Symbolic Regression for Data Storage with Side Information

Xiangwu Zuo
CSE Dept.
Texas A&M Univ.
dkflame@tamu.edu

Anxiao (Andrew) Jiang
CSE Dept.
Texas A&M Univ.
ajiang@cse.tamu.edu

Netanel Raviv
CSE Dept.
Washington Univ. in St. Louis
netanel.raviv@wustl.edu

Paul H. Siegel
ECE Dept.
Univ. of California, San Diego
psiegel@ucsd.edu

2022 IEEE Information Theory Workshop (ITW) | 978-1-6654-8341-4/22/$31.00 ©2022 IEEE | DOI: 10.1109/ITW54588.2022.9965879

***Abstract*—There are various ways to use machine learning to improve data storage techniques. In this paper, we introduce symbolic regression, a machine-learning method for recovering the symbolic form of a function from its samples. We present a new symbolic regression scheme that utilizes side information for higher accuracy and speed in function recovery. The scheme enhances latest results on symbolic regression that were based on recurrent neural networks and genetic programming. The scheme is tested on a new benchmark of functions for data storage.**

***Index Terms*—symbolic regression, data storage, side information, deep learning, genetic programming**

I. Introduction

There are various approaches of using machine learning to improve data storage techniques [5] [9] [13] [22] [27]. In this work, we introduce symbolic regression, a machine-learning method for recovering the symbolic form of a function from its samples. Specifically, let $\mathbf{x} = (x_1, x_2, \cdots, x_n) \in \mathbb{R}^n$ be n variables. Let $y = f(\mathbf{x}) \in \mathbb{R}$ be a function of $\mathbf{x}$. Let $S = \{(X_i, y_i) \mid i = 1, 2, \cdots, m\}$ be m samples of the function f, where each point $X_i \in \mathbb{R}^n$ is a sample of $\mathbf{x}$, and $y_i = f(X_i)$ is the corresponding value of y. Given such a dataset of samples S, the goal of *symbolic regression* is to recover the symbolic form of the function f (such as $y = -x_1 \log_2 x_1 - (1-x_1)\log_2(1-x_1)$ or $y = \sqrt{(x_1 - x_2)^2 + (x_3 - x_4)^2}$). The recovered function should not only fit the samples in S well, but also be as simple as possible.

Symbolic regression can have numerous applications to data storage. One application is modeling physical storage devices, such as non-volatile flash memories, where it can provide functional insights into evolving cell threshold voltage distributions, spatial inter-cell interference effects, and page-level bit error counts as the memory ages over read/write cycles or loses charge with infrequent use. Symbolic regression can bridge experimental data with theoretical models. Another application is aiding the analysis of storage schemes, such as modeling the performance of error-correcting codes, wear-leveling schemes, etc. Here it can bridge simulation data with theoretical models and provide insights for further analysis.

Symbolic regression is a challenging problem in AI. Its solution space is both discrete (in the symbolic form of the functions) and continuous (in the coefficients of the functions). The number of functions in the search space grows exponentially as the lengths of the functions increase. Existing symbolic-regression methods mainly use evolutionary algorithms or deep learning [2] [3] [4] [6] [7] [10] [11] [12] [14] [18] [19] [21] [23] [24] [25] [26]. In particular, a recent method [16] (which we shall call DSR-GP) improves the Deep Symbolic Regression (DSR) approach [17] and achieves state-of-the-art performance by combining deep neural networks with genetic programming.

In this work, we introduce a new symbolic regression approach that further improves DSR-GP by combining it with *side information*. Broadly speaking, side information can refer to any information that correlates with the ground-truth function f, although in this paper, we shall focus on a narrower type: functions that resemble all or a part of f. (Note that side information known as "prior knowledge" or "expert knowledge" has been explored in genetic programming for symbolic regression [14] [20].) We show that the new scheme, which we shall call DSR-GP-SI, can notably improve the performance of symbolic regression.

To help evaluate the potential of symbolic regression for data storage, we also present a new benchmark of functions that focus on storage systems. The new benchmark, along with existing benchmarks [16] [25], can be used to compare the performance of different symbolic regressions algorithms.

The rest of the paper is organized as follows. In Section II, we introduce representations of symbolic functions and their side information. In Section III, we present our symbolic-regression scheme using side information. In Section IV, we evaluate the scheme's performance on benchmark functions, including our new benchmark on data storage.

II. Side Information for Symbolic Regression

In this section, we first introduce representations of symbolic functions, including computation trees and the pre-order representation. We then discuss side information for symbolic regression, including sub-functions.

A. Pre-order Representation of Symbolic Functions

To represent symbolic functions, we need a set of operators S_{op}, a set of variables S_{var}, and a set of values for coefficients S_{coeff}. For example, if $S_{op} = \{+, -, \times, \div, \sin, \cos, \log, \exp\}$, $S_{var} = \{x_1, x_2\}$ and $S_{coeff} = \mathbb{R}$, then $\sin(x_1 - x_2) + 1.5\exp(x_1)$ is a valid symbolic function. If we wish to represent all coefficients using a placeholder "const", we can let $S_{coeff} = \{\text{const}\}$. The three sets – S_{op}, S_{var} and S_{coeff} – together form the *token library*.

