DEEP GENERATIVE SYMBOLIC REGRESSION WITH MONTE-CARLO-TREE-SEARCH

Anonymous authors
Paper under double-blind review

ABSTRACT

Symbolic regression (SR) is the problem of learning a symbolic expression from numerical data. Recently, deep neural models showed competitive performance compared to more classical Genetic Programming (GP) ones. Unlike their GP counterparts, they 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 results in low results 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) or physics (Schmidt & Lipson, 2009; Vaddireddy et al., 2020; Sun et al., 2022). Unfortunately, SR is intractable, because it entails composing inherently discrete objects (i.e., expression terms) to form a best-fitting expression (Virgolin & Pissis, 2022). Thus, typical approaches are heuristic. A famous example is Genetic Programming (GP), which searches by *evolving* random expressions (Poli et al., 2008).

Lately, there has been a growing interest in the SR community for approaches based upon neural networks (NNs). For example, Udrescu & Tegmark (2020) use 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`), use NNs to learn to generate expressions. Deep Symbolic Regression (DSR) by Petersen et al. (2019) uses an NN to produce expressions with policy gradients and, similar to GP, faces a *tabula rasa* setup of the problem: for any new dataset, DSR starts from scratch. On the other hand, other `DGSR` approaches exist that 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). This type of `DGSR` 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.

In this paper, we seek to overcome the limitations of inductive `DGSR`, by proposing a synergistic combination with Monte-Carlo Tree Search (MCTS) to build a search tree of expressions. A *mutation* policy, parameterized similarly to `DGSR`, is responsible for producing mutations that expand expressions from the search tree. 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 evaluate our method on the `SRBench` benchmark (La Cava et al.,

2021) and show that it achieves state-of-the-art performance by discovering expressions that are both simple and accurate.

2 Method

In this work, we see the learning process as building a tree, that we call *search tree*. Each node of the search tree corresponds to an expression f, and edges between nodes represent mutations. We present here the two ingredients of our method, the mutation policy respectively in charge of expanding expressions and the MCTS policy in charge of selecting which expression to mutate.

2.1 Mutation Policy

Our search process relies on a mutation policy M_θ, a distribution over promising mutations of a given expression $f \in \mathcal{F}$. Note that $\mathcal{F}$ is the space of possible expressions, which in SR is implicitly defined by the choice of expression terms to be combined (e.g., $+, -, \times, \div, \sin, \cos, \dots, x_1, x_2, \dots$). We represent expressions with trees (not to be confused with the search tree of MCTS), see Figure 1.

Definition of M_θ. We define mutations as transformations that can be applied on any node of a given expression tree. Table 1 contains the list of considered transformations, where A stands for an existing node of the expression tree and B stands for a new expression to be included in the tree. The $\rightarrow$ symbol stands as 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 mutation is a triplet $< A, op, B >$, where A is a node index from the considered expression (i.e., $A \in [\![0, k]\!]$ for an expression that contains k nodes), op is an operation from Table 1 and B is a new (sub-)expression that needs to be generated. A constant optimization step, detailed in Appendix F, can happen after the mutation. We call *mutation size* the size of the expression B.

The mutation policy M_θ provides a distribution over possible mutations. Rather than constraining B to be generated randomly, we parameterize $M_\theta(\mathcal{D}, f, \theta)$ following the transformer-based approach by Kamienny et al. (2022), conditioning on an additional input $f \in \mathcal{F}$, to decode a tokenized version of $< A, op, B >$ where A is expressed by the index on the expression tree where the mutation is applied. While this may allow to output an invalid mutation expression, this happens very rarely in practice as shown in Appendix E, thanks to an effective pre-training of the policy (described below). Our mutation distribution is different from those that are commonly used in GP-based approaches, in that the latter are not conditioned on $\mathcal{D}$ nor parameters (i.e., they are not updated via learning), and they can also make expressions smaller, whereas ours strictly enlarge them. This way, in our MCTS approach (Section 2.2), searching in depth means exploring larger expressions.

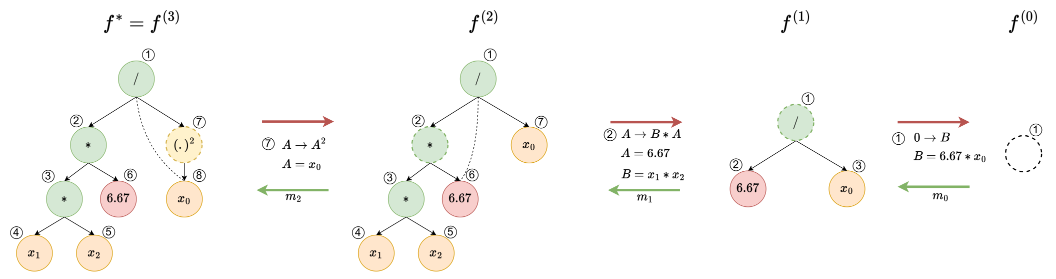

Figure 1: Given a starting ground-truth expression as a tree, we procedurally dismantle the tree until no node is left. This is done by, at each step (red arrows), a) picking a node (dashed contour), b) removing the picked node and, if the operator is binary, additionally remove the subtree rooted in 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. Then, we train the mutation model to assemble the tree back via subsequent mutations (green arrows), which revert the dismantling process.

