Deep Generative Symbolic Regression with Monte-Carlo-Tree-Search

Pierre-Alexandre Kamienny [1][2] **Guillaume Lample** [1] **Sylvain Lamprier** [3] **Marco Virgolin** [4]

Abstract

Symbolic regression (SR) is the problem of learning a symbolic expression from numerical data. Recently, deep neural models trained on procedurally-generated synthetic datasets showed competitive performance compared to more classical Genetic Programming (GP) algorithms. Unlike their GP counterparts, these neural approaches are trained to generate expressions from datasets given as context. This allows them to produce accurate expressions in a single forward pass at test time. However, they usually do not benefit from search abilities, which result in low performance compared to GP on out-of-distribution datasets. In this paper, we propose a novel method which provides the best of both worlds, based on a Monte-Carlo Tree Search procedure using a context-aware neural mutation model, which is initially pre-trained to learn promising mutations, and further refined from successful experiences in an online fashion. The approach demonstrates state-of-the-art performance on the well-known `SRBench` benchmark.

1. Introduction

Symbolic Regression (SR) is the problem of simultaneously finding the structure (operators, variables) and optimising the parameters (constants) of an expression that describes experimental data in an interpretable manner. SR can produce human-readable expressions that are particularly useful in natural sciences, e.g., materials sciences (Wang et al., 2019; Kabliman et al., 2021; Ma et al., 2022) or physics (Schmidt & Lipson, 2009; Vaddireddy et al., 2020; Sun et al., 2022; Cranmer et al., 2020; Hernandez et al., 2019; Udrescu & Tegmark, 2020). The recent benchmarking effort `SRBench` (La Cava et al., 2021) has shown that SR

algorithms can additionally outperform over-parametrized models, e.g., decision-tree ensembles or neural networks, on a set of real-world and synthetic datasets.

SR is a challenging task, which implies composing inherently discrete objects (operators and variables) to make the resulting expression fit well the given data. As seeking the optimal composition is intractable (Virgolin & Pissis, 2022), typical approaches to SR are based on heuristics. In fact, the leading algorithms on `SRBench` are modern versions of genetic programming (GP) (Koza, 1994), a class of genetic algorithms where evolving solutions need to be executed to determine their quality (i.e., SR expressions need to be evaluated on the data to determine their quality of fit).

Lately, there has been a growing interest in the SR community for neural network-based approaches. For example, Udrescu & Tegmark (2020) use neural networks (NNs) to explicitly detect data properties (e.g., symmetries in the data) useful to reduce the search space of SR. Other approaches, which we group together under the name of Deep Generative Symbolic Regression (`DGSR`), have used NNs to learn to generate expressions directly from the data. Similarly to GP, DSR (Petersen et al., 2019) and (Mundhenk et al., 2021) faces a *tabula rasa* setup of the problem for each new dataset. On the other hand, several `DGSR` approaches are inductive: they are pre-trained to predict an expression in a single forward pass for any new dataset, by feeding the dataset as input tokens (Biggio et al., 2021; Valipour et al., 2021; Kamienny et al., 2022). As such, these approaches have the appeal of generating expressions extremely quickly. However, their lack of a search component makes them unable to *improve* for the specific dataset at hand. This aspect can be particularly problematic when the given data is out-of-distribution compared to the synthetic data the NN was pre-trained upon.

A promising direction to cope with the limitations of inductive `DGSR` is therefore to include a suitable search strategy. The use of neural policies in Monte-Carlo Tree Search (MCTS) has led to improved exploration via long-term planning in the context of automated theorem proving with formal systems (Polu & Sutskever, 2020; Lample et al., 2022), algorithm discovery (Fawzi et al., 2022), and also games (Silver et al., 2016; 2018). In the context SR, search tree nodes are expressions (e.g., $x_1 + x_2$), and edges between

---

[1]Meta AI, Paris, France [2]ISIR MLIA, Sorbonne Université, France [3]LERIA, Université d'Angers, France [4]Centrum Wiskunde & Informatica, the Netherlands. Correspondence to: Pierre-Alexandre Kamienny <pakamienny@meta.com>.

them represent mutations (e.g., $x_2 \to (7 \times x_3)$, leading to the expression $x_1 + 7 \times x_3$). White et al. (2015); Sun et al. (2022) proposed a classic formulation of MCTS for SR, where possible mutations are pre-defined along with prioritization rules to decide which nodes to mutate in priority. Li et al. (2019) use a recurrent neural network to produce possible mutations, which is conditioned on shape constraints for the expressions, but not on the dataset. Most similar to our approach is the study of Lu et al. (2021), that uses a pre-trained model to sample promising but rather simple mutations (up to one expression term). However, the model is not fine-tuned on the specific dataset at hand as the search progresses. In our proposal, MCTS is augmented with neural policies that are both pre-trained and fine-tuned. Existing works have only showed good performance on simple benchmark problems (e.g., no observed competitive performance on `SRBench` real-world problems). Furthermore, we found in preliminary experiments that the combination of NN within MCTS with pre-training is key to achieve good performance.

Summary of the approach. In this paper, we seek to overcome the limitations of `DGSR`, by proposing a synergistic combination, which we call `DGSR-MCTS`, where MCTS is seeded with pre-trained `DGSR`, and `DGSR` is fine-tuned over time on multiple datasets simultaneously as the search progresses. A *mutation* policy is responsible for producing mutations that expand expressions from the tree search. A *selection* policy prioritizes which expression to mutate next, by trading-off between exploration (low number of visits) and exploitation (most promising expressions) using a critic network. The mutation policy first undergoes a pre-training phase. Then, both the mutation policy and the critic network are updated in online fashion, by leveraging results from search trials on new provided datasets.

We first position our approach in an unifying view of SR, that encapsulates both earlier work in GP and more recent deep learning-based approaches. Then, we describe our approach in detail, and evaluate it on the `SRBench` benchmark. We show that our approach achieves state-of-the-art performance, and discovers expressions that are both simple and accurate.

2. Unifying View of of SR

Let us denote by $\mathcal{F}$ the family of symbolic expressions[1] that form the search space of the SR problem. Generally speaking, $\mathcal{F}$ is defined by the set of building blocks from which expressions can be composed (Virgolin & Pissis, 2022).

---

[1]Note that even though different expressions may be functionally equivalent, this is normally not taken into account in existing approaches, as determining functional equivalence can be undecidable (Buchberger & Loos, 1982).

These usually include constants (possibly sampled from a distribution, e.g., $\mathcal{N}(0,1)$), variables, and basic operators as well as trigonometric or transcendental ones (e.g., $+, -, \times, \div, \sin, \cos, \exp, \log$).

