A Unified Framework for Deep Symbolic Regression

Anonymous Author(s)
Affiliation
Address
email

Abstract

The last few years have witnessed a surge in methods for symbolic regression, from advances in traditional evolutionary methods to novel deep learning-based methods. Individual works typically focus on advancing the state-of-the-art for one particular class of solution methods, and there have been few attempts to investigate the benefits of hybridizing or integrating multiple methods. In this work, we identify five standalone symbolic regression methods whose individual capabilities—spanning neural-guided search, genetic programming, and linear regression—provide broad coverage of the overall space of existing approaches, and we propose a strategy to hybridize them into a single modular, unified symbolic regression framework. Based on empirical evaluation using SRBench, a new community tool for benchmarking symbolic regression methods, our unified framework achieves state-of-the-art performance in its ability to (1) symbolically recover analytical expressions, (2) fit datasets with high accuracy, and (3) balance accuracy-complexity trade-offs, across 252 ground-truth and black-box benchmark problems, in both noiseless settings and across various noise levels. Finally, we provide practical use case-based guidance for constructing hybrid symbolic regression algorithms, supported by extensive, combinatorial ablation studies.

1 Introduction

Symbolic regression (SR) aims to identify a tractable mathematical expression to best fit a dataset. Specifically, given the set of pairs $\{(x_i = (x_{i,1}, \ldots, x_{i,d}), y_i)\}_{i=1}^n$, SR seeks a mathematical function $f : \mathbb{R}^d \to \mathbb{R}$ such that the residual $\|f(X) - y\|^2$ is minimized, where the matrix $X \in \mathbb{R}^{n \times d}$ has entries $X_{i,j} = x_{i,j}$, the vector $y \in \mathbb{R}^n$ has entries y_i, and we assume point-wise application of real-valued functions over vectors and matrices, i.e., $f(X) = (f(x_i))_{i=1}^n \in \mathbb{R}^n$ (we keep this convention throughout the paper for convenience).

Since the introduction of SR as an application of genetic programming (GP) (Koza, 1994), many different classes of methods—especially machine learning-based methods in recent years—have been proposed. These methods include recursive problem simplification (Udrescu et al., 2020), neural-guided search (Petersen et al., 2021a), large scale encoder training for generalization (Biggio et al., 2021), sparse linear representation using basis functions (Brunton et al., 2016), and combination of evo-

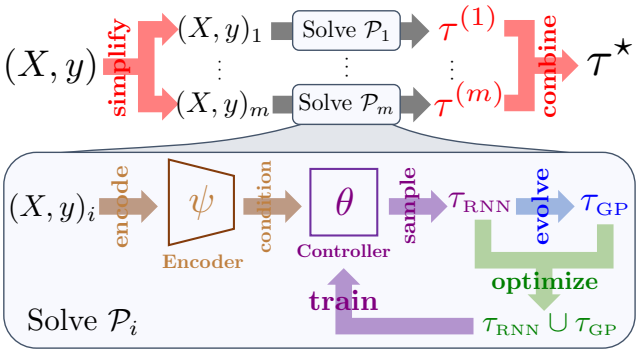

Figure 1: Unified deep symbolic regression. The five integrated algorithms are color-coded: AIF, NeSymReS, DSR, GP, SINDy.

Submitted to 36th Conference on Neural Information Processing Systems (NeurIPS 2022). Do not distribute.

lutionary search with learning (Mundhenk et al., 2021). Each of these independent advances exploits different information (e.g., graph modularity or constraints) to offer key capabilities for various aspects of the SR problem (e.g., search or generalization), but the specificity of each work resulted in design choices that pose limitations in other facets of the problem. This motivates an investigation of whether disparate methods are complementary to one another and how they can be effectively combined. In this paper, we combine the five aforementioned methods into a unified SR framework and show that they are indeed complementary.

Based on the SRBench pipeline for benchmarking SR algorithms (`https://github.com/cavalab/srbench`), our unified deep symbolic regression (uDSR) framework achieves a new state-of-the-art in SR: for ground-truth problems, uDSR ranks highest in both symbolic-based and accuracy-based solution rates for both noiseless settings and for multiple noise levels; for black-box problems, uDSR falls on the accuracy-complexity Pareto frontier alongside three other non-dominated algorithms.

Our main contributions in this work include: (1) a modular, unified framework for SR that integrates five disparate SR methods (illustrated in Figure 1); (2) a novel LINEAR token that identifies sparse coefficients of a linear combination of basis functions and can be used by almost any SR algorithm; (3) achieving new state-of-the-art for SR on SRBench in its metrics for symbolic recovery, solution accuracy, and accuracy vs complexity trade-offs; and (4) practical recommendations for designing SR algorithms, driven by extensive and combinatorial ablation studies that show the complementary benefits of these previously disjoint SR methods.

2 Background

We describe each of the five methods our framework unifies. For each method, we focus on a *key capability* and a *key limitation*[1], summarized in Table 1, which motivates our unification that captures all capabilities and mitigates all limitations via the inherent synergy among methods. We encourage readers to read the original works for additional background.

AI Feynman. AI Feynman 2.0 (AIF) exploits knowledge of physics and the given training data (X, y), by identifying simplifying properties (e.g. multiplicative separability) of the underlying functional form (Udrescu et al., 2020). Applying a simplification yields sub-problems (or one sub-problem) of lower dimensionality. Simplifications are applied recursively until all sub-problems (i.e. leaf nodes of the recursion tree) are 1-dimensional. For each sub-problem, the paper employs a combination of brute force search plus polynomial fit (BF/PF). Finally, solutions to the sub-problems are then combined back into a final solution to the original problem. While AIF positions itself as a standalone SR algorithm, it is more appropriately understood as a *SR problem simplification algorithm*, as pointed out in prior work (Petersen et al., 2021a; Biggio et al., 2021). From this general perspective, the use of BF/PF is simply a design choice, orthogonal to the core methodology, and may be enhanced by integration with more advanced search methods for solving sub-problems.

Deep symbolic regression. Deep symbolic regression (DSR) leverages the effectiveness of deep reinforcement learning for combinatorial search in SR (Petersen et al., 2021a). DSR employs a neural network *controller* with parameters θ to represent a distribution $p(\tau|\theta)$ over mathematical expressions $\tau = [\tau_1, \ldots, \tau_T]$, represented by the sequence of "tokens" τ_i (mathematical operators, input variables, or constants) in the pre-order (depth-first then left-to-right) traversal of the corresponding *expression tree* (see Koza (1994) for details). For instance, $\tau = [+, \text{SIN}, x_1, \text{EXP}, x_2]$ corresponds to expression $f(x) = \sin(x_1) + \exp(x_2)$. Expressions are sampled from the controller autoregressively, i.e. $p(\tau|\theta) = \prod_{i=1}^{T} p(\tau_i|\tau_{1:i-1}, \theta)$, and evaluated based on fitness $R(\tau, (X, y))$ to the (X, y) dataset. The controller is then trained to maximize the reinforcement learning objective:

$$J_{\text{RL}}(\theta) := \mathbb{E}_{\tau \sim p(\tau|\theta)} \left[R(\tau, (X, y)) \right], \tag{1}$$

using a risk-seeking policy gradient that aims to maximize best-case performance. Autoregressive sampling affords the ability to easily incorporate prior knowledge by directly modifying likelihoods of individual tokens $p(\tau_i|\tau_{1:i-1}, \theta)$ *in situ*, for example by precluding expressions with nested trigonometric operators. This dramatically reduces the search space, which has been shown to greatly

[1] We clarify that the key limitations are not necessarily issues with the algorithms themselves; often, they are conscious design decisions by the authors. For example, in the context of physics-based SR problems, AIF's use of BF/PF only strengthens their claim that recursive simplification alone is often sufficient to solve the problem. This work considers a wider context of challenging SR problems, in which these can be viewed as limitations.

improve the ability to recover ground-truth expressions (Petersen et al., 2021b). Whereas DSR focuses on discrete search and pruning the combinatorial search space, it does not exploit the (X, y) data. In fact, the (X, y) data itself only shows up in the reward computation in (1), which is a weak, aggregate learning signal. Thus, DSR trains *from scratch* for each new SR problem and cannot generalize to new problems.

Neural Symbolic Regression that Scales (NeSymReS). NeSymReS (Biggio et al., 2021) tackles the problem of *generalization* in SR: it seeks a *single* model that, once trained, produces a distribution over solutions to *any* SR problem. To accomplish this, the architecture uses a set transformer (Lee et al., 2019) component to learn an encoding of the (X, y) data. A second transformer network then maps the encoding to a distribution over expressions using autoregressive sampling. This model is trained end-to-end via supervised learning, using a large dataset of SR problems (input data (X, y) and corresponding ground truth label τ) produced by a programmatic expression generator (Lample and Charton, 2019). Given (X, y) at test time, beam search is used to find high-likelihood expressions. Treating test time as an inference problem reinforces the authors' claim that pre-training alone is sufficient to solve many SR problems. However, the lack of fine-tuning and the use of beam search means that NeSymReS may not be optimized for each test problem.

Genetic programming. Genetic programming (GP) refers to a well-established class of evolutionary algorithms commonly applied to SR in which a set or "population" of mathematical expressions is maintained and "evolved" over time using genetic operators such as mutation, crossover, and selection. We refer the reader to Langdon and Poli (2013) for an overview of GP. In this work, we consider only standard GP (as defined in Chapter 7 of Bäck et al. (2018)); however, our integration of GP into the unified framework applies to virtually any GP method. GP excels at exploring the search space to find complex expressions, but does not employ a neural network or other parameterized model to *learn* over time. This limitation was addressed in Mundhenk et al. (2021), where GP was combined with a learned expression generator, but not in concert with other methods considered here.

SINDy. SINDy is a simple, computationally expedient, and popular method for SR, particularly for learning systems of differential equations (Brunton et al., 2016). SINDy learns functions of the form $g = \sum_{i=1}^{k} \beta_i \phi_i = \Phi \beta^\intercal$, where $\Phi := (\phi_1, \ldots, \phi_k)$ comprise a library of k user-defined basis functions, each $\phi_i : \mathbb{R}^n \to \mathbb{R}$, and $\beta = (\beta_1, \ldots, \beta_k)$ is a vector of learned scalar coefficients. While SINDy employs LASSO (Tibshirani, 1996) to learn the values of β_i, we consider this a design choice for any arbitrary sparse linear regression method. While effective at quickly learning sparse expressions, restricting to a linear combination of pre-defined basis functions results in a massively reduced search space, precluding such functional forms as rationals and compositions. In our unified framework, we abstract the entire process of learning a linear model into one single discrete token, thereby restoring the nonlinear, combinatorial search space while retaining the ability to quickly learn sparse linear sub-components.

Table 1: Key capabilities and limitations of the five unified methods.

Method	Key capability	Key limitation
AIF	Exploits (X, y) data to simplify a SR problem into lower-dimensional sub-problems.	Underlying SR algorithm resorts to simple BF/PF.
DSR	Neural network learns over time, with the ability to incorporate *in situ* constraints.	Does not exploit the (X, y) data, which only appears in the (weak) reward function.
NeSymReS	Leverages big data, learning from many other problems by conditioning on the (X, y) data.	Limited to 3 dimensions, and does not fine-tune the search to a particular SR problem.
GP	Rapidly explores the search space via genetic operators.	Does not employ a neural network to learn over time.
SINDy	Quickly learns sparse coefficients of a linear combination of basis functions.	Operates on a dramatically reduced search space.

3 Related Work

In addition to AIF, DSR, NeSymReS, GP, and SINDy described above, we describe several other related SR methods that are not part of our unified framework. EQL$^{\div}$ presents a unique deep-learning framework for SR that employs neural networks whose activation functions are elementary operators, enabling end-to-end differentiable training (Sahoo et al., 2018). GrammarVAE encodes

from and decodes to parse trees that explicitly enforces syntactic correctness of generated expressions (Kusner et al., 2017). As end-to-end architectures, EQL$^{\div}$ and GrammarVAE would not facilitate straightforward integration of any of the five methods except AIF. The idea of pre-training networks for use in neural-guided search, used by NeSymReS and our unified framework, is inspired by Bello et al. (2017), who apply a similar method to the traveling salesman problem.

Previous work on hybrid approaches to SR mainly focus on improving a single aspect of one approach, such as alleviating the challenge of high dimensional data by extracting features for GP (Icke and Bongard, 2013), or improving the starting population of GP iterations (Mundhenk et al., 2021). Often, the motivation for combining separate methods comes from specific problem settings, such as the case of datasets with missing values (Al-Helali et al., 2018) or noisy high-dimensional measurements in systems with known physical constraints (Reinbold et al., 2021). In contrast, the unification in our work serves to enhance multiple general capabilities of SR, such as generalization, combinatorial search and optimization, and single-token abstraction of function spaces.

Our empirical analysis heavily leverages SRBench, a reproducible and open-source platform for benchmarking new SR methods against 14 established methods on accuracy and complexity for 122 real-world datasets and exact symbolic correctness for 130 ground-truth problems with varying noise levels (La Cava et al., 2021). SRBench has seen rapid adoption by recent work that use it to evaluate improvements in accuracy, exact solution rate, and solution complexity (Kamienny et al., 2022; Virgolin and Bosman, 2022; Zhang et al., 2022).

Lastly, we note a similarity between this work and Rainbow, a work that combines several deep reinforcement learning methods into a unified algorithm and performs extensive ablation studies (Hessel et al., 2018). Analogous to Rainbow's use of Deep Q Networks (DQN) as a base framework into which it integrates several improvements, this work uses DSR as a base framework into which we integrate AIF, NeSymReS, GP, and SINDy.

4 Methods

Our overall unification strategy is to employ individual methods as connected but non-overlapping modules in an algorithmic framework that leverages the key capabilities of each method. This is visualized in Figure 1. In an initial offline stage, pre-training a parametric controller model on a large dataset facilitates generalization to test problem instances (NeSymReS). In the online stage, given a specific problem instance, a recursive decomposition module (AIF) exploits modularity of mathematical expressions to produce sub-problems of lower dimensionality for the main trunk of uDSR. Within the trunk, an RNN continually learns over many iterations (DSR) and provides good candidates to seed the starting population for genetic programming (GP); high fitness populations produced by GP are combined with elite candidates for controller parameter updates via risk-seeking policy gradient. Finally, the RNN is permitted the use of a powerful symbolic token that encapsulates an entire space spanned by basis functions (SINDy).

Below we describe the integration of each module in detail. Algorithm 1 provides high-level pseudocode for uDSR. More detailed pseudocode is available in Appendix A Algorithm 2.

Table 2: Feature compatibility matrix for uDSR and the five integrated methods. FPO: Floating-point optimization.

	Recursive simplification	Search space pruning	Leverages big data	Evolutionary search	Sparse linear FPO	Nonlinear FPO	Neural-guided search
AIF	✓						✓
DSR		✓				✓	✓
NeSymReS			✓			✓	✓
GP				✓		✓	
SINDy					✓		
uDSR	✓	✓	✓	✓	✓	✓	✓

Beginning with DSR. For DSR, we begin with the method proposed by Petersen et al. (2021a), including two improvements to exploration (soft length prior and hierarchical entropy regularizer) introduced in follow-up works (Landajuela et al., 2021a,b). DSR includes a number of *in situ* priors and constraints to the search space, which help bias and/or prune the search, and have been shown to greatly improve performance (Landajuela et al., 2021b). We provide a complete list of DSR priors and constraints in Appendix B, including several new constraints required to unify all five methods

(described below). DSR also includes a CONST token that represents an arbitrary floating-point constant, whose value is optimized using a nonlinear optimizer, e.g. L-BFGS-B (Zhu et al., 1997).

