic Regression methods is the data used for its large scale pre-training. In this section, I describe the generation and pre-processing of the synthetic training and validation data.

Inspired by the consensus in the field, I first define the following:

- **Expression** $Z^{\boldsymbol{c}}$: Sequence of tokens that represents an equation in prefix notation including values for any constants $\boldsymbol{c}$, e.g. $Z^{\boldsymbol{c}} =$ `[ * , 1.23 , x1 ]` for the equation $y = 1.23 \cdot x_1$ with constants $\boldsymbol{c} = [1.23] \in \mathbb{R}^1$.
- **Skeleton** Z: Expression with placeholders for numeric constants, e.g. $Z =$ `[ * , <num> , x1 ]` for the equation $y = c_1 \cdot x_1$.
- **Function** $f_Z(\,\cdot\,;\boldsymbol{c})$: Compiled and executable form of a skeleton Z that takes a point $\boldsymbol{x}$ and constants $\boldsymbol{c}$ as input and outputs y.
- **Input Set** $X = \{\boldsymbol{x}_m\}_{m=1}^{M}$: Set of points at which a function is evaluated.
- **Output Set** $Y = \{y_m\}$: Set of values $y_m = f_Z(\boldsymbol{x}_m;\boldsymbol{c})$ for a given vector of constants $\boldsymbol{c}$.

Training data consists of pairs of input-output sets and their corresponding target skeleton.

3.3.1 Synthetic Data Generation

For the task of predicting a skeleton Z_i from a set of input-output pairs, training data is conceptually easy to generate in the opposite direction, in particular by sampling "random" skeletons Z_i and evaluating the function $f_{Z_i}(\,\cdot\,;\boldsymbol{c}_i)$ with some $\boldsymbol{c}_i$ at M randomly sampled points $X_i = \{\boldsymbol{x}_m\}_{m=1}^{M}$ to get $y_m = f_{Z_i}(\boldsymbol{x}_m;\boldsymbol{c}_i)$.

Following Biggio et al. (2021), I use the "Lample-Charton" algorithm (Lample and Charton 2019), denoted LC in the following, to randomly generate skeletons from the operators, variables, and the placeholder for constants listed in Table 1. The probability weights of adopted operators are the same as in (Biggio et al. 2021). Newly added operators follow the same weighting scheme to ensure consistency.

Since the Lample-Charton algorithm requires a pre-defined number of operators ought to be present in the generated expression, one starts by first sampling the number of operators N_{ops} from a specified distribution. While Biggio et al. (2021) sample from a uniform distribution $N_{\text{ops}} \sim \mathcal{U}(3,5)$, I instead sample from a positive exponent power distribution

$$N_{\text{ops}} \sim \mathcal{P}^{+}\big(N_{\text{ops}};[l,h],\beta\big) \propto N_{\text{ops}}^{\beta} \tag{1}$$

with lower and upper limits $l = 0$ and $h = 10$, and exponent $\beta = 1$. This effectively shifts the length distribution of the training data towards longer and thus more complex expressions and slightly counteracts the log-uniform nature of their approach to expression generation.

Then, given N_{ops}, a skeleton

$$Z \sim \text{LC}\big(N_{\text{ops}}, O_1, O_2, w_{O_1}, w_{O_2}, p_{\text{var}}\big) \tag{2}$$

is generated, where O_1 and O_2 are the sets of allowed unary and binary operators, w_{O_1} and w_{O_2} are the sets of weights (i.e. unnormalized probabilities) of each unary and binary operator, and p_{var} represents the probability of an elementary operand being a variable as opposed to a constant.

With three variables `x1` , `x2` , `x3` , I set $p_{\text{var}} = \frac{3}{4}$, meaning that an operand is equally likely to be a constant than any one variable. This provides an intuitive motivation and differs from NeSymReS (Biggio et al. 2021), who arbitrarily set $p_{\text{var}} = 0.8$.

Once a raw skeleton has been generated, it is then algebraically simplified into a more concise form (Section 3.3.2). Considering that the model should ideally generate expressions that explain the data as well as possible while only using as few operators as necessary, this arguably improves the quality of the training data and encourages the model to predict the least complex form of a skeleton, e.g. `[ x1 ]` instead of `[ ln , exp , x1 ]`, since $\ln(e^{x_1}) \equiv x_1$.

The generation of an expression is completed by sampling numerical values for each constant in the expression from the uniform distribution

$$\boldsymbol{c} \sim \mathcal{U}(-5, 5)^K \tag{3}$$

where K is the number of constants in the expression. I do not distinguish between additive and multiplicative constants as opposed to Biggio et al. (2021) and thus simplify this step.

Before the input and output sets form the expression can be generated, the number of support points

$$M \sim \mathcal{U}(16, 512) \tag{4}$$

and the lower and upper bounds of the support interval in each dimension $j = 1, ..., D$

$$\boldsymbol{a}, \boldsymbol{b} \sim \mathcal{U}(-10, 10)^D \tag{5}$$

are sampled. These are then used to sample the input set X via

$$X_{:j} \sim \mathcal{U}\big(a_j, b_j\big)^M \tag{6}$$

and by extension enable the calculation of

$$Y = f_Z(X; \boldsymbol{c}). \tag{7}$$

A visual summary of this process is shown later in Figure 5.

Training data is generated and decontaminated fully procedurally, including the compilation of expressions into executable code. Contamination of training data with test data is partially mitigated by rejecting training instances whose expression matches any test expressions symbolically. Additionally, the image $Y^* = f_{Z^*}(X^-; \boldsymbol{c}^-)$ of newly generated training expressions Z^* on a fixed random input set X^- with fixed constants $\boldsymbol{c}^-$ is compared

to the images of test on the same input set $\mathcal{Y}^- = \{f_Z(X^-; c^-) \mid Z \text{ excluded from training}\}$ and rejected if $Y^* \in \mathcal{Y}^-$.

Since this procedure is susceptible to the order in which the fixed constants appear in the expression, some functionally equivalent training expressions where constants appear in different orders may not be rejected. As described later in Section 3.3.2, this problem of non-unique symbolic representations for functionally equivalent expressions is partially mitigated by simplifying and sorting terms but still occurs to some degree.

For future experiments, I implement a more conservative procedure where expressions are stripped of their constants before being applied on X^-, yielding a set of images $\mathcal{Y}^- = \left\{f_{\text{strip}(Z)}(X^-) \mid Z \text{ excluded from training}\right\}$. Then, the image of the stripped form $Y^* = f_{\text{strip}(Z^*)}(X^-)$ of new expressions Z^* is compared against $\mathcal{Y}^-$ and the instance rejected if $Y^* \in \mathcal{Y}^-$ which aligns better with Biggio et al. (2021) who set additive constants to 0 and multiplicative constants to 1 for comparison.

While most experiments are performed with the former procedure, the latter is arguably the better choice for preventing information leakage, albeit at the cost of leaving out entire families of functions during training. Both approaches result in the same or similar test metrics (Appendix H).

Lastly, instances containing any NaNs in either X or Y are fully rejected.

3.3.2 Simplification & Pre-Processing

Many recent works rely on SymPy (Meurer et al. 2017) to handle the simplification of randomly generated expressions. However, I do not consider it a good option for the simplification of expressions when training Symbolic Regression methods at scale.

In the time I used SymPy during early development, I noticed a steady decrease in available system memory (48GB) during training that eventually resulted in out-of-memory crashes. After systematically isolating likely causes for this phenomenon, I narrowed the problem down to the `sympy.lamdify` method, which appeared to reserve memory that would not be freed when the lambda function it produced was deleted. I reported this memory leak to the SymPy contributors on GitHub which quickly sparked contributions to SymPy, PyPy, and CPython (!)[2].

While this issue may eventually be fixed in future releases, SymPy still represents a bottleneck for large scale training due to its comparatively slow simplification procedure (Section 5.1).

Additionally, I noticed a pattern of SymPy "simplifications" which I found unsuitable and which could not be adequately controlled. For example, SymPy would prefer writing $-A + B$ over $B - A$, which, when tokenized requires an additional token (`[ + , neg , A , B ]` for "$-A + B$" vs `[ - , B , A ]` for "$B - A$"). Moreover, I was dissatisfied with the complexity and

[2] https://github.com/sympy/sympy/issues/27216#issuecomment-2484124772

overhead associated with the conversion from tokenized expressions to SymPy equations and back.

Hence, I propose a fast, token-based simplification algorithm that recursively matches and transforms subtrees of an expression tree based on a rule set consisting of 65 hand-crafted rules (Algorithm 1, Appendix G).

```
FLASH-ANSR SIMPLIFICATION(
    Z: list[str] ▷ Tokenized expression ,
    T: int ▷ Maximum number of iterations ,
    O: set[str] ▷ Set of operators ,
    R: set ▷ Set of simplification rules
):
    Z_prev ← Z
    for m in 1..T: ▷ T masking iterations
        for n in 1..T: ▷ T simplify iterations
            S ← [ ] ▷ Initialize empty tack
            for z_i in reversed(Z):
                if z_i ∉ O:
                    ▷ Token is an elementary operand, skip
                    S.push(z_i)
                    continue
                ▷ Token is an operator, attempt simplification
                let A(z_i) be the arity of operator z_i
                S' ← [S.pop() for _ in A(z_i)] ▷ Pop term from stack
                if S' matches any rule r ∈ R:
                    S' ← r(S') ▷ Simplify term
                S.push(S') ▷ Push Simplified term onto stack
            Z ← flatten(S) ▷ Transform stack into flat prefix form
        ▷ Mask leftover special numeric operands
        Z ← Z.replace(<0>, <num>)
        Z ← Z.replace(<1>, <num>)
        ▷ Stop early when expression cannot be further simplified
        if Z == Z_prev:
            break
        Z_prev ← Z
    return Z_new
```

Algorithm 1: **Rule-Based Simplification Algorithm for ⚡ANSR.** A comprehensive list of simplification rules can be found in Appendix G.

Since Algorithm 1 operates on the same tokens used by the transformer decoder, it does not require any conversion layer between the tokenized and the "parsed" infix form of an expression. Furthermore, its small size and use of elementary data structures make it orders of magnitude faster than SymPy (Section 5.1). This advantage is really what makes the fully procedural generation of training data described in Section 3.3.1 feasible in the first place, and avoids the need for the workarounds seen in NeSymReS (Biggio et al. 2021), who resort to a "*partially pre-generated dataset [...] to speed up the generation of training data for the mini-batches*" (Biggio et al. 2021).

3.3.3 Data Analysis

As a validation of the described data generation method, I perform a brief data analysis of 100,000 generated instances. Starting with Figure 2, I display histograms of various properties of the training data.

Both the number of support points per instance and the values of sampled constants follow their respective target distributions $M \sim \mathcal{U}(16, 512)$ and $c_k \sim \mathcal{U}(-5, 5)$ with p-values of 0.08 and 0.73 by the Kolmogorov-Smirnov test as implemented in `scipy.stats.kstest`.

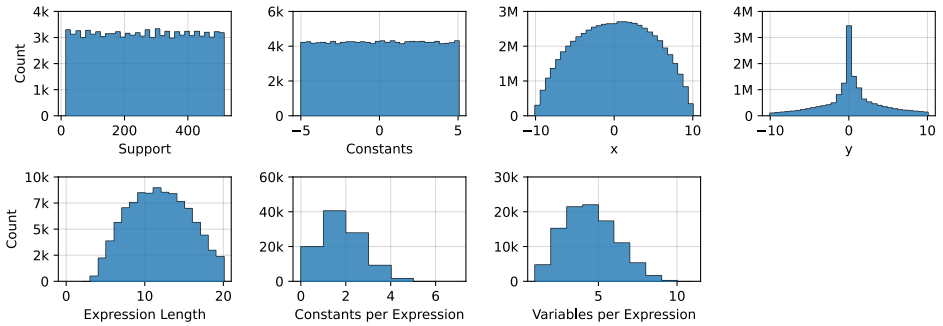

Figure 2: **Distributions of the Training Data.**
The input-set histogram includes data from all dimensions of x.

Although x is sampled from a symmetric uniform distribution (Equation 5, 6), the resulting training distribution is neither uniform not symmetric ($p = 0.0$), slightly but significantly favoring positive over negative values. This is likely an artifact of filtering out instances where $f_Z(X; \boldsymbol{c})$ contains NaNs when the function is undefined for $x < 0$ or $x \leq 0$ such as $y = \sqrt{x}$ or $y = \log(x)$. While a negation of the input also negates this argument, it is conceivable that the cost of an additional `neg` token combined with sampling within a limited number of operators may uphold this asymmetry.

Interestingly, the images of the training expressions overwhelmingly fall in the range of -1 to 1, again with a slight but significant favor for positive values ($p = 0.0$).

The distribution of expression lengths strongly deviates from the positive exponent power distribution (Equation 1) used to sample the number of operators, where one would expect

a linear increase in the number of expressions with increasing length. However, for one, this is greatly influenced by the rejection of instances containing NaNs: The more operators and operands there are in an expression, the higher the chances that at least one combination of them is not defined at one of the points in X. Furthermore, the simplification of expressions shifts the unsimplified distribution towards shorter expressions.

While the target distributions for the number of constants and variables per expression is nontrivial, their empirical histograms show no peculiarities and thus serve as a sanity check.

Since the Algorithm 1 ultimately adds and removes tokens to and from a sampled expression it is important to quantify this distribution shift. I therefore compare the normalized probability weights of operators that serve as inputs to the Lample-Charton Algorithm (Lample and Charton 2019) to the actual frequency of operators after simplification in Figure 3.

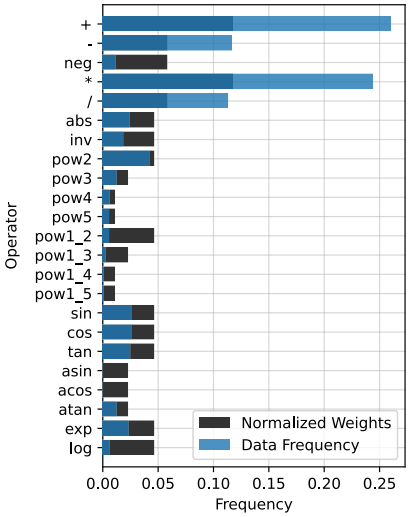

Figure 3: **Operator Weights and Frequencies in the Training Data.**

Simplified expressions contain about twice as many `+` , `-` , `*` , and `/` operators as specified by the weights, and correspondingly fewer other operators. The vanishingly small frequencies of operators like `asin` , `log` and `pow1_2` can be explained by domain issues where the randomly sampled input set contains points where the associated functions are not defined, resulting in NaNs in the output set and thus rejection of the instance.

The decrease in the frequency of `neg` tokens highlights the effectiveness of the simplification rules (Appendix G) which, for example, try to absorb `neg` tokens into already existing additions and subtractions.

Finally, I visualize all 1-D instances by superimposing their (x, y) coordinates on a logarithmic histogram (Figure 4). Despite the randomness associated with the data generation, the histogram shows a surprising amount of structure. A dense band around -1 to 1 and

the (off-)diagonal as well as the origin are covered by more samples than other areas. This could be due to frequent occurrences of $\sin(\cdot)$ and $\cos(\cdot)$ and particularly simple equations like $y = x_1$.

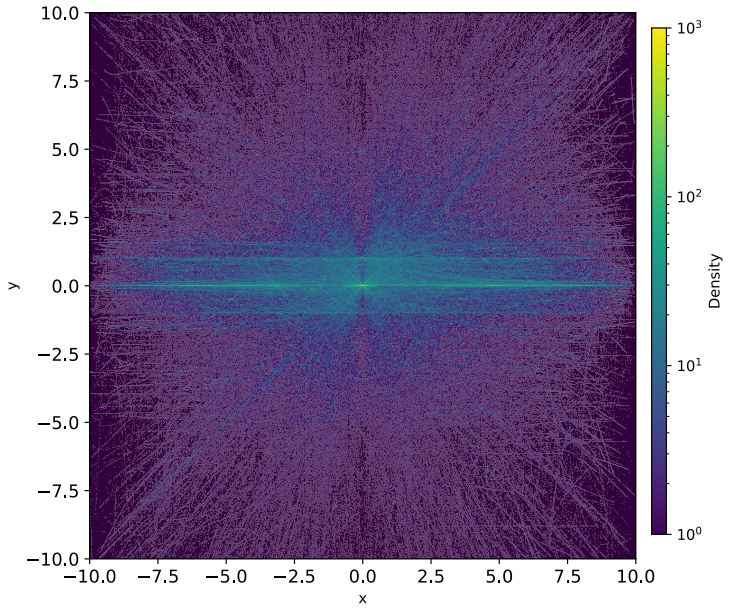

Figure 4: **Superimposed 1-D Instances from the Training Distribution.**

3.4 Training

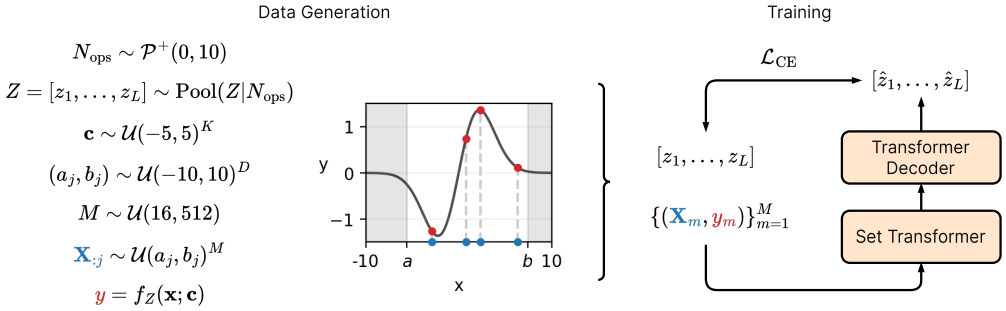

Figure 5: ⚡**ANSR Training on Fully Procedurally Generated Data.** Inspired by NeSymReS (Biggio et al. 2021).

Given synthetic data pairs containing skeletons Z_i and sampled points (X_i, Y_i) generated as in Section 3.3.1, the model is trained to maximize the next-token-probability conditioned on the set of input and output points $\mathcal{S} = \{(\boldsymbol{X}_i, y_i)\}_{i=1}^M$ and previous tokens $z_{<l}$ of the target expression in prefix notation. Formally, the cross-entropy loss for instances i of expression length L,

$$\mathcal{L}_{\text{CE}}(Z_i, X_i, Y_i) = -\sum_{l=1}^{L} \log p_\theta\big(z_{i,l} \mid z_{i,<l}\ ; X_i, Y_i\big), \tag{8}$$

is used as the training objective to obtain the optimal model parameters $\hat{\theta}$:

$$\hat{\theta} = \arg\min_\theta \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}_{\text{CE}}(Z_i, X_i, Y_i). \tag{9}$$

With the exception of the fully procedurally generated data in my work, this scheme is identical to NeSymReS (Biggio et al. 2021). A visual overview of the training process is given above in Figure 5.