A common formulation of SR poses to seek a well-fitting expression for the dataset at hand. We now generalize this idea to the case where good performance is sought across multiple datasets (as one usually seeks a generally-competent search algorithm rather than a dataset-specific one). Given Ω a distribution over datasets $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}_{i \leq N}$, with $(\mathbf{x_i}, y_i) \in \mathbb{R}^d \times \mathbb{R}$ a given point (descriptor, image) from a specific dataset $\mathcal{D}$, and a limited budget for the exploration process T, the general objective of SR is to define an algorithm that produce distributions of expressions $f \in \mathcal{F}$, that minimize the following theoretical risk at step T:

$$\mathcal{R}_{\Omega, \mathcal{F}} = \mathbb{E}_{\mathcal{D} \sim \Omega(\mathcal{D})} \mathbb{E}_{f \sim P_f^T(\mathcal{D})}[\mathcal{L}(f, \mathcal{D})] \qquad (1)$$

with $P_f^T(\mathcal{D})$ the final distribution over expressions provided by the considered algorithm. A typical loss function $\mathcal{L}$ considered in the SR literature is the negative R^2:

$$R^2(f, \mathcal{D}) = 1 - \frac{MSE(\mathbf{y}, f(\mathbf{x}))}{VAR(\mathbf{y})} = 1 - \frac{\sum_i (y_i - f(\mathbf{x}_i))^2}{\sum_i (y_i - \overline{y}_i)^2}. \qquad (2)$$

where $\overline{y}_i = (1/N) \sum_{i \leq N} y_i$.

Considering algorithms that start from $P_f^0(\mathcal{D})$ to incrementally build $P_f^T(\mathcal{D})$, the problem can be decomposed into two main steps: 1) define an accurate starting point $P_f^0(\mathcal{D})$ for the search process; 2) specify the exploration process that allows to update $P_f^0(\mathcal{D})$ to form $P_f^T(\mathcal{D})$ in a maximum of T iterations (search trials). Some approaches in the literature only consider the first step (i.e., $P_f^T(\mathcal{D}) = P_f^0(\mathcal{D})$ —no search process). Others only investigate step 2 (i.e., $P_f^0(\mathcal{D}) = P_f^0(\emptyset)$ —no inductive context), as described in the following.

There are many ways to define P_f^0. For example, traditionally in GP, the dataset does not play a role (i.e., it is $P_f^0(\emptyset)$), but there exist different strategies to randomly initialize the population (Looks, 2007; Pawlak & Krawiec, 2016; Ahmad & Helmuth, 2018). Since we deal with `DGSR`, we focus on $P_f^0(\mathcal{D})$ that entail pre-training from here on. Note that Ω is unknown during this pre-training, and that we only observe its datasets at search time (datasets from `SRBench` in our experiments).

2.1. Pre-training: How to define $P_f^0(\mathcal{D})$

The formulation of Equation (1) is reminiscent of meta-learning or few-shot learning where the goal is to train a model on a set of training tasks such that it can solve new ones more efficiently (Schmidhuber, 1987; Thrun & Pratt, 2012; Finn et al., 2017). In that vein, many approaches focus

Table 1. Unifying view of SR. θ represents weights of a probabilistic neural network that embodies P_f. ψ is parameters of a critic network (see Section 3).

Algorithm	Pre-train of P_f^0	P_f^t conditioned by	Update of P_f^t
GP (Poli et al., 2008)	No	Population & Genetic operators	Selection operators
EDA (Kim et al., 2014)	No	Explicit Factorization	Selection operators
E2E (Kamienny et al., 2022)	SSL on synthetic data	$\mathcal{D}$ & θ^0	No
DSR(Petersen et al., 2019)	No	θ^t	Update θ^t with policy gradients
uDSR (Landajuela et al., 2022)	SSL on synthetic data	$\mathcal{D}$ & θ^t	Update θ^t with policy gradients
DGSR-MCTS (Ours)	SSL and MCTS on synthetic data	MCTS using $\mathcal{D}$ & θ^t & ψ^t	Update θ^t & ψ^t via Selection & Imitation

on learning a generative model to induce an implicit form of P_f^0. However, since Ω is unknown during pre-training, those models are usually trained on large synthetic datasets built from randomly generated expressions (Lample & Charton, 2019).

Rather than focusing on the accuracy of the sampled expressions regarding some criteria such as Equation (2), the vast majority of these approaches are neural language models (and thus belong to the DGSR family) trained to express P_f^0 by auto-regressively predicting a given ground truth expression from the training data (e.g., using a recurrent neural network or a transformer).

Although these methods tend to produce expressions similar to the synthetic ones seen during pre-training, they have the advantage of explicitly considering the dataset as an input: P_f is conditioned on $\mathcal{D}$ (Biggio et al., 2021; Valipour et al., 2021; Kamienny et al., 2022). Consequently, pre-training approaches for DGSR can produce expressions as a one-shot inference process by the simple action of sampling, eliminating the need for search iterations. However, sampling from P_f^0 is limited as the accuracy improvement stops after a small number of samples (50 in (Kamienny et al., 2022)), and can perform badly on out-of-domain datasets given at test-time.

2.2. Search Process: How to build $P_f^T(\mathcal{D})$

Given a dataset $\mathcal{D}$, the aim of any process at search time is to define a procedure to build an accurate distribution $P_f^T(\mathcal{D})$ from the starting one $P_f^0(\mathcal{D})$, via a mix of exploration and exploitation. Most approaches seek the maximization of $\mathbb{E}_{f \sim P_f^T}[R^2(f, \mathcal{D})]$. Several approaches additionally include in that objective a functional $C(f)$ that penalizes overly-complex expressions (Vladislavleva et al., 2008; Chen et al., 2018). Even though there exists different definition of *complexity* (Kommenda et al., 2015; Virgolin et al., 2020), we consider *expression size* (i.e., the number of operators, variables and constants in the expression) as complexity measure.

In that aim, the learning process of different SR algorithms

can be unified into a general, iterative framework: until the termination condition is not met (e.g., runtime budged exceeded or satisfactory accuracy reached),

(i) sample one or more symbolic expressions $f \in \mathcal{F}$ using the current probability distribution P_f^t at step t;

(ii) update P_f^t using statistics of the sample, such as the accuracy of the expression.

Steps (i) and (ii) constitute an iteration of the framework. Different SR algorithms have considered various definitions of P_f^t and strategies to update it. Its definition can be implicit, explicit, or learned, and be biased towards certain expression structures. Table 1 provides a non-exhaustive summary of popular algorithms, which we elaborate upon in the following paragraphs.

Genetic Programming (GP) is a meta-heuristic inspired by natural evolution and is a popular approach to SR (Koza, 1994; Poli et al., 2008). GP implements step (i) by maintaining a population of expressions which undergoes stochastic modifications (via crossover and mutation), and step (ii) by selection, i.e., stochastic survival of the fittest, where better expressions are duplicated and worse ones are removed. In other words, sampling and updating P_f^t is implicitly defined by the heuristics that are chosen to realize the evolution.

Estimation of Distribution Algorithms (EDAs) attempt to be more principled than GP by explicitly defining the form of P_f^t (Salustowicz & Schmidhuber, 1997; Hemberg et al., 2012; Kim et al., 2014). Normally, P_f^t is factorized according to the preference of the practitioner, e.g., expression terms can be modelled and sampled independently or jointly in tuples, or chosen among multiple options as the search progresses, using, e.g., the minimum description length principle (Harik et al., 1999). EDAs use methods similar to those of GP to realize step (ii).

DSR (Petersen et al., 2019) proposed an approach where P_f^t is realized by an NN, whose parameters θ are updated using policy gradients with the accuracy of sampled expressions as rewards. Though being very general and

only biased by the model parametrization θ, this approach was found to generate very short expressions with low accuracy on `SRBench` (see Section 4). This is likely due to sparse reward and credit assignment issues typical of reinforcement learning (Sutton, 1984). (Mundhenk et al., 2021) adds GP search samples on top of DSR. Very recently, the work by Petersen et al. (2019) was integrated with a pre-training component (Landajuela et al., 2022), albeit in a large pipeline that also includes GP, NNs for search space reduction (Udrescu & Tegmark, 2020), and linear regression.