Integrating AIF via recursive simplifications. Our key insight to integrate AIF (Udrescu et al., 2020) is that its recursive problem simplification steps can be viewed as orthogonal to the underlying SR algorithm. While AIF resorts to BF/PF, we propose using a hybrid of the other four integrated algorithms (which we continue to describe in subsequent sections). To integrate AIF, we consider the starting SR problem as $\mathcal{P}_1$, a d-dimensional SR problem. One application of AIF's simplification step produces either one sub-problem $\mathcal{P}_2$, with dimensionality $d_2 < d$, or two sub-problems, $\mathcal{P}_2$ and $\mathcal{P}_3$, with dimensionalities d_2 and d_3 such that $d_2 + d_3 = d$, as well as metadata $\mathcal{M}$ containing information required to recombine sub-solutions back to to the full solution of $\mathcal{P}_1$. Recursive simplifications are applied until sub-problems at the leaf nodes of the recursion tree are each 1-dimensional. For each sub-problem $\mathcal{P}_i$ (including the starting problem, $\mathcal{P}_1$), we apply our hybrid SR algorithm to its corresponding dataset, $(X, y)_i$. Finally, using the metadata from each simplification step, solutions from all sub-problems are recombined to form a single, final solution to the original problem, $\mathcal{P}_1$.

Integrating SINDy via LINEAR token. Our design goal for integrating SINDy (Brunton et al., 2016) is to retain the ability to quickly learn linear sub-components of an expression, without sacrificing the combinatorial search space. That is, while SINDy finds expressions of the form $g(x) = \Phi(x)\beta^\mathsf{T}$, we seek to search the more generic space $f(x) = F(x, g(x))$, where F is a nonlinear function subject to certain constraints (see below), while still using linear regression to determine the coefficients of $g(x)$. We accomplish this by proposing to represent $g(x)$ as a special type of terminal token, which we call LINEAR. LINEAR serves as a placeholder for a SINDy-like solution, $\Phi(x)\beta^\mathsf{T}$, where the values of the coefficients β are determined only when a traversal has terminated. Thus, we abstract the entire process of learning a linear model into a single discrete token. In our approach, the function $F(x, w)$ must admit an expression tree with a single argument token for w and there must exist $G(x, v)$ such that $y = F(x, G(x, y)),\ \forall x, y \in \mathbb{R}^d \times \mathbb{R}$.

Given an expression $f(x) = F(x, \Phi(x)\beta^\mathsf{T})$, with F as described above, let $\overline{f}(\Phi\beta^\mathsf{T})$ denote the view of f as a function of β only. Naively, one could solve a nonlinear optimization problem to determine the values of β, just as DSR and NeSymReS optimize CONST tokens. However, nonlinear optimization is computationally expensive, scales poorly with more coefficients, and is challenging to induce sparsity. Instead, we consider the following *convex* optimization problem, for which there are efficient sparse linear solvers:

$$\underset{\beta \in \mathbb{R}^k}{\arg\min}\, L(\beta) := \|\Phi(X)\beta^\mathsf{T} - \overline{f}^{-1}(y)\|_2^2. \tag{2}$$

We attain this formulation by solving the equation $y = F(X, \text{LINEAR})$ for LINEAR, which gives $\text{LINEAR} = G(X, y)$. Since this requires taking the inverse of all ancestor operators of LINEAR—binary operators are reduced to unary operators by lexical closure, elaborated more programmatically below—we use $\overline{f}^{-1}(y) := G(X, y)$ to denote the view of G as a function of y only.

Equation (2) exhibits an attractive property that enables LINEAR to exactly recover non-linear expressions with linear sub-components. When LINEAR is chosen at the correct position in a traversal, in the sense that the true computational graph contains a linear combination of basis functions in the corresponding subgraph, we have that $\overline{f}^{-1}(y)$ exactly equals that linear combination. For example, given (X, y) data generated from a ground-truth expression $f^\star(x) = \sin(x_1) + \exp(1.2x_1^3 + 3.4x_1x_2)$, and a basis containing all monomials up to degree 3, the traversal $\tau = [+, \text{SIN}, x_1, \text{EXP}, \text{LINEAR}]$ yields $\overline{f}^{-1}(y) = \log(y - \sin(x_1))$, and the corresponding solution to (2) *exactly* recovers $f^\star$.

Programmatically, we solve (2) using the expression tree for τ and the following procedure (illustrated in Appendix C Figure 5; pseudocode is provided in Appendix A Algorithm 2): First, partially execute the tree by evaluating all nodes that are not direct ancestors of LINEAR. Second, redefine binary operators as unary operators by fixing (or "binding") the one pre-computed argument (this is called *partial application*—a type of *lexical closure*—in computer science). Third, successively apply the root node's inverse operation on the y data until LINEAR is the only token in the tree, resulting in $\overline{f}^{-1}(y)$. Finally, perform linear fit on (2) using the transformed data, $(X, \overline{f}^{-1}(y))$.

This procedure for abstracting a linear model into a single operator exhibits several complications: (1) it only works if there is exactly one LINEAR token, (2) it does not work in the presence of other placeholder tokens, e.g. the CONST token used in DSR, and (3) it does not work if LINEAR has

Algorithm 1 Unified deep symbolic regression. Color key: AIF, NeSymReS, DSR, GP, SINDy

input Symbolic regression problem $\mathcal{P}$ consisting of tabular data (X, y)
output Best fitting expression $\tau^\star$
parameters Pre-trained controller and encoder parameters $(\theta^\star, \psi^\star)$, objective function $J(\theta, \psi)$

```
P_1, ..., P_m ← use AIF to recursively simplify P into m sub-problems
for each sub-problem P_i and corresponding dataset (X, y)_i do
    θ = θ*, ψ* ← load pre-trained encoder and controller parameters
    while budget not exceeded do
        T_RNN ← sample N expressions from p(τ|θ, ψ*, (X, y)_i)
        T_GP ← evolve population by performing S generations of GP
        T ← combine T_RNN and top k expressions from T_GP
        T ← compute coefficients of LINEAR tokens
        θ ← train controller on T according to J(θ, ψ*)
    τ^(i) ← store best expression for problem P_i
τ* ← use AIF to combine solutions τ^(1), ..., τ^(m) into final solution
return τ*
```

non-invertible ancestors (e.g. SIN), as $y = F(X, \text{LINEAR})$ might not have a unique solution on LINEAR. Fortunately, seamlessly incorporating these restrictions into the search space is precisely what the DSR algorithm excels at. To this end, we enable LINEAR into our framework by introducing three constraints to the search space (further detailed in Appendix B): (1) there can be at most one LINEAR token; (2) the LINEAR token and CONST token are mutually exclusive—one cannot appear if the other is present; and (3) the LINEAR token cannot be the descendant of a non-invertible unary token, e.g. SIN. In this work, we consider the set of basis functions Φ given by the monomials of degree up to D and solve (2) using a sparsity-inducing least square method (see Appendix C for details).

Lastly, we observe that the original SINDy method is restored when $F(x, w) = w$. In that case, we recover the function $f(x) = \Phi(x)\beta^\intercal$ with trivial traversal $\tau = [\text{LINEAR}]$. Thus, our method expands the search space of the original SINDy method by allowing nonlinear function compositions. For instance, $F(x, w) = \frac{1}{w}$ leads to $f(x) = \frac{1}{\Phi(x)\beta^\intercal}$, which is outside SINDy's search space.

Integrating GP via population seeding. To our knowledge, hybridization of DSR and GP is the only combination of the five methods that has been explored to date. To summarize, we hybridize DSR with GP by abstracting the GP algorithm as an inner optimization loop within DSR. Each time DSR samples a batch of expressions, we use this batch as the starting population for an inner-loop GP algorithm. The top samples at the end of GP are combined with the DSR batch for training the neural network. This process leverages GP's ability to rapidly explore the search space, while using DSR's persistent neural network to mitigate the disadvantages of GP's non-parametric nature.

Integrating NeSymReS via large-scale pre-training. We integrate NeSymReS by merging the RNN used in DSR with a set transformer in an encoder-decoder fashion. The set transformer encodes a given dataset (X, y) into a latent representation h_0. This representation is passed as initial state for the RNN, which decodes it into an expression τ. The resulting architecture models the distribution $p(\tau|\theta, \psi, (X, y))$, where θ and ψ are the parameters of the RNN and set transformer, respectively. We pre-train the model end-to-end on many SR problems, using ground truth expressions, resulting in parameters $\theta^\star, \psi^\star$. Given a new dataset (X', y'), we propose to search for a symbolic expression by *fine-tuning* $p(\tau|\theta, \psi^\star, (X', y'))$ using reinforcement learning, with initial conditions $\theta_0 = \theta^\star$.

The pre-training phase requires a large dataset comprising SR problems, each defined by a set of points (X, y) along with true expression τ. To scale to arbitrary dataset size, we seek a problem instance generator $p_\mathcal{G}(\tau, (X, y)) = p_\mathcal{G}((X, y)|\tau)p_\mathcal{G}(\tau)$, from which we can sample expressions τ and corresponding (X, y) data. Full details of the sampling process used by $p_\mathcal{G}$ are provided in Appendix D. Briefly, we leverage the fact that the distribution induced by DSR includes a prior (Petersen et al., 2021b). To generate expressions, $p_\mathcal{G}(\tau) = p(\tau|\theta = \mathbf{0})$ directly samples from this prior, ignoring emissions from the RNN. Thus, the algorithm bootstraps or "imagines" expressions that it then uses for training data, rather than relying on an external expression generator as in NeSymReS. After sampling τ, $p_\mathcal{G}((X, y)|\tau)$ is obtained following the strategy in Biggio et al. (2021).

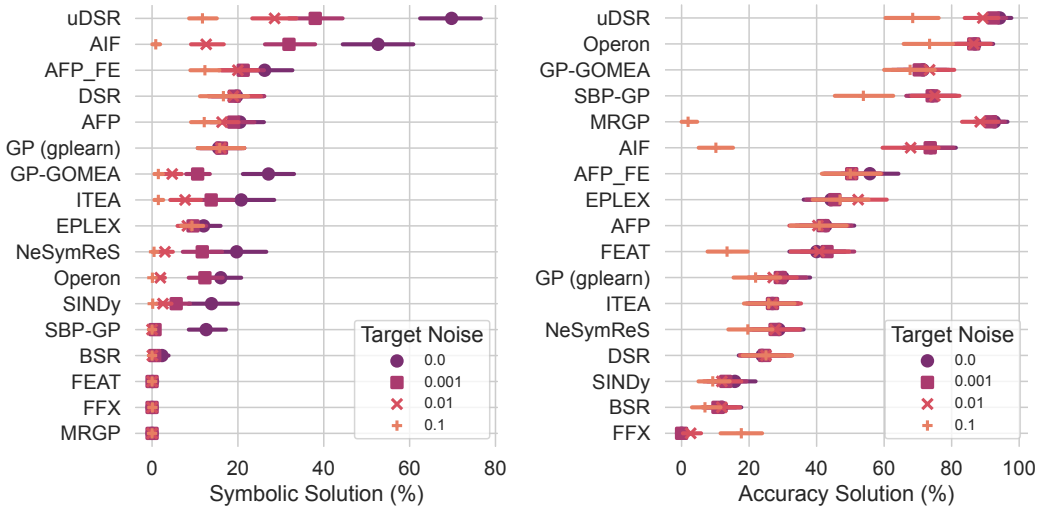

Figure 2: SRBench-generated comparisons of symbolic (left) and accuracy-based (right) solution rates for uDSR and 16 baseline SR methods, averaged across 130 ground-truth SR problems and four different noise levels.

In this work, we explore two different objectives for model pre-training. First, following Biggio et al. (2021), we consider the supervised learning objective:

$$J_{\text{SL}}(\theta, \psi) := \mathbb{E}_{\tau_g, (X,y)_g \sim p_{\mathcal{G}}} \left[\log p\left(\tau_g | \theta, \psi, (X, y)_g\right) \right].$$

During fine-tuning, ground-truth labels will not be available. To mitigate an objective mismatch between pre-training and fine-tuning objectives, we alternatively explore pre-training *à la* reinforcement learning Bello et al. (2017). Specifically, we consider:

$$J_{\text{RL}}(\theta, \psi) := \mathbb{E}_{(X,y)_g \sim p_{\mathcal{G}}} \left[\mathbb{E}_{\tau \sim p(\tau | \theta, \psi, (X,y)_g)} \left[R(\tau, (X, y)_g) \right] \right]. \tag{3}$$

In practice, this objective is maximized using a novel formulation of the risk-seeking policy gradient for multi-task learning (see Appendix D for details).

5 Experiments

Pre-training. We pre-train four models: two trained using SL and two trained using RL, each with and without the LINEAR token. Details of the pre-training setup can be found in Appendix D.

Experimental setup using SRBench. We empirically assess uDSR using SRBench (La Cava et al., 2021), an open-source and reproducible pipeline for benchmarking SR algorithms. SRBench features 130 problems with hidden ground-truth analytic solutions and 122 real-world datasets with no known analytic model ("black-box" problems) from the PMLB database (Olson et al., 2017). By running uDSR through the SRBench pipeline, we enable direct comparison to its curated results of 14 contemporary SR methods. The 14 baselines are described in Appendix E.

For all experiments, we use a minimal set of tokens: $+$, $-$, $\times$, $\div$, SIN, COS, EXP, LOG, SQRT, 1.0, CONST, and (except for the appropriate ablations) LINEAR. For simplicity, our choice of Φ (basis functions for LINEAR) includes only polynomial terms up to degree 3. Hyperparameters are listed in Appendix F Table 3. When possible, we reused the published hyperparameters from the five integrated methods.

We ran uDSR via SRBench on the 252 datasets with 10 random seeds each. Each run was performed for up to 2,000,000 expression evaluations per sub-problem, or until a 24 hr walltime limit was reached. For ground-truth problems, SRBench computes two metrics, each averaged across the 130 problems: (1) *symbolic solution* rate, based on symbolic equivalence using SymPy (Meurer et al., 2017) and (2) *accuracy solution* rate, defined as achieving an $R^2 > 0.999$ on held-out test data. For black-box problems, SRBench reports accuracy (R^2 on held-out test data) vs complexity (number of nodes in the SymPy-parsed expression tree).

Performance results. Figure 2 shows that uDSR outperforms all other 14 benchmarked methods both in symbolic solution (by a large margin) and accuracy solution rates. Figure 3 shows that uDSR falls on the Pareto frontier, alongside Operon (Kommenda et al., 2020) (at higher accuracy and complexity), and DSR and SINDy (at lower accuracy and complexity).

Notably, the previously published SRBench results had no clear winner across the three categories of (1) symbolic solution, (2) accuracy solution, and (3) Pareto efficiency. The top performers for symbolic solution rate (namely, AIF) differed from those for accuracy solution rate (namely, Operon and MRGP). Only Operon appeared as a top performer in two categories. However, in this work, we see that uDSR is highest in all three categories.