Inspired by common practice in Language Modelling and recent work by Kalra and Barkeshli (2024), I introduce a learning rate schedule including warmup and annealing. Additionally, I replace the Adam optimizer (Kingma and Ba 2017) used in NeSymReS (Biggio et al. 2021) with AdamW (Loshchilov and Hutter 2019) in its "amsgrad" variant (Reddi et al. 2018). Table 2 provides an overview of all relevant training parameters.

Parameter	⚡**ANSR**	**NeSymReS (Biggio et al. 2021)**
Total Steps	1.5M	1.5M
Batch Size	128	150
Learning Rate	Linear Warmup 0@0 $\rightarrow$ 10^{-4}@10k Linear Annealing 10^{-4}@10k$\rightarrow$ 0@1.5M	10^{-4}
Optimizer	AdamW (Loshchilov and Hutter 2019)	Adam (Kingma and Ba 2017)
Weight Decay	0.01	-

Table 2: **Training Parameters.**

During training the model thus sees 192 M instances amounting to about 1.3 TB of procedurally generated and sampled data and 31 GB of associated expressions.

3.5 Inference

Given a set of points $\mathcal{S} = \{(\boldsymbol{X}_i, y_i)\}_{i=1}^{M}$, equations are obtained in two steps:

Equivalence-Pruned Beam Search (EPBS)

First, candidate skeletons $\{\hat{Z}_k\}_{k=1}^{w}$ are generated by conditioning the transformer decoder on the Set-Transformer embeddings $\text{SetTRF}(\mathcal{S})$, and running beam search with a beam width w. Beam search begins with the `<bos>` token, and beams are considered complete when they reach the `<eos>` token. The positions of constants are predicted with the `<num>` token.

Since symbolically distinct beams can still represent skeletons that are functionally equivalent (e.g. `[ pow2 , x1 ]` and `[ * , x1 , x1 ]`), I employ an "equivalence-pruning" after every beam search iteration which simplifies completed beams, and discards them if their simplified form matches any already completed beam (Algorithm 3 in Appendix F).

The next-token-prediction tasks during beam search are mini-batched into a configurable amount of simultaneous forward passes per mini-batch (default: 128) . This enables efficient GPU usage while limiting memory requirements for large beam widths w or sequence lengths L.

Parsimony

Inspired by Biggio et al. (2021), I also add a "parsimony" parameter γ which penalizes the length of generated expressions. However, instead of penalizing the fit by adding the parsimony penalty to the MSE for each token, i.e.

$$\text{score}_k^{\text{abs}} = \text{MSE}_k + \gamma L_k, \tag{10}$$

I add the parsimony regularization to the logarithm of the Fraction of Variance Unexplained (FVU, defined in Equation 20):

$$\text{score}_k^{\text{log-rel}} = \log_{10}(\text{FVU}_k) + \gamma L_k \tag{11}$$

and sort by increasing $\text{score}^{\text{log-rel}}$.

Unless mentioned otherwise, I set $\gamma = 0$ in all experiments to get an unaltered impression of the raw method.

Constant Fitting

Once the specified number of beams have been generated, each beam is compiled into an executable function, and numeric placeholder tokens `<num>` are replaced with constants, e.g.

$$\hat{Z} = \texttt{[*, <num>, x1]} \rightarrow f_{\hat{Z}}(x_1, c_1) = c_1 \cdot x_1. \tag{12}$$

The vector of constants $\boldsymbol{c} = [c_1, ..., c_k]$ of a function $f_{\hat{Z}}(\boldsymbol{X}; \boldsymbol{c})$ is fitted via least squares

$$\hat{\boldsymbol{c}} = \underset{\boldsymbol{c}}{\text{argmin}} \sum_{i=1}^{M} \left(y_i - f_{\hat{Z}}(\boldsymbol{X}_i; \boldsymbol{c})\right)^2. \tag{13}$$

Many SOTA methods (Biggio et al. 2021; d'Ascoli et al. 2023; Petersen et al. 2021; Valipour et al. 2021) use the general purpose BFGS Algorithm (Broyden 1970) for this task. I instead opt for the Levenberg-Marquardt Algorithm (Levenberg 1944; Marquardt 1963) in SciPy (Virtanen et al. 2020) due to its specialization on least squares problems.

3.6 Evaluation

For evaluation and model comparison, I compute and average several instance-level metrics across four test sets with different characteristics. In the main evaluation and ablation study, 5000 random instances $(Z^{\boldsymbol{c}}, X, Y)$ are sampled as in Section 3.3.1 for each test set. For my

evaluation of PySR (Cranmer 2023) and NeSymReS (Biggio et al. 2021), I sample 1000 instances to limit the computational cost.

Each instance contains two input sets of $M = 512$ points. One set is used by the methods to perform Symbolic Regression while the other one is used as an instance-level validation set, resulting in fit metrics and validation metrics. Dropout layers (Srivastava et al. 2014) are disabled at test time.

In the following, I provide details about the test sets and metrics.

3.6.1 Test Sets

- **Feynman**: The *Feynman Symbolic Regression Database* compiled by (Udrescu and Tegmark 2020)[3] contains 120 equations from the *Feynman Lectures of Physics* Book Series. Only the subset of 43 equations with up to 3 input variables is used. E.g.

$$y = \frac{x_1}{\sqrt{1 - \frac{x_2^2}{x_3^2}}} \tag{14}$$

- **SOOSE-NC**: *Strictly Out-Of-Sample Equations without Constants* consisting of 200 randomly sampled equations by Biggio et al. (2021)[4] using the Lample-Charton Algorithm (Lample and Charton 2019). E.g.

$$y = \cos\left(\frac{\cos(x_2)^2}{(-x_1 + x_3)^2}\right) \tag{15}$$

- **Nguyen**: 12 equations without constants compiled from (Hoai et al. 2002; Johnson 2009; Keijzer 2003) by (Uy et al. 2011) and used in (Petersen et al. 2021). E.g.

$$y = x_1^6 + x_1^5 + x_1^4 + x_1^3 + x_1^2 + x_1 \tag{16}$$

- ⚡**ANSR-Pool-15**: A hard set comprised of 5000 approximately unique and randomly sampled and simplified equations with up to 15 operators and up to 3 variables using the LC Algorithm (Lample and Charton 2019), skewed to longer equations by setting $\beta = 2$ (Section 3.3), and constants sampled from $\mathcal{U}(-10, 10)$. E.g.

$$x_3 - x_2 \cdot \left(x_1 + \frac{c_1 \cdot (x_3)^{\frac{1}{5}}}{c_2 - x_3 + x_3^2 - x_2}\right) \tag{17}$$

All test expressions are simplified and approximately canonicalized (i.e. transformed into a unique form) with Algorithm 1 before their use in the evaluation, especially symbolic comparison. All test expressions are also excluded from the training data which differs from NeSymReS (Biggio et al. 2021) who do not exclude the Feynman Database from training nor comment on the contamination of their training data with the Nguyen dataset.

[3] Available at https://space.mit.edu/home/tegmark/aifeynman.html at the time of writing

[4] Available at https://github.com/SymposiumOrganization/NeuralSymbolicRegressionThatScales/blob/main/test_set/nc.csv at the time of writing

They also argue that the Nguyen dataset "*caps the maximum accuracy*" since it has "*[…] terms that appear in three ground truth equations that are not included in the set of equations that our model can fit, specifically* x^6*, and* x^y *[…]*" (Biggio et al. 2021). However, the term $x^6 = (x^3)^2$ could easily be constructed by the tokens `[ pow2 , pow3 , x ]`, and, while there is indeed no general power operator in the vocabulary (Table 1), the term $x^y = \exp[\ln(x^y)] = \exp[y \cdot \ln(x)]$ can still be represented as `[ exp , * , y , log , x ]` using available tokens, and can therefore be identified by the model, actually allowing it to achieve perfect *numeric* accuracy on this dataset in theory.

Furthermore, the added application of my parsing and approximate canonicalization methods on the test expressions transform the ground truth symbolic forms into compatible tokenized forms if possible[5], which therefore allows for near perfect *symbolic* accuracy too.

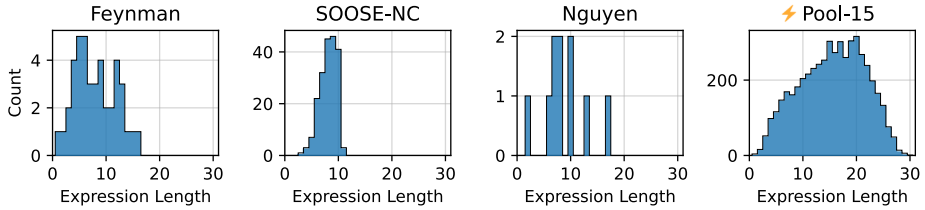

Figure 6: **Length Distribution of Expressions in the Test Sets.**

3.6.2 Metrics

I denote the general objective with ↑ for maximization and ↓ for minimization, and the theoretical bounds in squared brackets.

Biggio et al. (2021) rely on two definitions of "accuracy". The pointwise accuracy

$$A_1 \uparrow^{[0,1]} = \frac{1}{N} \sum_{i=0}^{N} \mathbf{1}[\forall m \in [1, M] : \texttt{np.isclose}(y_{im}, \hat{y}_{im}, \texttt{atol}, \texttt{rtol})] \tag{18}$$

with rather lenient absolute and relative tolerances of $\texttt{atol} = 10^{-3}$ and $\texttt{rtol} = 0.05$, and

$$A_2 \uparrow^{[0,1]} = \frac{1}{N} \sum_{i=0}^{N} \mathbf{1}[R_i^2 > 0.95]. \tag{19}$$

Fraction of Variance Unexplained (FVU)

Since the test expressions result in ranges of vastly different orders of magnitude, I determine the "relative" error of the predictions $\hat{y}$ to the ground truth y compared to the variance of the ground truth points:

$$\text{FVU}(y, \hat{y}) \downarrow^{[0,\infty]} = \frac{\sum_{i=1}^{N} (y_i - \hat{y}_i)^2}{\sum_{i=1}^{N} (y_i - \overline{y})^2} \tag{20}$$

[5]While x^6 can be rewritten as $(x^3)^2$, the term x^7 cannot be split into chained integer power operators

However, I find that the logarithm of the FVU provides a scale that agrees more with intuitive observations of the goodness of fit, and therefore use $\log_{10} \mathrm{FVU}(y, \hat{y})$ for numeric evaluation. The exponentiated arithmetic mean of the log FVU values effectively yields the geometric mean of FVUs which is less affected by outliers than the arithmetic mean. I argue that this choice of mean is more representative of the expected performance in the presence of outliers and facilitates comparison of values of vastly different magnitudes.

Recovery Rates

Inspired by the A_2 metric in (Biggio et al. 2021), I compute the Numeric ε-Recovery Rate across a test set of instances $i = 1, ..., N$

$$\%_{\mathrm{FVU} \leq \varepsilon} \uparrow^{[0,100]} \;=\; \frac{100}{N} \sum_{i=0}^{N} \mathbf{1}_{\mathrm{FVU}(y_i, \hat{y}_i) \leq \varepsilon} \tag{21}$$

which measures the percentage of instances where the relative error as given by the FVU is smaller than a vanishingly small threshold ε, or in other words, the percentage of "perfect fits". I opt for the float32 precision limit $\varepsilon = \varepsilon_{32} = 1.1920929 \cdot 10^{-7}$.

As a complement to this metric, I report the mean log FVU of imperfect fits

$$\log_{10} \mathrm{FVU}_{>\varepsilon} \downarrow^{[-6.92,\infty]} \;=\; \frac{1}{N} \sum_{i=0}^{N} \mathbf{1}_{\mathrm{FVU}(y_i, \hat{y}_i) > \varepsilon} \log_{10}(\max(\varepsilon, \mathrm{FVU}(y_i, \hat{y}_i))) \tag{22}$$

which gauges the model's ability to approximate functions it cannot identify exactly.

Furthermore, I consider the Symbolic Recovery Rate

$$\%_{\hat{Z}=Z} \uparrow^{[0,100]} \;=\; \frac{100}{N} \sum_{i=0}^{N} \mathbf{1}_{\hat{Z}_i = Z_i} \tag{23}$$

which indicates how many predicted skeletons exactly match their respective ground truth skeletons. Since Algorithm 1 only approximately canonicalizes skeletons, $\%_{\hat{Z}=Z} \uparrow^{[0,100]}$ may disregard cases where a predicted skeleton does not match the ground truth one-to-one while still being functionally equivalent, and thus is a rather pessimistic metric.

Edit Distances

To further quantify the symbolic similarity of skeletons, I use the Levenshtein Distance $\mathrm{LEV} \downarrow^{[0,\infty]}$ and Zhang-Sasha Tree-Edit Distance $\mathrm{ZSS} \downarrow^{[0,\infty]}$ (Zhang and Shasha 1989) which measure the number of token-based edit operations to transform the predicted prefix skeleton into the ground truth prefix skeleton.

Perplexity

Given a ground truth skeleton Z_i of length L_i and data X_i, Y_i, the perplexity evaluates to the exponentiation of a model's average negative log-likelihood of ground truth tokens:

$$\mathrm{PPL}_\theta(Z_i) \downarrow^{[1,\infty]} = \exp\left[-\frac{1}{L_i} \sum_{l=0}^{L_i} \log p_\theta\big(z_{i,l} \mid z_{i,<l} \,; X_i, Y_i\big) \right]. \tag{24}$$

Low perplexity corresponds to high certainty about correct next tokens, with the value 1 indicating 100% certainty. An intuitive albeit slightly unrealistic analogy can be obtained by observing that the perplexity of a uniform distribution over K options $p(v) = \frac{1}{K}$ for $v \in \{1, 2, ..., K\}$ is exactly K:

$$\mathrm{PPL}(p(v)) = \exp\left[-\frac{1}{K}\sum_{v=1}^{K}\left[\log\frac{1}{K}\right]\right] = \exp\left[-\log\frac{1}{K}\right] = \exp[\log K] = K, \tag{25}$$

meaning that the perplexity can be very roughly thought of as the number of equally likely tokens that the model chooses from at each step.

Wall Time

As a measure of efficiency and usability, I record the real time spent fitting a given model to a data instance during inference $T_{\text{wall}}\downarrow^{[0,\infty]}$. Although all considered methods use CPU-worker or GPU-batch parallelization, I do not focus on the total system time (i.e. summed over all workers) for two reasons:

For one, methods that have been designed to run on different hardware (CPU and GPU) cannot be compared in an entirely fair manner: Running GPU-intended ones on a CPU provides a valuable comparison of the estimated compute spent, but it barely reflects the actual use case and potential of those models.

Second, by allowing each method to use as much state-of-the-art, off-the-shelf compute (Table 6) as requested at default settings, the wall time provides a realistic estimate of the time a user would have to wait for results.

3.6.3 Uncertainty Estimation

Measurements and statistics obtained from a system that is exposed to randomness require appropriate estimations of their uncertainty (Forde and Paganini 2019). This includes controlling for the usual sources of randomness introduced by network initializations and stochastic optimizers (Clark et al. 2011), and treating test sets merely as samples of a test population.

While controlling for data and optimizer stochasticity by means of retraining and averaging the performance of several identical but differently initialized models is not feasible in this work, I compute all metrics from procedurally generated test sets of $\mathcal{O}(10^3)$ instances generated from skeletons in the fixed test pools, and approximate their confidence intervals with the percentile bootstrap method over the test instances as described in (Riezler and Hagmann 2024) with $\mathcal{O}(10^3)$ bootstraps.

Furthermore, I calculate p-values with the Approximate Randomization Test (Fisher et al. 1966; Noreen 1989) using $\mathcal{O}(10^3)$ random permutations to determine the significance of difference in results.

4 Experiments

4.1 Simplification Algorithm

To compare my simplification algorithm to SymPy (Meurer et al. 2017), I sample 10k unsimplified (raw) random skeletons with operators

$$N_{\text{ops}} \sim \mathcal{P}^{+}\left(N_{\text{ops}}; [l = 0, h = 15], \beta = 1\right) \tag{26}$$

corresponding to a maximum length of 31 tokens, and use 100 retry attempts to sample constants and input points that result in a valid output set. These raw skeletons are then simplified with Algorithm 1, and SymPy with the default rejection threshold for lengths before and after simplification of `ratio = 1.7` and a more conservative value of `ratio = 1.0`.

For the SymPy simplification, I add a compatibility layer that converts the prefix skeletons into their infix form. This infix form is then parsed and simplified with SymPy before being parsed back into a prefix skeleton with ⚡ANSR utility functions.

4.2 Evaluation

Unless otherwise stated, I use the following procedure and default parameters:

Models are fit with 32 equivalence-pruned beams, a maximum length of 32 tokens, and 8 optimizer restarts per successful beam on a "fitting" set of 512 points. Constants are optimized with the Levenberg-Marquardt Algorithm (Levenberg 1944) and use the same prior (distribution of initial guesses) as the training data, e.g. $c_k \sim \mathcal{U}(-5, 5)$ for the main model ⚡ v7.0 . Numeric metrics are calculated on both the fitting set and the independently sampled validation set of equal size.

⚡ANSR, NeSymReS (Biggio et al. 2021), and PySR (Cranmer 2023) are evaluated under the same procedure. While Biggio et al. (2021) run all models on the CPU to compare runtime, I challenge the superficial fairness this seems to entail. I decide to use all models in their recommended way with default configurations, and allow them to use all available resources (up to 32 CPU workers, up to 1 GPU, see Table 6) to reflect realistic out-of-the-box usage in a high-end consumer compute environment.

4.3 Ablation Study

Starting with the base model (⚡ v7.0), I selectively deactivate, roll back or diminish a part of my model to assess its contribution to the total performance. Table 3 compares NeSymReS (Biggio et al. 2021) with ⚡ v7.0 and lists all ablations carried out. Clarification of each parameter is given in the subsequent main text.

Model	Simplify	$p_{N_{\text{ops}}}$	S	p_{const}	r	EPBS	Opt.
NeSymReS	SymPy	$\mathcal{U}(2,5)$	10	$\mathcal{U}(1,5)$	4	No	BFGS
⚡v7.0	Algorithm 1	$\mathcal{P}^+([0,10],1)$	64	$\mathcal{U}(-5,5)$	8	Yes	LM
⚡v7.1	**none**	$\mathcal{P}^+([0,10],1)$	64	$\mathcal{U}(-5,5)$	8	Yes	LM
⚡v7.2	Algorithm 1	$\boldsymbol{\mathcal{P}^+([0,5],1)}$	64	$\mathcal{U}(-5,5)$	8	Yes	LM
⚡v7.3	Algorithm 1	$\mathcal{P}^+([0,10],1)$	**10**	$\mathcal{U}(-5,5)$	8	Yes	LM
⚡v7.4	Algorithm 1	$\mathcal{P}^+([0,10],1)$	64	$\boldsymbol{\mathcal{U}(1,5)}$	**4**	Yes	LM
⚡v7.5	Algorithm 1	$\mathcal{P}^+([0,10],1)$	64	$\mathcal{U}(-5,5)$	8	**No**	LM
⚡v7.7	Algorithm 1	$\mathcal{P}^+([0,10],1)$	64	$\mathcal{U}(-5,5)$	8	Yes	**BFGS**

Table 3: **Parameters in the Ablation Study.** Several parameters of the base model (⚡v7.0) are changed to the values of NeSymReS (Biggio et al. 2021) one at a time (*bold values*).