In the recent benchmarking effort `SRBench` by La Cava et al. (2021), modern GP algorithms have shown to be the most successful. An interesting property that distinguishes GP from other `DGSR` approaches is that its crossover and mutation operators tend to edit existing expressions (thus preserving substantial parts of them), rather than sampling them from scratch (Koza, 1994; Poli et al., 2008). The algorithm we develop in Section 3 expands expressions over time, similarly to GP. At the same time, our algorithm is powered by pre-training and its ability to learn over time, a feature missing from GP and other approaches.

3. Method

Following the unified framework of SR detailed in previous section, we derive our method, `DGSR-MCTS`, as an expert-iteration algorithm (Anthony et al., 2017), which iteratively 1) samples expressions from P_f^t and 2) updates P_f^t by improving the distribution via imitation learning (i.e., log-likelihood maximization of solution expressions).

Recall that in `DGSR` methods, sampling expressions from P_f^t involve producing tokens step-by-step by sampling from a next-token distribution with techniques such as Monte-Carlo sampling or Beam-search. Turning pre-trained `DGSR` methods into ones that can update P_f^t with expert-iteration is a challenging task as 1) reward signal can be very sparse (i.e., very few sequences may correspond to accurate expressions for the given dataset); 2) such left-to-right blind way of decoding does not allow for accurate planning; 3) using accuracy objectives to guide the decoding would be difficult with such auto-regressive approaches, since intermediate sequences of tokens are not valid expressions in that setting.

Rather, we propose to derive P_f^t as the distribution induced by a Monte-Carlo Tree Search (MTCS) process (Browne et al., 2012), which we iteratively refine via imitation learning on samples it produces that solve the problem. Our MCTS process also considers expression mutations, following a mutation policy M_θ^t which deals with transformations of existing expressions, rather than a greedy concatenation of tokens from a given vocabulary, allowing a more systematic evaluation of intermediate solutions for a more guided exploration. This section first explains our MCTS search

process then the way we pre-train the mutation policy from synthetic data.

3.1. MCTS Sampling

Our sampling process of new expressions is derived from an MCTS process, where each node of the search tree[2] corresponds to an expression $f \in \mathcal{F}$. Following a similar process as in (Lample et al., 2022), our MCTS process is made of three steps: 1) Selection, 2) Expansion, 3) Back-propagation, that we detail below. In our MCTS, M_θ is used to sample promising mutations that populate the MCTS search tree via expansions (described below). Besides M_θ, we additionally leverage a critic network C_ψ to assess how likely an expression can lead to a solution. We use the same NN (weights are shared) to realize M_θ and C_θ, with the exception that C_θ includes an additional head that is trained during the process to output a value in $\mathbb{R}$. These three steps are iteratively repeated 1000 times, a period[3] that we call *search trial*, before M_θ and C_ψ is updated.

Selection The selection step of MCTS aims at following a path in the search tree, from its root, i.e. the empty expression, to a leaf in the search tree, that trades off between exploration and exploitation. Following PUCT (Silver et al., 2016), at each new node f, we select its child f' that maximizes:

$$V(f') + p_{uct}E(f')M_\theta(f'|f, \mathcal{D}, \theta) \qquad (3)$$

with $E(f') = \frac{\sqrt{\sum_{f' \in \text{child}(f)} N(f')}}{1+N(f')}$. $p_{uct} \in \mathrm{R}^+$ is the exploration coefficient, $N(f)$ is the number of times f was selected so far during the search, and $V(f)$ is an estimate of the expected value of the expression, expressed as $v(f)/N(f)$, with $v(f)$ accumulating values of all the nodes depending on f in the tree search. This selection criteria balances exploration and exploitation, controlled by hyper-parameter p_{uct}.

Expansion Once the selection step reaches a leaf f of the tree, this leaf is expanded to produce new child nodes by applying K mutations to f, sampled from $M_\theta(\mathcal{D}, f, \theta)$. This leads to modified expressions $\{f'_k\}_{k \leq K}$. In our case, the distribution $P_f(f'|f, \mathcal{D}, \theta)$ is induced by the application of $m \sim M_\theta$ on f, resulting in $f' = m(f)$. For each $k \leq K$, we add a node f'_k to the search tree, as well as an edge between f'_k and f, labeled m_k.

Each expression resulting from an expansion is evaluated (with or without constant optimization as discussed in Section 4.1) to check whether it *solves* the SR problem. Here,

---

[2]Not to be confused with tree-based representations of expressions, which we discussed in the previous section.

[3]This corresponds to a single step i) in the unifying framework of Section 2.
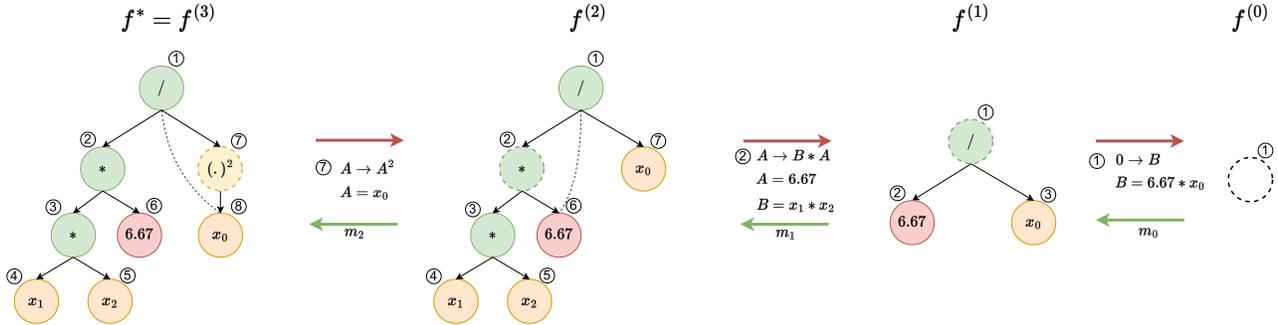
Figure 1. Example of data generation to train the mutation model. Given a starting ground-truth expression (e.g., $f^*(x_0, x_1, x_2) = 6.67x_1x_2/x_0^2$ as a tree, we procedurally dismantle the tree until no node is left. This is done by, at each step (red arrows), a) picking a node (dashed contour), b) removing the picked node and, if the operator is binary, additionally remove the subtree rooted in one of the two child nodes B, c) adding an edge (black dotted line) between the parent node and the remaining child node A to obtain a well-formed expression. When the picked node is the root node, the entire tree is removed, and the dismantling stops. Then, we train the mutation model to assemble the tree back via subsequent mutations (green arrows), which revert the dismantling process. The mutation model is conditioned on the current tree (initially empty) as well as the dataset $\mathcal{D}$.

we assess whether a relatively high accuracy is reached to determine whether the expression is a solution for the given dataset (we use $R^2 \geq 0.99$ in our experiments). Nodes that solve the problem obtain a value $v(f) = 1$. Others obtain an estimate from the critic network: $v(f) = C_\psi(\mathcal{D}, f)$. We remark that a simpler strategy is to define $v(f)$ as the accuracy of f, i.e. $v(f) = R^2(f, \mathcal{D})$. However, this strategy usually induces deceptive rewards, because a few mutations that lead to less accurate expressions (e.g., akin to sacrificing pieces in chess) may be needed before a very accurate expression is found (resp., we obtain a winning position). We confirmed that our strategy outperforms the naive use of accuracy in preliminary experiments

Back-Propagation   After each expansion, the value of every ancestor f' of any new node f is updated as $V(f') \leftarrow V(f') + v(f)$. Note that this form of back-propagation regards weighing the nodes of the MCTS search tree, and should not be confused with the algorithm used to train NNs.

