Efficient generator of mathematical expressions for symbolic regression

Sebastian Mežnar[1,2] · **Sašo Džeroski[1]** · **Ljupčo Todorovski[1,3]**

Abstract

We propose an approach to symbolic regression based on a novel variational autoencoder for generating hierarchical structures, HVAE. It combines simple atomic units with shared weights to recursively encode and decode the individual nodes in the hierarchy. Encoding is performed bottom-up and decoding top-down. We empirically show that HVAE can be trained efficiently with small corpora of mathematical expressions and can accurately encode expressions into a smooth low-dimensional latent space. The latter can be efficiently explored with various optimization methods to address the task of symbolic regression. Indeed, random search through the latent space of HVAE performs better than random search through expressions generated by manually crafted probabilistic grammars for mathematical expressions. Finally, EDHiE system for symbolic regression, which applies an evolutionary algorithm to the latent space of HVAE, reconstructs equations from a standard symbolic regression benchmark better than a state-of-the-art system based on a similar combination of deep learning and evolutionary algorithms.

Keywords Symbolic regression · Equation discovery · Generative models · Variational autoencoders · Evolutionary algorithms

✉ Sebastian Mežnar
  sebastian.meznar@ijs.si

  Sašo Džeroski
  saso.dzeroski@ijs.si

  Ljupčo Todorovski
  ljupco.todorovski@fmf.uni-lj.si

[1] Department of Knowledge Technologies, Jožef Stefan Institute, Jamova cesta 39, 1000 Ljubljana, Slovenia

[2] Jožef Stefan International Postgraduate School, Jamova cesta 39, 1000 Ljubljana, Slovenia

[3] Faculty of Mathematics and Physics, University of Ljubljana, Jadranska 21, 1000 Ljubljana, Slovenia

1 Introduction

Symbolic regression (also known as equation discovery) aims at discovering closed-form equations in collections of measured data (Schmidt & Lipson, 2009; Todorovski, 2017). Methods for symbolic regression explore the vast space of candidate equations to find those that fit the given data well. They often employ modeling knowledge from the domain of use to constrain the search space of candidate equations. The knowledge is usually formalized into grammars (Brence et al., 2021) or libraries of model components, such as entities and processes (Bridewell et al., 2008). Knowledge-based equation discovery methods have successfully solved practical modeling problems in various domains (Radinja et al., 2021; Simidjievski et al., 2020).

Grammars and libraries of model components are used to generate candidate expressions that might appear in the discovered equations. However, they must be manually crafted, which is a severe obstacle to their broader use. The central aim of this article is to develop a novel generative model of mathematical expressions that can be used for efficient symbolic regression. The model can be trained from a corpus of mathematical expressions from the domain of interest, thus automatically tailoring the space of candidate equations to the application at hand. The developed generative model must have two essential properties to be applicable in such a scenario. First, it should be trainable from a small number of mathematical expressions, e.g., collected from a textbook or from scientific literature in the application domain. Second, the model should encode the expressions in a low-dimensional latent space. The latter space can then be efficiently explored by optimization methods to solve the task of symbolic regression. Lowering the dimensionality of the latent space will significantly increase the efficiency of symbolic regression.

Recently, several variational autoencoders (VAEs) have been shown to be efficient generative models. CVAE (Gómez-Bombarelli et al., 2018) employs a VAE based on recurrent neural networks to encode discrete expressions into a continuous latent space and then decode points from the latent space back into discrete mathematical expressions. This decoder can be used to generate expressions. However, CVAE still generates invalid sequences and requires extensive training data to reduce the likelihood of generating invalid expressions (Kusner & Hernández-Lobato, 2016). The grammar variational autoencoder, GVAE (Kusner et al., 2017), and its successor, SD-VAE (Dai et al., 2018), employ a context-free grammar to ensure the syntactic validity of the generated expressions. Instead of directly training models on sequences, they model the distribution of parse trees that are produced by the grammar while deriving syntactically (and, in the case of SD-VAE, semantically) valid expressions.

We claim that grammars are an unnecessarily powerful and too general formalism for generating mathematical expressions. Grammars add syntactic categories to the expression symbols rendering the parse trees, i.e., the structures modeled with the autoencoder, more complex than the original sequences. This overhead on training expressions inevitably translates to a requirement for more extensive training data and a latent space with larger dimensionality, reducing the efficiency of optimization methods for symbolic regression operating in that latent space.

We propose a novel variational autoencoder for hierarchical data objects, HVAE, to address these issues. It builds upon the ideas of variational autoencoders for hierarchical data (Jin et al., 2021) and gated recursive convolutional neural networks (Cho et al., 2014a). HVAE combines simple atomic units with shared weights to encode and decode the individual nodes in the hierarchy. The atomic units are extensions of the standard

Step 1: Train a generative model
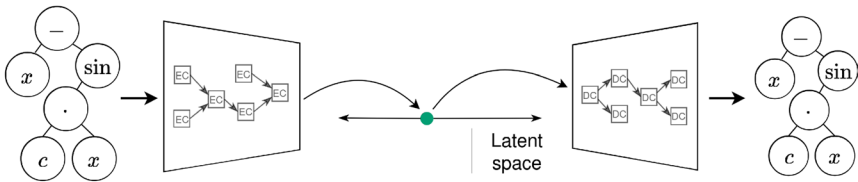
Step 2: Explore the latent space with EA
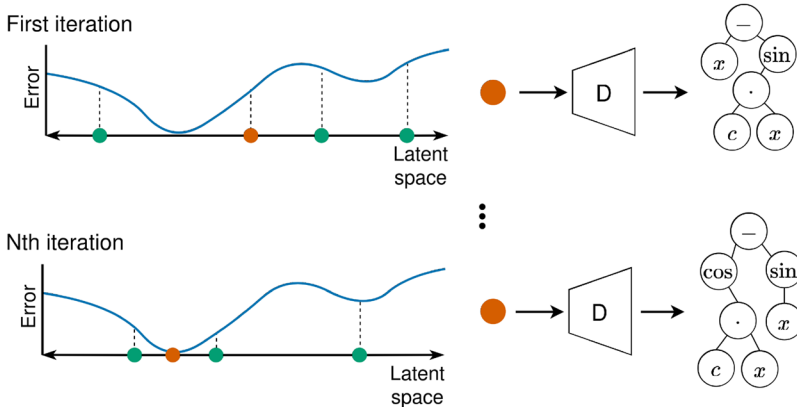
Fig. 1 A schematic representation of the EDHiE approach. In the first step, we train a HVAE model. In the second step, we explore the latent space of the HVAE model with an evolutionary algorithm. The red dot represents the best expression in a given iteration (Color figure online)

gated recurrent unit (GRU) cells. The encoding units are stacked into a tree that follows the hierarchy of the training object, and they encode the hierarchy bottom-up, compiling the codes of the descendants to encode the ancestor nodes. The decoding units proceed top-down and use the decoded symbols of the ancestor nodes to decide upon the need to extend the hierarchy with descendant nodes. We claim that HVAE can be efficiently trained to generate valid mathematical expressions from a training set of modest size, while using a low-dimensional latent space.

We exploit these expected properties of our HVAE to implement a novel approach for symbolic regression, EDHiE. It performs an evolutionary search through the latent space of a HVAE trained on mathematical expression trees as shown in Fig. 1. The genetic operations utilize the HVAE encoder to obtain the expressions' latent codes, generate new individuals with crossover and mutation in the latent space, and decode the latter back to mathematical expressions. EDHiE can then evaluate the fit of the obtained expressions against the measurements. We conjecture that the performance of EDHiE on standard benchmarks (Udrescu & Tegmark, 2020; Uy et al., 2011) would compare favorably to that of a state-of-the-art symbolic regression methods (Mundhenk et al., 2021). The results of our empirical evaluation of HVAE and EDHiE confirm our conjectures. HVAE can achieve better reconstruction of the training expressions with order-of-magnitude fewer training examples while using latent spaces with

fewer dimensions. EDHiE outperforms alternative methods for symbolic regression on the task of reconstructing the ten equations in the Ngyuen benchmark.

We can summarize the contributions of this work as follows:

- We propose HVAE, a variational autoencoder for hierarchical data, that can be efficiently trained to generate mathematical expressions from modest amounts of data, while using a low-dimensional latent space.
- We introduce EDHiE, a symbolic regression approach that exploits HVAE to efficiently search through the space of candidate equations.

The remainder of the paper is organized as follows. Section 2 reviews related work on generative models and symbolic regression. We introduce the hierarchical variational autoencoder HVAE and the symbolic regression approach EDHiE in Sect 3. Section 4 presents the results of the empirical evaluation of HVAE and EDHiE. Finally, Section 5 summarizes and discusses the contributions of the presented work and outlines directions for further research.

2 Related work

Most of the early successful applications of generative models have been in the domains of text, speech, images, and video, i.e., they have been mainly used for generating unstructured data objects composed of continuous data elements. The discrete data structures that generative models have most often tackled are strings or sequences of characters, where the data elements are discrete symbols. The models that use strings as input (and output) usually do so by training a recurrent neural network (Sherstinsky, 2020), most commonly using seq2seq autoencoders (Sutskever et al., 2014).

A major problem of sequence-to-sequence autoencoders is that they do not guarantee the syntactic correctness of the generated expressions. One way to solve this problem is to learn an additional validation model for checking the correctness of the generated sequence (Janz et al., 2017). Grammar variational autoencoders (GVAE) (Kusner et al., 2017) use context-free grammars for specifying the space of valid structured data objects. Each data object can be then represented as a sequence of grammar productions (rewrite rules) that derives it. In turn, GVAE encode sequences of rewrite rules that derive objects instead of the objects themselves. The structure of the decoder is constrained to generate valid sequences of rewrite rules that are then used together with the grammar to generate valid expressions.

Dai et al. (2018) propose the use of attribute grammars, i.e., context-free grammars that attach attributes to the grammar's syntactic categories. By prescribing properties and relationships between the attributes, such grammars can also encode semantic constraints on the derived data objects. The attribute grammars, together with SD-VAE, i.e., syntax-directed VAE, can generate expressions that are consistent with a set of both syntactic and semantic constraints. Alternative generalizations of grammars have been used for generative modeling of program source code in high-level languages (Bielik et al., 2016).

Most of the above approaches can also generate mathematical expressions. However, they need the complex formalism of grammars to generate more complex data structures, most often molecular structures (Dai et al., 2018; Gómez-Bombarelli et al., 2018). Since

mathematical expressions can be represented as simpler structures, i.e., binary trees, our work concerns generative models for hierarchical (tree-structured) data.
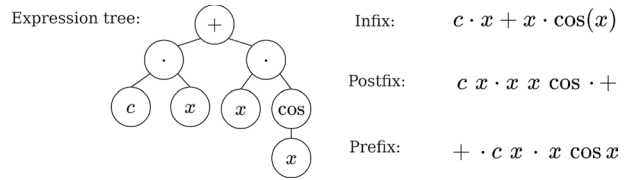
Hierarchical data have been tackled by generative models in several ways. By making a node depend on its parent and previous sibling, DRNN (Alvarez-Melis & Jaakkola, 2017) combines representations obtained from the depth-wise and width-wise recurrent cells to generate new nodes, which proves useful for recovering the structure of a tree. On the other hand, Tree-LSTM (Tai et al., 2015) and JT-VAE (Jin et al., 2021) focus on adapting equations for recurrent cells to encode (and decode) hierarchical structures more efficiently. Tree-LSTM proposes a generalization of the LSTM cell for encoding trees into a representation that proves effective for classification tasks and semantic relatedness of sentence pairs. JT-VAE adapts recurrent cells for tree message passing. Trees are used as scaffolding for the graph that represents molecules. Encoding and decoding are thus split into four parts: encoding of the graph, encoding of the tree, decoding of the tree, and decoding of the graph. While these adaptations are similar to the ones presented in our work, their focus is on encoding more general structures that are unrelated to mathematical expressions.

Note that our model falls into the general framework of gated *recursive* convolutional neural networks (Cho et al., 2014a) that combine atomic units with shared parameters in a hierarchy. The output of the root node produces a fixed-length encoding of a data object with an arbitrarily complex structure. Another model, marginally related to ours, is the one of equivalence neural networks (Allamanis et al., 2017). The encoding produced by these networks follows the expressions' semantic similarity and equivalence, in contrast to their syntactic similarity, which is followed by all the other approaches, including ours.

