Appendix for "A Unified Framework for Deep Symbolic Regression"

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, respectively, 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, LSPT, DSR, GP, LM

```
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 $\theta$ and encoder parameters $\psi$
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 $\theta, \psi$ 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 $\theta, \psi$ on $\mathcal{T}$ using RL    ▷ Train using RL

11: // Phase II: Test problem
12: $\mathcal{P}_1, \dots, \mathcal{P}_m \leftarrow \text{RecursiveSimplify}(\mathcal{P})$    ▷ 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 $s = 1, \dots, S$ 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 $\mathcal{T}_\text{GP}^{(1)} \cup \dots \cup \mathcal{T}_\text{GP}^{(S)}$    ▷ 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 $j = 1, \dots, N$ do
23:             if CONST $\in \tau^{(j)}$ then
24:                 $\xi^\star \leftarrow \arg\max_\xi R(\tau^{(j)}; \xi)$    ▷ Maximize $R$ w.r.t. $\xi$, e.g. with BFGS
25:                 $\tau^{(j)} \leftarrow \text{ReplaceConstants}(\tau^{(j)}, \xi^\star)$    ▷ 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 $\beta$, e.g. using LASSO
31:                 $\tau^{(j)} \leftarrow \text{ReplaceCoefficients}(\tau^{(j)}, \beta^\star)$    ▷ Replace LINEAR coefficients
32:         $\theta \leftarrow$ train $\theta$ (with $\psi$ fixed) on $\mathcal{T}$    ▷ Train using fine-tuning objective
33:     $\tau^{(i)} \leftarrow$ store best expression for problem $\mathcal{P}_i$
34: $\tau^\star \leftarrow \text{Combine}(\tau^{(1)}, \dots, \tau^{(m)})$    ▷ 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. In 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 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 of 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 LSPT.
- **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 LM