As mentioned before, selection, expansion, and back-propagation are repeated 1000 times, after which the search trial is completed. At completion, the parameters of M_θ and C_ψ are updated as described in Section 3.2. Finally, the MCTS search tree is reset, and built anew using updated M_θ and C_ψ during the next trial.

3.2. Learning critic C_ψ and M_θ

After each search trial, we update the two parametrized components C_ψ and M_θ. To that end, training samples from the previous search trials are stored in two separate first-in-first-out queues: a buffer stores mutation sequences $(f^{(\tau)}, m_t)$ that produced a solution expression f^* to update

M_θ[4], the other contains V values of nodes. For the latter, nodes that lead to a solution expression f^* are assigned a score of 1.0. Others are considered for training only if their visit count is higher than a given threshold N_{visits}^{min}, in order to focus training on sufficiently reliable estimates. At each training step, batches containing an equal proportion of mutation and critic data points are sampled from the queues to train M_θ and C_ψ respectively. Both training objectives are weighted equally. To prevent any mode collapse, we continue sampling training examples from the supervised data used to pre-train the mutation model M_θ, as described in Section 3.3.

Note that even though the pre-training data generation is biased in different ways, stochasticity of M_θ enables its adaptation over search trials thanks to its updates; for instance, even if M_θ was trained to output mutations with arguments of size $B \approx 10$ can learn mutations of size 1, and vice-versa. As a result, M_θ can automatically learn the appropriate (and dataset-dependent) size of mutations through MCTS search.

We set up our MCTS search to simultaneously operate on multiple datasets at the same time, so that M_θ and C_ψ can leverage potential information that is shared across different SR instances (transfer learning). Given a set of datasets, a *controller* schedules a queue of datasets that includes a proportion of unsupervised (i.e. the ground-truth expression is not known) datasets to send to *workers* in charge of handling search trials. To avoid catastrophic forgetting, we also continue training on pre-training examples from synthetic datasets i.e. with example mutations as described in Section 3.3, in the spirit of AlphaStar (Vinyals et al., 2019)

---

[4]If multiple such sequences exist, we select the one with the smallest number of mutations.

or HTPS (Lample et al., 2022) with human annotated data. M_θ and C_ψ updates are tackled by *trainers*.

3.3. Mutation Policy

Our search process relies on a mutation policy M_θ^t, a distribution over promising mutations of a given expression $f \in \mathcal{F}$. In what follows, we drop the t index from M_θ for clarity. Expressions are represented as a tree structure and mutations act by modifying it.

Definition of M_θ. We define mutations as transformations that can be applied on any node of a given expression tree. Table 2 contains the list of considered transformations, where A stands for an existing node of the expression tree and B stands for a new (sub-)expression to be included in the tree. The $\rightarrow$ symbol represents the replacement operation, e.g., $A \rightarrow A + B$ means that node A from f is replaced by a new addition node with A as its left child and B its the right one. Thus, a valid mutation from M_θ is a triplet $< A, op, B >$, where A is a node from the expression tree, op is an operation from Table 2 and B is a new expression whose generation is described below. A constant optimization step, detailed in Appendix E.2, can be performed after the mutation to better fit the data; we explore in Section 4 whether including constant optimization improves the performance. We call mutation *size* the size of the expression B.

Table 2. Set of operators that can be applied on a sub-expression A of a function, with optional argument B as a new sub-expression to include in the tree structure. 0 refers to the root node of the function.

Unary	$A \rightarrow \cos(A), A \rightarrow \sin(A), A \rightarrow \tan(A)$
	$A \rightarrow \exp(A), A \rightarrow \log(A)$
	$A \rightarrow A^{0.5}, A \rightarrow A^{-1}, A \rightarrow A^2, 0 \rightarrow B$
Binary	$A \rightarrow A + B, A \rightarrow A - B$
	$A \rightarrow A * B, A \rightarrow A/B$
	$A \rightarrow B + A, A \rightarrow B - A$
	$A \rightarrow B * A, A \rightarrow B/A$

The mutation policy M_θ provides a distribution over possible mutations. Rather than having B be generated completely at random, we parameterize M_θ so that it is $M_\theta : \Omega \times \mathcal{F}$, i.e. the mutations conditions on the dataset $\mathcal{D}$ and on the current expression f. The dependance from $\mathcal{D}$ is akin to the approach in inductive DGSR works, while the dependence on f is novel. Both are passed as inputs to M_θ as a sequence of tokens, f by its prefix notation and $\mathcal{D}$ as in (Kamienny et al., 2022). We use the transformer-based NN architecture from Kamienny et al. (2022) but task the model to decode token-by-token a sequence ω (flattened version of $< A, op, B >$) until a EOS token is reached. A is represented in ω as the index of that node (i.e., $\in [\![1, n]\!]$

for an expression that contains n nodes). While this may allow to output an invalid mutation expression, this happens very rarely in practice as shown in Appendix D, thanks to an efficient pre-training of the policy (described below).

We remark that our mutation distribution is different from those that are commonly used in GP, in that the latter are not conditioned on $\mathcal{D}$ nor parameters (i.e., they are not updated via learning), and they can also shrink the size of an expression or keep it as is, whereas our mutations strictly increase the expression. Note that it is possible to consider mutations that remove and/or replace parts of the expression, but we left exploring this to future work. We also restrict our mutation process to only generate expressions with less than 60 operators and without nesting operations other than the basic arithmetic ones $(+, -, \times, \div)$.

Pre-training of M_θ. Since our mutation policy M_θ is expected to produce mutations for a given expression, and not the final expression directly (as it is the case in the majority of DGSR approaches), it requires a specifically designed pre-training process. To that end, pre-training labeled examples (dataset & ground-truth expression with up to 10 features) are first sampled from a hand-crafted generator (Lample & Charton, 2019) as done in most pre-training NSR approaches (c.f. Appendix A). Next, given a ground-truth expression f^*, we extract a sequence of mutations $[m_l]_{\leq L}$ that iteratively map the empty expression $f^{(0)}$ to the final expression f^*. As illustrated in Figure 1, starting from the ground-truth expressions f^*, we deconstruct f^* by procedurally removing a node (and if the node is binary also one of its child subtree B) from the current f until we get to the empty expression $f^{(0)}$. After reversing this sequence, we obtain a training sequence of expressions and mutations $f^{(0)} \xrightarrow{m_1} f^{(1)} \xrightarrow{m_2} f^{(2)} \xrightarrow{m_3} \ldots \xrightarrow{m_L} f^{(L)} = f^*$ (more details in Appendix A). After tokenization, every mutation m_l serves as target for the pre-training process: M_θ is classically trained as a sequence-to-sequence encoder-decoder, using a cross-entropy loss, to sequentially output each token w_l^e from m_l, given the considered dataset $\mathcal{D}$, the previous expression $f^{(l-1)}$, and the sequence of previous tokens $w_l^{(<e)}$ from the target operator m_l.

