Discovering ordinary differential equations that govern time-series

Anonymous Author(s)
Affiliation
Address
email

Abstract

Natural laws are often described through differential equations yet finding a differential equation that describes the governing law underlying observed data is a challenging and still mostly manual task. In this paper we make a step towards the automation of this process: we propose a transformer-based sequence-to-sequence model that recovers scalar autonomous ordinary differential equations (ODEs) in symbolic form from time-series data of a single observed solution of the ODE. Our method is efficiently scalable: after one-time pretraining on a large set of ODEs, we can infer the governing laws of a new observed solution in a few forward passes of the model. Then we show that our model performs better or on par with existing methods in various test cases in terms of accurate symbolic recovery of the ODE, especially for more complex expressions.

1 Introduction

Science is commonly described as the "discovery of natural laws through experimentation and observation". Researchers in the natural sciences increasingly turn to machine learning (ML) to aid the discovery of natural laws from observational data alone, which is often abundantly available, hoping to bypass expensive and cumbersome targeted experimentation. While there may be fundamental limitations to what can be extracted from observations alone, recent successes of ML in the entire range of natural sciences provide ample reason for excitement. In this work, we focus on ordinary differential equations, a ubiquitous description of dynamical natural laws in physics, chemistry, and systems biology. For a first order ODE $\dot{y} := {}^{\partial y}/_{\partial t} = f(y, t)$, we call f (which uniquely defines the ODE) the underlying dynamical law. Informally, our goal is then to infer f in symbolic form given discrete time-series observations of a single solution $\{y_i := y(t_i)\}_{i=1}^n$ of the underlying ODE.

Contrary to "black-box-techniques" such as Neural Ordinary Differential Equations (NODE) [Chen et al., 2018] that aim at inferring a possible f as an arguably opaque neural network, we focus specifically on symbolic regression. From the perspective of the sciences, a law of nature is useful insofar as it is more broadly applicable than to merely describe a single observation. In particular, the reason to learn a dynamical law in the first place is to dissect and understand it as well as to make predictions about situations that differ from the observed one. From this perspective, a symbolic representation of the law (in our case the function f) has several advantages over block-box representations: they are compact and directly interpretable, they are amenable to analytic analysis, they allow for meaningful changes and thus enable assessment of interventions and counterfactuals.

In this work we present NSODE, a sequence-to-sequence transformer that maps observed trajectories, i.e., numeric sequences of the form $\{(t_i, y_i)\}_{i=1}^n$, directly to symbolic equations as strings, e.g.,

Submitted to NeurIPS 2021 AI for Science Workshop.

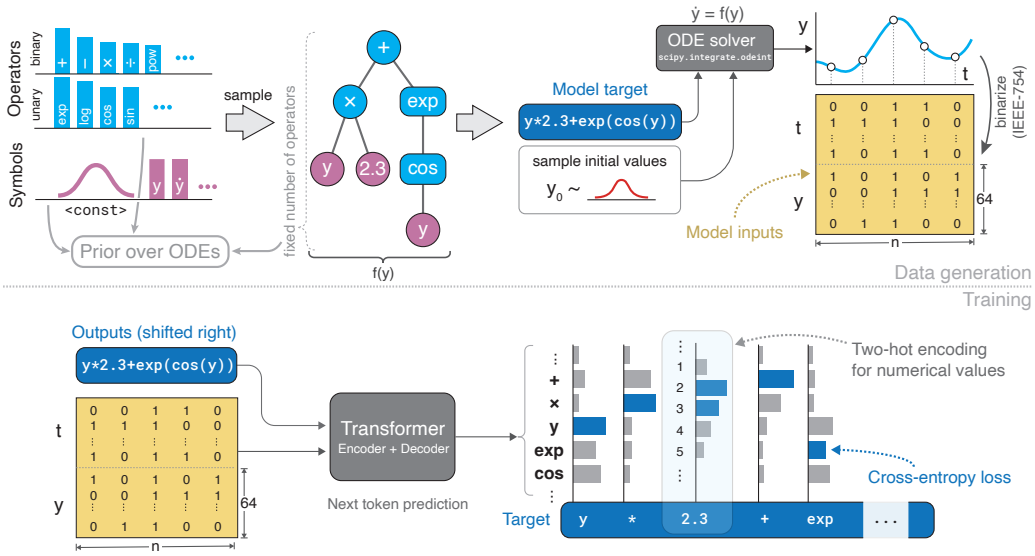


Figure 1: An overview illustration of the data generation (top) and training pipeline (bottom). Our dataset stores solutions in numerical (non-binarized) form on the entire regular solution time grid.

`"y**2+1.64*cos(y)"`, which is the prediction for f. This example directly highlights the benefit of symbolic representations in that the y^2 and $\cos(y)$ terms tell us something about the fundamental dynamics of the observed system; the constant `1.64` will have semantic meaning in a given context and we can, for example, make predictions about settings in which this constant takes a different value.

2 Background and Related Work

While NODE [Chen et al., 2018] (with a large body of follow up work) is perhaps the most prominent method to learn ODEs from data in black-box form, we focus on various works that infer governing laws in symbolic form. Classically, symbolic regression aims at regular functional relationships (mapping $(x, f(x))$ pairs to f instead of mapping trajectories $(t, y(t))$ to the governing ODE $\dot{y} = f(y, t)$) and has been approached by heuristics-based search, most prominently via genetic programming [Koza, 1993]. Genetic programming randomly evolves a population of prospective mathematical expressions over many iterations and mimics natural selection by keeping only the best contenders across iterations, where superiority is measured by user-defined and problem-specific fitness functions. More recently, symbolic regression has been approached with machine learning methods which exploit gradient information to optimize within the space of (finite) compositions of pre-defined basis functions. Brunton et al. [2016] use linear regression to identify a (sparse) linear combination of basis functions that yields the best fit for the observed data, while other approaches use neural networks with a diverse set of activation functions [Sahoo et al., 2018, Long et al., 2019, Liu et al., 2020]. All these techniques deploy strong sparsity-promoting regularizers and fit a separate model for each observed trajectory.

Alternatively, one can train a model to directly output the symbolic expressions. Supervised learning with gradient-based optimization for this approach is challenged by the formulation of a differentiable loss that measures the fit between the predicted symbolic expression and the observed data. Thus, prior work resorted to reinforcement learning [Petersen et al., 2021] or evolutionary algorithms [Atkinson et al., 2019, Costa et al., 2021] for gradient-free optimization. Furthermore, inspired by common properties of known natural laws, Udrescu et al. [2020] devise a hybrid approach that combines a gradient-free heuristic search with neural network-based optimization. This approach has been extended to work with dynamical systems by Weilbach et al. [2021].

The closest works to ours use pre-trained, attention-based sequence-to-sequence models for symbolic regression *of functional relationships* [Biggio et al., 2021, Valipour et al., 2021, Kamienny et al.,

2022, Vastl et al., 2022]. They exploit the fact that symbolic expressions for (multi-variate) scalar functions can be both generated and evaluated on random inputs cheaply, resulting in essentially unlimited training data. Large data including ground-truth expressions in symbolic form allow for a differentiable cross-entropy loss based directly on the symbols of the expression, instead of the numerical proximity between evaluations of the predicted and true expression. While the cross-entropy loss works well for operators and symbols (e.g. `+,exp,sin,x,y`), a naive implementation is inefficient for numerical constants, e.g., `1.452`. Previous works therefore resort to one of two strategies: 1) represent all constants with a special `<const>` token when training the sequence-to-sequence model and predict only the presence of a constant. Actual values are then inferred in a second, subsequent parameter estimation step where the structure of an expression is held fixed and only constants are optimized. This second optimization procedure comes with substantial computational cost as constants have to be fit per inferred expression. In particular, we highlight that it does not transfer to inferring ODEs as it would require to first solve the predicted ODE $\dot{y} = \hat{f}(y)$ to obtain predicted $\{\hat{y}_i\}_{i=1}^n$ values that can be compared to the set of observations $\{y_i\}_{i=1}^n$. While differentiable ODE solvers exist, optimizing constants this way is prohibitively expensive and typically highly unstable. 2) A second popular strategy consists in rounding constants within the range of interest so that they can be represented with a finite number of tokens. This second strategy avoids a subsequent optimization step and enjoys clever encoding schemes with improved token efficiency yet represents values with an inherent loss in precision. As an alternative, we develop a representation based on a "two-hot" encoding which avoids subsequent optimization steps as well as rounding.