Finally, our work is also related to algorithms for equation discovery and symbolic regression. Most of them generate candidate expressions for equations first and then estimate the values of their constant parameters by matching the equations against data in the second phase. Classical symbolic regression approaches (Cranmer, 2023; Guimerà et al., 2020; Schmidt & Lipson, 2009), based on evolutionary algorithms, use stochastic generators of expression trees: At the beginning, the expression trees are randomly sampled, and later on, they are transformed using the evolutionary stochastic operations of mutation and cross-over. In contrast, process-based modeling approaches (Bridewell et al., 2008) generate equations by following domain-specific knowledge (provided by the user) that specifies a set of entities (variables) and processes (interactions among entities). Grammar-based approaches to equation discovery employ user-specified context-free grammars (which can also be based on domain knowledge), deterministic (Todorovski & Dzeroski, 1997) or probabilistic (Brence et al., 2021), as efficient generators of expressions.

Recently, many symbolic regression approaches based on neural networks have been proposed (Biggio et al., 2021; d'Ascoli et al., 2022; Kamienny et al., 2022; Martius & Lampert, 2016; Petersen et al., 2021; Udrescu & Tegmark, 2020). In particular, Deep Symbolic Optimization, DSO approaches symbolic regression [among other optimization tasks (Mundhenk et al., 2021)] by combining neural networks and reinforcement learning with evolutionary algorithms. The neural networks are used to sample the individuals in the initial population of the evolutionary algorithm and are retrained at each iteration to focus on expressions leading to better fit. It is closely related to our work, since it combines similar methods. Yet our focus here is on efficient neural networks for generating mathematical expressions that are trained before the beginning of the evolutionary process.

Fig. 2 An expression tree with a height of four and three sequence-based representations of the corresponding mathematical expression

Expression tree:

Infix: $\quad c \cdot x + x \cdot \cos(x)$

Postfix: $\quad c \; x \cdot x \; x \; \cos \cdot +$

Prefix: $\quad + \cdot c \; x \cdot x \; \cos x$

3 Methodology

We start this section by briefly introducing the task of symbolic regression and the search space of mathematical expressions (Sect. 3.1). After this, we introduce variational autoencoders and the structure of the hierarchical variational autoencoder, HVAE (Sect. 3.2). We finish the section by explaining how to use HVAE for generating mathematical expressions and how to combine it with an evolutionary algorithm for symbolic regression (Sect. 3.3).

3.1 Symbolic regression and expression trees

Symbolic regression (SR) is the machine learning task of discovering equations in collections of measured data. Symbolic regression methods take a data set S consisting of multiple measurements of a set of real-valued variables $V = \{x_1, x_2, \ldots, x_p, y\}$, where y is a designated target variable. The output of SR is an equation of the form $y = f(x_1, x_2, \ldots, x_p)$, where the right-hand side of the equation is a closed-form mathematical expression. The equation should provide an optimal fit against the measurements from S, i.e., minimize the discrepancy between the observed values of the target variable y and values calculated by using the equation. Symbolic regression methods usually follow the parsimony principle, preferring simpler expressions over more complex ones.

Symbolic regression methods search through the space of candidate mathematical expressions for the right-hand side of the equation to find the one that optimally fits the measurements. Mathematical expressions can be represented in different ways. We commonly use the infix notation, where operators are placed between two sub-expressions they operate on, e.g., $A + B$, where A and B are sub-expressions. Infix notation uses parentheses to indicate the order in which the operations need to be performed. Prefix (Polish) or postfix (reverse Polish) notations do not need parentheses since the operators are written before or after the two sub-expressions, e.g., $+AB$ or $AB+$. The three notations correspond to different traversals of the nodes in an expression tree. The latter is a hierarchical data structure, where the inner nodes correspond to mathematical operators and functions, while the leaf nodes correspond to variables and constants.

In symbolic regression, the constants' values are fitted against the measured data from S, while variables include elements from V without the target variable. We assume binary expression trees since standard arithmetic operators are binary. We take that the second descendant node is null in the inner nodes corresponding to single-argument functions. We define the height of an expression tree as the number of nodes on the longest path from the root node to one of the leaves. Figure 2 depicts an example expression tree with a height of four, along with the corresponding mathematical expression in different notations.

Our model generates expression trees, as they have several advantages over sequences (strings). Firstly, it is easy to achieve syntactic correctness, since operators and functions

are in the inner nodes, while variables and constants are in the leafs. Secondly, information needs to travel at most $\log n$ steps up the tree (up to the tree's height) instead of n steps along the sequence (up to the length of the sequence). Lastly, sub-expressions can be encoded independently of each other during the encoding process.

3.2 Hierarchical variational autoencoder

In recent years, variational autoencoders (Kingma & Welling, 2014) have emerged as one of the most popular generative models. The reason for this is that, when trained correctly, variational autoencoders map the observed data with an unknown distribution into a latent representation with a known distribution. This results in a continuous latent space, from which one can sample and synthesize new data. In contrast to a (deterministic) autoencoder, where the encoder outputs a latent representation z that is directly fed into the decoder, the encoder in the variational autoencoder outputs the parameters for an approximate posterior distribution, e.g., μ and σ in the case of a latent space parameterized by a multivariate Gaussian distribution.

Thus, a representation z that is fed into the decoder is sampled from the underlying distribution with the learned parameters (μ, σ). The loss of the variational autoencoder is the reconstruction error, i.e., the difference between the input to the encoder and the output of the decoder. Additionally, variational autoencoders typically use Kullback–Leibler (KL) divergence (Kullback & Leibler, 1951) as the regularization term for the loss. The loss can thus be calculated as:

$$J(x, \mu_z, \sigma_z) = J_{\text{rec}}(x) + \lambda \cdot \text{KL}(\mathcal{N}(\mu_z, \sigma_z) \,\|\, \mathcal{N}(0, I)), \tag{1}$$

where $J_{\text{rec}}(x)$ is the reconstruction loss of x and $\lambda \geq 0$ the regularization cost parameter. In case the underlying distribution is Gaussian, KL divergence to an isotropic unit Gaussian can be estimated as

$$\text{KL}(\mathcal{N}(\mu_z, \sigma_z) \,\|\, \mathcal{N}(0, I)) = \frac{1}{2}\left(1 + \log \sigma_z^2 - \mu_z^2 - \sigma_z^2\right). \tag{2}$$

We use cost annealing (Bahuleyan, 2018) to focus on the reconstruction error (i.e., use small values of λ) at the beginning and then gradually shift the focus towards smoothness of the latent space by increasing the value of λ.

3.2.1 Model overview

Our approach uses a variational autoencoder structure that consists of an encoder and a decoder. The encoder takes tree-structured data as input and outputs a distribution in the latent vector space, represented with the mean (μ_z) and the logarithm of the variance ($\log \sigma_z$) vectors. The decoder works in the opposite direction, sampling a point from the latent vector space as input and transforming it into a binary expression tree. To make the backward propagation possible, we sample points with the reparametrization trick.

Trees are encoded recursively, starting from leafs and ending at root nodes. To encode a subtree with a root in n, we first encode its left and right subtrees. We then pass their codes, along with the symbol in the node n, as inputs to the encoding cell (further described in Sect. 3.2.2). This cell outputs the code of the subtree rooted in n. At the beginning of the recursion, in each leaf node, the codes corresponding to the (missing) children are assumed to be vectors of zeros. Once the root of the tree is encoded, its code is passed through two fully

(a) Encoding   (b) Decoding
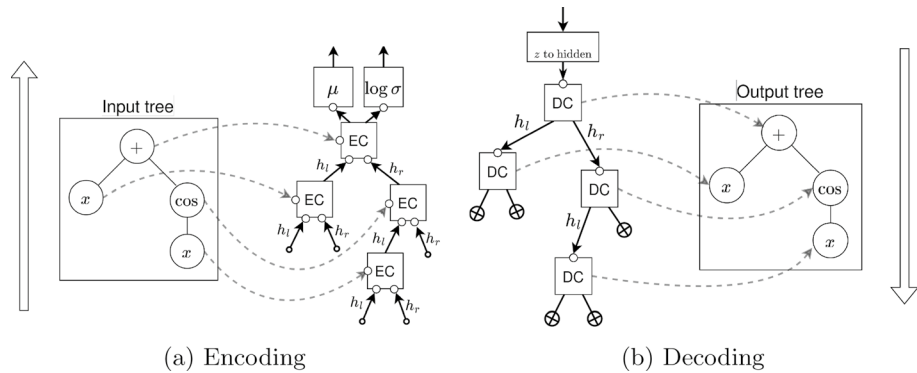
Fig. 3 The processes of **a** encoding and **b** decoding the expression tree of $x + \cos x$. The acronyms EC and DC stand for "encoding cell" (introduced in Sect. 3.2.2) and "decoding cell" (introduced in Sect. 3.2.3)

connected layers that give the mean and log-variance vectors that form the latent representation of the tree. Figure 3a illustrates the recursive encoding process on the expression $x + \cos x$.

The first layer of the decoder transforms the sampled point from the latent space into the code of the hierarchy. After this, the tree is generated recursively by passing the code of the current node (subtree) through the decoding cell (further described in Sect. 3.2.3). This cell takes the code of the node (subtree) as input and generates a node symbol, along with the codes of the two child nodes. There are three possible symbol types. If we encounter an operator, both child nodes are generated recursively. On the other hand, if the symbol represents a function, we only generate the left child. Lastly, if the symbol is either a variable or a constant, no further child nodes are generated in this branch. This process is shown in Fig. 3b, where the expression $x + \cos x$ is decoded.

During training, we follow the structure of the encoded tree and try to predict the correct node symbols. In turn, we jointly learn to predict the structure of the expression tree and the symbols inside the node, since the structure is determined by the symbols. We calculate the loss using cross-entropy on a sequence of symbols obtained with the in-order traversal of the expression tree. Some additional implementation details are explained in Appendix 4.

3.2.2 Encoder

The encoding proceeds in two phases. The first follows the hierarchy of the input and applies the encoding cell to each node of the hierarchy as described above. In the second phase, the code of the root node is transformed into the mean and log-variance vectors of the input's latent representation.
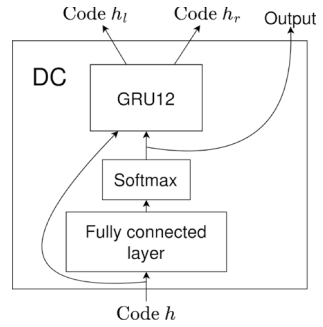
Encoding comprises a GRU21 cell, which we have adapted from the GRU cell (Cho et al., 2014b). The (output) code h in GRU21 is computed from the input vector x, and codes h_l of the left and h_r of the right child with the following equations:

$$r = \varphi_S(W_{ir}x + b_{ir} + W_{hr}(h_l \parallel h_r) + b_{hr}) \tag{3}$$

$$u = \varphi_S(W_{iu}x + b_{iu} + W_{hu}(h_l \parallel h_r) + b_{hu}) \tag{4}$$

Fig. 4 The structure of the decoding cell

$$n = \tanh(W_{in}x + b_{in} + r * (W_{hn}(h_l \mathbin{\|} h_r) + b_{hn})) \tag{5}$$

$$h = (1 - u) * n + \frac{u}{2} * h_l + \frac{u}{2} * h_r, \tag{6}$$

where φ_S denotes the Sigmoid activation function. In these expressions, r, u, and n represent the standard reset gate, update gate and candidate activation vectors from a GRU cell. When compared to the original equations of the GRU cell, Eqs. (3), (4), (5) exhibit two differences. First, instead of the code of the previous symbol in the sequence, the concatenation of the codes h_l and h_r of the child nodes is used (denoted by $(h_l \mathbin{\|} h_r)$). Second, the dimension of the weight matrices W_{hr}, W_{hu}, W_{hn} must be $\dim(h_l) + \dim(h_r)$ instead of $\dim(h)$. Thus, while Eq. (6) remains similar to the original one, we change the second term (from its usual form $u * h_{t-1}$) to $\frac{u}{2} * h_l + \frac{u}{2} * h_r$, to retain information from the codes of the two child nodes. Recall that $*$ denotes the element-wise multiplication of vectors.

In the second phase, the model transforms the code of the root node into the latent representation of the input expression through two fully-connected layers.

3.2.3 Decoder

The decoding also comprises two phases. In the first, a fully-connected layer transforms a point from the latent vector space into the code of the root node. In the second phase, the decoding cell is recursively deployed to decode each of the nodes in the expression tree.