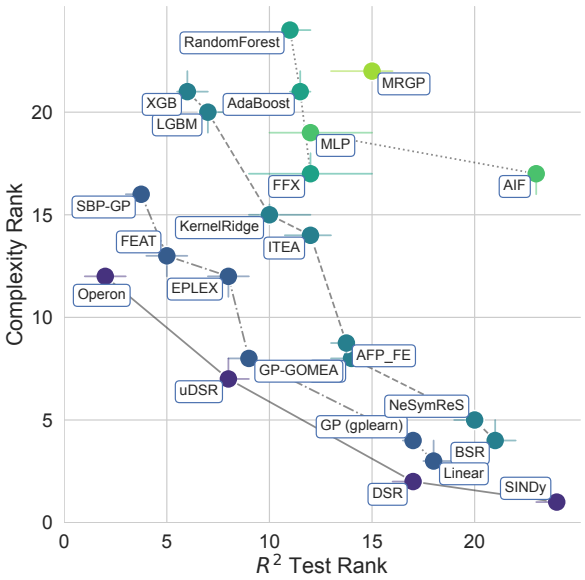

Figure 3: SRBench-generated comparison of test R^2 (on held-out test data) vs complexity for uDSR and 22 baseline regression/SR methods, averaged across 122 black-box SR problems.

6 Ablations and Discussion

Ablations. Since uDSR integrates many disparate components, it is critical to assess the relative contributions of each component in various contexts. To identify synergies, anti-synergies, and diminishing returns among the integrated methods, we perform a large-scale study of *all combinations* of the five integrated methods. (Standard "leave-one-out" style ablations in which we begin with the full uDSR and ablate one component at a time do not provide this full picture.) There are 48 total combinations: AIF $\in$ [on, off] $\times$ GP $\in$ [on, off] $\times$ LINEAR $\in$ [on, off] $\times$ pre-training $\in$ [SL, RL, off] $\times$ DSR $\in$ [on, off]. We define ablating each components as follows: For AIF, "off" means that we do not perform recursive problem-simplification; only the root problem is considered. For GP, "off" means we do not perform inner-loop GP algorithms between batches. For LINEAR, "off" means we exclude the LINEAR token from the library. For pre-training, SL and RL refer to whether we use the model pre-trained using SL or RL; "off" means we do not use any pre-trained model, and the controller architecture does not include the set transformer component. For DSR, "off" means the learning rate for the neural network is set to zero; notably, the controller is still used for sampling, including all priors and constraints. For each ablation, we run on the 130 ground-truth SRBench problems with 10 random seeds each ($48 \times 130 \times 10 = 62,400$ total runs), using a maximum of 500,000 expression evaluations per sub-problem.

Figure 4 shows the symbolic solution rate and accuracy solution rate computed by SRBench, with labels for select ablations we refer to in the subsequent discussion. Generally, performance increases with the number of enabled components, N. For a fixed N, GP and SINDy tend to yield the largest marginal performance gains. Refer to Appendix G Figure 9 for a visualization of the marginal performance gains yielded by each component.

Recommendations. The full uDSR algorithm, using all components and SL pre-training, achieves the best symbolic and accuracy success on SRBench. Nonetheless, we believe the right choice of SR algorithm should be guided by use case. Thus, based on the extensive ablation studies presented above, we provide our recommendations for the five integrated methods.

AIF's recursive simplifications have the attractive property of providing *strict improvement* over the base SR algorithm, since the root sub-problem in the sub-problem tree is simply the original problem; sub-problems further down the tree only have the ability to improve the end result. This property comes at the expense of incurring a roughly m-fold increase in computational cost, where m is the number of identified sub-problems. Empirically, our ablations confirm the strict improvement (e.g., compare 'A' with 'DSR' in Figure 4), though performance gains exhibit diminishing returns as the

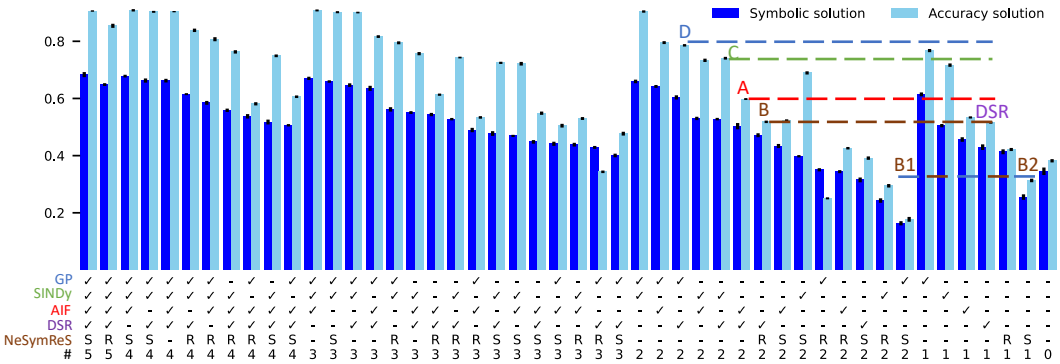

Figure 4: Combinatorial ablations of uDSR components on 130 SRBench ground-truth problems. Dashed lines indicate the performance of the best combination. Indicators below bars denote which components are enabled for each ablation. ✓: Component enabled. -: Component disabled. S: Pre-training enabled using SL. R: Pre-training enabled using RL. #: Number of enabled components. Error bars represent standard error across 10 random seeds per problem. Colored labels are referred to in the main text. Ablation 'D' is equivalent to Mundhenk et al. (2021). Ablation 'DSR' is equivalent to Landajuela et al. (2021b).

number of integrated methods increases. Thus, we suggest inclusion of AIF for use cases in which computational cost is not an important bottleneck. Finally, we note that we only considered AIF without its dimensionality analysis component, as this is really a pre-processing step orthogonal to the SR algorithm. For real-world problems with physical units, we highly recommend that users first perform dimensional analysis on their SR problem.

The best results were obtained using SL pre-training with all other components enabled. However, in general, pre-training across the ablations yielded mixed results. For example, the only significant anti-synergy among all components occurs between SL pre-training and GP (compare 'B1' and 'B2' in Figure 4). To gain further insight into which use cases may benefit from pre-training, we inspected the initial rewards for each pre-training modality *before* fine-tuning. As shown in Appendix D Figure 6, both SL and RL pre-trained models yield higher reward than randomly initialized models. This is especially the case for RL pre-training, where the traces of the initial expressions (see Appendix D Figure 8) appear very close to the real data. Thus, pre-training can be desirable for use cases with very low budgets (e.g. zero-shot learning), a point reinforced empirically by Biggio et al. (2021). However, as shown in Appendix D Figure 7, pre-trained models may have tendencies to become too "sharp" (i.e., exhibiting low entropy), resulting in reduced exploration that can hinder performance during fine-tuning. We conclude that to be effective in combination with other solution strategies in SR, pre-trained models must strike the delicate balance between high initial reward and sufficient initial diversity.

Both the LINEAR token and the inclusion of our GP component almost always provide large improvements in both symbolic solution and accuracy solution (e.g., compare 'C' and 'D' with 'DSR' in Figure 4). They also do not incur significant additional computational cost given a fixed budget of expressions (see Appendix H Table 4). Thus, we highly recommend their inclusion for most use cases. A limitation of the LINEAR token, and to a lesser extent the use of GP, is that it is inefficient in improving accuracy relative to its complexity, which we show in Appendix I Figure 10. Thus, users may not want LINEAR or GP for use cases in which they seek a low-complexity expression.

7 Conclusion

We introduce a modular framework for SR in which we carefully integrate five standalone SR algorithms in an attempt to maximize the benefits of each and cover each other's weaknesses. Our unification strategy focuses on *abstraction* of the constituent methods: using DSR as a baseline, we abstract AIF as an algorithmic wrapper, NeSymReS as a pre-training step, GP as an inner-optimization loop, and SINDy as a single optimizable token. Using the SRBench pipeline for benchmarking SR algorithms, uDSR demonstrates state-of-the-art performance across 252 benchmark problems. For future work, we leave open the questions of using more advanced modules for GP, such as GP-GOMEA (Virgolin et al., 2017), or controller architectures (such as transformers as alternatives to the RNN), as well as the possible benefits from inter-algorithm hyperparameter tuning. We hope that future work in SR considers additional strategies for hybridizing and/or abstracting SR algorithms.

References

Abolafia, D. A., Norouzi, M., Shen, J., Zhao, R., and Le, Q. V. (2018). Neural program synthesis with priority queue training. arXiv preprint arXiv:1801.03526.

Al-Helali, B., Chen, Q., Xue, B., and Zhang, M. (2018). A hybrid gp-knn imputation for symbolic regression with missing values. In Australasian Joint Conference on Artificial Intelligence, pages 345–357. Springer.

Arnaldo, I., Krawiec, K., and O'Reilly, U.-M. (2014). Multiple regression genetic programming. In Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation, pages 879–886.

Bäck, T., Fogel, D. B., and Michalewicz, Z. (2018). Evolutionary computation 1: Basic algorithms and operators. CRC press.

Bello, I., Pham, H., Le, Q. V., Norouzi, M., and Bengio, S. (2017). Neural combinatorial optimization with reinforcement learning.

Biggio, L., Bendinelli, T., Neitz, A., Lucchi, A., and Parascandolo, G. (2021). Neural symbolic regression that scales. In International Conference on Machine Learning, pages 936–945. PMLR.

Brunton, S. L., Proctor, J. L., and Kutz, J. N. (2016). Discovering governing equations from data by sparse identification of nonlinear dynamical systems. Proceedings of the national academy of sciences, 113(15):3932–3937.

de França, F. O. and Aldeia, G. S. I. (2021). Interaction–transformation evolutionary algorithm for symbolic regression. Evolutionary computation, 29(3):367–390.

Fortin, F.-A., De Rainville, F.-M., Gardner, M.-A. G., Parizeau, M., and Gagné, C. (2012). Deap: Evolutionary algorithms made easy. The Journal of Machine Learning Research, 13(1):2171–2175.

Friedman, J., Hastie, T., and Tibshirani, R. (2010). Regularization paths for generalized linear models via coordinate descent. Journal of statistical software, 33(1):1.

Hessel, M., Modayil, J., Van Hasselt, H., Schaul, T., Ostrovski, G., Dabney, W., Horgan, D., Piot, B., Azar, M., and Silver, D. (2018). Rainbow: Combining improvements in deep reinforcement learning. In Thirty-second AAAI conference on artificial intelligence.

Icke, I. and Bongard, J. C. (2013). Improving genetic programming based symbolic regression using deterministic machine learning. In 2013 IEEE Congress on Evolutionary Computation, pages 1763–1770. IEEE.

Jin, Y., Fu, W., Kang, J., Guo, J., and Guo, J. (2019). Bayesian symbolic regression. arXiv preprint arXiv:1910.08892.

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

Karpathy, A. and Fei-Fei, L. (2015). Deep visual-semantic alignments for generating image descriptions. In Proceedings of the IEEE conference on computer vision and pattern recognition, pages 3128–3137.

Kim, J. T., Larma, M. L., and Petersen, B. K. (2021). Distilling wikipedia mathematical knowledge into neural network models. arXiv preprint arXiv:2104.05930.

Kommenda, M., Burlacu, B., Kronberger, G., and Affenzeller, M. (2020). Parameter identification for symbolic regression using nonlinear least squares. Genetic Programming and Evolvable Machines, 21(3):471–501.

Koza, J. R. (1994). Genetic programming as a means for programming computers by natural selection. Statistics and computing, 4(2):87–112.

Kusner, M. J., Paige, B., and Hernández-Lobato, J. M. (2017). Grammar variational autoencoder. In International conference on machine learning, pages 1945–1954. PMLR.

La Cava, W., Helmuth, T., Spector, L., and Moore, J. H. (2019). A probabilistic and multi-objective analysis of lexicase selection and ε-lexicase selection. Evolutionary Computation, 27(3):377–402.

La Cava, W., Orzechowski, P., Burlacu, B., de Franca, F. O., Virgolin, M., Jin, Y., Kommenda, M., and Moore, J. H. (2021). Contemporary symbolic regression methods and their relative performance. In Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1).

La Cava, W., Singh, T. R., Taggart, J., Suri, S., and Moore, J. H. (2018). Learning concise representations for regression by evolving networks of trees. arXiv preprint arXiv:1807.00981.

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

Landajuela, M., Petersen, B. K., Kim, S., Santiago, C. P., Glatt, R., Mundhenk, N., Pettit, J. F., and Faissol, D. (2021a). Discovering symbolic policies with deep reinforcement learning. In International Conference on Machine Learning, pages 5979–5989. PMLR.

Landajuela, M., Petersen, B. K., Kim, S. K., Santiago, C. P., Glatt, R., Mundhenk, T. N., Pettit, J. F., and Faissol, D. M. (2021b). Improving exploration in policy gradient search: Application to symbolic optimization. arXiv preprint arXiv:2107.09158.

Langdon, W. B. and Poli, R. (2013). Foundations of genetic programming. Springer Science & Business Media.

Lee, J., Lee, Y., Kim, J., Kosiorek, A., Choi, S., and Teh, Y. W. (2019). Set transformer: A framework for attention-based permutation-invariant neural networks. In International Conference on Machine Learning, pages 3744–3753. PMLR.

Luke, S. and Panait, L. (2001). A survey and comparison of tree generation algorithms. In Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001), pages 81–88. Citeseer.

McConaghy, T. (2011). Ffx: Fast, scalable, deterministic symbolic regression technology. In Genetic Programming Theory and Practice IX, pages 235–260. Springer.

Meurer, A., Smith, C. P., Paprocki, M., Čertík, O., Kirpichev, S. B., Rocklin, M., Kumar, A., Ivanov, S., Moore, J. K., Singh, S., et al. (2017). Sympy: symbolic computing in python. PeerJ Computer Science, 3:e103.

Mundhenk, T., Landajuela, M., Glatt, R., Santiago, C. P., Petersen, B. K., et al. (2021). Symbolic regression via deep reinforcement learning enhanced genetic programming seeding. Advances in Neural Information Processing Systems, 34:24912–24923.

Olson, R. S., La Cava, W., Orzechowski, P., Urbanowicz, R. J., and Moore, J. H. (2017). Pmlb: a large benchmark suite for machine learning evaluation and comparison. BioData mining, 10(1):1–13.

Pawlak, T. P., Wieloch, B., and Krawiec, K. (2014). Semantic backpropagation for designing search operators in genetic programming. IEEE Transactions on Evolutionary Computation, 19(3):326–340.

Petersen, B. K., Larma, M. L., Mundhenk, T. N., Santiago, C. P., Kim, S. K., and Kim, J. T. (2021a). Deep symbolic regression: Recovering mathematical expressions from data via risk-seeking policy gradients. In International Conference on Learning Representations.

Petersen, B. K., Santiago, C., and Landajuela, M. (2021b). Incorporating domain knowledge into neural-guided search via in situ priors and constraints. In 8th ICML Workshop on Automated Machine Learning (AutoML).

Popova, M., Shvets, M., Oliva, J., and Isayev, O. (2019). Molecularrnn: Generating realistic molecular graphs with optimized properties. arXiv preprint arXiv:1905.13372.