3 Method

Problem setting. Given observations $\{(t_i, y_i)\}_{i=1}^n$ of a trajectory $y : [t_1, t_n] \to \mathbb{R}$ that is a solution of the ODE $\dot{y} = f(y)$, we aim to recover the function f in symbolic form—in our case as a string. In this work, we focus on time-invariant (or autonomous) ODEs (i.e., $f(y, t) = f(y)$). Such settings are a good starting point for investigation as they are commonly studied and can be thought of as "evolving on their own" without external driving forces or controls, i.e., once an initial condition is fixed the absolute time does not directly influence the evolution. We explicitly assume that the observed system actually evolves according to an ODE in canonical form $\dot{y} = f(y)$ such that f can be expressed in closed form using the mathematical operators seen during training (see Section 3.1). In this paper we restrict ourselves to the rich class of non-linear, scalar, first-order, autonomous ODEs but we discuss extensions of NSODE to higher-order systems of coupled non-autonomous ODEs in Appendix A.

3.1 Data Generation

Sampling symbolic expressions. To exploit large-scale supervised pretraining we generate a dataset of $\sim$63M ODEs in symbolic form along with numerical solutions for randomly sampled initial values. Since we assume ODEs to be in canonical form $\dot{y} = f(y)$, generating an ODE is equivalent to generating a symbolic expression $f(y)$. We follow Lample and Charton [2019], who sample such an expression $f(y)$ as a unary-binary tree, where each internal node corresponds to an operator and each leaf node corresponds to a constant or variable. The algorithm consists of two phases: (1) A unary-binary tree is sampled uniformly from the distribution of unary-binary trees with up to $k \in \mathbb{N}$ internal nodes, which crucially does not overrepresent small trees corresponding to short expressions. Here the maximum number of internal nodes K is a hyperparameter of the algorithm. (2) The sampled tree is "decorated", that is, each binary internal node is assigned a binary operator, each unary internal node is assigned a unary operator, and each leaf is assigned a variable or constant. Hence, we have to specify a distribution over the N_{bin} binary operators, one over the N_{una} unary operators, a probability $p_{\text{sym}} \in (0, 1)$ to decide between symbols and constants, as well as a distribution p_{c} over constants. For constants we distinguish explicitly between sampling an integer or a real value. Together with K, these choices uniquely determine a distribution over equations f and are described in detail in Appendix B. Figure 1 depicts an overview of the data generation procedure.

The pre-order traversal of a sampled tree results in the symbolic expression for f in prefix notation. After conversion to infix notation, we simplify each expression using the computer algebra system SymPy [Meurer et al., 2017], and filter out constant equations $f(y) = c$ as well as expressions that contain operators or symbols that were not part of the original distribution. We call the structure modulo the value of the constants of such an expression a **skeleton**. Any skeleton containing at least one binary operator or constant can be represented by different unary-binary trees. Vice versa many of the generated trees will be simplified to the same skeleton. To ensure diversity and to mitigate potential dataset bias towards particular expressions, we discard duplicates on the skeleton level. To further cheaply increase the variability of ODEs we sample N_{const} unique sets of constants per skeleton. When sampling constants we take care not to modify the canonical expression by adhering to the rules listed in Appendix B.1. We provide summary statistics on operator frequencies and expression complexities for the generated dataset in Appendix C. Here, **complexity** refers to overall count of symbols (e.g., y, or constants) as well as operators in an expression, a simple yet common measure in the symbolic regression literature.

Computing numerical solutions. We obtain numerical solutions for all ODEs via SciPy's interface [Virtanen et al., 2020] to the LSODA software package [Hindmarsh and Laboratory, 1982] with both relative and absolute tolerances set to 10^{-9}. We solve each equation on a fixed time interval $t \in [0, T]$ and store solutions on a regular grid of N_{grid} points. For each ODE, we sample up to N_{iv} initial values $y(0) = y_0$ uniformly from $(y_0^{\min}, y_0^{\max})$.[1] While LSODA attempts to select an appropriate solver, numerical solutions still cannot be trusted in all cases. Therefore, we check the validity of solutions via the following quality control check: we use 9th order central finite differences to approximate the temporal derivative of the solution trajectory (on the same temporal grid as the proposed solution), denoted by $\dot{y}_{\text{fd}}$, and filter out any solution for which $\|\dot{y}_{\text{fd}} - \dot{y}\|_\infty > \epsilon$, where we use $\epsilon = 1$.

3.2 Model Design Choices

NSODE consists of an encoder-decoder transformer with architecture choices listed in Appendix B.3. We provide a visual overview in Figure 1.

Representing input trajectories. A key difficulty in feeding numerical solutions $\{y_i\}_{i=1}^n$ as input is that their range may differ greatly both within a single solution as well as across ODEs. For example, the linear ODE $\dot{y} = c \cdot y$ for a constant c is solved by an exponential $y(t) = y_0 \exp(ct)$ for initial value $y(0) = y_0$, which may span many orders of magnitude on a fixed time interval. To prevent numerical errors and vanishing or exploding gradients caused by the large range of values, we assume each representable 64-bit float value is a token and use its IEEE-754 encoding as the token representation [Biggio et al., 2021]. We thus convert all pairs (t_i, y_i) to their IEEE-754 64 bit representations, channel them through a linear embedding layer before feeding them to the encoder.

Representing symbolic expressions. The target sequence (i.e., the string for the symbolic expression of f) is tokenized on the symbol-level. We distinguish two cases: (1) *Operators and variables:* for each operator and variable we include a unique token in the vocabulary. (2) *Numerical constants:* constants may come from both discrete (integers) as well as continuous distributions, as for example in `y**2+1.64*cos(y)`. Hence, it is unfeasible to include individual tokens "for each constant". Naively tokenizing on the digit level, i.e., representing real values literally as the sequence of characters (e.g., `"1.64"`), not only significantly expands the length of target sequences and thus the computational cost, but also requires a variable number of prediction steps for every single constant.

Instead, we take inspiration from Schrittwieser et al. [2020] and encode constants in a *two-hot* fashion. We fix a finite homogeneous grid on the real numbers $x_1 < x_2 < \ldots < x_m$ for some $m \in \mathbb{N}$, which we add as tokens to the vocabulary. The grid range (x_1, x_m) and number of grid points m are hyperparameters that can be set in accordance to the problems of interest. Our choices are described in Appendix B.3.

[1] Due to a timeout per ODE, fewer solutions may remain in cases when the numerical solver fails repeatedly.

For any constant c in the target sequence we then find $i \in \{1, \ldots, m-1\}$ such that $x_i \leq c \leq x_{i+1}$ and encode c as a distribution supported on x_i, x_{i+1} with weights α, β such that $\alpha x_i + \beta x_{i+1} = c$. That is, the target in the cross-entropy loss for a constant token is not a strict one-hot encoding, but a distribution supported on two (neighboring) vocabulary tokens resulting in a lossless encoding of continuous values in $[x_1, x_m]$. This two-hot representation can be used directly in the cross-entropy loss function.