Figure 4 depicts the structure of the decoding cell. The cell is composed of a fully connected layer, a softmax layer, and the GRU12 cell, an adaptation of the original GRU cell. The input code is first passed through the sequence of a fully-connected and a softmax layer. The latter creates the vector of probabilities, from which the most probable output symbol is chosen. If the output symbol is either a constant or a variable, the decoding stops. Otherwise, the output vector is also used as an input to the GRU12 cell, together with the code that is given as input into the decoding cell. The GRU12 cell produces two codes, one for the left and one for the right child.

GRU12 computes the two codes h_l and h_r for the child nodes using the input vector x and the code h with the following equations:

$$r = \varphi_S(W_{ir}x + b_{ir} + W_{hr}h + b_{hr}) \tag{7}$$

$$u = \varphi_S(W_{iu}x + b_{iu} + W_{hu}h + b_{hu}) \tag{8}$$

$$n = \tanh(W_{in}x + b_{in} + r * (W_{hn}h + b_{hn})) \tag{9}$$

$$d = (1 - u) * n + u * (h \,\|\!\!\!\!\,\| \, h) \tag{10}$$

$$d \equiv h_l \,\|\!\!\!\!\,\| \, h_r \tag{11}$$

There are two major differences between GRU12 and the original GRU cell. First, the vectors r, u, and n in Eqs. (7), (8), (9) are of dimension $2 \cdot \dim(h)$ instead of $\dim(h)$. Consequently, all bias vectors are of dimension $2 \cdot \dim(h)$, and all weight matrices have an output dimension of $2 \cdot \dim(h)$. Second, in Equation (10), the code h is concatenated with itself to make the dimensions in the equation match. Vector d is then split in half in Eq. 11. The first part is used as a code for the left child, while the second is used as a code for the right child.

3.3 Generating expressions for symbolic regression

Recall that the goal of symbolic regression is to efficiently search through the space of mathematical expressions and find the one that, when used on the right-hand of an equation, fits given measurements well. In this section, we explain how to use HVAE for generating expressions.

3.3.1 HVAE as a generative model

We can generate expressions in two ways, corresponding to two different symbolic regression scenarios. The first way, which aims at discovering equations from data, samples random vectors from the standardized Gaussian distribution $\mathcal{N}(0, I)$ in the latent space and passes them through the decoder.

On the other hand, we might want to generate expressions in a scenario that corresponds to the revision of existing equations to fit newly gathered data. Here, we want to generate mathematical expressions that are similar to the one given as input. Our approach achieves this by encoding an expression and sampling its immediate neighborhood in the latent space. We expect these points to be decoded into expressions similar to the one given as input. We will show that HVAE meets this expectation in Sect. 4.1.4.

3.3.2 Evolutionary algorithm operators

Finally, we can search the latent space spanned by our model with evolutionary algorithms (Koza, 1994), one of the most commonly used paradigms for symbolic regression. Evolutionary algorithms explore the search space by first randomly sampling individuals for the initial population. Then they repetitively generate new populations by combining pairs of individuals from the current population with the genetic operators of mutation and crossover.

An *individual* in a population is in our case a real-valued vector z, corresponding to the code of an expression tree in the latent vector space. Using the HVAE model, z can be decoded into an expression tree. To calculate the individual's fit against the training data, we first fit the values of the constant parameters in the decoded expression tree and then measure the error of the equation with the resulting expression on the right-hand side (with

respect to the training data). We generate the *initial population* by randomly sampling individuals from the Gaussian distribution $\mathcal{N}(0, I)$.

Crossover combines two individuals, referred to as parents z_A and z_B, into an offspring z_O. We generate the latter as a convex combination of z_A and z_B, i.e. $z_O = (1 - a) \cdot z_A + a \cdot z_B$, where a is sampled from the uniform distribution on the interval [0, 1]. For values of a close to 0 and 1, the offspring is close to one of the parents, while values of a close to 0.5 lead to an offspring equally dissimilar to both parents.

The *mutation* operator transforms an individual z into a mutated individual z_M. To perform a mutation, we first decode z into an expression tree and immediately encode it back into its latent space representation to obtain the value of σ_z. Now, we can mutate z into an individual with a syntactically similar expression by sampling from $\mathcal{N}(\mu_z, \sigma_z)$ or into a random individual by sampling the offspring z_O from $\mathcal{N}(0, I)$. Similarly to the case of crossover, we interpolate between these two extremes by sampling from $\mathcal{N}(a \cdot \mu_z + (1 - a) \cdot 0, a \cdot \sigma_z + (1 - a) \cdot I) = \mathcal{N}(a \cdot \mu_z, a \cdot \sigma_z + (1 - a) \cdot I)$, where a is randomly sampled from the uniform distribution on the interval [0, 1]. When a is close to 0, the offspring z_O is chosen at random (see the first paragraph of Sect. 3.3.1). On the other hand, when a is close to 1, z_O is syntactically similar to z (second paragraph of Sect. 3.3.1).

We implement the EDHiE (Equation Discovery with Hierarchical variational autoEncoders) approach for symbolic regression by combining HVAE with evolutionary algorithms using these operators. Our implementation uses pymoo (Blank & Deb, 2020) for evolutionary algorithms and ProGED (Brence et al., 2021) functionality for evaluating the fit of a candidate equation.

4 Evaluation

In this section, we will investigate the validity of our hypothesis that the hierarchical variational autoencoder is a more efficient generator of mathematical expressions than the alternative VAEs for sequences by conducting two series of computational experiments. In the first series, we are going to evaluate the performance and efficiency of HVAE on the task of generating mathematical expressions. In the second series, we will evaluate the performance of EDHiE on the symbolic regression downstream task.

4.1 The performance of HVAE

We start this section by introducing the experimental setup (Sect. 4.1.1). We continue with reporting the experimental results of evaluating HVAE with respect to the reconstruction error (Sect. 4.1.2), efficiency in terms of the size of training data needed, the dimensionality of the latent space (Sect. 4.1.3), and finally the smoothness of the latent space (Sect. 4.1.4). In Appendix 2, we further justify our claim that points close in the latent space of HVAE are decoded into similar expressions.

4.1.1 Experimental setup

Data sets We estimate the reconstruction error of the variational autoencoders on a collection of six synthetic data sets, ranging from small ones, including simple expressions, to large ones, including complex expressions. The data sets are as follows:

AE4-2k, AE5-15k, and AE7-20k have 2, 15, and 20 thousand mathematical expressions with trees with a maximum height of four, five, and seven. These expressions can contain constants, variables, and the operators +, −, ·, /, and ˆ.
Trig4-2k, Trig5-15k, and Trig7-20k are the same as above, but the expressions also contain the sine and cosine functions.

We create these data sets with the ProGED (Brence et al., 2021) system by randomly sampling mathematical expressions from a given probabilistic context-free grammar. The generated expressions are simplified using the Python library SymPy (Meurer et al., 2017). The context-free grammars that constrain the output of GVAE and the ones used to generate the data sets are documented in Appendix 1.

Parameter setting We train GVAE and CVAE for 150 epochs with the following values of their hyper-parameters: latent dimension = 128, hidden dimension = 128, batch size = 64, kernel sizes of the convolution layers = 2, 3, 4, and the ADAM optimizer (Kingma et al., 2015). For reconstruction results created with our approach (HVAE), the hyper-parameters are: latent size = 128, batch size = 32, and the ADAM optimizer with the default learning rate. For the first 1,800 iterations i, we calculate the regularization cost parameter λ using $\lambda_i = 0.5 \cdot (\tanh \frac{i-4,500}{2} + 1)$, after this, we set λ_i to $\lambda_{1,800}$.

Estimating the reconstruction error The Levenshtein distance (Levenshtein, 1965) (often referred to as the edit distance) quantifies the dissimilarity of two strings in terms of the number of insertion, removal, and substitution operations that are needed to transform one string into the other. We use this distance to test how well our autoencoder recreates expressions.

We first pass the expression through the VAE to get the predicted expression. If needed, we validate the syntactical correctness of the latter and transform it into an expression tree. We then traverse the input and the output trees in post-order (left child, right child, node symbol) to obtain the input and the output expressions in the postfix notation (which does not require parentheses and is hence more suited for calculating the distance between expressions). Finally, we calculate the edit distance between those two strings.

To estimate the reconstruction error on unseen expressions, we use five-fold cross-validation with the same splits across all methods. GVAE and CVAE sometimes produce invalid expressions, which we discard from the evaluation. Because of this, the results in Sects. 4.1.2 and 4.1.3 might be biased in favor of CVAE due to many syntactically incorrect expressions being discarded. Note that GVAE has fixed-size input (and output) that might be too short for encoding all the grammar rules needed to derive an expression. In those cases, GVAE returns empty strings, which we consider invalid expressions. CVAE, on the other hand, produces syntactically incorrect expressions such as $xc(/x)c) \sin \sin(c))$, $\cdot \cdot x - c \cdot / \sin(x))$, or $(/x(-x)c)$ (presented here in infix notation).

4.1.2 Out-of-sample reconstruction error

Table 1 compares the out-of-sample reconstruction error and the ratio of invalid expressions for the three variational autoencoders. Our hierarchical VAE significantly outperforms the other two methods on all data sets. An interesting observation is that GVAE works consistently better on expressions involving trigonometric functions, while HVAE and CVAE perform worse. The reason for the opposite effect is probably the following: for GVAE, functions only represent yet another production rule in the grammar, while for HVAE and CVAE they drastically change the structure of the expression (tree). This

Table 1 The out-of-sample reconstruction error and the percentages of syntactically incorrect expressions generated by the three variational autoencoders

Dataset	HVAE		GVAE		CVAE	
	Edit distance	Invalid	Edit distance	Invalid	Edit distance	Invalid
AE4-2k	**0.076** (± **0.024**)	**0.0** (± **0.0**)	3.959 (± 0.135)	0.2 (± 0.0)	3.873 (± 0.132)	33.8 (± 1.1)
Trig4-2k	**0.119** (± **0.026**)	**0.0** (± **0.0**)	3.199 (± 0.068)	**0.0** (± **0.0**)	3.619 (± 0.045)	48.3 (± 0.6)
AE5-15k	**0.079** (± **0.014**)	**0.0** (± **0.0**)	2.827 (± 0.280)	< 0.1 (± 0.0)	1.547 (± 0.466)	3.5 (± 0.0)
Trig5-15k	**0.093** (± **0.010**)	**0.0** (± **0.0**)	1.489 (± 0.195)	< 0.1 (± 0.0)	2.086 (± 0.346)	13.9 (± 0.0)
AE7-20k	**0.501** (± **0.017**)	**0.0** (± **0.0**)	5.201 (± 0.289)	< 0.1 (± 0.0)	3.654 (± 0.349)	9.9 (± 0.0)
Trig7-20k	**0.530** (± **0.036**)	**0.0** (± **0.0**)	3.423 (± 0.467)	< 0.1 (± 0.0)	3.660 (± 0.287)	26.3 (± 0.1)

Bold values indicate the best performing approaches (on a specific data set/expression)

translates to better performance of GVAE, as expressions with trigonometric functions are usually shorter, given that the nodes corresponding to the trigonometric functions have only one descendant instead of the usual two.

The percentages of invalid expressions generated by the approaches show that our approach always produces syntactically correct expressions, while GVAE and CVAE sometimes fail to produce valid outputs. The fraction of such expressions is quite small when the GVAE approach is used (see the explanation above) but quite significant when CVAE is used. Lastly, we can notice that, as expected, longer expressions are harder to recreate and thus have higher edit distance and a higher percentage of invalid expressions than shorter ones, provided enough training data is used.

4.1.3 Training efficiency and the latent space dimensionality

We proceed to test our conjectures about the efficiency of training the generators of mathematical expressions. We expect that HVAE would require less training data and a lower dimensionality of the latent space to achieve the same levels of reconstruction error in comparison to other approaches. The latter is especially important because of the exploration of the latent space, which is more efficient in low-dimensional latent spaces.

Figure 5a depicts the impact of the number of expressions in the training set on the reconstruction error for the three different generative models. Again, HVAE significantly outperforms the other two VAEs. Its reconstruction error is estimated to be consistently lower than 0.25, even when trained on 2 thousand examples only. This error is an order of magnitude lower than the lowest error of 1.5 achieved by the second best model, GVAE, when trained on the whole data set of 12 thousand examples.