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 a mixed-integer linear problem, but solving it with mixed-integer programming techniques would be prohibitively expensive. LASSO regression (Tibshirani, 1996), as used by (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,

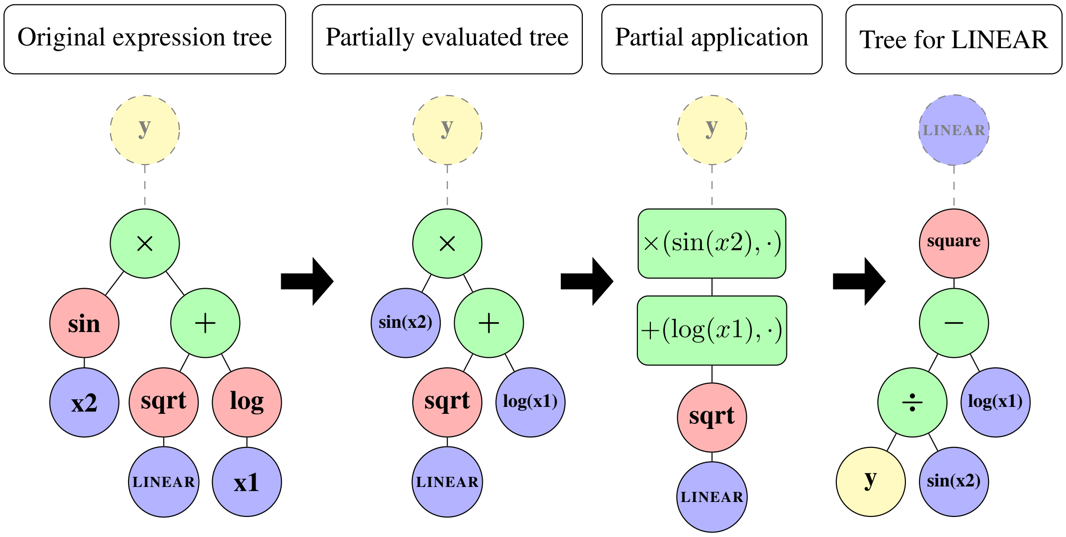

Figure 5: An example illustrating how to obtain the target data $\overline{f}^{-1}(y)$ for LINEAR. In this case, $F(x, \text{LINEAR}) = \sin(x_2)\left(\sqrt{\text{LINEAR}} + \log(x_1)\right)$ and $G(x, y) = \left(\frac{y}{\sin(x_2)} - \log(x_1)\right)^2$.

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

To efficiently carry out the above procedure repeatedly with Φ_X constant for a given SR problem, we compute $\Phi_X^\dagger$ and $(\Phi_X^\top \Phi_X)^{-1}$ 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 $(\Phi_X^\top \Phi_X)^{-1}$.

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 LSPT

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 architecture is chosen because it is permutation invariant, i.e., the output of the model does not change under any permutation of the rows of (X, y), and because it can process input sets of variable size. We use the ST architecture introduced in Lee et al. (2019) with the following hyperparameters (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 using an RNN as a decoder is inspired by image-to-caption systems (Karpathy and Fei-Fei, 2015).

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 LSPT "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 LSPT, 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 from the published algorithm in Biggio et al. (2021), which is restricted to problems with $d \leq 3$ 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 ~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 qualitatively closer to the real data than the randomly initialized model. Comparing traces between SL and RL pre-trained models, RL is 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.

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 instances, 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