Decoding constants. When decoding a predicted sequence, we check at each prediction step whether the argmax of the logits corresponds to one of the m constant tokens $\{x_1, \ldots, x_m\}$. If not, we proceed by conventional one-hot decoding to obtain predicted operators and variables. If instead the argmax corresponds to, for example, x_i, we also pick its largest-logit neighbor (x_{i-1} or x_{i+1}; suppose x_{i+1}), renormalize their probabilities by applying a softmax to all logits and use the resulting two probability estimates as weights α, β. Constants are then ultimately decoded as $\alpha x_i + \beta x_{i+1}$.

3.3 Evaluation and Metrics

Sampling solutions. To infer a symbolic expression for the governing ODE of a new observed solution trajectory $\{(t_i, y_i)\}_{i=1}^n$, all the typical policies such as greedy, sampling, or beam search are available. In our evaluation, we use beam search with 1536 beams and report top-k results with k ranging from 1 to 1536.

Metrics. We evaluate model performance both numerically and symbolically. For numerical evaluation we follow Biggio et al. [2020]: suppose the ground truth ODE is given by $\dot{y} = f(y)$ with (numerical) solution $y(t)$ and the predicted ODE is given by $\hat{\dot{y}} = \hat{f}(y)$. To compute numerical accuracy we first evaluate f and $\hat{f}$ on N_{eval} points in the interval $[\min(y(t)), \max(y(t))]$ (i.e., the interval traced out by the observed solution), which yields function evaluations $\texttt{gt} = \{\dot{y}_i\}_{i=1}^{N_{\text{eval}}}$ and $\texttt{pred} = \{\hat{\dot{y}}_i\}_{i=1}^{N_{\text{eval}}}$. We then assess whether `numpy.allclose`[2] returns `True` as well as whether the coefficient of determination $\mathrm{R}^2 \geq 0.999$.[3] Numerical evaluations capture how closely the predicted function approximates the ground truth function within the interval $[\min(y(t)), \max(y(t))]$.

However, a key motivation for symbolic regression is to uncover a *symbolic* mathematical expression that governs the observations. Testing for symbolic equivalence between ground truth expression $f(y)$ and a predicted expression $\hat{f}(y)$ is unsuitable in the presence of real-valued constants as even minor deviations between true and predicted constants render the equivalence false. Instead, we regard the predicted expression $\hat{f}(y)$ to be symbolically correct if $f(y)$ and $\hat{f}(y)$ can be made equivalent by modifying only the values of constants in the predicted expression $\hat{f}(y)$. This is implemented using SymPy's `match` function. In order not to alter the structure of the predicted expression, we constrain modifications of constants such that all constants remain non-zero and retain their original sign. This definition is thus primarily concerned with the structure of an expression, rather than precise numerical agreement. Once the structure is known, the inference problem becomes conventional parameter estimation. We report percentages of samples in a given test set that satisfies any individual metric (numerical and symbolic), as well as percentages satisfying symbolic and numerical metrics simultaneously.

4 Experiments

4.1 Benchmark Datasets

We construct several test sets to evaluate model performance and generalization in different settings.

- **testset-iv**: Our first test set assesses generalization within initial values not seen during training. It consists of 5793 ODEs picked uniformly at random from our generated dataset but re-sampled

[2] `numpy.allclose` returns True if `abs(a - b) <= (atol + rtol * abs(b))` holds element-wise for elements a and b from the two input arrays. We use `atol=1e-10` and `rtol=0.05`; a corresponds to predictions, b corresponds to ground truth.

[3] For observations y_i and predictions $\hat{y}_i$ we have $\mathrm{R}^2 = 1 - (\sum_i (y_i - \hat{y}_i)^2)/(\sum_i (y_i - \overline{y})^2)$.

initial values. We also employ the following constraints via rejection sampling: (a) All skeletons in testset-iv are unique. (b) As the number of unique skeletons increases with the number of operators, we allow at most 2000 examples per number of operators (with substantially fewer unique skeletons existing for few operators).

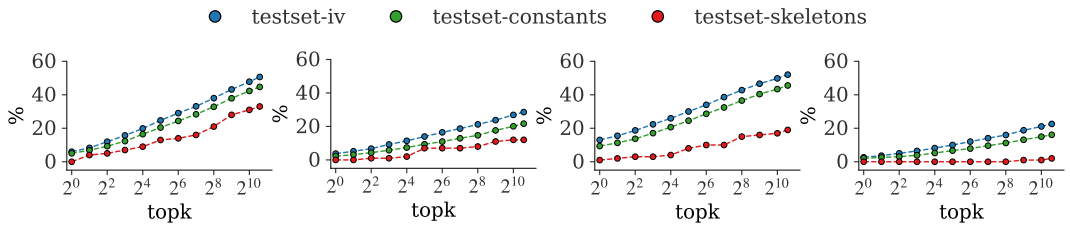
- **testset-constants**: Our second test set assesses generalization within unseen initial values and constants. It consists of 2720 ODEs picked uniformly at random from our dataset (ensuring unique skeletons and at most 1000 examples per number of operators as above), but re-sampled intial values and constants.
- **testset-skeletons**: In principle, we can train NSODE on all possible expressions (using only the specified operators and number ranges) up to a specified number of operators. However, even with the millions examples in our dataset, we have by far not exhausted the huge space of possible skeletons (especially for larger numbers of operators). Hence, our third test set contains 100 novel random ODEs with skeletons that were never seen during training.
- **testset-iv-163**: This is a subset of testset-iv motivated by the fact that most symbolic regression models we want to compare to require a separate optimization for every individual example, which was computationally infeasible for our testset-iv. For a fair comparison, we therefore subsampled up to 10 ODEs per complexity uniformly at random, yielding 163 examples.
- **testset-textbook**: To assess how NSODE performs on "real problems", we manually curated 12 scalar, non-linear ODEs from Wikipedia pages, physics textbooks, and lecture note from university courses on ODEs. These equations are listed in Table 7 in Appendix D. We note that they are all extremely simple compared to the expressions in our generated dataset in that they are ultimately mostly low order polynomials, some of which with one fractional exponent.
- **testset-classic**: To validate our approach on existing datasets we turn to benchmarks in the classic symbolic regression literature (inferring just the functional relationship between input-ouput pairs) and simply interpret functions as ODEs. In particular we include all scalar function listed in the overview in McDermott et al. [2012] which includes equations from many different benchmarks Keijzer [2003], Koza [1993, 1994], Uy et al. [2011], Vladislavleva et al. [2008]. For example, we interpret the function $f(y) = y^3 + y^2 + y$ from Uy et al. [2011] as an autonomous ODE $\dot{y}(t) = f(y(t)) = y(t)^3 + y(t)^2 + y$, which we solve numerically for a randomly sampled initial value as described before.

4.2 Baselines

We compare our method to recent popular baselines from the literature (see Section 2). We briefly describe them including some limitations here and defer all details to Appendix E. First, no baseline is suited directly to infer dynamical laws, but only to infer functional relationships. Therefore, all baselines fit a separate regression function mapping $y(t) \mapsto \hat{\dot{y}}(t)$ per individual ODE, using the coefficient of determination R^2 as optimization objective. Since derivatives $\hat{\dot{y}}(t)$ are typically not observed, we approximate them via finite differences using PySindy [de Silva et al., 2020]. Hence, all these methods crucially rely on regularly sampled and noise-free observations, whereas our approach can easily be extended to take those into account (see Appendix A).