I ablate the following:

- **Simplify**: The algorithm used to simplify and canonicalize the generated training skeletons
- $p_{N_{\text{ops}}}$: The distribution of the number of operators in each training expression (for details see Section 3.3)
- S: The number of the pooling seeds for the output of the Set Transformer Encoder (Lee et al. 2019) and thus the size of the encoding bottleneck
- p_{const}: The distribution of constants in training expressions
- r: The number of optimizer restarts during evaluation
- **EPBS**: Whether to use Equivalence-Pruned Beam Search (i.e. simplify and de-duplicate beams) during inference
- **Opt.**: The Optimizer used to fit numerical constants during inference. BFGS (Broyden 1970) or Levenberg-Marquardt (Levenberg 1944).

4.4 Inference Compute

To estimate how well the method scales with the compute spent during inference and validate the results of Biggio et al. (2021), I evaluate the base model ⚡v7.0 with varying beam widths $w \in [1, 2, 4, ..., 256, 512]$ and analyze the changes in the mean Numeric Recovery Rate $\%_{\text{FVU} \leq \varepsilon} \uparrow^{[0,100]}$, Symbolic Recovery Rate $\%_{\hat{Z}=Z} \uparrow^{[0,100]}$, Approximation Error $\log_{10} \text{FVU}_{>\varepsilon} \downarrow^{[-6.92,\infty]}$, F1 Score $\downarrow^{[0,1]}$, ZSS Tree-Edit Distance $\downarrow^{[0,\infty]}$, and Wall-Time in seconds $\downarrow^{[0,\infty]}$.

For the numeric metrics, I report both fit and validation values.

Furthermore, I compute the mean of the above metrics as a function of the beam width *and* the length of the ground truth expression, which gives insight into possible benefits of more inference compute at each complexity level.

4.5 Expression Length at Train & Test Time

For two models trained with different upper bounds of the expression length (Equation 1), ⚡ v7.0 ($h = 10$) and ⚡ v7.2 ($h = 5$), I analyze the mean logarithm of the FVU as a function of the ground truth expression length and determine the degree to which the complexity of training expressions influence the model's capabilities to recover complex equations during test time.

4.6 Variation of Constants

In this experiment, I probe the susceptibility of the method to variations in the values of constants, specifically μ and σ^2 in an unnormalized squared exponential

$$y = \exp\left(-\frac{(x_1 - \mu)^2}{\sigma^2}\right) \tag{27}$$

with 49 linearly spaced values for $\mu \in [-5, 5]$ and 49 logarithmically spaced values $\sigma^2 \in \left[10^{-1}, 10^{\frac{48}{18}} \approx 33.6\right]$ including $\sigma^2 = 1$. For each tuple (μ_i, σ_j^2), the models are tasked to fit five times to 500 uniformly sampled points in the interval $x_1 \in [-10, 10]$. I then compute the median FVU from the five fits.

Moreover, I show detailed fits and predicted equations for nine tuples of particular interest.

4.7 Input Sparsity

To evaluate the performance on sparse inputs, I iteratively halve the number of data points in the test instances, $M = 512, ..., 2, 1$. Compared to Biggio et al. (2021) who only show the rather lenient validation accuracies $A_{1/2}^{\text{iid/ood}}$, I report the Numeric Recovery Rate, $\%_{\text{FVU} \leq \varepsilon} \uparrow^{[0,100]}$, Symbolic Recovery Rate $\%_{\hat{Z}=Z} \uparrow^{[0,100]}$, Approximation Error $\log_{10} \text{FVU}_{>\varepsilon} \downarrow^{[-6.92,\infty]}$, F1 Score $\downarrow^{[0,1]}$, ZSS Tree-Edit Distance $\downarrow^{[0,\infty]}$, and Wall-Time in seconds $\downarrow^{[0,\infty]}$.

For the numeric metrics, I again report both fit and validation values.

4.8 Input Noise

Lastly, I investigate the method's behavior when exposed to noisy data with a variant of the QQ-plot. To this end, I add normally distributed noise of magnitudes $\alpha \in \{0.001, 0.01, 0.5, 1\}$ in units of standard deviations of the given data to the dependent variable:

$$\tilde{y}_m = y_m + \eta_m \cdot \sigma_y \quad \forall m = 1, ..., M \tag{28}$$

with $\eta_m \sim \mathcal{N}(\mu = 0, \sigma = \alpha)$. For this experiment, I use $M = 16$ points to create a scenario where noise would be hard to tell apart from sparse real data.

After fitting, I recover the ground truth residuals (i.e. the noise) of each point m

$$r_m = y_m - \tilde{y}_m \tag{29}$$

and the residuals of the predictions $\hat{y}$

$$\hat{r}_m = \hat{y}_m - \tilde{y}_m \tag{30}$$

and scale them by σ_y to obtain $Q_r = \frac{r}{\sigma_y}$ and $Q_{\hat{r}} = \frac{\hat{r}}{\sigma_y}$ for convenience.

Comparing r_m to $\hat{r}_m$ then allows for a calibration analysis

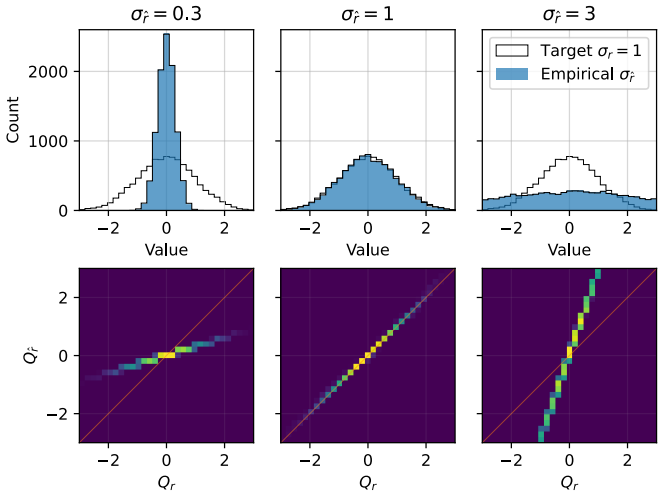

Figure 7: **Examples of QQ-Distributions for varying noise $\sigma_{\hat{r}}$.** Overfitting is present when the theoretical quantiles Q_r are larger than the empirical quantiles $Q_{\hat{r}}$ and vice versa.

5 Results

5.1 Simplification Algorithm

First, I compare the length of simplified expressions to the length of its original form for three simplification variants in Figure 8. While the default SymPy (Meurer et al. 2017) simplification barely reduces the length of the expression on average, setting its `ratio` parameter to 1 results in comparatively shorter expressions on average, especially at large lengths. However, despite setting this parameter such that it should only return expressions with lengths less or equal to the original length, "simplified" expressions often turn out longer than the original one. Although this phenomenon is not caused by SymPy per se, but rather subtleties in re-parsing the SymPy-simplified expression, it is clear that this inherent incompatibility between tokenized prefix expressions and SymPy objects presents a problem that is not adequately solved by current workarounds based on Biggio et al. (2021).

My proposed simplification algorithm overwhelmingly fulfills the expectation that simplified expressions should be shorter or at most as long as the original version, and results in the smallest mean simplified length across the majority of raw lengths.

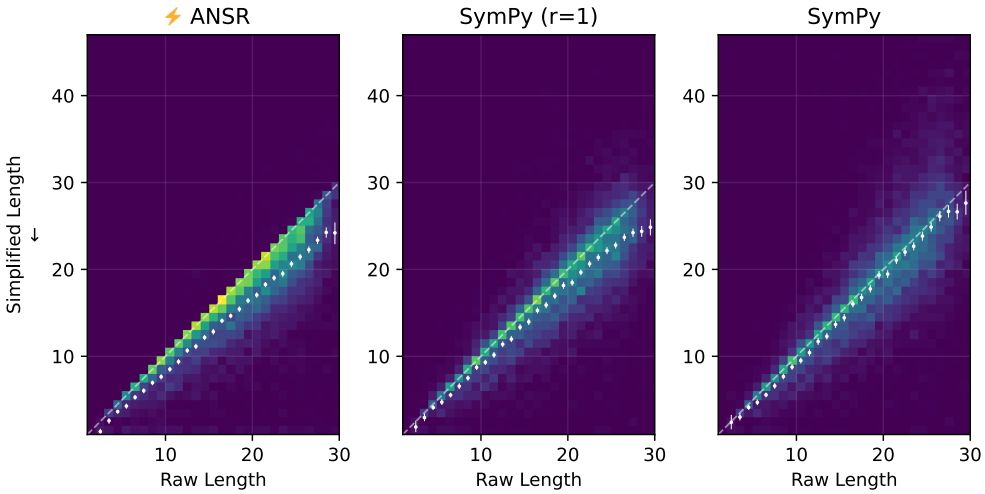

Figure 8: **Change of Expression Length after ⚡ANSR and SymPy Simplification.** ⚡ANSR reduces the length more reliably (clear edge at bisectrix in the histogram) and results in a smaller mean simplified length for any given raw length (*white points*, bootstrapped 95% CI, $n = 10^4$)

Additionally, Figure 9 shows that the proposed Algorithm 1 is orders of magnitude faster than SymPy (Meurer et al. 2017) and does not suffer from a rather annoying phenomenon where SymPy would be "stuck" trying to simplify particular expressions.

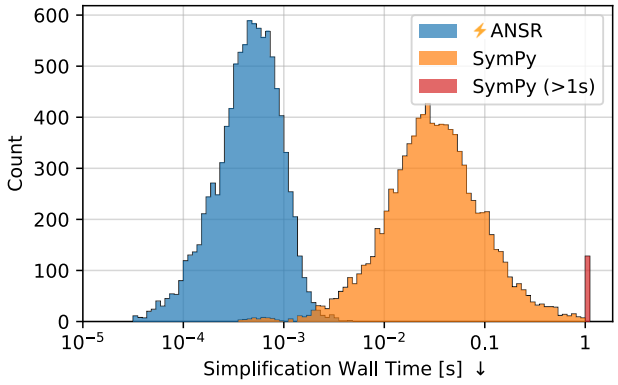

Figure 9: **Simplification Time per Expression.** Both histograms show recorded wall times of 10^4 simplifications performed by ⚡ANSR (*blue*) and SymPy (*orange*). Expressions for which SymPy could not find a simplification within 1 second are aggregated at $t = 1$ (*red*).

5.2 Model Comparison & Ablation Study

The new base model ⚡v7.0 outperforms both NeSymReS (Biggio et al. 2021) and PySR (Cranmer 2023) at perfectly explaining data numerically with the predicted expressions (+28.7pp and +4.1pp), and takes a small fraction of the time of NeSymReS with identical hyperparameters (98.8% time saved). PySR and ⚡v7.0 achieve very similar validation errors of almost FVU $= 10^{-3}$ for data which they cannot perfectly fit, an order of magnitude better than NeSymReS. Both amortized methods exhibit less overfitting when only approximating data.

Model	$\%_{\mathbf{FVU} \le \varepsilon}$ $\uparrow^{[0,100]}$		$\mathbf{log_{10}\,FVU}_{>\varepsilon}$ $\downarrow^{[-6.92,\infty]}$		**ZSS** $\downarrow^{[0,\infty]}$	**PPL** $\downarrow^{[1,\infty]}$	$\mathbf{T}_{\mathbf{wall}}$ **[s]** $\downarrow^{[0,\infty]}$
-	Fit	Val	Fit	Val	-	-	-
PySR	61.1 (59.5, 62.5) p = 0.00	60.2 (58.8, 61.7) p = 0.00	**−3.32** **(−3.42, −3.22)** p = 0.00	**−2.90** **(−3.03, −2.77)** p = 0.14	29.9 (29.3, 30.5) p = 0.00	— (—, —) p = —	3.99 (3.97, 4.01) p = 0.00
NeSymReS 100M	35.9 (34.4, 37.5) p = 0.00	35.6 (34.0, 37.1) p = 0.00	−1.75 (−1.83, −1.68) p = 0.00	−1.60 (−1.68, −1.51) p = 0.00	23.6 (23.2, 24.0) p = 0.00	— (—, —) p = —	85.9 (85.2, 86.7) p = 0.00
⚡v7.0	64.5 (63.8, 65.1) p = 1.00	64.3 (63.6, 65.0) p = 1.00	−2.97 (−3.02, −2.92) p = 1.00	**−2.80** **(−2.85, −2.74)** p = 1.00	**13.3** **(13.1, 13.4)** p = 1.00	**3.73** **(3.60, 3.86)** p = 1.00	1.03 (1.03, 1.03) p = 1.00
⚡v7.1 no simplify	60.6 (59.9, 61.3) p = 0.00	60.4 (59.7, 61.0) p = 0.00	−2.60 (−2.64, −2.55) p = 0.00	−2.46 (−2.50, −2.41) p = 0.00	13.4 (13.3, 13.6) p = 0.20	6.10 (5.97, 6.23) p = 0.00	1.12 (1.11, 1.12) p = 0.00
⚡v7.2 $N_{\text{ops}} \le 5$	59.2 (58.5, 59.8) p = 0.00	59.0 (58.3, 59.7) p = 0.00	−2.59 (−2.63, −2.55) p = 0.00	−2.44 (−2.49, −2.38) p = 0.00	**13.1** **(12.9, 13.2)** p = 0.10	10.1 (9.19, 11.1) p = 0.00	1.07 (1.07, 1.07) p = 0.00
⚡v7.3 $S = 10$	63.8 (63.1, 64.4) p = 0.15	63.5 (62.9, 64.1) p = 0.10	−2.83 (−2.87, −2.78) p = 0.00	−2.63 (−2.70, −2.51) p = 0.00	13.5 (13.3, 13.6) p = 0.07	4.08 (3.95, 4.19) p = 0.00	0.99 (0.99, 1.00) p = 0.00
⚡v7.4 $c \sim \mathcal{U}(1,5)$	**66.4** **(65.8, 67.1)** p = 0.00	**66.2** **(65.6, 66.8)** p = 0.00	−2.87 (−2.92, −2.82) p = 0.00	−2.70 (−2.75, −2.63) p = 0.02	13.5 (13.4, 13.7) p = 0.02	7.55 (7.23, 7.92) p = 0.00	0.98 (0.98, 0.98) p = 0.00
⚡v7.5 no EPBS	64.5 (63.9, 65.1) p = 0.97	64.3 (63.6, 64.9) p = 0.94	−2.97 (−3.02, −2.92) p = 0.96	**−2.81** **(−2.87, −2.75)** p = 0.74	13.4 (13.2, 13.5) p = 0.49	**3.80** **(3.67, 3.97)** p = 0.38	**0.90** **(0.90, 0.91)** p = 0.00
⚡v7.7 BFGS	64.3 (63.7, 65.0) p = 0.72	64.1 (63.5, 64.7) p = 0.68	−3.01 (−3.07, −2.92) p = 0.59	**−2.85** **(−2.91, −2.80)** p = 0.15	**13.0** **(12.8, 13.2)** p = 0.03	**3.73** **(3.62, 3.89)** p = 0.97	1.41 (1.40, 1.42) p = 0.00

Table 4: **Model Comparison and Ablation Study.** Bootstrapped 95% CI and AR-p values. Numbers represent mean (PPL: median) over all test sets.

⚡v7.0 also scores best on the ZSS tree-edit distance (Zhang and Shasha 1989) metric. It is important to note, however, that the test expressions are parsed and canonicalized with

Algorithm 1 before being compared to the raw predictions of each method. Since ⚡ANSR is trained to predict expressions in this format, it is unsurprising to see it achieve smaller ZSS scores.

The ablation study clearly highlights the importance of training on longer (more complex) expressions (vs ⚡v7.2) and simplifying training expressions (vs ⚡v7.1). Both are expected: If the model has never seen data sampled from complex expressions, it is unlikely to fit complex test expressions perfectly. Training on simplified and approximately canonicalized expressions may lead the model to learn unique and reliable forms of expressions to any given data. This in turn may pay off during beam search, where the beams are not littered with equivalent forms of the same expression, but rather a diverse set of different expression that enables more thorough exploration of different beam hypotheses.

5.3 Inference Compute

Figure 10 shows how various metrics improve with increase in beam width during inference. The Numeric Recovery Rate appears to increase logarithmically with the beam width until $w = 128$, from where on the method shows diminishing returns, which matches observations by Biggio et al. (2021).

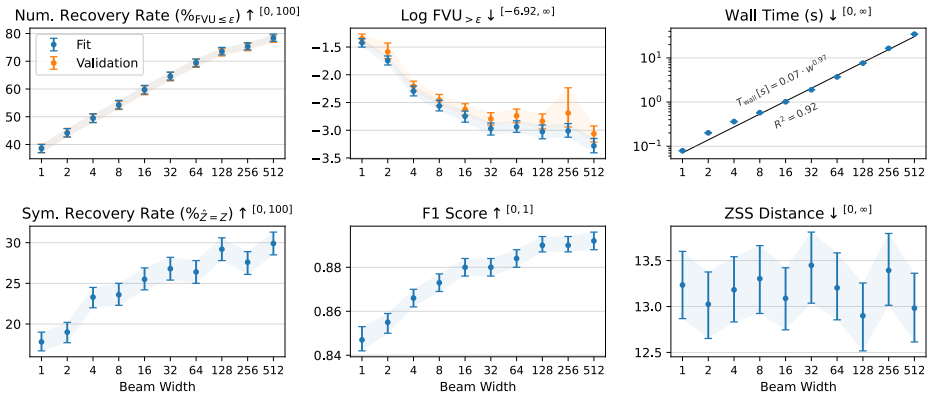

Figure 10: **Model Performance vs Beam Width During Inference (⚡v7.0).** Bootstrapped 95% CI of the mean.

The Symbolic Recovery Rate follows a similar trend but is less than half of the Numeric Recovery Rate, indicating that the method found solutions that are functionally equivalent but symbolically distinct from the ground truth. This could have two main reasons: For one, the data may not be uniquely identifiable by being almost perfectly explainable by more than one equation. Second, despite the effort to canonicalize predictions through equivalence pruning (Algorithm 3), the limited number of simplification rules make semantically identical but symbolically different predictions (i.e. reformulations of the same underlying expression) inevitable for sufficiently complicated expressions.

The numeric error for imperfect fits decreases by more than an order of magnitude from 1 to 32 beams before beginning to saturate at about FVU $\approx 10^{-3}$, leading to the hypothesis that the method benefits from more beams across the spectrum of difficulty. While Numeric Recovery Rate is very consistent between fit and validation data, there seems to be a tendency to overfit when the predicted expression only approximates the data.