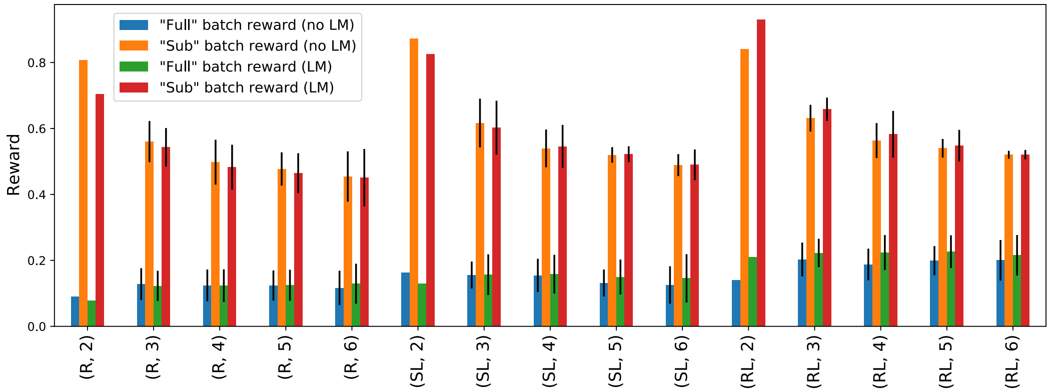

Figure 6: Average reward over 1000 expressions sampled from the pre-trained initial models (SL, RL) and a randomly initialized model (R) with and without the LINEAR token (LM) computed over the Feynman benchmarks. Results are stratified by problem dimension. The term “Full” refers to averages over the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

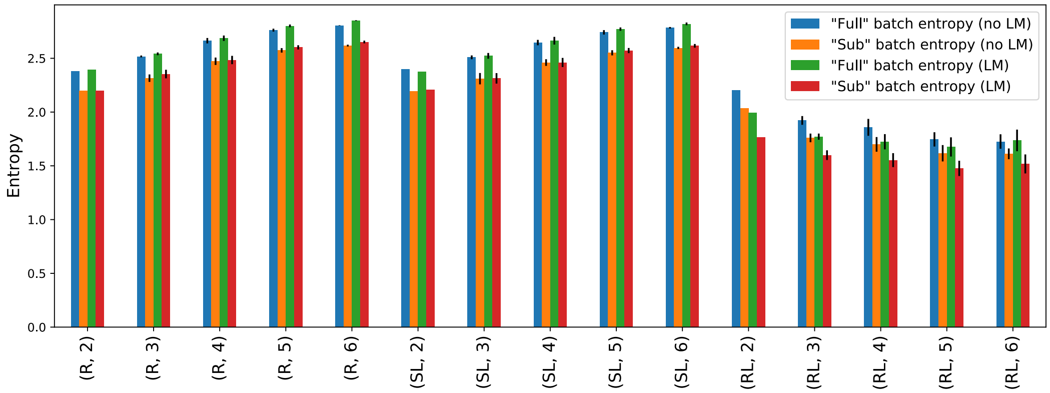

Figure 7: Average empirical entropy over 1000 expressions sampled from the pre-trained initial models (SL, RL) and a randomly initialized model (R) with and without the LINEAR token computed over the Feynman benchmarks. Results are stratified by problem dimension. The term “Full” refers to averages over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

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.

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 a 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),

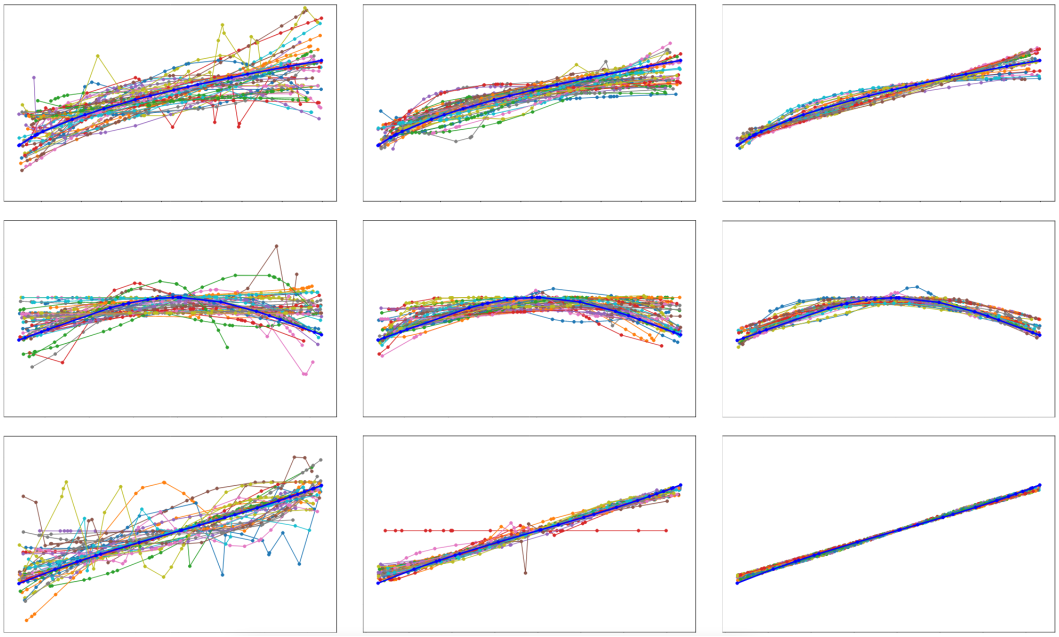

Figure 8: Traces of the top 5% of 1000 expression sampled from the pre-trained models for a series of one-dimensional problems. Columns correspond to controllers with random weights, SL pre-trained weights, and RL pre-trained weights, respectively. The blue traces correspond to the real (X, y) data.

giving:

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

A Monte Carlo estimate of the above is given by:

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

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
$m \leftarrow \text{dim}(\tau)$                                         ▷ Get dimensionality of $\tau$
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 $i \in 1, \dots, m$ do
    $a, b \sim \mathcal{U}(-10, 10)$                                    ▷ Sample domain extrema
    for $j \in 1, \dots, 500$ do
        $X_{i,j} \sim \mathcal{U}(\min(a, b), \max(a, b))$             ▷ Sample $X$ coordinate
$y = f(X)$                                                              ▷ Compute $y$ values
return $(X, y)$
```