Figure 5b shows the impact of the dimensionality of the latent space on the reconstruction error across different VAEs. In line with the previous results, HVAE significantly outperforms both CVAE and GVAE. HVAE with latent space of dimension 16 performs on par or better than GVAE and CVAE with latent spaces of 256 dimensions. We can see that the reconstruction error quickly raises when the latent space dimension is less than 32, but otherwise, the reconstruction error is consistently low. Even with a latent space size of 16, our approach is still comparable to the other two methods with a latent space of dimension 256. This allows us to reduce the

(a) Size of the training set   (b) Dimensionality of the latent space
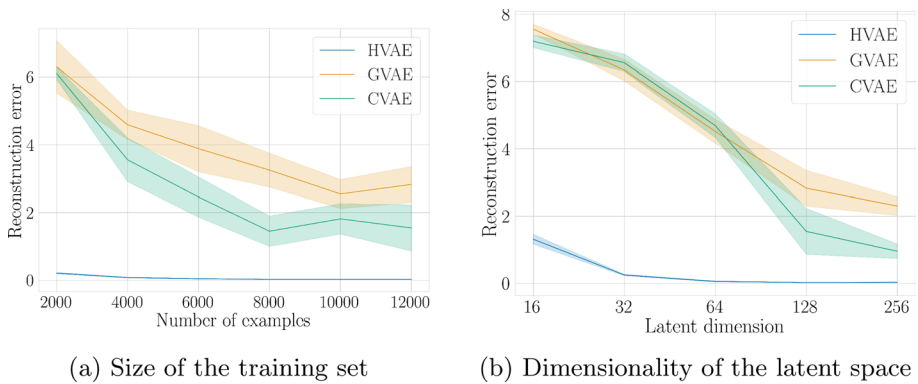
Fig. 5 The impact of the **a** training data set size and **b** dimensionality of the latent space on the reconstruction error of the three autoencoders

dimensionality of the latent space by two orders of magnitude, which makes HVAE an excellent candidate for generating expressions for symbolic regression.

The reason for the superior efficiency of HVAE is the use of expression trees, as subexpressions are always encoded into the same code, regardless of their position in the expression. This significantly reduces the space of possible codes and allows for training the model in a way that better generalizes to the repetitive subexpressions (subpatterns) it encounters.

4.1.4 Latent space smoothness

Finally, we expect the latent space of HVAE to be smooth in the sense that samples close to the latent representation of the input expression are decoded into expressions similar to the one given as input. We investigate the validity of this conjecture by applying linear interpolation (performing a *homotopic* transformation) between two expressions in the latent space. Assume that we are given two expressions, A and B. Using the model, we encode them into their latent representations z_A and z_B. We can then generate new latent representations z_α by combining the two representations with the formula $z_\alpha = (1 - \alpha) \cdot z_A + \alpha \cdot z_B$, where $\alpha \in \{i/n : i \in \mathbb{N} \wedge i \leq n\}$. Decoding the latent representations z_α in a smooth latent space should produce intermediate expressions that represent a smooth transition from A to B in n steps.

Table 2 shows the results of such a linear interpolation in the latent spaces of the different VAEs. HVAE and GVAE produce continuous latent spaces where the transition from expression A to expression B is indeed smooth. CVAE also produces a smooth transition, but some of the intermediate expressions might be syntactically incorrect. The second interpolation in the lower part of the table is an example of a smooth transition in the HVAE latent space. We can see that at each step only a few expression symbols change and that these changes are rarely redundant. Appendix 2 provides further examples of interpolations with visualizations of the expression trees.

Table 2 Examples of linear interpolation between two expressions in the latent spaces of the three VAEs. Expressions that are colored red are syntactically incorrect. Here, we set $n = 4$ and $\alpha = i/4, 0 \leq i \leq 4$

α	HVAE	GVAE	CVAE
Expression A	$c \cdot \cos c + \frac{c}{x} + \frac{x}{\sin x}$	$c \cdot \cos c + \frac{c}{x} + \frac{x}{\sin c}$	$c \cdot \cos c + \frac{c}{x} + \frac{x}{\sin c}$
$\alpha = 0$	$c \cdot x + \frac{c}{x} + \frac{x}{\sin x}$	$c \cdot \cos c + \frac{c}{x} + \frac{x}{\sin c}$	$c \cdot \cos c + \frac{x}{x^c} \cdot \sin x$
$\alpha = 0.25$	$c \cdot \sin c + \frac{c}{x} + x$	$c + \frac{c}{\cos c} + c \cdot x$	$\frac{c}{\cos c} + c - \frac{x}{c} \cdot \sin x$
$\alpha = 0.5$	$\frac{c \cdot \sin c}{c} + x$	$c + \frac{x}{c^{\cos c}} \cdot x$	$c - \cos c - \frac{x}{x} \cdot \sin c$
$\alpha = 0.75$	$\frac{\sin(x - \sin c)}{x}$	$x + c \cdot \frac{c}{x} - x$	$\cos(x(c) - c\cdot$
$\alpha = 1$	$\frac{\sin(x-c)}{x}$	$\frac{\sin(x-c)}{x}$	$\frac{\sin(x-c)}{x}$
Expression B	$\frac{\sin(x-c)}{x}$	$\frac{\sin(x-c)}{x}$	$\frac{\sin(x-c)}{x}$
Expression A	$x - \sin(c \cdot x)$	$x - \sin(c \cdot x)$	$x - \sin(c \cdot x)$
$\alpha = 0$	$x - \sin(c \cdot x)$	$\frac{x}{\sin(c \cdot x)}$	$x - \sin(c \cdot x)$
$\alpha = 0.25$	$x \cdot \sin c - \sin(c \cdot x)$	$\frac{x}{\sin(c \cdot x)}$	$x - \cos(c \cdot x)$
$\alpha = 0.5$	$c \cdot \sin x + x$	$c + \cos c$	$\frac{c}{\cos} - \cos \cdot c) - c$
$\alpha = 0.75$	$c \cdot \cos \frac{x}{c} + c$	$c \cdot x \cdot \frac{\cos(\frac{x}{c})}{c}$	$c \cdot x \cdot \frac{\cos(\frac{c}{x})}{c}$
$\alpha = 1$	$c \cdot x \cdot \cos \frac{x}{c} + c$	$c \cdot x \cdot \cos(\frac{x}{c}) + c$	$c \cdot x \cdot \cos(\frac{x}{c}) + c$
Expression B	$c \cdot x \cdot \cos \frac{x}{c} + c$	$c \cdot x \cdot \cos(\frac{x}{c}) + c$	$c \cdot x \cdot \cos(\frac{x}{c}) + c$

4.2 Evaluating EDHiE

In the second series of experiments, we evaluate the performance of EDHiE. We start the section by introducing the experimental setup (Sect. 4.2.1). We then report on the impact of the dimensionality of the latent space on the performance of symbolic regression (Sect. 4.2.2). Furthermore, we compare the performance of EDHiE with that of other methods for symbolic regression on the Nguyen benchmark (Sect. 4.2.3) and report the performance of EDHiE on the Feynman benchmark (Sect. 4.2.4).

4.2.1 Experimental setup

Data sets The Nguyen (Uy et al., 2011) benchmark contains eight equations with one non-target variable and two equations with two non-target variables. The right-hand sides of these equations are shown in the second column of Table 3. We generate two data sets (train and test) with five thousand simulated measurements for each equation. We use the train set to select the best expressions and the test set to evaluate their performance with the metrics described below. We sample points from the interval $[-20, 20]$ for equations 1–6, the interval $[0, 40]$ for equation 7, $[0, 80]$ for equation 8, and $[0, 20]$ for equations 9 and 10.

We further evaluate our approach on the 16 equations involving up to two variables from the Feynman benchmark (Udrescu & Tegmark, 2020). The right-hand sides of these equations are shown on in the last column of Table 3. Because equations in the Feynman benchmark represent real-world equations, each of the equations FM-3, FM-4, FM-5, and FM-7 contains two entries. Each entry comes with its own variables and data sets of measurements.

Evaluation process We compare the performance of EDHiE on the Nguyen benchmark equations to the performance of three other symbolic regression systems. ProGED (Brence et al., 2021) uses probabilistic grammars as generators of mathematical expressions.

Table 3 Expressions from the Nguyen (first two columns on the left-hand side) and Feynman (last three columns on the right-hand side) benchmarks

ID	Expression	ID	ID-Feynman	Expression
NG-1	$x^3 + x^2 + x$	FM-1	I.6.2a	$(2\pi)^{-0.5} e^{-\frac{x^2}{2}}$
NG-2	$x^4 + x^3 + x^2 + x$	FM-2	I.6.2	$(\sqrt{2\pi} \cdot y)^{-1} e^{-\frac{(x/y)^2}{2}}$
NG-3	$x^5 + x^4 + x^3 + x^2 + x$	FM-3	I.12.1, I.12.5	xy
NG-4	$x^6 + x^5 + x^4 + x^3 + x^2 + x$	FM-4	I.14.4, II.8.31	$0.5\, xy^2$
NG-5	$\sin x^2 \cdot \cos x - 1$	FM-5	I.25.13, I.29.4	x/y
NG-6	$\sin x + \sin(x + x^2)$	FM-6	I.26.2	$\arcsin(x \sin y)$
NG-7	$\ln(x + 1) + \ln(x^2 + 1)$	FM-7	I.34.27, III.12.43	$(2\pi)^{-1} xy$
NG-8	$\sqrt{x}$	FM-8	I.39.1	$1.5\, xy$
NG-9	$\sin x + \sin y^2$	FM-9	II.3.24	$\frac{x}{4\pi y^2}$
NG-10	$2 \sin x \cdot \cos y$	FM-10	II.11.28	$\frac{1+xy}{1-(0.\overline{3}\, xy)}$
		FM-11	II.27.18	xy^2
		FM-12	II.38.14	$\frac{x}{2 \cdot (1+y)}$

DSO (Petersen et al., 2021) combines deep neural networks with evolutionary algorithms. PySR (Cranmer, 2023) employs evolutionary optimization with operators directly applied to the expression trees. We run each system ten times on each equation and evaluate at most 100,000 sampled expressions. All approaches use the same library of tokens and/or grammars, further described in Appendix 1. When running PySR, we allow fitting the values of the constant parameters since it can not be turned off in the implementation. The dimensionality of the latent space of HVAE is set to 32; the ADAM optimizer uses the default learning rate. We elaborate on the setting of the dimensionality of the latent space in the next section. Appendix 3 gives the complete report on the experiments in latent spaces with varying dimensions.

Estimating the performance We use three metrics: the number of successful reconstructions, i.e., runs leading to an equation equivalent to the original one; the mean R^2 of the best equation; and the number of expressions sampled to achieve reconstruction. We consider a run successful if we find an expression where the RMSE between the target and predicted values is lower than 10^{-10}. To guarantee accurate reporting, we manually check if the original and expression with the lowest RMSE are equivalent. In each run, we use the expression with the lowest RMSE to calculate the bounded R^2 metric on the test set using the formula

$$R^2(\hat{y}, y) = \max\left(0, 1 - \frac{\sum_i (y_i - \hat{y}_i)^2}{\sum_i (y_i - \overline{y})^2}\right), \tag{12}$$

where $\hat{y}_i$ denotes the predicted value of the target variable (calculated by using the equation), y_i is the measured value of the target variable, and $\overline{y}$ is the mean value of y in the training data set. Lastly, we show the average number (across the ten runs) of unique expressions considered before reconstructing the original equation. To this end, we count the unique expressions that the symbolic regression system has considered before the reconstructed equation is encountered in the generation process for all the methods that report this.

Table 4 The performance of symbolic regression (number of successful reconstructions) by randomly sampling with CVAE, GVAE, and HVAE on the Nguyen benchmark

Equation/approach	CVAE			GVAE			HVAR		
Latent space size	32	64	128	32	64	128	32	64	128
NG-1	4	2	2	10	9	10	10	10	10
NG-2	0	0	0	2	4	3	10	5	9
NG-3	0	0	0	0	0	0	0	0	1
NG-4	0	0	0	0	1	0	0	0	0
NG-5	0	0	0	0	0	0	0	0	0
NG-6	0	0	0	2	0	0	4	4	0
NG-7	0	0	0	0	0	0	0	0	0
NG-8	10	10	3	10	10	10	10	10	10
Total/Mean	14	12	5	24	24	23	34	29	30

4.2.2 The impact of the dimensionality of the latent space

Let us start with a series of computational experiments exploring the latent space for encoding mathematical expressions with random sampling. Here, we perform symbolic regression by taking randomly sampled points in the latent space and decoding them into expressions that are then evaluated on the measurements/data. The expression that fits the data best is selected as the candidate for the discovered equation.

Table 4 shows the number of successful runs of the random sampling approaches based on the three VAEs, CVAE, GVAE, and HVAE. In the further discussion of results, we use the name HVAR for HVAE with random sampling. We can see here a typical example of the curse of dimensionality at work. When the symbolic regression algorithm explores high-dimensional latent spaces, it can easily slip into parts of those spaces that do not lead to optimal equations. This shows that the ability of HVAE to encode mathematical expressions in low-dimensional latent spaces is crucial for the performance of symbolic regression with HVAR.