The F1-Score between the best predicted and ground truth skeletons is decent even for a single beam ($0.847\,^{0.852}_{0.842}$) and only increases marginally to $0.892\,^{0.896}_{0.888}$ for 512 beams, meaning that the method generally identifies the correct operators and variables, and predicts few operators and variables that do not appear in the ground truth. The discrepancy between the high F1-Scores and the comparatively low perfect Numeric Recovery Rate especially at small beam widths could be the effect of instances with one or two misidentified operators that heavily impact the numeric error.

Equivalence-Pruned Beam Search (Algorithm 3) achieves linear time scaling with the number of beams at 70.0 ± 1.0 ms per beam .

Looking at the same metrics as a function of the bean width *and* the length of the ground truth expression in tokens (Figure 11) reveals that the model perfectly recovers longer equations at larger beam widths. However, no similar trend can be identified for other metrics. In particular, there seems to be a hard bound on the length of expressions that can be exactly symbolically recovered ($L \approx 12$).

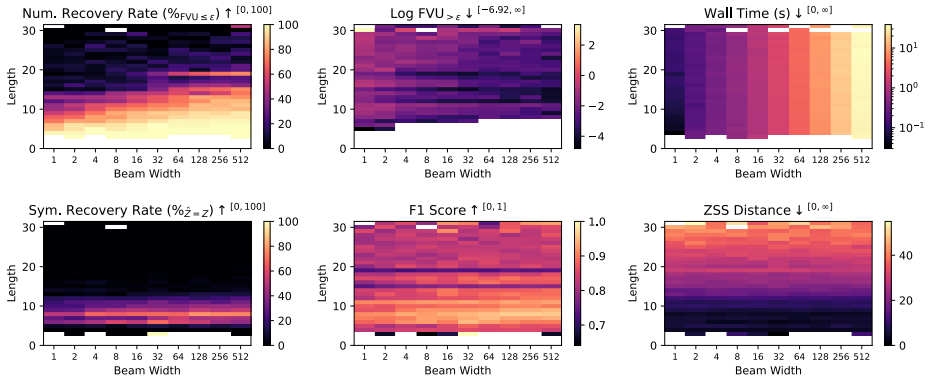

Figure 11: **Model Performance vs Beam Width During Inference and Test Expression Length (⚡v7.0).**

5.4 Expression Length at Train & Test Time

In Figure 12, I compare the logarithm of the FVU as a function of the test expression lengths in tokens for two models.

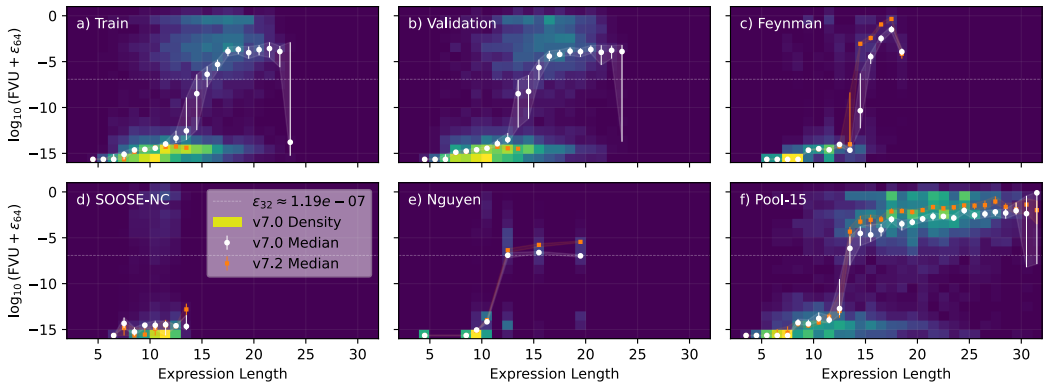

Figure 12: **$\log_{10}$FVU vs Test Expression Length for ⚡v7.0 ($h = 10$) and ⚡v7.2 ($h = 5$).**
Outliers not shown. Densities are scaled individually.

Overall, the logarithmic FVU appears to follow a logistic curve with two distinct "phases". Below $L_{\text{critical}} \approx 13 \pm 1$, expressions that perfectly describe the data up to floating point accuracy are found, while above, the method finds expressions that merely approximate the data to about 10^{-5} to 10^{-3} FVU. This phase transition coincides with the pattern seen in the Symbolic Recovery Rate in Figure 11, and suggests a causality between the ability to predict the ground truth expression exactly, and the resulting numeric error.

Comparing ⚡v7.0 ($h = 10$) to ⚡v7.2 ($h = 5$) reveals that both models have the same threshold of lengths of expressions which they can numerically recover, implying that training on more complex expressions does not necessarily shift the threshold of perfect recovery accordingly as one would naively assume. Only the FVU in the case of approximations above this threshold is consistently one order of magnitude worse for $h = 5$ than for $h = 10$.

5.5 Variation of Constants

Figure 13 compares the logarithm of the FVU of fits to $y = \exp\left(-\frac{(x-\mu)^2}{\sigma^2}\right)$ with varying mean μ and variance σ^2 for ⚡v7.0 trained on expressions with constants sampled from $c_k \sim \mathcal{U}(-5, 5)$, and the ablation ⚡v7.4 with constants $c_k \sim \mathcal{U}(1, 5)$ as in (Biggio et al. 2021).

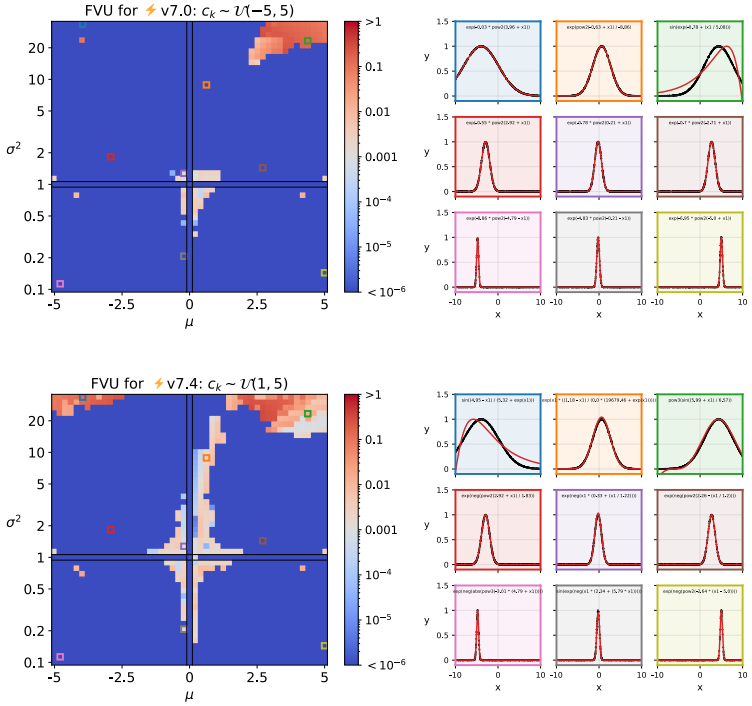

Figure 13: **FVU Under Variation of Constants μ and σ^2 in $y = \exp\left(-\frac{(x-\mu)^2}{\sigma^2}\right)$ for ⚡v7.0 with $c_k \sim \mathcal{U}(-5, 5)$ and ⚡v7.4 with $c_k \sim \mathcal{U}(1, 5)$ during Training.** Combinations of constants with particularly small ($|c| < 1$) or large out-of-distribution ($|c| > 10$) values are sometimes misidentified (*right grid*). Wider training priors (*top*) increase the reliability with respect to variations in constants.

Both models perfectly fit the data for the majority of constants combinations of moderate magnitude, but only find approximations of the ground truth close to $\mu = 0$ or $\sigma = 0$ and $\sigma \gg 1$ where the predicted skeletons are functionally different from the ground truth skeleton and do not allow for a perfect fit through the optimization of their constants. A likely hypothesis for these "areas" of imperfect fits is that the data generated by these constants combinations closely resemble data from different skeletons during training (e.g. ones that have vanishingly small μ or σ vs ones that have no constants at all). The learned embeddings of the Set Transformer then may not be detailed enough to reliably distinguish between the two, causing them to condition and mislead the Transformer decoder to output a different skeleton.

In these cases, one might still expect the beam search algorithm to find a set of skeletons that are incorrect to varying degrees but, by chance, would include the correct one. A particularly bad embedding, however, could explain why not just some, but all of the 32 beams do not match the ground truth expression. The decoder has just been conditioned to output a different one and cannot compensate by sampling many beams.

Comparing the two priors for constants during training, Figure 13 shows that including negative constants (as in ⚡v7.0 with $c_k \sim \mathcal{U}(-5, 5)$) improves the robustness against constants of small or large magnitude for the considered equation. This result appears to contradict the significantly better Numeric Recovery Rate of ⚡v7.4 over ⚡v7.0 in Table 4 and requires further studies to adequately discuss hypotheses.

Both heatmaps in Figure 13 also show signs of asymmetry with respect to shifts in the mean μ which is especially surprising since the model is trained on skeletons that feature constants sampled from a symmetric uniform distribution. While the details of this may come down to the incompleteness of the rule set for Algorithm 1, it is noteworthy that training the model includes sampling from a distribution that specifies the length of an expression. Imagining the length of each training expression as its "token budget", a negation generally requires an additional `neg` token. If this `neg` token cannot be fully absorbed into the constants of an expression due to a lack of simplification rules, the expression *with* the `neg` would be one token longer, and thus sampled less often or not at all if it exceeds the maximum length compared to the same expression *without* the `neg` token. This could influence the distribution of data that the model sees during training and thus result in an asymmetry of embeddings and subsequent predictions.

5.6 Input Sparsity

The results in Figure 16 give insight into the reliability of the method for data with varying numbers of points. Both Numeric (validation) and Symbolic Recovery Rates increase approximately logarithmically from $< 3\%$ at 1 point to their saturation values at 32 points. For data made of 1 or 2 points, the Fit Numeric Recovery Rate indicates severe overfitting, but surprisingly well-balanced predictions for data of at least 4 points.

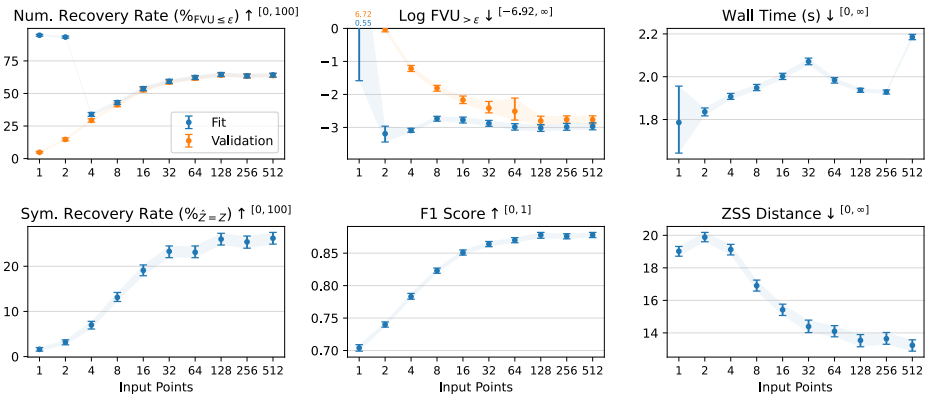

Figure 16: **Model Performance vs Number of Input Points (⚡v7.0).** Bootstrapped 95% CI of the mean.

It is unsurprising that with less data points, the Symbolic Recovery Rate decreases: The fewer points (i.e. constraints) there are, the more distinct functions can fit the data. We can

therefore expect the Symbolic Recovery Rate to decrease due to this identifiability problem. Furthermore, one could in principle argue that the *Numeric* Recovery Rate should stay the same or even increase for the same reason: Since more and more functions can fit the data of fewer and fewer points, an increasing proportion of predicted expressions should perfectly fit the data.

Interestingly, however, the method not only (expectedly) fails to generalize when fitting sparse data, but also struggles to fit to sparse data in the first place (!), resulting in a decrease by about 30 percentage points in the Fit *and* Validation Numeric Recovery Rate between 64 and 4 points. The inability to even fit data also appears to occur at a rate suspiciously proportional to its capability to generalize.

To better quantify this relation, I fit a linear model to the two quantities with SciPy's Orthogonal Distance Regression (Boggs and Rogers 2009; Virtanen et al. 2020) in Figure 17.

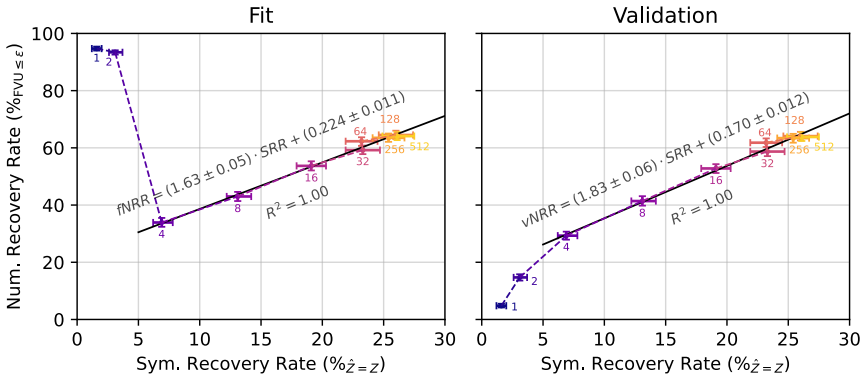

Figure 17: **Symbolic vs Numeric (left: Fit, right: Validation) Accuracy (⚡v7.0).** Bootstrapped 95% CI of the mean.

Ignoring the two outliers for 1 and 2 points, both Numeric and the Symbolic Recovery Rates (NRR and SRR) indeed behave proportionally to each other to a remarkable degree:

$$\text{fNRR} = (1.63 \pm 0.05) \cdot \text{SRR} + (22.4 \pm 0.011)\% \qquad R^2 = 0.998 \tag{31}$$

$$\text{vNRR} = (1.83 \pm 0.06) \cdot \text{SRR} + (17.0 \pm 0.012)\% \qquad R^2 = 0.998 \tag{32}$$

While in general, some may see this balance as a desirable property, I rather consider this inherent inability to overfit as a limitation of the method that needs attention in the future.

As a cherry-picked example of the practical implications of this effect, I apply fits to increasingly sparse data sampled from

$$y = 1.39 \cdot \sin(1.26 \cdot x_1) \tag{33}$$

in the interval $x_1 \in [-10, 10]$ and iteratively remove one point in Figure 18.

While equations with perfect fit error are found until 9 points, data with 8 to 3 points are only fit approximately by equations that display periodicity albeit without the ground truth $\sin(c_2 \cdot x_1)$ term.

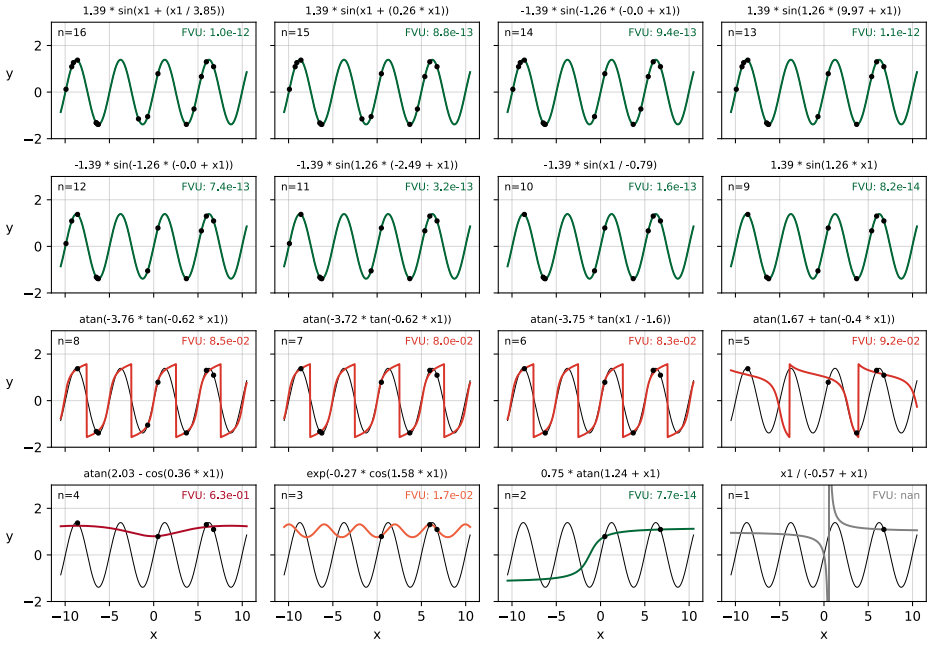

Figure 18: **Fits to Data with Decreasing Amount of Support. (⚡v7.0).** For sparse data, the recovered equations (*titles and colored curves*) differ from the ground truth (*gray curve*, Equation 33).

Interestingly, the subjective complexity of the predicted equations barely decrease with the number of available points to fit. Intuitively, one would, for example, fit a line $y = ax + b$ to two points. Yet the method chooses $y = a \cdot \arctan(x + b)$, possibly due to the monotonicity that the points exhibit.

Note that Figure 18 exclusively shows fits to data with less than or as many as the minimum number of points during training ($n \leq l = 16$).

5.7 Input Noise

Figure 19 compares the expected quantiles Q_r of the sampled noise in their distribution $\mathcal{N}(0, \eta \cdot \sigma_y)$ to the empirical quantiles $Q_{\hat{r}}$ of the noise to the predictions $\hat{y}$.

Overall, the method is well-calibrated with empirical quantiles close to the expectations, especially at large noise levels. This is unsurprising since small deviations from the fit to the ground truth only become statistically significant when the noise level is sufficiently small to tell noise from deviations.

For small noise levels, the results on the SOOCE-NC (Biggio et al. 2021) and Pool-15 test sets indicate two “modes” in the distribution. The majority of the fits is well-calibrated

(i.e. close to the diagonal $Q_r = Q_{\hat{r}}$) while other fits exhibit larger empirical quantiles than expected, indicating underfitting of comparable magnitude to the noise level. At least 50% of the Feynman (Udrescu et al. 2020) and Nguyen (Uy et al. 2011) fits on the other hand are still perfectly calibrated at noise level $\eta = 0.01$ with no significant sign of systematic underfitting. This discrepancy between test sets could be attributed to their inherent difficulty.