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

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

and

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

where $\mathrm{Q}_\alpha(x)$ is the empirical α quantile of the vector x. This motivates a procedure to compute an approximation of the gradient $\nabla_{\theta,\psi} J_{\mathrm{riskRL}}(\theta, \psi)$: 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 multidimensional 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 than 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 LSPT, 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 LM, 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 $x_1 x_2$.

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} (LSPT has two "on" states, denoted "RL" and "SL"). Figure 9 shows that individually enabling AIF, GP, LM (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, LM 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 LSPT is negative on average, but it provides positive improvement in certain cases; specifically, the highest-performing combination uses LSPT (RL) (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. LSPT 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

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)
LSPT			
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)	–	0.3	Tuned $\{0, 0.3\}$
Risk factor (RL only)	ε	0.05	Tuned $\{0.05, 0.2\}$
LM			
Basis functions	Φ	Monomials	Estimated
Maximum monomial degree	D	3	Estimated
Maximum non-zero terms	–	10	Estimated

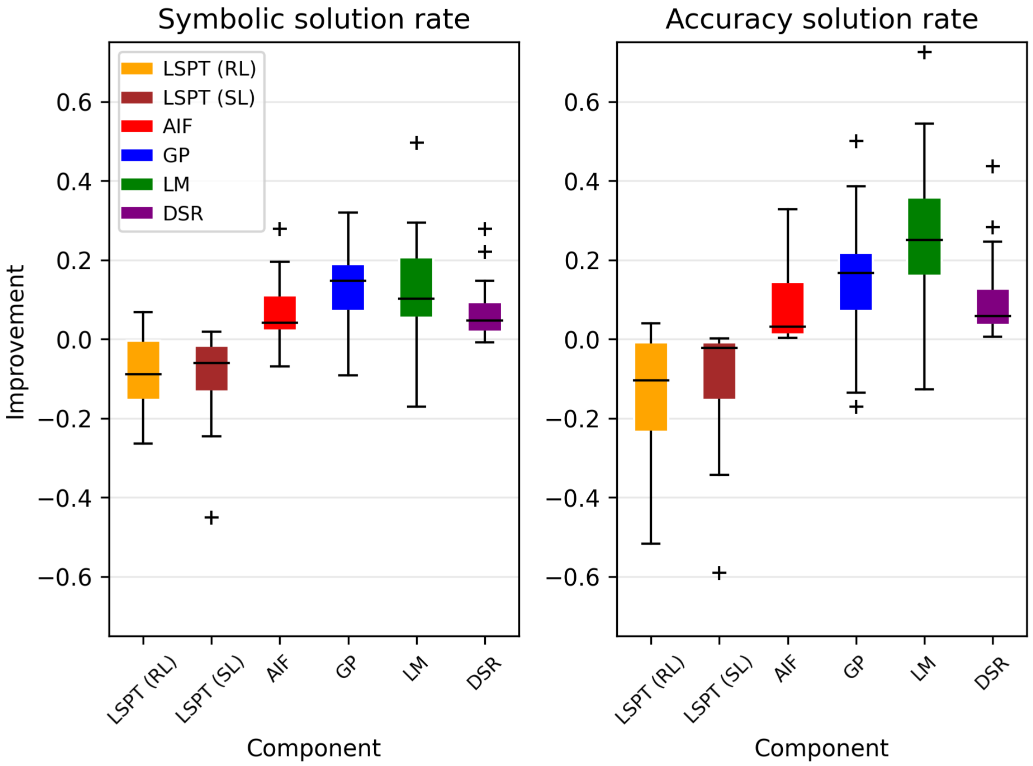

Figure 9: Distribution of improvement to symbolic solution rate (left) and accuracy solution rate (right) when each component is enabled, over all combinations of other components.

2-fold increase in runtime. LM 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., 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.

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.

GP	LM	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 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.