A symbolic function can be expressed as a *computation tree*, where every internal node is an operator and every leaf is either a variable or a coefficient. A computation tree for $f(x_1, x_2) = \frac{-x_1}{x_1+x_2} \log x_2 - \sin(2x_1)\cos x_2$ is illustrated in Fig. 1 (a).

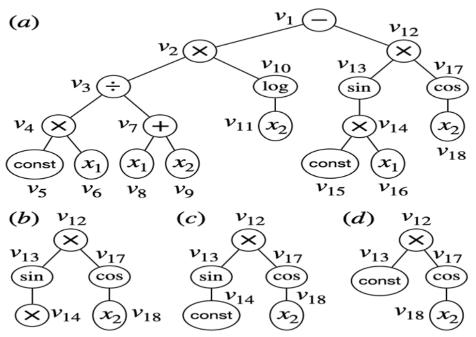

Fig. 1. Let $f(x_1, x_2) = \frac{-x_1}{x_1+x_2} \log x_2 - \sin(2x_1)\cos x_2$. (a) A computation tree $G = (V, E)$ for $f(x_1, x_2)$, where the constant coefficients – -1 and 2 – are represented by the placeholder token "const". Here v_1, v_2, $\cdots$, v_{18} are the pre-order traversal of G. (b) Let $r = v_{12}$ and $\mathcal{L} = \{\ell_1 = v_{14}\}$. We get the graph here after removing the descendants of ℓ_1. (c) Replace the token of ℓ_1 by the constant token "const". (d) Simplify the subtree by removing v_{14} and changing the token of v_{13} to "const" (because $\sin(\text{const})$ is also a constant), and we get the subtree (i.e., the sub-function) $\tilde{G}_{r,\mathcal{L}}$.

A particularly useful way to traverse a computation tree is the *pre-order traversal*, which is a specific implementation of depth-first search (DFS): we visit nodes in the tree using DFS; and if an internal node has multiple children, those children are visited from left to right. For the computation tree shown in Fig. 1 (a), its pre-order traversal is v_1, v_2, $\cdots$, v_{18}, whose corresponding sequence of tokens (from the token library) is "$- \times \div \times$ const $x_1 + x_1\ x_2$ log $x_2 \times$ sin $\times$ const x_1 cos x_2" and forms another representation of the symbolic function. Such a sequence of tokens shall be called the *pre-order representation* of the symbolic function. It is known that there is a one-to-one mapping between a computation tree and its pre-order representation, because the number of operands that an operator can take (which equals its number of children in the tree) is known in advance. (For example, the "+" operator takes two operands, while the "sin" operator takes one operand.)

B. Side Information and Sub-functions

Let $y = f(\mathbf{x})$ be the ground-truth function we look for. Current symbolic regression methods usually search for f in the vast function space without prior knowledge. [16] [17] There exist, however, various alternative ways to build mathematical models based on the given samples, which can provide useful candidate solutions to (or at least hints on) the function f. In many fields of science and engineering (e.g., physics, economics, information theory), people often model data by functions based on prior knowledge or experience (e.g., use an exponential function or a power-law function as a component to model the decaying tails of a bell-shaped distribution, use trigonometric functions as a component to model oscillating data, or use rational functions to model the solution to approximately linear equations). People can also make conjectures on the function f by analyzing the data samples (e.g., by analyzing their frequency spectrum). Such candidate functions, which may be an approximation of the ground-truth function f or just a part of f, can serve as useful *side information* for symbolic regression methods and help them search for f in a more targeted space, thus improving both the accuracy and the speed of symbolic regression.

Broadly speaking, any information correlated with the function f can be used as *side information*. In this work, however, we shall focus on functions that are a part of the ground-truth function f, and call such functions *sub-functions* or *sub-function side information (SF-SI)*. More specifically, let $G = (V, E)$ be a computation tree for f. A sub-function of G is defined as follows:

1) Let $r \in V$ be an internal node (i.e., a non-leaf node) in the tree G.
2) Let $\mathcal{L} = \{\ell_1, \ell_2, \cdots, \ell_{|\mathcal{L}|}\} \subset V$ be $|\mathcal{L}|$ distinct descendants of r, such that $\forall\ 1 \le i < j \le |\mathcal{L}|$, ℓ_i is neither an ancestor nor a descendant of ℓ_j.
3) Let G_r be the subtree of G rooted at node r. Let $\tilde{G}_{r,\mathcal{L}}$ denote the computation tree obtained from G_r as follows: (1) remove from G_r every node that is a descendant of any node in $\mathcal{L}$; (2) replace the token (i.e., an operator, variable or constant) of each node in $\mathcal{L}$ by the constant token "const"; (3) simplify the subtree as follows: for any internal node v in the subtree (which has an operator as its token), if all its children have "const" as their tokens, we remove all the descendants of v from the subtree, and change the token of v to "const"; we do the above repeatedly until the subtree cannot be simplified any further. Then, the obtained subtree $\tilde{G}_{r,\mathcal{L}}$ is a sub-function of G.