4. Experiments

In this section, we present the results of DGSR-MCTS. We begin by studying the performance on test synthetic datasets. Then, we present results on the SRBench datasets.

4.1. Analysis on synthetic datasets

In this sub-section, we consider a set of 1000 unseen synthetic expressions of which half are *in-domain* (exactly same generator described in Section 3 and Appendix A) and half are *out-of-domain* (bigger expressions with up-to

40 operators instead of 25). We provide a set of explorative experiments to bring insights on how different hyper-parameters contribute to the performance, as well as to select a good configuration of hyper-parameters for evaluation on `SRBench`. In what follows, we always select the best expression on a given dataset by evaluating the accuracy of each expression on the training set; as mentioned before, we consider a dataset to be *solved* if the R^2 achieved by the best expression is greater than 0.99. Pre-training was performed on 8 GPUs for a total time of 12 days. We controlled overfitting on the training set of expressions by i) using a sufficiently large training dataset, ii) controlling the cross-entropy loss and prediction accuracy on a held-out validation set of expressions. Each run in this subsection was obtained by MTCS search trials (which fine-tune M_θ and C_ψ) with a time limit of 24 hours, using 4 trainers (1 GPU/CPU each), 4 MCTS workers (1 GPU/CPU each).

Breadth or depth? First, we analyze whether, given a pre-trained M_θ, it is more desirable to explore in breadth or in depth the search tree. The number of samples implicitly influences the breadth/depth trade-off; the larger the number of samples, the more it will be encouraged to explore, whereas when there is little number of samples K, it is forced to go deep. We run a single search trial of 2000 iterations for different values of $K \in \{1, 2, 8, 16, 32\}$.

To compare these different configurations in a fair manner, we make a few design choices that we justify here. First, as shown in earlier work (Dick et al., 2020; Kommenda et al., 2020; Kamienny et al., 2022) and later in this section, optimization of constants contained in the expression can be important to reach high accuracy levels in SR. The deeper in the search tree an expression is, the bigger the expression is, as well as the larger the number of constants it includes, which can add degrees of freedom to the optimization. Because of this, depth can be expected to outperform breadth. For this reason and also because we consider constant optimization in a following ablation, we choose not to optimize constants in this experiment. Secondly, we impose that each configuration considers the same number of expressions. We realize this by allowing only a subset of K expressions to be visited out of the 32 that are sampled during expansion. As shown in Table 3, a choice of K that is between 8 and 16 seems to be the best compromise in both in-domain and out-of-domain datasets, therefore we will sample $K \in [\![8, 16]\!]$ before each expansion in what follows.

Big or small mutation sizes? Secondly, we do ablations on three M_θ models pre-trained on mutation examples generated via different strategies with varying mutation sizes (as defined in Section 3.3); the higher in the expression tree a node is picked (as described in step a) of the caption of Figure 1), the bigger the mutation tends to be. We consider

Table 3. **Percentage of solved datasets for different K**

K	In-Domain	Out-of-domain
1 (greedy)	9.6	0.8
2	10.8	2.4
8	44.6	**19.6**
16	**54.0**	18.4
32	42.8	10.2

`DGSR-MCTS@1` (respectively `DGSR-MCTS@10`), a model pre-trained on mutation sizes 1 (respectively approximately[5] 10), and finally `DGSR-MCTS@`∞, a model trained to output the target expression in a single iteration. Note that `DGSR-MCTS@`∞ is essentially reduces to approaches like those in (Biggio et al., 2021; Kamienny et al., 2022), as M_θ is tasked to predict the entire expression f^* from scratch $f^{(0)}$, while updating M_θ with expert-iteration as described in Section 3.

Interestingly, Table 4 shows mutation size of 10 performs better than size 1 both in-domain and out-of-domain and that the `DGSR-MCTS@`∞ does not generalize to ouf-of-domain datasets, confirming the importance of search.

Table 4. **% of solved datasets for different mutation sizes.**

Model	In-Domain	Out-of-domain
`DGSR-MCTS@1`	52.2	26.8
`DGSR-MCTS@10`	**74.8**	**44.0**
`DGSR-MCTS@`∞	72.4	16.8

How important is constant optimization? As shown in (Kamienny et al., 2022), optimizing constants predicted by a NN model with an optimizer like BFGS greatly improves performance on `SRBench`. Similarly, we study whether including constants optimization is important for `DGSR-MCTS` in the context of search. We remark that while we use constant optimization to compute accuracy, we store the non-optimized expression in the MCTS search tree. We make this choice because optimized constants may be out-of-distribution w.r.t. the pre-trained M_θ, which can lower its performance. As shown in Table 5, optimizing expression constants improves performance substantially. However, constant optimization comes at the price of speed, especially if done after every mutation.

4.2. `SRBench` results

We evaluate `DGSR-MCTS` on the regression datasets of the `SRBench` benchmark (La Cava et al., 2021), in particu-

---

[5]Exact mutation size cannot be guaranteed without special expression tree structures.
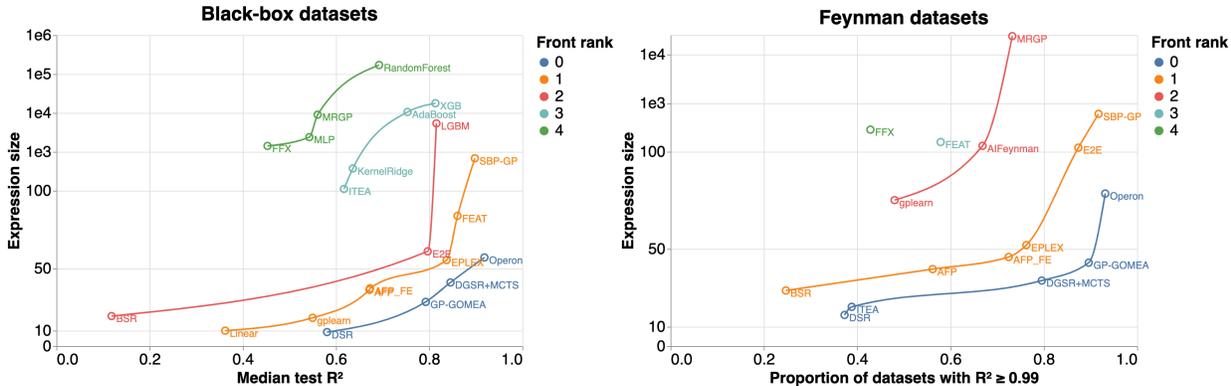
Figure 2. Performance on test splits of `SRBench`, respectively the median R^2 over black box datasets and the proportion of Feynman datasets where the R^2 is larger than 0.99. To nicely visualize the trade-off between accuracy and expression size, we use a linear scale for expression size values up to 100 then a logarithm scale. Note that AI-Feynman (Udrescu & Tegmark, 2020) was removed from the black-box plot for readability (scores $R^2 = -0.6$ and expression size 744).

Table 5. % **of solved datasets for different constant optimization strategies.** We compare constant optimization done: never, only on the best expression of each trial and after each mutation.