Pre-training of M_θ. Our mutation policy M_θ learns to produce intermediate mutations (in order to work within a search algorithm) and not the final expression in one shot like existing `DGSR` approaches. Thus, M_θ requires a novel pre-training process. To that end, pre-training data is first generated from a simulator as labeled examples (dataset & ground-truth expression with up to 10 features), similarly to Biggio et al. (2021); Kamienny et al. (2022). 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 expression f^*, we deconstruct f^* by procedurally removing a node (if the node is binary, also a child subtree B) from the current f until we get to $f^{(0)}$. After reversing this sequence, we obtain a training sequence of expressions and mutations $f^{(0)} \xrightarrow{m_1} f^{(1)} \xrightarrow{m_2} f^{(2)} \xrightarrow{m_3} \dots \xrightarrow{m_L} f^{(L)} = f^*$. After tokenization, given inputs $f^{(l-1)}$ and $\mathcal{D}$, mutation m_l serves as target for the pre-training process. We use a transformer network trained as a sequence-to-sequence encoder-decoder, using a cross-entropy loss and teacher-forcing as done by Kamienny et al. (2022).

2.2 MCTS Sampling

Following Lample et al. (2022), our MCTS is made of three steps, that are iteratively repeated until a given budget of iterations is exhausted: 1) Selection, 2) Expansion, 3) Back-propagation (explained below). Additionally, our MCTS uses a critic network C_ψ. This NN shares the same structure of M_θ, except for having an additional head that is trained during the search to learn to prioritize promising expressions to mutate.

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

$$V(f') + p_{uct}E(f')M_\theta(f'|f, \mathcal{D}, \theta) \tag{1}$$

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 from the nodes that descend from f in the tree search.

Expansion Once the selection step reaches a leaf f of the tree, this leaf is expanded to produce child nodes by applying K mutations $\{m_k\}_{k\leq K}$ sampled from $M_\theta(\mathcal{D}, f, \theta)$ to f. This leads to modified expressions $\{f'_k\}_{k\leq K}$. 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 of these new nodes f are evaluated to check whether they reach a sufficient R^2 accuracy on the dataset. 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. Nodes that solve the problem obtain a value $v(f) = 1$. Others obtain an estimate from the critic network: $v(f) = C_\psi(\mathcal{D}, f)$.

Back-Propagation After each expansion, the value of every ancestor f' of any new node f is updated as $V(f') \leftarrow V(f') + v(f)$. If at a given expansion, there exists a satisfactory expression f^* (e.g. a terminal node), we extract the sequence of $(f^{(\tau)}, m_t)$ leading to f^*. If multiple such sequences exist, we select the one with the smallest number of mutations. Then, we early-stop the process after back-propagating the corresponding values. Every 1000 iterations, we empty the search tree and explore with the now-updated model M_θ and C_ψ.

2.3 Learning Critic C_ψ and fine-tuning M_θ

After each search trial, we aim to update C_ψ and M_θ distribution. To that end, training samples from the search process are stored in two separate first-in-first-out queues: a buffer stores mutation sequences that produced an expression with $R^2 \geq 0.99$ for the finetuning of M_θ, the other contains values of nodes. For the latter, ancestors of f^*, an expression $R^2 \geq 0.99$, are assigned a score of 1.0. Given a value, nodes 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.

We launch MCTS simultaneously on multiple datasets to allow potentially-beneficial transfer learning between datasets. To avoid catastrophic forgetting that could result from sub-optimal mutations by the policy model, we also continue training on pre-training examples from synthetic datasets, in the spirit of AlphaStar (Vinyals et al., 2019) or HTPS (Lample et al., 2022).

3 Experiments

We evaluate our method `DGSR-MCTS` on the regression datasets of `SRBench` benchmark (La Cava et al., 2021), in particular the *"black-box"* datasets (no ground-truth expression is given) and the *"Feynman"* datasets (conversely, the underlying physics' equation is given). As M_θ 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) or `XGBoost` (Chen & Guestrin, 2016).

We run `DGSR-MCTS` with a budget of $500,000$ evaluations and a maximum time limit of 24 hours. `SRBench` imposes to use at most $500,000$ evaluations *per hyper-parameter configuration*, and allows for six configurations, yet we provide a single configuration. We alternate search trials with and without constants fine-tuning.