- **Sindy** [Brunton et al., 2016]: Sindy builds a (sparse) linear combination of a fixed set of (non-linear) basis functions. The resulting Lasso regression is efficient, but suffers from limited expressiveness. In particular, Sindy cannot easly represent nested functions or non-integer powers as all non-linear expressions have to be added explicitly to the set of basis functions. We cross-validate Sindy over a fairly extensive hyperparameter grid of 800 different combinations for each individual trajectory.
- **GPL**[4] (genetic programming): GPL(earn) maintains a population of programs each representing a mathematical expression. The programs are mutated for several generations to heuristically optimize a user defined fitness function. While not originally developed for ODEs, we can apply GPLearn on our datasets by leveraging the finite difference approximation. We use a population

[4] `gplearn.readthedocs.io/`



(a) allclose (b) $R^2 \geq 0.999$ (c) skeleton recovery (d) skel. recov. & allclose

Figure 2: Numerical and symbolic performance evaluation on testset-iv.

size of 1000 and report the best performance across all final programs. Compared to sindy, GPLearn is more expressive yet substantially slower to fit.

- **AIFeynman** [Udrescu and Tegmark, 2020, Udrescu et al., 2020]: AIFeynman is a physics-inspired approach to symbolic regression that exploits the insight that many famous equations in natural sciences exhibit well-understood functional properties such as symmetries, compositionality, or smoothness. AIFeynman implements a neural network based heuristic search that tests for such properties in order to identify a symbolic expression that fits the data. For every test sample AIFeynman computes a pareto front of solutions that trade off complexity versus accuracy. We report the best performance across all functions on the pareto front. Notably, AIFeynman performs quite an exhaustive search procedure such that running it even on a single equation took on the order of tens of minutes.

4.3 Results

Model Performance. Figure 2 shows NSODE's performance on our testset-iv, testset-constants, and testset-skeletons according to our numerical and symbolic metrics as well as combined skeleton recovery and allclose as we vary k in the top-k considered candidates of the beam search. Investing sufficient test-time-compute (i.e., considering many candidates) continuously improves performance. While we capped k at 1536 due to memory limitations, we did not observe a stagnation of the roughly logarithmic scaling of all performance metrics with k. This cannot be attributed to "exhaustion effects", where one may assume that all possible ODEs will eventually be among the candidates, because (a) the space of possible skeletons grows much faster than exponentially, and (b) the numerical metrics are extremely sensitive also to the predicted constant values in continuous domains.

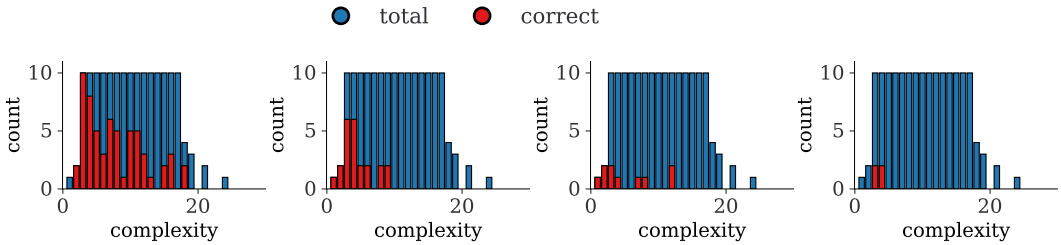
As one may expect, performance decreases as we move from only new initial conditions, to also sampling new constants, to finally sampling entirely unseen skeletons. On testset-iv with $k = 1536$ we achieve about 50% skeleton recovery and still successfully recover more than a third skeletons of testset-skeletons with similar numbers for allclose. The fact that the combined metric (symbolic + numerical) is only about half of that indicates that numerical and symbolic fit are indeed two separate measures, none of which need to imply the other. Hence, a thorough evaluation of both is crucial to understand model performance in symbolic regression tasks.

Comparison to baselines. In Table 1 we compare NSODE to all baselines using $k = 1536$ in our beam search; full results on all datasets can be found in Appendix F. We also include the average wallclock runtime per expression for each of the datasets.

First, we note that on our subsampled testset-iv-163, NSODE outperforms competing approaches in terms of skeleton recovery by a wide margin and also performs best in terms of joint skeleton recovery and numerical measures, which is a strong indication of actually having recovered the governing ODE accurately. By spending over 200x more time on its exhaustive heuristic search, AIFeynman manages to outperform NSODE in terms of numerical accuracy (R^2 and allclose). Figure 3 shows the number of skeletons recovered by each method given the complexity of equations, results for other datasets

Table 1: Comparing NSODE to the baselines. Results are average percentages across dataset. GPLearn often generates extremely long expressions which take SymPy up to half a minute to parse during evaluation. We denote this extra time in gray.

Dataset	Metric	NSODE	Sindy	GPLearn	AIFeynman
iv-163	skel-recov	**37.4**	3.7	2.5	14.1
	$R^2 \geq 0.999$	24.5	31.9	3.7	**49.7**
	allclose	42.3	25.8	14.7	**55.8**
	skel-recov & $R^2 \geq 0.999$	**15.3**	3.1	1.8	13.5
	skel-recov & allclose	**15.3**	3.1	1.8	13.5
	runtime [s]	5.4	**0.4**	29 +22	1203.6
textbook	skel-recov	41.7	33.3	8.3	**91.7**
	$R^2 \geq 0.999$	16.7	50	0.0	**75**
	allclose	25	58.3	8.3	**75**
	skel-recov & $R^2 \geq 0.999$	33.3	41.7	0	**66.7**
	skel-recov & allclose	8.3	33.3	1.8	**66.7**
	runtime [s]	6	**1**	23 +22	1267.1



(a) NSODE (b) AIFeynman (c) Sindy (d) GPLearn

Figure 3: Correctly recovered skeletons by each method on testset-iv-163 per complexity. AIFeynman and Sindy are mostly able to recover some of the low complexity skeletons, while NSODE performs much better also on higher complexities. GPLearn fails to recover most skeletons.

can be found in Appendix F. [5] While AIFeynman and Sindy recover some of the low complexity expressions, NSODE is the only method to also recover some of the more complex skeletons.

On testset-textbook, AIFeynman outperforms all other methods on numerical and symbolic metrics. This can be understood with regard to the dataset where $8/12$ expressions are polynomials with the remaining 4/12 expressions having a polynomial skeleton with fractional or negative exponents. These expressions are particularly favorable for the heuristics implemented by AIFeynman which explicitly attempt to fit a polynomial to the data. However, even on these simple examples AIFeynman takes over 200x longer than our method, which in turn clearly outperforms Sindy and GPLearn in terms of skeleton recovery.

5 Conclusion

We have developed a flexible and scalable method to infer ordinary differential equations $\dot{y} = f(y)$ from a single observed solution trajectory. Our method follows the successful paradigm of large-scale pretraining of attention-based sequence-to-sequence models on essentially unlimited amounts of simulated data, where the inputs are the observed solution $\{(t_i, y_i)\}_{i=1}^n$ and the output is a symbolic expression for f as a string. Once trained, our method is orders of magnitude more efficient than similarly expressive existing symbolic regression techniques that require a separate optimization for each instance and achieves strong performance in terms of skeleton recovery especially for complex expressions on various benchmarks.

[5] Due to simplification, complexity is not upper bounded by the number of nodes in a unary-binary tree.

References

S. Atkinson, W. Subber, L. Wang, G. Khan, P. Hawi, and R. Ghanem. Data-driven discovery of free-form governing differential equations. *arXiv preprint arXiv:1910.05117*, 2019.

L. Biggio, T. Bendinelli, A. Lucchi, and G. Parascandolo. A seq2seq approach to symbolic regression. In *Learning Meets Combinatorial Algorithms at NeurIPS2020*, 2020.