Based on the results of the experiments in Table 4 and Appendix 3, in the remainder of this section, we use 32-dimensional latent space for EDHiE.

4.2.3 Comparison on the Nguyen equations

In the next series of experiments, we compare the performance of HVAR, the random sampling method using HVAE, to the one of ProGED—the latter samples mathematical expressions using manually crafted probabilistic grammar. Table 5 reports the results of the comparison. The results show that the generator used within HVAE is not worse than the probabilistic grammar. To our surprise, HVAR outperforms ProGED significantly. First, it successfully reconstructs five (of the ten) equations from the Nguyen benchmark in ten runs, one more than ProGED. Second, for the three equations of NG-2, NG-6, and NG-9, the reconstruction is achieved faster, i.e., by evaluating fewer candidate expressions.

Furthermore, we check the contribution of the evolutionary approach in EDHiE over the random sampling method HVAR. To this end, we compare the last three columns of Table 5 with the last three columns of Table 6. EDHiE successfully reconstructs all ten

Table 5 Comparison of the performance of symbolic regression (number of successful reconstructions, R^2, and number of evaluated equations) with random sampling on the Nguyen benchmark. We compare sampling from a manually-crafted probabilistic grammar (ProGED) with sampling using a trained HVAE (HVAR)

Name	ProGED (Brence et al., 2021)			HVAR (Ours)		
	Successful	Average R^2	Evaluated	Successful	Average R^2	Evaluated
NG-1	**10**	**1.00 ($\pm$ 0.00)**	2374 ($\pm$ 1451)	**10**	**1.00 ($\pm$ 0.00)**	**901 ($\pm$ 1332)**
NG-2	2	1.00 ($\pm$ 0.01)	**7680 ($\pm$ 670)**	**10**	**1.00 ($\pm$ 0.00)**	9729 ($\pm$ 5337)
NG-3	0	1.00 ($\pm$ 0.01)	NA	0	1.00 ($\pm$ 0.01)	NA
NG-4	0	1.00 ($\pm$ 0.01)	NA	0	1.00 ($\pm$ 0.01)	NA
NG-5	0	0.01 ($\pm$ 0.01)	NA	0	0.00 ($\pm$ 0.00)	NA
NG-6	0	0.60 ($\pm$ 0.11)	NA	**4**	**0.81 ($\pm$ 0.20)**	**37619 ($\pm$ 2773)**
NG-7	0	0.99 ($\pm$ 0.01)	NA	0	0.99 ($\pm$ 0.01)	NA
NG-8	**10**	**1.00 ($\pm$ 0.00)**	**319 ($\pm$ 287)**	**10**	**1.00 ($\pm$ 0.00)**	392 ($\pm$ 456)
NG-9	1	0.56 ($\pm$ 0.14)	**12602 ($\pm$ 0)**	**5**	**0.83 ($\pm$ 0.21)**	23236 ($\pm$ 11844)
NG-10	0	0.65 ($\pm$ 0.11)	NA	0	0.55 ($\pm$ 0.08)	NA
Total/mean	23	0.78 ($\pm$ 0.31)		**39**	**0.81 ($\pm$ 0.31)**	

Bold values indicate the best performing approaches (on a specific data set/expression)

equations from the Nguyen benchmarks in at least one of the ten runs. In three cases, the equations are reconstructed in every run. Note also that the successful reconstructions with EDHiE require fewer evaluations of candidate equations than the random sampling approaches.

Table 6 compares EDHiE with PySR, which uses evolutionary operators on expression trees directly (i.e., without embedding them into a latent space), and DSO, that similarly to our approach, combines deep learning with evolutionary optimization. Overall, EDHiE performs better than the other two methods across all metrics[1]: it achieves the highest total number of successful reconstructions. EDHiE has more successful reconstructions for five equations than PySR and less for a single equation, NG-9. The superior performance of EDHiE relative to PySR indicates that evolutionary optimization is more efficient in the latent space than in the space of expression trees. For four equations, EDHiE achieves successful reconstruction more often than DSO. In the two instances of reconstructing NG-3 and NG-4, DSO achieves success twice as often as our method.

Finally, note that the experiments on the Nguyen benchmark were performed on noise-free synthetic data. The results of the experiments on synthetic data with added noise, reported in Appendix 3, show that EDHiE is robust to noise: The increasing noise level has little effect on the reconstruction success rate while significantly increasing the rank of the successfully reconstructed equation in the list of evaluated equations, sorted with respect to increasing RMSE. Appendix 3 also includes additional results on the Nguyen benchmark by random sampling of CVAE, GVAE, and HVAE latent space with varying number of dimensions.

---

[1] PySR does not report the number of evaluated equation. Hence, we could not include them in Table 6.

Table 6 Comparison of the performance of the symbolic regression systems EDHiE, DSO, and PySR on the Nguyen benchmark

Name	EDHiE (our)			DSO (Petersen et al., 2021)			PySR (Cramer, 2023)	
	Successful	Mean R^2	Evaluated	Successful	Mean R^2	Evaluated	Successful	Mean R^2
NG-1	**10**	**1.00 (± 0.00)**	**573 (± 261)**	**10**	**1.00 (± 0.00)**	4565 (± 327)	**10**	1.00 (± 0.00)
NG-2	**10**	**1.00 (± 0.00)**	**5803 (± 4148)**	**10**	**1.00 (± 0.00)**	12,206 (± 9186)	**10**	1.00 (± 0.00)
NG-3	6	1.00 (± 0.01)	20,931 (± 4858)	**10**	**1.00 (± 0.00)**	**8053 (± 3766)**	2	1.00 (± 0.01)
NG-4	3	1.00 (± 0.01)	**21,346 (± 4479)**	**8**	**1.00 (± 0.01)**	32,946 (± 15,613)	0	0.99 (± 0.01)
NG-5	**3**	0.32 (± 0.45)	**20,615 (± 8394)**	0	0.00 (± 0.00)	NA	0	0.16 (± 0.15)
NG-6	**8**	**0.88 (± 0.14)**	**12,772 (± 7923)**	1	0.59 (± 0.15)	49,599 (± 0)	4	0.86 (± 0.13)
NG-7	8	1.00 (± 0.01)	**19,203 (± 3595)**	**10**	**1.00 (± 0.00)**	22,579 (± 10,264)	7	0.99 (± 0.01)
NG-8	**10**	**1.00 (± 0.00)**	**405 (± 174)**	**10**	**1.00 (± 0.00)**	5521 (± 1779)	**10**	**1.00 (± 0.00)**
NG-9	8	0.95 (± 0.15)	**7041 (± 3933)**	2	0.60 (± 0.20)	39,786 (± 28,197)	**10**	**1.00 (± 0.00)**
NG-10	1	0.70 (± 0.17)	**31863 (± 6970)**	0	0.56 (± 0.10)	NA	1	**0.80 (± 0.16)**
Total/mean	**66**	**0.89 (± 0.21)**		61	0.78 (± 0.31)		54	0.88 (± 0.26)

Bold values indicate the best performing approaches (on a specific data set/expression)

Table 7 Results of EDHiE on the 16 equations from the Feynman database that include at most two non-target variables

Name	Successful	Mean R^2	Evaluated
FM-1	10	1.00 (± 0.00)	4311 (± 1914)
FM-2	0	0.98 (± 0.01)	NA
FM-3.1	10	1.00 (± 0.00)	38 (± 37)
FM-3.2	10	1.00 (± 0.00)	53 (± 28)
FM-4.1	10	1.00 (± 0.00)	184 (± 123)
FM-4.2	10	1.00 (± 0.00)	188 (± 204)
FM-5.1	10	1.00 (± 0.00)	63 (± 44)
FM-5.2	10	1.00 (± 0.00)	101 (± 109)
FM-6	0	0.99 (± 0.01)	NA
FM-7.1	10	1.00 (± 0.00)	43 (± 36)
FM-7.2	10	1.00 (± 0.00)	39 (± 39)
FM-8	10	1.00 (± 0.00)	62 (± 45)
FM-9	10	1.00 (± 0.00)	950 (± 72)
FM-10	5	0.99 (± 0.01)	22668 (± 21676)
FM-11	10	1.00 (± 0.00)	62 (± 42)
FM-12	10	1.00 (± 0.00)	924 (± 795)
Total/mean	135	1.00 (± 0.01)	

4.2.4 Results on the Feynman equations

In this section, we evaluate the ability of EDHiE to reconstruct real equations from the domain of physics included in the Feynman database. Table 7 presents the results of symbolic regression on a subset of 16 equations from the database with up to two non-target variables. EDHiE successfully reconstructs 13 equations in all the runs. Most of these equations are simple; thus, EDHiE explores small search spaces comprising less than two hundred evaluated expressions. A more complex equation FM-10 is reconstructed in five out of ten runs exploring more than 20 thousand expressions on average. The equation FM-6 could not be reconstructed in any of the runs since it includes the function arcsin that has not been included in our token library. Finally, EDHiE fails to reconstruct the most complex equation FM-2.

5 Discussion and conclusion

We introduce a novel variational autoencoder for hierarchies, HVAE, that can be efficiently trained to generate valid mathematical expressions represented as expression trees. Compared to generators based on variational autoencoders for sequences, HVAE has three significant advantages. First, it consistently generates valid expressions. Second, its performance is robust even for small training sets: HVAE trained from only two thousand expressions achieves much lower reconstruction error than sequential VAEs trained from 12 thousand examples. Third, the HVAE operating in 32-dimensional latent space has a lower reconstruction error than sequential VAEs with comparable latent spaces.

The ability of HVAE to encode mathematical expressions in a low-dimensional latent space makes it an excellent proxy for exploring the search space of candidate expressions in symbolic regression. Indeed, when performing a random search through the latent space, we achieve comparable performance with a random search through the space of candidate expressions defined by a manually crafted probabilistic grammar. EDHiE, a symbolic regression system that performs evolutionary optimization in the latent space of the HVAE, significantly outperforms methods based on random search and achieves performance comparable to the state-of-the-art symbolic regression system DSO based on a similar combination of evolutionary algorithms and deep learning. The comparison of EDHiE with PySR, a genetic programming approach operating on expression trees directly, shows the benefit of performing evolutionary optimization in the latent space.

HVAE has been used here for symbolic regression, but its potential to efficiently generate and encode hierarchies makes it useful in many different contexts, e.g., generating molecular structures or more general symbolic expressions. Analysis of its performance in these application domains is a promising direction for further research. Moreover, the ability of HVAE to learn from small corpora of expressions might prove helpful in retraining the generator after each generation of the evolutionary search, much like the iterative learning in DSO. This will narrow its focus to generating better expressions, leading to more accurate equations. In general, training the generator on expressions involved in mathematical models that have proved useful in a domain of interest will enable seamless integration and transfer of background knowledge in symbolic regression.

Appendix 1: Grammars and token libraries

In the empirical evaluation of the hierarchical autoencoder, we use several context-free grammars and different token libraries. Grammars used to generate synthetic data sets are probabilistic. Mathematical expressions in the data sets with a name prefix of *AE* include the five common binary arithmetic operators and are generated using the following grammar:

$$S \rightarrow SAF \,[0.4] \mid F \,[0.6]$$
$$A \rightarrow +\,[0.5] \mid -\,[0.5]$$
$$F \rightarrow FBT \,[0.4] \mid T \,[0.6]$$
$$B \rightarrow \cdot\,[0.5] \mid /\,[0.5]$$
$$T \rightarrow (S)\,[0.25] \mid c\,[0.375] \mid x\,[0.375]$$

Data sets with a name prefix *Trig* include, in addition, the trigonometric functions of sine and cosine and are generated using the same grammar as the one above, with different productions for the non-terminal (syntactic category) *T* and a new non-terminal *L*:

$$T \rightarrow (S)\,[0.15] \mid \cos(S)\,[0.05] \mid \sin(S)\,[0.05] \mid L\,[0.75]$$
$$L \rightarrow c\,[0.5] \mid x\,[0.5].$$

While we do not explicitly have the power operator in the grammars to be used during the generation of the data sets, exponentiation (and thus the power operator) can occur during

the simplification of the generated expressions. Because of this, expressions in the data sets also contain the power operator.