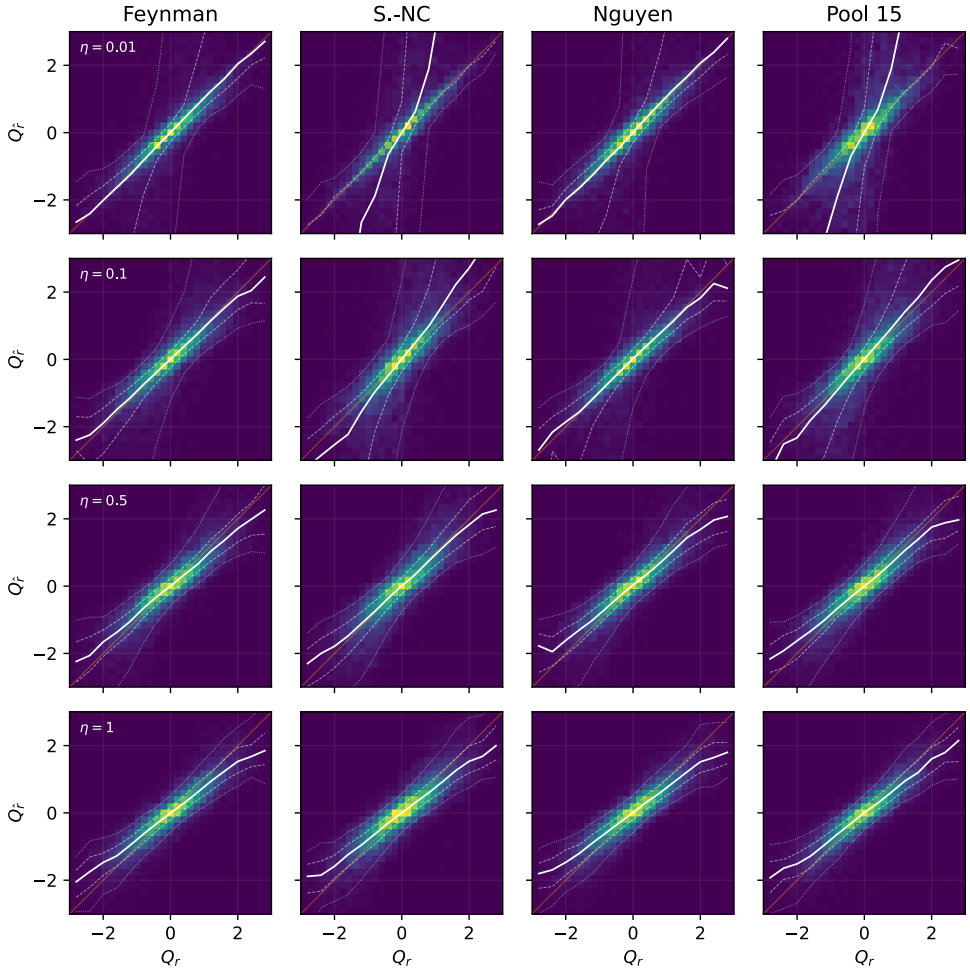

Figure 19: **QQ-Discrepancy for varying noise levels η (⚡v7.0).** Theoretical quantiles Q_r and empirical quantiles $Q_{\hat{r}}$ are superimposed for all test instances (*2d-histogram*) of every test set (*columns*) for different noise levels (*rows*). The identity $Q_r = Q_{\hat{r}}$ (*red diagonal line*) represents perfect calibration where the predictions $\hat{y}$ match the underlying (noiseless) ground truth y. The median empirical quantiles (*solid white lines*) and 25, 75 (*dashed lines*) and 5, 95 (*dotted lines*) percentiles over the test instances reveal good calibration for large noise levels $\eta \geq 0.1$ (i.e. $\geq 10\%$ relative noise).

Across all evaluated test sets and noise levels, there is also no significant sign of systematic overfitting, even though no parsimony penality is applied. Naively, one could hypothesize that the method could come up with some expression that always manages to (over-)fit

the data, for example by adding high-frequency trigonometric functions in its prediction. While I see this well-calibrated behavior as a promising sign, there may be use cases where an exact fit or fine-grained control over the tradeoff between fit error and complexity of the predicted expression is desired. This clearly calls for methods that support test-time control and regularization like NSwH (Bendinelli et al. 2023).

6 Conclusion

In this work, I have developed ⚡ANSR (Flash-ANSR), an improved method for Amortized Neural Symbolic Regression based on NeSymReS (Biggio et al. 2021). My method not only perfectly recovers +80% more test equations (FVU $< \varepsilon$) than the baseline, but does so while being 84 times faster thanks to the custom simplification algorithm as an alternative to SymPy (Meurer et al. 2017) which also enables fully procedural generation of training data. I have confirmed observations by Biggio et al. (2021) regarding the performance scaling with test time compute and input sparsity. Through various ablations of my method, the proposed changes have been assessed, finding that training on more complex equations does not directly result in more perfect recoveries of complex equations, but rather a reduction in FVU for the associated imperfect fits by an order of magnitude. Similarly, increasing in the embedding size significantly caused a reduction the approximation error ($p \leq 0.02$). Training with positive *and* negative constants resulted in the perhaps most interesting result, improving the robustness with respect to variations in constants, while slightly degrading overall performance.

The main failure mode is clearly given by the inability to recover long, complex equations. Training on longer equations somewhat surprisingly does not appear to help recover more of the longer equations perfectly. I expect that increasing the beam width will allow for perfect recovery of longer and longer equations, but it remains to be seen how well any purely amortized method can overcome this barrier inherent to the NP-hard problem that Symbolic Regression is (Virgolin and Pissis 2022).

One promising future approach might be an end-to-end solution that natively combines amortized large-scale pre-training with instance-level specialization through Reinforcement Learning or Genetic Programming. The hybrid approach proposed in uDSR (Landajuela et al. 2022) explores the combination of various paradigms, but strongly relies on existing state-of-the-art methods.

Future work may also focus on additional features and support, and various quality-of-life improvements. The current method, for example, is limited to 3 input variables although training on equations with up to $h = 10$ operators allows for $h + 1 = 11$ input variables. Amendments to the list of operators used during training and support for custom operators would increase the set of symbolically recoverable data.

While testing ⚡ANSR on SRBench (Cava et al. 2021) would improve clarity as to its performance compared to other state-of-the-art methods, I would rather encourage the benchmarking of methods at various compute limits as proposed by Biggio et al. (2021) and advocate for centrally collecting and reporting these performance curves as opposed to their performance for some arbitrary settings that may make a particular method look good on an accuracy chart.

Usability and performance may see improvements by incorporating control over properties of the predicted expressions as in NSRwH (Bendinelli et al. 2023), or by adding a contrastive loss term to encourage similar embeddings for data generated by equal skeletons (Li et al. 2022).

Since it appears that the number of training steps (1.5M) chosen by Biggio et al. (2021) is not compute-optimal (Appendix C), one might train the proposed transformer model for longer or investigate scaling laws of the method more broadly. Training could also benefit from recent work on schedule-free learning (Defazio et al. 2024).

References

[1] M. Virgolin and S. P. Pissis, "Symbolic Regression is NP-hard." [Online]. Available: https://arxiv.org/abs/2207.01018

[2] M. Cranmer, "PySR Research Showcase." [Online]. Available: https://ai.damtp.cam.ac.uk/pysr/papers/

[3] K. Champion, B. Lusch, J. N. Kutz, and S. L. Brunton, "Data-driven discovery of coordinates and governing equations," *Proceedings of the National Academy of Sciences*, vol. 116, no. 45, pp. 22445–22451, 2019.

[5] M. Cranmer, "Interpretable Machine Learning for Science with PySR and SymbolicRegression.jl." [Online]. Available: https://arxiv.org/abs/2305.01582

[4] B. Burlacu, G. Kronberger, and M. Kommenda, "Operon C++: An Efficient Genetic Programming Framework for Symbolic Regression," in *Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion*, in GECCO '20. Cancún, Mexico: Association for Computing Machinery, 2020, pp. 1562–1570. doi: 10.1145/3377929.3398099.

[7] M. Landajuela *et al.*, "Discovering symbolic policies with deep reinforcement learning," in *Proceedings of the 38th International Conference on Machine Learning*, M. Meila and T. Zhang, Eds., in Proceedings of Machine Learning Research, vol. 139. PMLR, 2021, pp. 5979–5989. [Online]. Available: https://proceedings.mlr.press/v139/landajuela21a.html

[6] M. Landajuela *et al.*, "A unified framework for deep symbolic regression," *Advances in Neural Information Processing Systems*, vol. 35, pp. 33985–33998, 2022.

[8] S.-M. Udrescu and M. Tegmark, "AI Feynman: a Physics-Inspired Method for Symbolic Regression." [Online]. Available: https://arxiv.org/abs/1905.11481

[9] L. Biggio, T. Bendinelli, A. Neitz, A. Lucchi, and G. Parascandolo, "Neural symbolic regression that scales," in *International Conference on Machine Learning*, 2021, pp. 936–945.

[10] B. K. Petersen, M. Landajuela, T. N. Mundhenk, C. P. Santiago, S. K. Kim, and J. T. Kim, "Deep symbolic regression: Recovering mathematical expressions from data via risk-seeking policy gradients," in *Proc. of the International Conference on Learning Representations*, 2021.

[11] F.-A. Fortin, F.-M. De Rainville, M.-A. Gardner, M. Parizeau, and C. Gagné, " DEAP: Evolutionary Algorithms Made Easy ," *Journal of Machine Learning Research*, vol. 13, pp. 2171–2175, 2012.

[12] W. Li *et al.*, "Transformer-based model for symbolic regression via joint supervised learning," in *The Eleventh International Conference on Learning Representations*, 2022.

[13] A. Vaswani *et al.*, "Attention Is All You Need." [Online]. Available: https://arxiv.org/abs/1706.03762

[14] G. Lample and F. Charton, "Deep Learning for Symbolic Mathematics." [Online]. Available: https://arxiv.org/abs/1912.01412

[15] A. Meurer *et al.*, "SymPy: symbolic computing in Python," *PeerJ Computer Science*, vol. 3, p. e103, Jan. 2017, doi: 10.7717/peerj-cs.103.

[16] M. Valipour, B. You, M. Panju, and A. Ghodsi, "Symbolicgpt: A generative transformer model for symbolic regression," *arXiv preprint arXiv:2106.14131*, 2021.

[17] C. R. Qi, H. Su, K. Mo, and L. J. Guibas, "Pointnet: Deep learning on point sets for 3d classification and segmentation," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 652–660.

[19] A. Radford, K. Narasimhan, T. Salimans, I. Sutskever, and others, "Improving language understanding by generative pre-training," 2018.

[20] A. Radford *et al.*, "Language models are unsupervised multitask learners," *OpenAI blog*, vol. 1, no. 8, p. 9, 2019.

[18] T. Brown *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[21] S. d'Ascoli, S. Becker, A. Mathis, P. Schwaller, and N. Kilbertus, "Odeformer: Symbolic regression of dynamical systems with transformers," *arXiv preprint arXiv:2310.05573*, 2023.

[22] M. Vastl, J. Kulhánek, J. Kubalík, E. Derner, and R. Babuška, "SymFormer: End-to-End Symbolic Regression Using Transformer-Based Architecture," *IEEE Access*, vol. 12, no. , pp. 37840–37849, 2024, doi: 10.1109/ACCESS.2024.3374649.

[23] T. Bendinelli, L. Biggio, and P.-A. Kamienny, "Controllable Neural Symbolic Regression." [Online]. Available: https://arxiv.org/abs/2304.10336

[24] Y. Li *et al.*, "Generative Pre-Trained Transformer for Symbolic Regression Base In-Context Reinforcement Learning." [Online]. Available: https://arxiv.org/abs/2404.06330

[25] S.-M. Udrescu, A. Tan, J. Feng, O. Neto, T. Wu, and M. Tegmark, "AI Feynman 2.0: Pareto-optimal symbolic regression exploiting graph modularity." [Online]. Available: https://arxiv.org/abs/2006.10782

[27] Z. Liu *et al.*, "Kan: Kolmogorov-arnold networks," *arXiv preprint arXiv:2404.19756*, 2024b.

[26] Z. Liu, P. Ma, Y. Wang, W. Matusik, and M. Tegmark, "KAN 2.0: Kolmogorov-Arnold Networks Meet Science." [Online]. Available: https://arxiv.org/abs/2408.10205

[28] S. L. Brunton, J. L. Proctor, and J. N. Kutz, "Discovering governing equations from data by sparse identification of nonlinear dynamical systems," *Proceedings of the National Academy of Sciences*, vol. 113, no. 15, pp. 3932–3937, Mar. 2016, doi: 10.1073/pnas.1517384113.

[29] P. Saegert, "On Data-Driven Discovery Of Symbolic Differential Equations From Unsuitable Coordinates Using SINDy-Autoencoders." [Online]. Available: https://github.com/psaegert/sindy-autoencoders-thesis

[30] J. Lee, Y. Lee, J. Kim, A. R. Kosiorek, S. Choi, and Y. W. Teh, "Set Transformer: A Framework for Attention-based Permutation-Invariant Neural Networks." [Online]. Available: https://arxiv.org/abs/1810.00825

[31] J. L. Ba, J. R. Kiros, and G. E. Hinton, "Layer Normalization." [Online]. Available: https://arxiv.org/abs/1607.06450

[32] L. Zhang, V. Tozzo, J. Higgins, and R. Ranganath, "Set Norm and Equivariant Skip Connections: Putting the Deep in Deep Sets," in *Proceedings of the 39th International Conference on Machine Learning*, K. Chaudhuri, S. Jegelka, L. Song, C. Szepesvari, G. Niu, and S. Sabato, Eds., in Proceedings of Machine Learning Research, vol. 162. PMLR, 2022, pp. 26559–26574. [Online]. Available: https://proceedings.mlr.press/v162/zhang22ac.html

[33] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A Simple Way to Prevent Neural Networks from Overfitting," *Journal of Machine Learning Research*, vol. 15, no. 56, pp. 1929–1958, 2014, [Online]. Available: http://jmlr.org/papers/v15/srivastava14a.html

[34] D. S. Kalra and M. Barkeshli, "Why Warmup the Learning Rate? Underlying Mechanisms and Improvements." [Online]. Available: https://arxiv.org/abs/2406.09405

[35] D. P. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization." [Online]. Available: https://arxiv.org/abs/1412.6980

[36] I. Loshchilov and F. Hutter, "Decoupled Weight Decay Regularization." [Online]. Available: https://arxiv.org/abs/1711.05101

[37] S. J. Reddi, S. Kale, and S. Kumar, "On the Convergence of Adam and Beyond," in *International Conference on Learning Representations*, 2018. [Online]. Available: https://openreview.net/forum?id=ryQu7f-RZ

[38] C. G. Broyden, "The Convergence of a Class of Double-rank Minimization Algorithms 2. The New Algorithm," *Ima Journal of Applied Mathematics*, vol. 6, pp. 222–231, 1970, [Online]. Available: https://api.semanticscholar.org/CorpusID:53666480

[39] K. Levenberg, "A METHOD FOR THE SOLUTION OF CERTAIN NON – LINEAR PROBLEMS IN LEAST SQUARES," *Quarterly of Applied Mathematics*, vol. 2,

pp. 164–168, 1944, [Online]. Available: https://api.semanticscholar.org/CorpusID:124308544

[40] D. W. Marquardt, "An Algorithm for Least-Squares Estimation of Nonlinear Parameters," *Journal of the Society for Industrial and Applied Mathematics*, vol. 11, no. 2, pp. 431–441, 1963, doi: 10.1137/0111030.

[41] P. Virtanen *et al.*, "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python," *Nature Methods*, vol. 17, pp. 261–272, 2020, doi: 10.1038/s41592-019-0686-2.

[42] N. X. Hoai, R. I. McKay, D. Essam, and R. Chau, "Solving the symbolic regression problem with tree-adjunct grammar guided genetic programming: The comparative results," in *Proceedings of the 2002 Congress on Evolutionary Computation. CEC'02 (Cat. No. 02TH8600)*, 2002, pp. 1326–1331.

[43] C. G. Johnson, "Genetic programming crossover: Does it cross over?," in *Genetic Programming: 12th European Conference, EuroGP 2009 Tübingen, Germany, April 15-17, 2009 Proceedings 12*, 2009, pp. 97–108.

[44] M. Keijzer, "Improving symbolic regression with interval arithmetic and linear scaling," in *European Conference on Genetic Programming*, 2003, pp. 70–82.

[45] N. Q. Uy, N. X. Hoai, M. O'Neill, R. I. McKay, and E. Galván-López, "Semantically-based crossover in genetic programming: application to real-valued symbolic regression," *Genetic Programming and Evolvable Machines*, vol. 12, pp. 91–119, 2011.

[46] K. Zhang and D. Shasha, "Simple fast algorithms for the editing distance between trees and related problems," *SIAM journal on computing*, vol. 18, no. 6, pp. 1245–1262, 1989.

[47] J. Z. Forde and M. Paganini, "The Scientific Method in the Science of Machine Learning." [Online]. Available: https://arxiv.org/abs/1904.10922

[48] J. H. Clark, C. Dyer, A. Lavie, and N. A. Smith, "Better hypothesis testing for statistical machine translation: Controlling for optimizer instability," in *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, 2011, pp. 176–181.

[49] S. Riezler and M. Hagmann, *Validity, Reliability, and Significance: Empirical Methods for NLP and Data Science - Second Edition*, Second. in Synthesis Lectures on Human Language Technologies. Springer, 2024. doi: https://doi.org/10.1007/978-3-031-57065-0.

[50] R. A. Fisher *et al.*, *The design of experiments*, vol. 21. Springer, 1966.

[51] E. W. Noreen, *Computer-intensive methods for testing hypotheses*. Wiley New York, 1989.

[52] P. T. Boggs and J. E. Rogers, "Orthogonal Distance Regression ∗," 2009. [Online]. Available: https://api.semanticscholar.org/CorpusID:15393564

[53] W. L. Cava *et al.*, "Contemporary Symbolic Regression Methods and their Relative Performance." [Online]. Available: https://arxiv.org/abs/2107.14351

[54] A. Defazio, X. A. Yang, H. Mehta, K. Mishchenko, A. Khaled, and A. Cutkosky, "The Road Less Scheduled." [Online]. Available: https://arxiv.org/abs/2405.15682

[55] S. Banerjee and A. Lavie, "METEOR: An Automatic Metric for MT Evaluation with Improved Correlation with Human Judgments," in *Proceedings of the ACL Workshop*

on Intrinsic and Extrinsic Evaluation Measures for Machine Translation and/or Summarization, J. Goldstein, A. Lavie, C.-Y. Lin, and C. Voss, Eds., Ann Arbor, Michigan: Association for Computational Linguistics, Jun. 2005, pp. 65–72. [Online]. Available: https://aclanthology.org/W05-0909/

[56] J. Kaplan *et al.*, "Scaling Laws for Neural Language Models." [Online]. Available: https://arxiv.org/abs/2001.08361

[58] NumPy, "numpy.hanning." [Online]. Available: https://numpy.org/doc/2.0/reference/generated/numpy.hanning.html

[57] R. Blackman and J. Tukey, "The measurement of power spectra Dover Publications," *Inc, New York*, 1958.

A Examples

In the following, I present fits of various one- and two-dimensional equations, highlighting both successes (Figure 20) and failures (Figure 21). The two-dimensional examples in Figure 22 and Figure 23 have been adopted from Biggio et al. (2021). For each fit, I report the FVU and joint log probability of the predicted skeleton.