L. Biggio, T. Bendinelli, A. Neitz, A. Lucchi, and G. Parascandolo. Neural symbolic regression that scales. In M. Meila and T. Zhang, editors, *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pages 936–945. PMLR, 18–24 Jul 2021. URL `https://proceedings.mlr.press/v139/biggio21a.html`.

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*, 113(15):3932–3937, 2016. ISSN 0027-8424. doi: 10.1073/pnas.1517384113. URL `https://www.pnas.org/content/113/15/3932`.

R. T. Chen, Y. Rubanova, J. Bettencourt, and D. Duvenaud. Neural ordinary differential equations. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, pages 6572–6583, 2018.

A. Costa, R. Dangovski, O. Dugan, S. Kim, P. Goyal, M. Soljačić, and J. Jacobson. Fast neural models for symbolic regression at scale, 2021.

B. de Silva, K. Champion, M. Quade, J.-C. Loiseau, J. N. Kutz, and S. Brunton. Pysindy: A python package for the sparse identification of nonlinear dynamical systems from data. *Journal of Open Source Software*, 5(49):1–4, 2020.

A. Hindmarsh and L. L. Laboratory. *ODEPACK, a Systematized Collection of ODE Solvers*. Lawrence Livermore National Laboratory, 1982. URL `https://books.google.de/books?id=9XWPmwEACAAJ`.

P.-A. Kamienny, S. d'Ascoli, G. Lample, and F. Charton. End-to-end symbolic regression with transformers. *arXiv preprint arXiv:2204.10532*, 2022.

M. Keijzer. Improving symbolic regression with interval arithmetic and linear scaling. In *European Conference on Genetic Programming*, pages 70–82. Springer, 2003.

J. R. Koza. *Genetic programming - on the programming of computers by means of natural selection*. Complex adaptive systems. MIT Press, 1993. ISBN 978-0-262-11170-6.

J. R. Koza. *Genetic programming II: automatic discovery of reusable programs*. MIT press, 1994.

G. Lample and F. Charton. Deep learning for symbolic mathematics. In *International Conference on Learning Representations*, 2019.

J. Liu, Z. Long, R. Wang, J. Sun, and B. Dong. Rode-net: learning ordinary differential equations with randomness from data. *arXiv preprint arXiv:2006.02377*, 2020.

Z. Long, Y. Lu, and B. Dong. Pde-net 2.0: Learning pdes from data with a numeric-symbolic hybrid deep network. *Journal of Computational Physics*, 399:108925, 2019.

J. McDermott, D. R. White, S. Luke, L. Manzoni, M. Castelli, L. Vanneschi, W. Jaskowski, K. Krawiec, R. Harper, K. De Jong, et al. Genetic programming needs better benchmarks. In *Proceedings of the 14th annual conference on Genetic and evolutionary computation*, pages 791–798, 2012.

A. Meurer, C. P. Smith, M. Paprocki, O. Čertík, S. B. Kirpichev, M. Rocklin, A. Kumar, S. Ivanov, J. K. Moore, S. Singh, T. Rathnayake, S. Vig, B. E. Granger, R. P. Muller, F. Bonazzi, H. Gupta, S. Vats, F. Johansson, F. Pedregosa, M. J. Curry, A. R. Terrel, v. Roučka, A. Saboo, I. Fernando, S. Kulal, R. Cimrman, and A. Scopatz. Sympy: symbolic computing in python. *PeerJ Computer Science*, 3:e103, Jan. 2017. ISSN 2376-5992. doi: 10.7717/peerj-cs.103. URL `https://doi.org/10.7717/peerj-cs.103`.

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.

S. Sahoo, C. Lampert, and G. Martius. Learning equations for extrapolation and control. In *International Conference on Machine Learning*, pages 4442–4450. PMLR, 2018.

J. Schrittwieser, I. Antonoglou, T. Hubert, K. Simonyan, L. Sifre, S. Schmitt, A. Guez, E. Lockhart, D. Hassabis, T. Graepel, et al. Mastering atari, go, chess and shogi by planning with a learned model. *Nature*, 588(7839):604–609, 2020.

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

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. *arXiv preprint arXiv:2006.10782*, 2020.

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*, 12(2):91–119, 2011.

M. Valipour, M. Panju, B. You, and A. Ghodsi. SymbolicGPT: A Generative Transformer Model for Symbolic Regression. In *Preprint Arxiv*, 2021. URL `https://arxiv.org/abs/2106.14131`.

M. Vastl, J. Kulhánek, J. Kubalík, E. Derner, and R. Babuška. Symformer: End-to-end symbolic regression using transformer-based architecture. *arXiv preprint arXiv:2205.15764*, 2022.

P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. J. Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. J. Carey, İ. Polat, Y. Feng, E. W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17: 261–272, 2020. doi: 10.1038/s41592-019-0686-2.

E. J. Vladislavleva, G. F. Smits, and D. Den Hertog. Order of nonlinearity as a complexity measure for models generated by symbolic regression via pareto genetic programming. *IEEE Transactions on Evolutionary Computation*, 13(2):333–349, 2008.

J. Weilbach, S. Gerwinn, C. Weilbach, and M. Kandemir. Inferring the structure of ordinary differential equations. *arXiv preprint arXiv:2107.07345*, 2021.

M. Zaheer, G. Guruganesh, K. A. Dubey, J. Ainslie, C. Alberti, S. Ontanon, P. Pham, A. Ravula, Q. Wang, L. Yang, et al. Big bird: Transformers for longer sequences. *Advances in Neural Information Processing Systems*, 33:17283–17297, 2020.

A Possible extensions

While we have focused exclusively on the huge class of scalar, autonomous, first-order ODEs, we believe that our approach can scale also to non-autonomous, higher-order, systems of ODEs.

Non-autonomous equations. Since our model is provided with time t as inputs, it is capable of learning functions $f(y, t)$ depending on y and t explicitly. Hence, extending our approach to non-autonomous ODEs is simply a matter of adding a symbol for t in our data generation.

Irregular samples and noise. Due to the separate data generation and training phase, it is straight forward to train NSODE on corrupted input sequences, where we could for example add observation noise (need not be additive) to the $\{y_i\}_{i=1}^n$ or randomly drop some of the observations on the regular observation grid to simulate irregularly sampled observations. We expect these models to still perform well and in particular much better than the baselines, since their targets are approximated derivatives, which are highly sensitive to noise and irregular samples.

Systems of equations. For a system of K first-order ODEs, the data generation requires two updates: first, we have to generate K equations $(f_k)_{k\in\{1,\dots,K\}}$, one for each component. In addition, each of those functions can depend not only on y (and t in the non-autonomous case), but K different components $\{y^{(k)}\}_{k\in\{1,\dots,K\}}$. This is easily achieved by allowing for K tokens $y^{(k)}$ in the data generation. We can then simply augment the input sequence to the transformer to contain not only (t, y) as for the scalar case, but $(t, y^{(1)}, y^{(2)}, \dots, y^{(K)})$ for a system of K variables. Finally, we have decide on a way to set the target for the transformer, i.e., how to represent the system of equations in symbolic form as *a single sequence*. A straight forward way to achieve this is to introduce a special delimitation character, e.g., `"|"` to separate the different components. Note that the order in which to predict the K equations is dictated by the order in which they are stacked for the input sequence, hence this information is in principle available to the model. For example, in the case of a two-dimensional system, where $y = (y^{(1)}, y^{(2)}) \in \mathbb{R}^2$, we could have the model perform the following mapping

$$\{(t_i, y_i^{(1)}, y_i^{(2)})\}_{i=1}^n \rightarrow \texttt{"-1.4*y}^{\texttt{(1)}}\texttt{*y}^{\texttt{(2)}}\texttt{+cos(t) | sin(y}^{\texttt{(1)}}\texttt{+y}^{\texttt{(2)}}\texttt{)"},$$

where the separator `|` delimits the two components of $f(y, t)$ corresponding to $\dot{y}^{(1)}$ and $\dot{y}^{(2)}$.