In addition, GVAE also needs grammars for generating valid expressions. When applied to the data sets with a name prefix of *AE*, GVAE uses the following grammar:

$$S \rightarrow S + T \mid S - T \mid S \cdot T \mid \frac{S}{T} \mid S^T \mid T$$
$$T \rightarrow (S) \mid x \mid c$$

For the data sets with the name prefix of *Trig*, GVAE uses the same grammar as above with different productions for the non-terminal symbol *T*:

$$T \rightarrow (S) \mid \sin(S) \mid \cos(S) \mid x \mid c.$$

CVAE uses the token library $\{+, -, \cdot, /, \hat{}, x, c, (,), "\}$ for data sets with the name prefix *AE*, and $\{+, -, \cdot, /, \hat{}, \sin, \cos, x, c, (,), "\}$ for data sets with the name prefix *Trig*.

For experiments on the Nguyen benchmark, we use the grammar:

$$E \rightarrow E + F \,[0.2] \mid E - F \,[0.2] \mid F \,[0.6]$$
$$F \rightarrow E \cdot T \,[0.2] \mid E/T \,[0.2] \mid T \,[0.6]$$
$$T \rightarrow V \,[0.4] \mid (E)P \,[0.2] \mid (E) \,[0.2] \mid R(E) \,[0.2]$$
$$V \rightarrow x \,[1.0]$$
$$P \rightarrow {}^{\hat{}2} \,[0.39] \mid {}^{\hat{}3} \,[0.26] \mid {}^{\hat{}4} \,[0.19] \mid {}^{\hat{}5} \,[0.16]$$
$$R \rightarrow \sin \,[0.2] \mid \cos \,[0.2] \mid e\hat{} \,[0.2] \mid \log \,[0.2] \mid \mathrm{sqrt} \,[0.2]$$

for ProGED, GVAE, and to generate training examples for HVAE. HVAE and DSO use the token library $\{x, +, -, \cdot, /, {}^{\hat{}2}, {}^{\hat{}3}, {}^{\hat{}4}, {}^{\hat{}5}, \sin, \cos, \exp, \log, \mathrm{sqrt}\}$, while CVAE uses token $\{(,), "\}$ in addition to the tokens used by HVAE and DSO. For expressions with two non-target variables, we add token y and change the non-terminal symbol *V* to:

$$V \rightarrow x \,[0.5] \mid y \,[0.5].$$

For experiments on the Feynman benchmark, we use the grammar:

$$E \rightarrow E + F \,[0.2] \mid E - F \,[0.2] \mid F \,[0.6]$$
$$F \rightarrow E \cdot T \,[0.2] \mid E/T \,[0.2] \mid T \,[0.6]$$
$$T \rightarrow V \,[0.4] \mid c \,[0.3] \mid A \,[0.3]$$
$$A \rightarrow (E)P \,[0.1] \mid (E) \,[0.55] \mid R(E) \,[0.35]$$
$$V \rightarrow x \,[1.0]$$
$$P \rightarrow {}^{\hat{}2} \,[0.8] \mid {}^{\hat{}3} \,[0.2]$$
$$R \rightarrow \sin \,[0.25] \mid \cos \,[0.25] \mid e\hat{} \,[0.25] \mid \mathrm{sqrt} \,[0.25]$$

to generate training examples for HVAE and tokens $\{x, c, +, -, \cdot, /, {}^{\hat{}2}, {}^{\hat{}3}, \sin, \cos, \exp, \mathrm{sqrt}\}$. For expressions with two non-target variables, we add token y and change the non-terminal symbol *V* to:

$$V \rightarrow x \,[0.5] \mid y \,[0.5].$$

Appendix 2: Additional latent space smoothness results

Additional examples of linear interpolation with HVAE are shown in Table 8. We can see that the space is continuous, as the expressions smoothly transition from Expression 1 to Expression 2. This is best seen in example 1 from the Trig5-15k data set, where in each step, only a few (relevant) symbols change. In the first step, $\sin c$ and x change to c. Then in the next step, $c + c$ and $\sin x$ change to x. In the next step $\frac{x}{c}$ changes to c and x/c changes to $x \cdot \sin c$. In the last step $x + c$ changes to x^c.

While most of the time, expressions change gradually, this is not always the case. This can be best seen in the examples from the AE4-2k data set, where only the expressions at $\alpha = 0.5$ differ from the starting ones.

Since we write expressions as a sequence, it is not always obvious how the underlying expression tree changes during interpolation. Visualization of the gradual change for two pairs of expressions is shown in Fig. 6.

Trees are being transformed by changing their structure and the symbols inside nodes. These transformations are interrelated as changing a constant or a variable into an operator also transforms the structure. This is best seen in the transition between $\alpha = 0$ and $\alpha = 0.25$ for the expression trees at the bottom of Fig. 6. However, the

Table 8 Linear interpolation of examples in the HVAE latent space. The first row shows examples from the AE4-2k data set, the second examples from the AE7-20k data set, and the third from the Trig5-15k data set. Here $n = 4$ and $\alpha = \frac{i}{4}, 0 \leq i \leq 4$

Dataset	α	Example 1	Example 2
	Expression A	$c \cdot (x + c) + \frac{x^c}{c}$	$c - x \cdot c + x$
	$\alpha = 0$	$c \cdot (x + c) + \frac{x^c}{c}$	$c - x \cdot c + x$
	$\alpha = 0.25$	$c \cdot (x + c) + \frac{x^c}{c}$	$c - x \cdot c + x$
AE4-2k	$\alpha = 0.5$	$c \cdot x + \frac{x+c}{c}$	$c - c \cdot x + x$
	$\alpha = 0.75$	$c - x + \frac{x+c}{c}$	$\frac{c + c \cdot x}{c}$
	$\alpha = 1$	$c - x + \frac{x+c}{c}$	$\frac{c + c \cdot x}{c}$
	Expression B	$c - x + \frac{x+c}{c}$	$\frac{c + c \cdot x}{c}$
	Expression A	$\frac{c \cdot x^c + x^c}{c} + c$	$\frac{c \cdot x}{c - x} + c - x + c$
	$\alpha = 0$	$\frac{c \cdot x^c + x^c}{c} + c$	$\frac{c \cdot x}{c - x} + c - x + c$
	$\alpha = 0.25$	$\frac{c + x^c}{c} + c$	$c \cdot (c - x) + c - x + c$
AE7-20k	$\alpha = 0.5$	$\frac{c \cdot x^c}{c} + \frac{x}{c}$	$c \cdot x \cdot c - x + c$
	$\alpha = 0.75$	$c - \frac{x^c}{c}$	$\frac{c \cdot x^c}{c + x} + x$
	$\alpha = 1$	$c - \frac{x^c}{c}$	$\frac{c \cdot x^c}{c + x} + x$
	Expression B	$c - \frac{x^c}{c}$	$\frac{c \cdot x^c}{c + x} + x$
	Expression A	$c + \sin c + \frac{\sin x}{x} - \frac{x}{c}$	$x^c \cdot \cos c + x$
	$\alpha = 0$	$c + \sin c + \frac{\sin x}{x} - \frac{x}{c}$	$x^c \cdot \cos c + x$
	$\alpha = 0.25$	$c + c + \frac{\sin x}{c} - \frac{x}{c}$	$x \cdot \cos c + x$
Trig5-15k	$\alpha = 0.5$	$x + \frac{x}{c} - \frac{x}{c}$	$c \cdot \cos c + \cos x^c$
	$\alpha = 0.75$	$x + c - x \cdot \sin c$	$c + \cos \frac{x^c}{c}$
	$\alpha = 1$	$x^c - x \cdot \sin c$	$c + \cos \frac{x^c}{c}$
	Expression B	$x^c - x \cdot \sin c$	$c + \cos \frac{x^c}{c}$
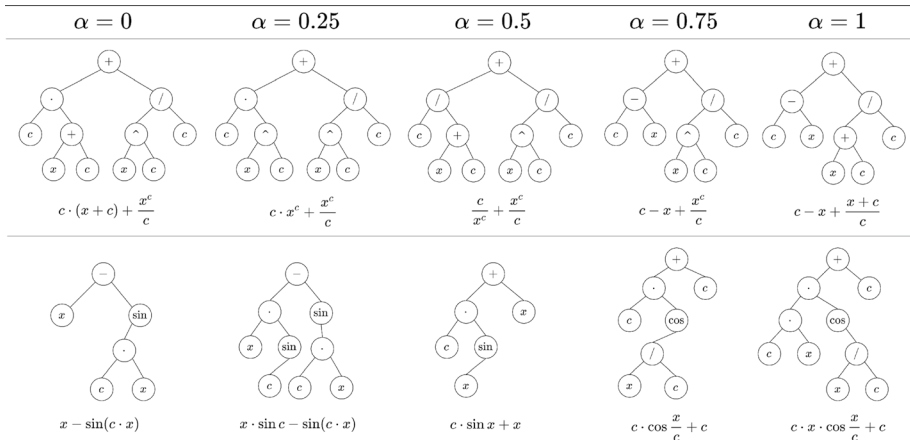
Fig. 6 Examples of linearly interpolated mathematical expressions visualized as expression trees

structure of the expression tree does not always change. This is most noticeable for transitions between $\alpha = 0$ and $\alpha = 0.5$ for expression trees at the top of Fig. 6. Here only an operator changes at each step.

Appendix 3: Additional results on the Nguyen benchmark

In this appendix, we show additional results of the empirical evaluation. First, we show the performance of HVAR on the first eight equations from the Nguyen benchmark with random sampling (By sampling points from the standardized Gaussian distribution) and different dimensions of the latent space in Sect. 1. Next, we show the learning curves of approaches HVAR, ProGED, and EDHiE in Sect. 2. After this, we show the performance of EDHiE on noisy data in Sect. 3. Finally, we conclude this section in 4 by showing the performance of CVAE with random sampling and GVAE with both random sampling and an evolutionary algorithm.

A3.1 Dimensions

Table 9 shows the results of evaluation using HVAE with different dimensions of the latent vector space. Here new expressions are generated by randomly sampling points from the standardized Gaussian distribution and decoding them. We can see overall, models, where the dimension of the latent vector space is either 16 or 32, perform the best. HVAE 32 produces expressions that usually have a slightly higher mean R^2, while HVAE 16 usually needs to evaluate less unique expressions to generate the desired one. Because we prefer the number of successful runs and the mean R^2 metrics, we select the model with the latent vector space dimension 32 for experiments where the HVAE model is coupled together with evolutionary algorithms.

Table 9 The performance of HVAR (number of successful reconstructions, R^2, and number of evaluated equations) with varying number of latent dimensions on the Nguyen benchmark

Name	HVAR 16			HVAR 32			HVAR 64			HVAR 128		
	Successful	Mean R^2	Evaluated	Successful	Mean R^2	Evaluated	Successful	Mean R^2	Evaluated	Successful	Mean R^2	Evaluated
NG-1	10	1.00 (± 0.00)	500 (± 378)	10	1.00 (± 0.00)	901 (± 1332)	10	1.00 (± 0.00)	1985 (± 2270)	10	1.00 (± 0.00)	1544 (± 1523)
NG-2	10	1.00 (± 0.00)	4407 (± 2904)	10	1.00 (± 0.00)	9729 (± 5337)	5	1.00 (± 0.01)	14435 (± 8279)	9	1.00 (± 0.01)	14261 (± 13074)
NG-3	1	1.00 (± 0.01)	16,595 (± 0)	0	1.00 (± 0.01)	NA	0	1.00 (± 0.01)	NA	1	1.00 (± 0.01)	24562 (± 0)
NG-4	0	1.00 (± 0.01)	NA	0	1.00 (± 0.01)	NA	0	1.00 (± 0.01)	NA	0	1.00 (± 0.01)	NA
NG-5	0	0.00 (± 0.00)	NA	0	0.00 (± 0.01)	NA	0	0.01 (± 0.01)	NA	0	0.04 (± 0.07)	NA
NG-6	3	0.75 (± 0.18)	13,345 (± 425)	4	0.81 (± 0.13)	37619 (± 2773)	4	0.74 (± 0.22)	26169 (± 12187)	0	0.57 (± 0.09)	NA
NG-7	0	1.00 (± 0.01)	NA	0	0.99 (± 0.01)	NA	0	1.00 (± 0.01)	NA	0	1.00 (± 0.01)	NA
NG-8	10	1.00 (± 0.00)	345 (± 276)	10	1.00 (± 0.00)	392 (± 456)	10	1.00 (± 0.00)	233 (± 171)	10	1.00 (± 0.00)	273 (± 205)
Total/mean	34	0.84 (± 0.33)		**34**	**0.85 (± 0.32)**		29	0.84 (± 0.33)		30	0.83 (± 0.33)	