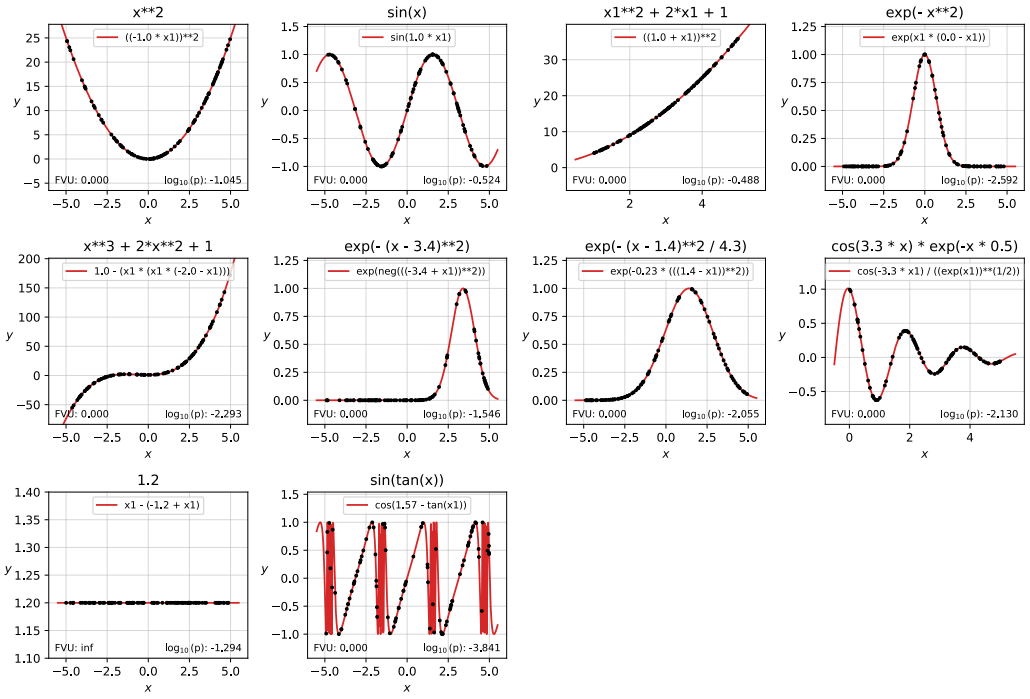

Figure 20: **1D Examples of Perfect Fits.** 100 data points (*black dots*) sampled from a ground truth expression (*titles*) are fit with ⚡v7.0 using Equivalence-Pruned Beam Search with 32 beams, 8 optimizer restarts and a parsimony of $\gamma = 0.05$ (*red curves*). The method finds concise expressions that perfectly fit the data.

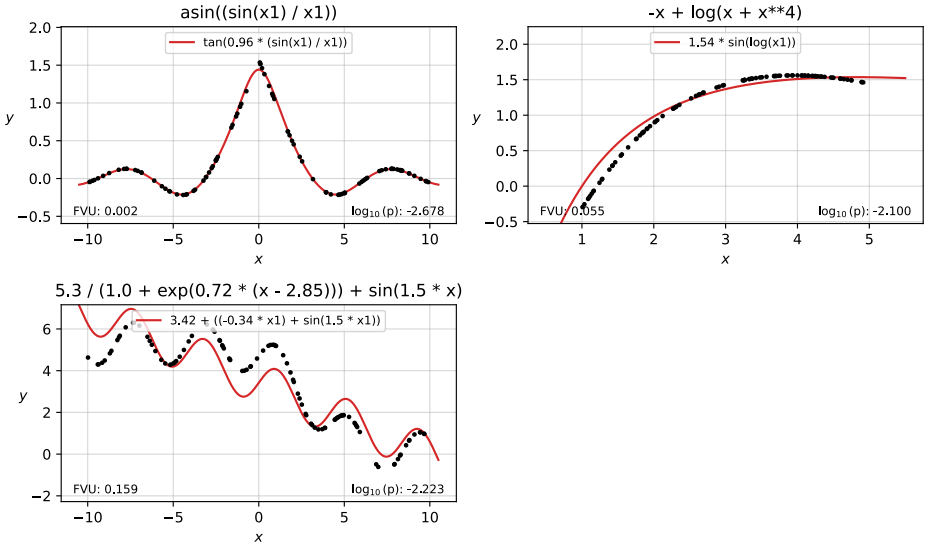

Figure 21: **1D Examples of Approximate Fits.** 100 data points (*black dots*) sampled from a ground truth expression (*titles*) are fit with ⚡v7.0 using Equivalence Pruned Beam Search with 32 beams, 8 optimizer restarts and a parsimony of $\gamma = 0.05$ (*red curves*). With the limited inference time, some data cannot be fit perfectly and only the coarse shape is reflected in the found expressions.

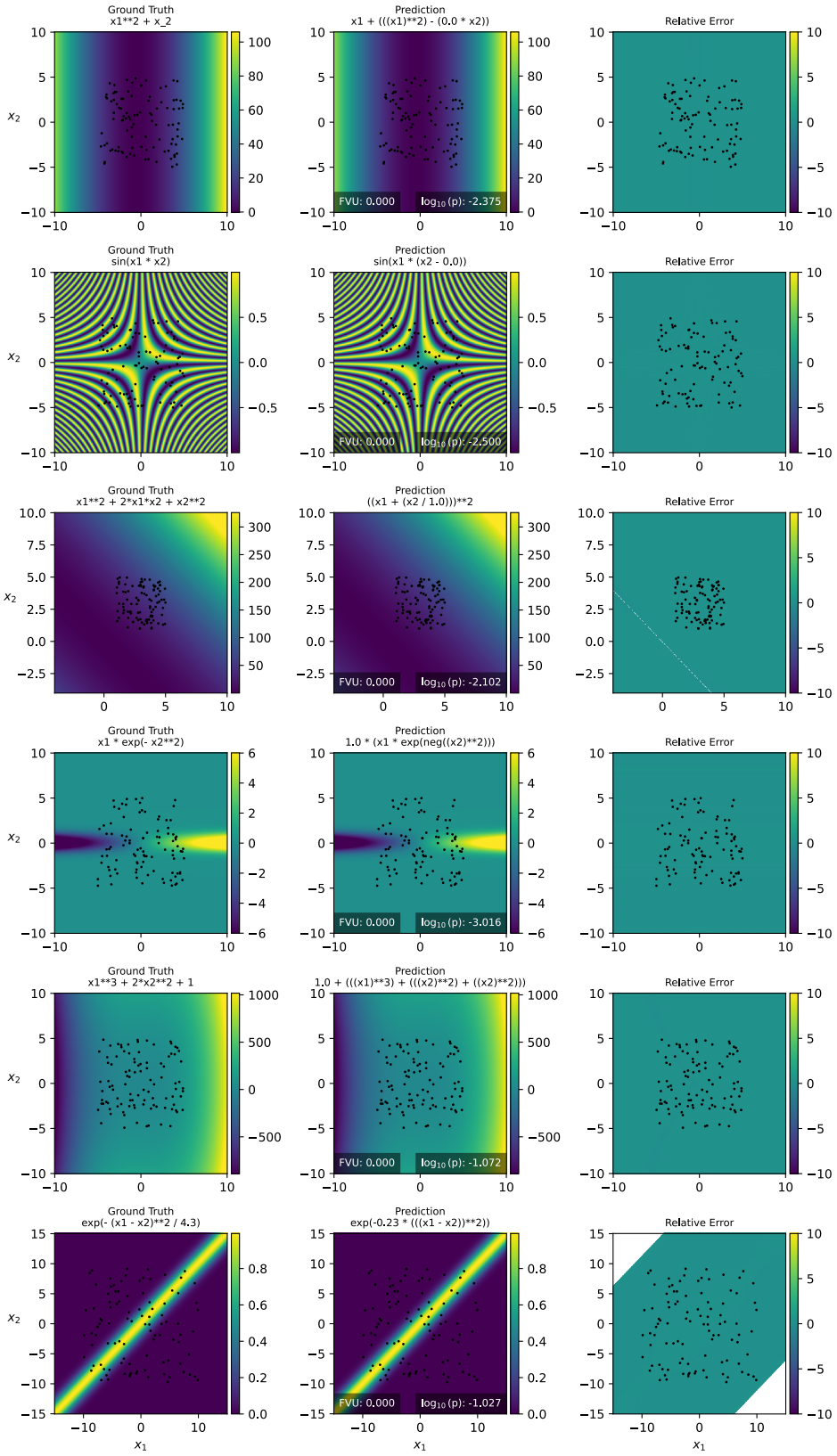

Figure 22: **2D Examples (1/2).** 100 data points (*black dots*) sampled from a ground truth expression (*left columns*) are fit with ⚡v7.0 using Equivalence Pruned Beam Search with 32 beams, 8 optimizer restarts and a parsimony of $\gamma = 0.05$ (*middle columns*).

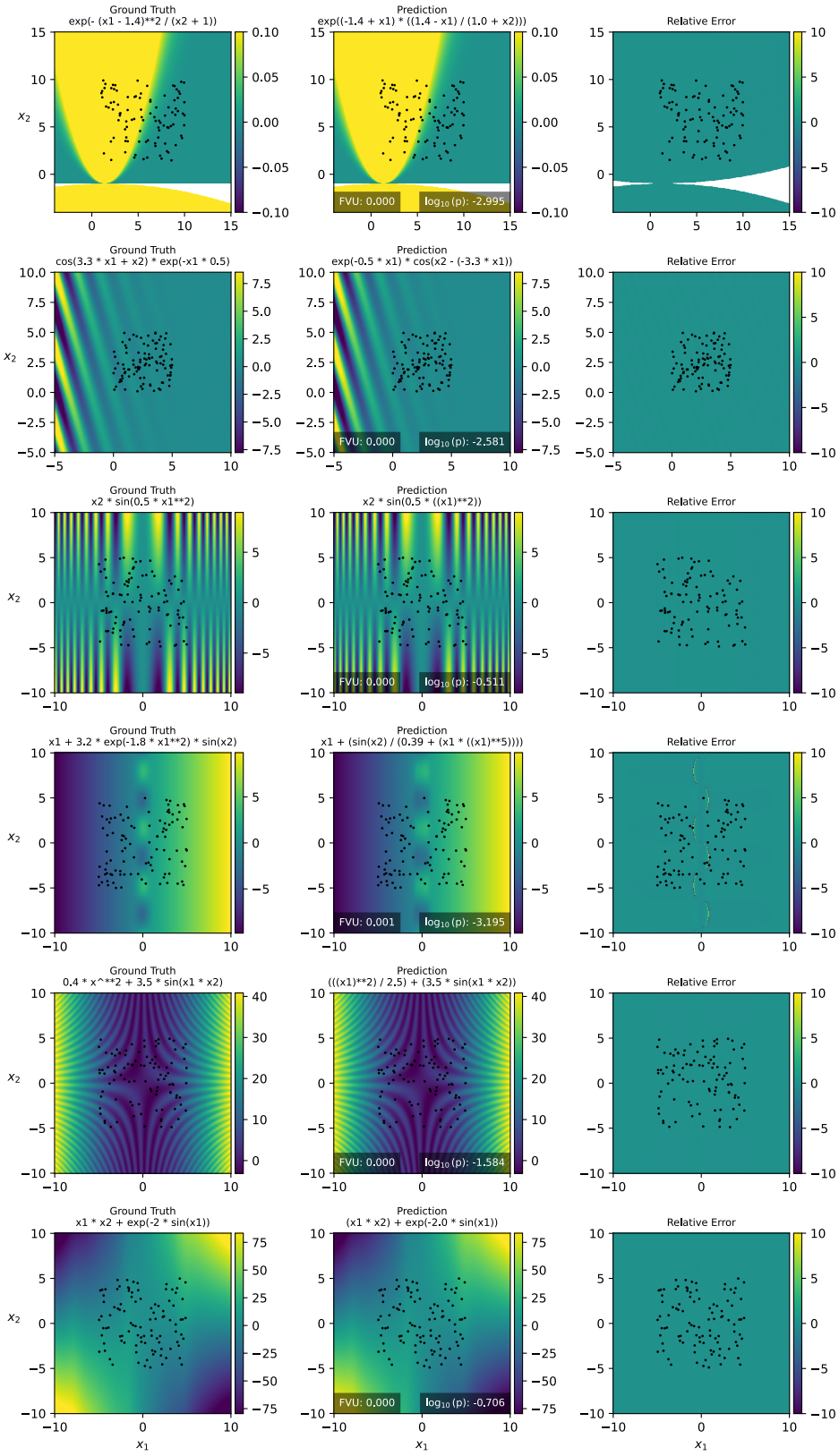

Figure 23: **2D Examples (2/2).** 100 data points (*black dots*) sampled from a ground truth expression (*left columns*) are fit with ⚡v7.0 using Equivalence Pruned Beam Search with 32 beams, 8 optimizer restarts and a parsimony of $\gamma = 0.05$ (*middle columns*).

B All Metrics

In Table 5, various additional metrics are compared across the ⚡v7.x model family, PySR (Cranmer 2023) and NeSymReS-100M (Biggio et al. 2021), denoted N100.

Metric	PySR	N100	⚡v7.0	⚡v7.1	⚡v7.2	⚡v7.3	⚡v7.4	⚡v7.5	⚡v7.7
$\%_{\text{valid}} \uparrow^{[0,100]}$	**100** **(100, 100)** $p = 1.00$	94.1 (93.3, 94.7) $p = 0.00$	**100** **(100, 100)** $p = 1.00$	**100** **(99.9, 100)** $p = 0.50$	**100** **(100, 100)** $p = 1.00$	**100** **(100, 100)** $p = 1.00$	**100** **(100, 100)** $p = 1.00$	**100** **(100, 100)** $p = 0.58$	**100** **(100, 100)** $p = 0.63$
$\%_{\text{FVU} \leq \varepsilon} \uparrow^{[0,100]}$ Numeric Recovery Rate (Fit)	61.1 (59.5, 62.5) $p = 0.00$	35.9 (34.4, 37.5) $p = 0.00$	64.5 (63.8, 65.1) $p = 1.00$	60.6 (59.9, 61.3) $p = 0.00$	59.2 (58.5, 59.8) $p = 0.00$	63.8 (63.1, 64.4) $p = 0.15$	**66.4** **(65.8, 67.1)** $p = 0.00$	64.5 (63.9, 65.1) $p = 0.97$	64.3 (63.7, 65.0) $p = 0.72$
$\%_{\text{FVU} \leq \varepsilon} \uparrow^{[0,100]}$ Numeric Recovery Rate (Val)	60.2 (58.8, 61.7) $p = 0.00$	35.6 (34.0, 37.1) $p = 0.00$	64.3 (63.6, 65.0) $p = 1.00$	60.4 (59.7, 61.0) $p = 0.00$	59.0 (58.3, 59.7) $p = 0.00$	63.5 (62.9, 64.1) $p = 0.10$	**66.2** **(65.6, 66.8)** $p = 0.00$	64.3 (63.6, 64.9) $p = 0.94$	64.1 (63.5, 64.7) $p = 0.68$
$\%_{f(X) \approx Y} \uparrow^{[0,100]}$ NeSymReS A_{M}^{M} (Biggio et al. 2021)	79.0 (77.7, 80.2) $p = 0.01$	50.1 (48.6, 51.8) $p = 0.00$	80.8 (80.2, 81.4) $p = 1.00$	75.8 (75.2, 76.4) $p = 0.00$	74.5 (73.9, 75.1) $p = 0.00$	79.9 (79.4, 80.5) $p = 0.03$	**82.0** **(81.5, 82.5)** $p = 0.00$	80.3 (79.8, 80.9) $p = 0.26$	**81.4** **(80.9, 81.9)** $p = 0.10$
$\%_{R^2 > 0.95} \uparrow^{[0,100]}$ NeSymReS A_{M}^{M} (Biggio et al. 2021)	**91.1** **(90.2, 92.0)** $p = 0.56$	65.4 (63.8, 66.8) $p = 0.00$	**90.8** **(90.4, 91.2)** $p = 1.00$	87.2 (86.8, 87.7) $p = 0.00$	86.6 (86.1, 87.1) $p = 0.00$	**90.0** **(89.6, 90.4)** $p = 0.00$	**90.9** **(90.5, 91.3)** $p = 0.74$	**90.8** **(90.5, 91.2)** $p = 0.90$	**90.9** **(90.5, 91.3)** $p = 0.64$
$\%_{\hat{Z}=Z} \uparrow^{[0,100]}$ Symbolic Recovery Rate	0.20 (0.10, 0.40) $p = 0.00$	7.00 (6.20, 7.80) $p = 0.00$	26.1 (25.5, 26.8) $p = 1.00$	**27.8** **(27.2, 28.4)** $p = 0.00$	26.0 (25.4, 26.6) $p = 0.76$	24.9 (24.3, 25.6) $p = 0.01$	26.5 (25.9, 27.1) $p = 0.37$	25.8 (25.3, 26.4) $p = 0.46$	26.3 (25.7, 27.0) $p = 0.63$
$\log_{10} \text{FVU}_{>\varepsilon} \downarrow^{[-6.92,\infty]}$ Approximation Error (Fit)	**−3.32** **(−3.42, −3.22)** $p = 0.00$	−1.75 (−1.83, −1.68) $p = 0.00$	−2.97 (−3.02, −2.92) $p = 1.00$	−2.60 (−2.64, −2.55) $p = 0.00$	−2.59 (−2.63, −2.55) $p = 0.00$	−2.83 (−2.87, −2.78) $p = 0.00$	−2.87 (−2.92, −2.82) $p = 0.00$	−2.97 (−3.02, −2.92) $p = 0.96$	−3.01 (−3.07, −2.92) $p = 0.59$
$\log_{10} \text{FVU}_{>\varepsilon} \downarrow^{[-6.92,\infty]}$ Approximation Error (Val)	**−2.90** **(−3.03, −2.77)** $p = 0.14$	−1.60 (−1.68, −1.51) $p = 0.00$	**−2.80** **(−2.85, −2.74)** $p = 1.00$	−2.46 (−2.50, −2.41) $p = 0.00$	−2.44 (−2.49, −2.38) $p = 0.00$	−2.63 (−2.70, −2.51) $p = 0.00$	−2.70 (−2.75, −2.63) $p = 0.02$	**−2.81** **(−2.87, −2.75)** $p = 0.74$	**−2.85** **(−2.91, −2.80)** $p = 0.15$
LEV $\downarrow^{[0,\infty]}$ Levenshtein Distance	11.4 (11.2, 11.6) $p = 0.00$	8.56 (8.42, 8.70) $p = 0.00$	**4.98** **(4.92, 5.04)** $p = 1.00$	5.16 (5.10, 5.22) $p = 0.00$	5.07 (5.00, 5.14) $p = 0.06$	5.07 (5.00, 5.13) $p = 0.08$	5.12 (5.06, 5.19) $p = 0.00$	**5.00** **(4.94, 5.06)** $p = 0.66$	**4.91** **(4.84, 4.97)** $p = 0.10$
ZSS $\downarrow^{[0,\infty]}$ Tree-Edit Distance	29.9 (29.3, 30.5) $p = 0.00$	23.6 (23.2, 24.0) $p = 0.00$	**13.3** **(13.1, 13.4)** $p = 1.00$	13.4 (13.3, 13.6) $p = 0.20$	**13.1** **(12.9, 13.2)** $p = 0.10$	13.5 (13.3, 13.6) $p = 0.07$	13.5 (13.4, 13.7) $p = 0.02$	13.4 (13.2, 13.5) $p = 0.49$	**13.0** **(12.8, 13.2)** $p = 0.03$
METEOR $\uparrow^{[0,1]}$ (Banerjee and Lavie 2005)	0.36 (0.36, 0.37) $p = 0.00$	0.46 (0.45, 0.47) $p = 0.00$	**0.67** **(0.67, 0.67)** $p = 1.00$	0.66 (0.65, 0.66) $p = 0.00$	0.64 (0.64, 0.64) $p = 0.00$	0.66 (0.66, 0.66) $p = 0.00$	**0.67** **(0.66, 0.67)** $p = 0.39$	**0.67** **(0.67, 0.67)** $p = 0.52$	**0.67** **(0.67, 0.67)** $p = 0.55$
F1 $\uparrow^{[0,1]}$	0.68 (0.67, 0.68) $p = 0.00$	0.71 (0.70, 0.72) $p = 0.00$	**0.88** **(0.88, 0.88)** $p = 1.00$	**0.88** **(0.88, 0.88)** $p = 0.35$	0.87 (0.87, 0.88) $p = 0.00$	**0.88** **(0.87, 0.88)** $p = 0.00$	**0.88** **(0.88, 0.88)** $p = 0.36$	**0.88** **(0.88, 0.88)** $p = 0.87$	**0.88** **(0.88, 0.88)** $p = 0.76$
Precision $\uparrow^{[0,1]}$	0.75 (0.74, 0.75) $p = 0.00$	0.70 (0.69, 0.71) $p = 0.00$	**0.90** **(0.89, 0.90)** $p = 1.00$	**0.90** **(0.90, 0.90)** $p = 0.02$	**0.90** **(0.90, 0.90)** $p = 0.40$	0.89 (0.89, 0.90) $p = 0.02$	0.89 (0.89, 0.89) $p = 0.00$	**0.90** **(0.90, 0.90)** $p = 0.52$	**0.90** **(0.89, 0.90)** $p = 0.43$
Recall $\uparrow^{[0,1]}$	0.64 (0.64, 0.65) $p = 0.00$	0.75 (0.74, 0.76) $p = 0.00$	0.87 (0.87, 0.87) $p = 1.00$	0.87 (0.87, 0.87) $p = 0.02$	0.86 (0.86, 0.86) $p = 0.00$	0.87 (0.87, 0.87) $p = 0.01$	**0.88** **(0.88, 0.88)** $p = 0.00$	0.87 (0.87, 0.87) $p = 0.94$	**0.87** **(0.87, 0.88)** $p = 0.63$
RR $\uparrow^{[0,1]}$ Reciprocal Rank	— (—, —) $p = —$	— (—, —) $p = —$	**0.83** **(0.83, 0.83)** $p = 1.00$	0.74 (0.74, 0.74) $p = 0.00$	0.78 (0.78, 0.79) $p = 0.00$	0.82 (0.82, 0.82) $p = 0.00$	0.80 (0.80, 0.80) $p = 0.00$	**0.83** **(0.83, 0.83)** $p = 0.44$	**0.83** **(0.83, 0.83)** $p = 0.60$
PPL $\downarrow^{[1,\infty]}$ Perplexity	— (—, —) $p = —$	— (—, —) $p = —$	**3.73** **(3.60, 3.86)** $p = 1.00$	6.10 (5.97, 6.23) $p = 0.00$	10.1 (9.19, 11.1) $p = 0.00$	4.08 (3.95, 4.19) $p = 0.00$	7.55 (7.23, 7.92) $p = 0.00$	**3.80** **(3.67, 3.97)** $p = 0.38$	**3.73** **(3.62, 3.89)** $p = 0.97$
$T_{\text{wall}} \downarrow^{[0,\infty]}$ Wall Time	3.99 (3.97, 4.01) $p = 0.00$	85.9 (85.2, 86.7) $p = 0.00$	1.03 (1.03, 1.03) $p = 1.00$	1.12 (1.11, 1.12) $p = 0.00$	1.07 (1.07, 1.07) $p = 0.00$	0.99 (0.99, 1.00) $p = 0.00$	0.98 (0.98, 0.98) $p = 0.00$	**0.90** **(0.90, 0.91)** $p = 0.00$	1.41 (1.40, 1.42) $p = 0.00$