GP	LM	AIF	DSR	LSPT	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

I Accuracy vs complexity for GP and LM components

In our ablations on ground-truth SR problems, both GP and the LINEAR token (LM) 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 ∈ [on, off] × LM ∈ [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

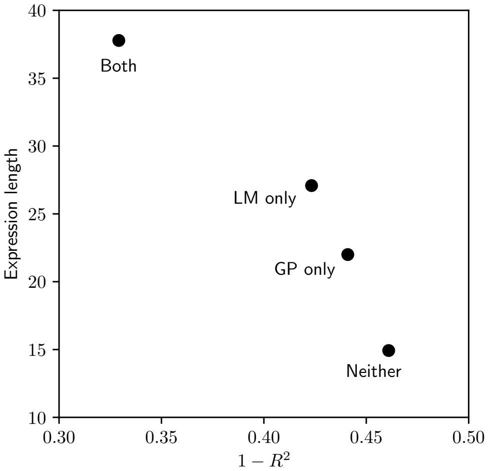

Figure 10: Accuracy vs complexity trade-offs for uDSR ablations with and without GP, and with and without the LINEAR token (LM). Each point is the median of 10 independent runs, averaged over the 122 SRBench black-box SR problems.

in the SymPy-parsed expression tree). This reinforces how the modularity of uDSR allows it to be configured based on use case.

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[3], which includes the base implementation of DSR and facilities to integrate GP using the Distributed Evolutionary Algorithms in Python software[4] (Fortin et al., 2012). We integrate AIF using its open-source implementation[5]. We implement the LINEAR token using standard numerical libraries. The set transformer used for pre-training is based on an open-source TensorFlow implementation[6]. Our empirical evaluation uses SRBench[7]. The SRBench benchmark datasets are taken from the open-source PMLB database[8] (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 solution strategies. In this section, we identify the closest *standalone SR algorithm* in the literature corresponding to each single-component uDSR ablation and summarize the key methodological differences.

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

---

[3] `https://github.com/brendenpetersen/deep-symbolic-optimization`. BDS-3-Clause license.
[4] `https://github.com/DEAP/deap`. LGPL-3.0 license.
[5] `https://github.com/SJ001/AI-Feynman`. MIT license.
[6] `https://github.com/arrigonialberto86/set_transformer`. Unlicensed.
[7] `https://github.com/cavalab/srbench`. GPL-3.0 license.
[8] `https://github.com/EpistasisLab/pmlb`. MIT license.

tokens that were tailored to the Feynman benchmark problems, whereas our component used a more generic set of tokens.

LSPT. The closest standalone algorithm to our LSPT component is NeSymReS (Biggio et al., 2021). The key difference between NeSymReS (Biggio et al., 2021) and our LSPT component lies in how expressions are sampled at test time. NeSymReS uses *beam search* at test time to identify high-likelihood expressions under the model conditioned on the test problem. In contrast, our LSPT 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 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. Our GP component is most similar to a vanilla, "Koza-style" GP algorithm (Koza, 1994). 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.

LM. Our LM component is most similar to the SINDy algorithm (Brunton et al., 2016). The key difference between the published, standalone SINDy algorithm (Brunton et al., 2016) and our LINEAR token is that the original SINDy algorithm finds expressions of 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 LM-only uDSR ablation includes LINEAR within nonlinear functional forms. Two minor differences are that: (1) The original SINDy algorithm used LASSO to learn sparse coefficients, whereas our module uses a modified version of least squares, and (2) The original SINDy algorithm chose 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 solution strategies. However, note that in Figures 2 and 3, the SRBench 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 single-component uDSR ablation to the corresponding "closest" standalone SR method. 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 LSPT-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