Reinbold, P. A., Kageorge, L. M., Schatz, M. F., and Grigoriev, R. O. (2021). Robust learning from noisy, incomplete, high-dimensional experimental data via physically constrained symbolic regression. Nature communications, 12(1):1–8.

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

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

Schmidt, M. and Lipson, H. (2011). Age-fitness pareto optimization. In Genetic programming theory and practice VIII, pages 129–146. Springer.

Schmidt, M. D. and Lipson, H. (2008). Coevolution of fitness predictors. IEEE Transactions on Evolutionary Computation, 12(6):736–749.

Tibshirani, R. (1996). Regression shrinkage and selection via the lasso. Journal of the Royal Statistical Society: Series B (Methodological), 58(1):267–288.

Udrescu, S.-M., Tan, A., Feng, J., Neto, O., Wu, T., and Tegmark, M. (2020). Ai feynman 2.0: Pareto-optimal symbolic regression exploiting graph modularity. Advances in Neural Information Processing Systems, 33:4860–4871.

Virgolin, M., Alderliesten, T., and Bosman, P. A. (2019). Linear scaling with and within semantic backpropagation-based genetic programming for symbolic regression. In Proceedings of the genetic and evolutionary computation conference, pages 1084–1092.

Virgolin, M., Alderliesten, T., Witteveen, C., and Bosman, P. A. (2017). Scalable genetic programming by gene-pool optimal mixing and input-space entropy-based building-block learning. In Proceedings of the Genetic and Evolutionary Computation Conference, pages 1041–1048.

Virgolin, M., Alderliesten, T., Witteveen, C., and Bosman, P. A. (2021). Improving model-based genetic programming for symbolic regression of small expressions. Evolutionary computation, 29(2):211–237.

Virgolin, M. and Bosman, P. A. (2022). Coefficient mutation in the gene-pool optimal mixing evolutionary algorithm for symbolic regression. arXiv preprint arXiv:2204.12159.

Zhang, H., Zhou, A., Qian, H., and Zhang, H. (2022). Ps-tree: A piecewise symbolic regression tree. Swarm and Evolutionary Computation, 71:101061.

Zhu, C., Byrd, R. H., Lu, P., and Nocedal, J. (1997). Algorithm 778: L-bfgs-b: Fortran subroutines for large-scale bound-constrained optimization. ACM Transactions on mathematical software (TOMS), 23(4):550–560.

Checklist

1. For all authors...
   (a) Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope? [Yes]
   (b) Did you describe the limitations of your work? [Yes] **Several limitations are discussed in Section 6.**
   (c) Did you discuss any potential negative societal impacts of your work? [N/A]
   (d) Have you read the ethics review guidelines and ensured that your paper conforms to them? [Yes]
2. If you are including theoretical results...
   (a) Did you state the full set of assumptions of all theoretical results? [N/A]
   (b) Did you include complete proofs of all theoretical results? [N/A]
3. If you ran experiments...
   (a) Did you include the code, data, and instructions needed to reproduce the main experimental results (either in the supplemental material or as a URL)? [Yes] **To preserve anonymity, code is included for reviewers as supplemental material; if accepted, this will be replaced with a public URL. Code includes SRBench launch scripts used to generate results.**
   (b) Did you specify all the training details (e.g., data splits, hyperparameters, how they were chosen)? [Yes] **Hyperparameters used and hyperparameter selection are described in Section 5 and Appendix F.**
   (c) Did you report error bars (e.g., with respect to the random seed after running experiments multiple times)? [Yes] **All figures and tables with results include error bars.**
   (d) Did you include the total amount of compute and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)? [Yes] **Computational resources used are described in Appendix J.**
4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets...
   (a) If your work uses existing assets, did you cite the creators? [Yes] **Software and dataset assets are described and cited in Appendix J. Our benchmarks use the PMLB database, cited in Section 5.**
   (b) Did you mention the license of the assets? [Yes] **Asset licences are described in Appendix J**
   (c) Did you include any new assets either in the supplemental material or as a URL? [N/A]
   (d) Did you discuss whether and how consent was obtained from people whose data you're using/curating? [N/A]
   (e) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? [N/A]
5. If you used crowdsourcing or conducted research with human subjects...
   (a) Did you include the full text of instructions given to participants and screenshots, if applicable? [N/A]
   (b) Did you describe any potential participant risks, with links to Institutional Review Board (IRB) approvals, if applicable? [N/A]
   (c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? [N/A]

Appendix

A Expanded pseudocode for uDSR

In Algorithm 2, we provide pseudocode for uDSR in more detail than Algorithm 1. Note that CONST and LINEAR optimization is only described in Lines 22-25 and Lines 26-31, respecitvely, but these also occur implicitly within GP operators (Line 19) and during RL pre-training (Line 10).

Algorithm 2 Unified deep symbolic regression. Color key: AIF, NeSymReS, DSR, GP, SINDy

input Symbolic regression problem $\mathcal{P}$ consisting of tabular data (X, y)
output Best fitting expression $\tau^\star$

1: // Phase I: Pre-training
2: Initialize controller with RNN parameters θ and encoder parameters ψ
3: **while** budget not exceeded **do**
4: $\mathcal{T}^\star \leftarrow \{\tau_i^\star \sim p_\mathcal{G}(\cdot)\}_{i=1}^B$ ▷ Sample B problem instances from prior
5: $\mathcal{D} \leftarrow \{(X, y)_i \sim \text{DatasetGenerator}(\tau_i^\star)\}_{i=1}^B$ ▷ Sample B corresponding (X, y) datasets
6: **if** SL pre-training **then**
7: $\theta, \psi \leftarrow$ train θ, ψ on $\mathcal{T}$ using SL ▷ Train using SL
8: **else if** RL pre-training **then**
9: $\mathcal{T} \leftarrow \{\tau_i^{(j)} \sim p(\cdot|\theta, \psi, (X, y)_i\}_{i,j=1}^{B,N_B}$ ▷ Sample N_B expressions for each dataset
10: $\theta, \psi \leftarrow$ train θ, ψ on $\mathcal{T}$ using RL ▷ Train using RL

11: // Phase II: Test problem
12: $\mathcal{P}_1, \ldots, \mathcal{P}_m \leftarrow$ RecursiveSimplify$(\mathcal{P})$ ▷ Use AIF to recursively simplify $\mathcal{P}$ into m sub-problems
13: **for** each sub-problem $\mathcal{P}_i$ and corresponding dataset $(X, y)_i$ **do**
14: $\theta, \psi \leftarrow \theta^\star, \psi^\star$ ▷ Load pre-trained contorller and encoder weights
15: **while** budget not exceeded **do**
16: $\mathcal{T}_{\text{RNN}} \leftarrow \{\tau^{(j)} \sim p(\cdot|\theta, \psi, (X, y)_i)\}_{j=1}^N$ ▷ Sample N expressions, conditioned on $(X, y)_i$
17: $\mathcal{T}_{\text{GP}}^{(0)} \leftarrow \mathcal{T}_{\text{RNN}}$ ▷ Seed GP with RNN batch
18: **for** $s = 1, \ldots, S$ **do**
19: $\mathcal{T}_{\text{GP}}^{(s)} \leftarrow \Gamma\left(\mathcal{T}_{\text{GP}}^{(s-1)}\right)$ ▷ Apply GP operator
20: $\mathcal{T}_{\text{GP}} \leftarrow$ top k from $\mathcal{T}_{\text{GP}}^{(1)} \cup \cdots \cup \mathcal{T}_{\text{GP}}^{(S)}$ ▷ Filter best k samples from GP
21: $\mathcal{T} \leftarrow \mathcal{T}_{\text{RNN}} \cup \mathcal{T}_{\text{GP}}$ ▷ Join RNN and best GP samples
22: **for** $j = 1, \ldots, N$ **do**
23: **if** CONST $\in \tau^{(j)}$ **then**
24: $\xi^\star \leftarrow \arg\max_\xi R(\tau^{(j)}; \xi)$ ▷ Maximize R w.r.t. ξ, e.g. with BFGS
25: $\tau^{(j)} \leftarrow$ ReplaceConstants$(\tau^{(j)}, \xi^\star)$ ▷ Replace placeholder constants

26: **if** LINEAR $\in \tau^{(j)}$ **then**
27: Let $y = f(X, \text{LINEAR})$ ▷ Instantiate expression
28: Find G such that LINEAR $= G(X, y)$ ▷ Solve equation for LINEAR
29: Compute $y' = G(X, y)$ ▷ Evaluate G on $(X, y)_i$ data
30: $\beta^\star \leftarrow \arg\min_\beta \| \sum_i \beta_i \phi_i(X) - y' \|_2^2$ ▷ Optimize β, e.g. using LASSO
31: $\tau^{(j)} \leftarrow$ ReplaceCoefficients$(\tau^{(j)}, \beta^\star)$ ▷ Replace LINEAR coefficients
32: $\theta \leftarrow$ train θ (with ψ fixed) on $\mathcal{T}$ ▷ Train using fine-tuning objective
33: $\tau^{(i)} \leftarrow$ store best expression for problem $\mathcal{P}_i$
34: $\tau^\star \leftarrow$ Combine$(\tau^{(1)}, \ldots, \tau^{(m)})$ ▷ Use AIF to combine m solutions into final solution
35: **return** $\tau^\star$

B List of *in situ* priors and constraints

Autoregressive sampling within DSR affords the opportunity to apply user-specified priors and constraints to the search space *in situ*, i.e., during the sampling process. Each step of autoregressive sampling, the DSR controller emits logits ℓ_θ. Before these logits are passed through a softmax, DSR computes a *prior* denoted $\ell_\circ$, which is an additional set of logits that are computed using all available information so far (namely, the partial traversal and the (X, y) data). The prior logits are added to the controller emissions before a token is sampled via $\tau_i \sim \text{Categorical}(\text{Softmax}(\ell_\theta + \ell_\circ))$. Priors allow users to incorporate *a priori* knowledge during the search. Notably, when a prior includes $-\infty$ entries, this constrains particular tokens from being sampled, thereby pruning the combinatorial search space. Such priors have been used by several works using DSR (Petersen et al., 2021a,b; Landajuela et al., 2021b; Mundhenk et al., 2021; Kim et al., 2021) as well as other non-SR works using similar autoregressive sampling procedures (Popova et al., 2019; Landajuela et al., 2021a).

These priors are enabled during pre-training, both by the problem instance generator and the controller, as well as during fine-tuning. When integrated with GP, constraints are applied *post hoc*. Specifically, if a genetic operator would result in an expression tree that violates any constraint, that genetic operator is reverted, and instead the parent expression tree is returned.

We first summarize priors that have already been included in works utilizing DSR:

- **Length constraint.** Following Petersen et al. (2021a), traversals are constrained to have a length no less than 4 and no more than 64. The minimum length constraint is implemented by constraining terminal tokens when selecting would would end the traversal before the minimum length. For example, given the partial traversal $\tau = [\text{SIN}]$, terminal tokens are constrained because choosing one would end the traversal with length 2. The maximum length constraint is implemented by constraining unary and/or binary tokens when selecting one, followed by selection of only terminal tokens, would result in a traversal that exceeds the maximum length. For example, given a partial traversal consisting of 31 instances of $+$, binary tokens would be constrained because choosing one, followed by 32 terminal tokens, would not complete the end the traversal by length 64.
- **Trigonometry constraint.** Following Petersen et al. (2021a), trigonometric tokens that would be descendants of other trigonometric tokens are constrained. For example, given the partial traversal $\tau = [+, x_1, \text{COS}, \div, 1.0]$, SIN and COS are constrained because they would be descendants of COS.
- **Constant constraint.** Following Petersen et al. (2021a), we constrain that constant tokens (namely, CONST and 1.0) cannot be the only unique child of a token, because the result would simply be another constant. For example, given the partial traversal $\tau = [+, \text{CONST}]$ or $\tau = [\text{SIN}]$, CONST and 1.0 are constrained.
- **CONST repeat constraint.** Following Petersen et al. (2021a), we limit the maximum number of CONST tokens to 3 per expression, since nonlinear optimization of CONST tokens can be computationally expensive.
- **Soft length prior.** Following Landajuela et al. (2021b), we include a soft length prior that biases expressions to have length somewhere between the minimum and maximum. See Landajuela et al. (2021b) for details.
- **Uniform arity prior.** Following Landajuela et al. (2021b), we include a prior that biases the controller to have an equal likelihood of selecting a binary, unary, or terminal token. See Landajuela et al. (2021b) for details.

Next, we describe priors that are newly introduced in this work:

- **Available input variables constraint.** Our architecture used for pre-training assumes a maximum of $d = 6$ input dimensions. However, not all problems include all 6 dimensions. Since the dimensionality of an SR problem is known at runtime, we constrain input variables beyond the dimensionality of the given problem. For example, x_5 and x_6 are always constrained for 4-dimensional problems. This constraint is enabled when using NeSymReS.
- **Constraints for LINEAR.** As discussed in Section 4, the LINEAR token requires three constraints. First, CONST and LINEAR are mutually exclusive. For example, given the

partial traversal $\tau = [+, \text{CONST}]$, LINEAR cannot be included anywhere in this traversal. Second, LINEAR cannot be a direct ancestor of a non-invertible token (namely, SIN, COS, or SQRT). For example, given the partial traversal $\tau = [\text{SQRT}, +, x_1]$, LINEAR cannot be chosen because SQRT is a direct ancestor. Third, LINEAR can only be selected at most once per traversal. These constraints are enabled whenever LINEAR is in the library.

- **Use each input variable constraint.** We include a constraint that each input variable, $x_1, \ldots, x_d$ for a d-dimensional sub-problem, must appear in the traversal at least once. This constraint is implemented by preventing an expression from ending if it would end without including all input variables (or the LINEAR token, which may arbitrarily include all input variables). For example, given the partial traversal $\tau = [\times, +, x_1, x_3]$ for a 3-dimensional problem, terminals other than x_2 or LINEAR are constrained because x_2 has not yet been chosen. This constraint is enabled for all sub-problems up to 10 dimensions.

C Additional details on integrating SINDy

C.1 Efficient sparse linear regression

From the perspective of the LINEAR token, the linear regression problem to solve can be described as:

$$\min_{\beta \in \mathbb{R}^k} \quad \|\Phi_X \beta - y'\|_2^2, \tag{4}$$

where $\Phi_X \in \mathbb{R}^{m \times k}$ is a Vandermonde matrix (data) computed by applying basis functions $\Phi(X)$ to input data X, $\beta \in \mathbb{R}^k$ is the vector of coefficients (unknown), and $y' \in \mathbb{R}^m$ (denoted $\overline{f}^{-1}(y)$ in Section 4) is the independent variable (data) computed by solving for the LINEAR token as described in Section 4 and Algorithm 2. In addition, we would like β to be sparse, with at most $l \ll k$ non-zero entries, to avoid over-fitting in cases where the functional form of the expression is not correct. Under this constraint, we have (4):

$$\begin{aligned} \min_{\beta \in \mathbb{R}^k} \quad & \|\Phi_X \beta - y'\|_2^2 \\ \text{s.t.} \quad & \|\beta\|_0 \leq k \end{aligned} \tag{5}$$

In our application, we need to solve (5) many thousands of times (each time LINEAR is selected), for identical matrices Φ_X (coming from independent variables in the given SR problem) and varying values of y' (coming from solving different candidate expression trees for the LINEAR token).