Table 5: **Detailed Model Comparison and Ablation Study.** Bootstrapped 95% CI and AR-p values. Numbers represent mean (PPL: median) over all test sets.

C Training Curves

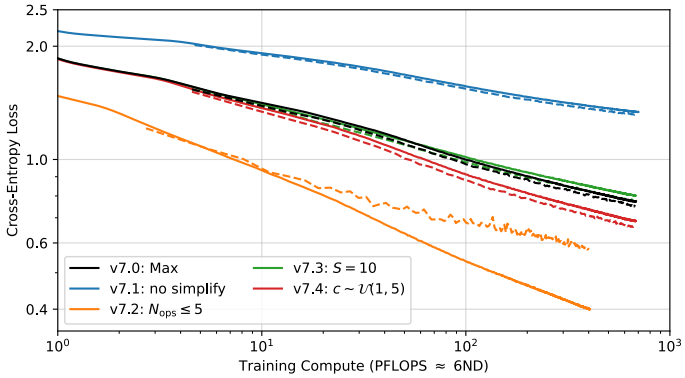

Figure 24: **Training Curves of all Trained Models.** Note that the validation set (*dashed line*) is held out, but sampled from the same distribution as the training data (*solid line*). The amount of compute (*x-axis*) is estimated following (Kaplan et al. 2020). Cross-Entropy values (*y-axis*) are smoothed by convolution with the Hanning window (Blackman and Tukey 1958; NumPy 2024) for visualization with window sizes $W_{\text{train}} = 5001, W_{\text{val}} = 3$.

D Tools

This work is implemented in Python 3.11. I rely on conda to manage the virtual environment, and pip to install dependencies and the main package. Version control is handled by Git and GitHub (student Pro version) for code, and Git LFS and Huggingface (free version) for models, data, and results. I use Weights & Biases to log model performance and system metrics during training. Further system metrics are collected with HWiNFO64 (Pro version).

I use the following AI-assisted tools to assist with software engineering: ChatGPT (free version, up to o3-mini with reasoning), Claude 3.7 Sonnet (free version), Phind (free version, up to Phind-70B) and Phind for VSCode, Kagi Search and Assistant (Ultimate version), GitHub Copilot and GitHub Copilot CLI (both included in student Pro version).

The English abstract was translated to German with DeepL to ensure consistency. Other than that, **no AI-assisted tools were used to directly or indirectly write any part of this work.**

To explore related work, I use ResearchRabbit, Connected Papers (free version), and Google Scholar. This work was written on Typst (Pro version). All compute has been performed on personal hardware (Table 6).

Property	System A	System B
CPU	Ryzen 9 9950X	Ryzen 9 7950X
RAM	DDR5 64GB 6000	DDR5 64GB (48GB WSL) 6000
GPU	NVIDIA RTX 4090	NVIDIA RTX 4090
OS	Ubuntu 24.04.1 LTS	Ubuntu 22.04.5 LTS (WSL2, Windows 11)
conda	24.9.2	23.10.0

Table 6: **Compute systems.**

E Cost

Model	Training Time (h)
⚡ v7.0	86.88
⚡ v7.1	65.50
⚡ v7.2	67.69
⚡ v7.3	82.20
Σ	320.27

Table 7: **Training Time for all Models and Ablations.**

The total recorded compute time of this project on Weights & Biases including trial runs during development is **5808 h** which includes occasional simultaneous training on both compute systems. Assuming an estimated average power consumption of $350\text{W} \pm 10\%$ the energy cost amounts to about **2030 kWh** $\pm$ **10%** for training. All evaluations took approximately $85\,\text{h} \pm 20\%$ (including development) with a reduced power consumption of $250\text{W} \pm 10\%$ on average, resulting in an additional **21.3kWh** $\pm$ **22.4%**.

F Algorithms

FlashANSR.fit(
  X: tensor $\triangleright \in \mathbb{R}^{M \times D}$,
  y: tensor $\triangleright \in \mathbb{R}^{M \times 1}$,
  $\mathcal{E}$: ExpressionSpace,
  w: int $\triangleright$ *beam width*,
  L: int $\triangleright$ *maximum beam length*,
  EPBS: bool $\triangleright$ *use equivalence pruning*,
  r: int $\triangleright$ *optimizer restarts*,
  parsimony: float
):

1 results = [ ]
2 $(\boldsymbol{B}, \log \boldsymbol{p}) \leftarrow \text{beam\_search}(X, y, w, L, \text{EPBS}, \mathcal{E})$
3 $\boldsymbol{B}^{\text{dec}} \leftarrow \{\mathcal{E}.\text{tokenizer.decode}(b) \mid b \in B\}$
4 **for** each pair $\left(b_k^{\text{dec}}, \log p_k\right) \in \left(\boldsymbol{B}^{\text{dec}}, \log \boldsymbol{p}\right)$:
5 **if** $\mathcal{E}.\text{is\_valid}\left(b_k^{\text{dec}}\right)$:
6 $\mathcal{R} \leftarrow \text{Refiner}(\mathcal{E})$
7 $\mathcal{R}.\text{fit}\left(b_k^{\text{dec}}, X, y, r\right)$
8 **if** refiner succeeds:
9 $\text{score} = \mathcal{R}.\text{loss} + \text{parsimony} \cdot \text{len}\left(b_k^{\text{dec}}\right)$
10 $\text{results.append}\left(\left(\mathcal{R}, \left\{b_k^{\text{dec}}, \log p, \text{score}\right\}\right)\right)$
11 $\text{results.sort}(key = \text{score})$
12 **return** results

Algorithm 2: ⚡**ANSR Inference.** Beams of candidate skeletons are generated by conditioning a Transformer decoder on a learned embedding of the data $\mathcal{S} = \{(\boldsymbol{X}_i, y_i)\}_{i=1}^{M}$ and sampling tokens with beam search (*line 2*). The decoded beams $\boldsymbol{B}^{\text{dec}}$ are checked for syntactic validity (*line 5*) before being passed to the "Refiner" class that fits any constants in the expression represented by each beam (*line 7*). The score is calculated according to Equation 11 (*line 9*). Results from all beams are collected and sorted by score (*lines 10, 11*). The first result contains the best fit.

```
EQUIVALENCE PRUNED BEAM SEARCH(
    X: tensor ▷ ∈ ℝ^{M×D},
    y: tensor ▷ ∈ ℝ^{M×1},
    ℰ: ExpressionSpace,
    w: int ▷ beam width,
    L: int ▷ maximum sequence length,
    m: int ▷ mini-batch size,
    EPBS: bool ▷ use equivalence pruning
):
 1  B = { ([ℰ.tokenizer.encode(<bos>)], 0) }     ▷ Initialize Beam with <bos>
 2  B_completed = [ ]
 3  for l in 1..L:
 4      Z_l, S_l = [ ], [ ]     ▷ Lists that store incomplete beams & their scores
 5      ▷ Check whether any beam is complete
 6      for (b, s) in B:
 7          if b[−1] = ℰ.tokenizer.encode(<eos>):
 8              B_completed.append((b, s))
 9          else:
10              Z_l.append(b)
11              S_l.append(s)
12      if Z_l = [ ]: break     ▷ All beams are complete
13      ▷ Predict Next-Token Probabilities
14      p = TRF(B, SetTRF(X, y)) ▷ ∈ ℝ^w
15      ▷ Create candidates by combining every beam with every token of the
        Vocabulary
16      C ← { ( b ‖ [t], s + log(p_t) ) ∀t ∈ V ∀(b, s) ∈ (Z_l, S_l) }
17      C.sort(key = 1)
18      if EPBS:
19          ▷ Simplify newly completed beams & deduplicate
20          F = {(ℰ.simplify(c), s) for (c, s) ∈ C with (c[−1] =
            ℰ.tokenizer.encode(<eos>)) and ℰ.simplify(c) ∉ B_completed)}
21          B ← F[: w]
22      else:
23          B ← C[: w]
24      if B = [ ]: break     ▷ all beams completed
25  B_completed.append(B)     ▷ Append all remaining beams
26  return top_k_sequences(B_completed, k = w)
```

Algorithm 3: **Equivalence-Pruned Beam Search (EPBS).** Equivalence-Pruning immediately simplifies and deduplicates newly completed beams (*line 20*) that are generated with the typical beam-search algorithm.

G Simplification Rules

Algorithm 1 relies on a list of simplification rules that specify which algebraic terms are to be replaced by functionally equivalent but simpler terms. Rules were found semi-automatically by exhaustively constructing expression trees Z from least complex to more complex ones and recording their image $f_Z(X; \boldsymbol{c})$ on a fixed random input set X with fixed randomly sampled constants $\boldsymbol{c}$. If two expressions Z_i and Z_j yield the same image $f_{Z_i}(X; \boldsymbol{c}) = f_{Z_j}(X; \boldsymbol{c})$ and one of the expressions is shorter (i.e. has less tokens) than the other, say w.l.o.g $|Z_j| < |Z_i|$, I take note of patterns in Z_i and formulate a simplification rule that is as general as possible and transforms $Z_i \to Z_j$.

Furthermore, a special rule sorts operands of commutative binary operators and of other compatible tree structures by their length. Operands whose representation as an expression has fewer tokens are moved to appear before more complex operands, e.g.

$$\texttt{[+ , * , <num> , x2 , x1]} \to \texttt{[+ , x1 , * , <num> , x2]} \tag{34}$$

I hypothesize that the autoregressive generation of expression benefits from early commitments to generate finished subtrees that serve as a more informative condition on later subtrees than a long stack of operator tokens at the beginning followed by many leaf node tokens at the end.

In the following tables, I present the rules I found with this procedure. I define the following symbols for various special operators and operands:

- $\texttt{Op}^{+}$: Positive operator
- $\texttt{Op}^{s}$: Symmetric operator
- $\texttt{Op}^{a}$: Anti-symmetric operator
- $\texttt{Op}^{a\uparrow}$: Antisymmetric monotonous increasing operator
- $\texttt{Op}^{c}$: Commutative operator
- `N`: Numeric operand (i.e. an undefined constant)
- `L`: Numeric literal (e.g. -1.234)
- `A, B, C, D`: Any finite operand

Rule	Input		Output
1	[Op, N, N]	→	[N]
2	[Op, L_1, L_2]	→	[Op(L_1, L_2)]
3	[*, A, A]	→	[pow2, A]
4	[*, abs, A, abs, B]	→	[abs, *, A, B]
5	[*, A, neg, A] [*, neg, A, A]	→ →	[neg, pow2, A] [neg, pow2, A]
6	[*, A, +, B, B]	→	Consider alternative [*, B, +, A, A]
7	[*, A, +, A, A]	→	[+, pow2, A, pow2, A]
8	[*, -, A, B, -, B, A]	→	[neg, pow2, -, A, B]
9	[*, -, A, B, -, C, D]	→	Consider alternatives [*, -, B, A, -, D, C] [*, -, C, D, -, A, B] [*, -, D, C, -, B, A]
10	[/, A, 1]	→	[A]
11	[/, 1, A]	→	[inv, A]
12	[/, abs, A, A]	→	[/, A, abs, A]
13	[/, +, A, A, +, B, B]	→	[/, A, B]
14	[/, -, A, B, -, B, A]	→	[neg, <1>]
15	[/, -, A, B, -, C, D]	→	Consider alternative [/, -, B, A, -, D, C]
16	[+, A, 0] [+, 0, A]	→	[A]
17	[+, *, A, B, *, A, C]	→	[*, A, +, B, C]
18	[+, /, A, B, /, C, B]	→	[/, +, A, C, B]
19	[+, abs, A, abs, A]	→	[abs, +, A, A]
20	[-, A, 0]	→	[A]
21	[-, 0, A]	→	[neg, A]
22	[-, *, A, B, *, A, C]	→	[*, A, -, B, C]
23	[-, /, A, B, /, C, B]	→	[/, -, A, C, B]
24	[*, -, A, B, neg, C] [/, -, A, B, neg, C]	→ →	[*, -, B, A, C] [/, -, B, A, C]
25	[*, A, neg, B] [/, A, neg, B]	→ →	[neg, *, A, B] [neg, /, A, B]

Rule	Input		Output
26	[*, neg, C, -, A, B] [/, neg, C, -, A, B]	→ →	[*, C, -, B, A] [/, C, -, B, A]
27	[*, neg, A, B] [/, neg, A, B]	→ →	[neg, *, A, B] [neg, /, A, B]
28	[/, abs, A, abs, B] [*, abs, A, abs, B] [+, abs, A, abs, B]	→ →	[abs, /, A, B] [abs, *, A, B] [abs, +, A, B]
29	[inv, exp, A]	→	[exp, neg, A]
30	[exp, +, A, A]	→	[pow2, exp, A]
31	[pow1_<i>, pow<i>, A] with even integer i	→	[abs, A]
32	[pow<i>, inv, A] with integer i	→	[inv, pow<i>, A]
33	Chain of Power Operators	→	Contracted Chain
34	[abs, $\mathrm{Op}^{\star}$, A]	→	[$\mathrm{Op}^{\star}$, A]
35	[abs, inv, A]	→	[inv, abs, A]
36	[$\mathrm{Op}^{a\uparrow}$, abs, A]	→	[abs, $\mathrm{Op}^{a\uparrow}$, A]
37	[Op^{s}, neg, A]	→	[Op^{s}, A]
38	[Op^{s}, -, A, B]	→	Consider [Op^{s}, -, B, A]
39	[Op^{s}, abs, A]	→	[Op^{s}, A]
40	[inv, inv, A] [neg, neg, A]	→	[A]
41	[Op^{a}, neg, A]	→	[neg, Op^{a}, A]
42	[neg, $\mathrm{Op}^{a}_{1:k}$, *, -, A, B, C] [neg, $\mathrm{Op}^{a}_{1:k}$, /, -, A, B, C]	→ →	[$\mathrm{Op}^{a}_{1:k}$, *, -, B, A, C] [$\mathrm{Op}^{a}_{1:k}$, /, -, B, A, C]
43	[neg, $\mathrm{Op}^{a}_{1:k}$, *, A, -, B, C] [neg, $\mathrm{Op}^{a}_{1:k}$, /, A, -, B, C]	→ →	[$\mathrm{Op}^{a}_{1:k}$, *, A, -, C, B] [$\mathrm{Op}^{a}_{1:k}$, /, A, -, C, B]
44	[neg, Op^{a}, -, A, B]	→	[Op^{a}, -, B, A]
45	[*, inv, A, inv, B] [+, neg, A, neg, B]	→ →	[inv, *, A, B] [neg, +, A, B]
46	[*, /, A, B, /, B, A] [+, -, A, B, -, B, A]	→ →	[<1>] [<0>]