Given a computation tree G, let $L(G)$ denote the number of nodes in G. One way to measure the similarity between G and its sub-function $\tilde{G}_{r,\mathcal{L}}$ is

$$\gamma(G, \tilde{G}_{r,\mathcal{L}}) \triangleq \frac{L(\tilde{G}_{r,\mathcal{L}})}{L(G)},$$

which is in the range $[0, 1]$.

An example of sub-functions is shown in Fig. 1, where $f(x_1, x_2) = \frac{-x_1}{x_1+x_2} \log x_2 - \sin(2x_1)\cos x_2$. The sub-function for $r = v_{12}$ and $\mathcal{L} = \{v_{14}\}$ (obtained via the steps illustrated in Fig. 1 (b) through (d)) is "$\times$ const $\cos$ x_2" in its pre-order representation and "$\text{const} \cos x_2$" in the common form. Its similarity with f can be measured as $\gamma(G, \tilde{G}_{r,\mathcal{L}}) = \frac{4}{18} = \frac{2}{9}$.

Note that a function f can sometimes have several computation-tree representations (e.g., due to commutative/associative properties of certain operations). And functions of different symbolic forms may be equivalent (e.g., $\sin(2x) = 2 \sin x \cos x$). The equivalence of functions can be detected using existing tools (e.g., SymPy [15]). Also note that in practice, since f is initially unknown, when we use alternative knowledge (as discussed earlier) to generate side-information functions, there is no guarantee that they are sub-functions of f. However, as a theoretical approach, it is valid/worthwhile to study how sub-functions can impact the performance of symbolic regression as they have direct correlations with f and are practically common to obtain. Furthermore, it is straightforward to extend the study here to general side-information functions that are not necessarily sub-functions of f, by considering the distance between those general functions and f's sub-functions (e.g., their edit distance in their pre-order representations) as an additional factor that affects the performance of symbolic regression.

III. Symbolic Regression using Side Information

In this section, we present the new symbolic regression (SR) scheme DSR-GP-SI, which extends the existing SR schemes – the DSR scheme based on recurrent neural network (RNN) and risk-seeking reinforcement learning [17], and the DSR-GP scheme combining RNN with genetic programming [16] – by incorporating side information. The overall DSR-GP-SI scheme is illustrated in Fig. 2. We first present its components' details, then present the overall scheme.

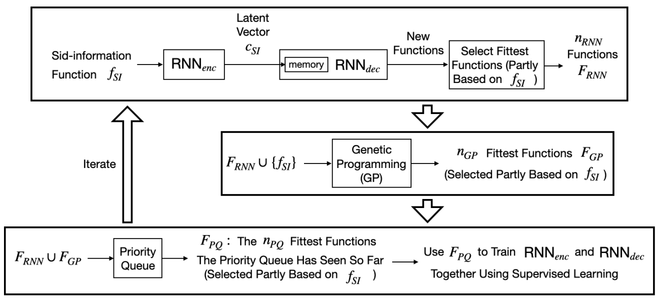
Sid-information Function f_{SI}
RNN$_{enc}$
Latent Vector c_{SI}
memory
RNN$_{dec}$
New Functions
Select Fittest Functions (Partly Based on f_{SI})
n_{RNN} Functions F_{RNN}
Iterate
$F_{RNN} \cup \{f_{SI}\}$
Genetic Programming (GP)
n_{GP} Fittest Functions F_{GP} (Selected Partly Based on f_{SI})
$F_{RNN} \cup F_{GP}$
Priority Queue
F_{PQ}: The n_{PQ} Fittest Functions The Priority Queue Has Seen So Far (Selected Partly Based on f_{SI})
Use F_{PQ} to Train RNN$_{enc}$ and RNN$_{dec}$ Together Using Supervised Learning

Fig. 2. The Symbolic Regression Scheme DSR-GP-SI using side information.

A. The Genetic Programming (GP) Component

Genetic programming (GP) is an important component of the symbolic regression scheme presented here. It has been explored extensively in various previous works [16] [19]. Let $y = f(\mathbf{x})$ be the ground-truth function we look for. Let $S = \{(X_i, y_i) \mid i = 1, 2, \cdots, m\}$ be our $|S| = m$ samples of the function f. GP takes a set of candidate functions for f (called a "population") as input, and helps them evolve into a new set of candidate functions for f (called "a new population") that are hopefully better, i.e., with improved "fitness", whose precise meaning will be presented below. (Note that in the GP component here, as well as in the RNN component to be presented next, the functions all use "const" as the placeholder token for constant coefficients. When we measure the fitness of a generated function, we need to convert its "const" tokens to real numbers. As in [17], we optimize the values of those coefficients – using a non-linear optimization algorithm such as BFGS – such that they maximize the fitness of the function.) Each such iteration that changes one old population into a new population is called a "generation". Given a population $\mathcal{T}_{GP}^0$, we use α generations to turn it into α populations $\mathcal{T}_{GP}^1$, $\mathcal{T}_{GP}^2$, $\cdots$, $\mathcal{T}_{GP}^{\alpha}$, whose fitness is expected to improve progressively. Then n_{GP} "fittest" functions are selected from $\mathcal{T}_{GP}^{\alpha}$ as the output of the GP component.