Problem (5) is non-convex. It can be re-formulated as an mixed-integer linear problem, but solving it with mixed-integer programming techniques would be prohibitively expensive. LASSO regression (Tibshirani, 1996), as used by SINDy (Brunton et al., 2016), is the standard approach to deal with sparse regression, but presents two issues for our application: setting the weights of the L_1 penalty is non-trivial, and it is still one or two orders of magnitude more expensive than regular least squares.

We propose a heuristic approach to solve (5) based on standard least squares, statistical indicators, and constrained quadratic programming. We start by computing the minimizer of the unconstrained problem in (4), $\beta^{\text{LS}} := \Phi_X^{\dagger} y'$, where $\Phi_X^{\dagger} = (\Phi_X^{\mathsf{T}} \Phi_X)^{-1} \Phi_X^{\mathsf{T}}$ is the Moore-Penrose pseudo-inverse of Φ_X, as well as the corresponding p-values of each coefficient β^{LS} for the null hypothesis $\beta_i^{\text{LS}} = 0$ for $i = 1, \ldots, k$ (t-test). We then find the $k - l$ coefficients with the largest p-values among β^{LS} and store their indices in $I^0 \in \mathbb{N}^{k-l}$. Finally, we solve the following constrained quadratic programming problem:

$$\begin{aligned} \min_{\beta \in \mathbb{R}^k} \quad & \|\Phi_X \beta - y'\|_2^2 \\ \text{s.t.} \quad & C\beta = 0 \;\; (z) \end{aligned} \tag{6}$$

where $C \in \{0, 1\}^{(k-l) \times k}$, with $C_{i, I_i^0} = 1$ for all $i = 1, \ldots, k - l$, and 0 otherwise, and $z \in \mathbb{R}^{k-l}$ are dual variables. The constraints in (6) ensure that all entries at I^0 positions zero in its minimizer $\beta^\star$, which can be found by solving the following linear system:

$$\begin{bmatrix} \Phi_X^{\mathsf{T}} \Phi_X & C^{\mathsf{T}} \\ C & 0 \end{bmatrix} \begin{bmatrix} \beta^\star \\ z^\star \end{bmatrix} = \begin{bmatrix} \Phi_X^{\mathsf{T}} y' \\ 0 \end{bmatrix} \tag{7}$$

The linear system in (7) has significant structure, which we exploit as follows. Observe that $\beta^\star = \beta^{\text{LS}} - (\Phi_X^{\mathsf{T}} \Phi_X)^{-1} C^{\mathsf{T}} z^\star$, thus $C(\Phi_X^{\mathsf{T}} \Phi_X)^{-1} C^{\mathsf{T}} z^\star = C\beta^{\text{LS}}$. Because Φ_X is a Vandermonde matrix,

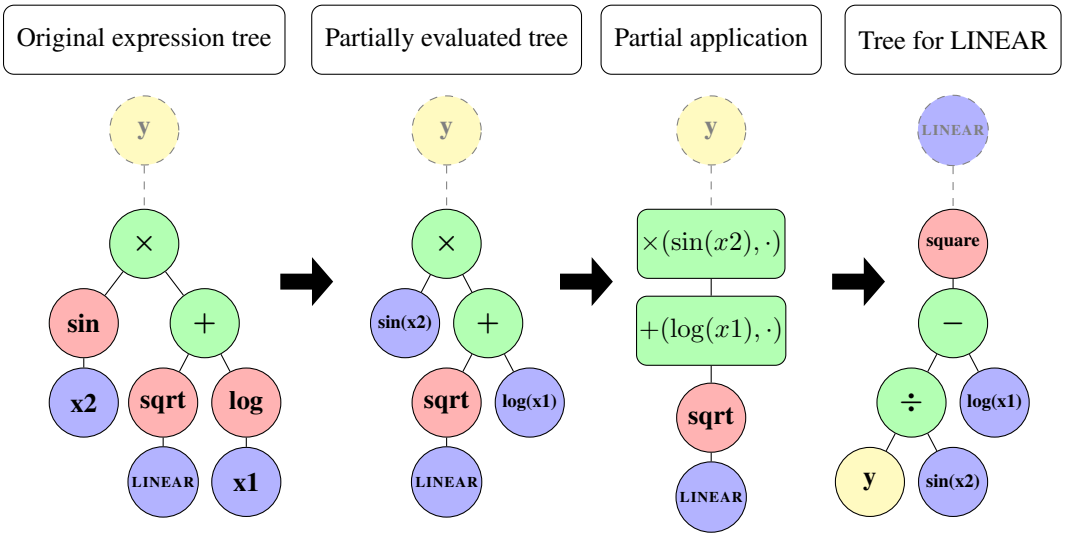

Figure 5: An example illustrating how to obtain the target data $\overline{f}^{-1}(y)$ for LINEAR. In this case, $F(x, \text{LINEAR}) = \sin(x_2)\left(\sqrt{\text{LINEAR}} + \log(x_1)\right)$ and $G(x, y) = \left(\frac{y}{\sin(x_2)} - \log(x_1)\right)^2$.

$\Phi_X^\mathsf{T}\Phi_X$ is symmetric positive definite, and so is $C(\Phi_X^\mathsf{T}\Phi_X)^{-1}C^\mathsf{T}$. Furthermore, since C is an elementary matrix we can form $C(\Phi_X^\mathsf{T}\Phi_X)^{-1}C^\mathsf{T}$ very efficiently whenever $(\Phi_X^\mathsf{T}\Phi_X)^{-1}$ is available. Thus, we solve the linear system in (7) by first solving $C(\Phi_X^\mathsf{T}\Phi_X)^{-1}C^\mathsf{T}z^\star = C\beta^{\text{LS}}$ using Cholesky decomposition, then computing $\beta^\star = \beta^{\text{LS}} - (\Phi_X^\mathsf{T}\Phi_X)^{-1}C^\mathsf{T}z^\star$.

To efficiently carry out the above procedure repeatedly with Φ_X constant for a given SR problem, we compute $\Phi_X^\dagger$ and $(\Phi_X^\mathsf{T}\Phi_X)^{-1}$ only once and store the results for all future solutions to the LINEAR token with the same data (i.e., for the duration of the given SR problem). Thus, after the first solution to LINEAR, subsequent solutions do not require recomputing $\Phi_X^\dagger$ or $(\Phi_X^\mathsf{T}\Phi_X)^{-1}$.

C.2 Example illustration of solving LINEAR token

In Figure 5, we show an example of how to obtain the target data programmatically solve for target data $\overline{f}^{-1}(y) = G(x, y)$ given $F(x, \text{LINEAR})$, which is then used to compute the sparse coefficients of the LINEAR token according to (6).

D Additional details on integrating NeSymReS

D.1 Pre-training architecture

Following Biggio et al. (2021), we use a set transformer (ST) (Lee et al., 2019) as the encoder for each input dataset (X, y). This choice of architecture is motivated by its permutation invariant property, i.e., the output of the model does not change under any permutation of the rows of (X, y), and by its ability to process input sets of variable size. We use the ST architecture introduced in Lee et al. (2019) with the following hyperparameter (we refer to the paper for their precise definitions): embedding dimensions for ST encoder and ST decoder is 20, number of attention heads in ST encoder and ST decoder is 2 and number of seed vectors in Pooling Multihead Attention Block is 2.

To interface with the remainder of the RNN-based controller, the encoding of (X, y) is used to produce the initial cell and hidden state of the RNN (illustrated in Figure 1), which in this way acts as a decoder. For our RNN architecture (consisting of 1 layer of 32 LSTM cells), the decoder outputs $h_0 \in \mathbb{R}^{64}$; the 32 first entries are used for the initial RNN cell state and the last 32 for the initial RNN hidden state. This way of merging RNN within encoder-decoder architectures is inspired by image-to-caption systems (Karpathy and Fei-Fei, 2015).

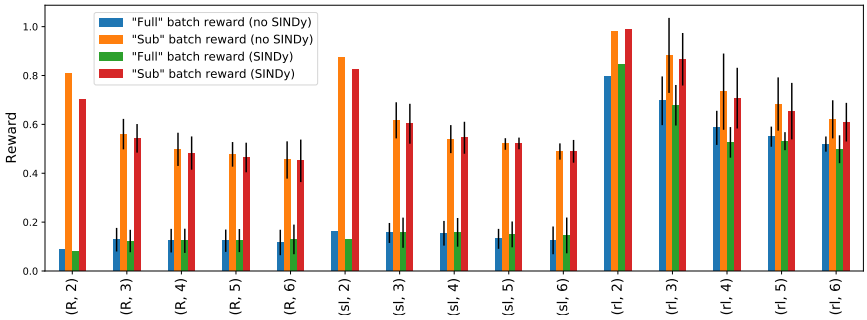

Figure 6: Average reward over 1000 expressions sampled from the pre-trained initial models (sl, rl) and a randomly initialized model (R) with and without the LINEAR token computed over the Feynman benchmarks. Results are stratified by problem dimension. The term "Full" refers to averages over the entire batch and "Sub" refers to averages over the top 5% of samples in the batch.

Following Biggio et al. (2021), to allow the same model to treat problems of different dimensionality (i.e., problems with different number of columns in X), we keep the number of input columns d constant, and implement a zero-padding strategy in which unused columns of X are zero-padded up to d columns. We use $d = 6$ for all experiments. During pre-training, randomly generated problem instances are thus limited to 6 dimensions.

At test time, when problems can yield arbitrary dimension, we disable the encoder architecture whenever dimensionality exceeds 6, using zero-valued initial state and randomly initialized controller weights (just as in ablations with NeSymReS "off"). For ablations including AIF, this "triage" is applied per sub-problem. For example, given a 10-dimensional problem, the root node will not leverage the pre-trained model, but sub-problems up to 6 dimensions will use the pre-trained model. This flexibility maximizes the use of the pre-trained models, and is made available by virtue of combining NeSymReS, AIF, and DSR to enable pre-trained models, enable sub-problems with sufficiently low dimensionality, and enable training without a pre-trained model, respectively. This is different than Biggio et al. (2021), which does not work for problems greater than $d = 3$ dimensions.

The whole encoder-decoder model is trained on millions of expressions and corresponding (X, y) datasets that are randomly generated *on-the-fly* during pre-training. Problem instance generation is described below. To enable all possible ablation experiments, we pre-trained four models: two trained using SL and two trained using RL, each with and without the LINEAR token. Each pre-trained model was trained for 8 days on a single GPU, seeing a total of $\sim$15M different problem instances each.

D.2 Characterization of pre-trained models

In this section, we characterize the pre-trained models $p(\tau|\theta^\star, \psi^\star, (X, y))$ where $\theta^\star, \psi^\star$ are pre-trained using either SL or RL objectives. As a reference, we also provide results obtained with $p(\tau|\theta^R, \psi^R, (X, y))$ where (θ^R, ψ^R) are randomly initialized.

In Figures 6 and 7, we show the average reward and entropy, respectively, over 1000 expressions sampled from each pre-trained model, computed across the 116 Feynman benchmark SR problems (which all have dimensionality at most 6) stratified by problem dimensionality. The entropy is computed column-wise over the batch of traversals, i.e., $\frac{1}{L}\sum_{i=1}^{L} \bar{H}[\{\tau_i^{(j)}\}_{j=1}^{1000}]$, where $\bar{H}$ is empirical entropy and L is the maximum traversal length in the batch.

Figure 8 shows traces of the top 5% of the 1000 samples from randomly initialized, SL pre-trained, or RL pre-trained models, over a series of one-dimensional problems. By visual inspection, both SL and RL pre-trained models produce traces that are clearly qualitatively closer to the real data than the randomly initialized model. Comparing traces between SL and RL pre-trained models, RL is clearly closer to the real data than SL. This is not surprising because the RL objective directly aims to reduce residuals (i.e., generate traces close to the real data). In contrast, the SL objective only learns a mapping, though in the SR search space, a small deviation in traversal space may result in a large reduction in fitness to the data.

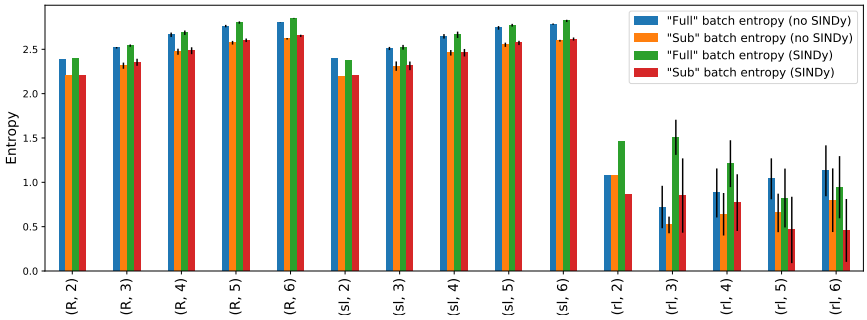

Figure 7: Average empirical entropy over 1000 expressions sampled from the pre-trained initial models (sl, rl) and a randomly initialized model (R) with and without the LINEAR token computed over the Feynman benchmarks. Results are stratified by problem dimension. The term "Full" refers to averages over the entire batch and "Sub" refers to averages over the top 5% of samples in the batch.

D.3 Problem instance generation

Here we describe additional details for the SR problem instance generator, used to generate on-the-fly expressions τ and corresponding datasets (X, y) during SL and RL pre-training. Pseudocode for this process is given in Algorithm 3

To sample pre-training problem instancess, we leverage the fact that the distribution induced by DSR includes a prior (Petersen et al., 2021b). Specifically, DSR samples each token τ_i according to $\tau_i \sim \text{Categorical}(\text{Softmax}(\ell_\theta + \ell_\circ))$, where ℓ_θ are logits emitted by the RNN under current parameters θ, and $\ell_\circ$ are the logits defined by the prior (see Appendix B and Landajuela et al. (2021b) for details). To generate training problems, we ignore emissions ℓ_θ and only sample from the prior, $\ell_\circ$. Since softmax is defined only up to a constant, this process is equivalent to sampling from a prior-informed uniform distribution, i.e. $\ell_\theta \equiv$ constant.

After sampling an expression τ, we generate (X, y) data from it similarly to Biggio et al. (2021). First, if there is a LINEAR token in τ, we sample linear coefficient values using a spike-and-slab distribution. Similarly, if there are any CONST tokens in τ, we sample their values according to $\mathcal{U}(-5, 5)$. Next, we generate 500 data points using the expression induced by τ. To do so, for each dimension of X, we sample its domain extrema a, b according to $\mathcal{U}(-10, 10)$, then sample each X coordinate according to $\mathcal{U}(\min(a, b), \max(a, b))$. Finally, we instantiate τ as an expression f and compute y values as $y = f(X)$. To ensure numerical stability of τ and the corresponding (X, y) data, we re-sample coordinates if $|y| > 100$ or is undefined until obtaining 500 points. If 500 points cannot be obtained within 5000 attempts, (e.g. given $a, b < 0$ and $f(X) = \log(X)$), τ is discarded and re-sampled.