Higher-order equations. It is well known that any higher order (system of) ODEs can be reduced to a first-order system of ODEs. Specifically, a d-th order system of K variables can be reduced to an equivalent first-order system of $d \cdot K$ variables. Hence, one can handle higher-order systems analogously as before with multiple separator tokens. One obstacle in this case is that when only observations of $y(t)$ are given, one first needs to obtain observed derivatives to reduce a higher-order system to a first-order system. These would in turn have to be estimated from data, which suffers from the same challenges we have mentioned previously (instability under noise and irregular samples).

Finally, when we want to have a single model deal with higher-order equations of unknown order, or systems with differing numbers of variables, it remains on open question how to have the model automatically adjust to the potentially vastly differing input dimensions or how to automatically detect the order of an ODE.

B Implementation Details

B.1 Rules to Resample Constants

As described in Section 3.1, we generate ODEs as unary-binary trees, convert them into infix notation and parse them into a canonical form using `sympy`. From each skeleton we then create up to 25 ODEs by sampling different values for the constants. When resampling constants we want to ensure that we do not accidentally modify the skeleton as this would additionally burden our model with resolving potential ambiguities in the grammar of ODE expressions. Furthermore, we do not want to reintroduce duplicate samples on the skeleton level after carefully filtering them out previously. We therefore introduce the following sampling rules for constants:

1. Do not sample constants of value 0.
2. When the original constant in the skeleton is negative, sample a negative constant, otherwise sample a positive constant.
3. Avoid base of 1 in power operations as $1^x = 1$.
4. Avoid exponent of 1 and -1 in power operations as $x^1 = x$ and $x^{-1} = 1/x$.
5. Avoid coefficients of value 1 and -1 as $1 \cdot x = x$ and $-1 \cdot x = -x$
6. Avoid divisions by 1 and -1 as $x/1 = x$ and $x/-1 = -x$

B.2 Data generation

As discussed in the main text, the choices of the maximum number of internal nodes per tree K, the choice and distribution over N_{bin} binary operators, the choice and distribution over N_{una} unary operators, the probability with which to decorate a leaf with a symbol p_{sym} (versus a constant with $1 - p_{\text{sym}}$), and the distribution p_{c} over constants uniquely determine the training distribution over ODEs f. These choices can be viewed as flexible and semantically interpretable tuning knobs to choose a prior over ODEs. For example, it may be known in a given context, that the system follows a "simple" law (small K) and does not contain exponential rates of change (do not include $\exp$ in the unary operators), and so on. The choice of the maximum number of operators per tree, how to sample the operators, and how to fill in the leaf nodes define the training distribution, providing us with flexible and semantically meaningful tuning knobs to choose a prior over ODE systems for our model. We summarize our choices in Tables 2 to 4, where $\mathcal{U}$ denotes the uniform distribution. Whenever a leaf node is decorated with a constant, the distribution over constants is determined by first determining whether to use an integer or a real value with equal probablity. In case of an integer, we sample it from p_{int}, and in case of a real-valued constant we sample it from p_{real} shown in Table 2. Finally, when it comes to the numerical solutions of the sampled ODEs, we fixed the parameters in Table 5 for our experiments.

We highlight that there is no such thing as "a natural distribution over equations" when it comes to ODEs. Hence, ad-hoc choices have to be made in one way or another. However, it is important to note that neither our chosen range of integers nor the range of real values for constants are in any way restrictive as they can be achieved by appropriate rescaling. In particular, the model itself represents these constant values merely be non-numeric tokens and interpolates between those anchor tokens (our two-hot encoding) to represent continuous values. Hence, the model is entirely agnostic to the actual numerical range spanned by these fixed grid tokens, but the relative accuracy in recovering interpolated values will be constant and thus scale with the absolute chosen range. Therefore, scaling p_{int} and p_{real} by essentially any power of 10 does not affect our findings. Similarly, the chosen range of initial values $(y_0^{\min}, y_0^{\max})$ is non-restrictive as one could simply scale each observed trajectory to have its starting value lie within this range.

Table 2: Parameter settings for the data generation.

parameter	K	N_{bin}	N_{una}	p_{sym}	p_{int}	p_{real}
value	5	5	5	0.5	$\mathcal{U}(\{-10, \ldots, 10\} \setminus \{0\})$	$\mathcal{U}((-10, 10))$

Table 3: Binary operators with their relative sampling frequencies

operator	$+$	$-$	$\cdot$	$\div$	pow
probability	0.2	0.2	0.2	0.2	0.2

Table 4: Unary operators with their relative sampling frequencies. The last column denotes the unary minus as in e.g. $f(y) = -y$. We do not explicitly sample this operator but keep it after simplification.

operator	sin	cos	exp	$\sqrt{\ }$	log	$-$
probability	0.2	0.2	0.2	0.2	0.2	0

Table 5: Parameters for numerical solutions of sampled ODEs.

parameter	N_{const}	N_{iv}	T	N_{grid}	$(y_0^{\min}, y_0^{\max})$
value	25	25	4	1024	$(-5, 5)$

B.3 Model

Architectural choices. We use an encoder-decoder transformer and list our architectural choices in Table 6. For the two-hot representation of symbolic constants as described in Section 3.2, we choose an equidistant grid $-10 = x_1 < x_2 < \ldots < x_m = 10$ with $m = 21$.

While not relevant for our dataset (as we check for convergence of the ODE solvers), we remark that the input-encoding via IEEE-754 binary representations also graciously represents special values such as `nan` or `inf` without causing errors. Those are thus valid inputs that may still provide useful training signal, e.g., "the solution of the ODE of interest goes to `inf` quickly".

Training details. Our model is trained on 4 Nvidia A100 GPUs for 18 epochs after which we evaluate the best model based on the validation loss. We choose a batchsize of 600 samples and use a linear learning rate warm-up over 10,000 optimization step after which we keep the learning rate constant at 10^{-4}. During training we use teacher forcing to guide sequential predictions. Teacher forcing is straightforward to implement for one-hot encoded tokens (operators, symbols) which are simply passed through a learnable embedding layer before their embedded representations are fed to the decoder. In contrast, two-hot encoded tokens (constants) can not be passed through an embedding layer directly. For a generic constant in the target sequence represented as $\alpha x_i + \beta x_{i+1}$, we thus instead provide the linear combination of the two embeddings α $\texttt{embed(x}_\texttt{i}\texttt{)} + \beta$ $\texttt{embed(x}_\texttt{i+1}\texttt{)}$ as decoder input. We represent symbolic input to the model in prefix format which relieves the model from correctly predicting opening and closing parentheses.

Evaluation. We use $N_{\text{eval}} = 100$ for the evaluation of our numerical metrics.