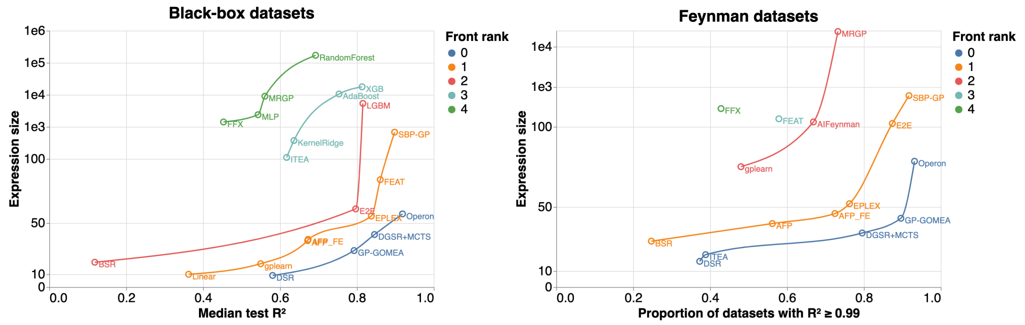

Figure 2: Performance on test splits of `SRBench`, respectively the median R^2 over black box datasets and the proportion of Feynman datasets where the R^2 is larger than 0.99. To nicely visualize the trade-off between accuracy and expression size, we use a linear scale for expression size values up 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).

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 by La Cava et al. (2021). We visualize the trade-off between accuracy and simplicity of expressions obtained by the different algorithms and compute front ranks by Pareto-dominance as defined in Appendix D; the lower and righter the better. `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.

4 Conclusion

In this work, we introduced a competitive SR algorithm that addresses the limits of `DGSR` by incorporating search as a tool to further improve performance. We showed `DGSR-MCTS` performs on par with the state-of-the-art SR algorithm GP-GOMEA in terms of accuracy-complexity trade-off.

REFERENCES

Luca Biggio, Tommaso Bendinelli, Alexander Neitz, Aurelien Lucchi, and Giambattista Parascandolo. Neural symbolic regression that scales, 2021.

Bogdan Burlacu, Gabriel Kronberger, and Michael Kommenda. 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`.

Tianqi Chen and Carlos Guestrin. 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.

F. O. de Franca and G. S. I. Aldeia. 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`.

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

Evgeniya Kabliman, Ana Helena Kolody, Johannes Kronsteiner, Michael Kommenda, and Gabriel Kronberger. Application of symbolic regression for constitutive modeling of plastic deformation. *Applications in Engineering Science*, 6:100052, 2021.

Pierre-Alexandre Kamienny, Stéphane d'Ascoli, Guillaume Lample, and François Charton. End-to-end symbolic regression with transformers. *arXiv preprint arXiv:2204.10532*, 2022.

William La Cava, Patryk Orzechowski, Bogdan Burlacu, Fabricio Olivetti de Franca, Marco Virgolin, Ying Jin, Michael Kommenda, and Jason H Moore. Contemporary symbolic regression methods and their relative performance. *arXiv preprint arXiv:2107.14351*, 2021.

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

Guillaume Lample, Marie-Anne Lachaux, Thibaut Lavril, Xavier Martinet, Amaury Hayat, Gabriel Ebner, Aurélien Rodriguez, and Timothée Lacroix. Hypertree proof search for neural theorem proving. *arXiv preprint arXiv:2205.11491*, 2022.

Aaron Meurer, Christopher P Smith, Mateusz Paprocki, Ondřej Čertík, Sergey B Kirpichev, Matthew Rocklin, AMiT Kumar, Sergiu Ivanov, Jason K Moore, Sartaj Singh, et al. Sympy: symbolic computing in python. *PeerJ Computer Science*, 3:e103, 2017.

Brenden K Petersen, Mikel Landajuela Larma, T Nathan Mundhenk, Claudio P Santiago, Soo K Kim, and Joanne T Kim. Deep symbolic regression: Recovering mathematical expressions from data via risk-seeking policy gradients. *arXiv preprint arXiv:1912.04871*, 2019.

Ricardo Poli, William B Langdon, and Nicholas F McPhee. A field guide to genetic programming, 2008.

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

David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.

Fangzheng Sun, Yang Liu, Jian-Xun Wang, and Hao Sun. Symbolic physics learner: Discovering governing equations via monte carlo tree search. *arXiv preprint arXiv:2205.13134*, 2022.

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

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

Mojtaba Valipour, Bowen You, Maysum Panju, and Ali Ghodsi. Symbolicgpt: A generative transformer model for symbolic regression. *arXiv preprint arXiv:2106.14131*, 2021.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pp. 5998–6008, 2017.

Oriol Vinyals, Igor Babuschkin, Wojciech M Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H Choi, Richard Powell, Timo Ewalds, Petko Georgiev, et al. Grandmaster level in starcraft ii using multi-agent reinforcement learning. *Nature*, 575(7782):350–354, 2019.

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

Marco Virgolin, Tanja Alderliesten, Cees Witteveen, and Peter AN Bosman. Improving model-based genetic programming for symbolic regression of small expressions. *Evolutionary computation*, 29 (2):211–237, 2021.

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

A Data generation

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).

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 List of operators

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

D Front rank

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.

E 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 2, we make certain choices to restrict what kind of expressions can be generated (see Section 2.1). 9% of expressions are discarded because these constraints are violated.

F Constant optimization

As done in Kamienny et al. (2022), constants predicted by the model can be given as a initial values to an external optimization algorithm. In practice, we use Broyden–Fletcher–Goldfarb–Shanno algorithm (BFGS) with batch size 256 and a timeout of 1 second.

G 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).

H 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