Constant optimization	In-Domain	Out-of-domain
Never	74.8	44.0
Only best expression	77.2	59.4
All expressions	**79.6**	**66.2**

lar the *"black-box"* datasets (no ground-truth expression is given) and the *"Feynman"* datasets (conversely, the underlying physics' equation is given). As our approach is trained on datasets with up to 10 features, its use on higher-dimensional datasets requires feature selection. Following (Kamienny et al., 2022), we consider only datasets with at most 10 features so that our results are independent of the quality of a feature selection algorithm. This leads to 57 black-box datasets and 119 Feynman datasets. Each dataset is split into 75% training data and 25% test data using sampling with a random seed (we use 3 seeds per dataset, giving a total of 528 datasets). We consider all baselines provided as part of `SRBench`, which includes GP algorithms, e.g GP-GOMEA (Virgolin et al., 2021), Operon (Burlacu et al., 2020), ITEA (de Franca & Aldeia, 2020), `DGSR` algorithms, DSR (Petersen et al., 2019) and `E2E` (Kamienny et al., 2022) as well as classic machine learning regression models, e.g., multi-layer perceptron (Haykin, 1994) and XGBoost (Chen & Guestrin, 2016).

We run `DGSR-MCTS` with a budget of $500,000$ evaluations (equivalently mutations) and a maximum time limit of 24 hours. `SRBench` imposes to use at most 500000 evaluations per hyper-parameter configuration, and allows for six configurations, yet we provide a single configuration (resulting from Section 4.1); we use the pre-trained M_θ with

mutation size 10, with $K \in [\![8, 16]\!]$ and alternate search trials with and without constants optimization.
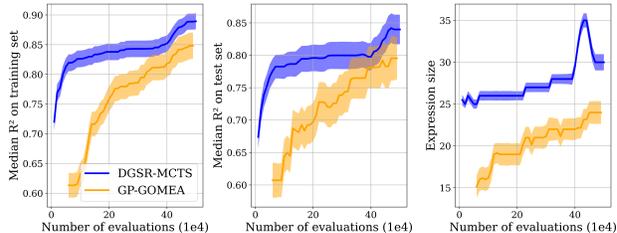
Figure 3. Mean $\pm$ confidence interval performance on the black-box datasets over the number of evaluated expressions for `DGSR-MCTS` and its closest competitor, GP-GOMEA, on the black-box datasets. Thanks to pre-training, `DGSR-MCTS` achieves high-levels of R^2 (and larger expressions) much more quickly than GP-GOMEA. On the training set, `DGSR-MCTS` is consistently superior across the entire search process. On the test set and towards the end of the search, `DGSR-MCTS` and GP-GOMEA achieve similar results, due to a larger generalization gap for larger expressions.

The performance of all SR algorithms is illustrated in Figure 2 along two metrics, accuracy on the test data (as measured by R^2) and expression size computed by counting all operators, variables and constants in an expression, after simplification by SymPy (Meurer et al., 2017). Results are aggregated by taking the average over seeds for each dataset, then the median for black-box datasets and mean for Feynman as done in (La Cava et al., 2021). We visualize the trade-off between accuracy and simplicity of expressions obtained by the different algorithms; the lower and righter the better. We compute front ranks by Pareto-dominance. An algorithm Pareto-dominates another if it is not worse in all metrics and it is strictly better in at least one of them. The definition of ranks is then recursive: an algorithm is at rank

0 if there exists no algorithm that Pareto-dominates it; for successive ranks, an algorithm is at rank i if, when excluding the algorithms up to rank $i - 1$, there exist no algorithm that Pareto-dominates it. `DGSR-MCTS` is placed on the rank 0-front on both black-box and Feynman datasets. GP-GOMEA and `DGSR-MCTS` seem to be the best approaches for achieving simple-yet-accurate models, and interestingly switch place in their trade-off between the two metrics on the black-box and Feynman datasets. We additionally plot the performance over time on the black-box datasets against this baseline in Figure 3. Another interesting point is the difference between `DGSR-MCTS` and `E2E`; `DGSR-MCTS` achieve better test accuracy (0.846 and 0.797 respectively) with less complex expressions (41 and 61 respectively) on the black-box datasets. On 80% (resp. 87% for `E2E`) of Feynman datasets, we achieve $R^2 \geq 0.99$ with expression sizes of 33 (resp. 121).

Ablations. Finally, we present several ablations in Table 6. Namely, we observe whether using synthetic datasets for training C_ψ and fine-tuning M_θ is better than not doing so, and whether our strategy of training `DGSR-MCTS` simultaneously on multiple datasets is better than training iteratively, i.e., one dataset at the time. Our findings suggest that utilizing synthetic datasets has a positive effect on the performance of our model, particularly on the Feynman datasets, which may be attributed to the similarities between the synthetic and Feynman datasets (similar expression sizes, non-noisy observations...) On the black-box dataset (real-world scenarios), training on all datasets simultaneously appears to result in better performance than using synthetic datasets alone, likely due to the sharing of gradients across multiple datasets. Overall, our results indicate that both components play a significant role in the strong performance of `DGSR-MCTS`.

Table 6. **Ablations for different training configurations.** We report the same performance metrics used in Figure 2. Respective expression sizes (not shown here) remain similar.

Use synthetic datasets	Simultaneous training	Black-box	Feynman
no	no	0.801	0.655
yes	no	0.812	0.748
no	yes	0.823	0.689
yes	yes	**0.846**	**0.796**

5. Limitations

Our approach is subject to the known limitations of the Transformer models as in (Kamienny et al., 2022). For instance, learning on large context lengths is challenging and necessitates significant GPU memory resources. This limitation could be circumvented by the use of Transformers specifically designed for large inputs, such as LongFormer (Beltagy et al., 2020). The main source of latency in our algorithm comes from sampling mutations (i.e. forwarding the Transformer model), an aspect shared by other DGSR methods that use large pre-trained transformers. While GP approaches can run on CPU only, GPUs can be useful to batch computations (to generate mutations in our case) for DGSR methods.

6. Conclusion

In this work, we introduced a competitive SR algorithm that address the limits of `DGSR` by incorporating search as a tool to further improve performance of inductive `DGSR` approaches. We positioned our method within a unifying view of SR algorithms that, we hope, will shed light on inner workings of different class of algorithms. We showed `DGSR-MCTS` performs on par with state-of-the-art SR algorithms such as GP-GOMEA in terms of accuracy-complexity trade-off, while being more efficient in terms of number of expression evaluations.

Future work may concern the extension of the proposed approach in a meta-learning framework (Schmidhuber, 1987; Thrun & Pratt, 2012; Finn et al., 2017), where pre-training is performed, either via self-supervised or reinforcement learning, with the objective to reduce the required search budget on a broad family of target datasets.

7. Acknowledgments

References

Ahmad, H. and Helmuth, T. A comparison of semantic-based initialization methods for genetic programming. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, pp. 1878–1881, 2018.

Anthony, T., Tian, Z., and Barber, D. Thinking fast and slow with deep learning and tree search. *Advances in neural information processing systems*, 30, 2017.

Beltagy, I., Peters, M. E., and Cohan, A. Longformer: The long-document transformer. *arXiv preprint arXiv:2004.05150*, 2020.