In our symbolic regression scheme, among the initial population $\mathcal{T}_{GP}^0$, one function f_{SI} is a side-information function obtained via alternative knowledge (as discussed earlier), while the remaining $|\mathcal{T}_{GP}^0| - 1$ functions are generated by a recurrent neural network (RNN) to be introduced later. (The scheme can be easily generalized to having more side-information functions in $\mathcal{T}_{GP}^0$. And we can replicate f_{SI} in $\mathcal{T}_{GP}^0$ if we want to strengthen the survival of its "genes".) To measure the "fitness" of a function $\hat{f}$, let us first define several metrics:

1) NRMSE (normalized root-mean-square error): let $\mu_y = \frac{1}{|S|}\sum_{i=1}^{|S|} y_i$ be the mean of y, and let $\sigma_y = \sqrt{\frac{1}{|S|}\sum_{i=1}^{|S|}(y_i - \mu_y)^2}$ be the standard deviation of y. Then, NRMSE$(\hat{f}) \triangleq \frac{1}{\sigma_y}\sqrt{\frac{1}{|S|}\sum_{i=1}^{|S|}(\hat{f}(X_i) - y_i)^2}$.
2) PORD (pre-order representation distance): let $d_{Levenshtein}(\hat{f}, f_{SI})$ denote the Levenshtein distance between the pre-order representations of $\hat{f}$ and f_{SI} (i.e., the minimum number of insertions, deletions and substitutions needed to turn one pre-order representation into the other). Let $L(\hat{f})$ and $L(f_{SI})$ denote the numbers of tokens in the pre-order representations of $\hat{f}$ and f_{SI}, respectively. Then PORD$(\hat{f}, f_{SI}) \triangleq \frac{d_{Levenshtein}(\hat{f}, f_{SI})}{L(\hat{f}) + L(f_{SI})}$.
3) NDSIF (normalized distance to side-information function f_{SI}): let $\delta > 0$ be a constant parameter. Then NDSIF$(\hat{f}, f_{SI}) \triangleq \frac{1}{|S|}\sum_{i=1}^{|S|} \frac{|\hat{f}(X_i) - f_{SI}(X_i)|}{\max\{|f_{SI}(X_i)|, \delta\}}$.

In our scheme, a function f is considered to have good "fitness" if it not only fits the samples in S well, but also is close to the provided side-information functions f_{SI} (both in terms of sample values and in terms of their symbolic forms). So we define the "fitness" of $\hat{f}$ as $R_{fit}(\hat{f}, f_{SI}) \triangleq$

$$\frac{w_1}{1 + \text{NRMSE}(\hat{f})} + \frac{w_2}{1 + \text{PORD}(\hat{f}, f_{SI})} + \frac{w_3}{1 + \text{NDSIF}(\hat{f}, f_{SI})}$$

where w_1, w_2 and w_3 are constant parameters. The greater $R_{fit}(\hat{f}, f_{SI})$ is, the better $\hat{f}$ is considered to be.

In each of the α "generations", we turn the old population $\mathcal{T}_{GP}^i$ into a new population $\mathcal{T}_{GP}^{i+1}$ (for $i = 0, 1, \cdots, \alpha - 1$) via three types of "evolutionary operations": mutation, crossover and selection. As in [16] [19], a *mutation* operation changes a function f_1 into a new function f_2 by randomly replacing a subtree in the computation tree of f_1 by another randomly generated subtree; a *crossover* operation changes two functions f_1 and f_2 into two new functions f_1' and f_2' by swapping subtrees in their computation trees; and a *selection* operation decides which functions in the current population are kept (instead of removed) for the next population, using methods such as tournament selection. (In tournament selection, in each round, a small group of functions are randomly selected from the current population, and the function with the best fitness – in our case, the maximum value of $R_{fit}(\hat{f}, f_{SI})$ for a function $\hat{f}$ – is selected.) Sufficiently many new functions are generated using mutation and crossover, and selection is used to keep the fittest of them. The new population $\mathcal{T}_{GP}^{i+1}$ is made to have the same number of functions as $\mathcal{T}_{GP}^{i}$ does.

Note that certain functions do not seem to appear in practice, such as nested trigonometric functions (e.g., $\cos(2+\sin(x))$) or adjacent inverse operators (e.g., $\ln e^x$). We follow the constraints proposed in [17] to remove such functions. Furthermore, since symbolic regression looks for short functions, we require all functions to contain at most T tokens in their pre-order representations, for a constant parameter T.