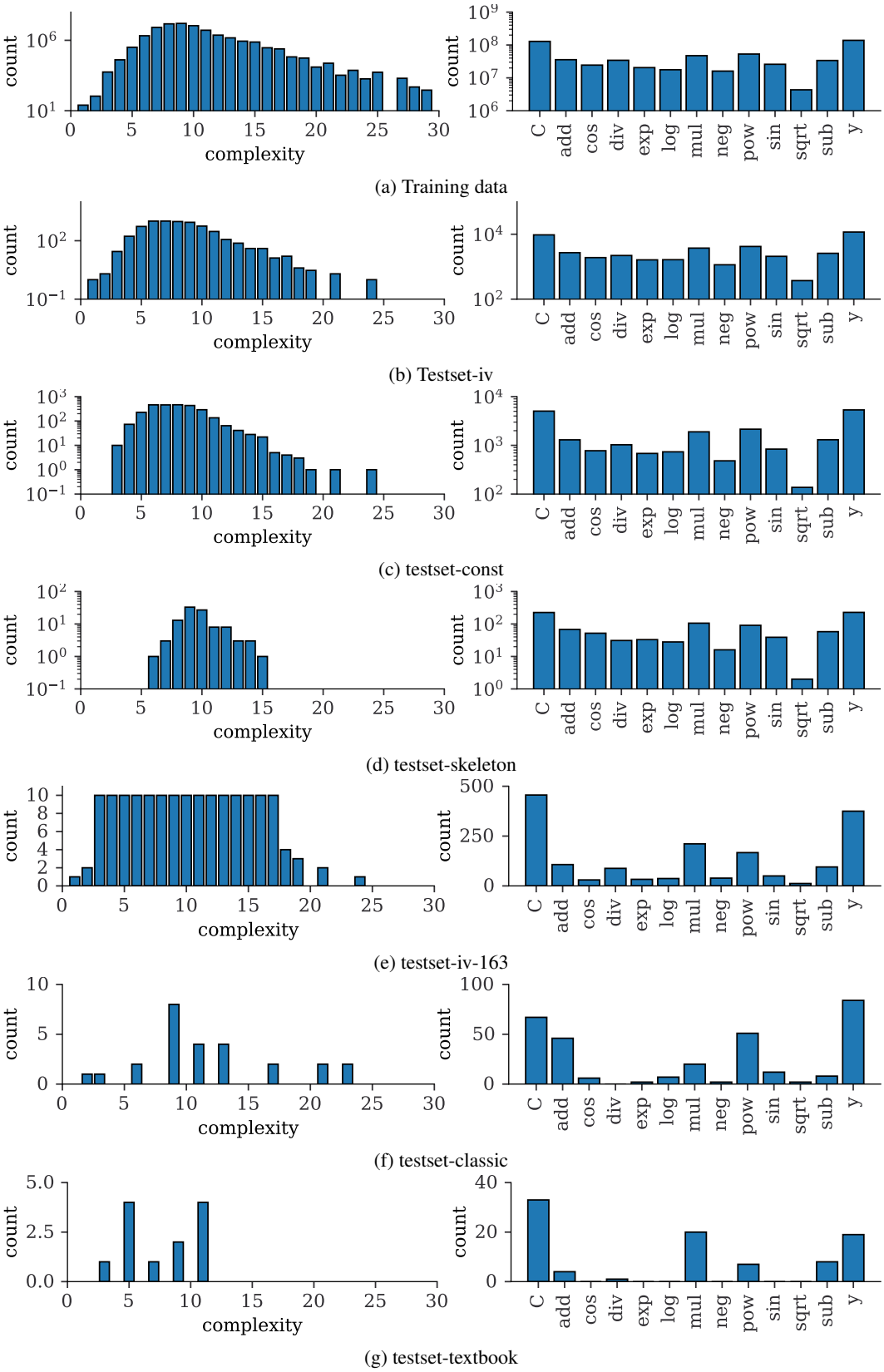
C Dataset statistics

We provide an overview over the complexity distribution and the absolute frequency of all operators (after simplification) for all datasets in Figure 4. We can see that our self-generated dataset covers by far the larges complexity whereas both complexities and operator diversity are much lower for equations in the classic and textbook ODEs.

Table 6: Overview of our model architecture.

	Encoder	**Decoder**
architecture	BigBird[†]	BigBird
layers	6	6
heads	16	16
embed. dim.	512	512
forward dim.	2048	2048
activation	gelu	gelu
vocab. size	-	43
position enc.	learned	learned
parameters	23.3M	23.3M

[†]We chose the BigBird [Zaheer et al., 2020] implementation with full attention available in Hugging Face.



(a) Training data

(b) Testset-iv

(c) testset-const

(d) testset-skeleton

(e) testset-iv-163

(f) testset-classic

(g) testset-textbook

Figure 4: Distribution of complexity and operators for all datasets. Complexity is defined as the number of symbols, constants and operators in an expression.

Table 7: Equations of the textbook dataset.

Name	**Equation** $f(x)$	**simplified**	$\mathbf{y_0}$
autonomous Riccati	$0.6 \cdot y^2 + 2 \cdot y + 0.1$	$0.6 \cdot y^2 + 2 \cdot y + 0.1$	-0.2
autonomous Stuart-Landau	$-2.2/2 \cdot y^3 + 1.31 \cdot y$	$-1.1 \cdot y^3 + 1.31 \cdot y$	0.1
autonomous Bernoulli	$-1.3 \cdot y + 2.1 \cdot y^{2.2}$	$-1.3 \cdot y + 2.1 \cdot y^{2.2}$	0.6
compound interest	$0.1 \cdot y$	$0.1 \cdot y$	9
Newton's law of cooling	$-0.1 \cdot (y - 3)$	$0.3 - 0.1 \cdot y$	9
Logistic equation	$0.23 \cdot y \cdot (1 - y)$	$0.23 \cdot (y - y^2)$	9
Logistic equation with harvesting	$0.23 \cdot y \cdot (1 - 0.33 \cdot y) - 0.5$	$0.23 \cdot y - 0.76 \cdot y^2 - 0.5$	9
Logistic equation with harvesting 2	$2 \cdot y \cdot (1 - y/3) - 0.5$	$2 \cdot y - 0.66 \cdot y^2 - 0.5$	0.7
Solow-Swan	$y^0.5 \cdot (0.9 \cdot 8 - (3 + 2.5) \cdot y^{1-0.5})$	$7.2 \cdot y^{0.5} - 5.5 \cdot y$	0.1
Tank draining	$-\sqrt{2 \cdot 9.81} \cdot (2/9)^2 \cdot \sqrt{y}$	$-0.21 \cdot y^{0.5}$	1
Draining water through a funnel	$-(0.5^2/4) \cdot \sqrt{2 \cdot 9.81} \cdot (\sin 1/\cos 1)^2 \cdot y^{-1.5}$	$-0.67/y^{1.5}$	3
velocity of a body thrown vertically upwards	$-9.81 - 0.9 \cdot y/8.2$	$-0.10 \cdot y - 9.81$	0.1

D Textbook equations dataset

Table 7 list the equations we collected from wikipedia, textbooks and lecture notes together with the initial values that we solved them for. We can also see that almost all of these equations simplify to low-order polynomials.

E Baselines

We here describe more detail on the optimization of the baseline comparison models.

Sindy. We use the implementation available in PySindy [de Silva et al., 2020] and instantiate the basis functions with polynomials up to degree 10 as well as all unary operators listed in Table 4. When fitting sindy to data we often encountered numerical issues especially when using high-degree polynomial or the exponential function. To attenuate such issues we set the highest degree of the polynomials per sample to the highest degree present in the ground truth. Secondly, when numerical issues are caused by a particular basis function, we discard this basis function for the current sample and restart the fitting process. We run a separate full grid search for every ODE over the following hyper-parameters and respective values (these all include the default values):

- optimizer-threshold (`np.logspace(-5, 0, 10)`): Minimum magnitude for a coefficient in the weight vector to not be zeroed out.
- optimizer-alpha ($[0.001, 0.0025, 0.005, 0.01, 0.025, 0.05, 0.1, 0.2]$): L2 regularizer on parameters.
- finite differences order ($[2, 3, 5, 7, 9]$): Order of finite difference approximation.
- maximum number of optimization iterations ([20, 100]): Maximum number of optimization steps.