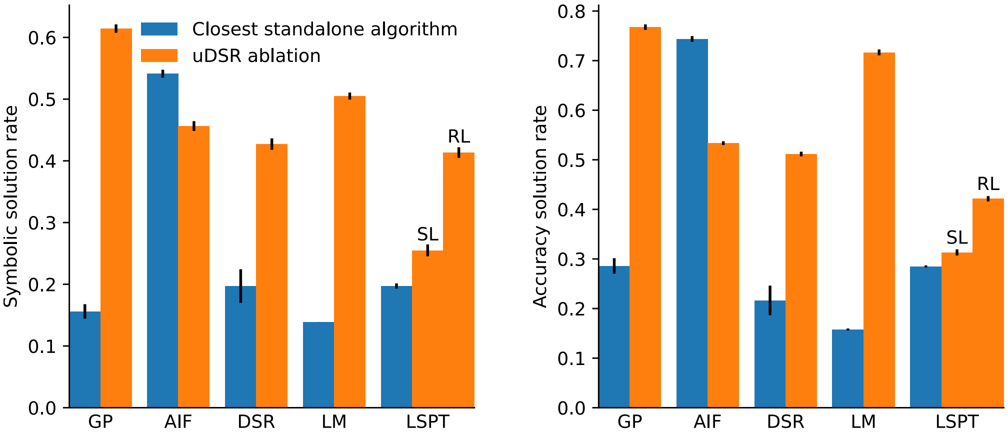

Figure 11: Comparisons between each of the five single-component uDSR ablations and its closest corresponding standalone SR algorithm, averaged across 130 ground-truth noiseless SR problems. Left: symbolic solution rate. Right: accuracy solution rate. Error bars represent standard error across 10 random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

of DSR given by Landajuela et al. (2021b), performance is identical to the DSR-only uDSR ablation, by construction. The LM-only uDSR ablation greatly outperforms standalone SINDy. This is because SINDy only optimizes a single linear solution, whereas the LM-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 discovery

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 *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 LSPT 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 $\tau = [\text{exp, LINEAR}]$ 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.