(a) NG-1

(b) NG-5

(c) NG-6

(d) NG-8

(e) NG-9

(f) NG-10
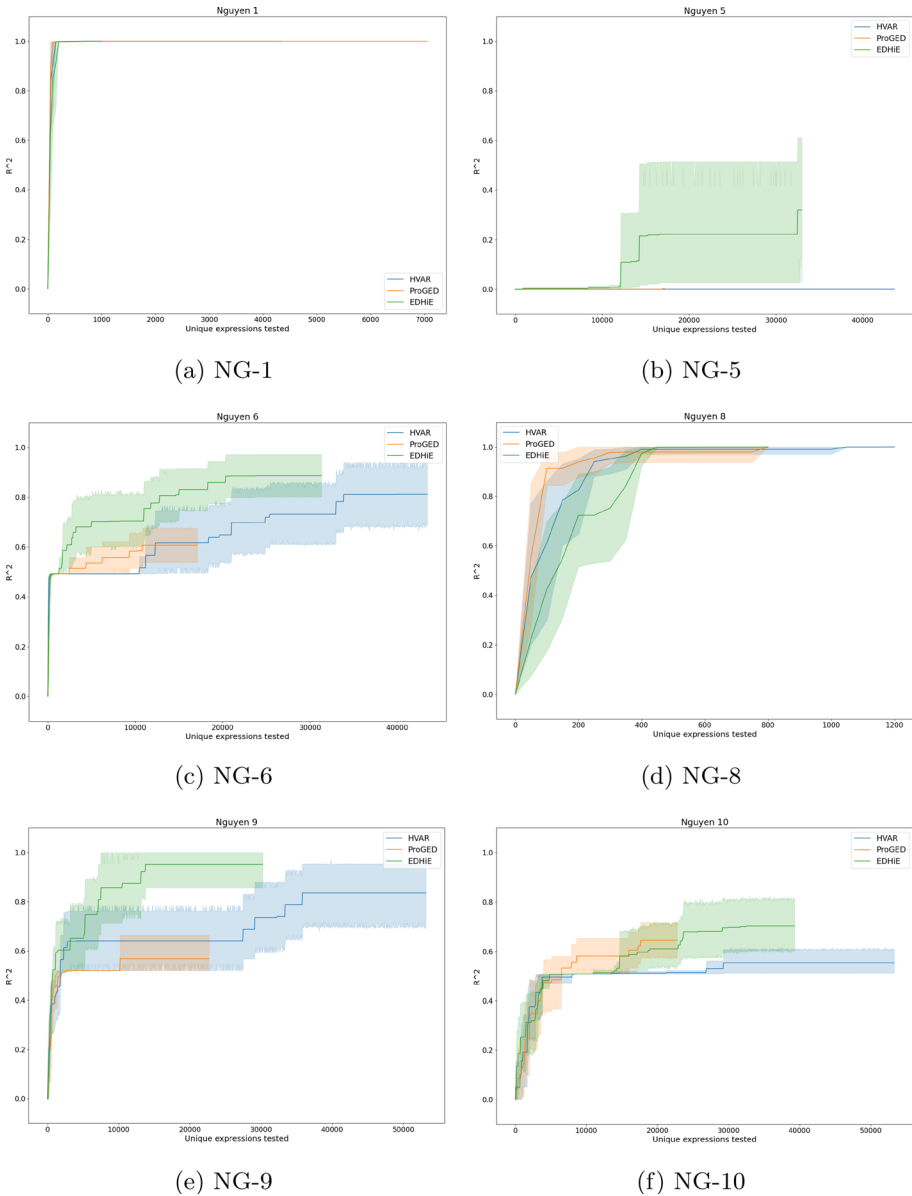
Fig. 7 Learning curves for HVAR, ProGED, and EDHiE on the selected equations from the Nguyen benchmark. Curves for the equations *NG-2*, *NG-3*, *NG-4*, and *NG-7* are omitted since they resemble the curve for the *NG-1* equation

A3.2 Learning curves

Optimization algorithms continually enhance solutions over time by iteratively exploring the input space to minimize an objective function or maximize performance. The learning curve serves as a valuable measure for evaluating algorithmic performance,

illustrating how the chosen metric evolves as optimization progresses. Steep improvements in the learning curve indicate rapid convergence towards better solutions, while plateaus or slow convergence suggest challenges in finding superior solutions. By analyzing the learning curve, we can be gain an insight into the algorithm's effectiveness, convergence, stability, and potential for further improvement.

Figure 7 shows the learning curves of HVAR, ProGED, and EDHiE on six equations from the Nguyen benchmark. We can see that overall EDHiE performs the best as it achieves the highest R^2 score and needs to test the least expression to do so. This is not true on equation NG-8, where the other two approaches find the desired expressions quicker. This happens because all approaches find the desired expression quickly and the evolution part of EDHiE does not yet come into effect.

A3.3 Robustness to noise

In practical scenarios, working with noisy data is common, making it crucial for symbolic regression approaches to perform well in the presence of noise. The performance of EDHiE on noisy data is demonstrated in Table 10. To generate noisy data sets, we sample values ϵ from a Gaussian distribution $\mathcal{N}(0, I)$ and add them to the target values y using the formula $\tilde{y} = y \cdot (1 + \eta\epsilon)$, where η represents the noise level.

To evaluate our approach on noisy data, we employ two metrics: the number of successful runs and the mean rank. We execute our approach on noisy data, rank all the generated expressions based on their RMSE, and evaluate these expressions using noiseless data. A run is considered successful if we find an expression that achieves an RMSE below 10^{-10} on noiseless data. In the case of a successful run, we record the rank of the first expression with an RMSE below 10^{-10} and use it to calculate the mean rank.

The results demonstrate that the number of successful runs remains relatively consistent across different noise levelsm indicating the robustness of our approach. Additionally, we can see that the mean rank increases as the amount of noise rises. This outcome is expected, as higher noise levels allow more expressions to overfit the noisy data. In practical applications, expressions that overfit can be eliminated by assigning complexity scores to each expression and selecting less complex expressions from the Pareto front.

A3.4 Performance of CVAE and GVAE on symbolic regression

Table 11 shows the results of the CVAE baseline. Models presented in the table use the same parameters as they do in Sect. 4.1.1 apart from the dimension of the latent vector space. We can see that this baseline performs very poorly, as it finds only the two simplest equations. The main reason for this is the high number of invalid expressions (more than 96.8%) the baseline produces.

Lastly, Table 12 shows the results of the GVAE baseline. We can see that GVAE performs better than CVAE but worse than HVAE. For the GVAE Evo approach we use the GVAE baseline with the latent space dimension 64 together with the evolutionary operators presented in Sect. 3.3.2. Here, different models find different equations: GVAE 32 finds the equation *NG-6*, while GVAE 64 finds *NG-4*, and GVAE Evo *NG-3*. Overall GVAE Evo performs the best as it successfully finishes 4 runs more than other models.

Table 10 The performance of EDHiE with varying level of noise added to synthetic data from the Nguyen benchmark

η	0.01		0.02		0.05		0.1		0.2	
Name	Successful	Mean rank	Successful	Mean rank	Successful	Mean rank	Successful	Mean rank	Successful	Mean rank
NG-1	10	7.30 (6.74)	10	6.00 (6.20)	10	9.00 (8.04)	10	14.60 (16.27)	10	23.80 (13.91)
NG-2	10	1.10 (0.30)	10	1.20 (0.40)	10	6.80 (3.09)	10	175.10 (42.15)	10	222.80 (28.72)
NG-3	5	39.40 (4.41)	3	32.67 (20.98)	3	34.33 (9.81)	2	50.50 (1.50)	3	169.33 (25.85)
NG-4	2	85.00 (38.00)	4	57.00 (6.16)	2	52.50 (6.50)	0	NA	2	210.50 (4.50)
NG-5	4	1.00 (0.00)	1	1.00 (0.00)	2	1.00 (0.00)	4	1.00 (0.00)	5	1.00 (0.00)
NG-6	6	1.00 (0.00)	6	1.00 (0.00)	6	1.00 (0.00)	9	1.00 (0.00)	7	1.00 (0.00)
NG-7	10	1.00 (0.00)	8	1.00 (0.00)	9	1.00 (0.00)	9	1.11 (0.31)	8	10.00 (3.46)
NG-8	10	1.00 (0.00)	10	1.50 (1.50)	10	1.70 (1.19)	10	3.30 (3.44)	10	5.10 (4.39)
NG-9	9	1.00 (0.00)	7	1.00 (0.00)	9	1.00 (0.00)	8	1.00 (0.00)	8	1.00 (0.00)
NG-10	1	1.00 (0.00)	0	NA	1	1.00 (0.00)	1	1.00 (0.00)	1	1.00 (0.00)
Total	67	7.33 (18.38)	59	7.37 (16.10)	62	6.61 (11.91)	63	32.75 (65.10)	64	55.42 (87.83)

Table 11 Results of symbolic regression by random sampling of the CVAE latent space with varying number of dimensions on the Nguyen benchark

	CVAE 32			CVAE 64			CVAE 128		
Name	Successful	Mean R^2	Invalid	Successful	Mean R^2	Invalid	Successful	Mean R^2	Invalid
NG-1	4	1.00 (± 0.01)	96,851 (± 57)	2	1.00 (± 0.01)	98,131 (± 57)	2	1.00 (± 0.01)	99,243 (± 25)
NG-2	0	1.00 (± 0.01)	96,816 (± 38)	0	1.00 (± 0.01)	98,143 (± 38)	0	1.00 (± 0.01)	99,247 (± 18)
NG-3	0	0.99 (± 0.01)	96,857 (± 59)	0	0.99 (± 0.01)	98,131 (± 47)	0	0.97 (± 0.07)	99,250 (± 22)
NG-4	0	0.99 (± 0.01)	96,867 (± 39)	0	0.99 (± 0.01)	98,151 (± 60)	0	0.99 (± 0.01)	99,238 (± 17)
NG-5	0	0.00 (± 0.00)	96,846 (± 46)	0	0.00 (± 0.00)	98,148 (± 46)	0	0.00 (± 0.00)	99,239 (± 18)
NG-6	0	0.49 (± 0.01)	96,843 (± 55)	0	0.49 (± 0.02)	98,127 (± 43)	0	0.09 (± 0.15)	99,227 (± 24)
NG-7	0	0.71 (± 0.29)	96,837 (± 49)	0	0.60 (± 0.32)	98,108 (± 27)	0	0.14 (± 0.28)	99,249 (± 31)
NG-8	10	1.00 (± 0.00)	96,801 (± 48)	10	1.00 (± 0.00)	98,146 (± 39)	3	0.58 (± 0.42)	99,229 (± 34)
Total/mean	**14**	**0.77 (± 0.34)**	**96,839 (± 20)**	12	0.75 (± 0.34)	98,135 (± 13)	5	0.59 (± 0.42)	99240 (± 8)

Bold values indicate the best performing approaches (on a specific data set/expression)

Table 12 Results of symbolic regression by random sampling and evolutionary optimization in the GVAE latent space with varying number of dimensions on the Nguyen benchark

Name	GVAE 32			GVAE 64			GVAE 128			GVAE Evo		
	Successful	Mean R^2	Invalid	Successful	Mean R^2	Evaluated	Successful	Mean R^2	Invalid	Successful	Mean R^2	Invalid
NG-1	10	1.00 (± 0.00)	15,972 (± 95)	9	1.00 (± 0.01)	58,229 (± 149)	10	1.00 (± 0.00)	72,863 (± 96)	10	1.00 (± 0.00)	0 (± 1)
NG-2	2	1.00 (± 0.01)	15,917 (± 111)	4	1.00 (± 0.01)	58,295 (± 150)	3	1.00 (± 0.01)	72,840 (± 110)	7	1.00 (± 0.01)	0 (± 1)
NG-3	0	1.00 (± 0.01)	15,985 (± 107)	0	1.00 (± 0.01)	58,383 (± 153)	0	1.00 (± 0.01)	72,802 (± 101)	1	1.00 (± 0.01)	1 (± 0)
NG-4	0	1.00 (± 0.01)	15,900 (± 115)	1	1.00 (± 0.01)	58,318 (± 169)	0	1.00 (± 0.01)	72,848 (± 105)	0	1.00 (± 0.01)	0 (± 0)
NG-5	0	0.00 (± 0.01)	15,896 (± 102)	0	0.00 (± 0.01)	58,322 (± 137)	0	0.00 (± 0.00)	72,808 (± 97)	0	0.00 (± 0.01)	0 (± 1)
NG-6	2	0.59 (± 0.20)	15,925 (± 120)	0	0.49 (± 0.00)	58,327 (± 162)	0	0.49 (± 0.00)	72,787 (± 183)	0	0.53 (± 0.09)	1 (± 1)
NG-7	0	0.92 (± 0.02)	15,987 (± 123)	0	0.92 (± 0.00)	58,268 (± 107)	0	0.91 (± 0.03)	72,865 (± 116)	0	0.92 (± 0.00)	0 (± 1)
NG-8	10	1.00 (± 0.00)	15,986 (± 132)	10	1.00 (± 0.00)	58,252 (± 157)	10	1.00 (± 0.00)	72,825 (± 186)	10	1.00 (± 0.00)	0 (± 1)
Total/mean	24	0.81 (± 0.33)	15,946 (± 37)	24	0.81 (± 0.34)	58,299 (± 45)	23	0.80 (± 0.34)	72,830 (± 27)	28	0.81 (± 0.34)	0 (± 0)

Appendix 4: Implementation details

This section provides implementation details that are not part of the methodology but are crucial for the reproducibility of our approach and its implementation. These details include batching, encoding/decoding, and training.