B. The Recurrent Neural Network (RNN) Component

Recurrent neural networks (RNNs) are also an important component in our symbolic regression scheme. We use two RNNs here: an *encoder* RNN$_{enc}$, and a *decoder* RNN$_{dec}$. The use of RNN$_{enc}$ is to transform the side-information function f_{SI} to a latent vector c_{SI}, which is used to initialize the memory of RNN$_{dec}$. The use of RNN$_{dec}$ is to generate new functions useful for symbolic regression.

Specifically, for RNN$_{enc}$ (whose memory is initialized with an all-zero vector), it takes the sequence of tokens t_{SI}^1, t_{SI}^2, $\cdots$, t_{SI}^{β} in a pre-order representation of f_{SI} as its input (where each token is an operator, variable or "const" from the token library and is one-hot encoded), and outputs a sequence of β vectors. The last output vector (which encodes the whole function f_{SI}) becomes our latent vector c_{SI}.

For RNN$_{dec}$, we initialize its memory as c_{SI}. We can use RNN$_{dec}$ to generate many functions (one at a time) as follows (in the same way as in [16] [17]). For simplicity, let us assume each operator in the token library has either one or two operands. (That is by far the most common case in practice, and it can be readily extended to the more general case where an operator may have three or more operands.) To generate a function $\hat{f}$, RNN$_{dec}$ will generate the sequence of tokens $\hat{t}_1$, $\hat{t}_2$, $\cdots$, $\hat{t}_\gamma$ in its pre-order representation (one token at a time) in an auto-regressive way:

1) For $i = 2, 3, \cdots, \gamma$, let t_p^i be the parent node (actually, its token) of $\hat{t}_i$ in the computation tree of $\hat{f}$. If t_p^i is a binary operand and $\hat{t}_i$ is its right child, let t_{sib}^i be the token of the left child of t_p^i; otherwise, let t_{sib}^i take the NULL value. (By default, let t_p^1 and t_{sib}^1 both take NULL values.) Note that both t_p^i and t_{sib}^i are before $\hat{t}_i$ in the pre-order traversal. So by the time RNN$_{dec}$ generates $\hat{t}_i$, the values of t_p^i and t_{sib}^i are already known.
2) For $i = 1, 2, \cdots, \gamma$, RNN$_{dec}$ takes (t_p^i, t_{sib}^i) as input (both one-hot encoded), and outputs a probability distribution $P_i = (p_1^i, p_2^i, \cdots, p_\tau^i)$ (via a softmax activation function), where τ is the number of tokens in the token library and p_j^i is probability for its j-th token. We sample a token from the token library following the distribution P_i, which becomes our next token $\hat{t}_i$ in $\hat{f}$. (The memory of RNN$_{dec}$ is also updated. Note that the memory encodes not only f_{SI} but also all the tokens $\hat{t}_1$, $\hat{t}_2$, $\cdots$, $\hat{t}_{i-1}$ generated so far. The tokens (t_p^i, t_{sib}^i) are used as input because they are topologically near $\hat{t}_i$ in the computation tree [17].)
3) When generating $\hat{t}_i$ as above, if a token in the token library would make the function $\hat{f}$ violate any specified constraint (as we have discussed for "genetic programming" in the previous subsection [17]), we reduce its sampling probability to 0 and re-normalize the probability distribution P_i. This ensures all generated functions are always valid. The pre-order representation of $\hat{f}$ is generated token by token. The generation process ends once the computation tree becomes complete, and we learn the concrete value of γ (i.e., the length of $\hat{f}$) then.

In the above description, we assume RNN$_{enc}$ and RNN$_{dec}$ have been trained. To train them together, we need a set of sample functions that (hopefully) have good fitness. We use supervised learning for training, and use cross-entropy (between the output probability distribution P_i and the one-hot encoding for the corresponding token in the sample function) as the loss function. (To increase the diversity of generated functions, an additional loss term can be added that equals the entropy of P_i times a negative constant. [8] [17]) Compared to the function-generation process presented above, at each step of training, the input tokens are from the *given sample function* instead of the *generated function*. And as we will see next, the two RNNs alternate between training and function generation, in a way similar to reinforcement learning.