Algorithm 3 Symbolic regression pre-training problem instance generator

input Expression generator $p_\mathcal{G}(\tau)$
output Symbolic regression problem instance (X, y)
$\tau \sim p_\mathcal{G}(\cdot)$ ▷ Sample expression
$m \leftarrow \dim(\tau)$ ▷ Get dimensionality of τ
if LINEAR $\in \tau$ **then**
  $c_i \sim 0.5\mathcal{U}(-10, 10) + 0.5\delta(0)$ ▷ Sample linear coefficient values
for CONST $\in \tau$ **do**
  $c \sim \mathcal{U}(-5, 5)$ ▷ Sample constant value
for $i \in 1, \ldots, m$ **do**
  $a, b \sim \mathcal{U}(-10, 10)$ ▷ Sample domain extrema
  for $j \in 1, \ldots, 500$ **do**
    $X_{i,j} \sim \mathcal{U}(\min(a, b), \max(a, b))$ ▷ Sample X coordinate
$y = f(X)$ ▷ Compute y values
return (X, y)

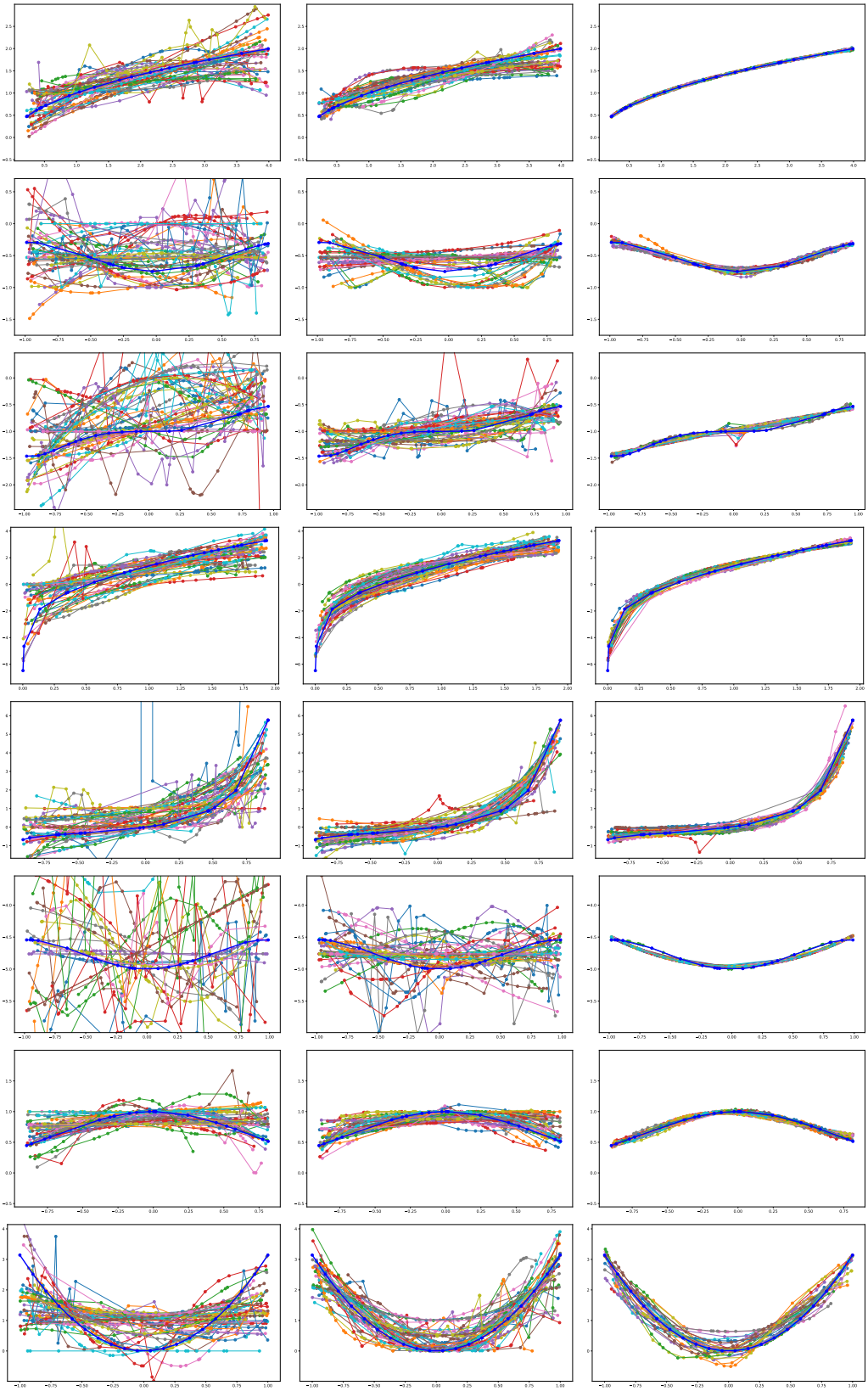

Figure 8: Traces of the top 5% of 1000 expression sampled from the pre-trained models for a series of one-dimensional problems. Columns correspond to controllers with random weights, SL pre-trained weights, and RL pre-trained weights, respectively. Blue trace corresponds to the real (X, y) data.

D.4 RL pre-training using risk-seeking policy gradients

Direct optimization of (3) would maximize *expected* reward of sampled expressions for a given $s = (X, y)$ (to simplify notation, we use s to refer to a dataset (X, y) in this section). However, in SR we are generally interested in best case performance, i.e.,

$$\text{argmax}_\tau R(\tau, s),\ \forall s. \tag{8}$$

As as surrogate (relaxed) version of (8), we consider the following *dual* formulation of the problem:

$$\text{argmax}_{(\theta,\psi)} J_{\text{riskRL}}(\theta, \psi) := \mathbb{E}_{s_g \sim p_\mathcal{G}} \left[\mathbb{E}_{\tau \sim p(\tau|\theta,\psi,s_g)} \left[R(\tau, s_g) | R(\tau, s_g) \geq R_\varepsilon(\theta, \psi, s_g) \right] \right],$$

where $R_\varepsilon(\theta, \psi, s_g)$ is the $(1-\varepsilon)$-quantile of the random variable $R(\tau, s_g),\ \tau \sim p(\tau|\theta, \psi, s_g)$. While (8) is not suitable for gradient-based optimization (as $R(\tau, s)$ is not continuous), we can compute the gradient of $J_{\text{riskRL}}(\theta, \psi)$ using the risk-seeking policy gradient introduced in Petersen et al. (2021a), giving:

$$\begin{aligned}
\nabla_{\theta,\psi} J_{\text{riskRL}}(\theta, \psi) &= \mathbb{E}_{s_g \sim p_\mathcal{G}} \left[\nabla_{\theta,\psi} \mathbb{E}_{\tau \sim p(\tau|\theta,\psi,s_g)} \left[R(\tau, s_g) | R(\tau, s_g) \geq R_\varepsilon(\theta, \psi, s_g) \right] \right] \\
&= \mathbb{E}_{s_g \sim p_\mathcal{G}} \left[\mathbb{E}_{\tau \sim p(\tau|\theta,\psi,s_g)} \left[(R(\tau, s_g) - R_\varepsilon(\theta, \psi, s_g)) \right.\right. \\
&\qquad \left.\left. \nabla_{\theta,\psi} \log p(\tau|\theta, \psi, s_g) | R(\tau, s_g) \geq R_\varepsilon(\theta, \psi, s_g) \right] \right].
\end{aligned}$$

A Monte Carlo estimate of the above is given by:

$$\nabla_{\theta,\psi} J_{\text{riskRL}}(\theta, \psi) \approx \frac{1}{B \cdot \varepsilon N} \sum_{i=1}^{B} \sum_{j=1}^{N_B} (R(\tau_i^{(j)}, s_i) - \tilde{R}_\varepsilon(\theta, \psi, s_i)) \nabla_{\theta,\psi} \log p(\tau_i^{(j)}|\theta, \psi, s_i) \mathbf{1}_{R(\tau_i^{(j)}, s_i) \geq \tilde{R}_\varepsilon(\theta,\psi,s_i)}, \tag{9}$$

with $\mathbf{1}_C$ the characteristic function for set C, $s_1, \ldots, s_B \sim p_\mathcal{G}$,

$$\begin{cases} \tau_1^{(1)}, \ldots \tau_1^{(N_B)} \sim p(\tau|\theta, \psi, s_1) \\ \qquad \vdots \\ \tau_B^{(1)}, \ldots \tau_B^{(N_B)} \sim p(\tau|\theta, \psi, s_B) \end{cases},$$

and

$$\begin{cases} \tilde{R}_\varepsilon(\theta, \psi, s_1) = \mathrm{Q}_{1-\varepsilon}(R(\tau_1^{(1)}, s_1), \ldots, R(\tau_1^{(N_B)}, s_1)) \\ \qquad \vdots \\ \tilde{R}_\varepsilon(\theta, \psi, s_B) = \mathrm{Q}_{1-\varepsilon}(R(\tau_B^{(1)}, s_B), \ldots, R(\tau_B^{(N_B)}, s_B)) \end{cases},$$

where $\mathrm{Q}_\alpha(x)$ is the empirical α quantile of vector x. This motivates a procedure to compute an approximation of the gradient $\nabla_{\theta,\psi} J_{\text{riskRL}}(\theta, \psi)$: first, sample B problem instances $s_i = (X, y)_i$ from $p_\mathcal{G}$, then, sample N_B expressions $\tau_i^{(j)}$ per problem instance s_i from $p(\tau|\theta, \psi, s_i)$, then compute (9) using their rewards $R(\tau_i^{(j)}, s_i)$ and the empirical $(1-\varepsilon)$-quantile.

E Short descriptions of the 14 original SRBench baselines

Below we provide short descriptions of the 14 original SR baseline methods used by SRBench in Figures 2 and 3. We refer readers to the original papers for additional details.

- **Age-fitness Pareto optimization (AFP):** Schmidt and Lipson (2011) propose a GP approach that avoids premature convergence by adding a multi-dimensional optimization objective that evaluates solutions based on fitness and age.
- **AFP with co-evolved fitness estimates (AFP-FE):** Schmidt and Lipson (2009) extend standard AFP with a new method for fitness estimation (Schmidt and Lipson, 2008).
- **AIF:** The AIF algorithm (Udrescu et al., 2020) used in this work, using BF/PF as its underlying search algorithm.

- **Bayesian symbolic regression (BSR):** Jin et al. (2019) propose a method that incorporates prior knowledge and samples expression trees from a posterior distribution using an efficient Markov Chain Monte Carlo algorithm that is robust to hyperparameter settings and finds more concise solutions than purely GP-based approaches.
- **Deep symbolic regression (DSR):** The DSR algorithm (Petersen et al., 2021a) used in this work. Notably, this uses the predecessor to the version of DSR used in this work, and does not include improvements introduced in Landajuela et al. (2021b) or Mundhenk et al. (2021).
- **ε-lexicase selection (EPLEX):** La Cava et al. (2019) propose a method that improves the parent selection process in GP by considering performance on data samples individually instead of in aggregate or average, rewarding expressions that perform well on harder parts of the problem.
- **Feature engineering automation tool (FEAT):** La Cava et al. (2018) propose a method that aims at finding simple solutions that generalize well by maintaining an archive of solutions with accuracy-complexity trade-offs to improve generalization and interpretation.
- **Fast function extraction (FFX):** McConaghy (2011) propose a non-evolutionary technique for SR based on pathwise learning (Friedman et al., 2010) that generates a set of solutions that trade off error versus complexity, while being orders of magnitude faster than GP and providing deterministic convergence.
- **GP version of the gene-pool optimal mixing evolutionary algorithm (GP-GOMEA):** Virgolin et al. (2021) propose to combine GP with linkage learning, an approach that learns a model of interdependencies to estimate what patterns to propagate and proposes small and interpretable solutions.
- **GP (gplearn):** Basic, Koza-style GP (Koza, 1994) for SR using the open software library gplearn (https://github.com/trevorstephens/gplearn). This implementation is very similar to the GP component used in uDSR.
- **Interaction-transformation evolutionary algorithm (ITEA):** de França and Aldeia (2021) introduce a mutation-based evolutionary algorithm based on six mutation heuristics that allows both learning high-performing solutions and extracting the importance of each original feature of a data set as an analytical function.
- **Multiple regression genetic programming (MRGP):** Arnaldo et al. (2014) propose a method that decouples and linearly combines sub-expressions via multiple regression on the target variable as a cost-neutral extension to basic GP.
- **SR with Non-linear least squares (Operon):** Kommenda et al. (2020) incorporate non-linear least squares optimization into GP as a local search mechanism for offspring selection to improve generalization.
- **Semantic backpropagation genetic programming (SBP-GP):** Virgolin et al. (2019) modify the random desired operator algorithm (Pawlak et al., 2014), a GP approach using semantic backpropagation, by incorporating the principles of linear scaling, making the algorithm much more effective despite being computationally more expensive.

F Hyperparameter selection

Hyperparameters for uDSR are listed in Table 3. In general, we used default hyperparameters from of each constituent method, with a few exceptions we justify below. For DSR, we used default hyperparameters given by Mundhenk et al. (2021), with the exception of increasing maximum expression length from 30 to 64 to accommodate more challenging problems that those attempted in Mundhenk et al. (2021). Note that, following Mundhenk et al. (2021), these hyperparameters use priority queue training (Abolafia et al., 2018) during fine-tuning as an alternative to the risk-seeking policy gradient. For GP, we used all default hyperparameters given by Mundhenk et al. (2021). For AIF, we used all default hyperparameters given by Udrescu et al. (2020). For NeSymReS, we used all default hyperparameters for the set encoder given by Lee et al. (2019). For pre-training, we increased the maximum input dimension of the set transformer from 3 (in Biggio et al. (2021)) to 6 to accommodate higher dimensional problems. For RL pre-training, since this has never been performed for SR to date, we estimated the learning rate and batch size, and tuned the entropy weight

and risk factor via a small grid search of entropy weight $\in \{0, 0.3\}$ and risk factor in $\in \{0.05, 0.2\}$. For SINDy, since we used a novel formulation of sparse linear regression instead of LASSO, we set the maximum number of terms to 10. For simplicity, we considered basis functions comprising monomials up to degree 3, e.g. $x_1^2 x_3$, x_2^3, or $x_1 x_2$.

Table 3: Hyperparameters for uDSR. Most values were selected from the default values of their appropriate works. Values marked "estimated" were hand-selected and not tuned. Values marked "tuned" were tuned during pre-training using grid search.