A4.1 Batching

Since our approach works on expression trees with varying structures, we cannot employ standard batching methods. Instead, we represent a batch of expression trees with a Python object we refer to as a "batched node". A batched node contains a list of symbols, a left (batched) child node, and a right (batched) child node.

Since an expression tree may not contain all the nodes in the batched tree, the list of symbols within a batched node might include an empty string as a placeholder for the missing symbol. During training, each batched node also contains a target matrix, a prediction matrix, and a mask vector. The target matrix comprises one-hot symbol encodings, with empty strings represented by zero values. The decoding cell generates the prediction matrix, which, in turn, predicts the target matrix. Finally, the mask vector is a binary vector where the value at index i equals one if the node appears in the expression tree i and equals zero otherwise.

A4.2 Encoding/decoding

The encoding process involves traversing the batched node using a post-order traversal. We first visit and encode the left (batched) child, followed by the right (batched) child. Finally, we generate the code for the batched node using codes obtained from the child nodes.

Decoding is performed in reverse. We start by decoding the (batched) root code, which yields a list of symbols, the codes for the left and right (batched) child nodes, and the masks for the left and right child nodes. The masks for the left and right child nodes are calculated using symbols produced by the decoding cell and the mask of the current node. Specifically, if the value of the current mask at a given position is 0, the corresponding values in the mask for the left and right child nodes remain 0. Otherwise, the masks are assigned values appropriate to the symbol type: both 1 for an operator, 1, and right 0 for a function, and both 0 for a variable or constant. When all values in a mask become 0, the decoding process for this branch terminates.

A4.3 Training

Expression trees have two components: the binary tree structure and the symbols within the nodes. However, the structure of the binary tree can be inferred from the symbols present in the nodes. Therefore, it is sufficient for our approach to learn to reconstruct the symbols occurring in the nodes.

When training the model, we restrict the output tree's structure to match the input structure. Specifically, we utilize the input batched node and incorporate a prediction matrix into each node. We then calculate the reconstruction error using the target and prediction sequences obtained through an in-order traversal of the batched node. While

computing the cross-entropy loss, we apply a masking technique to exclude nodes that do not occur in an expression tree from the loss calculation, effectively removing them from the training process.

Author contributions Conceptualization: SM, LT; methodology: SM, LT; writing—original draft preparation: SM, LT; writing—review and editing: SM, LT, SD; funding acquisition: SD; resources: SM; supervision: LT, SM; software: SM; Visualization: SM; data curation: SM; Investigation:SM; validation: SM.

Availability of data and materials All data used in this work is available in repositories https://github.com/smeznar/HVAE.

Code availability The implementation of HVAE and EDHiE and the scripts needed for performing their evaluation, presented in this article, can be found at https://github.com/smeznar/HVAE.

Declarations

Conflict of interest Not applicable.

Ethics approval Not applicable.

Consent to participate Not applicable.

Consent for publication Not applicable.

References

Allamanis, M., Chanthirasegaran, P., Kohli, P., & Sutton, C. (2017). Learning continuous semantic representations of symbolic expressions. In *Proceedings of the 34th international conference on machine learning* (Vol. 70, pp. 80–88). JMLR.org

Alvarez-Melis, D., & Jaakkola, T. (2017). Tree-structured decoding with doubly-recurrent neural networks. In *ICLR*.

Bahuleyan, H. (2018). Natural language generation with neural variational models.

Bielik, P., Raychev, V., & Vechev, M. (2016). PHOG: Probabilistic model for code. In Balcan, M. F., & Weinberger, K. Q. (eds.), *Proceedings of the thirty-third international conference on machine learning. Proceedings of machine learning research*, New York, New York, USA (Vol. 48, pp. 2933–2942).

Biggio, L., Bendinelli, T., Neitz, A., Lucchi, A., & Parascandolo, G. (2021). Neural symbolic regression that scales. In *Proceedings of 38th international conference on machine learning (ICML). Proceedings of*

machine learning research (Vol. 139, pp. 936–945). PMLR. https://proceedings.mlr.press/v139/biggio21a.html

Blank, J., & Deb, K. (2020). pymoo: Multi-objective optimization in python. *IEEE Access, 8*, 89497–89509.

Brence, J., Todorovski, L., & Džeroski, S. (2021). Probabilistic grammars for equation discovery. *Knowledge-Based Systems, 224*, 107077. https://doi.org/10.1016/j.knosys.2021.107077

Bridewell, W., Langley, P., Todorovski, L., & Džeroski, S. (2008). Inductive process modeling. *Machine Learning, 71*(1), 1–32. https://doi.org/10.1007/s10994-007-5042-6

Cho, K., van Merrienboer, B., Bahdanau, D., & Bengio, Y. (2014a). On the properties of neural machine translation: Encoder–decoder approaches. https://doi.org/10.48550/ARXIV.1409.1259

Cho, K., van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., & Bengio, Y. (2014b). Learning phrase representations using RNN encoder–decoder for statistical machine translation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)* (pp. 1724–1734). Association for Computational Linguistics, Doha, Qatar. https://doi.org/10.3115/v1/D14-1179

Cranmer, M. (2023). Interpretable machine learning for science with PySR and SymbolicRegression.jl

Dai, H., Tian, Y., Dai, B., Skiena, S., & Song, L. (2018). Syntax-directed variational autoencoder for structured data

d'Ascoli, S., Kamienny, P.-A., Lample, G., & Charton, F. (2022). Deep symbolic regression for recurrent sequences. arXiv preprint arXiv:2201.04600

Gómez-Bombarelli, R., Wei, J. N., Duvenaud, D., Hernández-Lobato, J. M., Sánchez-Lengeling, B., Sheberla, D., Aguilera-Iparraguirre, J., Hirzel, T. D., Adams, R. P., & Aspuru-Guzik, A. (2018). Automatic chemical design using a data-driven continuous representation of molecules. *ACS Central Science, 4*(2), 268–276. https://doi.org/10.1021/acscentsci.7b00572

Guimerà, R., Reichardt, I., Aguilar-Mogas, A., Massucci, F. A., Miranda, M., Pallarès, J., & Sales-Pardo, M. (2020). A Bayesian machine scientist to aid in the solution of challenging scientific problems. *Science Advances, 6*(5), 6971. https://doi.org/10.1126/sciadv.aav6971

Janz, D., van der Westhuizen, J., Paige, B., Kusner, M. J., & Hernández-Lobato, J. M. (2017). Learning a generative model for validity in complex discrete structures. In *The sixth international conference on learning representations*. https://doi.org/10.48550/ARXIV.1712.01664

Jin, W., Barzilay, R., & Jaakkola, T. (2021). Junction tree variational autoencoder for molecular graph generation. In *Artificial intelligence in drug discovery* (pp. 228–249). The Royal Society of Chemistry. https://doi.org/10.1039/9781788016841-00228

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

Kingma, D. P., & Ba, J. (2015). Adam: A method for stochastic optimization. In Y. Bengio, Y. LeCun (eds.), *3rd international conference on learning representations, ICLR 2015, San Diego, CA, USA, May 7–9, 2015, conference track proceedings*.

Kingma, D. P., & Welling, M. (2014) Auto-encoding variational Bayes. In 2nd international conference on learning representations, ICLR 2014, Banff, AB, Canada, April 14–16, 2014, Conference Track Proceedings.

Koza, J. R. (1994). Genetic programming as a means for programming computers by natural selection. *Statistics and Computing, 4*(2), 87–112. https://doi.org/10.1007/BF00175355

Kullback, S., & Leibler, R. A. (1951). On information and sufficiency. *The Annals of Mathematical Statistics, 22*(1), 79–86. https://doi.org/10.1214/aoms/1177729694

Kusner, M. J., & Hernández-Lobato, J. M. (2016). GANS for sequences of discrete elements with the Gumbel-softmax distribution. arxiv:1611.04051

Kusner, M. J., Paige, B., Hernández-Lobato, J. M. (2017). Grammar variational autoencoder. In: Proceedings of the 34th international conference on machine learning—volume 70. ICML'17, pp. 1945–1954. JMLR.org. arxiv:1703.01925

Levenshtein, V. I. (1965). Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady, 10*, 707–710.

Martius, G., & Lampert, C. H. (2016). Extrapolation and learning equations.

Meurer, A., Smith, C. P., Paprocki, M., Čertík, O., Kirpichev, S. B., Rocklin, M., Kumar, A., Ivanov, S., Moore, J. K., Singh, S., Rathnayake, T., Vig, S., Granger, B. E., Muller, R. P., Bonazzi, F., Gupta, H., Vats, S., Johansson, F., Pedregosa, F., … Scopatz, A. (2017). Sympy: Symbolic computing in python. *PeerJ Computer Science, 3*, 103. https://doi.org/10.7717/peerj-cs.103

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

Petersen, B. K., Landajuela, M., Mundhenk, T. N., Santiago, C. P., Kim, S. K., & Kim, J. T. (2021). Deep symbolic regression: Recovering mathematical expressions from data via risk-seeking policy gradients. In *Proceedings of the international conference on learning representations*.

Radinja, M., Škerjanec, M., Šraj, M., Džeroski, S., Todorovski, L., & Atanasova, N. (2021). Automated modelling of urban runoff based on domain knowledge and equation discovery. *Journal of Hydrology, 603*, 127077. https://doi.org/10.1016/j.jhydrol.2021.127077

Schmidt, M., & Lipson, H. (2009). Distilling free-form natural laws from experimental data. *Science, 324*(5923), 81–85. https://doi.org/10.1126/science.1165893

Sherstinsky, A. (2020). Fundamentals of Recurrent Neural Network (RNN) and Long Short-Term Memory (LSTM) network. *Physica D: Nonlinear Phenomena, 404*, 132306. https://doi.org/10.1016/j.physd.2019.132306

Simidjievski, N., Todorovski, L., Kocijan, J., & Dzeroski, S. (2020). Equation discovery for nonlinear system identification. *IEEE Access, 8*, 29930–29943. https://doi.org/10.1109/access.2020.2972076

Sutskever, I., Vinyals, O., & Le, Q. V. (2014). Sequence to sequence learning with neural networks. In *Proceedings of the 27th international conference on neural information processing systems—volume 2. NIPS'14* (pp. 3104–3112). MIT Press

Tai, K. S., Socher, R., & Manning, C. D. (2015). Improved semantic representations from tree-structured long short-term memory networks. In *Proceedings of the 53rd annual meeting of the Association for Computational Linguistics and the 7th international joint conference on natural language processing* (Vol. 1: Long Papers, pp. 1556–1566). Association for Computational Linguistics, Beijing, China. https://doi.org/10.3115/v1/P15-1150

Todorovski, L., & Dzeroski, S. (1997). Declarative bias in equation discovery. In *Proceedings of the fourteenth international conference on machine learning. ICML '97* (pp. 376–384). Morgan Kaufmann Publishers Inc.

Todorovski, L. (2017). Equation discovery. In C. Sammut & G. I. Webb (Eds.), *Encyclopedia of machine learning and data mining, Chap. 5* (pp. 410–414). Springer. https://doi.org/10.1007/978-1-4899-7687-1_258

Udrescu, S.-M., & Tegmark, M. (2020). AI Feynman: A physics-inspired method for symbolic regression. *Science Advances, 6*(16), 2631.

Uy, N. Q., Hoai, N. X., O'Neill, M., McKay, R. I., & Galván-López, E. (2011). Semantically-based crossover in genetic programming: Application to real-valued symbolic regression. *Genetic Programming and Evolvable Machines, 12*(2), 91–119.