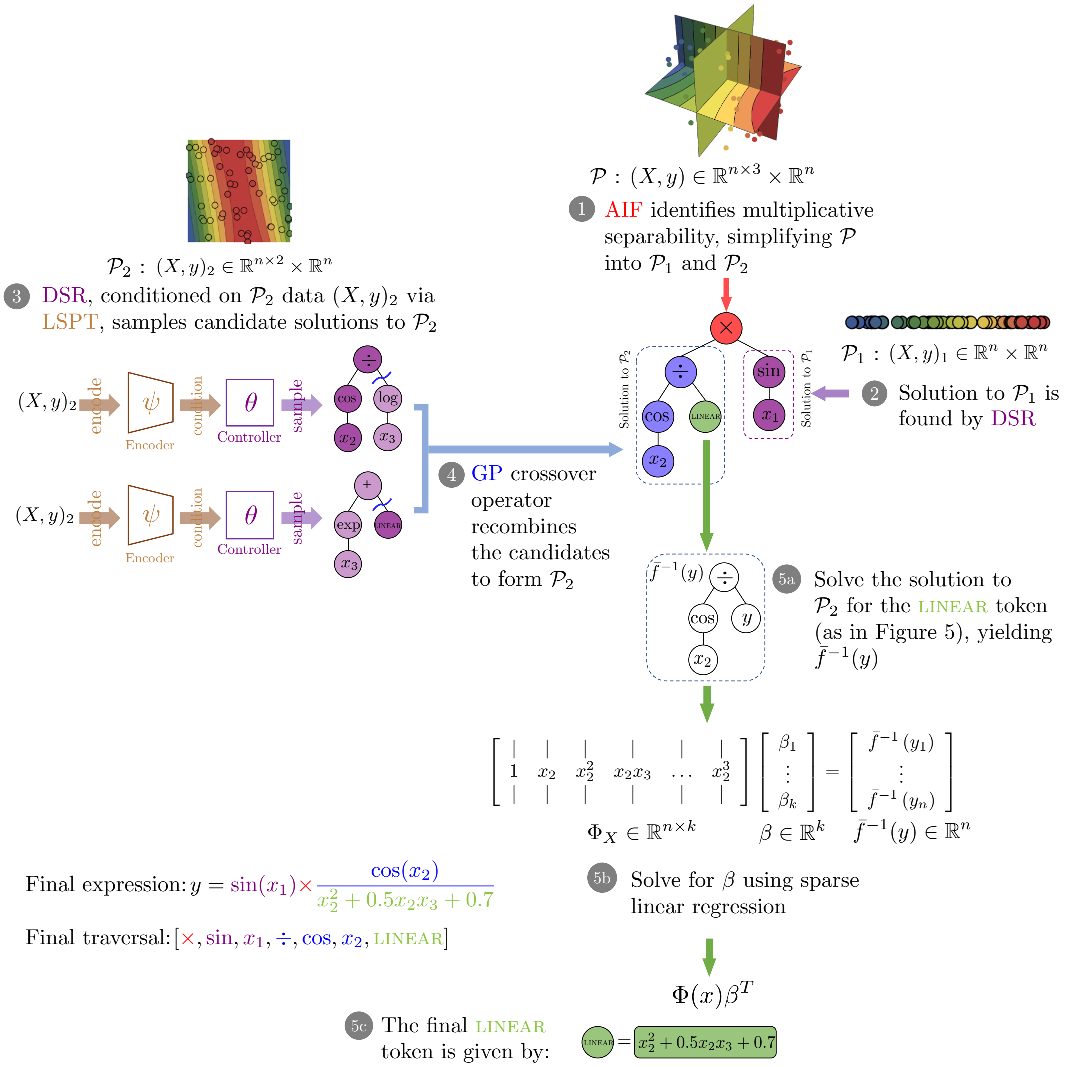
$\mathcal{P} : (X, y) \in \mathbb{R}^{n\times 3} \times \mathbb{R}^n$
1 AIF identifies multiplicative separability, simplifying $\mathcal{P}$ into $\mathcal{P}_1$ and $\mathcal{P}_2$
$\mathcal{P}_2 : (X, y)_2 \in \mathbb{R}^{n\times 2} \times \mathbb{R}^n$
3 DSR, conditioned on $\mathcal{P}_2$ data $(X, y)_2$ via LSPT, samples candidate solutions to $\mathcal{P}_2$
$(X, y)_2$
encode
ψ
Encoder
condition
θ
Controller
sample
÷
cos
log
x_2
x_3
+
exp
LINEAR
x_3
$\mathcal{P}_1 : (X, y)_1 \in \mathbb{R}^n \times \mathbb{R}^n$
2 Solution to $\mathcal{P}_1$ is found by DSR
×
sin
x_1
Solution to $\mathcal{P}_2$
Solution to $\mathcal{P}_1$
4 GP crossover operator recombines the candidates to form $\mathcal{P}_2$
$\bar{f}^{-1}(y)$
y
5a Solve the solution to $\mathcal{P}_2$ for the LINEAR token (as in Figure 5), yielding $\bar{f}^{-1}(y)$
1 x_2 x_2^2 x_2x_3 … x_2^3
β_1 ⋮ β_k
$\bar{f}^{-1}(y_1)$ ⋮ $\bar{f}^{-1}(y_n)$
$\Phi_X \in \mathbb{R}^{n\times k}$
$\beta \in \mathbb{R}^k$
$\bar{f}^{-1}(y) \in \mathbb{R}^n$
5b Solve for β using sparse linear regression
$\Phi(x)\beta^T$
5c The final LINEAR token is given by:
LINEAR $= x_2^2 + 0.5x_2x_3 + 0.7$
Final expression: $y = \sin(x_1) \times \frac{\cos(x_2)}{x_2^2 + 0.5x_2x_3 + 0.7}$
Final traversal: $[\times, \sin, x_1, \div, \cos, x_2,$ LINEAR$]$

Figure 12: Illustrative example of how the five interacting uDSR components work together to find an expression for dataset $\mathcal{P}$. In the illustration of datasets $\mathcal{P}, \mathcal{P}_1$ and $\mathcal{P}_2$, points' coordinates are given by their X values (giving 3, 2 and 1-dimensional representations respectively) and points are colored by their y value. In the final solution, some parts of the expression may be traced back to particular components. Color key: AIF, LSPT, DSR, GP, LM.