Hyperparameter	**Symbol**	**Value**	**Source**
DSR			
Batch size	N	500	Mundhenk et al. (2021)
Risk factor	ε	0.02	Mundhenk et al. (2021)
Priority queue size	–	10	Mundhenk et al. (2021)
Learning rate	–	0.0025	Mundhenk et al. (2021)
Entropy weight	–	0.3	Landajuela et al. (2021b)
Entropy weight decay factor	–	0.7	Landajuela et al. (2021b)
Reward function	R	Inverse NRMSE	Landajuela et al. (2021b)
LSTM layer sizes	–	[32]	Landajuela et al. (2021b)
Minimum expression length	–	4	Landajuela et al. (2021b)
Maximum expression length	–	64	Estimated
Maximum constants	–	3	Landajuela et al. (2021b)
Soft length prior mean	–	10	Landajuela et al. (2021b)
Soft length prior standard deviation	–	5	Landajuela et al. (2021b)
GP			
Generations	S	25	Mundhenk et al. (2021)
Crossover probability	–	0.5	Mundhenk et al. (2021)
Mutation probability	–	0.5	Mundhenk et al. (2021)
Tournament size	–	5	Mundhenk et al. (2021)
Samples used to train controller	k	50	Mundhenk et al. (2021)
AIF			
Learning rate	–	0.01	Udrescu et al. (2020)
Training epochs	–	500	Udrescu et al. (2020)
Hidden layer sizes	–	[128, 128, 64, 64]	Udrescu et al. (2020)
Brute force try time	–	600	Udrescu et al. (2020)
Polynomial fit degree	–	3	Udrescu et al. (2020)
NeSymReS			
Maximum input dimension	d	6	Estimated
ST embedding size	–	20	Lee et al. (2019)
ST attention heads	–	2	Lee et al. (2019)
ST seed vectors	–	2	Lee et al. (2019)
Learning rate (SL)	–	0.0005	Estimated
Problem instances per batch (SL)	B	40	Estimated
Learning rate (RL)	–	0.0005	Estimated
Problem instances per batch (RL)	B	50	Estimated
Samples per problem instance (RL only)	N_B	10	Estimated
Entropy weight (RL only)	–	0	Tuned $\{0, 0.3\}$
Risk factor (RL only)	ε	0.2	Tuned $\{0.05, 0.2\}$
SINDy			
Basis functions	Φ	Monomials	Estimated
Maximum monomial degree	D	3	Estimated
Maximum non-zero terms	–	10	Estimated

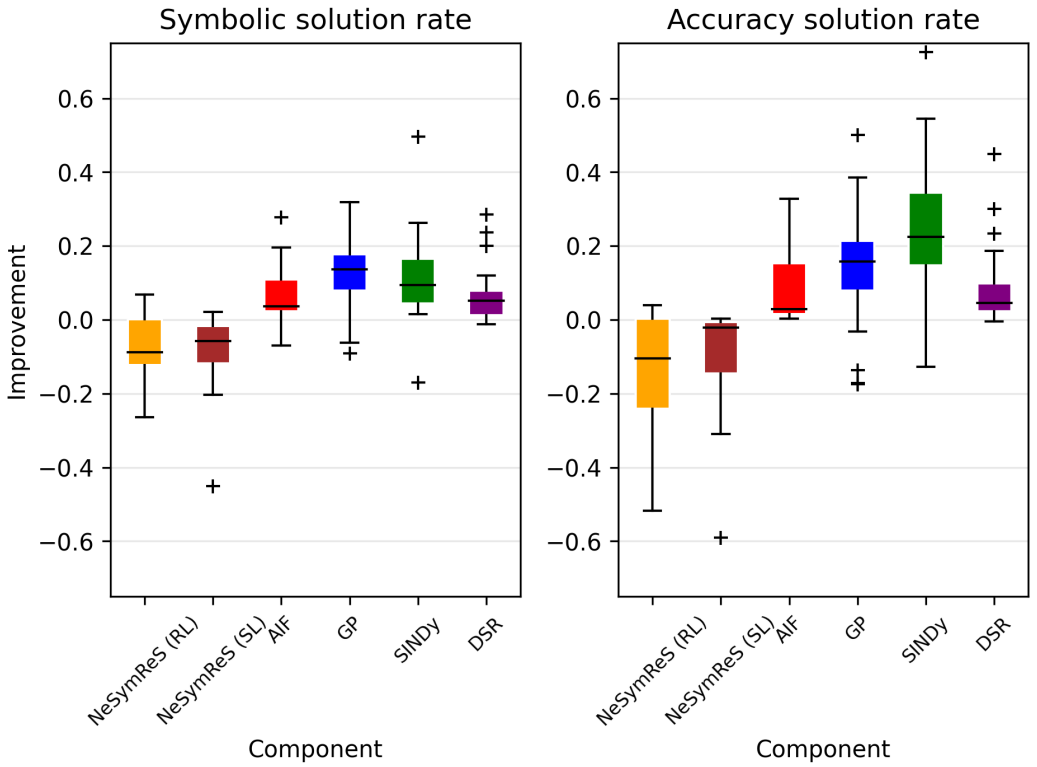

Figure 9: Distribution of improvement to symbolic solution rate (left) and accuracy solution rate (right) when each component is enabled, over all combinations of other components.

G Marginal gains

The modularity of uDSR allows one to activate each individual component independently of other components. We measured the improvement $r(c_{i,\text{on}}, c_{-i}) - r(c_{i,\text{off}}, c_{-i})$ for $r =$ symbolic solution rate and $r =$ accuracy solution rate, for each component c, for all possible combinations of settings for other components c_{-i} (NeSymReS has two "on" states, denoted "RL" and "SL"). Figure 9 shows that individually enabling AIF, GP, SINDy (i.e., LINEAR token), and DSR leads to positive improvement in symbolic solution rate and accuracy solution rate, averaged over all possible configurations of other components. Notably, SINDy almost always improves the symbolic solution rate, AIF always improves the accuracy solution rate, and DSR provides the highest worst case improvement. Pre-training with NeSymReS is negative on average, but it provides positive improvement in certain cases; specifically, the highest-performing combination uses NeSymReS (SL) (see Figure 4).

H Runtimes

We evaluate the runtime of uDSR and its constituent components in two ways: (1) runtime per expression evaluation and (2) runtime per SR problem.

For runtime per expression evaluation, we consider the components GP, SINDy, and DSR. NeSymReS does not significantly affect runtime per expression because the embedding at test time is computed only once, and the subsequent architecture and algorithm is otherwise the same. While AIF affects overall runtime by generating sub-problems, it does not affect runtime per expression. We show average runtime per expression evaluation in Table 4. The effect of GP ranges from no effect to a 2-fold increase in runtime. SINDy actually decreases runtime in many cases. This can occur because LINEAR "competes" with the CONST token, as both are terminal tokens and the tokens are mutually exclusive (see Appendix B). Since (non-linear) CONST optimization is typically much slower than LINEAR optimization, this results into an overall reduction in runtime. Lastly, disabling DSR (i.e.,

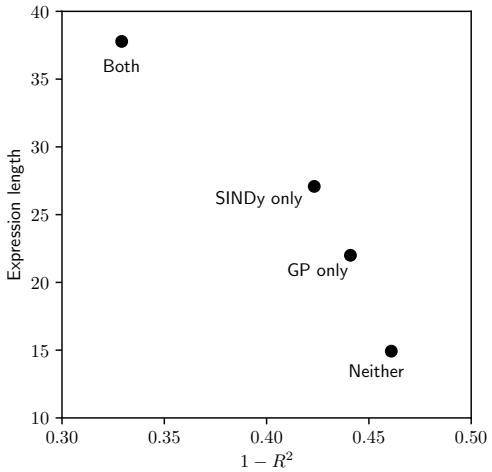

Figure 10: Accuracy vs complexity trade-offs for uDSR ablations with and without GP, and with and without the LINEAR token (SINDy). Each point is the median of 10 independent runs, averaged over the 122 SRBench black-box SR problems.

not training the controller) has little to no effect on runtime, demonstrating that backpropagation is not a computational bottleneck for uDSR at test time.

Average runtime per SR problem is given in Table 5 for all uDSR ablations. Because uDSR uses early stopping, these values are largely dominated by how early and how often each ablation achieves the early stopping criterion (defined as in Petersen et al. (2021a) as achieving a normalized MSE less than 10^{-16}), as opposed to reaching all 500,000 expression evaluations per problem-instance or the 24 hr walltime limit. The fold-increase in computational cost incurred by AIF is also evident in these results.

GP	SINDy	DSR	Runtime per expression (s)
0	0	0	0.0605 ± 0.0014
		1	0.0606 ± 0.0015
	1	0	0.0605 ± 0.0021
		1	0.0905 ± 0.0021
1	0	0	0.1105 ± 0.0878
		1	0.1205 ± 0.0774
	1	0	0.0705 ± 0.0364
		1	0.0905 ± 0.0364

Table 4: Runtime per expression on a single CPU for uDSR and several ablations on the 14 Strogatz benchmark SR problems. 0: Component disabled. 1: Component enabled.

I Accuracy vs complexity for GP and SINDy components

In our ablations on ground-truth SR problems, both GP and the LINEAR token (SINDy) yield considerable gains in symbolic solution rate and accuracy solution rate. However, GP can lead to longer expressions due to crossover operators (referred to as "bloat"), while LINEAR can result in longer expressions by adding many linear terms. To assess the trade-offs between accuracy and complexity, we performed a small ablation study comprising four configurations: GP $\in$ [on, off] $\times$ SINDy $\in$ [on, off], with all other components enabled. In Figure 10, we see that the four configurations form a Pareto front in terms of accuracy (R^2 on held-out test data) vs complexity (expression length, computed as number of nodes in the SymPy-parsed expression tree). This reinforces how the modularity of uDSR allows it to be configured based on use case.

GP	SINDy	AIF	DSR	NeSymReS	Runtime (min)		
0	0	0	0	0	47.5	±	26.1
				RL	527.9	±	496.5
				SL	55.5	±	25.1
			1	0	147.2	±	122.4
				RL	323.1	±	240.3
				SL	68.7	±	40.1
		1	0	0	193.3	±	183.3
				RL	1061.4	±	550.5
				SL	227.2	±	198.7
			1	0	435.8	±	465.3
				RL	672.1	±	569.0
				SL	289.1	±	288.2
	1	0	0	0	39.4	±	35.8
				RL	251.7	±	205.4
				SL	46.6	±	35.0
			1	0	39.1	±	41.7
				RL	137.9	±	108.7
				SL	45.3	±	38.0
		1	0	0	170.5	±	209.9
				RL	549.2	±	462.8
				SL	206.5	±	227.7
			1	0	167.9	±	223.7
				RL	391.4	±	450.5
				SL	193.8	±	237.0
1	0	0	0	0	102.0	±	108.2
				RL	215.3	±	183.8
				SL	30.9	±	53.7
			1	0	129.4	±	135.4
				RL	214.8	±	173.5
				SL	59.7	±	89.1
		1	0	0	336.6	±	474.5
				RL	478.1	±	437.5
				SL	117.8	±	142.3
			1	0	383.6	±	520.8
				RL	484.0	±	435.0
				SL	185.9	±	238.8
	1	0	0	0	14.4	±	18.8
				RL	42.0	±	43.7
				SL	13.8	±	17.2
			1	0	16.4	±	21.5
				RL	34.5	±	31.7
				SL	18.1	±	24.1
		1	0	0	90.7	±	162.6
				RL	132.6	±	176.1
				SL	72.6	±	121.7
			1	0	106.5	±	189.4
				RL	128.5	±	180.7
				SL	89.9	±	155.2

Table 5: Runtime per SR problem (with early stopping) on a single CPU for uDSR ablations on the 130 ground-truth SR problems. 0: Component disabled. 1: Component enabled.

J Software, data, and computational resources

Our implementation of uDSR combines several open-source software projects. For DSR we use the Deep Symbolic Optimization software framework[2], which includes the base implementation of DSR and facilities to integrate GP using the Distributed Evolutionary Algorithms in Python software[3] (Fortin et al., 2012). We integrate AIF using its open-source implementation[4]. We implement the

[2] `https://github.com/brendenpetersen/deep-symbolic-optimization`. BDS-3-Clause license.

[3] `https://github.com/DEAP/deap`. LGPL-3.0 license.

[4] `https://github.com/SJ001/AI-Feynman`. MIT license.

LINEAR token using standard numerical libraries. The set transformer used for pre-training is based on an open-source TensorFlow implementation[5]. Our empirical evaluation uses SRBench[6]. The SRBench benchmark datasets are taken from the open-source PMLB database[7] (Olson et al., 2017).

Pre-training experiments were performed on a single NVIDIA TitanX GPU. All other experiments were performed on a single TOSS 3 2.4 GHz CPU. A total of 73,700 independent trials of uDSR were run across all experiments. Approximately 1% of runs were stopped early at the 24 hr walltime limit.

K Originally published methods versus corresponding uDSR components

K.1 Methodological differences

uDSR integrates the *key capabilities* (listed in Table 1) of each of the five integrated algorithms; the algorithms are not simply reused in their original, published form (with the exception of DSR, as discussed below). While Section 4 details the methodological differences between the originally published algorithms and their corresponding uDSR component, for convenience, below we summarize these key differences for each algorithm.

AIF. The key difference between the published, standalone AIF algorithm (Udrescu et al., 2020) and our use of recursive simplification is that the original AIF obtains expressions using BF/PF as its underlying SR algorithm, whereas our module obtains expressions sampled from a parameterized distribution over expressions. Another detail is that the original AIF used a slightly different set of tokens that were tailored to the Feynman benchmark problems, whereas our component used a more generic set of tokens.

NeSymReS. The key difference between the published, standalone NeSymReS and our pre-training component lies in how expressions are sampled at test time. Standalone NeSymReS uses *beam search* at test time to identify high-likelihood expressions under the model conditioned on the test problem. In contrast, our NeSymReS module *fine-tunes* the controller (conditioned on the test problem) by continuing to train non-encoder parameters at test time. Differences during pre-training are that standalone NeSymReS generates problems of up to 3 input variables, whereas our module generates problems of up to 6, and that NeSymReS trains using SL, whereas our module can alternative train using RL using a modified version of the risk-seeking policy gradient.

GP. There are two key differences between the published, standalone GP algorithm (Koza, 1994) and our GP component. (1) In standalone GP, starting populations are sampled using the traditional "half and half" method, in which half the starting population is sampled using the "full" protocol and the other half is sampled using the "grow" protocol, as defined in Luke and Panait (2001). In contrast, uDSR's GP starting populations are sampled from the controller. (2) In standalone GP, there is only a single starting population per run, and all generations are performed until the algorithm completes. In our module, a relatively small number of generations are performed before a new starting population is sampled from the controller. Thus, GP module operates in a random restart-like fashion, restarting with new starting populations as the controller learns.

SINDy. The key difference between the published, standalone SINDy algorithm (Brunton et al., 2016) and our LINEAR token is that the original SINDy uses a fixed functional form (given basis functions Φ), whereas LINEAR occurs within a functional form "wrapper" sampled from the distribution over expressions. Essentially, the original SINDy algorithm computes a single functional form with traversal $\tau = [\text{LINEAR}]$, whereas even the SINDy-only uDSR ablation includes LINEAR within nonlinear functional forms. Two minor differences are that: (1) The original SINDy used LASSO to learn sparse coefficients, whereas our module uses a modified version of least squares, and (2) The original SINDy includes both linear and nonlinear basis functions in Φ, e.g. $\sin(x_1)$, whereas we used a simpler set of monomials up to degree 3.

DSR. Unlike the other four components, our DSR module does not exhibit any algorithmic changes relative to the published DSR in Landajuela et al. (2021b), as DSR was used as a starting point into which we integrated the other four methods. However, note that in Figures 2 and 3, the SRBench

[5] `https://github.com/arrigonialberto86/set_transformer`. Unlicensed.
[6] `https://github.com/cavalab/srbench`. GPL-3.0 license.
[7] `https://github.com/EpistasisLab/pmlb`. MIT license.