Biggio, L., Bendinelli, T., Neitz, A., Lucchi, A., and Parascandolo, G. Neural symbolic regression that scales, 2021.

Browne, C. B., Powley, E., Whitehouse, D., Lucas, S. M., Cowling, P. I., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., and Colton, S. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games*, 4(1):1–43, 2012.

Buchberger, B. and Loos, R. Algebraic simplification. In *Computer Algebra*, pp. 11–43. Springer, 1982.

Burlacu, B., Kronberger, G., and Kommenda, M. Operon c++: An efficient genetic programming framework for symbolic regression. In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion*, GECCO '20, pp. 1562–1570, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450371278. doi: 10.1145/3377929.3398099. URL https://doi.org/10.1145/3377929.3398099.

Chen, Q., Zhang, M., and Xue, B. Structural risk minimization-driven genetic programming for enhancing generalization in symbolic regression. *IEEE Transactions on Evolutionary Computation*, 23(4):703–717, 2018.

Chen, T. and Guestrin, C. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, pp. 785–794, 2016.

Cranmer, M., Sanchez Gonzalez, A., Battaglia, P., Xu, R., Cranmer, K., Spergel, D., and Ho, S. Discovering symbolic models from deep learning with inductive biases. *Advances in Neural Information Processing Systems*, 33:17429–17442, 2020.

de Franca, F. O. and Aldeia, G. S. I. Interaction-Transformation Evolutionary Algorithm for Symbolic Regression. *Evolutionary Computation*, pp. 1–25, 12 2020. ISSN 1063-6560. doi: 10.1162/evco_a_00285. URL https://doi.org/10.1162/evco_a_00285.

Dick, G., Owen, C. A., and Whigham, P. A. Feature standardisation and coefficient optimisation for effective symbolic regression. In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference*, pp. 306–314, 2020.

Fawzi, A., Balog, M., Huang, A., Hubert, T., Romera-Paredes, B., Barekatain, M., Novikov, A., R Ruiz, F. J., Schrittwieser, J., Swirszcz, G., et al. Discovering faster matrix multiplication algorithms with reinforcement learning. *Nature*, 610(7930):47–53, 2022.

Finn, C., Abbeel, P., and Levine, S. Model-agnostic meta-learning for fast adaptation of deep networks. In *International conference on machine learning*, pp. 1126–1135. PMLR, 2017.

Harik, G. et al. Linkage learning via probabilistic modeling in the ECGA. *IlliGAL report*, 99010, 1999.

Haykin, S. *Neural networks: a comprehensive foundation*. Prentice Hall PTR, 1994.

Hemberg, E., Veeramachaneni, K., McDermott, J., Berzan, C., and O'Reilly, U.-M. An investigation of local patterns for estimation of distribution genetic programming. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pp. 767–774, 2012.

Hernandez, A., Balasubramanian, A., Yuan, F., Mason, S. A., and Mueller, T. Fast, accurate, and transferable many-body interatomic potentials by symbolic regression. *npj Computational Materials*, 5(1):1–11, 2019.

Kabliman, E., Kolody, A. H., Kronsteiner, J., Kommenda, M., and Kronberger, G. Application of symbolic regression for constitutive modeling of plastic deformation. *Applications in Engineering Science*, 6:100052, 2021.

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

Kim, K., Shan, Y., Nguyen, X. H., and McKay, R. I. Probabilistic model building in genetic programming: A critical review. *Genetic Programming and Evolvable Machines*, 15(2):115–167, 2014.

Kommenda, M., Beham, A., Affenzeller, M., and Kronberger, G. Complexity measures for multi-objective symbolic regression. In *Computer Aided Systems Theory–EUROCAST 2015: 15th International Conference, Las Palmas de Gran Canaria, Spain, February 8-13, 2015, Revised Selected Papers 15*, pp. 409–416. Springer, 2015.

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

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

La Cava, W., Orzechowski, P., Burlacu, B., de Franca, F. O., Virgolin, M., Jin, Y., Kommenda, M., and Moore, J. H. Contemporary symbolic regression methods and their relative performance. *arXiv preprint arXiv:2107.14351*, 2021.

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

Lample, G., Lachaux, M.-A., Lavril, T., Martinet, X., Hayat, A., Ebner, G., Rodriguez, A., and Lacroix, T. Hypertree proof search for neural theorem proving. *arXiv preprint arXiv:2205.11491*, 2022.

Landajuela, M., Lee, C., Yang, J., Glatt, R., Santiago, C. P., Aravena, I., Mundhenk, T. N., Mulcahy, G., and Petersen, B. K. A unified framework for deep symbolic regression. In *Advances in Neural Information Processing Systems*, 2022.

Li, L., Fan, M., Singh, R., and Riley, P. Neural-guided symbolic regression with asymptotic constraints. *arXiv preprint arXiv:1901.07714*, 2019.

Looks, M. On the behavioral diversity of random programs. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, pp. 1636–1642, 2007.

Lu, Q., Tao, F., Zhou, S., and Wang, Z. Incorporating actor-critic in monte carlo tree search for symbolic regression. *Neural Computing and Applications*, 33(14):8495–8511, 2021.

Ma, H., Narayanaswamy, A., Riley, P., and Li, L. Evolving symbolic density functionals. *arXiv preprint arXiv:2203.02540*, 2022.

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. Sympy: symbolic computing in python. *PeerJ Computer Science*, 3:e103, 2017.

Mundhenk, T. N., Landajuela, M., Glatt, R., Santiago, C. P., Faissol, D. M., and Petersen, B. K. Symbolic regression via neural-guided genetic programming population seeding. *arXiv preprint arXiv:2111.00053*, 2021.

Pawlak, T. P. and Krawiec, K. Semantic geometric initialization. In *Genetic Programming: 19th European Conference, EuroGP 2016, Porto, Portugal, March 30-April 1, 2016, Proceedings 19*, pp. 261–277. Springer, 2016.

Petersen, B. K., Larma, M. L., Mundhenk, T. N., Santiago, C. P., Kim, S. K., and Kim, J. T. Deep symbolic regression: Recovering mathematical expressions from data via risk-seeking policy gradients. *arXiv preprint arXiv:1912.04871*, 2019.

Poli, R., Langdon, W. B., and McPhee, N. F. A field guide to genetic programming, 2008.

Polu, S. and Sutskever, I. Generative language modeling for automated theorem proving. *arXiv preprint arXiv:2009.03393*, 2020.

Salustowicz, R. and Schmidhuber, J. Probabilistic incremental program evolution. *Evolutionary computation*, 5 (2):123–141, 1997.

Schmidhuber, J. Evolutionary principles in self-referential learning. on learning now to learn: The meta-meta-meta...-hook. Diploma thesis, Technische Universitat Munchen, Germany, 14 May 1987. URL http://www.idsia.ch/~juergen/diploma.html.

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

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.

Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., et al. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144, 2018.

Sun, F., Liu, Y., Wang, J.-X., and Sun, H. Symbolic physics learner: Discovering governing equations via monte carlo tree search. *arXiv preprint arXiv:2205.13134*, 2022.

Sutton, R. S. *Temporal credit assignment in reinforcement learning*. University of Massachusetts Amherst, 1984.

Thrun, S. and Pratt, L. *Learning to learn*. Springer Science & Business Media, 2012.