For every ODE, sindy is fit using solution trajectory in the initial interval $[0, 2]$ and validated on the interval $(2, 4]$. The grid search thus results in a ranking of models with different hyper-parameter configurations. Instead of evaluating only the performance of the best model, we report top-k performance across the ranked hyper parameter configurations. Sindy is computationally highly efficient yet limited in its expressiveness, in particular it can not represent nested functions or non-integer powers.

GPLearn. We instantiate GPLearn with a constant range of $(-10, 10)$ and all binary operators listed in Table 3 and all unary operators listed Table 4 except for the exponential function which caused

numerical issues. We keep the default hyper-parameters but run a grid search across the parsimony coefficient ({0.0005, 0.001, 0.01, 0.05, 0.1, 0.2, 0.5, "auto"}) which trades of fitness versus program length. We choose R^2 as fitness function.

AIFeynman. We use the AIFeynman implementation from [] and run the algorithm with the following (default) hyper-parameters:

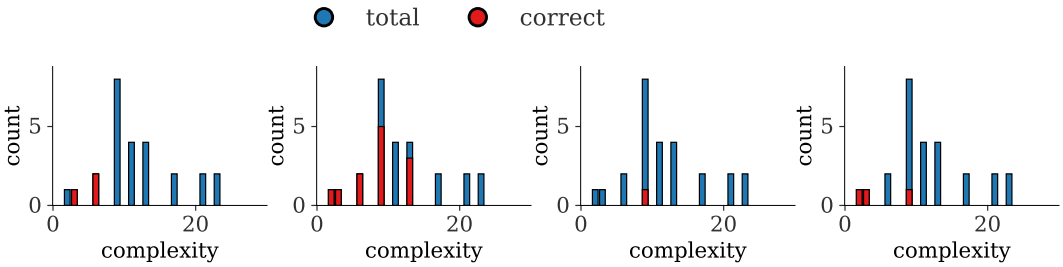
- Brute force try time: 60 seconds
- Number of epochs for the training : 500
- Operator set: 14
- Maximum degree of polynomial tried by the polynomial fit routine: 4

F Detailed results

We provide a comprehensive summary of performances of all models on all datasets in Table 8. Additionally, Figures 5 to 8 show the number of correctly recovered skeletons by each method per complexity.

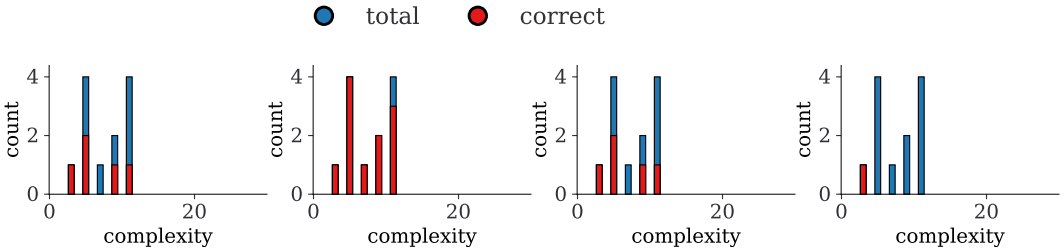
Table 8: Detailed performance results for all methods on all (applicable) datasets.

Dataset	Metric	NSODE	Sindy	GPLearn	AIFeynman
testset-iv	skel-recov	52.0	-	-	-
	$R^2 \geq 0.999$	28.6	-	-	-
	allclose	50.6	-	-	-
	skel-recov & $R^2 \geq 0.999$	17	-	-	-
	skel-recov & allclose	22.6	-	-	-
	runtime [s]	5.3	-	-	-
testset-constant	skel-recov	45.6	-	-	-
	$R^2 \geq 0.999$	21.7	-	-	-
	allclose	44.7	-	-	-
	skel-recov & $R^2 \geq 0.999$	9.8	-	-	-
	skel-recov & allclose	16.1	-	-	-
	runtime [s]	5.3	-	-	-
testset-skel	skel-recov	19	-	-	-
	$R^2 \geq 0.999$	12	-	-	-
	allclose	33	-	-	-
	skel-recov & $R^2 \geq 0.999$	1	-	-	-
	skel-recov & allclose	2	-	-	-
	runtime [s]	5.3	-	-	-
testset-iv-163	skel-recov	**37.4**	3.7	2.5	14.1
	$R^2 \geq 0.999$	24.5	31.9	3.7	**49.7**
	allclose	42.3	25.8	14.7	**55.8**
	skel-recov & $R^2 \geq 0.999$	**15.3**	3.1	1.8	13.5
	skel-recov & allclose	**15.3**	3.1	1.8	13.5
	runtime [s]	5.4	**0.4**	29 +22	1203.6
classic	skel-recov	11.5	0	3.8	**46.2**
	$R^2 \geq 0.999$	57.7	57.7	23.1	**88.5**
	allclose	80.8	57.7	30.8	**88.5**
	skel-recov & $R^2 \geq 0.999$	0	0	7.7	**46.2**
	skel-recov & allclose	0	0	7.7	**46.2**
	runtime [s]	5.2	**0.6**	23 +22	1291.6
textbook	skel-recov	41.7	33.3	8.3	**91.7**
	$R^2 \geq 0.999$	16.7	50	0.0	**75**
	allclose	25	58.3	8.3	**75**
	skel-recov & $R^2 \geq 0.999$	33.3	41.7	0	**66.7**
	skel-recov & allclose	8.3	33.3	1.8	**66.7**
	runtime [s]	6	**1**	23 +22	1267.1



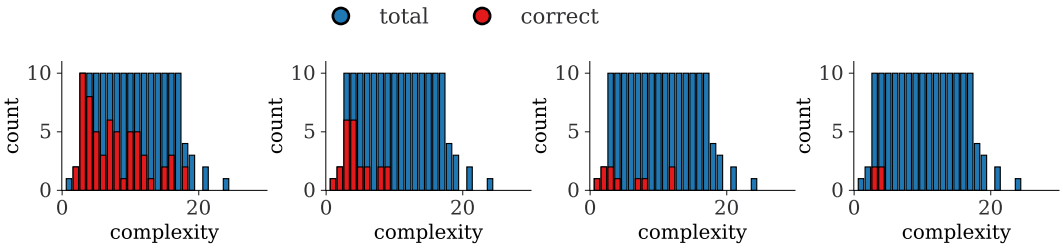
(a) NSODE (b) AIFeynman (c) Sindy (d) GPLearn

Figure 5: Correctly recovered skeletons by each method on the classic benchmark dataset per complexity.



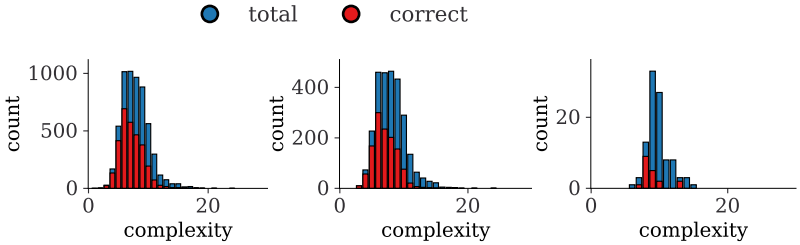
(a) NSODE (b) AIFeynman (c) Sindy (d) GPLearn

Figure 6: Correctly recovered skeletons by each method on the textbook dataset per complexity.



(a) NSODE (b) AIFeynman (c) Sindy (d) GPLearn

Figure 7: Correctly recovered skeletons by each method on testset-iv-163 per complexity.



(a) Testset-iv (b) Testset-constant (c) Testset-skeleton

Figure 8: Correctly recovered skeletons by NSODE on testset-iv, testset-constants, and testset-skeletons.