C. The Overall Symbolic Regression Scheme

We now present our overall symbolic-regression scheme. It is illustrated in Fig. 2. The scheme searches for good functions in iterations, and it ends either when a function is found that matches the m samples in S sufficiently well, or when it exceeds a given time budget. In each iteration, the scheme works as follows:

1) Let f_{SI} be the given side-information function. We use RNN$_{enc}$ to transform it to a latent vector c_{SI}, and use c_{SI} to initialize the memory of RNN$_{dec}$. We then use RNN$_{dec}$ to generate N_{RNN} functions, and select from them n_{RNN} functions of the highest fitness $R_{fit}(\hat{f}, f_{SI})$. Let F_{RNN} denote those n_{RNN} functions.
2) Let $\mathcal{T}_{GP}^0 = F_{RNN} \cup \{f_{SI}\}$ be the initial population for the genetic programming (GP) component. Use α generations to generate a new population $\mathcal{T}_{GP}^{\alpha}$, and select from it n_{GP} functions of the highest fitness $R_{fit}(\hat{f}, f_{SI})$. Let F_{GP} denote those n_{GP} functions.
3) Feed the functions in $F_{RNN} \cup F_{GP}$ to a persistent maximum reward priority queue (MRPQ) [1], which saves the n_{PQ} functions of the highest fitness it has seen so far. Then, the n_{PQ} functions in the priority queue (denoted by F_{PQ}) are used as sample functions to train the two RNNs RNN$_{enc}$ and RNN$_{dec}$ together.

The above iterative function-generation/RNN-training process is similar to reinforcement learning. The RNNs and GP first generate new functions (which is "exploration"), then the fittest functions among them are used to train the RNNs (which is "learning from past experience"). Over time, the RNNs and GP generate functions of higher and higher fitness.

IV. Performance Evaluation

In this section, we evaluate the performance of the DSR-GP-SI scheme on several benchmarks of functions, including a new benchmark that focuses on data storage. We compare it to state-of-the-art symbolic regression schemes DSR [17] and DSR-GP [16], which the DSR-GP-SI scheme evolved from.

The DSR-GP-SI scheme is configured as follows. The weights in the "fitness" function $R_{fit}(\hat{f}, f_{SI})$ are $w_1 = 0.7$, $w_2 = 0.1$, $w_3 = 0.2$, and $\delta = 0.01$ in its NDSIF component. In each iteration of the scheme, the RNN component generates 500 new functions, all of which are used as the initial population of the GP component. (In the results reported here, this configuration works well. If the RNNs do not generate functions well, n_{RNN} can be decreased.) GP runs $\alpha = 10$ generations in each iteration, and its output has $n_{GP} = 30$ functions. The priority queue stores $n_{PQ} = 10$ fittest functions. The scheme ends either when a sufficiently good function (whose NRMSE is less than 10^{-12}) is found, or when the RNNs have generated 60000 function in total (as a time limit). All functions are required to have at most $T = 30$ tokens. Sub-functions are generated randomly as side information for each benchmark function. The two schemes DSR and DSR-GP are configured similarly.

We compare the performance of the three schemes on two existing symbolic regression benchmarks: Nguyen [25] and Livermore [16]. To better evaluate the potential of symbolic regression for data storage, we also present a new benchmark of 20 functions (which we shall call the *DS benchmark*), shown in Table I. Each function gets $m = 20^n$ random samples for its sample set S (where n is the number of input variables of the function $f(x_1, x_2, \cdots, x_n)$) by letting each input variable x_i sample 20 values uniformly randomly in a small range (such as range $(-1, 1)$, $(0, 1)$ or $(0, 5)$).

The performance of the three schemes for the DS benchmark is shown in Fig. 3. It can be seen that having side information can help improve the recovery rate (the fraction of times of recovering functions correctly) substantially; and generally speaking, the more side information (measured by similarity between the ground-truth function and the sub-function), the better. Furthermore, even when "similarity" is small, there often still exist (short) sub-functions that can improve the recovery rate significantly. That can be seen from those gray circles near the top of the figure (whose recovery rates are close or equal to 1).

Similar performance improvement by side information can be observed for the Nguyen and Livermore benchmarks. (These two benchmarks have comparatively simpler functions, so most functions can be recovered fully by all three schemes. However, for those functions that DSR or DSR-GP fails to recover, DSR-GP-SI obtains a similar performance gain as above.) DSR-GP-SI also achieves substantial reduction in its running time for all three benchmarks. It is because for those functions that DSR or DSR-GP fails to recover, the scheme has to keep searching for functions until the preset time limit is reached. (Due to page limitation, we skip the details here.)

TABLE I
BENCHMARK FUNCTIONS ON DATA STORAGE.