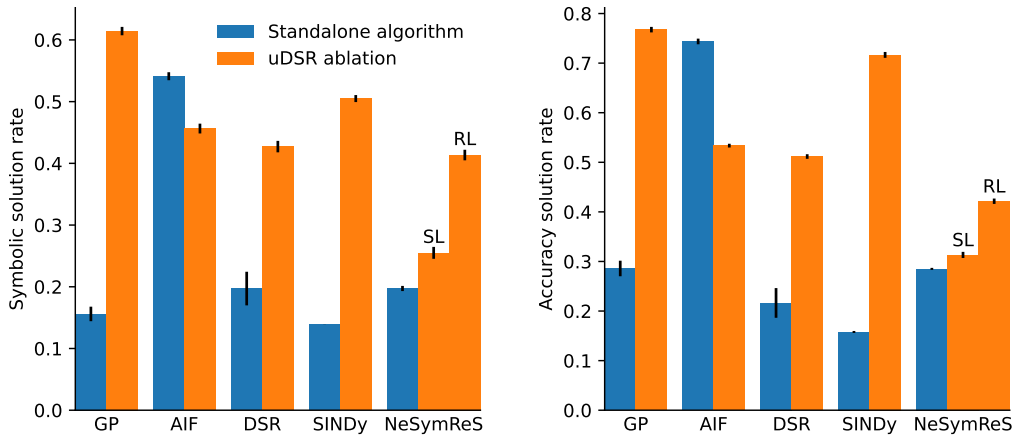

Figure 11: Comparisons between the five original standalone algorithms and their corresponding single-component uDSR ablation, averaged across 130 ground-truth noiseless SR problems. Left: symbolic solution rate. Right: accuracy solution rate. Error bars represent standard error across 10 random seeds per problem.

baseline used the original DSR version in Petersen et al. (2021a) without the improvements introduced in Landajuela et al. (2021b).

K.2 Performance differences

The methodological differences described above may result in differences in performance. In Figure 11, we compare each original, standalone method to the corresponding uDSR ablation, in which all other components are disabled. Note that Figure 11 uses the same raw data as in Figures 2 and 4, but is reformatted here to enable facile comparison. Also note that, besides the methodological differences listed above, there are additional differences in the experimental setting used between the original papers and this work. For example, AIF uses a different token set than the AIF-only uDSR ablation. For simplicity, for the original methods, we use the experimental workflow used in the original papers.

AIF is the only standalone algorithm that outperforms the corresponding AIF-only uDSR ablation. This may be due to AIF using a different token set tailored to solving the Feynman benchmark problems, and/or because BF/PF outperforms sampling from a randomly-initialized controller. The GP-only uDSR ablation greatly outperforms standalone GP. This is consistent with the finding by Mundhenk et al. (2021) that standard GP excels in a random restart-like fashion. The two NeSymReS-only uDSR ablations (particularly with RL pre-training) outperform standalone NeSymReS, which may be attributed to the fact that standalone NeSymReS was only trained on SR problems of up to 3 input variables, whereas most SRBench problems have > 3 input variables. (For problems with > 3 input variables, it cannot select input variables beyond x_3.) The DSR-only uDSR ablation outperforms the standalone DSR used by SRBench (Petersen et al., 2021a); however, when using the updated version of DSR given by Landajuela et al. (2021b), performance is identical to the DSR-only uDSR ablation, by construction. The SINDy-only uDSR ablation greatly outperforms standalone SINDy. This is because standalone SINDy only optimizes a single linear solution, whereas the SINDy-only uDSR ablation is still able to embed the LINEAR token into highly non-linear functional forms (in particular, rational expressions).

L Illustrative example of solution formation

By tracing the formation of an expression solution, it is possible to identify which tokens in the solution τ (i.e. which parts of the expression) can be attributed to which individual uDSR components. In Figure 12, we provide an illustrative example of this process, which highlights how the five uDSR components work together to find a final solution. In this illustrative example: (1) AIF first identifies

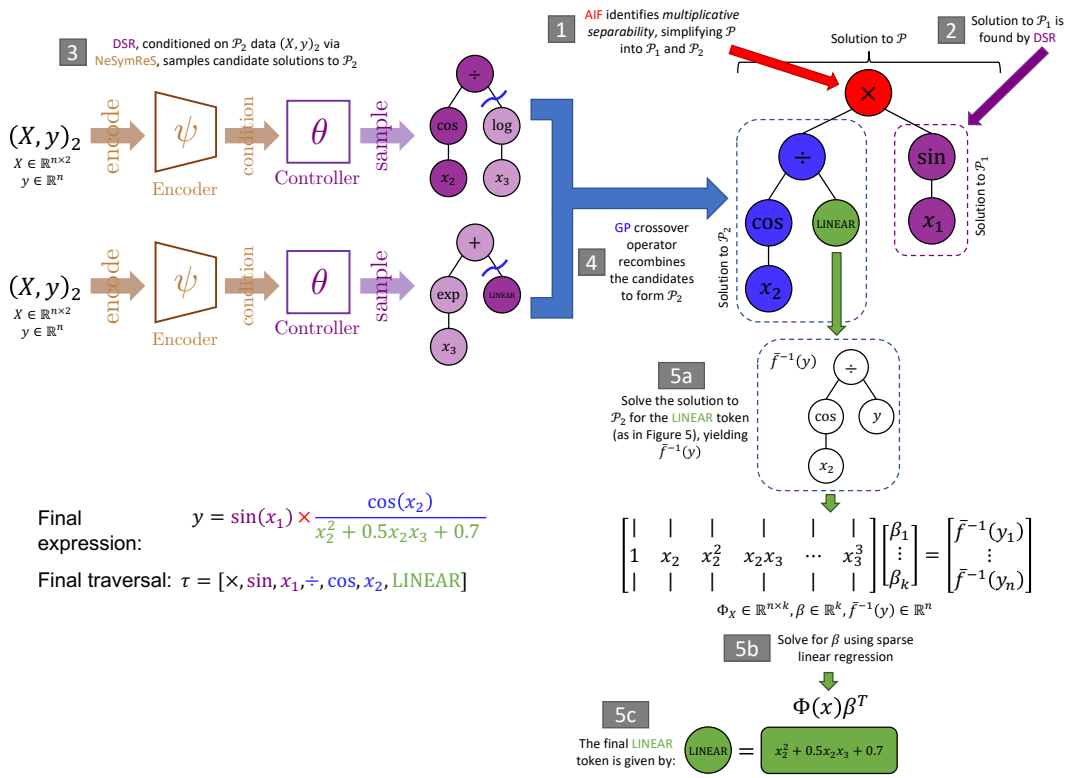

Figure 12: Illustrative example of how the five interacting uDSR components work together to find a solution. In the final solution, some parts of the expression may be traced back to particular components. Color key: AIF, NeSymReS, DSR, GP, SINDy.

multiplicative separability, which simplifies the original problem $\mathcal{P}$ into two sub-problems, $\mathcal{P}_1$ and $\mathcal{P}_2$. The solution to $\mathcal{P}$ is then the product of the solutions to $\mathcal{P}_1$ and $\mathcal{P}_2$; thus, AIF essentially learned that the root node of the expression tree is $\times$. (2) The simple solution to $\mathcal{P}_1$ is found quickly by DSR. (3) DSR, conditioning the controller on $\mathcal{P}_1$ data $(X, y)_2$ using the NeSymReS encoder, samples candidate solutions to $\mathcal{P}_2$. (4) The GP crossover operator recombines two candidate solutions into the solution to $\mathcal{P}_2$. (5) Lastly, since the solution to $\mathcal{P}_2$ includes a LINEAR token, the linear combination of basis functions is optimized according to the (X, y) data of $\mathcal{P}_2$.

M Additional details

M.1 Behavior of LINEAR for non-finite target data.

When solving for target data $\overline{f}^{-1}(y)$ given $F(x, \text{LINEAR})$ (by performing successive application of the root node's inverse operations), it may be the case that infinite or undefined values arise. For example, given the traversal $\tau = [\text{EXP}, \text{LINEAR}]$ and negative values in y, the function for target data $\overline{f}^{-1}(y) = \log(y)$ produces undefined values. In these cases, the LINEAR token returns a default value of 1, i.e. coefficients β are all zero except for the constant This process is similar to the CONST token defaulting to a value of 1 when L-BFGS-B encounters undefined values (Petersen et al., 2021a). Note that if the candidate functional form is correct (e.g. the traversal $\tau =$ [exp, LINEAR] when the (X, y) data is generated from ground-truth expression $e^{x_1^2+0.5x_2}$), then, by construction, it will never be the case that $\bar{f}^{-1}(y)$ yields undefined values.

M.2 Additional details for SRBench

SRBench includes metrics for symbolic-based performance, accuracy-based performance, and solution complexity. "Symbolic solution rate" is computed by comparing the ground-truth SymPy

expression to the SymPy-parsed expression returned by the algorithm. Floating-point values are first rounded to 3 decimals. Expressions are then deemed symbolically equivalent if their symbolic difference is "0" or their symbolic quotient is "1." "Accuracy solution rate" is based on the coefficient of determination (R^2) on a held-out test dataset exceeding 0.999. R^2 is a useful metric because it is normalized across datasets, allowing averaging performance metrics across multiple problems. "Complexity" is defined by the number of nodes in the SymPy-parsed expression tree.

Most of the baselines in Figures 2 and 3 report results directly curated by SRBench. (SRBench enables reproducibility, including across random number seeds, making such direct comparisons appropriate.) The original SRBench study (La Cava et al., 2021) already included baselines for standalone GP, DSR, and AIF. Since standalone SINDy and NeSymReS were not included in the original SRBench study, we executed these ourselves using the SRBench pipeline. For NeSymReS (Biggio et al., 2021), since it was trained on only up to 3 input variables, it ignores additional input variables if present.

M.3 Extrapolation and recommendations for NeSymReS

A key aspect regarding recommended use of NeSymReS is knowledge of how well the distribution over (X, y) datasets sampled by problem instance generator $p_{\mathcal{G}}$ is thought to be representative of the user's problems of interest. In particular, a key characteristic of $p_{\mathcal{G}}$ is the extrema considered for the domain of input variables ($[-10, 10]$ in this study). A test time problem outside this domain constitutes an extrapolation problem. While uDSR has mechanisms to escape such extrapolation cases, either by fine-tuning or via components that do not depend on input variable domains (e.g. AIF or GP), including the pre-trained network may be unnecessary or harmful. We investigate this in Table 6, in which we show the performance of uDSR and baselines against the six SRBench ground-truth problems whose input data exceed that considered during pre-training, $[-10, 10]$. Notably, standalone NeSymReS (which does not fine-tune) yields 0% in both symbolic and accuracy solution rates, suggesting that pre-training *without fine-tuning* on problems outside the pre-training input ranges can result in catastrophic failure. On the other hand, uDSR yields 100% in both symbolic and accuracy solution rates over these problems, suggesting that pre-training was not harmful, and that the algorithm's fine-tuning and/or other components can render uDSR robust to this type of extrapolation.

Table 6: Performance of symbolic and accuracy-based solution rates for uDSR and 16 baselines, averaged across the six SRBench problems whose input variables contain data outside the pre-training range, $[-10, 10]$.

Algorithm	Symbolic solution (%)	Accuracy solution (%)
uDSR	**100**	**100**
Operon	11.7	91.7
SINDy	50	50
FEAT	0	87.5
GP-GOMEA	20.8	56.3
SBP-GP	2.1	70.8
MRGP	0	64.6
AIF	16.7	37
AFP-FE	6.7	26.7
BSR	0	29.2
ITEA	0	29.2
AFP	2.1	4.2
FFX	0	6.3
DSR	2.1	2.1
EPLEX	0	2.1
GP (gplearn)	0	0
NeSymReS	0	0

In Figure 13, we also show how the minimum and maximum input variables of the Feynman, Strogatz, and black-box SRBench problems compare to those of the pre-training dataset generator.

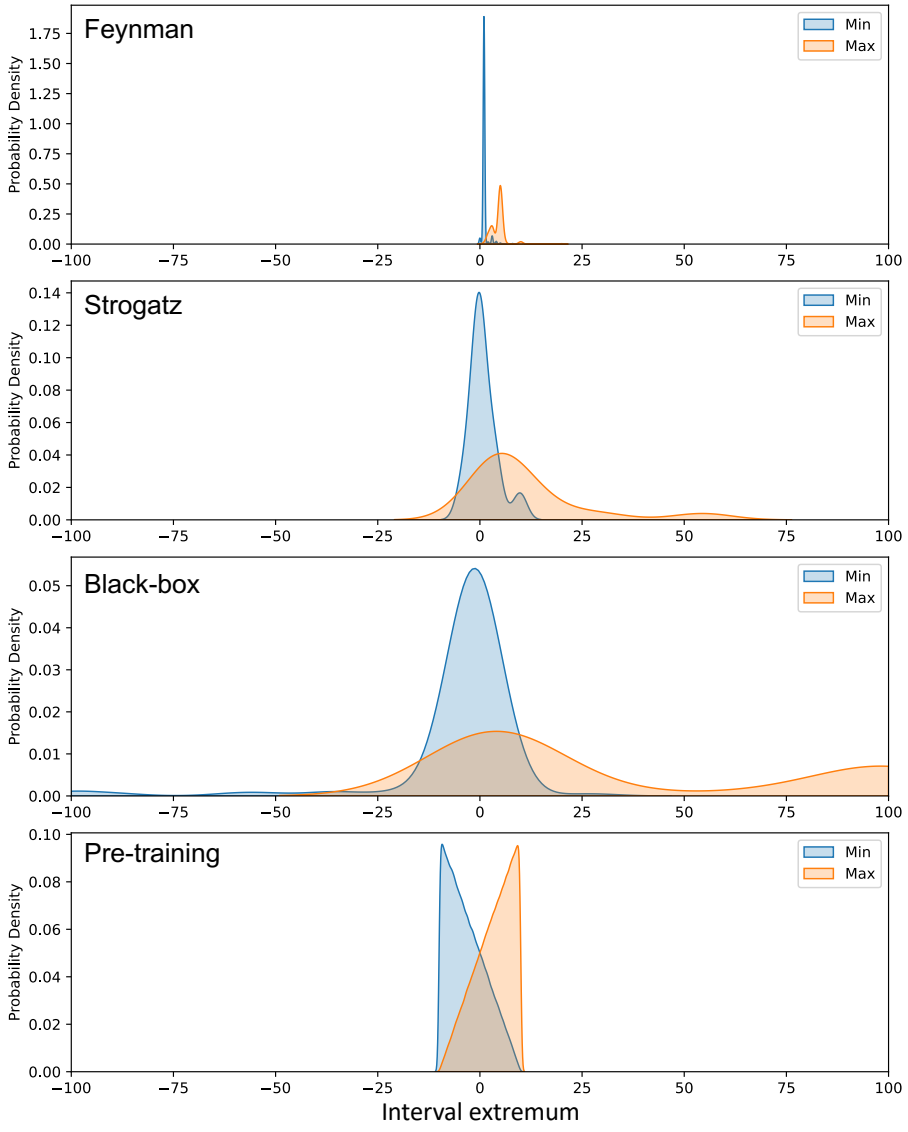

Figure 13: Kernel density estimates of the minimum (blue) and maximum (orange) input variable values across the 116 Feynman, 14 Strogatz, and 122 black-box SRBench problems, as well as 1M problems sampled from our pre-training dataset generator. Axes are truncated to [-100, 100] to highlight the regions of higest density.