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

Vaddireddy, H., Rasheed, A., Staples, A. E., and San, O. Feature engineering and symbolic regression methods for detecting hidden physics from sparse sensor observation data. *Physics of Fluids*, 32(1):015113, 2020.

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

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. Attention is all you need. In *Advances in neural information processing systems*, pp. 5998–6008, 2017.

Vinyals, O., Babuschkin, I., Czarnecki, W. M., Mathieu, M., Dudzik, A., Chung, J., Choi, D. H., Powell, R., Ewalds, T., Georgiev, P., et al. Grandmaster level in starcraft ii using multi-agent reinforcement learning. *Nature*, 575 (7782):350–354, 2019.

Virgolin, M. and Pissis, S. P. Symbolic regression is NP-hard. *Transactions on Machine Learning Research*, 2022. URL `https://openreview.net/forum?id=LTiaPxqe2e`.

Virgolin, M., De Lorenzo, A., Medvet, E., and Randone, F. Learning a formula of interpretability to learn interpretable formulas. In *Parallel Problem Solving from Nature–PPSN XVI: 16th International Conference, PPSN 2020, Leiden, The Netherlands, September 5-9, 2020, Proceedings, Part II 16*, pp. 79–93. Springer, 2020.

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

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

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

White, D. R., Yoo, S., and Singer, J. The programming game: evaluating mcts as an alternative to gp for symbolic regression. In *Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation*, pp. 1521–1522, 2015.

A. Data generation

Ground-truth expressions. To generate a large synthetic dataset with examples $(\mathcal{D}, f^*)$, we first sample N observations/features $\mathbf{X} \in \mathrm{R}^{N \times D}$ where D is uniformly sampled between 1 and 10 with a mixture of Gaussians as in (Kamienny et al., 2022), then consider sampling: a) an empty unary-binary tree from (Lample & Charton, 2019) generator with between 5 and 25 internal nodes, b) assign a random operator on nodes and either a variable $\{x_d\}_{d \leq D}$ or float constant drawn from a normal distribution on leaves. We then simplify the ground-truth expression with SymPy.

The only difference with (Kamienny et al., 2022) lies in the fact that the generated expressions are much smaller by not enforcing all variables to appear sampled expressions, therefore letting the model learn the ability to do feature selection, as well as to having much less constants (they apply linear transformations with probability 0.5 on all nodes/leaves), therefore providing more interpretable expressions (model size is divided by 2).

Example mutations. From a ground-truth expression f^*, we generate a sequence of example mutations that go from the empty expression to f^* by iteratively removing parts of the expression tree. To do so, at any given step, we randomly pick an internal node such that the size of the subtree argument B is large enough, and apply the backward mutation, i.e. adding an edge between the parent node and the remaining child node A. When the remaining expression is too small, the mutation's operator becomes $0 \rightarrow B$ where B is the remaining expression.

B. Data representation

As done in most (Kamienny et al., 2022), floats are represented in base 10 foating-point notation, rounded to four significant digits, and encoded as sequences of 3 tokens: their sign, mantissa (between 0 and 9999), and exponent (from `E-100` to `E100`). Expressions are represented by their Polish notations, i.e. the breadth-first search walk, with numerical constants are represented as explained above. A dataset is represented by the concatenation of all tokenized (x_i, y_i) pairs where vectors representation is just the flattenized tokens of each dimension value. The combination of both the expression and dataset yields the representation of states by concatenating both representations and using special separators between the expression f and dataset $\mathcal{D}$. Actions are represented by the concatenation (with special separators) of i) the node index (integer in base 10) on which to apply the mutation, ii) the operator token, iii) the tokenized expression B if the operator is binary.

C. Model details

Since the number of tokens transformers can use as context is limited by memory considerations and possible learnable long-range dependencies, we restrict to 100 the number of input data points used as input to M, the subset being sampled at each expansion. We also train our model on datasets with at most 10 variables, as in (Kamienny et al., 2022).

D. Exceptions detected ablations

Earlier `DGSR` work has showed that Transformer models (Vaswani et al., 2017) were able to learn almost perfect semantics of symbolic expressions, resulting in 99% of valid expressions in (Kamienny et al., 2022). In this work, we noticed that our policy model was also able to manipulate the expression structure quite well as around 90% of sampled mutations resulted in valid expressions. Similarly we noticed that malformed mutations, e.g., invalid index on which expression node to apply an operation, argument B that cannot be parsed or or absent B when operator is binary, represent less than 1% of errors.

As mentioned in Section 3.3, we constrain our model to discard expressions that are too complex (more than 60 operators/variables/constants) or have nested complicated operators (e.g. $\cos(\cos(X))$, $\log(\log(X))$), in order to promote simpler expressions as explained, resulting in a great trade-off between accuracy and complexity as shown in Section 4.2. Note that it would be possible to enforce a greater trade-off between accuracy and complexity, e.g. using strategies mentioned in (La Cava et al., 2021). This results in 9% of expressions being discarded because these constraints are violated.

E. Search details

E.1. Definition of satisfactory expressions

We concluded from preliminary experiments that considering f^* to be satisfactory if $R^2 \geq 0.99$ performed best as it provided high-quality samples while dramatically reducing search times compared to perfect fitting. Systematically estimating what accuracy can be achieved with a given complexity is not possible without, e.g., resorting to another algorithm that operates under the same complexity constraints and can act as an oracle. We also tried estimating accuracy on validation set of points by running XGBoost on a train set, however for `SRBench` datasets, accuracy can greatly vary according to the way the dataset is split.

E.2. Constant optimization

We use Broyden–Fletcher–Goldfarb–Shanno algorithm (BFGS) with batch size 256, early stopping if accuracy does

not improve after 10 iterations and a timeout of 1 second.

E.3. Search hyper-parameters

In this work, we employ a distributed learning architecture, similar to that proposed in (Lample et al., 2022). Since the optimal hyper-parameters of search are not necessarily the same for all datasets, the controller samples these hyper-parameters from pre-defined ranges for each different search trial:

The proposed model depends on many of hyper-parameters, specifically those pertaining to the decoding of the mutation model and the search process. Determining the optimal values for these hyper-parameters poses a significant challenge in practice for several reasons. Firstly, the model is in a state of continual evolution, and thus the optimal hyper-parameter values may also change over time. For instance, if the model exhibits an excessive level of confidence in its predictions, it may be necessary to increase the decoding temperature to promote diversity in mutations. Secondly, the optimal values of the hyper-parameters may be specific to the dataset under consideration. Finally, the sheer number of hyper-parameters to be tuned, coupled with the high computational cost of each experiment, renders the task of determining optimal values infeasible. To circumvent these issues, rather than fixing the hyper-parameters to a specific value, they are sampled from predefined ranges at the beginning of each search trial. The specific decoding parameters and the distribution utilized are as follows:

- Number of samples K per expansion. Distribution: uniform on range [8,16].

- Temperature used for decoding. Distribution: uniform on range [0.5, 1.0].

- Length penalty: length penalty used for decoding. Distribution: uniform on range [0, 1.2].

- Depth penalty: an exponential value decay during the backup-phase, decaying with depth to favor breadth or depth. Distribution: uniform on discrete values [0.8, 0.9, 0.95, 1].

- Exploration: the exploration constant p_{uct}. 1