Benchmark	***Expression***
1. (Normal distribution)	$f(x) = \frac{1}{\sqrt{2\pi}\sigma} \exp[-(x-\mu)^2/(2\sigma^2)]$
2. (Folded normal distribution) for $x \geq 0$	$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} + \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x+\mu)^2}{2\sigma^2}}$
3. (Logistic distribution)	$f(x) = \frac{e^{-x}}{(1+e^{-x})^2}$
4. (Root Mean Squared Errors)	$f(x_1, x_2, x_3) = \sqrt{(x_1 - c_1)^2 + (x_2 - c_2)^2 + (x_3 - c_3)^2}$
5. (Entropy) for $0 < x < 1$	$f(x) = -x \log_2 x - (1-x) \log_2(1-x)$
6. (AWGN channel capacity) for x_1, $x_2 > 0$	$f(x_1, x_2) = \frac{1}{2} \log\left(1 + \frac{x_1}{x_2}\right)$
7. (AWGN channel capacity) for x_1, $x_2 > 0$	$f(x_1, x_2) = \frac{1}{2} \log_2(1 + \frac{x_1}{x_2^2})$
8. (AWGN channel capacity) for x_1, x_2, $x_3 > 0$	$f(x_1, x_2, x_3) = x_1 \log\left(1 + \frac{x_2}{x_1 x_3}\right)$
9. (AWGN channel capacity) for x_1, x_2, x_3, $x_4 > 0$	$f(x_1, x_2, x_3, x_4) = x_1 \log_2(1 + \frac{x_2 x_3}{x_1 x_4})$
10. (On LDPC code)	$f(x_1, x_2, x_3) = [1 + \exp(-2x_1 x_2/x_3^2)]^{-1}$
11. (On LDPC code)	$f(x_1, x_2) = \log\left(\frac{1+e^{x_1+x_2}}{e^{x_1}+e^{x_2}}\right)$
12. (Transition response)	$f(x) = \frac{1}{1+(2x/\tau)^2}$
13. (SNR) for x_1, $x_2 > 0$	$f(x_1, x_2) = 10 \log_{10} \frac{x_1^2}{x_2^2}$
14. (On Z-transform)	$f(x) = \frac{x}{x-a}$
15. (On Z-transform)	$f(x) = \frac{Tx^{-1}}{(1-x^{-1})^2}$
16. (On Z-transform)	$f(x) = \frac{(\sin \alpha nT)x^{-1}}{1-2\cos(\alpha T)x^{-1}+x^{-2}}$
17. (On Z-transform)	$f(x) = \frac{(\cos \alpha nT)x^{-1}}{1-2\cos(\alpha T)x^{-1}+x^{-2}}$
18. (differential entropy) for $x_1, x_2 > 0$	$f(x_1, x_2) = \frac{1}{2} \log((2\pi e)^{x_1} x_2)$
19. (Rate distortion) for x_1, $x_2 > 0$	$f(x_1, x_2) = \frac{1}{2} \log \frac{x_1^2}{x_2}$
20. (entropy maximization) for x_1, $x_2 > 0$	$f(x_1, x_2) = \frac{1}{x_2} e^{-\frac{x_1}{x_2}}$

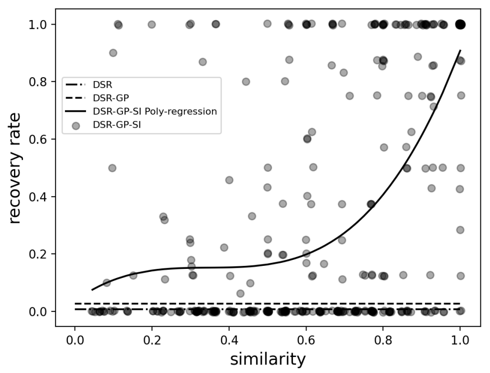

Fig. 3. Performance of DSR-GP-SI (solid curve and gray circles), DSR-GP (dashed line) and DSR (dash-dotted line) for the DS benchmark. Here the x-axis is the "similarity" $\gamma(G, \tilde{G}_{r,\mathcal{L}})$ between the ground-truth function and the sub-function side information, and the y-axis is the "recovery rate", i.e., the fraction of times a ground-truth function is correctly recovered. (Note that "similarity" is relevant to DSR-GP-SI, but not to DSR or DSR-GP.) Each gray circle corresponds to a particular benchmark function and a particular sub-function, where numerous experiments for DSR-GP-SI were performed and their recovery rate was shown. (Note that different sets of experiments may produce gray circles that overlap each other. The more overlapping, the darker the gray color becomes.) The solid curve shows the trend of the gray circles: for the average recovery rates corresponding to different "similarity" values (averaged over all functions in the DS benchmark and their experimented sub-functions for each given "similarity"), the solid curve is their polynomial regression (a generalization of linear regression) with polynomial degree 3. The average recovery rates of DSR and DSR-GP (which use no side information) are 0.007 and 0.027, respectively, which are shown as two horizontal lines for easy comparison.

References

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

[2] S. Ahn, J. Kim, H. Lee, and J. Shin, "Guiding deep molecular optimization with genetic exploration," in *Proc. Conference on Neural Information Processing Systems (NeurIPS)*, 2020.

[3] B. Al-Helali, Q. Chen, B. Xue and M. Zhang, "A genetic programming-based wrapper imputation method for symbolic regression with incomplete data," in *Proc. IEEE Symposium Series on Computational Intelligence (SSCI)*, pp. 2395–2402, 2019.

[4] B. Al-Helali, Q. Chen, B. Xue and M. Zhang, "Genetic programming with noise sensitivity for imputation predictor selection in symbolic regression with incomplete data," in *Proc. IEEE Congress on Evolutionary Computation (CEC)*, pp. 1–8, 2020.

[5] D. Bar-Lev, I. Orr, O. Sabary, T. Etzion, and E. Yaakobi, "Deep DNA storage: scalable and robust DNA storage via coding theory and deep learning," arXiv: 2109.00031, Nov. 2021.