Rule	Input		Output
47	`[*, inv, A, /, B, C]` `[+, neg, A, -, B, C]`	→ →	`[/, /, B, A, C]` `[-, -, B, A, C]`
48	`[*, inv, A, B]` `[+, neg, A, B]`	→ →	`[/, B, A]` `[-, B, A]`
49	`[* /, B, A, A]` `[* /, B, A, A]`	→	`[B]`
50	`[*, A, inv, B]` `[+, A, neg, B]`	→ →	`[/, A, B]` `[-, A, B]`
51	`[+, A, -, B, A]` `[*, A, /, B, A]`	→ →	`[B]`
52	`[+, A, +, B, -, B, A]` `[*, A, *, B, /, B, A]`	→ →	`[+, B, B]` `[*, B, B]`
52	`[+, A, +, B, -, B, A]` `[*, A, *, B, /, B, A]`	→ →	`[+, B, B]` `[*, B, B]`
53	`[+, A, -, B, C]` `[*, A, /, B, C]`	→ →	Consider `[+, B, -, A, C]` Consider `[*, B /, A, C]`
54	`[/, *, A, B, A]` `[-, +, A, B, A]`	→	`[B]`
55	`[/, *, A, B, C]` `[-, +, A, B, C]`	→ →	`[*, A, /, B, C]` `[+, A, -, B, C]`
56	`[/, /, A, B, C]` `[-, -, A, B, C]`	→ →	Consider `[/, /, A, C, B]` Consider `[-, -, A, C, B]`
57	`[/, inv, A, B]` `[-, neg, A, B]`	→ →	`[inv, *, A, B]` `[neg, +, A, B]`
58	`[/, A, inv, B]` `[-, A, neg, B]`	→ →	`[*, A, B]` `[+, A, B]`
59	`[/, A, *, A, B]` `[-, A, +, A, B]`	→ →	`[inv, B]` `[neg, B]`
60	`[/, A, /, B, C]` `[-, A, -, B, C]`	→ →	`[*, A, /, C, B]` `[+, A, -, C, B]`

Rule	Input		Output
61	`[-, A, A]` `[/, A, A]`	→ →	`[<0>]` `[<1>]`
62	`[inv, /, A, B]` `[neg, -, A, B]`	→ →	`[/, B, A]` `[-, B, A]`
63	`[inv, *, A, /, B, C]` `[neg, +, A, -, B, C]`	→ →	`[/, /, C, B, A]` `[-, -, C, B, A]`
64	`[*, *, A, B, C]` `[+, +, A, B, C]`	→ →	`[*, A, *, B, C]` `[+, A, +, B, C]`
65	`[Op`^{c}`, A, B]`	→	Consider `[Op`^{c}`, B, A]`

H Decontamination Methods

Model	$\%_{\mathrm{FVU} \le \varepsilon}$ $\uparrow^{[0,100]}$		$\log \mathrm{FVU}_{>\varepsilon}$ $\downarrow^{[-6.92,\infty]}$		ZSS $\downarrow^{[0,\infty]}$	PPL $\downarrow^{[1,\infty]}$	$\mathrm{T}_{\mathrm{wall}}$ [s] $\downarrow^{[0,\infty]}$
-	Fit	Val	Fit	Val	-	-	-
⚡v7.0	**64.5** **(63.7, 65.1)** p = 1.00	**64.3** **(63.7, 65.0)** p = 1.00	−2.97 (−3.02, −2.92) p = 1.00	**−2.80** **(−2.85, −2.74)** p = 1.00	**13.3** **(13.1, 13.4)** p = 1.00	**3.73** **(3.60, 3.86)** p = 1.00	1.03 (1.03, 1.03) p = 1.00
⚡v7.0-d	**64.3** **(63.6, 64.9)** p = 0.63	**64.0** **(63.4, 64.7)** p = 0.62	−2.84 (−2.89, −2.80) p = 0.00	−2.68 (−2.73, −2.63) p = 0.00	**13.4** **(13.2, 13.5)** p = 0.49	4.74 (4.60, 4.89) p = 0.00	**0.66** **(0.66, 0.66)** p = 0.00

Table 12: **Exact-Match-Rejection ⚡v7.0 & Family-Match-Rejection ⚡v7.0-d.** Both decontamination approaches result in the same Numeric Recovery Rate on the test sets, but rejecting expressions whose structure without constants matches test expressions stripped of their constants increases the Approximation Error by $\frac{10^{-2.84}}{10^{-2.97}} - 1 \approx 35\%$ (Δ FVU $= 0.0004$ in absolute terms) and increases the median test perplexity by about 1. Bootstrapped 95% CI and AR-p values. Numbers represent mean (PPL: median) over all test sets.

I Test Set Comparison

In Table 13 and Table 14, I compare the Numeric Recovery Rate (Section 3.6.2) and the ZSS Tree-Edit Distance (Zhang and Shasha 1989) of all models across all test sets.

Model	Train	Val	Feynman	S.-NC	Nguyen	Pool-15
PySR	— (—, —) p = —	— (—, —) p = —	69.2 (66.3, 72.1) p = 0.00	60.6 (57.6, 63.5) p = 0.00	**79.3** **(76.8, 81.9)** p = 0.01	**35.4** **(32.4, 38.4)** p = 0.53
NeSymReS 100M	— (—, —) p = —	— (—, —) p = —	51.4 (48.5, 54.6) p = 0.00	26.4 (23.7, 29.2) p = 0.00	54.2 (51.1, 57.4) p = 0.00	11.3 (9.40, 13.3) p = 0.00
⚡ v7.0	56.0 (54.7, 57.4) p = 1.00	56.6 (55.2, 57.9) p = 1.00	**75.9** **(74.7, 77.1)** p = 1.00	**70.8** **(69.5, 72.0)** p = 1.00	74.9 (73.6, 76.1) p = 1.00	**36.3** **(34.9, 37.6)** p = 1.00
⚡ v7.1 no simplify	46.8 (45.4, 48.2) p = 0.00	47.6 (46.3, 48.9) p = 0.00	70.1 (68.8, 71.4) p = 0.00	66.2 (64.9, 67.5) p = 0.00	73.1 (72.0, 74.3) p = 0.05	33.0 (31.7, 34.2) p = 0.00
⚡ v7.2 $N_{\text{ops}} \leq 5$	**84.3** **(83.3, 85.4)** p = 0.00	**86.9** **(86.0, 87.9)** p = 0.00	69.6 (68.3, 70.9) p = 0.00	**70.7** **(69.5, 71.9)** p = 0.95	67.0 (65.7, 68.3) p = 0.00	29.3 (28.1, 30.6) p = 0.00
⚡ v7.3 $S = 10$	51.6 (50.3, 52.9) p = 0.00	54.1 (52.7, 55.5) p = 0.01	**75.0** **(73.8, 76.2)** p = 0.29	69.7 (68.4, 71.0) p = 0.25	76.2 (75.0, 77.3) p = 0.16	**34.1** **(32.8, 35.5)** p = 0.03
⚡ v7.4 $c \sim \mathcal{U}(1, 5)$	57.6 (56.2, 59.0) p = 0.12	60.8 (59.4, 62.1) p = 0.00	**76.7** **(75.5, 77.8)** p = 0.36	**73.0** **(71.8, 74.2)** p = 0.02	**79.5** **(78.4, 80.6)** p = 0.00	**36.4** **(35.1, 37.7)** p = 0.99
⚡ v7.5 no EPBS	55.1 (53.8, 56.4) p = 0.38	55.8 (54.5, 57.2) p = 0.45	**75.8** **(74.7, 77.0)** p = 1.00	**72.7** **(71.5, 73.9)** p = 0.04	73.3 (72.0, 74.5) p = 0.08	**36.0** **(34.7, 37.3)** p = 0.74
⚡ v7.7 BFGS	62.7 (61.3, 64.0) p = 0.00	65.6 (64.3, 66.8) p = 0.00	**76.9** **(75.7, 78.1)** p = 0.30	**72.8** **(71.6, 74.1)** p = 0.04	71.8 (70.5, 73.0) p = 0.00	**35.9** **(34.5, 37.1)** p = 0.60

Table 13: **Numeric Recovery Rate in $\%_{\text{FVU} \leq \varepsilon} \uparrow^{[0,100]}$ with Default Configurations.** The existing test sets (Feynman, SOOSE-NC, Nguyen; Section 3.3) can be perfectly solved to about 75% with state-of-the-art models (PySR (Cranmer 2023), ⚡ANSR) when enforcing user-friendly default compute limits. The proposed hard test set Pool-15 reveals that all models struggle to perfectly identify complex equations in the given compute limit. Bootstrapped 95% CI and AR-p values. Training and validation sets are model-specific.

Model	Train	Val	Feynman	S.-NC	Nguyen	Pool-15
PySR	— (—, —) p = —	— (—, —) p = —	25.74 (24.56, 27.10) p = 0.00	27.05 (25.93, 28.18) p = 0.00	27.14 (25.98, 28.30) p = 0.00	39.14 (38.21, 40.12) p = 0.00
NeSymReS 100M	— (—, —) p = —	— (—, —) p = —	18.37 (17.52, 19.18) p = 0.00	21.65 (21.05, 22.31) p = 0.00	20.74 (19.82, 21.70) p = 0.00	32.49 (31.81, 33.18) p = 0.00
⚡ v7.0	14.34 (14.07, 14.61) p = 1.00	14.05 (13.80, 14.31) p = 1.00	**11.09** **(10.84, 11.35)** p = 1.00	7.65 (7.41, 7.89) p = 1.00	**12.48** **(12.13, 12.78)** p = 1.00	**21.89** **(21.54, 22.24)** p = 1.00
⚡ v7.1 no simplify	22.38 (22.13, 22.64) p = 0.00	22.23 (21.98, 22.50) p = 0.00	11.44 (11.15, 11.71) p = 0.07	7.71 (7.47, 7.96) p = 0.68	**12.03** **(11.70, 12.37)** p = 0.07	22.49 (22.18, 22.85) p = 0.02
⚡ v7.2 $N_{\text{ops}} \leq 5$	**6.49** **(6.32, 6.68)** p = 0.00	**6.37** **(6.21, 6.54)** p = 0.00	11.29 (11.04, 11.56) p = 0.27	**6.15** **(5.95, 6.36)** p = 0.00	**11.95** **(11.62, 12.26)** p = 0.02	22.91 (22.57, 23.25) p = 0.00
⚡ v7.3 $S = 10$	15.14 (14.88, 15.40) p = 0.00	14.48 (14.22, 14.76) p = 0.03	11.40 (11.13, 11.67) p = 0.11	7.84 (7.63, 8.08) p = 0.22	**12.54** **(12.20, 12.86)** p = 0.75	**22.16** **(21.81, 22.48)** p = 0.29
⚡ v7.4 $c \sim \mathcal{U}(1, 5)$	13.68 (13.41, 13.97) p = 0.00	13.42 (13.14, 13.69) p = 0.00	11.89 (11.65, 12.18) p = 0.00	7.39 (7.16, 7.63) p = 0.14	**11.96** **(11.66, 12.26)** p = 0.02	22.95 (22.61, 23.28) p = 0.00
⚡ v7.5 no EPBS	14.55 (14.28, 14.82) p = 0.27	14.28 (13.98, 14.56) p = 0.25	**11.20** **(10.93, 11.45)** p = 0.59	7.46 (7.23, 7.69) p = 0.27	12.63 (12.32, 12.93) p = 0.55	**22.17** **(21.84, 22.51)** p = 0.28
⚡ v7.7 BFGS	12.17 (11.90, 12.43) p = 0.00	11.48 (11.21, 11.73) p = 0.00	**10.72** **(10.46, 10.97)** p = 0.06	7.15 (6.93, 7.39) p = 0.01	**12.51** **(12.18, 12.85)** p = 0.84	**21.62** **(21.29, 21.93)** p = 0.25

Table 14: **ZSS Edit Distance $\downarrow^{[0,\infty]}$ with Default Configurations.** The edit distances of the ⚡v7.x model family is much smaller on the SOOSE-NC test set by Biggio et al. (2021) than on the other sets, likely due to the shared use of the LC-Algorithm (Lample and Charton 2019) for the generation of skeletons in the training ⚡v7.x models and the creation of SOOSE-NC. Surprisingly, NeSymReS does not exhibit this bias. Bootstrapped 95% CI and AR-p values. Training and validation sets are model-specific.

J Cumulative Distributions

For maximum transparency, Figure 25 contains cumulative distributions of a selection of metrics across all models and test sets.

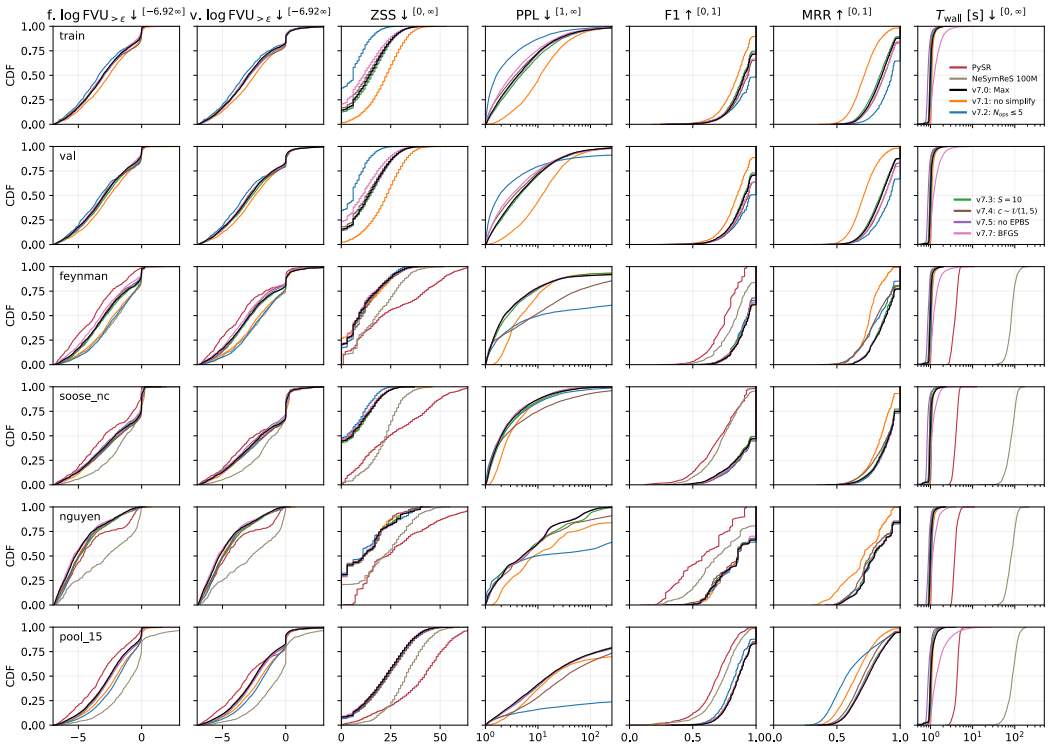

Figure 25: **Cumulative Distributions of Various Metrics for all Tested Models.** The approximation error is divided into fit (*first column*) and validation (*second column*) data splits of each instance. Training on short expressions (⚡v7.2, *blue curve*) reduces the perplexity (*PPL*) on training and validation expressions, but severely shifts the distribution towards large values on the Feynman and Pool-15 test sets that include longer expressions. The F1-distributions of both PySR and NeSymReS have a clear tail starting at around 0.25 while the ⚡v7.x model family overwhelmingly scores above 0.5. NeSymReS, PySR and ⚡v7.1 have a wall time distribution that is significantly larger than the other models. MRR denotes the Mean Reciprocal Rank of ground truth tokens in the predicted logits.

K Operator Embeddings

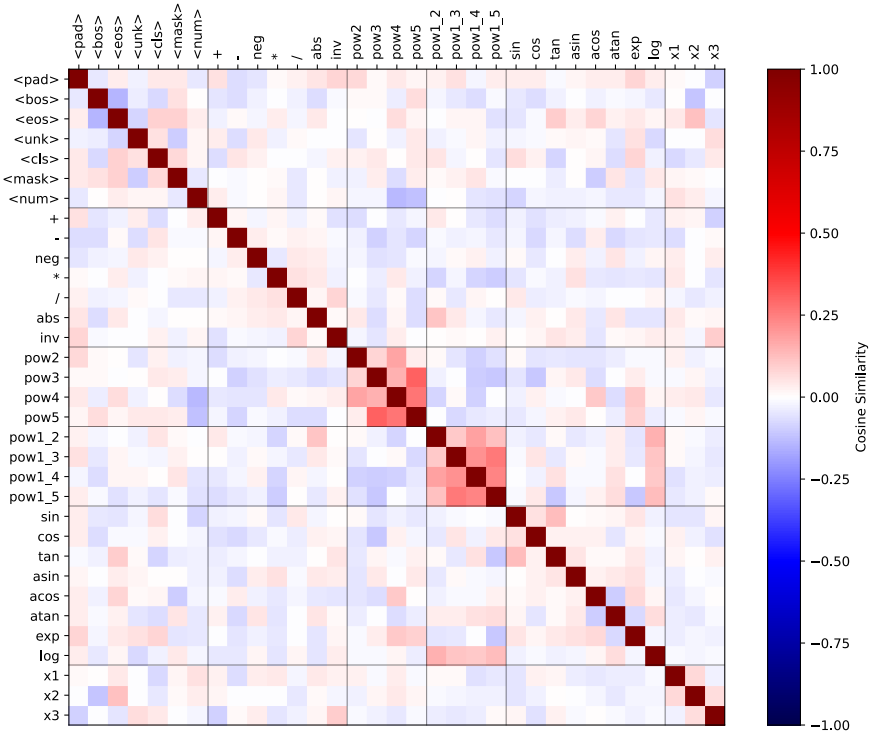

Figure 26: **Pairwise Cosine Similarity of Tokens in the ⚡ANSR Vocabulary after Training ⚡v7.0 .** The embeddings of power and root operators are comparatively similar, respectively. Root operator embeddings also show systematic similarity with the logarithm operator.

Acknowledgements

I would like to thank my dear friend Neelkanth Rawat for proofreading and providing very helpful critique of my work. A big thank you also goes to my supervisor Prof. Ullrich Köthe for his endless stream of ideas and insights and exceptional lectures on machine learning which inspired me to work in this field.

Thank you Prof. Artur Andrzejak, and my dear friends Ana Carsi and Kushal Gaywala with whom I have had the great pleasure of doing a practical in feedback-based code generation. You have taught me valuable lessons and I sincerely enjoyed our joint work.

I thank Prof. Anette Frank for her amazing lectures on Formal Semantics and Formal Syntax. No other person could have better prepared me to work in areas related to Natural Language Processing with Transformers. To this day, I refer to your lecture material when discussing NLP concepts or pondering about the wondrous patterns of language.

❤