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 SINDy and NeSymReS (the standalone SR methods closest to our LM and LSPT components, respectively) were not included in the original SRBench study, we

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
LM	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

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 LSPT

A key aspect regarding recommended use of LSPT 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, 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.

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.

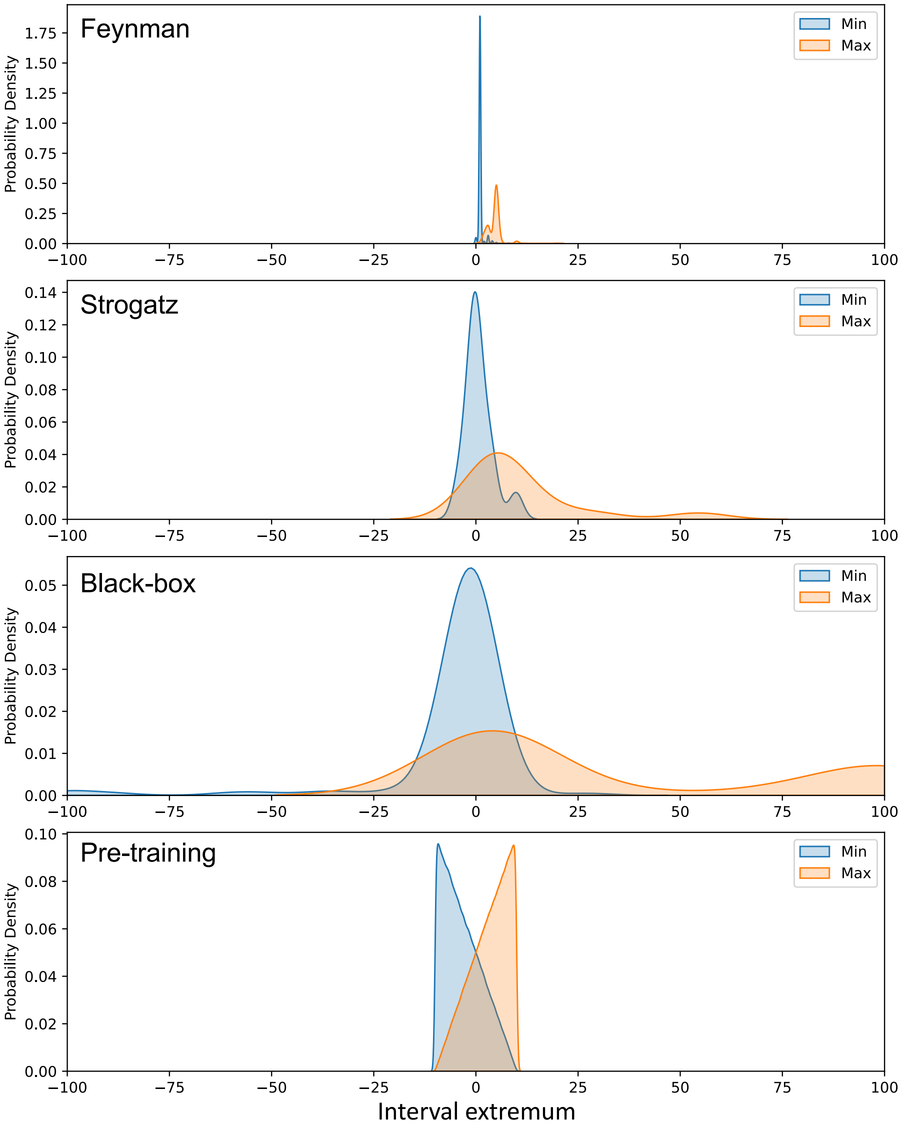

Figure 13: Kernel density estimates of the minimum (blue) and maximum (orange) input variable values across the 116 Feynman, 14 Strogatz, and 122 black-box SRBench problems, as well as 1M problems sampled from our pre-training dataset generator. Axes are truncated to [-100, 100] to highlight the regions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