[6] Q. Chen, M. Zhang and B. Xue, "Feature selection to improve generalization of genetic programming for high-dimensional symbolic regression," in *IEEE Transactions on Evolutionary Computation*, vol. 21, no. 5, pp. 792–806, Oct. 2017.

[7] C. Ferreira, "Function finding and the creation of numerical constants in gene expression programming," in *Advances in Soft Computing (AICS)*, 2003.

[8] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, "Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor," arXiv preprint arXiv:1801.01290, 2018.

[9] K. Huang, P. H. Siegel and A. Jiang, "Functional error correction for robust neural networks," in *IEEE Journal on Selected Areas in Information Theory (JSAIT)*, vol. 1, no. 1, pp. 267–276, May 2020.

[10] S. Kim, P. Y. Lu, S. Mukherjee, M. Gilbert, L. Jing, V. Čeperić, and M. Soljačić, "Integration of neural network-based symbolic regression in deep learning for scientific discovery," in *IEEE Transactions on Neural Networks and Learning Systems*, vol. 32, no. 9, pp. 4166–4177, Sept. 2021.

[11] G. Lample and F. Charton, "Deep learning for symbolic mathematics," in International Conference on Learning Representations, 2020.

[12] M. Landajuela, B. K. Petersen, S. Kim, C. P. Santiago, R. Glatt, N. Mundhenk, J. F. Pettit, and D. Faissol, "Discovering symbolic policies with deep reinforcement learning," in *Proc. International Conference on Machine Learning*, pp. 5979–5989, PMLR, 2021b.

[13] Y. Liu, S. Wu, and P. H. Siegel, "Bad page detector for NAND flash memory," *11th Annual Non-Volatile Memories Workshop (NVMW)*, La Jolla, CA, March 8–10, 2020. [Online] Available: $http://nvmw.ucsd.edu/nvmw2020-program/unzip/current3/nvmw2020-final35.pdf$.

[14] Q. Lu, J. Ren, and Z. Wang, "Using genetic programming with prior formula knowledge to solve symbolic regression problem," in *Computational intelligence and neuroscience*, vol. 2016, article ID 1021378, http://dx.doi.org/10.1155/2016/1021378.

[15] A. Meurer *at al.*, Sympy: symbolic computing in python, *PeerJ Computer Science*, 3:e103, Jan. 2017.

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

[17] B. K. Petersen, M. Landajuela, T. N. Mundhenk, C. P. Santiago, S. K. Kim, and J. T. Kim, "Deep symbolic regression: Recovering mathematical expressions from data via risk-seeking policy gradients," in *Proceeding of the International Conference on Learning Representations (ICLR)*, 2021.

[18] S. S. Sahoo, C. H. Lampert and G. Martius, "Learning equations for extrapolation and control," arXiv: 1806.07259, June 2018.

[19] M. Schmidt and H. Lipson, "Distilling free-form natural laws from experimental data," in *Science*, vol. 324, pp. 81–85, April 2009.

[20] M. D. Schmidt and H. Lipson, "Incorporating expert knowledge in evolutionary search: a study of seeding methods," in *Proc. 11th Annual Conference on Genetic and Evolutionary Computation (GECCO'09)*, pp. 1091–1098, Montreal, Canada, July 2009.

[21] M. D. Schmidt and H. Lipson, "Coevolution of fitness predictors," in *IEEE Transactions on Evolutionary Computation*, vol. 12, no. 6, pp. 736–749, December 2008.

[22] N. Sree Prem, "An application of machine learning to bad page prediction in multilevel flash," M.S. thesis, Electrical and Computer Engineering, University of California San Diego, La Jolla, CA, 2019. [Online] Available: $https://escholarship.org/uc/item/8ng6723d$.

[23] S. Udrescu and M. Tegmark, "AI Feynman: a physics-inspired method for symbolic regression," in *Science Advances*, vol. 6, no. 16, eaay:2631, April 2020.

[24] S. Udrescu, A. Tan, J. Feng, O. Neto, T. Wu, and M. Tegmark, "AI Feynman 2.0: pareto-optimal symbolic regression exploiting graph modularity," in *Proc. 34th Conference on Neural Information Processing Systems (NeurIPS)*, Vancouver, Canada, 2020.

[25] N. Q. Uy, N. X. Hoai, M. O'Neill, R. I. McKay, and E. Galvan-Lopez, "Semantically-based crossover in genetic programming: application to real-valued symbolic regression," in *Genetic Programming and Evolvable Machines*, vol. 12, 91–119, 2011.

[26] H. Zhang and A. Zhou, "RL-GEP: symbolic regression via gene expression programming and reinforcement learning," in *Proc. International Joint Conference on Neural Networks (IJCNN)*, pp. 1–8, July 2021.

[27] S. Zheng, C. Ho and P. H. Siegel, "Modeling flash memory channels using conditional generative nets," arXiv: 2111.10039, Nov. 2021.